Draft Proceedings of the

32nd International Symposium on Implementation and Application of Functional Languages (IFL 2020)

University of Kent, UK
2nd – 4th September 2020

In cooperation with:
ACM SIGPLAN

Table of Contents

End-user feedback in multi-user workflow systems

Nico Naus
nico.naus@ou.nl
Open University of The Netherlands

Johan Jeuring
Open University of The Netherlands
Utrecht University
j.t.jeuring@uu.nl

ABSTRACT

Workflow systems are more and more common due to the automation of business processes. The automation of business processes enables organizations to simplify their processes, improve services and contain costs. A problem with using workflow systems is that processes once known by heart, are now hidden from the user. This, combined with time pressure, lack of experience and an abundance of options, makes it harder for a user to make the right choices. To aid users of these systems, we have developed a multi-user rule-based problem-solving framework that can be instantiated for many workflow systems. It provides hints to the end user on how to achieve her goals and makes life for the programmer easier, as she only needs to instantiate the framework instead of programming an ad-hoc solution. Our approach consists of two parts. First, we present a domain-specific language (DSL) that offers commonly used constructs for combining components of different rule-based problems. Second, we use generic search algorithms to solve various kinds of problems. We show a practical implementation with an example workflow system. We show that this system fulfills several desirable properties.

CCS CONCEPTS

• **Theory of computation → Formal languages and automata theory**.

KEYWORDS

Functional programming, iTasks, Domain specific languages, Workflows

ACM Reference Format:
Nico Naus and Johan Jeuring. 2020. End-user feedback in multi-user workflow systems. In *The 32nd symposium on Implementation and Application of Functional Languages.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Due to the automation of business processes, more and more workflow systems are being used to manage and perform tasks. The Dutch coastal guard uses a workflow system to monitor the seas and to aid in emergencies [17]. Hospitals use systems like EPIC or WebPT to manage patients, assign tasks and monitor treatment. Teachers use intelligent tutoring systems to give students immediate and personalised feedback on their exercises.

A downside of using workflow systems is that a process that was once known by heart, is now hidden from the user. This, combined with the fact that there might be time pressure, lack of experience or an abundance of options, makes it harder for end users to make the right choices between different options and to achieve the goal a user has in mind.

To overcome this problem, we want to assist a user in reaching her goals more efficiently. This is commonly done by employing a decision support system (DSS) [29]. Many different types of DSS exist, but they have several components in common. A DSS has some model that represents the domain in which a decision needs to me made. Using data about the current situation, together with the model, some kind of analysis is performed. The specific analysis used differs per DSS. Based on the results of the analysis, the DSS suggests a decision to the user.

Using a DSS has many advantages [24]. The productivity of individual users is improved. Users spend less time on the administrative aspects of the tasks they need to perform, and spend less time manipulating data. The quality and speed of the decisions is increased. Time spent on retrieving decision-relative information is reduced, and fact-based decision making is stimulated.

Traditional DSS have several downsides. First of all, since a DSS relies on a model of the problem, these systems are very rigid. If the problem is not modelled, the DSS cannot be used. When the problem or the domain are altered or expanded, a programmer needs to go back and change the model accordingly. A second downside is the large financial investment that is required [24].

To overcome the downsides of using a DSS, while still being able to enjoy its benefits, we present a multi-user rule-based problem solver. Our system consists of two parts; a domain-specific language (DSL) that allows programmers to express multi-user rule-based problems, and several generic solving algorithms that calculate traces to the goal, from which hints can be produced.

The advantage of our system is that it is much easier to model a multi-user rule-based problem. On top of that, once the model has been described, there is no need to develop a custom analysis. Once the model has been expressed in our DLS, one of the generic solving algorithms can be used to find a solution.

In previous work [19], we have presented a single-user rule-based problem-solving framework with a practical implementation. This paper presents both a formal multi-user rule-based problem-solving framework, as well as a practical implementation. The trace semantics of the framework is shown to be sound and complete with respect to the regular semantics of our DSL, using a property verification tool.

2 PROBLEM DESCRIPTION

Our goal is to describe a generic framework for autonomously generating hints that end-users of workflow systems can use to achieve their goal(s). By a workflow system we mean a system that automates workflows, allows multiple users to collaborate, and works on some kind of shared data.

Van der Aalst et al. [2] have identified common patterns of workflow systems. We will use this set to specify constructs used in workflow systems.

2.1 Constructs

The following constructs are common in most workflow systems.

Sequence	Perform activities one after the other.
Parallel Split	Perform multiple activities at the same time.
Exclusive Choice	Choose exactly one activity from a list.
Milestone	Make an activity available when the state is in a certain condition.
Interleaving	Perform activities in an arbitrary order.
Multi-Choice	Choose one or more activities from a list.
Arbitrary Cycles	Repeat part of a workflow an arbitrary number of times.

In traditional workflow systems, steps can pass data along to the next step, as well as work on shared data. We simplify our model of workflow systems to only consider shared data. Therefore we do not need constructs like explicit synchronisation points and discriminators, as described by van der Aalst et al. Steps can immediately observe the result of every other step through the shared data, instead of having to wait on incoming branches.

In addition to the constructs above, we want to support multiple users. The most straightforward way to accommodate this is by means of an Assign construct, which assigns an activity to a user or possibly a set of users. Such a construct is heavily used in for example the iTasks workflow framework [23].

2.2 Hints

The purpose of our framework is to give hints to end-users of workflow systems; information that they can use to achieve their goal. What is the best treatment to select for a certain patient? What action needs to be taken when a fire breaks out on a ship?

We use traces consisting of sequences of steps that lead users to states in which the goal has been reached. From these traces, richer feedback information can be constructed. For example, next-step hints can be constructed by returning the first element in the trace.

The traces that we want to generate are composed of sets of steps per time unit, where multiple users can perform a step in each time unit. Section 2.2 lists an example of such a trace, where $state_i$ is the state at time i. Application of all steps in one time unit leads to the next state in the trace.

For this to work properly, we require that all steps performed in one time unit are independent of each other. This means that the order of applying steps to the original state does not affect the resulting state. On top of that, we require that every user performs at most one step per time unit.

$$state_0 \xrightarrow{\begin{pmatrix} user_1 & : & step_1 \\ user_2 & : & step_2 \end{pmatrix}} state_1 \xrightarrow{\begin{pmatrix} user_1 & : & step_3 \\ & & \\ user_3 & : & step_4 \end{pmatrix}} state_2$$

Figure 1: A visualization of a trace

2.3 Research question

In the following sections, we will answer the question: how do we, for any given multi-user workflow problem, calculate traces that lead to a solution state?

We aim to answer this question by first tackling the issue of dealing with different multi-user workflow systems. By defining a domain-specific language that allows for the uniform description of problems, we can treat each of them in a similar manner. Then, to calculate the partial traces, we employ search algorithms from artificial intelligence.

3 PROBLEM FORMALISATION

A multi user workflow problem can be considered a well-defined artificial intelligence (AI) problem [26], which consists of the following components.

Initial state	The state of the problem that you want to solve.
Operator set	The set of steps that can be taken, together with their effects.
Goal test	A predicate that is True if the problem is solved.
Path Cost function	A function that describes the cost of each operation.

This is similar to workflows: the state of the workflow system is the initial state, the steps users can take are the operator set, the goal the user has in mind is the goal test and finally the resources a workflow uses can be captured in a path cost function.

By choosing a uniform way to describe workflow problems as well-defined AI problems, we can treat each problem within the same framework.

Currently, several languages exist to allow programmers to describe workflow and rule-based problems, but none of them are completely suitable for our purposes. Workflow languages allow programmers to model complex behaviour that makes calculating a path to the goal of a user very complex or even unfeasible. They are therefore not suitable for our purposes. Existing rule-based problem modeling languages like PDDL [18], STRIPS [8], SITPLAN [9] and PLANNER [12] have limitations that prevent us from fully describing the problems from the workflow domain. These languages do not support higher order definitions, and most of them only support a finite state-space. Higher order definitions make it much easier to reuse code, and reduce the amount of modeling needed to express a problem. To overcome these disadvantages, we design our own rule-based problem modeling language.

We opt for a domain-specific language (DSL) that is embedded in a language that supports higher order programming. This means that our DSL is expressed in a standard programming language, called the host language. Embedding a DSL into a host language

$$
\begin{array}{lcl}
\text{RuleTree } a & = & \text{Seq [RuleTree } a] \\
 & | & \text{Choice [RuleTree } a] \\
 & | & \text{Par [RuleTree } a] \\
 & | & \text{Assign } u \text{ (RuleTree } a) \\
 & | & \text{Leaf (CR } a) \\
 & | & \text{Empty} \\
 & & \\
\text{CR } a & = & \text{Cond (Pred } a) \text{ (CR } a) \\
 & | & \text{Rule } n \text{ (Effect } a) \\
 & | & u \ @ \text{ (Rule } n \text{ (Effect } a)) \\
 & & \\
\text{Pred } a & = & a \to \text{Bool} \\
\text{Effect } a & = & a \to a \\
\text{Goal } a & = & a \to \text{Bool} \\
n & \in & \text{set of names} \\
u & \in & \text{set of user identifiers}
\end{array}
$$

Figure 2: Syntax of our rule-based problem DSL

has the specific advantage that we can use all features from the host language in specifying programs in our DSL. In this case, we are particularly interested in using abstraction, application and recursion from the host language.

Figure 2 lists the components of our DSL.

With the DSL, we want to cover all workflow constructs mentioned in section 2.1, as well as the elements of a well-defined AI problem as listed above.

We use a slightly simplified definition of an AI problem however. If there is a cost associated with a certain operation, we encode this as an effect on the state. This is the common way to encode these effects in workflow systems. Therefore, we do not need a path cost function.

The (initial) state is modelled by a value of type a.

The operator set is represented by a tree structure we call a RuleTree. This tree structure describes how operations, which we call rules, relate to each other. There are four ways to combine RuleTrees; in sequence (Seq), by choosing among them (Choice), in parallel (Par), or by assigning them to a user (Assign). These correspond to the workflow constructs sequence, exclusive choice, parallel split, and user assignment. The Milestone construct is modelled by means of a condition. Interleaving and multi-choice can be built from these constructs. For arbitrary cycles we rely on the host language to provide abstraction and application.

The design of the RuleTree DSL is loosely based on strategy language from the Ideas framework [11], iTask combinators [23], and the strategy language presented by Visser and others [34].

Finally, the goal test is represented by the predicate Goal a. These three components make up our DSL for describing rule-based problems.

The leaves of the RuleTree are CR a and Empty. Here, CR a is either a Cond or an actual rule, where the rule can be assigned to a user u (u@(Rule n (Effect a))) or unassigned (Rule n (Effect a)), with n the name of the rule and Effect a the effect of the rule on the state. Conds can be nested. Rules can be seen as steps, tasks or the smallest units in which work can be divided.

Conditions are part of the leaves, and guard a CR a, which may contain another condition. A single leaf is considered to be an atomic action. This prevents conflicts between rules and conditions when leaves are executed in parallel.

We implement the DSL as an embedded DSL in Haskell. This allows us to use standard Haskell functions to construct for example a RuleTree. We chose not to implement recursion in our DSL, but instead make use of recursion in the host language. The advantage of this is that we can keep our DSL simple and small. Implementing recursion in the DSL requires adding abstraction and application, making the DSL significantly more complex. Most rule-based problem can be encoded in this DSL, and as long as there is an appropriate solving algorithm available, our framework can generate hints for it.

3.1 Semantics

Figure 3 defines what it means to apply an entire RuleTree to a state. The result of the application is a set of end states that can be reached.

Application of a RuleTree is rather straightforward, except for the Seq and Par cases. If an error occurs inside a Seq, denoted by $\lightning$, the whole sequence needs to be aborted since the next step does not become available. This can occur when a condition does not hold, or when a choice has to be made out of zero elements.

We are only interested in the final states that can be reached, not in the intermediate states. As a consequence, we can view the semantics of Par as interleaving of the individual steps contained in the sub-trees. The function step takes a RuleTree and calculates a set of tuples containing all steps that can be applied at this point and the remaining RuleTree. This result is then used by the RuleTree application to interleave all possible steps, and calculate the final state.

4 TRACE SEMANTICS

We are not so much interested in the final state that is reached, but rather in the steps that users can take to transition between states. To calculate these steps, we use a trace semantics. The trace semantics consists of two parts, namely the firsts and empty observations over RuleTrees, and the function traces that makes use of these observations.

We introduce two new constructs that will be used to define the two parts.

$$
\begin{array}{lcl}
\text{RuleSet } a & = & \mathcal{P}(\text{CR } a)) \\
\text{Trace } a & = & \text{Step } a \text{ (RuleSet } a) \text{ (Trace } a) \\
 & | & \text{State } a
\end{array}
$$

4.1 RuleTree **observations**

The basis of the trace semantics of our multi-user rule-based problem consists of the functions $\mathcal{F}$ and $\mathcal{E}$, listed in Figure 4 and Figure 5.

The function $\mathcal{F}$ (firsts) produces a set of elements of the form $(\bar{R}, rt)$, where $\bar{R}$ is a set of CR a-elements. $\bar{R}$ contains all rules that are executed at the same time. It contains at most one rule per user and all rules in this set are independent.

Function $\mathcal{E}$ (empty) checks if a RuleTree is empty. A RuleTree is considered empty if at least one of the applications of the tree

$$
\begin{array}{ll}
\cdot : \text{RuleTree } a \times a \longrightarrow \mathcal{P}(a) & \\
(\text{Seq } (rt{:}rts)) \cdot s & \\
\quad \mid rt \cdot s = \lightning \mapsto \lightning & \\
\quad \mid rt \cdot s \neq \lightning \mapsto \{x \mid s' \in rt \cdot s, x \in (\text{Seq } rts) \cdot s'\} & \\
(\text{Seq } [\,]) \cdot s & = \{s\} \\
(\text{Choice } (rt{:}rts)) \cdot s & = rt \cdot s \cup (\text{Choice } rts) \cdot s \\
(\text{Choice } [\,]) \cdot s & = \lightning \\
(\text{Par } (rt{:}rts)) \cdot s & \\
\quad = \{(\text{Par } (rt'{:}rts)) \cdot (r \cdot s) \mid (r, rt') \in \text{step } rt\} & \\
\quad \cup \{(\text{Par } (rt{:}rts')) \cdot (r \cdot s) \mid (r, rts') \in \text{step } (\text{Par } rts)\} & \\
(\text{Par } [\,]) \cdot s & = \{s\} \\
(\text{Assign } u\ rt) \cdot s & = rt \cdot s \\
(\text{Leaf } (\text{Cond } p\ r)) \cdot s & \mid \neg p\ s \mapsto \lightning \\
 & \mid\ p\ s \mapsto (\text{Leaf } r) \cdot s \\
(\text{Leaf } (\text{Rule } n\ e)) \cdot s & = \{e\ s\} \\
\text{Empty} \cdot s & = \{s\}
\end{array}
$$

$$
\begin{array}{ll}
\text{step} : \text{RuleTree } a \longrightarrow \mathcal{P}(\text{RuleTree } a \times \text{RuleTree } a) & \\
\text{step } (\text{Seq } (rt{:}rts)) & \\
\quad = (\bigcup\{\text{step } (\text{Seq } rts) \mid (\text{Empty}, \text{Empty}) \in \text{step } rt\}) & \\
\quad \cup \{(r, \text{Seq } (rt'{:}rts)) \mid (r, rt') \in \text{step } rt\} & \\
\text{step } (\text{Seq } [\,]) & = \{(\text{Empty}, \text{Empty})\} \\
\text{step } (\text{Choice } (rt{:}rts)) & = \text{step } rt \cup \text{step } (\text{Choice } rts) \\
\text{step } (\text{Choice } [\,]) & = \emptyset \\
\text{step } (\text{Par } (rt{:}rts)) & \\
\quad = (\bigcup\{\text{step } (\text{Par } rts) \mid (\text{Empty}, \text{Empty}) \in \text{step } rt\}) & \\
\quad \cup \{(r, \text{Par } (rt'{:}rts)) \mid (r, rt') \in \text{step } rt\} & \\
\quad \cup \{(r', \text{Par } (rt{:}rts')) \mid (r', \text{Par } rts') \in \text{step } (\text{Par } rts)\} & \\
\text{step } (\text{Par } [\,]) & = \{(\text{Empty}, \text{Empty})\} \\
\text{step } (\text{Assign}\, u\ rt) & = \text{step } rt \\
\text{step Leaf } c & = \{(\text{Leaf } c, \text{Empty})\} \\
\text{step Empty} & = \{(\text{Empty}, \text{Empty})\}
\end{array}
$$

Figure 3: Semantics of RuleTree **application**

$$
\begin{array}{ll}
\mathcal{F} : \text{RuleTree } a \times a \longrightarrow \mathcal{P}(\text{RuleSet } a \times \text{RuleTree } a) & \\
\mathcal{F}\ (\text{Seq } (rt{:}rts), s) & \\
\quad = \begin{cases} \{(\bar{R}, \text{Seq } (rt'{:}rts)) \mid (\bar{R}, rt') \in \mathcal{F}\ (rt, s)\} & \\ \cup \{x \mid \mathcal{E}(rt), x \in \mathcal{F}\ (\text{Seq } rts, s)\} & \mathcal{F}\ (rt, s) \not\equiv \lightning \\ \lightning & \mathcal{F}\ (rt, s) \equiv \lightning \end{cases} & \\
\mathcal{F}\ (\text{Seq } [\,], s) & = \emptyset \\
\mathcal{F}\ (\text{Choice } (rt{:}rts), s) & = \mathcal{F}\ (rt, s) \cup \mathcal{F}\ (\text{Choice } rts, s) \\
\mathcal{F}\ (\text{Choice } [\,], s) & = \lightning \\
\mathcal{F}\ (\text{Par } [rt_1, \cdots, rt_n], s) & \\
\quad = \{(\bar{R}, \text{Par } [rt'_1, \cdots, rt'_n]) & \\
\quad \mid\ (\bar{R}_i, rt'_i) \in (\mathcal{F}(rt_i, s) \cup \{(\emptyset, rt_i)\}) & \\
\quad ,\ \bar{R} = \bar{R}_1 \cup \cdots \cup \bar{R}_n & \\
\quad ,\ \bar{R} \neq \emptyset & \\
\quad ,\ \forall u_x@r_i, u_y@r_j \in \bar{R} : r_i(r_j \cdot s) = r_j(r_i \cdot s) & \\
\quad ,\ \forall u_x@r_p, u_y@r_q \in \bar{R} : r_p \neq r_q \Rightarrow u_x \neq u_y\} & \\
\mathcal{F}\ (\text{Par } [\,], s) & = \emptyset \\
\mathcal{F}\ (\text{Assign } u\ rt, s) & = \mathcal{F}\ (\text{applyAssign}(rt, u), s) \\
\mathcal{F}\ (\text{Leaf } c, s) & = \{(\{c\}, \text{Empty})\} \\
\mathcal{F}\ (\text{Empty}, s) & = \emptyset
\end{array}
$$

Figure 4: Semantics of the firsts observation $\mathcal{F}$

$$
\begin{array}{ll}
\mathcal{E} : \text{RuleTree } a \longrightarrow \text{Bool} & \\
\mathcal{E}\ (\text{Seq } (rt{:}rts)) & = \mathcal{E}\ (rt) \wedge \mathcal{E}\ (\text{Seq } rts) \\
\mathcal{E}\ (\text{Seq } [\,]) & = \text{True} \\
\mathcal{E}\ (\text{Choice } (rt{:}rts)) & = \mathcal{E}\ (rt) \vee \mathcal{E}\ (\text{Choice } rts) \\
\mathcal{E}\ (\text{Choice } [\,]) & = \text{False} \\
\mathcal{E}\ (\text{Par } (rt{:}rts)) & = \mathcal{E}\ (rt) \wedge \mathcal{E}\ (\text{Par } rts) \\
\mathcal{E}\ (\text{Par } [\,]) & = \text{True} \\
\mathcal{E}\ (\text{Assign } u\ rt) & = \mathcal{E}\ (rt) \\
\mathcal{E}\ (\text{Empty}) & = \text{True} \\
\mathcal{E}\ (\text{Leaf } c) & = \text{False}
\end{array}
$$

Figure 5: Semantics of the empty observation $\mathcal{E}$

does not execute any rules. For example, the empty list sequence (Seq [ ]) is empty, since it holds no rules. A tree can be empty even when $\mathcal{F}$ returns a ruleset. This is the case for the RuleTree Choice [Seq [ ], Rule n e], for example, since when one chooses the first element, no rule is applied. But $\mathcal{F}$ returns a set containing Rule n e.

Building the set of first rulesets is not trivial in a multi-user setting. This especially shows in the case of Par. This is due to the fact that parallel RuleTrees allow multiple users to execute rules at the same time.

To calculate $\mathcal{F}$ (Par rts, s), we calculate $\mathcal{F}$ for every RuleTree that is executed in parallel. Since we do not have to execute a rule from every parallel RuleTree at each step, we add the empty ruleset with the original RuleTree $((\emptyset, rt_i))$ to the set of $\mathcal{F}$. For each RuleTree rt_i in rts, we now pick one element of this $\mathcal{F}$ set that also contains the empty ruleset. Then, we put all the selected rules for each rt_i together to build the total ruleset $\bar{R}$. The remaining RuleTree is built by concatenating all rt'_i elements. These could just be the original RuleTree rt_i, if the selected element was the empty set.

$$\cdot : \text{RuleSet } a \times a \longrightarrow a$$

$$\bar{R} \cdot s = \begin{cases} \bar{R} \setminus \{\text{Leaf } (u \ @ \ (\text{Rule } \ n \ e))\} \cdot (e \ s) & \text{Leaf } (u \ @ \ (\text{Rule } \ n \ e)) \in \bar{R} \\ \bar{R} \setminus \{\text{Leaf } (\text{Rule } \ n \ e)\} \cdot (e \ s) & \text{Leaf } (\text{Rule } n \ e) \in \bar{R} \\ \bar{R} \setminus \{\text{Leaf } (\text{Cond } \ p \ c)\} \cdot (c \cdot s)) & \text{Leaf } (\text{Cond } \ p \ c) \in \bar{R}, p \ s \\ \lightning & \text{Leaf } (\text{Cond } \ p \ c) \in \bar{R}, \neg(p \ s) \end{cases} \qquad \begin{aligned} \lightning \cup \lightning &= \lightning \\ A \cup \lightning &= A \\ \lightning \cup A &= A \\ A \cup B &= \{x \mid x \in A \vee x \in B\} \end{aligned}$$

$$\text{applyAssign} : \text{RuleTree } \ a \times \text{User} \longrightarrow \text{RuleTree } \ a$$

$$\begin{aligned} \text{applyAssign}(\text{Seq } [rt_1, \cdots, rt_n], u) &= \text{Seq } [\text{Assign } \ u \ rt_1, \cdots, \text{Assign } \ u \ rt_n] \\ \text{applyAssign}(\text{Choice } [rt_1, \cdots, rt_n], u) &= \text{Choice } [\text{Assign } \ u \ rt_1, \cdots, \text{Assign } \ u \ rt_n] \\ \text{applyAssign}(\text{Par } [rt_1, \cdots, rt_n], u) &= \text{Par } [\text{Assign } \ u \ rt_1, \cdots, \text{Assign } \ u \ rt_n] \\ \text{applyAssign}(\text{Leaf } (\text{Cond } \ p \ c), u) &= \text{Leaf } (\text{Cond } p \ (\text{Assign } \ u \ r)) \end{aligned} \qquad \begin{aligned} \text{applyAssign}(\text{Assign } \ u_2 \ rt, u_1) &= \text{Assign } u_2 \ rt \\ \text{applyAssign}(\text{Leaf } \ r, u) &= \text{Leaf } (u@r) \\ \text{applyAssign}(\text{Empty}, u) &= \text{Empty} \end{aligned}$$

Figure 6: auxiliary definitions

$$\text{traces} : \text{RuleTree } \ a \times a \longrightarrow \mathcal{P}(\text{Trace } a)$$

$$\text{traces } (rt, s) = \begin{cases} \{\text{State } \ s \mid \mathcal{E} \ (rt)\} & \\ \cup \{s \xrightarrow{\bar{R}} x \mid (\bar{R}, rt') \in \mathcal{F} \ (rt, s), \ x \in \text{traces } (rt', \bar{R} \cdot s)\} & \mathcal{F} \ (rt, s) \neq \lightning \\ \emptyset & \mathcal{F} \ (rt, s) = \lightning \end{cases}$$

Figure 7: Definition of the traces **function**

Three conditions must hold for any $\bar{R}$. First, we require $\bar{R}$ to be non-empty. Second, we require every pair of elements in $\bar{R}$ to be independent, meaning that the order of application to s does not influence the resulting state. And third, we verify that there is at most one rule assigned to every user.

Function $\mathcal{F}$ relies on several auxiliary functions listed in Figure 6.

4.2 Traces of RuleTrees

Now that we have defined the firsts and empty observation, the traces function can be constructed. Figure 7 lists the definition of this function.

The function traces takes a RuleTree and state, and returns the set of all possible traces. $\mathcal{F}$ is called on the RuleTree. This returns a ruleset, paired with the remaining RuleTree. These rulesets represent every possible action that can be taken. For each ruleset, a new state is calculated by applying the set to the current state. Then traces is calculated recursively to calculate the rest of the trace. When a RuleTree is empty ($\mathcal{E}(rt)$), the trace is completed, and the current state is returned.

This completely describes our trace semantics.

5 SOLVING ALGORITHMS

For the purpose of constructing hints, traces are of limited interest. A RuleTree includes all steps that can be taken, and therefore possibly also incorrect steps. Instead, we would like to obtain traces that end in a state that satisfies the goal the user is trying to reach.

To achieve this, we develop several solving algorithms. All algorithms return traces that may not completely apply the RuleTree, as opposed to the traces function, which only returns traces that have fully applied the RuleTree.

5.1 Breadth First Trace

The first algorithm we introduce is a breadth first trace algorithm, BFTrace. It performs a breadth first search, to find a state that satisfies the goal condition g. Figure 8 lists its definition.

Going over the definition from top to bottom, one of three cases applies.

- If the goal is satisfied, the set containing only the current state is returned.
- If there exists one or more expansions that satisfy the goal, the traces that belong to those expansions are returned.
- If none of the expansions satisfies the goal test, BFTrace is called recursively.

5.2 Heuristic Trace

A possible disadvantage of the breadth first trace is that it expands all traces, and can be very slow or even infeasible, depending on the complexity of the problem. An often used solution is to perform a best first search. This method uses a heuristic function to score each expansion, and then selects the best state to further expand. If in the set of current expanded traces e there is an expansion that fulfills the goal condition, it is returned, else we recurse on the expansions that have the lowest heuristic score. The definition of our heuristic trace function is given in Figure 9.

hTrace takes as argument a tuple containing the goal test g, a heuristic scoring function h and the set of current expansions e. We

$$\begin{array}{lll}
\text{BFTrace} : \text{Goal } a \times \text{RuleTree } a \times a \longrightarrow \mathcal{P}(\text{Trace } a) & & \\
\text{BFTrace } (g, rt, s) & \mid g\ s & \mapsto \{\text{State } s\} \\
 & \mid \neg g\ s, \exists(\bar{R}, rt', s') \in \text{expand } (rt, s) : g\ s' & \mapsto \{s \xrightarrow{\bar{R}} \text{State } s' \mid (\bar{R}, rt', s') \in \text{expand } (rt, s), g\ s'\} \\
 & \mid \neg g\ s, \forall(\bar{R}, rt', s') \in \text{expand } (rt, s) : \neg(g\ s') & \mapsto \{s \xrightarrow{\bar{R}} x \mid (\bar{R}, s', rt') \in \text{expand } (rt, s), x \in \text{BFTrace } (g, rt', s')\}
\end{array}$$

Figure 8: BFTrace **search algorithm definition**

$$\begin{array}{l}
\text{hTrace} : (\text{Goal } a) \times (a \longrightarrow \text{Integer}) \times \mathcal{P}(\text{RuleTree } a \times a \times \text{Trace } a) \longrightarrow \mathcal{P}(\text{Trace } a) \\
\text{hTrace } (g, h, e) \mid \exists(rt, s, x) \in e : g\ s \mapsto \{x \mid (rt, s, x) \in e, g\ s\} \\
\qquad\qquad\qquad \mid \forall(rt, s, x) \in e : \neg g\ s \mapsto \text{hTrace } (g, h, \text{lowExp} \cup \text{high}) \\
\quad \textbf{where} \\
\qquad \text{high} \quad = \{(rt, s, x) \mid (rt, s, x) \in e, \forall(\_, s_i, \_) \in e : h\ s > h\ s_i\} \\
\qquad \text{low} \quad = \{(rt, s, x) \mid (rt, s, x) \in e, \exists(\_, s_i, \_) \in e : h\ s \leq h\ s_i\} \\
\qquad \text{lowExp} \quad = \{(rt', s', x \xrightarrow{\bar{R}} \text{State } s') \mid (rt, s, x) \in \text{low}, (\bar{R}, rt', s') \in \text{expand } (rt, s)\}
\end{array}$$

Figure 9: hTrace **search algorithm definition**

require h to be a monotonically decreasing function, which returns a lower value as the state comes closer to the desired goal g. Initially, this set will contain only one element, namely $(rt, s, \text{Leaf } s)$, where rt is the initial RuleTree, s the initial state, and Leaf s the trace that just contains the current state. If the set of current expansions contains one or more traces that lead to the goal, the algorithm returns those traces. If none of the expansions satisfies the goal, the expansions are scored using the scoring function h, and divided into two sets, one containing the lowest scoring expansions, and one containing the others. The lowest scoring expansions are then expanded. hTrace is called recursively on the union of the expanded traces and the low scoring traces.

6 IMPLEMENTATION

Our framework has been implemented in Haskell. Haskell is a purely functional programming language. It has a static type system and lazy evaluation. While this helps with the implementation, it is not crucial to the realisation of the system.

```
 firsts  :: Eq a => RuleTree a -> a
                               -> Maybe[(RuleSet a, RuleTree a)]
empty  ::          RuleTree a -> Bool
expand :: Eq a => RuleTree a -> a
                               -> Maybe[(RuleSet a, a, RuleTree a)]
 traces  :: Eq a => RuleTree a -> a -> [Trace a]

BFTrace :: Eq a => (Goal a) -> [(RuleTree a, a, [(a,RuleSet a)])]
                             -> [Trace a]
heuristicTrace :: Eq a => (Goal a) -> (a -> Int)
                        -> [(RuleTree a, a, [(a,RuleSet a)])]
                        -> [Trace a]
```

Listing 1: Type signatures of framework implementation

Listing 1 lists the types of the functions that correspond to the functions described in Sections 3 to 5. The full implementation can be found online [1].

We have also implemented two examples that use the framework to generate hints: Tic Tac Toe and a command and control system. Both examples are included in the full implementation available online. We discuss the command and control example in the following section.

6.1 Properties of the traces function

To validate our definition of $\mathcal{F}$, $\mathcal{E}$, expand and traces, we want to show them to be correct.

We do this by verifying the traces function to be sound and complete with respect to the RuleTree application semantics.

We consider traces to be sound if, for any RuleTree rt and initial state s, there exists an end state in the result of $rt \cdot s$ that is equal to the end state reached by every trace in traces(rt, s).

We consider traces to be complete if for all elements in the set of end states from the application of the RuleTree, there exists an element from traces, such that the end state of this trace is equal to the element of the end state set. Instead of showing soundness and completeness separately, we verify Conjecture 6.1, from which we can deduce the two.

Conjecture 6.1 (Correctness of traces). *For all* RuleTrees rt *and states* s *we have:*

$$\{s_n \mid s \xrightarrow{\bar{R}_1} \cdots \xrightarrow{\bar{R}_n} s_n \in \text{traces } (rt, s)\} = rt \cdot s.$$

We verify that our implementation works correctly by testing the correctness properties as formulated in Conjecture 6.1, using

[1]https://github.com/niconaus/rule-tree-semantics

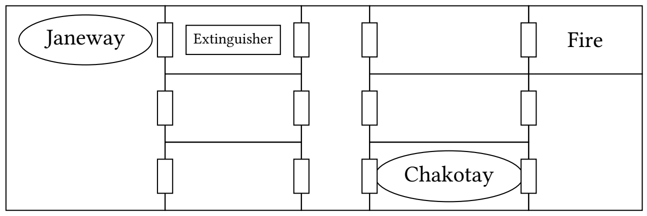


Figure 10: Rendering of an example initial state for the simplified Command & Control system with two workers: Janeway and Chakotay

QuickCheck [7]. QuickCheck generates random test cases for properties, based on the type signature of the input of a property. The translation of this Conjecture to Haskell is listed in listing 2.

```
rtEquality :: RuleTree [Int] -> [Int] -> Property
rtEquality rt s = (fromList (traceS rt s))
                    === (fromList (appS rt s))
```

Listing 2: Correctness property expressed in Haskell

6.2 Command & Control system

We take a look at a Command & Control application that was developed in cooperation with the Netherlands Royal Navy [31]. The goal of this application is to model workflows on board a navy ship. This includes workers, sensors, mission goals, resources and systems. Tasks can be assigned to users working on the vessel, and sensors are used to monitor the current situation.

For the sake of this example, we use a simplified version of the complete ship application. Workers can walk around the ship. When a fire breaks out, the workers have to walk to an extinguisher, pick it up, walk to the fire, and put it out. A visual representation of this example is shown in Figure 10.

The code below shows how we describe the problem in our DSL. Only the most important definitions are given. For the goal and heuristic functions, only the type signature is given here. The complete definitions can be found online [2].

```
data SimulationState = SimulationState [[Room]]
                                       (M.Map User Agent)

data User = User String
data Agent = Agent RoomNumber   -- Current position
                   Inventory
                   User         -- User that controls Agent

data Room = Room RoomNumber
                 (Int,Int)      -- Room coordinates
                 [Exit]         -- Rooms it has doors to
                 Inventory
                 RoomState
                 Int            -- Room depth

data Exit = ENorth RoomNumber
          | EEast  RoomNumber
          | ESouth RoomNumber
          | EWest  RoomNumber

data Inventory = NoItem | Extinguisher
data RoomState = Normal | Fire

shipTree :: RuleTree SimulationState
shipTree = Parallel (map (\usr -> Assign usr
                               (shipSimulation usr))
                         [Janeway, Chakotay])

shipSimulation :: User -> RuleTree SimulationState
shipSimulation usr
 = times 10
    (Choice
      [ Leaf (Condition (canPickup usr) (pickUp usr))
      , Leaf (Condition (canExtinguish usr)
      , Choice (map (\x -> (Leaf
                              (Condition (canMove usr)
                               (Rule (show x)
                                  (applyMove usr x )))))
                     [1..10])])

shipState     :: SimulationState
shipNotOnFire :: Goal SimulationState
shipHeuristic :: SimulationState -> Int

solveShip = heuristicTrace shipNotOnFire
                           shipHeuristic
                           [(shipTree, shipState, [])]
```

[2]https://github.com/niconaus/rule-tree-semantics

The first line models the state, and `shipTree` expresses the RuleTree, with the help of `shipSimulation`.

Assuming the system itself is also implemented in Haskell, the existing code from the implementation can be used when defining the RuleTree. Functions like `pickUp`, `canExtinguish` and `applyMove` can be the exact same code as the system implementation.

`shipNotOnFire` is the goal condition, and `shipHeuristic` is the heuristic used to score each state. To solve this problem, we plug these functions into the generic `heuristicTrace` algorithm, together with the RuleTree and a state, as shown on the last line. When we execute `solveShip`, we get back a trace that will lead the workers on the ship to the quickest way to extinguish all fires, if possible. If there is only a single fire, instructions for only one user will be generated. If there are multiple fires, both workers will perform actions at the same time, as described by the ruleTree.

This example clearly shows the advantage of our system: a programmer only needs to define the problem by describing it as a ruleTree, possibly reusing existing code, come up with a goal function and a heuristic, and then gets a multi-user solver for free.

7 RELATED WORK

A lot of work exists that attempts to assist users of workflow systems, from many different angles.

7.1 Assistive TopHat

In earlier work, we have presented a different approach to generate next-step hints for workflow systems, called Assistive TopHat [20]. Instead of letting the programmer relate the existing workflow to a generic flow structure, the structure of the workflow language itself is utilized. A symbolic execution engine developed previously for the workflow language TopHat [21] is used to generate all paths towards a user defined goal. From these paths, next-step hints are constructed and returned to the user.

The major downside of this approach is that a symbolic execution engine has to be either available or to be build for the target workflow language. The approach presented in this paper works for most workflow systems, and only requires programmers to relate their system to the RuleTrees.

7.2 Rule-based problem modelling

We follow in a long tradition of creating (domain-specific) languages that allow programmers to model rule-based problems, such as planning problems. Some of the early languages written for this purpose are STRIPS [8], PLANNER [12] and SITPLAN [9]. Most of these are based on the same principles as our approach, namely to describe state, operator set and goal test. For example, a STRIPS problem is defined as $\langle P, O, I, G \rangle$, where P is the set of states the problem can be in, O the set of operators, I the initial state, and G the goal state [6].

A more recent language is PDDL [18]. Version one of the language, from 1998, consists of a domain description, action set, goal description and effects. Again, these ideas coincide with our notion of a problem formalization. The PDDL standard has been updated several times [14], and there are many variants currently in use. These variants include MA-PDDL [15], which can deal with multiple agents, and PPDDL [35], which supports probabilistic effects.

The language we present is different from all of the aforementioned languages in several ways. Our language is a DSL, embedded in Haskell. This means that the programmer can use the full power of Haskell when constructing the problem description in our DSL. The languages mentioned above are not embedded in any language and therefore the programmer is limited to the syntax of the DSL in constructing the problem description. Another big difference is the fact that in all of the other languages mentioned, except PDDL, the state-space is finite. For example, in SITPLAN, part of the problem description is a finite set of possible situations, and in STRIPS, the set of states is defined as a finite set of conditions that can be either true or false. In our DSL, we do not limit the set of possible states. This allows us to describe many more problems in our DSL, but at the same time makes solving them harder.

The second part of our approach is to solve the problem described in our DSL. Comparing to other approaches, both SITPLAN and PDDL rely on general solvers, just like our approach. In fact, PDDL was initially designed as a uniform language to compare different planning algorithms in the AIPS-98 competition [18]. STRIPS and PLANNER however, do include a specific solving algorithm.

For each of the frameworks that we discussed in this section, there has been some research on generically solving problems. The Ideas framework includes a set of feedback services to generate hints for the user. For example, the basic.allfirsts service generates all steps that can be taken at a certain point in the exercise [10]. For the iTasks framework, a system was developed to inspect current executions by using dynamic blueprints of tasks [30]. It can give additional insight in the current and future states, but does not act as a hint-system and does not take a goal into account.

7.3 Workflow Analysis

Our work is also related to tools that analyse workflow systems. Basu and Blanning introduce metagraph [4] to describe workflows so that they can be better evaluated. Other approaches apply workflow mining to evaluate implementations [1]. Stutterheim et al. [32] present a system for generating visualisations from the source code of workflow systems implemented in the iTasks workflow framework. Their system Tonic also features dynamic inspection and limited path prediction. These approaches do not use their analyses to assist the end-user. Instead they focus on workflow and business optimisation from the system design perspective.

Research has also been done on systems that help end users in making choices. These decision support systems usually leverage some artificial intelligence approach like probabilistic reasoning [22] or planning [13]. These are all solutions that are custom made for a specific workflow system instance.

7.4 Decision Support Systems

As mentioned in the first section of this paper, a Decision Support System is defined as a system that models a certain domain and then assists the user in making choices by using analysis techniques [29]. There exists a great variety in both domains where DSSs are applied, as well as their implementation. Clinical DSSs support making decisions about the treatment of individual patients [5]. There are agricultural DSSs aimed to improve land use, planning and management of soil [25]. The biggest area of application is management and business [33]. Here, DSSs help managers make the right choices faster, better allocate resources or identify trends.

The basic design of a DSS consists of some representation of the domain, a reasoning engine and a way to communicate with the user.

Using a DSS has many advantages [24]. It improves the productivity of individuals, improves the quality of decisions and the speed with which they are made. Organizational control is improved, as well as communication between workers.

Employing a DSS comes with several challenges. First of all, there is a large financial risk involved, since it requires a significant investment [24]. The model that is used in the DSS limits the applicability of the system. When the domain or the problem changes, the model needs to be updated as well. Social issues may come up as well, workers may resist the change that comes with a DSS.

7.5 Electronic Performance Support Systems

Electronic Performance Support Systems (EPSS) focus on workers or individuals that have to achieve a certain goal or complete a task, but who do not yet have sufficient knowledge or are not sufficiently skilled yet. They facilitate on the job training by providing the user with just-in-time information on the task that they are working on [27].

An EPSS is typically composed of a user interface, giving access to generic tools like documentation and help systems, and application specific support tools such as tutorials [3]. Usually, the EPSS is geared towards the specific domain it is being used in, a certain business setting for example.

An EPSS can provide workers with just in time information on how to perform certain tasks. It cannot however assist hem in making decisions based on the precise situation that they are in. Only general documentation, help and guidelines can be offered.

The aim of our next-step hint system is not necessarily to provide training to workers, but to assist them with a specific goal and situation.

8 CONCLUSIONS

In this paper, we have demonstrated how to construct a complete and sound framework for calculating hints for multi-user workflow systems. By means of a DSL, we are able to describe problems in a uniform way, and make them tractable to generic solving algorithms. These algorithms produce traces that lead to the goal of the user. Besides a formal system, we have also presented a practical implementation. We have implemented two examples, one of which we have described in this paper.

To our knowledge, we are the first to describe a workflow solving system that works generically on a broad range of problems structured in a workflow.

8.1 Future work

Following the research presented in this paper, we see four different questions remaining.

8.2 iTasks integration

Currently, iTasks is the most used Task Oriented Programming (TOP) implementation. This programming paradigm calls the smallest pieces of work performed in a workflow environment *tasks*. These tasks are combined into bigger tasks using a combinator language. This programming structure makes it very suitable for the techniques described in this paper. The example presented in this paper is quite ad-hoc; programmers need to relate their structure to the RuleTree structure for each workflow system separately. The RuleTree structure could instead be integrated in the iTasks language to allow programmers to integrate the rule-based problem description in the actual task specification.

8.3 Hint presentation

The current implementation is a mere proof of concept. It is possible to calculate next-step hints, but there is currently no way to display hints in a user friendly manner. The information calculated by the system potentially contains duplicate hints and redundant or irrelevant information.

The same holds for the user defined goals, there is no user friendly way to set a goal. When implementing the hint framework into real-world applications, some research has to be done to determine how to display end-user hints and how to set goals.

8.4 Testing the effect of hints

The effectiveness of hints has been shown in other research, especially in the intelligent tutoring community [16, 28]. To validate the approaches proposed in this paper, it would be interesting to conduct empirical studies. This would allow us to determine the effectiveness of next-step hints in workflow systems.

8.5 Other kinds of feedback

In this paper, we focus mainly on providing next-step hints. Of course, there are many other possible forms of feedback.

In certain cases, it might be that a more general hint is more didactically effective. For example, when solving a math problem, it could be more useful to first tell a student what approach she could try, before actually suggesting a concrete step.

In interactive programs, it might be the case that certain steps are not available to a user. It would be useful to inform the user, why a step is unavailable. For example, it could be that she needs to wait on her colleague to perform some action.

A different angle would be to look at managers' information. It is possible to build a manager's overview with information on the progress of tasks in an ad-hoc manner, but we are also interested in developing a more generic way to offer managers feedback.

REFERENCES

[1] Wil M. P. van der Aalst. 2011. *Process Mining - Discovery, Conformance and Enhancement of Business Processes.* Springer.

[2] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. 2003. Workflow Patterns. *Distributed and Parallel Databases* 14, 1 (2003), 5–51.

[3] Philip Barker and Ashok Banerji. 1995. Designing electronic performance support systems. *Innovations in Education and Training International* 32, 1 (1995), 4–12.

[4] Amit Basu and Robert W. Blanning. 2000. A Formal Approach to Workflow Analysis. *Information Systems Research* 11, 1 (2000), 17–36.

[5] Eta S Berner and Tonya J La Lande. 2007. Overview of clinical decision support systems. In *Clinical decision support systems.* Springer, 3–22.

[6] Tom Bylander. 1994. The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence* 69, 1-2 (1994), 165–204.

[7] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00.* 268–279.

[8] Richard Fikes and Nils J. Nilsson. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2, 3-4 (1971), 189–208.

[9] N.I. Galagan. 1979. Problem description language SITPLAN. *Cybernetics and Systems Analysis* 15, 2 (1979), 255–266.

[10] Bastiaan Heeren and Johan Jeuring. 2014. Feedback services for stepwise exercises. *Science of Computer Programming* 88 (2014), 110–129.

[11] Bastiaan Heeren, Johan Jeuring, and Alex Gerdes. 2010. Specifying Rewrite Strategies for Interactive Exercises. *Mathematics in Computer Science* 3, 3 (2010), 349–370.

[12] Carl Hewitt. 1969. PLANNER: A Language for Proving Theorems in Robots. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence.* 295–302.

[13] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. 1998. Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence* 101, 1-2 (1998), 99–134.

[14] Daniel L. Kovacs. 2011. BNF definition of PDDL 3.1. (2011).

[15] Daniel L. Kovacs. 2012. A Multi-Agent Extension of PDDL3. *WS-IPC 2012* (2012), 19.

[16] James A Kulik and JD Fletcher. 2016. Effectiveness of intelligent tutoring systems: a meta-analytic review. *Review of Educational Research* 86, 1 (2016), 42–78.

[17] Bas Lijnse, Jan Martin Jansen, and Rinus Plasmeijer. 2012. Incidone: A Task-Oriented Incident Coordination Tool. In *Proceedings of ISCRAM.*

[18] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. PDDL-the planning domain definition language. *AIPS-98 planning committee* 3 (1998), 14.

[19] Nico Naus and Johan Jeuring. 2017. Building a generic feedback system for rule-based problems. In *Trends in Functional Programming - 17th International*

Symposium, TFP 2016. Springer.

[20] Nico Naus and Tim Steenvoorden. 2020. Generating next step hints for task oriented programs using symbolic execution. In *Trends in Functional Programming - 21st International Conference, TFP'20*.

[21] Nico Naus, Tim Steenvoorden, and Markus Klinik. 2019. A symbolic execution semantics for Tophat. In *IFL'19 (accepted for publication)*.

[22] Judea Pearl. 1989. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann.

[23] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. 2012. Task-oriented programming in a pure functional language. In *Principles and Practice of Declarative Programming, PPDP'12*. 195–206.

[24] Daniel J Power. 2002. *Decision support systems: concepts and resources for managers*. Greenwood Publishing Group.

[25] Diego de la Rosa, Francisco Mayol, Elvira Díaz-Pereira, Miguel Fernandez, and Diego de la Rosa Jr. 2004. A land evaluation decision support system (MicroLEIS DSS) for agricultural soil protection: With special reference to the Mediterranean region. *Environmental Modelling and Software* 19, 10 (2004), 929–942.

[26] Stuart J. Russell and Peter Norvig. 2010. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

[27] Paul Van Schaik, Robert Pearson, and Philip Barker. 2002. Designing electronic performance support systems to facilitate learning. *Innovations in Education and Teaching International* 39, 4 (2002), 289–306.

[28] Ramesh Sharda, Steve H Barr, and James C MCDonnell. 1988. Decision support system effectiveness: a review and an empirical test. *Management science* 34, 2 (1988), 139–159.

[29] Jung P. Shim, Merrill Warkentin, James F. Courtney, Daniel J. Power, Ramesh Sharda, and Christer Carlsson. 2002. Past, present, and future of decision support technology. *Decision Support Systems* 33, 2 (2002), 111–126.

[30] Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2015. Static and Dynamic Visualisations of Monadic Programs. In *Implementation and Application of Functional Languages, IFL'15*. 1–13.

[31] Jurriën Stutterheim, Peter Achten, and Rinus Plasmeijer. 2016. C2 Demo. (2016).

[32] Jurriën Stutterheim, Rinus Plasmeijer, and Peter Achten. 2014. Tonic: An Infrastructure to Graphically Represent the Definition and Behaviour of Tasks. In *Trends in Functional Programming - 15th International Symposium, TFP'14*. 122–141.

[33] Efraim Turban. 1988. *Decision support and expert systems: Managerial perspectives*. Macmillan.

[34] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. 13–26.

[35] Hakan L.S. Younes and Michael L. Littman. 2004. PPDDL1. 0: The language for the probabilistic part of IPC-4. In *Proc. International Planning Competition*.

Asynchronous Shared Data Sources

Mart Lubbers
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
mart@cs.ru.nl

Haye Böhm
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
haye.bohm@gmail.com

Pieter Koopman
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
pieter@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences
Radboud University
Nijmegen, The Netherlands
rinus@cs.ru.nl

ABSTRACT

to appear

KEYWORDS

Task Oriented Programming, Uniform Data Sources, Functional Programming, Distributed Applications, Clean

ACM Reference Format:
Mart Lubbers, Haye Böhm, Pieter Koopman, and Rinus Plasmeijer. 2021. Asynchronous Shared Data Sources. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'20).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Complex applications deal with many different types of data. These data sources may represent data from the system itself, a database, shared memory, external data streams or even physical data retrieved by a person. Consequently, each data source has different intrinsic properties, methods and swiftness of accessing and may therefore require each their own separate interface.

Shared Data Sources (SDSs) are an extension of Uniform Data Sources (UDSs) [3] and provide an atomic, uniform and composable interface over abstract data for functional languages. This abstract data can be anything ranging from data in a state, interaction with the file system to system resources such as time and random numbers. SDSs are wholly defined by their atomic read and write functions, i.e. given a linear state, they either read or write the source and yield the state again and no synchronisation is required. Furthermore, by slightly defunctionalising the read and write functions a first-order parametric view can be created with which it is possible to implement a lean and mean notification mechanism [1]. These properties make them suitable for Task Oriented Programming (TOP) frameworks such as iTask [4] and mTask [2] to share data between tasks. A downside of this data model is that access to the underlying data is synchronous. In other words, it finishes in one go and while doing so, everyone else has to wait. This makes it strenuous to implement data sources for which the operations might depend on Operating System (OS) functionality such as *select*.

1.1 Research contribution

The research contribution of this paper are two extensions to the SDSs:

- The deeply embedded DSL housing the SDSs is changed to a class-based embedding allowing for finer control and combinators with more precise constraints.
- The functions for the SDS operations are changed to rewriting functions making the SDSs asynchronous (ASDSs). A server can then choose to interleave operations so that it can do things while operations are in progress in the background.
- A proof of concept implementation of the novel model in the iTasks framework showing practical ASDSs modelling data from the internet or interleaved calculations.

2 SHARED DATA SOURCES

2.1 Uniform Data Sources

UDSs are housed in a single data structure parametrized by a read and write type and the monad in which they operate. All constructors of the type represent a type of UDS or combinator. For example, the following constructors contain UDS definitions for sources that read directly in the monad and write in it.

$$
\begin{array}{rl}
::\ \mathit{UDS}\ m\ r\ w = & \mathit{Source}\ (m\ r)\ (w \leftarrow m\ ())\\
\mid & \exists r'\ w' : \mathit{CRead}\ (\mathit{UDS}\ m\ r'\ w)\ (r' \rightarrow \mathit{UDS}\ m\ r\ w')\\
\mid & \ldots
\end{array}
$$

The read and write functions operate on this data type. In practice these functions contain some error handling as well.

```
read :: (UDS m r w) → m r
read (Source rfun _) = rfun world
read (CRead sds rfun) = read sds >>= read . rfun
read ...

write :: w (UDS m r w) → m ()
write w (Source _ wfun) = wfun w
write w (CRead sds _) = write w sds
write ...
```

In single threaded systems such as the iTask system, UDSs can be used for SDSs as well. The read and write operations are atomic for the threaded unique state, hence all data sources are under the exclusive control of the iTask server and no synchronisation is necessary.

2.2 Parametric Lenses

Parametric lenses are an extension to SDSs allowing the programmer to focus on parts of the data. By defunctionalising the SDS combinators, parts of the shared data can be read, written and a task can be notified when a relevant portion of the share changes. The parameter of the SDS is added as an extra type parameter to the datatype:

```
:: SDS m p r w =  Source (p → m r) (p w → m())
               |  ∃p′ r′ w′ : LensRead (SDS m p′ r′ w′) (p → p′)
                              (p r′ → r) (p w r′ → w′)
               |  ...
```

3 CLASS-BASED SHARED DATA SOURCES

Lifting the SDS access functions to classes allows every type of SDS to be defined in their own datatype. This increases modularity since it removes the need to revisit the original datatype and update all of its functionality but allows you to create a new type of SDS orthogonally.

```
class Readable sds where
    read :: (sds m p r w) → m r | Monad m
class Writable sds where
    write :: w (sds m p r w) → m () | Monad m
```

Furthermore, by parametrising the SDS datatype with the SDS type of the possible children, fine-grained constraints can be placed on the class instances as is shown later. For example, the *Source* SDS can now be defined by just combining a *ReadSource* and a *WriteSource* and putting them in a datatype.

```
:: ReadSource m p r w = ReadSource (p → m r)
:: WriteSource m p r w = WriteSource (p w → m ())
:: RWPair sdsr sdsw m p r w = RWPair (sdsr m p r w) (sdsw m p r w)
```

In the previous models, a readonly SDS was just a regular SDS for which writing was is a no-op. A write only SDS was just a regular SDS from which only unit could be read. With the novel approach, these operations are downright impossible because the program would already be rejected during compilation.

```
instance Readable ReadSource where ...
instance Writable WriteSource where ...
instance Readable (RWPair sdsl sdsr) | Readable sdsl where ...
instance Writable (RWPair sdsl sdsr) | Writable sdsr where ...
```

4 ASYNCHRONOUS SHARED DATA SOURCES

Changing the model to be asynchronous requires changing the results of the read and write operations. If it happens to be that an operation is not done yet, a new SDS is yielded that can be used to continue the operation. This new SDS does not have to be of the same type as the original SDS but it has to have at least the same constraint, i.e. *Readable* for the read operation and *Writable* otherwise.

```
:: ReadResult m p r w =   Read r
                      |   ∃.sds : Reading (sds m p r w) & Readable sds
:: WriteResult m p r w =  Written ()
                      |   ∃.sds : Writing (sds m p r w) & Writable sds
```

This approach still allows for the synchronous approach with helper functions:

```
getShare :: (sds m () r w)  → m r | Monad m & read sds
getShare s = read s () >>= λv → case v of
             Reading s = getShare s
             Read r = pure r

setShare :: w (sds m () r w) → m () | Monad m & write sds
setShare w s = write s () w >>= λv → case v of
             Writing s = setShare w s
             Written _ = pure ()
```

5 ASYNCHRONOUS SDSS IN ITASKS

TOP is a declarative programming paradigm...

6 RELATED WORK

To appear

7 CONCLUSION

To appear

8 FUTURE WORK

To appear

ACKNOWLEDGMENTS

This research is partly funded by the Royal Netherlands Navy.

REFERENCES

[1] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric lenses: change notification for bidirectional lenses. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, 9. http://dl.acm.org/citation.cfm?id=2746333

[2] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2019. Interpreting Task Oriented Programs on Tiny Computers. In *Proceedings of the 31th Symposium on the Implementation and Application of Functional Programming Languages*. ACM, Singapore, 12.

[3] Steffen Michels and Rinus Plasmeijer. 2012. Uniform data sources in a functional language. (2012).

[4] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-oriented programming in a pure functional language. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*. ACM, 195–206.

Dynamic Editors for Well-Typed Expressions

Pieter Koopman
Radboud University
Nijmegen, The Netherlands
pieter@cs.ru.nl

Steffen Michels
TOP Software Solutions, The Netherlands
The Netherlands
steffen@top-software.nl

Rinus Plasmeijer
Radboud University
Nijmegen TOP Software Solutions
The Netherlands
rinus@cs.ru.nl

ABSTRACT

Interactive systems can require complex input from their users. A grammar specifies the allowed expressions in such a Domain Specific Language, DSL. An algebraic DataType, ADT, is a direct representation of such a grammar. For most end-users a structured editor with pull-down menus is much easier to use than a free text editor. The iTask system can derive such structured editors based on an ADT using datatype generic programming. However, the input DSL has often also semantical constraints, like proper use of types and variables. A solution is to use a shallow embedded DSL or a DSL based on a Generalized ADT to specify the input. However, such a specification cannot be handled by datatype generic programming. Hence, one cannot derive structured editors for such a DSL.

As a solution we introduce structured web-editors that are based on dynamic types. These dynamic types are more expressive; they can express the required DSL constraints. In the new dynamic editor library we need to specify just the dynamic relevant for the DSL. The library takes care of displaying the applicable instances to the user and calls itself recursively to create the arguments of the dynamic functions. In this paper we show how this can be used to enforce the requires constraints on ADTs, to create structured web-editors for shallow embedded DSLS, and to create those editors for GADT based DSLs.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Graphical user interface languages**.

ACM Reference Format:
Pieter Koopman, Steffen Michels, and Rinus Plasmeijer. 2020. Dynamic Editors for Well-Typed Expressions. In *IFL20: The 32nd Symposium on Implementation and Application of Functional Languages.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Many programs accept quite complex inputs specified by some Domain Specific Language, DSL, Most domain experts prefer structured text editors with pull-down menu's to create an inout over a free text editor since the structured editor provides more guidance. An Algebraic DataType, ADT, can representation of the syntax of the input language directly. The iTask system can derive structured web-editors for such an ADT based DSL by generic programming [3, 4, 6]. This yields a web-editors that yields proper instances of the ADT for free.

The input DSL has typically also semantic constraints like the proper definition of identifiers and type restrictions. The system can only enforce correctness of the ADT in the host language, but not the additional DSL constraints. Implementing a type-checker of the DSL is significant and nontrivial work. Moreover, it acts too late; it rejects the DSL construct that the user has made by the structure editor instead of guiding the user during the creation of the expression.

The dynamics in Clean offer a convenient way to perform dynamic type-checks and runtime unification [2, 15]. In this paper, we build web-editors based on these dynamic using the new dynamic editor library of the iTask system. With this library one has to specify just one dynamic for each DSL construct. The system selects the items that can be applied in the current context and creates appropriate web-editors for the arguments of the construct.

Section 2 shows how we can derive a structured editor for a ADT based DSL. In Section 3 we show how we can enforce type constraints on this DSL by the new dynamic editor library.

The functions used in a shallow embedded DSL can express most constraints of the DSL. Since the DSL consists of functions instead of an ADT one cannot derive a structured editor for such a DSL. In Section 4 we show how we can define a type-safe structured editor for such a DSL using our dynamic editors.

Generalized ADTs use richer types to express the constraints of the DSL in the datatype [4, 6]. In this paper we use a version of GADT based on bimaps [2, 6]. Due to functions and existentially quantified type-variables used in those types we cannot derive structured editors for those GADTs. In Section 5 we create structured editors for such a GADT using the dynamic editors.

We have used dynamic editors successfully to create queries over the combination of ships, their movements, history, owners and cargo in a system for the Dutch coast guard. We are developing an application to assign task dynamically to Super Sensors [8]. This system is based on cheap and energy-efficient microprocessors instead of Raspberry Pi's using mTask our DSL for programming the IoT [11, 12]. The mTask system itself is a Tagless DSL [5] that interoperates seaminglessly with the iTask system.

The main contributions of this paper are:

- it introduces dynamic editors, these structured web-editors are used to create DSL-expressions interactively while enforcing type-constraints on the fly;
- we demonstrate how to enforce type-constraints on editors for ordinary ADTs;

IFL20, September 2–4, 2020, Kent
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$XX.00
https://doi.org/10.1145/1122445.1122456

- we show how to create type-safe editors for shallow embedded DSLs;
- we present concise type-safe dynamic web-editors for GADTs;
- we demonstrate how a pool of type variables is used to ensure that only existing variables of the desired type can be used in the dynamic editor. Even GADTs are not able to do this on their own.

The examples used in this paper are all well-known DSLs to demonstrate the power of our approach. This is a somewhat atypical use of the dynamic editors and certainly not a limitation of the approach.

The dynamic editor library used in this paper is part of the standard iTask system available at https://clean.cs.ru.nl/ITasks. The examples in this paper will be published on github as soon as the paper is accepted.

2 BASIC WEB-EDITORS

The iTask system [3, 16] is a DSL embedded in Clean [1, 17] for Task Oriented programming, TOP. This system offers basic tasks like web-editors as well as combinators to compose subtasks to larger tasks. In this paper, we focus on web-editors. We use a very limited set of combinators that will be described briefly at their first use.

In iTask there are web-editors for basic types like integers, Booleans, lists, record fields strings. The system is able to derive tailor-made web-editors for ADTs using generic programming. This implies that the usual restrictions of generics apply; all types must be known and function types are not allowed. The system works fine for (recursive) ADTs and records.

As an example, we show an ADT representing a simple DSL over integer and Boolean values with a limited number of operations.

```
:: Expr
    = Int  Int
    | Bool Bool
    | Add  Expr Expr        // Integer addition
    | And  Expr Expr        // Boolean conjunction
    | Eq   Expr Expr        // Equality for integers and Booleans
    | If   Expr Expr Expr   // Conditional expression
```

The smallest program making an interactive structural web editor for `Expr` in iTask is:

```
derive class iTask Expr

Start :: *World → *World
Start world = doTasks (updateInformation [] (Int 0)) world
```

A few screen shots in Figure 1 from the browser illustrate the behavior of this program.

Our example `Expr` reveals already the limitations of this system. Since expressions like `Add (Int 1) (Bool False)` are well-typed in the host language they are accepted by the editor. However, in DSL terms we consider it to be a type-error.

There are several solutions to such problems. First, we can make more sophisticated ADTs as explained in Section 2.1. Next, we can make a dynamic editor for the type above that enforces well-typed instances as shown in Section 3. Finally, one can use other representations of the DSL that is able to enforce the required type-constraints. In Section 4 we use a dynamic editor for a shallow

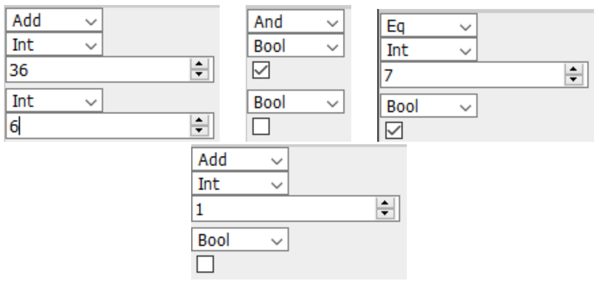


Figure 1: Some screenshots of the editor for expression in use.

embedded DSL. Section 5 uses a GADT like representation of the DSL.

2.1 Better Algebraic Data-Types

Inspired by the Nielson's we can use separate data for Integer and Boolean expressions in our DSL [14]. We use `ExprI` for Integer expressions and `ExprB` for Boolean expressions. We can derive editors for these types and make a web-editor just like for `Expr` above.

```
:: ExprI = Int  Int  | Add ExprI ExprI | IfI ExprB ExprI ExprI
:: ExprB = Bool Bool | And ExprB ExprB | IfB ExprB ExprB ExprB
         | EqB ExprB ExprB | EqI ExprI ExprI
```

Although this works perfectly for this tiny example the limitations are also obvious. Since there is no overloading in this representation we had to copy the condition `If` and the equality `Eq` to cope with different types. With only two types and a small set of operations in the DSL this is bearable. However, this quickly becomes unpleasant for more serious DSLs. We get an additional datatype for each type in the DSL and copies of overloaded operators for those types.

3 DYNAMIC EDITORS FOR RESTRICTING ALGEBRAIC DATA TYPES

Another solution to prevent type problems in a DSL is by type-checking the expressions entered by the user. Preferably the system performs the type checks on-the-fly; while the expression is created by the user. Such a dynamic type check can be quite tricky since we want overladed operators, like equality, in the DSL. This implies that the type of arguments is not always known. We want a system that checks the type of the arguments as soon as the subexpressions are given. It should not be delayed until the entire expression is known.

As a consequence, we need a runtime type-checker that is able to handle overloading. Immediate extensions are class restrictions and cooperation with the type-system in the host-language. Implementing such a type-checker is a nontrivial and significant effort. Instead of implementing such a type-checker, we will use the dynamic types of Clean to guarantee well-typed expressions in our DSL [2, 15]. In the next sub-section, we briefly review this dynamic system.

3.1 Dynamics

In Clean a value of any type can be transformed to the type `Dynamic`. Instances of `Dynamic` are ordinary values in Clean. So, dynamic values can be stored in data-structures like lists, be the argument or result of a function and so on.

The keyword **dynamic** is used to transform a value of any type to a value of type `Dynamic`. One can specify a type of the value but this is not necessary if the compiler can determine the type. Some typical examples are **dynamic** `36`, **dynamic** `False::Bool`, **dynamic** `(+) 1::Int→Int` and **dynamic** `map::∀ a b:(a→b) [a]→[b]`. When the type is completely polymorphic, one has to add the class constraint `TC` for the type.

```
toDynamic :: a → Dynamic | TC a
toDynamic x = dynamic x
```

For instance, we can pack expressions of various types as dynamic in a list.

```
list :: [Dynamic]
list = [dynamic 7, dynamic (fib, 4), dynamic (str2int, "30")
       ,dynamic "1", dynamic (+) 1]
```

It makes only sense to pack values in a `Dynamic` if we can also unpack them. Since Clean is a strongly typed language, this has to be done in such a way that it does not break the strong type-system. This is achieved by a pattern match on the types stored in a `Dynamic` value. The alternative is not applicable if the type does not match the pattern. This is demonstrated in the function `dSum` that sums some of the dynamics in the give list to an integer value:

```
dSum :: [Dynamic] → Int
dSum [a::Int       : rest] = a + dSum rest
dSum [t::(a→Int,a): rest] = fst t (snd t) + dSum rest
dSum [dyn          : rest] = dSum rest
dSum list                  = 0
```

When we apply `dSum` to the `list` above the result will be `42`. This example demonstrates that we can use type variables in the dynamic type match. A type variable from the static type of the function containing the dynamic match is denoted as `a^`.

```
toA :: Dynamic → Maybe a | TC a
toA (x :: a^) = Just x
toA _         = Nothing
```

3.2 Dynamic Editors

The `DynamicEditor` library in the iTask system allows programmers to specify web-editors for types that cannot be handled by the deriving mechanism. This library uses as much as possible of the existing iTask infrastructure. This guarantees a smooth interaction and a similar look and feel.

A dynamic editor is parameterized by a list of (grouped) editor elements of type `DynamicCons`. This dynamic constructor is an abstract type that can be constructed by `functionConsDyn`. This function has two strings and a dynamic as arguments. The first string is a unique identifier used to turn dynamic editor values to ordinary values. The second string is the name shown in the editor to the user. The dynamic contains the value produced by this editor element. If this dynamic contains a function the system is used recursively to produce the arguments needed to turn the function into its result type. This might include type variables and dynamic unification. Only those elements that can produce a value of the demanded type are shown when the system can determine the desired result.

```
:: DynamicEditor a =: DynamicEditor [DynamicEditorElement]
:: DynamicEditorElement
    = DynamicCons      DynamicCons
    | DynamicConsGroup String [DynamicCons]
```

```
functionConsDyn :: DynamicConsId String Dynamic → DynamicCons
:: DynamicConsId :== String
```

With `dynamicEditor` a dynamic editor can be turned into a real iTask editor that produces a `DynamicEditorValue`. There is a variant from this function that is parameterized by a state, we will illustrate its use in Section 4.1. This is a somewhat complicated internal representation of the state of a dynamic editor. With `valueCorrespondingTo` we can extract the actual value from such a state.

```
dynamicEditor             :: (DynamicEditor a)
                          → Editor (DynamicEditorValue a) | TC a
parametrisedDynamicEditor :: (p→DynamicEditor a)
                          → Editor (p,DynamicEditorValue a)
                          | TC a & gEq{|★|}, JSONEncode{|★|}
                          , JSONDecode{|★|} p
valueCorrespondingTo :: (DynamicEditor a) (DynamicEditorValue a)
                     → a | TC a
```

The internals of this editor are somewhat complicated because it has to do many things simultaneously; creating an editor, selecting and displaying its elements, creating arguments, use the dynamic system or unification and producing tailor-made errors messages if argument unification fails. Fortunately, the use of the system is less complicated. The remainder of this paper describes several ways to use these dynamic editors.

3.3 A Type-Safe Dynamic Expression Editor

We start with a type-safe editor for the expression from Section 2. We use the same datatype to represent the expressions. During construction of the expression, we use a phantom type that mimics the type represented by the expression constructed. The additional type is a reusable solution to add a phantom type `b` to a type given as the type parameter `a`.

```
:: Typed a b =: Typed a
```

```
exprEditor :: DynamicEditor Expr
exprEditor = DynamicEditor
 [ DynamicCons $ functionConsDyn "Expr" "(enter expr)"
    (dynamic λ(Typed e) → e :: ∀ a: (Typed Expr a) → Expr)
 , DynamicConsGroup "Integer"
   [ functionConsDyn "Int" "integer value"
      (dynamic λi → Typed (Int i) :: Int → Typed Expr Int)
   , functionConsDyn "Add" "add"
      (dynamic λ(Typed x) (Typed y) → Typed (Add x y) ::
      (Typed Expr Int) (Typed Expr Int) → Typed Expr Int)
   ]
 , DynamicConsGroup "Boolean"
   [ functionConsDyn "Bool" "Boolean value"
       (dynamic λb → Typed (Bool b) :: Bool → Typed Expr Bool)
   , functionConsDyn "And" "and"
       (dynamic λ(Typed x) (Typed y) → Typed (And x y) ::
       (Typed Expr Bool) (Typed Expr Bool) → Typed Expr Bool)
   , functionConsDyn "Eq.Int" "eq Int"
       (dynamic λ(Typed x) (Typed y) → Typed (Eq x y) ::
       (Typed Expr Int) (Typed Expr Int) → Typed Expr Bool)
   , functionConsDyn "Eq.Bool" "eq Bool"
       (dynamic λ(Typed x) (Typed y) → Typed (Eq x y) ::
       (Typed Expr Bool) (Typed Expr Bool) → Typed Expr Bool)
   ]
 , DynamicConsGroup "Conditional"
```

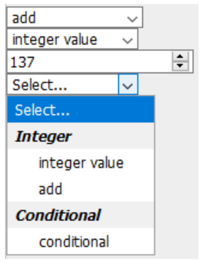


Figure 2: The dynamic editor.

```
  [ functionConsDyn "If" "conditional"
      (dynamic λ (Typed c) (Typed t) (Typed e) → Typed (If c t e) ::
     ∀ a: (Typed Expr Bool) (Typed Expr a) (Typed Expr a)
           → Typed Expr a)
  ]
, DynamicConsGroup "Editors"
  [ customEditorCons "Int.Val"  "enter integer value" intEditor
  , customEditorCons "Bool.Val" "enter boolean value" boolEditor
  ]
]

intEditor :: Editor Int
intEditor = gEditor{|★|}
```

Figure 2 shows the generated editor in action. The user is constructing an addition with `137` as the first argument. Only the options that can produce an integer value are shown in the dropdown box for the second argument.

This approach works, but it has two drawbacks. First, the host language is not able to check the given phantom types. An erroneous type in the definition is happily accepted by the system. For instance, `dynamic i→ Typed (Int i) :: Int → Typed Expr Bool` with result type `Bool` instead of `Int` will treat expressions like `Int 1` incorrectly as a Boolean in the DSL. The second drawback is that there is no overloading in the equality. Below we introduce solutions for these limitations.

In the actual implementation, we use some tuning combinators to improve the layout. This tuning is omitted here for brevity.

4 DYNAMIC EDITORS FOR SHALLOW EMBEDDED DSLS

Dynamic editors are focussed on creating instances of datatypes. In deep embedded DSL the expressions are function applications. nevertheless, we can use dynamic editors to create expressions in a deep embedded DSL. The key step here is to pack these functions in a datatype.

4.1 Identifiers in the DSL

Many DSLs contain identifiers. These identifiers are used to indicate variables, functions, function arguments etc. The identifiers are typically distinguished by a unique name or number. There are two, related, problems with these identifiers in a type-safe DSL. First, we have to guarantee that all used identifiers are indeed defined in the given DSL program. Second, it is important to ensure that the identifier represents a value of the desired type.

In this paper, we tackle these problems by using a user-defined set of typed identifiers. The user of the editor can always add new variables, even while constructing a DSL expression. All variables get an initial value at their definition. In the DSL expression editor, the user can only select an element from the current set of variables[1].

Variables in the DSL are represented by records of type `Bind`. The editor ensures that the strings identifying the variables are unique. Since the identifiers are bound to values of various types, we store these values as dynamics in the `State`.

```
:: Bind a = {idnt::String, val::a}
:: State :== [Bind Dynamic]
```

A list of bindings is convenient in the iTask editor for variables. Since we intend to use relatively small and simple DSL-expressions, the list is also efficient enough. Without much effort, we can replace the list with a more efficient storage structure, like a `map`.

To work with this state in iTask editors we derive everything in the class `iTask`. To set values in the state and get them from the state we have the obvious functions.

```
derive class iTask Bind

getVal :: String State → Maybe Dynamic
setVal :: String Dynamic State → State
```

To ensure that the dynamic editor is always using the current list of identifiers we put this state in a Shared Data Source, SDS. With standard iTask technology we make an editor task for this state

```
identifierEditor :: (SimpleSDSLens State) → Task State
identifierEditor sds =
 (   Title "Identifiers" @≫
     editChoiceWithShared [ChooseFromGrid showBinding] sds Nothing)
 ||-  (Title "Add new identifier" @≫
       Hint "Identifier names must be unique" @≫
       forever
          (get sds @ map (λb→b.idnt) ≫=λvars →
           enterInformation [] >>*
           [OnAction
              (Action "Add")
              (ifValue (λdef → not (isMember (def.idnt) vars))
              (λdef → upd (λl →
                sort [{idnt=def.idnt, val=varVal def.val}: l]) sds))
           ]))
```

The value of each identifier is stored as a dynamic. To make such values we use an additional type in the task `identifierEditor`. In the current dynamic editor is able to use the types integer, Boolean as well as (higher-order) functions over these types. Since we cannot make functions directly, we use the datatype `IdType` to specify the type of the desired function. The function `idDyn` creates a dynamic with a value of the desired type.

```
:: IdType = Int | Bool | Fun IdType IdType

idDyn :: IdType → Dynamic
idDyn Int  = dynamic 0
idDyn Bool = dynamic False
idDyn (Fun x y)
 = case (idDyn x, idDyn y) of (a::a, b::b) = (dynamic (λa→b) :: a→b)
```

[1]It would be more convenient to construct the set of defined variables on-the-fly while constructing the DSL-expression. The current version of iTask editors is not capable to change the state on-the-fly. Such an extension editors is currently under construction.

Finally, `showBinding` is used to display the type of the dynamic type of the current identifier bindings as a table in the task.

```
showBinding :: (Bind Dynamic) → Bind String
showBinding {idnt,val}
 = {idnt=idnt, val=toString (typeCodeOfDynamic val)}
```

For convenience the editor below starts with some predefined identifiers. However, it is perfectly possible to make all identifiers dynamically.

4.2 A Dynamic Editor for the Lambda-Calculus

To demonstrate the power of this approach we show how to make a dynamic editor for the simply typed lambda calculus. For convenience, we add some integer functions, equality, conditionals and the Y-combinator to our DSL. Every values in our shallowly embedded DSL is a function with the state as an argument. Since we have no side-effects, the state is not a reduction result and is not threaded through the evaluator.

```
:: Val a :== (State → a)
```

Our DSL for the shallowly embedded lambda-calculus is the set of functions below. Integer manipulations like subtraction and multiplication are omitted for brevity.

```
:: Lam a :== (State → a)

ap :: (Lam (a→b)) (Lam a) → Lam b
ap f x = λs.(f s) (x s)

abs :: String (Lam b) → Lam (a→b) | TC a
abs v body = λs. λarg.body (setVal v (dynamic arg) s)

Y :: (Lam (a→a)) → Lam a
Y f = ap f (Y f)

var :: String → (Lam a) | TC a
var v = λs.case getVal v s of
          Just (a::a^) = a
          _ = abort (v +++ " not properly bound")

lit :: a → Lam a | == a
lit a = λs.a

add :: (Lam Int) (Lam Int) → Lam Int
add x y = λs.x s + y s

eq :: (Lam a) (Lam a) → Lam Bool | == a
eq x y = λs.x s == y s

If :: (Lam Bool) (Lam a) (Lam a) → Lam a
If c t e = λs.if (c s) (t s) (e s)

and :: (Lam Bool) (Lam Bool) → Lam Bool
and x y = λs.x s && y s
```

To make a dynamic editor for this lambda calculus we use the tooling introduced in the previous section and the type `Val a` introduced here. For most constructs, there is a dynamic editor clause corresponding to the function listed above.

The handling of variables deserves some additional attention. We distinguish variable introduction for `abs` and variable use as an element of an expression. For variable introduction we use `Name` to select a variable from the shared state. The identity function used as the first argument is used to ensure the type-constraint `TC` on expressions in our DSL. When a dynamic requires such a class restriction it is turned by the compiler into an additional dictionary argument of that dynamic function. There is no easy way to make such dictionaries in our dynamic editors. Hence, we need some other way to tell the compiler which instance of the class has to be used. The function `toName` convert an element from the state to the corresponding dynamic construct containing the name.

For applied occurrences of identifiers, we use `toDynamicCons`. This function transforms a state element to typed dynamic that will extract the corresponding value from the state during evaluation.

```
:: Name a = Name (a→a) String & TC a

toName :: (Bind Dynamic) → DynamicCons
toName {idnt,val} = case val of
  (x::t) = functionConsDyn ("Name." +++ idnt) idnt
                           (dynamic (Name id idnt) :: Name t)

toDynamicCons :: (Bind Dynamic) → DynamicCons
toDynamicCons {idnt, val} = case val of
  (x::t) = functionConsDyn ("Var." +++ idnt) idnt
                           (dynamic (var idnt) :: Val t)
```

With these elements, we can construct a dynamic editor for our typed shallowly embedded lambda calculus. The cases for name introduction and variable application are:

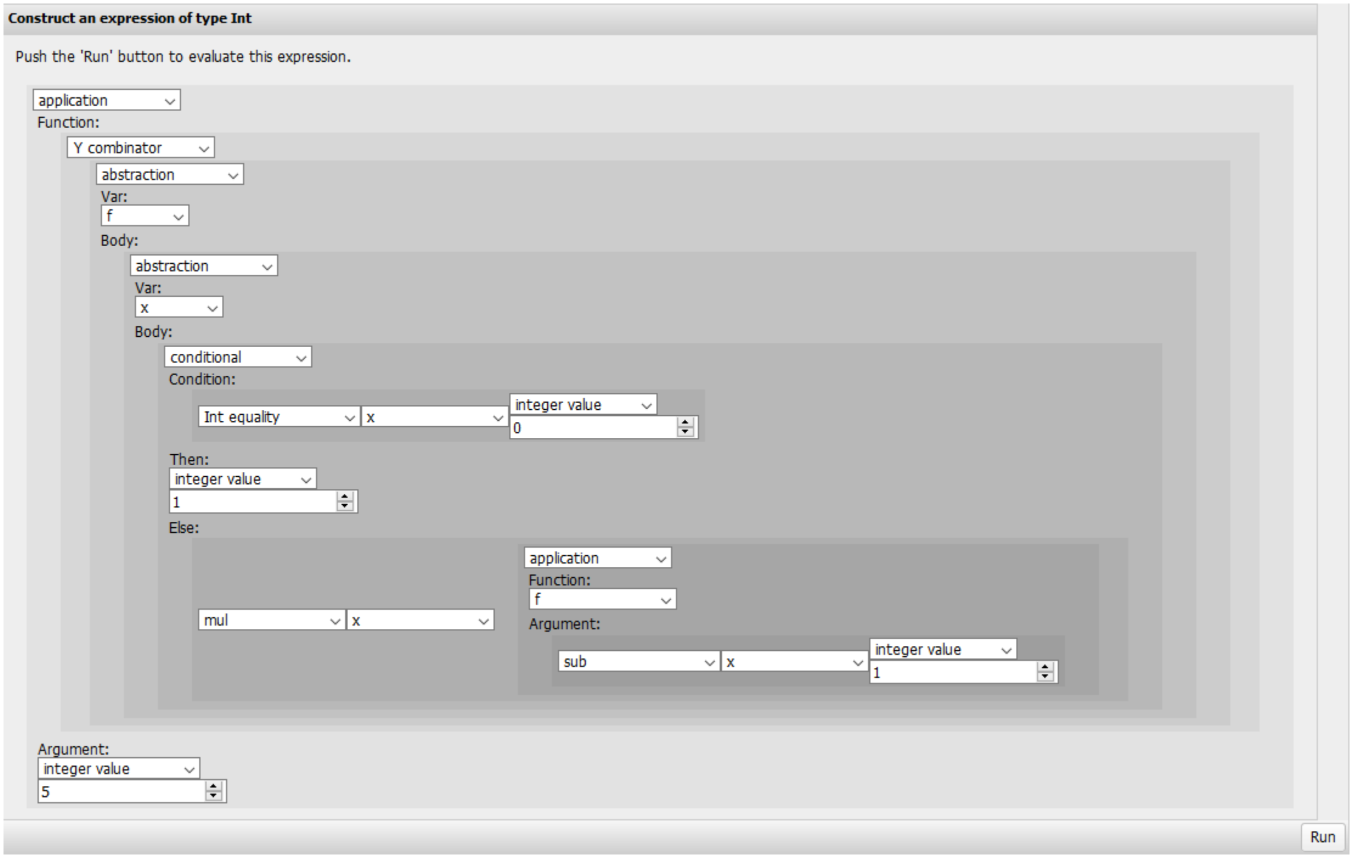
```
exprEditor :: State → DynamicEditor (Val v)
exprEditor state = DynamicEditor
 [ DynamicConsGroup "Variables" (map toDynamicCons state)
 , DynamicConsGroup "Names"     (map toName state)
 ..
```

For the other cases we skip the groups, names and layout information. For brevity, we just list the relevant dynamics for the construction of the editor.

```
 ,dynamic λi → (lit i) :: Int → Val Int
 ,dynamic λx y → (add x y) :: (Val Int) (Val Int) → Val Int
 ,dynamic λb → (lit b) :: Bool → Val Bool
 ,dynamic λx y → (and x y) :: (Val Bool) (Val Bool) → Val Bool
 ,dynamic λx y → (eq x y) :: (Val Int) (Val Int) → Val Bool
 ,dynamic λx → (Not x) :: (Val Bool) → Val Bool
 ,dynamic λc t e → (If c t e) ::
                  ∀ a: (Val Bool) (Val a) (Val a) → Val a
 ,dynamic λf x → (ap f x) :: ∀ a b: (Val (a→b)) (Val a) → Val b
 ,dynamic λf → (Y f) :: ∀ a b: (Val ((a→b)→(a→b))) → Val (a→b)
 ,dynamic λ(Name f x) (body) → (λs a.(abs x body) s (f a))
             :: ∀ a b: (Name a) (Val b) → Val (a→b))
 ]
```

The complete task ensures that only values of type `Int` or `Bool` can be produced. This guarantees that we have not to cope with abstractions as a result. Any abstraction will be applied to an appropriate argument.

Figure 3 shows the editor in action. The complete iTask program runs this editor in parallel with the editor to introduce identifiers from Section 4.1. The program contains a button to evaluate the current expression. After such a reduction the result is shown. Since the value of the editor is stored in a shared data store, we can always



Construct an expression of type Int

Push the 'Run' button to evaluate this expression.

application
Function:
Y combinator
abstraction
Var:
f
Body:
abstraction
Var:
x
Body:
conditional
Condition:
Int equality
x
integer value
0
Then:
integer value
1
Else:
mul
x
application
Function:
f
Argument:
sub
x
integer value
1
Argument:
integer value
5

Figure 3: The editor for shallow embedded lambda calculus in action.

return to the value in the editor with a button and update it. The dynamics ensure that one can only make well-typed expressions.

The value created with this editor in Figure 3 is the application of the familiar factorial function applied to the argument 5. From the state we used the variable `f` with type `(Int→Int)→(Int→Int)` and `x` of type `Int`. Reduction yields the value 120.

```
ap (Y (abs "f"
      (abs "x" (If (eq (var "x") (int 0))
                   (int 1)
                   (mul (var "x")
                        (ap (var "f") (sub (var "x") (int 1))))))))
   (lit 5)
```

In this example, we use the types of the functions in the shallow embedded DSL themselves instead of some phantom type that carries user-provided additional information. In contrast to the approach of the previous section, the types of the DSL themselves are checked by the compiler of the host language. This guarantees that runtime type errors can not occur. Our variable store guarantees that all variables used are defined and well-typed. The worst that can happen is that a variable is only defined in the state, but not introduced properly in the DSL expression. The state still provides a value of the correct type when this would occur.

This example shows that we can make typed editors for a Turing complete DSL with our dynamic editors. We like to stress again that this example is chosen to demonstrate the power of the approach. We expect that most DSL handle by actual applications are more domain-specific and less complicated.

5 DYNAMIC EDITORS FOR GENERALIZED ALGEBRAIC DATA-TYPES

Above we have demonstrated that one can use dynamic editors to impose type restrictions on a deep embedded DSL to obtain only properly typed instanced and to that dynamic editors can be used to create well-typed instances of a shallow embedded DSL. In this section, we will review the possibility to improve the design of the deep embedded DSL in such a way that the datatype enforces well typed expressions. For this purpose, we use a version of GADT's based on bimaps [2, 6]. A bimap indicates the transformation between two datatypes in both directions. This approach has the advantage that there is no extension of the Hindly-Milner type-system needed. The drawback is that we have to indicate the desired type equalities explicitly in bimaps.

In this paper, we use a record `BM` for these bimaps. We need only two transformation functions from `a` to `b` and from `b` to `a`. In more complex situations we need transformations of other kinds, e.g., `tab::∀ t:(t a)→t b`. For our application we only need the instance `bm` of this type. It tells the compiler that the types `a` and `b` are equal.

```
:: BM a b = {ab :: a → b, ba :: b → a}

bm :: BM a a
bm = {ab = id, ba = id}
```

The simple expression type from Section 2 is extended with a type variable as argument that mimics the result type. The type becomes:

```
:: Expr a
    =      Lit             a
    |      Add  (BM a Int)  (Expr Int)  (Expr Int)
    |      And  (BM a Bool) (Expr Bool) (Expr Bool)
    | ∃ b: Eq   (BM a Bool) (Expr b)    (Expr b)        & type b
    |      If               (Expr Bool) (Expr a)  (Expr a)

class type a | toString2, iTask a
```

The `Lit a` replaces `Int` and `Bool`. The type variable `a` nicely indicates the appropriate result type here. For the integer addition, `Add`, the arguments must be expressions representing an integer value. Hence, their type is `Expr Int`. The additional argument `BM a Int` indicates that the result is also an integer by binding `a` to `Int`. This additional argument is the first argument to allow currying. The logical conjunction `And` is very similar, we just replaced `Int` by `Bool`. For the overloaded equality, `eq`, we need a new type variable for the type of elements to compare. In a datatype, we need to introduce such a variable explicitly by `∃ b:`, in functions we can introduce such variables implicitly. We use the same variable for both argument expression to enforce that they have the same type. The argument `BM a Bool` indicates again that the result of the equality is a Boolean value. The class restriction `& type b` enforces that any type `b` used here should be a member of the listed class. This class contains just the collection of all type-restrictions we impose on the types used. By putting all restrictions in a separate type class we can add a restriction needed in a new view without toughing existing code [9, 10]. It is necessary to enforce all class constraints here since we have no access to the type variable later on. Finally, the first argument of the conditional, `If`, must be a Boolean expression. The other arguments are expressions of the same type as result type.

The actual expression datatype used in the next example contains more constructors. The introduction of typed variables by `Var String` is the most noticeable.

To evaluate expression we use the same state as in the previous example. Just like in the previous example there are no side effects. This implies that we can pass the state down during evaluation, but the state does not have to be part of the result. There is no reason to pass the state around, neither explicitly nor hidden in a monad. The evaluation is implemented as the function `evalE`. For a variable this function does a dynamic pattern match to the type `a` of the evaluator by `(x::a^)`. For the constructors `Add`, `And` and `Eq` the result type is not `a` but `Int` or `Bool`. We use the function `bm.ba` to convince the type-checker that this is indeed the required type. This explicit type-equality ensures that all expressions are correctly typed.

```
evalE :: (Expr a) State → a | TC a
evalE expr state = case expr of
  Lit x      = x
  Var s      = case getVal s state of Just (x::a^) = x
  Add bm x y = bm.ba (evalE x state +   evalE y state)
  And bm x y = bm.ba (evalE x state &&  evalE y state)
  Eq  bm x y = bm.ba (evalE x state === evalE y state)  // generic eq
  If  c t e  = if (evalE c state) (evalE t state) (evalE e state)
```

Other views of this DSL, like transforming expressions to strings, do not need the bimaps at all. For example, the instance of the class to transform an expression to a string is:

```
instance toString2 (Expr a) | toString2 a where
 toString2 expr = case expr of
  Lit x = toString2 x
  Var s     = "(Var " +++ s +++ ")"
  Add _ x y = "(Add " +++ toString2 x +++ " " +++ toString2 y +++ ")"
  And _ x y = "(And " +++ toString2 x +++ " " +++ toString2 y +++ ")"
  Eq  _ x y = "(Eq "  +++ toString2 x +++ " " +++ toString2 y +++ ")"
  If c t e  =
   "(If " +++ toString2 c +++ toString2 t +++ toString2 e +++ ")"
```

In contrast to the expression created by the editor in Section 3.3, correctly typed expressions in the DSL are not a property enforced by a smart editor, but by the type-system of our host language. The only place where the evaluator can fail is that a used variable does not occur in the state, or has another type in the state. We reuse the approach from the lambda-calculus editor of Section 4.2 to ensure that variables do exist with the desired type.

It is not possible to derive a web-editor for the type `Expr a`. The generics used in such a derivation cannot handle the functions in the record `BM a b`. Moreover, the actual type introduced by `∃ b:` is not known, hence the generic system cannot make a generic representation of the type `b`. Finally, the generic system cannot handle class restrictions on variables like `toString2, iTask b`. For those reasons, we have to construct a dynamic editor for `Expr a`.

5.1 A Type for Tasks Expressions

To demonstrate the power of our dynamic editors we make editors over task expressions. They cover an important subset of the iTask system. The type `Expr a` is used for ordinary expressions in those task expressions, like the argument of a `Rtrn`.

The type for task expressions follows the same pattern as the `Expr a`. Variables are explicitly introduced by existential quantifiers. All restrictions needed in the views are collected in the class `type`.

```
:: TaskExpr a
    = Rtrn (Expr a)
    | IF (Expr Bool) (TaskExpr a) (TaskExpr a)
    | EnterInfo (BM a a) String
    | UpdateInfo String (Expr a)
    | ViewInfo String (Expr a)
    | UpdateSharedInfo String String
    | ViewSharedInfo String String
    | ∃ b: Bind (TaskExpr b) String (TaskExpr a) & type b
    | ∃ b: Seq  (TaskExpr b) (TaskExpr a) & type b
    | ∃ b.c: Both (BM a (b,c)) (TaskExpr b) (TaskExpr c)
            & type b & type c
    | One (TaskExpr a) (TaskExpr a)
    | Select [Button a]
    | ∃ b: All (BM a [b]) [TaskExpr b] & type b
    | Forever (TaskExpr a)
    | Get String
    | Set (Expr a) String
:: Button a = Button String (TaskExpr a)
```

We will briefly discuss these constructors.

`Rtrn` lifts the argument from a plain expression to a task expression.
`IF` the conditional choice of tasks.
`EnterInfo` creates an enter information editor for the user. The string specifies a hint given to the user. The bimap is needed here to fix the type of the value to enter.
`UpdateInfo` creates an update information editor. Here the type is fixed by the value to be updated.
`ViewInfo` shows the given value to the user. In contrast with `UpdateInfo` the user cannot change this value.
`UpdateSharedInfo` an update for a shared data source.
`ViewSharedInfo` shows the value of a shared data source to the user.
`Bind` the monadic bind. The string denotes the name of the variable used to bind the value of the first task.
`Seq` the monadic sequence of two tasks.
`Both` execute both tasks in parallel. The combination has a result when both tasks have a result.
`One` return the value of the first task that is finished. When both task finishes on the same event the result of the first task is returned.
`Select` the user is shown all buttons labelled by the string that is argument of `Button`. When the user selects the button the corresponding task is executed.
`All` executes all tasks is the list. The construct terminates when all task are terminated by a list with the values of all tasks.
`Forever` repeats the given argument task forever.
`Get` returns the current value of the given shared data source.
`Set` updates the value of the shared data source.

These operations all have an equivalent in the iTask system. In the selection of elements from the iTask system to include in this DSL we have focussed on selecting illustrative features. This is by no means a complete coverage of the iTask system. That is not the intention of this example, nor of the dynamic editors introduced in this paper.

5.2 Types in our DSL

The amount of overloading in our previous DSLs was limited due to the small number of datatypes handled by those languages. The DSL developed here works for any datatype that implements the class `type`. We use the same approach to offer variables as shown in Section 4.1. There are two minor differences. In order to make type identifiers we use the data type `VarVal` instead of `IdType`. It contains integers, Booleans, a record type for persons[2], tuples and list over types. The record `Bind` is extended with a Boolean valued field `share` to indicate if this identifier is a SDS or a plain variable.

```
:: VarVal = Int | Bool | Person | Pair VarVal VarVal | List VarVal
```

5.3 A Dynamic Editor for Expressions and iTasks

Due to the occurrence of functions, existential quantified variables and class restrictions in the types `Expr a` and `TaskExpr a` editors in the iTask system cannot be derived. In this section, we will outline how a dynamic editor for these types looks.

The first thing to notice is that there is only one dynamic editor despite the fact that there are different types involved, e.g. `Expr`, `TaskExpr`, `Int`, `Bool`, `Person`, `Button` etc. For the programmer, this implies that there is a limited separation of concerns. Fortunately, the dynamic editor is a list of items. Whenever desired we can append lists to compose one big editor from smaller ones and to reuse code. For the user of the dynamic editor, this is not much of an issue. The dynamic editor will only display the elements that produce an element of the desired type at runtime.

[2]Record Type `Person` is just added to show that we are not limited to primitive types.

5.3.1 Fixed Types. A small number of constructs in our DSL have a fixed type of arguments and results. These cases are handled by fully specified dynamic types in the editor items. Some typical examples are:

```
, (dynamic λx y → (Add bm x y) :: (Expr Int) (Expr Int)→Expr Int)
, (dynamic λx y → (And bm x y) // type is derived
```

5.3.2 Overloaded Constructs. A number of constructs in the DSL is fully overloaded, i.e., there are no class restrictions on the type variables. Defining dynamic editor entries for those constructs requires the introduction of a new type variable in the type of the dynamic. Apart from this type introduction in the dynamic, the items follow the scheme of the fixed type cases.

```
, (dynamic λe      → (Rtrn e) ::
                     ∀ b: (Expr b) → TaskExpr b)
, (dynamic λx y    → One x y ::
                     ∀ b: (TaskExpr b) (TaskExpr b) → TaskExpr b)
, (dynamic λs      → EnterInfo bm s :: ∀ b: String → TaskExpr b)
, (dynamic λi      → (Lit i) :: ∀ b: b → Expr b)
, (dynamic λc t e → (If c t e) ::
                     ∀ b: (Expr Bool) (Expr b) (Expr b) → (Expr b))
```

5.3.3 Class Restrictions. The most challenging part of creating dynamic editors is the correct handling of the type class constraints for the existentially quantified variables. A typical example is `∃ b: Eq (BM a Bool) (Expr b) (Expr b) & type b`. When using dynamic editors those existentially quantified variables carry over to type class restrictions in the dynamic functions. Like any type-class restriction (that is not solved at compile time), those restrictions are transformed to an additional, first, function argument. This dictionary argument contains the appropriate functions for the actual type. This implementation of class restrictions is normally completely invisible. However, if we make dynamics with class restrictions those dictionaries become visible. The dynamic editor will ask the user of the program to provide such a dictionary. Alas, these dictionaries are no ordinary object in Clean and cannot be provided in the editor by the user.

We see two solutions to this problem. The easiest solution is to solve the overloading by specifying a specific instance of the class. The other way is to use an other way to provide the appropriate types and associated dictionaries.

5.3.4 Solving Overloading for Class Restrictions. The easiest solution to handle the problem with dictionaries is to ensure that they are not needed. When the Clean compiler is able to determine the types in an application of a function with overloading, the compiler will replace the functions from the class with the appropriate instances or provide the appropriate dictionary.

For the example `∃ b: Eq (BM a Bool) (Expr b) (Expr b) & type b` we can achieve this by using the following dynamic editor instances.

```
[functionConsDyn "Eq.Int" "equal int"
 (dynamic λx y → Eq bm x y :: (Expr Int) (Expr Int)→Expr Bool)
```

```
,functionConsDyn "Eq.Bool" "equal bool"
  (dynamic λx y → Eq bm x y :: (Expr Bool) (Expr Bool)→Expr Bool)
```

For the `All` construct from the task expressions we can use the same approach.

```
,listConsDyn "All" "all tasks"
  (dynamic (id, λlist → All bm list) ::
  ((TaskExpr Int) → TaskExpr Int, [TaskExpr Int] → TaskExpr [Int]))
```

This approach works fine if there are a small number of relevant types. Otherwise, we end up with many editor entries for the same operator. There are infinitely many types introduced by the construct in Section 5.2. Hence, it is impossible to list them all. In practices, a relatively small number of types is usually sufficient.

5.3.5 Providing Dictionaries. The other way to provide a dictionary is by applying a function that delivers the desired dictionary. We illustrate this by a small example. Function `f1` requires the class constraint `toString a` to ensure that the function `toString` exists for the argument type `a`. We have argued that such a class constraint becomes a dictionary in our dynamic editor and hence an (unwanted) additional argument in the editor. The datatype `Dict` contains just the identity function for a type `a` that is an instance of the class `toString`[3]. The function `intDict` yields an instance for the type `Int`. In function `f2` we use the user defined type `Dict` to ensure that there is an instance of toString for the type `a`. The `Start` rule contains an application of this construct.

```
f1 :: a → String | toString a    // Class constraint required
f1 x = toString x

:: Dict a = Dict (a→a) & toString a

intDict :: Dict Int
intDict = Dict id

f2 :: (Dict a) a → String        // Note: no explicit class constraint
f2 (Dict f) x = toString (f x)

Start = f2 intDict 42
```

In our dynamic editors we use the same idea. We need the class constraint `type` for `a`, `Expr a` and `TaskExpr a`. Hence the type for explicit Types, `ET` is slightly bigger.

```
:: ET a =
  ET (a→a) ((Expr a)→Expr a) ((TaskExpr a)→TaskExpr a) & type a
```

We use the dynamic editors to create the required instances for arbitrary nested types described by `VarVal` from Section 5.2.

```
, (dynamic ET id id id :: ET Int)
, (dynamic ET id id id :: ET  Bool)
, (dynamic ET id id id :: ET Person)
, (dynamic λ(ET f _ _) (ET g _ _) → ET (λ(b,c)→(f b,g c)) id id
         :: ∀ b c: (ET b) (ET c) → ET (b,c))
, (dynamic λ(ET f1 f2 f3) → ET (λx → map f1 x) id id
         :: ∀ b: (ET b) → ET [b])
```

[3]In simple applications `Dict` does not need the identity function argument, a phantom type will do. The function is needed for proper binding when there are multiple type variables.

Using these explicit type constructs to handle class restriction implies that the user of the editor has to indicate the type, even if there is only a single option. Fortunately, the dynamic editor will only show this single option and hence type selection is easy for the user.

Some typical examples of the use of these explicit type selections are:

```
, (dynamic λ(ET _ f _) x y → Eq bm (f x) (f y) ::
         ∀ b: (ET b) (Expr b) (Expr b) → Expr Bool)
, (dynamic λ(ET _ _ f) x (ET _ _ g) y → Both bm (f x) (g y) ::
         ∀ b c: (ET b) (TaskExpr b) (ET c) (TaskExpr c)
             → TaskExpr (b,c))
, (dynamic λ(ET f _ _) pair → Fst (fixFst f pair) ::
         ∀ b c: (ET c) (Expr (b,c)) → Expr b)
```

5.3.6 Names for Identifiers and Shares. The final issue is the selection of names from the state maintained in the dynamic editor. The state contains two kinds of names: plain identifiers for the DSL and the names of SDSs in the DSL. The Boolean field in the record `Bind` distinguishes these categories of names. For instance:

```
:: Share a = Share String

toShare :: (Bind Dynamic) → DynamicCons
toShare {idnt,val} = case val of
  (x::t) = functionConsDyn ("Share." +++ idnt) idnt
                           (dynamic (Share idnt) :: Share t)
```

The appropriate shares in the dynamic editor are created by:

```
map toShare (filter (λvar → var.share) state)
```

The application of using identifiers and shares is illustrated in the next editor items:

```
, (dynamic λ(ET _ _ f) x (Name name) y → Bind (f x) name y ::
  ∀ b c: (ET c) (TaskExpr c) (Name c) (TaskExpr b) → TaskExpr b)
, (dynamic λ(Share s)    → Get s    :: ∀ b: (Share b) → TaskExpr b)
, (dynamic λ(Share s) e → Set e s ::
  ∀ b: (Share b) (Expr b) → TaskExpr b)
```

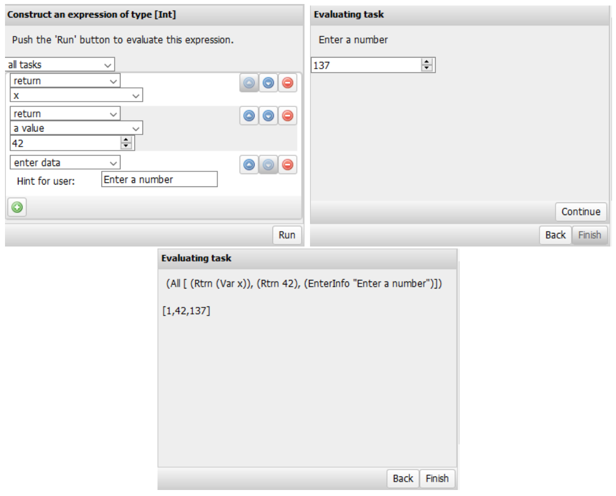
5.3.7 Using the DSL. We used this DSL in an iTask program where the user can interactively define identifiers and shares in as well as a task expression of the chosen type. When the editor contains a complete task expression it can be transformed to a plain iTask expression that is executed in the same simulator by pressing the `Run` button. The user can always return to the editor by using the `back` button. This is illustrated by the screenshots in Figure 4.

To execute the DSL-expression it is transformed from the internal editor representation to an expression of type `TaskExpr` by `valueCorrespondingTo (exprEditor state) value`. Next we create all SDSs in `state` and evaluate the task expression with a function similar to `evalE :: (Expr a) State → a`.

6 RELATED AND FUTURE WORK

There is a plethora of ways to create web-pages is various programming languages these days. See [7] for an up-to-date overview of Haskell based systems. Yesod [19], Happstack [20] and Servant [13] aim to make type safe web-pages. They all specify web-pages by defining their elements and handler while we specify DSL constructs and generate the web-pages.

The low-code approach aims to develop complete applications interactively [22]. This name was coined by Richardson [18]. Gartner


Construct an expression of type [Int]

Push the 'Run' button to evaluate this expression.

all tasks
return
x
return
a value
42
enter data
Hint for user: Enter a number

Evaluating task

Enter a number

137

Evaluating task

(All [(Rtrn (Var x)), (Rtrn 42), (EnterInfo "Enter a number")])

[1,42,137]

Figure 4: Screenshots of the task editor and simulator.

calls this the Magic Square [21]. Although the aims and techniques of these approaches have similarities with our goals, there is also an important difference. We want to provide structured input for a running program instead of creating those programs in such a way.

Once the iTask editors are able to pass a state around we will use this to replace the global state of identifiers with a dynamically maintained set of identifiers. We are looking for even better ways to handle class restrictions in the DSL.

7 DISCUSSION

Using algebraic datatypes for structured input has many advantages. The datatype can match the syntax of the input and the iTask system can derive structured web-editors for those datatypes. This enables end-user to created well-typed inputs while the programmer only has to specify the algebraic datatypes matching the grammar. Most end-users highly prefer those structured editors over free-text editors.

For more complex inputs, the syntax of the input mimicked by the datatype is not enough to ensure correctness. There are also semantic restrictions on the input that are not captured by the ADT. A type-checker over the ADT detects the problems too late, after the end-user has created the input, and is typically rather complicated to make.

In this paper, we show how the new dynamic editor library of the iTask system uses the type checks of the native dynamics to create a structured editor that obeys the type constraints. The dynamic editor selects dynamically which items match the required type and shows those to the user. The system is used recursively for the arguments of the construct selected by the user. We show how one can use these dynamic editors in three different ways. First, we use a phantom type to enforce type-correctness on ordinary datatypes. Next, we used the function types of a shallow embedded DSL directly in the dynamic editor to enforce the required type constraints. Finally, the type-parameters of a GADT based deep embedded DSL can be used to enforce well-typed DSL-expressions.

Class constraints for overloaded functions appear to be the trickiest part of these editors. This is due to the implementation of these overloaded functions with class restrictions; the actual functions of the instance of the class are passed as an additional dictionary argument to the functions. The dynamics reveal this argument but do not provide a way to make those dictionaries. We demonstrated an elegant way to provide these dictionaries by an additional dynamic argument that let the end-user chose the actual type required in the application. In this way, we have to provide just one dynamic for each overloaded construct in our input DSL.

By limiting the identifiers in the structured editor to a pool of dynamically created typed values we can even prevent that the end-user selects undefined or ill-typed identifiers. This is more powerful than a plain GADT can ensure.

REFERENCES

[1] P.M. Achten. 2007. Clean for Haskell98 Programmers - A Quick Reference Guide -. (July 13 2007). http://www.mbsd.cs.ru.nl/publications/papers/2007/achp2007-CleanHaskellQuickGuide.pdf
[2] Peter Achten, Artem Alimarine, and Marinus J. Plasmeijer. 2002. When Generic Functions Use Dynamic Values. In *Implementation of Functional Languages, IFL 2002, Madrid, Spain (LNCS)*, Ricardo Pena and Thomas Arts (Eds.), Vol. 2670. Springer, 17–33. https://doi.org/10.1007/3-540-44854-3_2
[3] Peter Achten, Pieter Koopman, and Rinus Plasmeijer. 2015. An Introduction to Task Oriented Programming. In *Central European Functional Programming School: CEFP 2013*, Viktória Zsók, Zoltán Horváth, and Lehel Csató (Eds.). Springer, 187–245. https://doi.org/10.1007/978-3-319-15940-9_5
[4] Artem Alimarine and Rinus Plasmeijer. 2002. A Generic Programming Extension for Clean. In *Implementation of Functional Languages*, Thomas Arts and Markus Mohnen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 168–185.
[5] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *J. Funct. Program.* 19, 5 (Sept. 2009), 509–543. http://dx.doi.org/10.1017/S0956796809007205
[6] James Cheney and Ralf Hinze. 2004. A Lightweight Implementation of Generics and Dynamics. *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop* (06 2004).
[7] HaskellWiki. 2019. Applications and libraries/GUI libraries — HaskellWiki. (2019). https://wiki.haskell.org/index.php?title=Applications_and_libraries/GUI_libraries&oldid=63014 [accessed 6-April-2020].
[8] Kristian Hentschel, Dejice Jacob, Jeremy Singer, and Matthew Chalmers. Supersensors: Raspberry Pi Devices for Smart Campus Infrastructure. In *4th International Conference on Future Internet of Things and Cloud, FiCloud 2016*, Muhammad Younas, Irfan Awan, and Winston Seah (Eds.). IEEE, 58–62.
[9] John Hughes. 1999. Restricted Data Types in Haskell. In *Proceedings of the 1999 Haskell Workshop*.
[10] Will Jones, Tony Field, and Tristan Allwood. 2012. Deconstraining DSLs. (2012). https://doi.org/10.1145/2364527.2364571
[11] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*. ACM, New York, NY, USA, 11. https://doi.org/10.1145/3183895.3183902
[12] Pieter Koopman and Rinus Plasmeijer. A Shallow Embedded Type Safe Extendable DSL for the Arduino. In *Revised Selected Papers of the 16th International Symposium on Trends in Functional Programming - Volume 9547 (TFP 2015)*. Springer-Verlag, Berlin, Heidelberg, 104âĂŞ123.
[13] Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. 2015. Type-Level Web APIs with Servant: An Exercise in Domain-Specific Generic Programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*. ACM, 1âĂŞ12. https://doi.org/10.1145/2808098.2808099
[14] Hanne Riis Nielson and Flemming Nielson. 1992. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., USA.
[15] Marco Pil. 1998. Dynamic Types and Type Dependent Functions. In *Selected Papers from the 10th International Workshop on 10th International Workshop (IFL âĂŹ98)*. Springer-Verlag, Berlin, Heidelberg, 169âĂŞ185.
[16] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-oriented programming in a pure functional language. In *Proceedings of the 14th PPDP symposium*. ACM, 195–206. https://doi.org/10.1145/2370776.2370801
[17] Rinus Plasmeijer and Marko van Eekelen. 2012. Clean language report. (2012). https://clean.cs.ru.nl/Documentation
[18] Clay Richardson and John R Rymer. 2014. New Development Platforms Emerge For Customer-Facing Applications. (2014). www.forrester.com

[19] Michael Snoyman. 2015. *Developing Web Apps with Haskell and Yesod, 2nd Edition*. O'Reilly Media.
[20] Happstack team. Happstack. (????). happstack.com [accessed 6-April-2020].
[21] Paul Vincent, Kimihiko Lijjima, Mark Driver, Jason Wong, and Yefim Natis. 2019. Magic Quadrant for Enterprise Low-Code Application Platforms. (2019). www.gartner.com
[22] Wikipedia contributors. 2020. Low-code development platform — Wikipedia. (2020). https://en.wikipedia.org/w/index.php?title=Low-code_development_platform&oldid=944262991 [accessed 14-March-2020].

Asymmetric Composable Web Editors in iTasks

Bas Lijnse
Radboud University
b.lijnse@cs.ru.nl

Rinus Plasmeijer
Radboud University
rinus@cs.ru.nl

Abstract

Generic web-based editors have been an integral feature of the iTask Framework since its conception, and even predate it in the form of the iData library. The availability of generic editors is useful for prototyping, but as applications mature, the need for increased control over editor behaviour arises. This can be accomplished by creating customised editors. Unfortunately defining custom editors is no trivial task. The interface for composing editors is useful for common cases, but is too abstract to enable the creation of arbitrary editors. The low-level interface for creating editors from scratch is sufficiently powerful, but exposes many implementation details which makes it complicated to use. In this paper we present a new interace and composition API for editors in iTasks. This new approach is based on an asymmetric typed interface for editors with separate type parameters for data that is consumed and data that is produced by the web editors. We demonstrate the new possibilities by reconstructing a previously builtin editor as a composition of simpler editors and various other examples.

ACM Reference Format:
Bas Lijnse and Rinus Plasmeijer. 2020. Asymmetric Composable Web Editors in iTasks. In *Proceedings of IFL 2020: Symposium on Implementation and Application of Functional Languages (IFL 2020).* ACM, New York, NY, USA, 1 page. https://doi.org/??

A subtyping system for Erlang

Sven-Olof Nyström
Department of Information Technology
Uppsala University
Sweden
svenolof@it.uu.se

Abstract

We present a type system for Erlang based on subtyping. Subtyping systems can reason about types that are more general, for example the universal type that can represent any value.

We present a new theoretical approach which offers a bridge between theory and practical implementation. At the core of the implementation is a propagation algorithm that is very close to algorithms already in the literature. What is new are the theoretical underpinnings which serve as a guide when extending the type system.

Using this approach, we have developed a subtyping system for Erlang. The implementation checks ordinary Erlang programs (though naturally not all Erlang programs can be typed, and sometimes it is necessary to add specifications of functions and data types). We describe the implementation of the type checker and give performance measurements.

1 Introduction

This paper presents a static type system for Erlang, a functional programming language with dynamic typing. The type system is designed with HM as a starting point, but relies on subtyping to provide a greater flexibility. The type system is *safe*; programs that type should be free from type errors at run-time.

In functional programming, the standard approach to static typing is to use Hindley-Milner type inference [15]. Hindley-Milner type inference (HM) traces its roots to simply typed lambda calculus and has many strong points–it is quite simple, is easy to implement, allows parametric polymorphism and is fast in practice. It is used by many functional programming languages (for example SML and Haskell). However, HM has some important limitations. One is that a recursive data type must be defined using constructors that are specific to that data type. Thus, one constructor cannot be a part of more than one data type.

It has been noted by many authors that the limitations of HM could be overcome by allowing subtyping. This will allow, for example, data types that overlap and the universal type that can represent any value. However, in practice subtyping systems either tend to be limited and lacking in features, or conversely, to be complex and difficult to extend.

Our framework intend to offer an flexible and extensible approach to subtyping. To type Erlang, the framework has been extended with features that allow it to reason about a rich domain of data types. The type system does not require types to be explicitly defined and is able to infer complex types from usage in a program.

Subtyping is structural, thus (for example) a type may be a subtype of another type where a constructor has been added. The type system is quite general and permits many types that could not be expressed in a HM type system. Some are probably not useful, for example a type which only allows lists of even length, or a type which builds lists backwards. Others are clearly useful and practical, for example the universal type which is the supertype of every other type and subtyping between recursive data types.

One important difference from earlier appraches is how programs are typed. When the subtyping system types a program, it generates, like other subtyping systems, a set of constraints. The constraints capture the problem of typing the program–thus the program is typable if and only if there is a solution to the constraint system. We show that to determine whether a solution exists, it is sufficient to check the consistency of the constraint system–no assumptions needs to be made about the domain of types, and we show how to do this checking.

We describe the implementation of the type checker and give performance measurements. The implementation checks ordinary Erlang programs (though naturally not all Erlang programs can be typed, and sometimes it is necessary to add specifications of functions and type declarations). The implementation is in Erlang and can itself be typed.

We have two reasons for choosing Erlang: the language is dynamically typed, thus the run-time system is already adopted to a richer range of values and there are many programs that take advantage of the flexibility. Second, Erlang does not allow side effects that modify data structures. This simplifies the design of the subtyping system.

The algorithm for type inference does not extract type information for function definitions in a human-readable format, instead the checker compares function definitions to specifications and conversely, that functions are used according to specification.

Our main contributions are a new way of designing an extensible type system and a type checker for Erlang based on this methodology. If we use the methodology to design a very simple type system, we end up with a type checker that is not that different from what is already in the literature. However, for more complex type systems that reflect the features of modern programming languages, our approach allows a systematic way of introducing more features, making sure that the type system agrees with the operational semantics and that the implementation of the type checker will succeed exactly when the program types.

The rest of this paper is organised as follows. In Section 2 we give an overview of our approach by defining a subtyping system for a simple formal language based on lambda calculus. We describe a simple type checker for this language, discuss how the type system can be extended to manage more powerful constraint languages and link the type system to the semantics of the programming language through the subject reduction property. In Section 3 we extend the subtyping system with constructors, filters (that allow a form of discriminated unions) and conversion of constructors. In Section 4 we discuss the problem of adapting a static type system to Erlang

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and describe how type declarations and function specifications can be added to Erlang code. Section 5 describes the implementation. Section 6 presents our experimental evaluation and Section 7 places our results in the context of earlier work.

2 How to build a type system

Let's start with an overview of the approach. We first develop a simple type checker for a simple functional language. The algorithm for type checking is straight-forward, but an important question we also try to answer is: How can the type checker guarantee safety?

2.1 What is a type?

It is possible to give the domain of types a simple inductive definition based on the syntactic form of type expressions. Inductive definitions are familiar to any computer scientist and have well-understood properties.

However, sometimes inductively defined type domains are not what we want. The most obvious limitation is that a domain of this kind lacks solutions to circular equations such as $X = (t \to X)$. Now, there have been many attempts to overcome these difficulties by extending the set of types to include recursive types, see for example [2], but we then lose the simplicity of inductively defined types.

A definition by Barendregt [3] suggests a different approach:

Definition (Barendregt 11A.1)**.** *A* type structure *is of the form* $S = \langle |S|, \leqslant, \to \rangle$, *where* $\langle |S|, \leqslant \rangle$ *is a poset and* $\to : |S|^2 \to |S|$ *is a binary operation such that for a, b, a′, b′* $\in |S|$ *one has*

$$a' \leqslant a \;\&\; b \leqslant b' \Rightarrow (a \to b) \leqslant (a' \to b').$$

The structure intended is an *algebraic structure*, i.e., a set of elements with some operations on the set, where the operations should satisfy some algebraic properties. (A poset is a partial order, i.e., the relation $\leqslant$ is transitive, anti-symmetric and reflexive.) The interesting thing is that elements of a type structure do not need to look like type expressions. The properties of Barendregt's type structures are the ones typically seen in subtyping systems; the $\leqslant$ relation defines a partial order, and the arrow operation (which takes two types and builds a new type) satisfies the rule

$$\frac{a' \leqslant a \quad b \leqslant b'}{(a \to b) \leqslant (a' \to b')}$$

In other words, this definition says that any combination of a set $|S|$ with an ordering $\leqslant$ and a binary operation $\to$ over $|S|$ that satisfies the properties is a type structure.

An inductive definition of types (together with some appropriate definition of $\leqslant$) satisfies the definition of type structures. However, there are other interesting type structures, for example type structures with infinite types.

Now, it would be useful to show that it is possible to type the program using *some* type structure, even if we did not know precisely *which* type structure. This is the approach taken in this paper. But first we need to refine the definition of type structures.

For our purposes, the axioms in Barendregt's definition are insufficient as there would not be any program that did *not* type. This would in turn make the problem of type checking rather uninteresting. To remedy this we introduce a set of *atomic types*, types that are distinct from each other and from function types.

Let T, U range over types. Assume a set Atom of atoms, and let A, B range over types associated with these values. Also require that if two function types $T \to U$ and $T' \to U'$ are related, i.e., if $(T \to U) \leqslant (T' \to U')$, then it holds that $T' \leqslant T$ and $U \leqslant U'$. Barendregt calls type structures that satisfy this property *invertible* but we will assume that all type structures satisfy this property.

Definition 2.1. A *type structure S* is an algebraic structure of the form

$$S = \langle |S|, \leqslant, \to, \mathsf{Atom}, \mathsf{a} \rangle$$

such that

1. the relation $\leqslant \subseteq |S| \times |S|$ is transitive and reflexive,
2. $(\to) : |S| \times |S| \to |S|$ is a binary operation where for types $T, T', U, U' \in S$ we have $(T \to U) \leqslant (T' \to U')$ iff $T' \leqslant T$ and $U \leqslant U'$,
3. Atom is some set,
4. $a : \mathsf{Atom} \to |S|$,
5. for $A, B \in \mathsf{Atom}$, $A \neq B$, it never holds that $a(A) \leqslant a(B)$, and
6. for $A \in \mathsf{Atom}$ and types T, U it never holds that $a(A) \leqslant (T \to U)$ or $(T \to U) \leqslant a(A)$.

2.2 Simple constraints

Let $X, Y \in \mathsf{TVar}$ be the set of *type variables.* Also let $A, B \in \mathsf{Atom}$ be the set of *atomic types.* Let the set of *type expressions* $t, u, v, w \in \mathsf{TExp}$ be defined as follows:

1. $\mathsf{TVar} \subseteq \mathsf{TExp}$,
2. $A \in \mathsf{TExp}$, if $A \in \mathsf{Atom}$.
3. $(t_1 \to t_2) \in \mathsf{TExp}$, if $t_1, t_2 \in \mathsf{TExp}$,

Let the set of *constraints* $\varphi \in \mathsf{Constraint}$ be formulas of the following forms (where $\bot$ is the inconsistent constraint):

1. $t_1 \leqslant t_2$, for $t_1, t_2 \in \mathsf{TExp}$
2. $\bot$

A *constraint system G* is a finite set of constraints.

We express the properties of type structures as derivation rules for constraints; that the $\leqslant$ relation is reflexive and transitive, properties of the $\to$ operator, and things that must *not* occur, for example $A \leqslant (t \to u)$ for some atomic type A and arbitrary types t and u.

To describe situations which must not occur, we use $\bot$ to indicate inconsistency, for example in rule AW. Now, it should be easy to see that the derivation rules for constraints of Figure 1 correspond exactly to the axioms given for type structures in Definition 2.1.

We say that a constraint system G is *consistent* if $G \vdash \bot$ does not hold, and, conversely, that G inconsistent if $G \vdash \bot$ holds. Naturally we are only interested in consistent constraint systems. We would not expect to find a solution for a constraint system containing, say, a constraint $A \leqslant (t \to u)$.

For example, if $G = \{A \leqslant X, X \leqslant B\}$, where A and B are distinct atoms, we have by rule (T) that $G \vdash A \leqslant B$ and by rule (AA) that $G \vdash \bot$, i.e., the constraint system is inconsistent.

2.3 Some mathematical logic

To solve constraint systems, or, more precisely, to determine whether a constraint system can be solved, we will turn to mathematical logic. Mathematical logic is a complex subject and we will only mention some basic definitions and results. We will be brief as details are not important for the rest of the paper. Textbooks on the subject will provide further information, see for example [24].

In first order predicate logic, a *sentence* may be composed of predicate symbols and expressions (as the constraints defined earlier). A sentence may also be composed using the usual connectives ($\wedge$, $\vee$

$$\frac{\phi \in G}{G \vdash \phi}\,(\in)$$

$$\frac{}{G \vdash t \leqslant t}\,(\mathrm{R}) \qquad \frac{G \vdash t \leqslant u, \qquad G \vdash u \leqslant v}{G \vdash t \leqslant v}\,(\mathrm{T})$$

$$\frac{G \vdash t' \leqslant t, \qquad G \vdash u \leqslant u'}{G \vdash (t \to u) \leqslant (t' \to u')}\,(\mathrm{W})$$

$$\frac{G \vdash (t \to u) \leqslant (t' \to u')}{G \vdash t' \leqslant t}\,(\mathrm{WL}) \qquad \frac{G \vdash (t \to u) \leqslant (t' \to u')}{G \vdash u \leqslant u'}\,(\mathrm{WR})$$

$$\frac{G \vdash A \leqslant (t \to u)}{G \vdash \bot}\,(\mathrm{AW}) \qquad \frac{G \vdash (t \to u) \leqslant A}{G \vdash \bot}\,(\mathrm{WA})$$

$$\frac{G \vdash A \leqslant B \qquad A \neq B}{G \vdash \bot}\,(\mathrm{AA})$$

Figure 1. Derivation rules for constraints. The rules define the relation $\vdash$.

and $\neg$) and existential and universal quantifiers. As for constraint systems, we say that a set of sentences is *consistent* if a contradiction cannot be derived.

Each rule of Figure 1 can be expressed as a sentence, for example rule (T) can be stated

$$\forall XYZ.X \leqslant Y \land Y \leqslant Z \implies X \leqslant Z.$$

Since we assume that a constraint system is finite the conjunction of the constraints of a constraint system form a sentence (type variables are mapped to logic variables and all such variables are existentially quantified).[1] Thus, the combination of the derivation rules and a constraint system G forms a set of sentences. It should be clear that if a constraint system is consistent, then the corresponding set of sentences is also consistent.

A *structure* (interpretation) is a set of values with a set of symbols; constants (that map to values), functions (that map to operations on the set) and relation symbols over the set [24, Section 3.2]. It should be easy to see that a type structure is also a structure. A structure is said to be a *model* of a set of sentences if each of the sentences holds in the structure [24, Section 3.4].

In the context of constraint solving, a solution to a constraint system corresponds to a model of the constraint system. If we want to determine whether a constraint system can be solved, but we are not interested in the details of the solution, we can use a result by Henkin, known as the *model existence property* [10], see also [24, Section 4.1]. Henkin shows, by an explicit construction, that for a consistent set of sentences, it is possible to construct a structure that is a model of the set of sentences.

In other words, if a constraint system is consistent, there is some type structure for which it has a solution. In Section 2.7 we describe a straight-forward algorithm for checking the consistency of a constraint system. If the algorithm finds that the constraint system is consistent, the program types.

2.4 Lambda calculus

We first develop a system for subtype inference for lambda calculus. We will later look at variations of lambda calculus extended with important features of Erlang and discuss how they can be typed. Lambda calculus is a simple and efficient formalism. Lambda calculus is close to functional programming, and particularly suited for reasoning about types.

We extend lambda terms to include terms that represent atoms. Given a set of variables $x \in$ Var and a set of atoms $A \in$ Atom, the set of lambda terms is inductively defined as:

$$M ::= x \mid M_1 M_2 \mid \lambda x.M_1 \mid A$$

We will let the variables M, N, P range over lambda terms. A term which is an atom will have the atom as type.

We say that an occurrence of a variable in a lambda term is *free* if it is not "bound" by a lambda. In lambda calculus, terms are considered to be equivalent up to renaming of bound variables, for example, the terms $\lambda x.x$ and $\lambda y.y$ represent the same function (the identity function). Thus a lambda term may have many syntactic representations. We will always assume that the representation of M is chosen such that any free variable is not also bound, and no variable is bound in more than one sub-term.

The semantics of lambda calculus can now be expressed using a single reduction rule:

$$(\lambda x.M)N \longrightarrow_\beta M\,[x := N]\,.$$

A lambda term is said to be a *redex* if it can be on the left hand side in this rule above, i.e., if it is of the form $(\lambda x.M)N$. Clearly, for any redex M there is a lambda term M' such that $M \longrightarrow M'$.

We say that $M \longrightarrow M'$ if there is some sub-term N of M such that $N \longrightarrow_\beta N'$, and M' is the result of replacing one occurrence of N in M with N'. We write $M \twoheadrightarrow N$ if there is a sequence

$$M_1 \longrightarrow M_2 \longrightarrow \ldots \longrightarrow M_n$$

with $M = M_1$ and $N = M_n$.

2.5 Typing lambda calculus

An *environment* Γ is a set $\{x_1 : t_1, \ldots, x_n : t_n\}$ where the x_i are distinct variables, and the t_i are type expressions. For a variable x, an environment should contain at most one binding $x : t$ of x.

A typing is written $\Gamma \Vdash M : t$ and indicates that the lambda term M has the type t in environment Γ. (We use the symbol $\Vdash$ for typings to reduce the risk of confusion between derivations in the constraint system and typings.) If Γ is empty, we will sometimes write $\Vdash M : t$.

As it is often convenient to make the constraints of a typing explicit, we will sometimes write $\Gamma \Vdash M : t\ [G]$ to indicate that the typing $\Gamma \Vdash M : t$ holds, provided that the constraint system G can be solved. Naturally, whenever $\Gamma \Vdash M : t$ there is some constraint system G such that $\Gamma \Vdash M : t\ [G]$. For the reader's convenience we show the rules on this format in Figure 2.

The first three type rules are the standard rules of simply typed lambda calculus (see for example [3, Figure 1.6]). The subsumption rule says simply that any type can be replaced with a more general type [17].

From a practical point of view the subsumption rule poses some difficulties as it can be inserted anywhere in the derivation of a typing. The other rules are associated with different ways of building terms, so that the tree shape of the derivation of a typing for a term is given by the term. Now, since the subtyping relation is reflexive, an application of the subsumption rule may be inserted anywhere in the typing, and since it is transitive, two consequtive

[1]An alternative would be to introduce a constant symbol for each type variable and map each type variable to the corresponding constant symbol.

$$\frac{(x : t) \in \Gamma}{\Gamma \Vdash x : t\ [\emptyset]} \qquad \text{(axiom)}$$

$$\frac{\Gamma \Vdash M : t \to u\ [G_1] \qquad \Gamma \Vdash N : t\ [G_2]}{\Gamma \Vdash MN : u\ [G_1 \cup G_2]} \qquad \text{(application)}$$

$$\frac{\Gamma \cup \{x : t\} \Vdash M : u\ [G]}{\Gamma \Vdash \lambda x.M : t \to u\ [G]} \qquad \text{(abstraction)}$$

$$\frac{}{\Gamma \Vdash A : A\ [\emptyset]} \qquad \text{(atom)}$$

$$\frac{\Gamma \Vdash M : t\ [G]}{\Gamma \Vdash M : u\ [G \cup \{t \leqslant u\}]} \qquad \text{(subsumption)}$$

Figure 2. Typing rules with explicit constraint systems.

$$\begin{aligned}
&w_\Gamma(x) = \langle \emptyset, \Gamma(x) \rangle \\
&w_\Gamma(M_1 M_2) = \langle G_1 \cup G_2 \cup \{t_1 \leqslant (t_2 \to X)\}, X \rangle \\
&\qquad \text{where } X \text{ is a fresh type variable,} \\
&\qquad\qquad \langle G_1, t_1 \rangle = w_\Gamma(M_1), \text{ and} \\
&\qquad\qquad \langle G_2, t_2 \rangle = w_\Gamma(M_2) \\
&w_\Gamma(\lambda x.M_1) = \langle G_1, Y \to t_1 \rangle \\
&\qquad \text{where } Y \text{ is a fresh type variable,} \\
&\qquad\qquad \Gamma_1 = \Gamma \cup [x : Y], \text{ and} \\
&\qquad\qquad \langle G_1, t_1 \rangle = w_{\Gamma_1}(M_1) \\
&w_\Gamma(A) = \langle \emptyset, A \rangle
\end{aligned}$$

Figure 3. Explicit construction of constraint system

applications of the subsumption rule may be combined. Thus there is for any derivation of a typing an equivalent derivation where every other rule is an application of the subsumption rule. In other words, it is always possible to find a derivation of the typing with a shape that is given by the term. This is discussed in more detail by Kozen et al. [12] and Palsberg and O'Keefe [18].

We use this insight in an explicit construction of the constraint system that needs to be solved to type the term. For a type environment Γ and a term M, the function $w_\Gamma(M)$ defined in Figure 3 computes a pair of a constraint system G and a type expression t. The constraint system G has a solution exactly in the cases M can be typed. In other words the constraint system required to type a lambda-term can be constructed by a straight-forward traversal of the term. The term types exactly when the constraint system has a solution.

2.6 Safety

A desirable property of a type system is *safety*. This is usually taken to mean that if a program types, certain errors should not occur at run time. Milner [15] shows that a program that types is "semantically free of type violation", i.e., that "for example, an integer is never added to a truth value or applied to an argument". One way to show this property is via the *subject reduction property*.

The subject reduction property states an invariant for typings; if M is a term that types, that is, $\Gamma \Vdash M : t$, for some environment Γ and type expression t, and M reduces in one or more steps to some other term ($M \twoheadrightarrow N$) then that term will have the same type, $\Gamma \Vdash N : t$. If N is a term that cannot type, for example an application of an arithmetic operation to strings, then the subject reduction property guarantees that no term that types can be reduced to N. (The word "subject" refers to the term M in a typing $\Gamma \Vdash M : t$.)

Lemma 2.2. *If* $M \twoheadrightarrow N$ *and* $\Gamma \Vdash M : t\ [G]$ *then* $\Gamma \Vdash N : t\ [G]$.

The original proof of the subject reduction property for lambda calculus was given by Curry and later extended to subtyping by Mitchell [16, 17]. See also [3, Section 1.2 and 11.1]

2.7 Checking that a program types

A lambda term M types if there is some derivation of the typing $\Vdash M : t\ [G]$, where the constraint system G has a solution. By the model existence property (Section 2.3) it is sufficient to show that the constraint system G is consistent. We will now describe an algorithm for checking consistency of a constraint system.

Definition 2.3. Given a constraint system G, define $(G)_n$, for $n \geq 0$, to be the smallest sets that satisfy the following:

1. $(G)_0 = G$,
2. for all n, $(G)_{n+1} \supseteq (G)_n$,
3. for all even $n > 0$, if the constraint $(t \to u) \leqslant (t' \to u')$ is in $(G)_{n-1}$, then the constraints $t' \leqslant t$ and $u \leqslant u'$ are in $(G)_n$, and
4. for all odd $n > 0$, if the constraints $t \leqslant X$ and $X \leqslant u$ are in $(G)_{n-1}$, then $(t \leqslant u) \in (G)_n$.

Let $G^* = \bigcup_n (G)_n$.

The complexity of constructing G^* can be determined by a simple argument [8]. First, note that G^* only contains type expressions present in G. Thus if the size of G is n, and G contains no more than n expressions, there are less than n^2 inequalities in G^*, which sets a bound to the space used by the construction. When an inequality $t \leqslant u$ is added to the constraint system, the algorithm must examine inequalities of the forms $t' \leqslant t$ and $u \leqslant u'$ (in the *odd* step). This may, at worst, require work proportional to the number of expressions in G, thus the cost of adding one constraint is $O(n)$ and the worst-case complexity of the algorithm is $O(n^3)$.

The definition of G^* might seem unnecessarily restrictive as it would not add to the complexity of computing G^* if Item 4 of the definition was generalised to allow arbitrary expressions instead of a variable. However, it turns out that this seemingly straight-forward change would make the proof of Theorem 2.4 more complicated, in particular Proposition 2.7 would need to be restated.

We say that a constraint system is *locally consistent* if G^* does not contain any immediately inconsistent constraints such as $\bot$, $A \leqslant (t \to U)$, $(t \to U) \leqslant A$, or $A \leqslant B$, for distinct atoms A and B. It turns out that local consistency coincides with consistency.

Theorem 2.4. *A constraint system G is consistent iff G is locally consistent.*

It should be clear that a consistent constraint system is also locally consistent. To show the converse, that a locally consistent constraint system is consistent, we consider the proof rules of Figure 1. The question we ask is: If we can deduce $G \vdash \varphi$ in a single application of one of the rules, how will $(G \cup \{\varphi\})^*$ differ from G^*?

Rules ($\in$), (WL) and (WR) are applied in the computation of G^*, so if $G \vdash \varphi$ can be deduced using one application of one of these rules we have $\varphi \in G^*$. As G is assumed to be locally consistent the rules (AW), (WA) and (AA) can be excluded as their use implies that G is *not* locally consistent.

We next consider the remaining rules (R), (T) and (W) and show that while they add new constraints, local consistency will not change. We state properties of these derivation rules in the following propositions. They can be shown by induction over n.

Proposition 2.5 (Rule R)**.** *Suppose that G is a constraint system, t some type expression and φ an inequality. Let $H = G \cup \{t \leqslant t\}$.*

Whenever $\varphi \in (H)_n$ it holds either that

1. *$\varphi \in (G)_n$, or*
2. *$\varphi = (u \leqslant u)$, some subexpression u of t.*

Proposition 2.6 (Rule T)**.** *Suppose that G is locally consistent and contains the constraints $t \leqslant t'$ and $t' \leqslant t''$. Let $H = G \cup \{t \leqslant t''\}$.*

Whenever a constraint $u \leqslant v$ occurs in $(H)_n$, there are type expressions $w_1, w_2, \ldots, w_m$ and an integer k such that $w_1 = u$, $w_m = v$, and the constraint $w_i \leqslant w_{i+1}$ occurs in $(G)_k$, for $i < m$.

Proposition 2.7 (Rule W)**.** *Let G be a constraint system containing the constraints $t \leqslant t'$ and $u \leqslant u'$. Let $\varphi = ((t' \to u) \leqslant (t \to u'))$ and $H = G \cup \{\varphi\}$. It follows that whenever a constraint ψ occurs in $(H)_n$, we have either*

1. *$\psi \in (G)_n$, or*
2. *$\psi = \varphi$.*

The proof of the theorem uses these propositions to show that a sequence of applications of the derivation rules R, T and W to a locally consistent constraint system always lead to a locally consistent constraint system. In other words, it is not possible to derive $\perp$ from a locally consistent constraint system, thus if a constraint system is locally consistent it is also consistent.

2.8 How to extend the constraint language

In our framework, introducing new forms of type expressions is entirely unproblematic, since without any derivation rules that operate on them, it is not possible to use the new expressions to prove new things. Adding derivation rules is a different matter. A new derivation rule allows us to draw new conclusions, thus it could cause a previously consistent constraint system to become inconsistent. We will consider a simple example; the addition of a universal type. We will show how universal types can be accommodated in our framework.

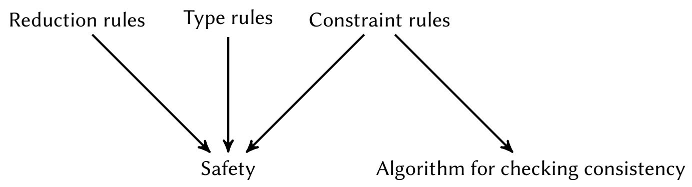
We use the symbol 1 for the type expression that represents the universal type. The additional rules are stated in Figure 4.

Rule (U) states that 1 is the greatest type according to the subtyping order. For any type t, we can conclude that t is a subtype of 1. Rules (UW) and (UA) state that no type given by an atom expression or an arrow expression may be greater than the universal type. More explicitly, if a constraint which states that the universal type is a subtype of (for example) an atomic type is encountered a contradiction can be derived. To handle these rules in our framework, we define constraints of the forms $1 \leqslant (t \to u)$ and $1 \leqslant A$ to be immediately inconsistent. We also need to show that rule (U) preserves local consistency.

Generally speaking our framework can be extended to accommodate new derivation rules, provided that they fall into one of three categories:

1. Rules that describe situations where inconsistency follows from a constraint. Such constraints can be included in the set of immediately inconsistent constraints, provided that it is possible to implement a constant-time test that recognises them. In our example Rules (UW) and (UA) fall into this category,
2. Rules that preserve local consistency as discussed in Section 2.7. Our example has one such rule; Rule (U).
3. Rules that need to be expressed in the computation of G^*. Such rules must not introduce new type expressions (as that could affect complexity and might even cause the computation to loop). We have seen one rule that falls into this category: Rule (W) of Figure 1.

2.9 Workflow in the design of a type system



The dependencies are summarised in the diagram above. If the reduction rules (that describe the operational semantics of the programming language) are modified or extended, the derivation rules (of constraints) need to be sufficiently powerful to show safety properties (in particular, the subject reduction property), thus it may be necessary to introduce new constraint rules. A change in the constraint rules may in turn require a change in the constraint checking algorithm. On one hand the derivation rules need to be sufficiently powerful to guarantee the subject reduction property, on the other hand they must not be so expressive that they cannot be implemented efficiently.

3 The extended lambda calculus

We extend the simple language of Section 2 to accommodate the Erlang programming language. First, Erlang has a rich set of data type constructors (in contrast to the simple language which only has atoms and functions). A type can be described by a set of constructors, each applied to a number of types. An Erlang program, for example:

```
f({leaf, X}) -> ...
f(Y) -> ...
```

can easily distinguish between data built using a particular constructor and data that is *not.* Thus we want to be able to isolate data that does not match a constructor, both in the extended lambda calculus and in the constraint language. In the extended lambda calculus we express this using a special construct, the *open case expression.* The constraint language uses *filters* to separate the part of a type that is built using a particular set of constructors. Filters

$$\frac{}{t \leqslant 1} \quad \text{(U)}$$

$$\frac{1 \leqslant (t \to u)}{\perp} \quad \text{(UW)}$$

$$\frac{1 \leqslant A}{\perp} \quad \text{(UA)}$$

Figure 4. Derivation rules for the universal types.

are also used to reason about discriminated unions. Last, in some cases we need to allow *conversion* between types created using different constructors.

3.1 Constructors

The data types of Erlang include tuples, lists, atoms, integers and floating point values. To make the formalization more uniform, we model all these values and types as *constructors*. In the formal development we assume a set of constructors $c \in C$, where each constructor has an arity. Each argument of a constructor is either covariant or contravariant (the only constructors with contravariant arguments are function types). Constructors form terms in the extended lambda calculus and type expressions in the constraint language, thus constructors build both data and types.

For the representation of function types we reserve a constructor c_λ with arity 2 that does not occur in any term. The first argument of c_λ is contravariant, the second covariant.

We will always assume that a constructor is used with the correct number of arguments, thus we will often omit reference to the arity of the constructor. We will sometimes refer to a term or a type expression of the form $\langle c\ \ldots\rangle$ as a *constructor term* or a *constructor type expression.*

The set of constructors will include constructors for the various data types, for example atoms, lists and tuples.

3.2 Filters and unions

Erlang has a fixed set of constructors that can be used to build recursive data types. This should be contrasted with the situation in programming languages based on Hindley-Milner type checking, where each data type has its own set of constructors.

Consider, for example, the following Haskell data type definition:

```
data Tree = Leaf Integer
          | Branch Tree Tree
```

This type definition introduces the constructors `Leaf` and `Branch` (and they cannot be used to build data structures of any other type). In our system, the corresponding data type might be defined

```
+type tree() = {leaf, integer()}
             + {branch, tree(), tree()}.
```

Here, the data structure uses the tagged tuples `{leaf, ...}` and `{branch, ...}` as constructors. They may of course be used in other parts of the program.

The specification language can express that a type is a union of two types, but there are limitations. Consider an inequality

$$\langle c_1\ \ldots\rangle \cup \langle c_2\ \ldots\rangle \leqslant X,$$

where c_1 and c_2 are distinct constructors. A constraint of this form could occur if one wanted to type an Erlang function that is specified to accept the tree data type as a parameter. Expressing this in the constraint language is easy:

$$\langle c_1\ \ldots\rangle \leqslant X, \langle c_2\ \ldots\rangle \leqslant X.$$

However, sometimes we want to put the union on the right-hand side of the inequality. An inequality of this type would take the form:

$$Y \leqslant \langle c_1\ \ldots\rangle \cup \langle c_2\ \ldots\rangle \tag{1}$$

and could occur (for example) when the type checker verifies that a function does indeed return a tree. As noted by several authors, for example [1], the combination of union types and function types makes the typing problem substantially more difficult. Instead, we present a solution that handles disjoint unions.

Instead of adding union types to our constraint language, we introduce a new form of type expressions which we will call *filters.* A constraint that uses a filter takes the form

$$X \upharpoonright S \leqslant t,$$

where S is a set of constructors, X is a type variable and t is a type expression. Filters may only occur on the left-hand side of an inequality. (Applying filters to other type expressions or allowing filters on the right-hand side of $\leqslant$ would not cause any major difficulties but would complicate the derivation rules and the consistency checking algorithm.)

The idea is that a filter only lets through those subtypes of X that use a constructor which is a member of S. This can be expressed in the following derivation rule:

$$\frac{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant X \qquad G \vdash X \upharpoonright S \leqslant u \qquad c \in S}{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant u} \tag{F}$$

Note that this derivation rule fits Category 3 of Section 2.8 as the algorithm for G^* can be easily extended to capture this rule.

It should be stressed that the meaning of filters is exactly the one given by the derivation rule. The rule states that when a constraint $X \upharpoonright S \leqslant u$ holds, and some type expression $t = \langle c\ \ldots\rangle$, such that $c \in S$, appears on the left hand side of X, i.e., $t \leqslant X$, we can deduce $t \leqslant u$, i.e., the filter will let t pass.

Turning back to our example (1), checking that the type of Y belongs to either of the two type expressions $\langle c_1\ \ldots\rangle$ and $\langle c_2\ \ldots\rangle$ can be expressed with the constraints

$$Y \upharpoonright \{c_1\} \leqslant \langle c_1\ \ldots\rangle \quad \text{and} \quad Y \upharpoonright S \leqslant \langle c_2\ \ldots\rangle$$

where S is the largest set of constructors that does not contain c_1[2]. Thus the first filter will match only those type expressions that use the constructor c_1, but the second filter will match those that do *not* use c_1. (Please recall that we assumed that c_1 and c_2 were distinct.)

Constraints involving filters are typically generated when typing pattern matching and for function specifications and type definitions.

3.3 Open case statements

As mentioned, Erlang makes it easy to write code that performs a case analysis on a data structure depending on whether it belongs to one subtype or not. In the extended lambda calculus we express this mechanism through *open case terms.* These take the form

$$\begin{aligned}&\mathsf{case}(M,\\ &\quad \langle c\ x_1 \ldots x_n\rangle \Rightarrow N,\\ &\quad y \Rightarrow P).\end{aligned}$$

The idea is that if the term M matches the pattern $\langle c\ x_1 \ldots x_n\rangle$, the first branch, the term N, is selected. The second branch is only selected when the term does *not* match the pattern. The syntactic form used here was first considered by Heintze [8] in the context of set-based analysis.

[2]The representation of constraints allows filter expressions where either the set is finite or the set of constructors *not* in the set is finite.

Table 1. Some constructors in the Erlang type system. The variable n ranges over non-negative integers and a over Erlang atoms.

Constructor	Description	Arity
$\mathbf{tuple}^n_a$	tagged tuple	$n-1$
$\mathbf{tuple}^n$	untagged tuple	n
tuple	uniform tuple	1
$\mathbf{atom}_a$	a specific atom	0
atom	any atom	0
any	universal type	0

3.4 Conversion

Since Erlang was not designed as a typed language from the start, the way constructors are used by applications, libraries and built-in primitives sometimes makes it difficult to determine which attributes of a data structure should be thought of as a constructor. For example, while it is clear that each atom should have its own type, there are operations that work on any atom, so one would like a type that represented *all* atoms. We have seen tagged tuples, but sometimes tuples are not tagged (different untagged tuples are only distinguished by their length), thus one would like a type for untagged tuples for each length. Some Erlang primitives treat tuples as arrays, so one would also like a constructor that describes tuples of any length.

In this section we consider the set of constructors required to reason about Erlang programs. We give two conversion relations, both named $\lhd$, over terms and type expressions. These determine when a term (or a type expression) written using one constructor may be converted to some term (or type expression) using another. It is straight-forward to extend the reduction relation for open case expressions to allow conversion of terms. Subtyping can now be defined using the $\lhd$ relation. However, the definition of subtyping does not fit the type checking algorithm as one inequality could be due to the combination of several rules. Thus we give a second, equivalent, definition in a form that fits the type checking algorithm. Finally we consider the interaction between filters and conversion.

3.4.1 Conversion of terms and type expressions. We consider the constructors listed in Table 1. (Typing Erlang requires other constructors, but the ones listed here are the most interesting.) The constructor $\mathbf{tuple}^n_a$ represents a tuple of length n which is tagged with the atom a. This constructor has arity $n-1$, as the first element of the tuple is implicit. For untagged tuples of length n we use the constructor $\mathbf{tuple}^n$ which of course has arity n. The constructor **tuple** (of arity 1) is used when a tuple is uniform, i.e., each element of the tuple has the same type. For an atom a, the nullary constructor $\mathbf{atom}_a$ represents that atom, in other words, the term $\langle \mathbf{atom}_a \rangle$ is that atom. The type expression $\langle \mathbf{atom}_a \rangle$ gives us the type consisting of the atom a. The type expression $\langle \mathbf{atom} \rangle$ gives the type of all atoms, and the type expression $\langle \mathbf{any} \rangle$ the universal type.

Many of these constructors, for example $\mathbf{atom}_a$, for some atom a, and $\mathbf{tuple}^n$, for some integer n, play a role in both at run-time and in the type checker. Others, such as **any**, the constructor for the general type, are only meaningful in the type checker.

3.4.2 Conversion and subtyping. We start by specifying relations $\lhd$ over terms and type expression. We define these relations as the minimal transitive and reflexive relation which satisfies

Figure 5. Conversion over terms

$$\langle \mathbf{tuple}^n_a\ M_2\ \ldots\ M_n \rangle \lhd \langle \mathbf{tuple}^n\ \langle \mathbf{atom}_a \rangle\ M_2\ \ldots\ M_n \rangle$$
$$\langle \mathbf{tuple}^n\ M\ \ldots\ M \rangle \lhd \langle \mathbf{tuple}\ M \rangle$$
$$\langle \mathbf{atom}_a \rangle \lhd \langle \mathbf{atom} \rangle$$
$$M \lhd \langle \mathbf{any} \rangle$$

Figure 6. Conversion over type expressions

$$\langle \mathbf{tuple}^n_a\ t_2\ \ldots\ t_n \rangle \lhd \langle \mathbf{tuple}^n\ \langle \mathbf{atom}_a \rangle\ t_2\ \ldots\ t_n \rangle$$
$$\langle \mathbf{tuple}^n\ t\ \ldots\ t \rangle \lhd \langle \mathbf{tuple}\ t \rangle$$
$$\langle \mathbf{atom}_a \rangle \lhd \langle \mathbf{atom} \rangle$$
$$t \lhd \langle \mathbf{any} \rangle$$

the properties stated in figures 5 and 6, for arbitrary terms M, $M_1, \ldots, M_n$, type expressions $t, t_1, \ldots, t_n$ and atoms a. Note the parallel-ls between the two relations.

The definition of $\lhd$ allows conversions such as

$$\langle \mathbf{tuple}^2_{\mathtt{leaf}}\ t \rangle \lhd \langle \mathbf{tuple}^2\ \langle \mathbf{atom}_{\mathtt{leaf}} \rangle\ t \rangle.$$

In other words, a tagged tuple is also an untagged tuple.

We can now give a subtype rule that allows conversion:

$$\frac{G \vdash t \lhd u}{G \vdash t \leqslant u}\ (\lhd)$$

Keep in mind that these type rules describe the manipulation of type expressions. According to the final rule, we can show that the type given by the type expression t is a subtype of the corresponding type given by u whenever t can be converted to u.

However, this rule is difficult to implement directly. In Section 3.5 we discuss a formulation of this rule which is more suitable for implementation.

3.4.3 Conversion in the extended lambda calculus. In the extended lambda calculus, conversion comes into play in the open case expressions. For example, if the pattern of an open case expression is an untagged tuple, and the term being matched is a tagged tuple, the matching may succeed (if the lengths of the tuples are the same).

In a case expression, a term may be converted to fit a pattern.

$$\mathrm{case}(M, \langle c\ x_1 \ldots x_n \rangle \Rightarrow N, y \Rightarrow P) \longrightarrow N\,[x_1 := M_1, \ldots, x_n := M_n]\,,$$
$$\text{where } \mathrm{M} \lhd \langle \mathrm{c}\ \mathrm{M}_1 \ldots \mathrm{M}_\mathrm{n} \rangle$$

For example, when the term being matched is a tagged tuple and the pattern is an untagged tuple, we have the conversion

$$\langle \mathbf{tuple}^2_{\mathtt{leaf}}\ M \rangle \lhd \langle \mathbf{tuple}^2\ \langle \mathbf{atom}_{\mathtt{leaf}} \rangle\ M \rangle.$$

3.4.4 Filters and conversion. We require that in all constraints $X \upharpoonright S \leqslant t$ the set S is up-closed, i.e., when $c \in S$ and $\langle c\ \ldots \rangle \lhd \langle c'\ \ldots \rangle$ we also have $c' \in S$.

The intuition is that if a type can pass a filter, a more general type can also pass the filter. For example, the constructor **any** is a member of any non-empty filter.

3.5 Conversion in the type checker

Note however, that Rule ($\lhd$) does not quite fit the type checking algorithm (Section 2.7) as a constraint $t \leqslant u$ may need to be resolved via a combination of the ($\lhd$) rule and a generalisation of Rule W of

Figure 7. Derivation rules for extended constraints.

$$\frac{\varphi \in G}{G \vdash \varphi} \quad (\in)$$

$$\frac{}{G \vdash t \leqslant t} \quad (\mathrm{R})$$

$$\frac{G \vdash t \leqslant u, \qquad G \vdash u \leqslant v}{G \vdash t \leqslant v} \quad (\mathrm{T})$$

$$\frac{G \vdash \mathsf{coerce}(t, u)}{G \vdash t \leqslant u} \quad (\mathrm{Ci})$$

$$\frac{G \vdash t \leqslant u}{G \vdash \mathsf{coerce}(t, u)} \quad (\mathrm{Ce})$$

$$\frac{G \vdash t \leqslant u, \qquad t = \langle c\ \ldots\rangle, \qquad u = \langle d\ \ldots\rangle, \qquad c \not\ll d}{G \vdash \bot} \quad (\mathrm{C}\bot)$$

$$\frac{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant X \qquad G \vdash X \upharpoonright S \leqslant u \qquad c \in S}{G \vdash \langle c\ t_1 \ldots t_n\rangle \leqslant u} \quad (\mathrm{F})$$

Figure 1. However, it is easy to define a general rule that combines these rules. Given a constraint $t \leqslant u$, where both t and u are constructor expressions, we can find a finite set S of constraints (using only proper sub-expressions of t and u) such that for any constraint system G, $G \vdash t \leqslant u$ iff $G \vdash \varphi$, for all $\varphi \in S$.

We define the function coerce as follows:

1. $\mathsf{coerce}(\langle c\ t_1, \ldots, t_n\rangle, \langle c\ u_1, \ldots, u_n\rangle) = \{\varphi_1, \ldots, \varphi_n\}$. where $\varphi_i = (t_i \leqslant u_i)$, if the ith argument of c is covariant, and $\varphi_i = (u_i \leqslant t_i)$ otherwise.
2. $\mathsf{coerce}(\langle \mathbf{tuple}^n_a\ t_2 \ldots t_n\rangle, \langle \mathbf{tuple}^n\ u_1 \ldots u_n\rangle) = \{\langle \mathbf{atom}_a\rangle \leqslant u_1, t_2 \leqslant u_2, \ldots, t_n \leqslant u_n\}$,
3. $\mathsf{coerce}(\langle \mathbf{tuple}^n\ t_1 \ldots t_n\rangle, \langle \mathbf{tuple}\ u\rangle) = \{t_1 \leqslant u, \ldots, t_n \leqslant u\}$,
4. $\mathsf{coerce}(\langle \mathbf{atom}_a\rangle, \langle \mathbf{atom}\rangle) = \emptyset$.
5. $\mathsf{coerce}(t, \langle \mathbf{any}\rangle) = \emptyset$.

For any pair of constructor expressions t and u, $\mathsf{coerce}(t, u)$ provides a set of constraints that need to hold in order for the constraint $t \leqslant u$ to hold.

3.6 Putting everything together

Let the set of *type expressions* $t, u \in \mathsf{TExp}$ be the least set such that:

1. $\mathsf{TVar} \subseteq \mathsf{TExp}$, and
2. $\langle c\ t_1 \ldots t_n\rangle \in \mathsf{TExp}$, where n is the arity of c, and $t_i \in \mathsf{TExp}$, for $i \leq n$.

Let the set of *constraints* $\varphi \in \mathsf{Constraint}$ be formulas of the following forms:

1. $t \leqslant u$, for $t, u \in \mathsf{TExp}$,
2. $X \upharpoonright S \leqslant t$, for $X \in \mathsf{TVar}$, $t \in \mathsf{TExp}$, and S an up-closed set of constructors, and
3. $\bot$.

We give the derivation rules for extended constraints in Figure 7 and the type rules for the extended lambda calculus in Figure 8. A filter expression $X \upharpoonright c$ is a shorthand for $X \upharpoonright S$, where S is the smallest up-closed set containing c. Similarly, we use $X \setminus c$ as a shorthand for $X \upharpoonright S$, where S is the largest up-closed set *not* containing c. We also write $c \ll c'$ if $c = c'$ or there are type expressions $t = \langle c\ \ldots\rangle$ and $u = \langle c'\ \ldots\rangle$ such that $t \lhd u$, and $c \not\ll c'$ if $c \ll c'$ does *not* hold.

Figure 8. Typing rules for extended lambda calculus.

$$\frac{(x : t) \in \Gamma}{\Gamma \Vdash x : t} \quad (\text{axiom})$$

$$\frac{\Gamma\,[x \mapsto t] \Vdash M : u}{\Gamma \Vdash \lambda x.M : \langle c_\lambda\ t\ u\rangle} \quad (\text{abstraction})$$

$$\frac{\Gamma \Vdash M : \langle c_\lambda\ t\ u\rangle \qquad \Gamma \Vdash N : t}{\Gamma \Vdash MN : u} \quad (\text{application})$$

$$\frac{\Gamma \Vdash M_i : t_i, \quad 1 \leq i \leq n}{\Gamma \Vdash \langle c\ M_1 \ldots M_n\rangle : \langle c\ t_1 \ldots t_n\rangle} \quad (\text{constructor})$$

$$\frac{\begin{array}{c}\Gamma \Vdash M : t \\ t \leqslant X \\ X \upharpoonright c \leqslant \langle c\ u_1 \ldots u_n\rangle \\ \Gamma\,[x_1 \mapsto u_1, \ldots, x_n \mapsto u_n] \Vdash N : w \\ X \setminus c \leqslant Z \\ \Gamma[y \mapsto Z\,] \Vdash P : w\end{array}}{\Gamma \Vdash \mathsf{case}(M, \langle c\ x_1 \ldots x_n\rangle \Rightarrow N, y \Rightarrow P) : w} \quad (\text{case})$$

$$\frac{\Gamma \Vdash M : t \qquad t \leqslant u}{\Gamma \Vdash M : u} \quad (\text{subsumption})$$

4 How to make Erlang statically typed

As noted by Mitchell [17], the type rules of a subtyping system are more general than those of a Hindley-Milner type system. Thus the subtyping system should be able to type any program typable in Hindley-Milner by simply removing data type definitions and using predefined constructors instead of those given in data type definitions. Indeed, it has been our experience that if a program is written as if it was intended for a Hindley Milner type system, it will type under the subtyping system.

The subtyping system should in principle be able to type check a complex program, relying only on top-level specifications and deducing internal data types. In practice, it is a good idea to introduce function specifications and data types declarations for various intermediate function definitions and data types as this will help locating the sources of type errors and speed up type checking.

4.1 Type definitions and function specifications

The system accepts source files containing Erlang code, type definitions and function specifications. One example:

```
-module(example1).
%: +type list(X) = [] + [X|list(X)].

%: +func append :: list(X) * list(X) -> list(X).
append([A | B], C) ->
    [A | append(B, C)];
append([], C) -> C.

%: +func dup :: list(integer()) -> list(integer()).
dup(S) ->
```

```
    append(S, S).
```

(In Erlang, lower case identifiers without arguments indicate atoms, upper case identifiers are variables.) The first type definition gives the polymorphic and recursive type `list()`, which is of course either the empty list constructor (`[]`) or the cons constructor applied to the type parameter and the list type, (`[X|list(X)]`). It should be stressed that nothing about the list constructors is hard-coded in the type checker. Everything the type checker knows about lists and the two list constructors is present in the specifications above. It is possible to specify and use a list type based on other constructors, or, conversely, to use the lists constructors to build other types.

The character combination "`%:`" is treated as white space by the scanner of the type system while the Erlang compiler sees the character `%` as the start of a comment. Thus the type definitions and specifications in the module above will be read by the type checker but ignored by the compiler.

Type definitions use the keyword `type` and give a name to a type. Specification use the keyword `spec` and state that an Erlang function should implement a certain type. The type checker will verify that this is indeed the case. For example, the specification of the function `append` simply states that the function takes two lists as arguments, and returns a list of the same type.

In the language for types, an atom followed by an argument list (for example: `list(integer())`) indicates either a type constructor or a type defined in some type definition. Some constructors use different syntax, for example the empty list `[]` and the list constructor `[...|...]`. Also, atoms, tagged and untagged tuples have the same syntax as in Erlang. As Erlang allows a function to have any number of arguments, we use a function type constructor for each number of arguments. Examples of function types with zero, one, or two arguments:

```
() -> atom(),   integer() -> atom(),
integer() * float() -> atom().
```

Among other primitive types in the source language are `atom()`, the type of atoms, and `integer()`, the type of integers.

Finally, we use the notation `any()` and `none()` for the universal and empty types, respectively.

4.2 Unsafe features

4.2.1 Non-exhaustive case analysis. In some programming languages, for example Standard ML, the type system will give a warning if the case analysis is incomplete. Consider, for example, a function that returns the last element of a list but has no clause for the case when the list is empty. In contrast, a Haskell compiler will accept this function without warnings. There are certainly good arguments for either choice. In the type system described here, we chose to follow Haskell's approach and accept such programs.

4.2.2 Promises. Many functions in the standard library are ill-suited for static typing. For example, there are many operations that may return a value of any type. Among these are primitives for process communication and functions that read data from a file or from standard input. Such operations should be declared to return the universal type, but typically code that uses one of these operations expects values of a particular type.

Rather than barring programmers from using such operations, our system includes a primitive `promise` that allows the programmer to assert that a variable has a particular type. We illustrate the use of the primitive with a simple example.

```
%: +func f::() -> integer().
f() ->
    {ok, X} = io:read(">"),
%:  promise X :: integer(),
    X.
```

Now, promises are unsafe in the sense that if the programmer lies to the type system in a promise the type system will trust the promise. A cautious programmer could of course insert code that checked the promise, and in a more well-integrated system such tests could be inserted automatically.

Using promises it is possible to leave one part of a program untyped, thus it is possible to gradually introduce static typing in a dynamically typed program.

The implementation of the type system uses promises in four locations. Two uses occur in the module `program` and are associated with calls to the function `get_value` of the library module `proplists`, which extracts a field from a property list. Since a property list may store any value, and different types of values are associated with different properties, there is no way to statically determine the type of one particular field.

The two other uses of `promise` occur in a module which expands records. These uses of a promise could perhaps be avoided by more careful coding and better use of polymorphism.

4.2.3 External modules. The type system checks one module at the time. If a second module is referenced, and specifications are available, the type system will under default settings use the specifications instead of analysing the second module. Naturally, until the second module is also checked, there is no way of knowing whether the specification in the second module really conform with the actual code.

In the type system, there are some places where typing relies on specifications, but the type system has not checked that the specifications match the corresponding function definitions. For example, the parser which is based on the standard Erlang parser is not checked. Instead, the abstract syntax tree which is generated by the parser is specified separately. Also, there is a module that implements a modified version of the standard Erlang preprocessor. The type system relies on specifications of three functions of that module that are not checked. There are also seven functions in standard libraries (involving IO, the file system and timers) that are not checked. Perhaps more importantly, the module `lists` which implements various operations on lists could not be checked. The reason is that many functions in that library manipulate lists of tuples. Consider for example `keysort`, which takes an integer (the key) and a list of tuples that are sufficiently long to contain the key. The list is sorted by the position given by the key. To type programs that use this function, a specification should reflect not only that the second argument and the result are both lists of tuples but also that the tuples of the result are of the *same* type as the input tuples.

4.3 One feature of Erlang that the type system cannot handle

While the type rules can reason about programs that use higher-order functions, Erlang offers other forms of indirect function calls are harder to analyse.

Erlang's built in function `apply` allows the destination of a function call to be computed at run-time. Thus depending on input any exported function of any module may be called. Currently, the type system will simply reject programs that use `apply` and related primitives.

5 The Implementation

5.1 Predicates

The front end of the type checker translates function definitions (i.e., Erlang code), type definitions and specifications into constraints. These constraint systems are organised in *predicates*.

A predicate takes the form

$$\text{predicate } \mathit{Name}\,[X_1, ..., X_n]\ \mathit{Body}$$

where *Name* is an identifier, $X_1, \ldots, X_n$ are variables (the parameters of the predicate), *Body* is a set of constraints and calls. In other words, a predicate is a way of associating a name and some parameters with a constraint system. A *call* is of the form

$$\text{call } \mathit{Name}\,[Y_1, \ldots, Y_m]$$

We refer to $Y_1, \ldots, Y_m$ as the *arguments* of the call. Free variables of *Body* that are not among the parameters are treated as local variables.. The resolution of a call is simple; the call is replaced with the body of the predicate where every variable that occurs as a parameter is replaced with the corresponding argument, and other free variables are replaced with fresh variables. Thus a collection of predicates together with a top-level call can be expanded into a constraint system.

After the predicates have been generated, the remaining phases of the type checker do not rely on any other information beside the structure of the predicates and the constraint systems explicit in the predicates.

In the examples below, we use the following format: Variables are upper case. A constraint always takes the form $t_1 \leqslant t_2$, where t_1 and t_2 are type expressions. Type expressions are variables, filter expressions or built using a constructor. In the latter case, they are written $\langle c\ t_1, \ldots, t_n\rangle$ where c is a constructor and $t_1, \ldots, t_n$ are type expressions. Among constructors are the nullary constructor $\mathbf{atom}_a$, for any atom a, the n-ary constructor $\mathbf{tuple}^n$, for any n, the binary list constructor **cons** and the constructor of the empty list **nil**. For functions we use a special notation and write $([t_1, \ldots, t_n] \to u)$ for a type expressions that describes a function of n arguments that takes arguments of the types $t_1, \ldots, t_n$ and returns a result of type u. Thus the constructor has arity $n + 1$. The reason for using multi-argument function types is of course that Erlang distinguishes between functions with different number of arguments. Filter expressions are either of the form $X \upharpoonright c$ or $X \setminus c$ where X is a variable and c a constructor.

Type definitions and specifications are used in two situations, when generating a type according to a definition or a function specification, and when checking that a supplied type matches. Since these two cases require different constraints and don't interact we have found it convenient to separate the two cases and define two predicates for each type definition and function specification; a *lower* predicate and an *upper* predicate.

5.1.1 Examples.

We show predicates for simple type definitions. First, a simple type definition with two alternatives:

```
+type bool() = true + false.
```

Consider first the lower predicate for the type `bool()`.

$$\text{predicate } \mathit{type\_lower\_bool}\,[T]$$
$$([] \to A) \leqslant T, \tag{2}$$
$$\langle \mathbf{atom}_{\mathtt{true}}\rangle \leqslant A, \tag{3}$$
$$\langle \mathbf{atom}_{\mathtt{false}}\rangle \leqslant A. \tag{4}$$

Like all predicates for type definitions, the predicate takes a single parameter (T). As the type `bool` does not take any parameters, the predicate generates a function type without parameters (2). The result type (A) describes the possible values of a variable or expression of type `bool()`. There are two possible values, the atom `true` or the atom `false`.

Next, the upper predicate for `bool()`.

$$\text{predicate } \mathit{type\_upper\_bool}\,[T]$$
$$([] \to A) \leqslant T, \tag{5}$$
$$A \upharpoonright \mathbf{atom}_{\mathtt{true}} \leqslant \langle \mathbf{atom}_{\mathtt{true}}\rangle, \tag{6}$$
$$A \setminus \mathbf{atom}_{\mathtt{true}} \leqslant \langle \mathbf{atom}_{\mathtt{false}}\rangle. \tag{7}$$

In the upper predicates for `bool()`, we use filters (as explained in Section 3.2) to isolate the two cases. When filtered with the set $\{\mathbf{atom}_{\mathtt{true}}\}$ the type of A must be a subtype of $\langle \mathbf{atom}_{\mathtt{true}}\rangle$ (6). The third line (7) uses a filter to exclude any use of the atom `true`. If A is not the atom `true`, the type of A must be a subtype of the atom `false`.

Next we consider a simple parametric type.

```
+type option(X) = none + {some, X}.
```

Both the lower and upper predicate define the type of T as a function with one parameter. In the lower predicate, the parameter X is introduced in constraint (8) and used in constraint (10). By constraint (9) the result may be the atom `none` and by constraint (10) the result may be a tuple, where the second element is given by the parameter X.

$$\text{predicate } \mathit{type\_lower\_option}\,[T]$$
$$([X] \to A) \leqslant T, \tag{8}$$
$$\langle \mathbf{atom}_{\mathsf{none}}\rangle \leqslant A, \tag{9}$$
$$\langle \mathbf{tuple}^2_{\mathsf{some}}\ X\rangle \leqslant A. \tag{10}$$

The upper predicate follows a similar pattern. The type passed to the predicate needs to be a function type with one parameter, as the type is parametric (11). Filters are used to distinguish between the cases when the result of the supplied type is the atom none (12) and when it is *not* (13).

$$\text{predicate } \mathit{type\_upper\_option}\,[T]$$
$$T \leqslant ([X] \to A), \tag{11}$$
$$A \upharpoonright \mathbf{atom}_{\mathsf{none}} \leqslant \mathbf{atom}_{\mathsf{none}}, \tag{12}$$
$$A \setminus \mathbf{atom}_{\mathsf{none}} \leqslant \langle \mathbf{tuple}^2_{\mathsf{some}}\ X\rangle. \tag{13}$$

We end with a (non-parametric) recursive type,

```
+type intlist() = [] + [integer() | intlist()].
```

Both the lower and upper predicate are recursive. In the lower predicate, the recursive call supplies the type of the rest of the list

(18).

$$\begin{aligned}
&\text{predicate } \mathit{type\_lower\_intlist}\ [T] \\
&\quad ([] \to A) \leqslant T, && (14)\\
&\quad \langle \mathbf{nil} \rangle \leqslant A, && (15)\\
&\quad \langle \mathbf{cons}\ \langle \mathbf{integer} \rangle\ B \rangle \leqslant A, && (16)\\
&\quad \text{call } \mathit{type\_lower\_intlist}\ [U], && (17)\\
&\quad U \leqslant ([] \to B). && (18)
\end{aligned}$$

In the upper predicate, constraint (23) in combination with the call (22) gives an upper bound to the rest of the list (B). The next section describes how recursive predicates are replaced with constraints.

$$\begin{aligned}
&\text{predicate } \mathit{type\_upper\_intlist}\ [T] \\
&\quad T \leqslant ([] \to A), && (19)\\
&\quad A \upharpoonright \mathbf{nil} \leqslant \langle \mathbf{nil} \rangle, && (20)\\
&\quad A \setminus \mathbf{nil} \leqslant \langle \mathbf{cons}\ \langle \mathbf{integer} \rangle\ B \rangle, && (21)\\
&\quad \text{call } \mathit{type\_upper\_intlist}\ [U], && (22)\\
&\quad ([] \to B) \leqslant U. && (23)
\end{aligned}$$

5.2 Recursion in predicates

Let's first look at the case where a predicate is recursive, but there are no mutually recursive predicates. Consider the predicate *type_lower_intlist* of the previous section. It contains a call call *type_lower_intlist* $[U]$. The strategy is to simply merge the parameters with the arguments in the recursive calls, i.e., replace all of them with a set of variables. The recursive calls can now be removed. In the example, this gives us the following predicate:

$$\begin{aligned}
&\text{predicate } \mathit{type\_lower\_intlist}\ [T] \\
&\quad ([] \to A) \leqslant T, && (24)\\
&\quad \langle \mathbf{nil} \rangle \leqslant A, && (25)\\
&\quad \langle \mathbf{cons}\ \langle \mathbf{integer} \rangle\ B \rangle \leqslant A, && (26)\\
&\quad T \leqslant ([] \to B). && (27)
\end{aligned}$$

Now, it is possible that this approach to recursion will sometimes be overly aggressive; one can create a recursive predicate where merging recursive calls in this manner gives a non-recursive predicate where the body is inconsistent, but where a more conservative approach would have avoided inconsistency. However, as our approach is more general than Hindley-Milner typing (which also treats recursive calls as monomorphic), it seems safe to assume that it will work well in practice.

To handle mutual recursion, it is useful to view the predicates as a directed graph where each predicate is a node and an edge connects two predicates if there is a call in the first predicate to the second. Mutually recursive predicates form strongly connected components. Predicates that form a strongly connected component are combined into a new predicate. The parameter list of this predicate is the concatenation of the parameter lists of the predicates it replaces.

Non-recursive calls between predicates are treated as polymorphic. Thus each such call results in the duplication of constraints. To reduce the cost of duplication, various constraint simplification algorithms are applied before duplication.

5.3 The constraint solver

The constraint solver uses a graph representation where nodes are type variables and edges are labeled with filters and represent constraints of the form $X \upharpoonright S \leqslant Y$. With each node, say for a variable X, we associate constructor expressions t such that $t \leqslant X$ (*supports*) and $X \leqslant t$ (*covers*). As suggested by Heintze [9], we do not compute a representation of the transitive closure. Instead, when a link $X \upharpoonright S \leqslant Y$ is added, a depth first search collects the direct and indirect supports of X and a second dfs collects the covers of Y. The covers and supports are then combined.

The solver (and the rest of the type checker) is written in a pure functional style, with the exception of IO and calls to the `timer` library.

5.4 Constraint simplification

Since our implementation of polymorphic type checking sometimes requires a constraint system to be duplicated, it is reasonable to use constraint simplification to (hopefully) improve performance. We have developed two approaches to constraint simplification.

Our starting point is a constraint system G and the set of variables P which serve as an interface to the constraints in G. The constraint system G represents a definition of a function or a type definition. The constraint system needs to be duplicated if it is used in different contexts.

In the first simplification, we consider *reachability*, i.e., the set of constraints in G that can be reached from P. The second simplification considers *stability*. Given a constraint system G and a set of visible variables P it sometimes happens that a variable is reachable, but that there is no need to duplicate the variable if the constraint system is duplicated. Suppose, for example:

$$P = \{X\}, G = \{\langle \mathbf{cons}\ Y\ Z \rangle \leqslant X, \langle \mathbf{cons}\ Y_1\ Y_2 \rangle \leqslant Y\}. \qquad (28)$$

Even if G is used in different contexts, there is no need to duplicate the variables Y, Y_1 and Y_2. This situation occurs for example if G is the constraint system of a function that returns a complex data structure but the data structure does not depend on the input. Obviously, the type of the result will always be the same.

6 Measurements

Table 2 lists modules of the type checker itself and the time required to check the modules. In Table 3 we find a collection of modules that were developed independently of the type checker. Module `barnes` implements the Barnes-Hut algorithm, a simulation of the n-body problem. Module `smith`, which implements the Smith–Waterman algorithm for local sequence alignment, gives a clear example of the performance impact of specifications of intermediate function definitions; the first version (`smith0`) only specifies the top function, the second (`smith`) version which also specifies two internal functions is several times faster. In these (`smith`) modules, a line of code had to be modified to use explicit matching as the algorithm used a more general type internally but always returned a more restricted type.

The following four modules are part of a program that computes the mandelbrot set. The module `mandelbrot` uses process communication to transmit intermediate results and as the type system cannot reason about the types of data transmitted in messages, a promise was needed to provide the types. Also, `mandelbrot` uses an older primitive to spawn processes (that requires a function call

Table 2. Modules in the type checker. LOC: lines of code. LOD: lines of specifications and type definitions. Blank lines and comments are not counted. The final column shows time (in seconds) to check the module.

Module	LOC	LOD	Time
agenda	200	15	0.55
coalesce	220	16	0.34
conn	181	38	0.10
convert	815	112	3.83
graph	137	13	0.06
match	130	30	0.24
poly	321	20	1.43
pos	14	7	0.01
program	310	49	2.31
reach	398	26	0.57
record_expand	88	22	0.39
rfilter	139	14	0.08
sanity	197	12	0.50
scfa_file	44	5	0.04
solver	419	65	0.48
walker	695	102	1.77
worklist	26	8	0.02

Table 3. Modules from a collection of benchmark programs. The table is organised as Table 2.

Module	LOC	LOD	Time
barnes	182	4	0.09
smith0	76	1	0.63
smith	76	11	0.04
complex	31	10	0.02
mandelbrot	37	8	0.02
image	59	12	0.03
render	22	2	0.03
gb_trees	303	30	0.11
gb_sets	512	31	0.21
ordsets	148	21	0.06

of the form described in Section 4.3) which was replaced by the modern version.

The modules `gb_trees`, `gb_sets` and `ordsets` come from the standard library distributed with the Erlang implementation. The modules required only minimal changes to type. In some cases, a function used a more general type internally but returned a value that was of a more specific type. This was the case in the two library modules `gb_trees` and `gb_sets`, and the `smith` modules. Here, an explict matching had to be inserted to help the type system discover that a value of a more specific type was returned. For example, in the `gb_trees` module, a function definition

```
insert(Key, Val, {S, T}) when is_integer(S) ->
    S1 = S+1,
    {S1, insert_1(Key, Val, T, ?pow(S1, ?p))}.
```

had to be rewritten:

```
insert(Key, Val, {S, T}) ->
    S1 = S+1,
    {Key1, Value, Smaller, Bigger}
        = insert_1(Key, Val, T, ?pow(S1, ?p)),
    {S1, {Key1, Value, Smaller, Bigger}}.
```

The type checker was set to use constraint simplification and "prefer specifications", i.e., to use specifications of functions, when available, when checking Erlang code containing function calls. All measurements have been run on a 1.3 GHz Intel Core i5 (a 2013 Macbook Air). In the measurements, only one core was used. The Erlang implementation used a BEAM byte-code emulator.

7 Related work

Kozen et al. [12] showed that the problem of checking whether a term in lambda calculus can be typed by a subtyping system could be determined in $O(n^3)$ time. They give an inductive definition of a type structure and give an efficient algoritm for checking whether a term has a type. It seems, however, that relying on an inductive definition of the type structure makes it difficult to extend the approach to handle programming languages and type systems with more features.

Marlow and Wadler [14] describe an early prototype of a subtyping system for Erlang written in Haskell and report promising results; the type system has been applied to thousands of lines of library code and no difficulties are antipicated. However, Erlang has many features that the type system should not be able to handle, and even "nice" Erlang programs sometimes do things that should be hard to reason about in their type system. Their constraint language is based on one by Aiken and Wimmers [1]. However, there are some changes, for example, their system is restricted to discriminated unions that should give about the same expressiveness as the filter concept described in Section 3.2. They implement polymorphism by deriving a most general type for function definitions. This is unlike the current paper and others, for example [19] and [7] where a constraint system represents the set of possible types. The problem with their approach to polymorphism seems to be that a function may be typed in many different ways and there are cases where there is no most general type. The authors give an illustrative example (Section 9.3) where the type system finds a type for a function definition which is not the one one would expect. They show how their type system can be extended to handle higher-order functions (which were not part of Erlang at the time) but note that a solution would then be incomplete [23]; the type system might fail to prove correctness of certain programs. The problem seems most pressing when reasoning about type definitions.

Eifrig et al. [5] present an interesting approach to subtyping. Types are *constrained*, i.e., each type comes with a constraint system which gives a rich type system, though it does not seem that this gives any additional expressiveness compared to the approach in this paper. Like this paper, Eifrig uses a propagation algorithm to determine whether a constraint system is acceptable. Unlike this paper, there is no attempt to link the propagation algorithm to a definition of consistency using derivation rules, instead they give a subject reduction proof where they show that each reduction step preserves the outcome of the propagation algorithm. This is unsatisfactory from a theoretical point of view, a practical problem is that it makes the type system hard to extend; any modified version of the type system requires a new algorithm for checking constraints. Since the proof of the subject reduction property depends on the algorithm every new version of the algorithm needs a new version of the (rather tedious) proof. Any mistakes in the design of the algorithm will become apparent at a late stage, and it will be hard to tell whether the problem is due to a mistake in the design of the algorithm or in the underlying type system.

Typed Scheme [22] requires that the program contains specifications of all functions and data structures. Thus, the problem of type checking is in some regards much simpler as there is only a limited need to deduce types for immediate values. In contrast, the system presented here can deduce complex intermediate data structures.

There are several other recent attempts to integrate static and dynamic typing that rely on some form of subtyping but not in combination with type inference, for example [6, 11, 20, 21, 25, 26]. These systems rely on run-time type checks in the conversion from dynamically typed values to values with static types.

Dolan and Mycroft [4] present a subtype system for SML, extended with a universal type, an empty type and a record concept. Interestingly, they report that their type system has principal types. However, as the principal types are not minimal (in fact, a function may have an infinite set of principal types of unbounded size) the advantage of principal types is unclear. Their implementation of polymorphism relies on heuristic simplification of principal types, analogous to the constraint simplification algorithms exploited in this work.

Most type systems (including the one presented in this paper) attempt to guarantee some degree of safety from type errors at run-time. Lindahl and Sagonas [13] take the opposite approach and give a type system for Erlang that that only rejects programs that are *guaranteed* to fail. This allows the type system to work with programs that were not written with static typing in mind.

8 Conclusions

Designing a static type system for a programming language that was not designed for static typing poses many challenges. Sometimes typing a program requires some minor adjustments, sometimes there are features that seem fundamentally unsuited for static typing. More interesting are situations that seem amenable to static typing, if only the type system was a little bit more powerful.

The subtyping system we have developed is a generalisation of Hindley-Milner type inference. As Hindley-Milner type inference has been used in functional programming for decades, we expected that a generalisation should be capable of handling most functional programs that did not involve any exotic features of Erlang. Experience has confirmed this expectation; programs that would have typed in, say, an SML implementation will indeed type here.

The interesting question is: which programs can be typed under a subtyping system that cannot be typed by the Hindley-Milner system? There are some obvious situations. For example: a complex type that uses fewer constructors than another and is thus a subtype or a type that uses the same constructors as another (but is otherwise unrelated). The use of the subtyping system in the typing of the implementation of the subtyping system has offered some insight in the practical aspects of using subtyping in development.

References

[1] Alexander Aiken and Edward L. Wimmers. 1993. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture.* 31–41.

[2] Roberto M Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (1993), 575–631.

[3] Henk Barendregt, Wil Dekkers, and Richard Statman. 2013. *Lambda calculus with types.* Cambridge University Press.

[4] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017).* ACM, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

[5] Jonathan Eifrig, Scott Smith, and Valery Trifonov. 1995. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science* 1 (1995), 132–153.

[6] Cormac Flanagan. 2006. Hybrid Type Checking. In *POPL'06* (Charleston, South Carolina, USA). ACM, New York, NY, USA, 245–256.

[7] Cormac Flanagan and Matthias Felleisen. 1999. Componential Set-based Analysis. *ACM Trans. Program. Lang. Syst.* 21, 2 (March 1999), 370–416. https://doi.org/10.1145/316686.316703

[8] Nevin Heintze. 1994. Set-Based Analysis of ML Programs. In *ACM Conference on Lisp and Functional Programming.* 306–317.

[9] Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Programming Language Design and Implementation (PLDI).* 254–263.

[10] Leon Henkin. 1949. The Completeness of the First-Order Functional Calculus. *Journal of Symbolic Logic* 14, 3 (1949), 159–166.

[11] Kenneth Knowles and Cormac Flanagan. 2010. Hybrid Type Checking. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 6 (Feb. 2010), 34 pages.

[12] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. 1994. Efficient inference of partial types. *J. Comput. System Sci.* 49, 2 (1994), 306–324.

[13] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical type inference based on success typings. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming.* ACM, 167–178.

[14] Simon Marlow and Philip Wadler. 1997. A practical subtyping system for Erlang. *ACM SIGPLAN Notices* 32, 8 (Aug. 1997), 136–149.

[15] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17 (1978), 348–375.

[16] John C. Mitchell. 1984. Coercion and Type Inference. In *Principles of Programming Languages.* ACM, 175–185.

[17] John C. Mitchell. 1991. Type inference with simple subtypes. *Journal of Functional Programming* 1 (1991), 245–285.

[18] Jens Palsberg and Patrick O'Keefe. 1995. A type system equivalent to flow analysis. *ACM Toplas* 17, 4 (July 1995), 576–599.

[19] François Pottier. 2001. Simplifying subtyping constraints: a theory. *Information and Computation* 170, 2 (2001), 153–183.

[20] Jeremy G. Siek and Ronald Garcia. 2012. Interpretations of the gradually-typed lambda calculus. In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming.* ACM, 68–80.

[21] Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

[22] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. *ACM SIGPLAN Notices* 43, 1 (2008), 395–406.

[23] Valery Trifonov and Scott Smith. 1996. Subtyping constrained types. In *Static Analysis.* Springer, 349–365.

[24] Dirk van Dalen. 2013. *Logic and structure, fifth edition.* Springer-Verlag.

[25] Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *Programming Languages and Systems.* Springer, 1–16.

[26] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL'10* (Madrid, Spain). New York, NY, USA, 377–388. https://doi.org/10.1145/1706299.1706343

Zero-Cost Constructor Subtyping

Andrew Marmaduke
The University of Iowa
andrew-marmaduke@uiowa.edu

Christopher Jenkins
The University of Iowa
christopher-jenkins@uiowa.edu

Aaron Stump
The University of Iowa
aaron-stump@uiowa.edu

ABSTRACT

Constructor subtyping is a form of subtyping where two inductive types can be related as long as the inductive signature of one is a subsignature of the other. To be a subsignature requires every constructor of the smaller datatype to be present in the larger datatype (modulo subtyping of the constructors' types). In this paper, we describe a method of impredicative encoding for datatype signatures in Cedille that allows for highly flexible support of constructor subtyping, where the subtyping relation is given by a derived notion of *type inclusion* (witnessed by a heterogeneously-typed identity function). Specifically, the conditions under which constructor subtyping is possible between datatypes are fully independent of the order in which constructors are listed in their declarations. After examining some extended case studies, we formulate *generically* a sufficient condition for constructor subtyping in CDLE using our technique.

ACM Reference Format:
Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. 2020. Zero-Cost Constructor Subtyping. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Inductive datatypes are the least set of terms generated by their constructions. Constructor subtyping arises when we interpret the subtype relation as the subset relation between the sets of terms. Equivalently, one can interpret constructor subtyping as the subset relation between sets of constructors treated as uninterpreted constants. For example, the inductive datatype for natural numbers, $\mathbb{N}$, is represented by the set $\{0, succ\}$ and the inductive datatype for the (unquotiented) integers, $\mathbb{Z}$, is represented by the set $\{0, succ, negate\}$. It is trivial to see that $\{0, succ\} \subseteq \{0, succ, negate\}$ which implies $\mathbb{N} \subseteq \mathbb{Z}$.

Subtyping allows for function and proof reuse, and constructor subtyping in particular enriches the subtype relation to include relationships when an inductive datatype is a subset of any other inductive datatype. Function overloading is a natural use case of subtyping and although constructor subtyping is not required for overloading functions, the kinds of overloads that are possible benefit from the enriched relation. Additionally, constructor subtyping yields a form of incremental definition where a datatype is extended with additional constructors, implicitly inheriting the signature of the extended datatype.

The concept of constructor subtyping is attributed to Kent Peterson and (independently) A. Salvesen by Coquand [8], but was developed by Barthe et al. [3, 4] into new calculi that directly support constructor subtyping. However, Barthe did not investigate constructor subtyping for indexed inductive datatypes. Furthermore, his calculus has a weak notion of canonical elements in the presence of type arguments. In this paper, we describe a highly flexible approach to constructor subtyping in the Cedille programming language, where the subtype relation is a derived notion of type inclusion — directly analogous to the set inclusion discussed earlier.

Precisely, our contributions are:

(1) a method of impredicative encoding of datatype signatures in Cedille that treats datatype constructor lists as truly unordered sets, allowing users or language implementors their choice of labeling set and assignment of those labels to constructors;
(2) a demonstration that this method supports highly flexible constructor subtyping, where the subtyping relation is a derived notion of type inclusion: for two compatible constructors to be identified, it is necessary only that they be assigned the same label;
(3) we examine three case studies: natural numbers as a subtype of integers, lists as a subtype of vectors with a tree branching constructor, and a language extension of the simply typed lambda calculus by numeral expressions;
(4) we prove *generically* (for any labeling type and label-indexed family of constructor argument types) a sufficient condition for subtyping of datatypes, by instantiating the framework of Firsov et al. [11] for generic (and efficient) encodings of inductive datatypes in Cedille;
(5) finally, all presented derivations and examples are formalized in Cedille (https://github.com/cedille/cedille-developments/tree/master/constructor-subtyping).

In the following section we provide the necessary background of Cedille's core theory and describe the features that are required to implement constructor subtyping (Section 2). Next, we present the core idea behind the lambda encoding and describe the derivation in Cedille (Section 3). After, we explore three case studies involving parametric and indexed datatypes (Section **??**). We then formulate generically a sufficient condition for subtyping of datatypes for the form of signatures produced by our method of encoding (Section 5). The paper is concluded by remarking on related work (Section 4) and summarizing our results (Section 5).

Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$\frac{\Gamma, x : T \vdash t' : T' \quad x \notin FV(|t'|)}{\Gamma \vdash \Lambda\, x{:}T.\, t' : \forall\, x{:}T.\, T'} \qquad \frac{\Gamma \vdash t : \forall\, x{:}T'.\, T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \text{ -}t' : [t'/x]T}$$

$$|\Lambda\, x{:}T.\, t| \;=\; |t| \qquad\qquad |t \text{ -}t'| \;=\; |t|$$

Figure 1: Implicit Functions

2 BACKGROUND ON CEDILLE

Cedille is a dependently typed programming language whose core theory is Calculus of Dependent Lambda Eliminations (CDLE) [25]. It extends the extrinsically typed Calculus of Constructions (CoC) with three additional primitives: the implicit (or erased) function types of Miquel [22], the dependent intersection type of Kopylov [17], and an equality type of erased terms. Critically, lambda-encoded datatypes supporting an induction principle is derivable in CDLE where it is not in CoC [14]. Moreover, efficient lambda encodings exist which alleviate prior concerns with lambda-encoded inductive data [11]. In the remainder of this section we review the three additional typing constructs that are added to CoC to form CDLE.

2.1 Implicit Functions and Erasure

Erasure in CDLE (denoted by vertical bars, e.g. $|t|$) defines what is operationally relevant in the theory. It can be understood as a kind of program extraction that produces an untyped λ-term. Typing information such as type abstractions or type annotations are all erased. Implicit functions, the rules for which are listed in Figure 1, give a way of expressing when a *term* should also be treated as operationally irrelevant.

We write a capital lambda to denote abstraction by either a type or an erased term (e.g. $\Lambda\, X.\, \Lambda\, y.\, \lambda\, x.\, x$ for type X and term y), a center dot for type-application (e.g. $T_1 \cdot T_2$ or $t \cdot T$), a dash for erased-term application (e.g. t_1 -t_2), and juxtaposition for term-to-term and type-to-term application (e.g. $t_1\ t_2$ and $T\ t$). In types, we use the standard forall quantifier symbol for both erased function types and type quantification (e.g. $\forall\, X{:}\star.\, T_2$ and $\forall\, x{:}T_1.\, T_2$). For convenience, we write an open arrow for an implicit function type that is not dependent (i.e. $T_1 \Rightarrow T_2$). In contrast, relevant dependent functions are written with the capital greek pi (i.e. $\Pi\, x{:}T_1.\, T_2$) and a single arrow when not dependent (i.e. $T_1 \rightarrow T_2$). The typing rules for implicit functions are similar to those for ordinary ones, except for additional concerns of erasure. To introduce an implicit function, there is a syntactic restriction that the bound variable does not occur free in the erasure of the body of the function; this justifies the erasure of the elimination form, which completely removes the given argument.

2.2 Dependent Intersections

In an extrinsically typed theory such as CDLE terms do not have unique types. If we view all types as categorizing sets of ($\beta\eta$-equivalence classes of) terms, then an intersection type is interpreted precisely as a set intersection. Additionally, this idea has a dependent counterpart appropriately called a dependent intersection, the rules for which are listed in Figure 2, Syntactically, the introduction form of a dependent intersection is a pair with the

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : [t_1/x]T_2 \quad |t_1| = |t_2|}{\Gamma \vdash [t_1, t_2] : \iota\, x{:}T_1.\, T_2}$$

$$\frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.1 : T_1} \qquad \frac{\Gamma \vdash t : \iota\, x{:}T_1.\, T_2}{\Gamma \vdash t.2 : [t.1/x]T_2}$$

$$|[t_1, t_2]| \;=\; |t_1| \qquad |t.1| = |t| \qquad |t.2| = |t|$$

Figure 2: Dependent Intersection

$$\frac{FV(t\ t') \subseteq dom(\Gamma)}{\Gamma \vdash \beta\{t'\} : \{t \simeq t\}} \qquad \frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t' : [t_2/x]T}{\Gamma \vdash \rho\ t\ @\ x.T - t' : [t_1/x]T}$$

$$\frac{\Gamma \vdash t : \{t_1 \simeq t_2\} \quad \Gamma \vdash t_1 : T}{\Gamma \vdash \varphi\ t - t_1\ \{t_2\} : T}$$

$$|\beta\{t'\}| \;=\; |t'| \qquad |\rho\ t\ @\ x.T - t'| \;=\; |t'|$$

$$|\varphi\ t - t_1\ \{t_2\}| \;=\; |t_2|$$

Figure 3: Equality

constraint that the terms of the pair are $\beta\eta$-equal modulo erasure. This equality restriction on the components of the pair allows the erasure rule for the dependent intersection to forget one of the components, recovering our intuition for set intersection. We write the type of a dependent intersection with the greek iota (i.e. $\iota\, x{:}T_1.\, T_2$), the introduction of dependent intersections with braces (i.e. $[t_1, t_2]$), and projections with a dot followed by a numeral for the first or second projection (i.e. $t.1$ or $t.2$).

2.3 Equality and Top

The propositional equality type of Cedille internalizes the judgemental $\beta\eta$-conversion (modulo erasure) of theory, the rules of which are listed in Figure 3. Reflexive equalities are introduced with the β-axiom after a (potentially empty) series of rewrites (written with the Greek letter rho and a type guide to specify the position of the rewrite). The β-axiom allows for any well-scoped term to be used as the inhabitant of the equality. This, in combination with the fact that equality witnesses are erased from rewrites, makes the equality type effectively proof irrelevant. This has an additional consequence of allowing any trivially true equality type to be isomorphic to a top type (i.e. a type that contains all λ-terms, including non-terminating terms). We take advantage of this, defining the type Top as the type of proofs that $\lambda\, x.\, x$ is equal to itself. Additionally, the equality type has a strong form of the direct computation rule of [2], allowing a term's type to be changed to the type of another term if those two terms are provably equal. The direct computation rule is written with the Greek letter phi, typeset as φ.

The top type in particular may be considered controversial as it allows for any well-scoped term of the untyped lambda calculus to be well-typed, including the Y combinator and Ω. However, in our development of constructor subtyping a top type is integral.

$$\frac{\Gamma \vdash f : S \to T \quad \Gamma \vdash t : \Pi\, x{:}S.\, \{f\ x \simeq x\}}{\Gamma \vdash \mathsf{intrCast}\ \text{-}f\ \text{-}t : \mathsf{Cast} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash t : \mathsf{Cast} \cdot S \cdot T}{\Gamma \vdash \mathsf{cast}\ \text{-}t : S \to T}$$

$$|\mathsf{intrCast}\ \text{-}f\ \text{-}t| = \lambda\, x.\, x \qquad |\mathsf{cast}\ \text{-}t| = \lambda\, x.\, x$$

Figure 4: Type inclusions

Indeed, the interpretation of inductive datatypes as sets of uninterpreted constants foreshadows, in part, how Cedille is able to derive inductive datatypes that support constructor subtyping. Moreover, the consequence of allowing any term to be well-typed does not cause inconsistency of the logical theory of CDLE [27].

2.4 Type inclusions

Capitalizing CDLE's extrinsic typing, dependent intersections, and the direct computation law of the equality type, we may now summarize how type inclusions are defined (see [16] for more details). For all types S and T, Cast $\cdot S \cdot T$ is defined as the type all functions f which are provably equal to the identity function:

$$Cast \cdot A \cdot B = \iota\, f{:}A \to B.\, \{f \simeq \lambda\, x.\, x\}$$

For convenience, we present Cast axiomatically via a set of introduction, elimination, and erasure rules (Figure 4). The introduction form intrCast -f -t takes as (operationally irrelevant) arguments a function f of type S and T and a proof t that, for all terms $x : S$, $f\ x$ is provably equal to x. The direct computation rule φ provides the justification for this rule: functions of type $S \to T$ that are merely extensionally equal to $\lambda\, x.\, x$ can be used to prove that the latter *itself* has type $S \to T$. Operationally, using a witness t of the inclusion of a type S into T via the elimination form cast -t is then just an application of the identity function at type $S \to T$.

2.5 Other datatypes

Throughout the rest of the paper, we will treat as primitives the following basic datatypes. We assume a countably infinite set L of labels with decidable equality, whose elements are identifiers distinct from all variable names (e.g. lzero, lsucc, lpred, ...). We assume we have a finite product type (written $A \times B$) with projections fst and snd. Additionally, we left-associate products such that $S_1 \times \ldots \times S_n$ is equal to $S_1 \times (S_2 \times \ldots S_n) \ldots)$. For the case study on language extension for simply-typed lambda calculus, we assume lists and a function for testing list membership, *in*, returning boolean values (tt and ff). All such types, with corresponding recursion and induction principles, are derivable in CDLE (omitted, c.f. [26] for an explanation of the recipe for deriving datatypes with induction).

3 ORDER-INVARIANT LAMBDA ENCODINGS

We introduce a high-level syntax to both have a convenient syntax for defining the signature of an inductive datatype and to demonstrate how a syntax supporting constructor subtyping might look. The proposed syntax will follow a similar style found in many functional languages. For example, the type of natural numbers is defined:

$$\texttt{data}\ Nat : \star = zero : Nat \mid succ : Nat \to Nat$$

Constructor subtyping can be introduced with two operations: type extension and constructor equality constraints. Type extension allows a new datatype to be defined by extending a previously defined datatype with new constructors. For example, the type of integers defined by extending the type of natural numbers:

$$\texttt{data}\ Int\ \texttt{extends}\ Nat\ \texttt{with}\ pred : Int \to Int$$

With type extension, the type Int is defined with three constructors: $zero$, $succ$, and $pred$. Additionally, the corresponding constructors between Nat and Int are equal with respect to underlying equality of the theory. Critically, this definition makes Nat a subtype of Int. Thus, any function argument that accepts an Int value can also accept a Nat value.

Equality constraints on constructors allow for a more precise correspondence between constructors of datatypes. These constraints make the order the constructors appear in the type definition irrelevant and allow for only a subcollection of constructors to be shared. For example, a type of natural numbers with a unique zero but a shared successor in reverse order:

$$\begin{aligned}&\texttt{data}\ Nat1\ = succ : Nat1 \to Nat1 \mid one : Nat1\\&\texttt{where}\ Nat1.succ = Nat.succ\end{aligned}$$

Both of these type operations can be simulated by the other. A type extension between two types is simulated by constructor equality constraints by first defining the smaller type (in terms of constructor count) and then defining the larger type with constructor equality constraint for every constructor present in the smaller type. Constructor equality constraints between types is obtained by defining intermediate types for any shared subsignatures between the interacting types (e.g. a type with only a $succ$ constructor for Nat and $Nat1$).

3.1 Deriving Inductive Datatypes

Church encoded data is an appropriate testing grounds when deriving inductive datatypes. For this reason we begin with a refresher on Church encodings and describe why constructor subtyping fails for inductive data that are derived from this encoding. After, we discuss how the situation can be amended to support constructor subtyping but still with Church-style folds.

The Church encoding of an inductive datatype identifies an inductive datatype with its iteration scheme. Thus, the interpretations for the constructors of the corresponding datatype are encoded as an ordered list of arguments to that scheme. For example, the type of Church encoded natural numbers is

$$CNat = \forall\, X{:}\star.\ \underbrace{X}_{\text{zero}} \to \underbrace{(X \to X)}_{\text{successor}} \to X$$

where the first input interprets zero and the second input interprets successor. The constructors are then defined by returning the corresponding argument to the iteration scheme applied to the arguments of the constructor. For instance, the successor function is defined in the following way:

$$succ\ n = \Lambda X.\, \lambda\, z.\, \lambda\, s.\, s\ (n \cdot X\ z\ s) \qquad (1)$$

Suppose we wanted to define an integer type that was a supertype of the above defined natural number type. The naive approach would be to add a constructor for predecessor to the list:

$$CInt = \forall X{:}\star.\ \underbrace{X}_{\text{zero}} \to \underbrace{(X \to X)}_{\text{successor}} \to \underbrace{(X \to X)}_{\text{predecessor}} \to X$$

$$succ\ n = \Lambda X.\,\lambda z.\,\lambda s.\,\lambda\,\mathbf{p}.\,s\ (n \cdot X\ z\ s\ \mathbf{p}) \tag{2}$$

Unfortunately, this causes the definition of successor defined by (2) for the Church encoded integer type to be unequal to the natural number successor defined by (1) because of the additional lambda abstraction and application. Observe that constructors are being disambiguated by the order they appear in the Church encoded type definition. To implement constructor subtyping we need constructors to instead be disambiguated in an order-invariant way in the type signature.

With that in mind, we first pick a type to represent constructor labels (we use L introduced in Section 2) where the value of the label will disambiguate constructors. Second, we attempt to package the constructors for the type signature inside a function space with respect to this label type. A first attempt gives the following definition of a cs-natural number.

$$CSNat = \forall X{:}\star.\,(\mathsf{L} \to ?) \to X$$

However, there is no obvious way to split on the value of the label to select the correct type for a given constructor. With Cedilles equality type and a pair type we could specify the desired type for a given label constant:

$$\begin{aligned} CSNat =& \forall X{:}\star.\,(\Pi\,\ell{:}\mathsf{L}.\\ &\quad(\{\ell \simeq \mathsf{lzero}\} \to X)\\ &\quad\times(\{\ell \simeq \mathsf{lsucc}\} \to X \to X) \to X \end{aligned}$$

However, now the structure of the pair type is disambiguating the constructors instead of the value of the label. All we have accomplished is an uncurried form of the original Church encoding. To solve this problem we introduce a layer of indirection where we pick a supertype for all possible constructor type, the top type. Additionally, when the value of the label matches the constraint for a given constructor we add an erased type that specifies how to retype the computational content stored in Top at the desired constructor type. In a way, this layer of indirection separates constructors into three components: a disambiguating label, the computational content (an erased lambda term), and the *implicit* permission to use that computational content at a particular type. We are able to proceed because the number of constructors is finite — which allows for an enumeration of label value constraints. Before presenting the value derivation for $CSNat$ we first introduce two important abstractions that capture the core idea described above: a weak sigma type and a view type.

3.2 Weak Sigmas and Views

Sigma types are derived in Cedille like all other inductive datatypes, but the presence of implicit functions allows for a variation on dependent pairs where the second component is irrelevant. We call these variations weak sigmas (written $\sigma\,x : A.\,B$). They are presented axiomatically in Figure 5. Construction of a weak sigma is similar to that of ordinary sigma types, except that the term t_2

$$\frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B\ t_1}{\Gamma \vdash (t_1, \text{-}t_2) : \sigma\,x{:}A.\,B}$$

$$\frac{\Gamma \vdash t_1 : \sigma\,x{:}A.\,B \quad \Gamma, x{:}A, y{:}B\ x \vdash t_2 : T \quad y \notin FV(|t_2|)}{\Gamma \vdash \mathsf{unpack}\ t_1\ \mathsf{as}\ (x, \text{-}y)\ \mathsf{in}\ t_2 : T}$$

$$|(t_1, \text{-}t_2)| = \lambda\,f.\,f\ |t_1| \qquad \text{where } f \notin FV(|t_1|)$$
$$|\mathsf{unpack}\ t_1\ \mathsf{as}\ (x, \text{-}y)\ \mathsf{in}\ t_2| = |t_1|\ \lambda\,x.\,|t_2|$$

Figure 5: Weak Sigmas

$$\frac{\Gamma \vdash t_1 : \mathsf{Top} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t : \{t_1 \simeq t_2\}}{\Gamma \vdash \mathsf{intrView}\ t_1\ \text{-}t_2\ \text{-}t : View \cdot T\ t_1}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{Top} \quad \Gamma \vdash t_2 : \mathsf{View} \cdot T\ t_1}{\Gamma \vdash \mathsf{retype}\ t_1\ \text{-}t_2 : T}$$

$$|\mathsf{intrView}\ t_1\ \text{-}t_2\ \text{-}t| = |t_1| \qquad |\mathsf{retype}\ t_1\text{-}t_2| = |t_1|$$

Figure 6: Type views

does not occur in the erasure of the expression $(t_1, \text{-}t_2)$. Although a first projection for weak sigmas can be given, a second projection is not possible. Therefore, we give a positive presentation, with the elimination form unpack t_1 as $(x, \text{-}y)$ in t_2 extending the typing context with fresh variables corresponding to the components of the given $t : \sigma\,x : A.\,B\ x$, with the additional restriction that the second component y not occur free in the erasure of the body t_2.

Views represent an internalization of CDLE's extrinsic approach to typing, allowing to state as a proposition that an untyped term can be given a certain type. They are defined using dependent intersections and equality, shown below:

$$\mathsf{View} \cdot A\ t = \iota\,z{:}A.\,\{z \simeq t\}$$

where A is a type and t is a term of type Top. The axiomatic presentation is given in Figure 6.

It may seem counter-intuitive that a witness of View $\cdot\,A\ t$ should contain a term of type A when thinking of views as a separation of typing information from terms. The situation is illuminated when considering how such witnesses are constructed and used.

We define the introduction form intrView as:

$$\mathsf{intrView} \cdot A\ t\ \text{-}a\ \text{-}eq = [\varphi\ eq - a\ \{t\}, \beta\{t\}]$$

Notice that the direct computation rule is used to ascribe to the term t the type A of a where only the term t remains computational relevant. Thus, the value a at type A need only be supplied implicitly when constructing a view.

Witnesses v of View $\cdot\,A\ t$ are proof-irrelevant, provided the term t is safe to occur in operationally relevant positions (i.e., t contains no free variables under erasure restrictions). This is demonstrated by the elimination for retype $\cdot\,A\ t\text{-}v$ which produces a term of type A that is definitionally equal to t. It is defined as

$$\mathsf{retype} \cdot A\ t\ \text{-}v = \varphi\ v.2 - v.1\{t\}$$

where A is a type, t is a term of type Top, and v is a view of t at A. Again, we are only required to know the value of t relevantly, with the view witness only required implicitly.

3.3 Finishing the Encoding

Now, with weak sigmas and views defined, we can derive $CSNat$ as suggested:

$$\begin{aligned} CSNatPack \cdot X\ \ell\ t &= (\{\ell \simeq \mathsf{lzero}\} \to \mathsf{View} \cdot X\ t) \\ &\quad \times (\{\ell \simeq \mathsf{lsucc}\} \to \mathsf{View} \cdot (X \to X)\ t) \\ CSNat &= \forall X{:}\star.\,(\Pi\,\ell{:}\mathsf{L}.\,\sigma\,t{:}\mathsf{Top}.\,CSNatPack \cdot X\ \ell\ t) \to X \end{aligned}$$

The layer of indirection is implemented using the weak sigma type which contains the computationally relevant information in the first component under the type Top and the view information in the implicit second component. The type permission information is implemented almost exactly as the uncurried church encoding except instead of obtaining the constructors explicitly we obtain permission to view the computational content at the constructor type. Notice that we have split the definition into two parts, a type representing the constructor type permissions (as well as the assignment of labels to a given constructor) and a type representing the layer of indirection between the computational content and type views. This scheme generalizes: the packaging type is any type of nested pairs where the components are functions spaces between a label constraint and a view of the computational data at the desired type. Thus, any type can be defined in the same way with only the packaging type changed.

Now the disambiguation of constructors is out of the way of allowing casts between inductive datatypes with a different number of constructors. However, there are two outstanding questions about this definition: what is the overhead of the packaging type and how should labels be assigned to constructors.

First, the packaging, via a weak sigma type, only imposes one layer of indirection in the definition. However, it does require unpacking the constructors when performing a fold. This unpacking function can always be defined as a sequence of equality comparisons on the occurring labels. With this implementation the cost is the same as the number of constructors times the cost of comparing labels for equality. In our formalization, as a matter of convenience, we use a natural number type to represent labels which has a linear time cost to compare for equality.

This means that our choice of a label type can not be made lightly. With lambda encoded data the cost of equality can be made logarithmic by using a tree structure for the label types, but in a more mature language (such as Idris, Agda, or Coq) the standard natural number type may be internally represented more efficiently. Additionally, a type representing bits may also be present which would be the ideal label type (assuming that a limit of 2^{64} constructors is acceptable). If we assume that the cost of equality between labels is a `memcmp` between bits then the cost of unpacking is the same as the number of constructors for that datatype.

Second, the assignment of labels has a few variations that have different trade-offs. The obvious assignment is to give every constructor a unique label *except* when the user specifies when two constructors should be equal. Another variation is to assign labels based off the order of constructors. This is precisely how subtyping between datatypes works in Cedille as of version 1.1.2, and why there is zero-cost reuse between certain inductive datatypes with the same inductive structure [9]. There is no, in the authors' opinion, best choice about how labels should be assigned to constructors as long as the selected method is coherent and predictable.

4 CASE STUDIES

4.1 Naturals and Integers

We return to our recurring example as a warmup for the exposition of our proposed encoding. Using the same definition of $CSNat$ found in the last section we are able to define the successor constructor by picking the correct label, unpacking. and applying the type information.

$$\begin{aligned} succ\ n =& \Lambda X.\,\lambda f.\,\mathsf{unpack}\ (f\ \mathsf{lsucc})\ \mathsf{as}\ (t_1, \text{-}t_2)\ \mathsf{in} \\ &\quad \mathsf{let}\ s = \mathsf{retype}\ t_1\ \text{-}(\underline{\mathsf{snd}\ t_2\ \beta}) \\ &\quad \mathsf{in}\ s\ (n\ f) \end{aligned} \tag{3}$$

Note that the underlined expression is erased from the definition.

A $CSInt$ type can be defined merely by changing the packaging type.

$$\begin{aligned} CSIntPack \cdot X\ \ell\ t &= (\{\ell \simeq \mathsf{lzero}\} \to \mathsf{View} \cdot X\ t) \\ &\quad \times (\{\ell \simeq \mathsf{lsucc}\} \to \mathsf{View} \cdot (X \to X)\ t) \\ &\quad \times (\{\ell \simeq \mathsf{lpred}\} \to \mathsf{View} \cdot (X \to X)\ t) \\ CSInt &= \forall X{:}\star.\,(\Pi\,\ell{:}\mathsf{L}.\,\sigma\,t{:}\mathsf{Top}.\,CSIntPack \cdot X\ \ell\ t) \to X \end{aligned}$$

Moreover, the definition of successor is exactly the same *except* how the type information in the weak sigma is extracted.

$$\begin{aligned} succ\ n =& \Lambda X.\,\lambda f.\,\mathsf{unpack}\ (f\ \mathsf{lsucc})\ \mathsf{as}\ (t_1, \text{-}t_2)\ \mathsf{in} \\ &\quad \mathsf{let}\ s = \mathsf{retype}\ t_1\ \text{-}(\underline{\mathsf{snd}\ (\mathsf{fst}\ t_2)\ \beta}) \\ &\quad \mathsf{in}\ s\ (n\ f) \end{aligned} \tag{4}$$

Like the definition in (3) the underlined section in (4) is erased, but the two definitions are identical otherwise! Therefore, the erasures of these definitions are α-convertible. As an aside, the predecessor function is of course unequal because the associated label, which is computationally relevant in the constructor, is different from any other label used in the definition of $CSNat$.

4.2 Lists and Vector Trees

Zero-cost reuse between lists and vectors is already possible in the current version (1.1.2) of Cedille [9]. However, the direction of reuse from lists to vectors requires a dependent form of casts which demonstrates additional difficulties that may arise with defining constructor subtyping directly as done by Barthe [4]. Moreover, when defining a list type using the general approached previously described it is trivial to prove that the nil constructor for lists are *always* equal regardless of the parameterized type. Unlike in Barthes developments, type applications do not get in the way of equalities between terms.

We define lists and vector tree using the higher level syntax introduced at the beginning of Section 3.

$$\mathtt{data}\ List\ (A:\star) : \star = nil : List \mid cons : A \to List \to List$$

To handle type parameters the packaging type of the order-invariant lambda encoding must take an additional type argument. There is no other changes to the general scheme aside from adding the type parameter as an input to the type kind as is standard in CoC for parameterized types.

A vector type is a length indexed list and because the length value can be treated as erased it is clear from both prior work on re-use and from earlier developments in this work that the constructors for the types will be equal. To add a constructor subtyping flair to the example we consider a vector-tree type which is a regular vector type extended with a branching constructor:

$$\begin{array}{l}
\texttt{data}\ \mathit{VecTree}\ (A:\star) : \mathbb{N} \to \star = \\
\mid \mathit{nil} : \mathit{VecTree}\ 0 \\
\mid \mathit{cons} : \forall\, n{:}\mathbb{N}.\, A \to \mathit{VecTree}\ n \to \mathit{VecTree}\ (n+1) \\
\mid \mathit{branch} : \forall\, a, b{:}\mathbb{N}.\, \mathit{VecTree}\ a \to \mathit{VecTree}\ b \to \mathit{VecTree}\ (a+b) \\
\texttt{where}\ \mathit{List.nil} = \mathit{VecTree.nil} \mid \mathit{List.cons} = \mathit{VecTree.cons}
\end{array}$$

From an implementation perspective there are two important considerations. First, it is not clear if the extension high-level syntax will work in situations where a type is being extended to an indexed type. In particular, there is no clear choice of how the indexed values of the *cons* constructor for *VecTree* should be instantiated via a direct extension. Thus, it seems that the constructor equality constraint syntax may be better suited for general definitions of inductive datatypes with constructor subtyping. Second, how is the implementation to decide when the constructor equality constraints are true or false. In the case of Cedille the solution is simple. The derivation of the inductive type, indexed or not, is rote. Once the two inductive datatypes are independently derived the equality constraints on constructors need only be forced up-to definitional equivalence, by checking that the two constructors are already $\beta\eta$-convertible without rewrites.

To prove that there is a cast from *List* to *VecTree* we need a dependent variant of type inclusion.

$$\mathrm{DepCast} \cdot A \cdot B = \iota\, f{:}\Pi\, a{:}A.\, B\ a.\, \{f \simeq \lambda\, x.\, x\}$$

Now the desired cast property can be expressed in the following way,

$$\mathrm{DepCast} \cdot (\mathit{List} \cdot A) \cdot (\lambda\, l.\, \mathit{VecTree} \cdot A\ (\mathit{length}\ l))$$

where *length* is a function computing the length of a *List*.

4.3 Language Extensions

In this subsection we study yet another example of indexed inductive datatypes. In particular, we consider an indexed inductive datatype encoding the simply typed λ-calculus and an indexed inductive datatype encoding an extension of that calculus with numerals and addition.

To derive an inductive type encoding the simply typed λ-calculus we first define an auxiliary type encoding the internal types with two constructors,

$$\texttt{data}\ \mathit{Typ} : \star = \mathit{base} : \mathit{Typ} \mid \mathit{arr} : \mathit{Typ} \to \mathit{Typ} \to \mathit{Typ}$$

Now we are able to define the simply typed λ-calculus:

$$\begin{array}{l}
\texttt{data}\ \mathit{Stlc} : \mathit{List} \cdot \mathit{Typ} \to \mathit{Typ} \to \star = \\
\mid \mathit{var} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{Typ}.\, \Pi\, T{:}\mathit{Typ}. \\
\qquad \{\mathit{in}\ \Gamma\ T\ \simeq\ \mathsf{tt}\} \Rightarrow \mathbb{N} \to \mathit{Stlc}\ \Gamma\ T \\
\mid \mathit{abs} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{Typ}.\, \Pi\, A{:}\mathit{Typ}.\, \Pi\, B{:}\mathit{Typ}. \\
\qquad \mathit{Stlc}\ (\mathit{cons}\ A\ \Gamma)\ B \to \mathit{Stlc}\ \Gamma\ (\mathit{arr}\ A\ B) \\
\mid \mathit{app} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{Typ}.\, \Pi\, A{:}\mathit{Typ}.\, \Pi\, B{:}\mathit{Typ}. \\
\qquad \mathit{Stlc}\ \Gamma\ (\mathit{arr}\ A\ B) \to \mathit{Stlc}\ \Gamma\ A \to \mathit{Stlc}\ \Gamma\ B
\end{array}$$

In order to extend this language with numerals *Typ* must first be extended with an encoded type of numerals:

$$\texttt{data}\ \mathit{ETyp}\ \texttt{extends}\ \mathit{Typ}\ \texttt{with}\ \mathit{nat} : \mathit{ETyp}$$

where *ETyp* stands for "extended-*Typ*". Because extension yields a type inclusion by construction we have $\mathit{Cast} \cdot \mathit{Typ} \cdot \mathit{ETyp}$. Finally, we extend *Stlc* with two constructors for numerals and a primitive addition function.

$$\begin{array}{l}
\texttt{data}\ \mathit{EStlc} : \mathit{List} \cdot \mathit{ETyp} \to \mathit{ETyp} \to \star = \\
\mid \mathit{var} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{ETyp}.\, \Pi\, T{:}\mathit{ETyp}. \\
\qquad \{\mathit{in}\ \Gamma\ T\ \simeq\ \mathsf{tt}\} \Rightarrow \mathbb{N} \to \mathit{EStlc}\ \Gamma\ T \\
\mid \mathit{abs} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{ETyp}.\, \Pi\, A{:}\mathit{ETyp}.\, \Pi\, B{:}\mathit{ETyp}. \\
\qquad \mathit{EStlc}\ (\mathit{cons}\ A\ \Gamma)\ B \to \mathit{EStlc}\ \Gamma\ (\mathit{arr}\ A\ B) \\
\mid \mathit{app} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{ETyp}.\, \Pi\, A{:}\mathit{ETyp}.\, \Pi\, B{:}\mathit{ETyp}. \\
\qquad \mathit{EStlc}\ \Gamma\ (\mathit{arr}\ A\ B) \to \mathit{EStlc}\ \Gamma\ A \to \mathit{EStlc}\ \Gamma\ B \\
\mid \mathit{num} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{ETyp}.\, \mathbb{N} \to \mathit{EStlc}\ \Gamma\ \mathit{nat} \\
\mid \mathit{add} : \Pi\, \Gamma{:}\mathit{List} \cdot \mathit{ETyp}. \\
\qquad \mathit{EStlc}\ \Gamma\ \mathit{nat} \to \mathit{EStlc}\ \Gamma\ \mathit{nat} \to \mathit{EStlc}\ \Gamma\ \mathit{nat} \\
\texttt{where}\ \mathit{Stlc.var} = \mathit{EStlc.var} \\
\mid \mathit{Stlc.abs} = \mathit{EStlc.abs} \\
\mid \mathit{Stlc.app} = \mathit{EStlc.app}
\end{array}$$

Notice that every occurrence of *Typ* in the definition of *Stlc* is replaced instead with the more general type *ETyp*. This allows for numeral abstractions and higher-order numeral functions as expected in an extension to the simply typed λ-calculus.

Moreover, there is a cast between *Stlc* and *EStlc* because there is a cast between *Type* and *ETyp* and because every constructor of *Stlc* is accounted for in *EStlc*. The constructors are necessary to ensure the shape of the types are compatible, and the cast between *Typ* and *ETyp* is necessary to show that the constructor types form a cast as well between *Stlc* and *EStlc*. As long as all constructors form a cast between their respective types (in the correct direction) the types themselves will also form a cast.

5 GENERIC SUBTYPING FOR INDUCTIVE DATATYPES

In this section, we instantiate the efficient generic impredicative encoding of inductive datatypes by Firsov et al. [11] with the scheme we have proposed for defining datatype signatures to support constructor subtyping. We start with a review of this development (for brevity presented axiomatically as a set of type formation, introduction, and elimination rules), including a natural definition of covariance of a type scheme in terms of type inclusions (casts). We then discuss the generalization of the notion of covariance of F to the containment of F in another signature G [1, 15] (c.f. [9] for a formulation using indexed types and type inclusions). If two covariant type schemes F and G are in this containment relation, it follows that the datatype μF is a subtype of μG. Finally, we instantiate F and G to the general shape of our proposed encoding of signatures, $X \mapsto \Sigma\, a{:}A.\, \sigma\, t{:}\mathsf{Top}.\, B \cdot X\ a\ t$ (where A is the labeling type and B is the type family of constructor argument types), and give a sufficient condition for when two signatures of this shape are in the containment relation.

5.1 Review: Generic Mendler-style encoding

The definitions from [11] we use for our generic result, listed in Figure 7, are: μF, the datatype given by the signature F; in, the generic datatype constructor; Mono, the property of type schemes that they are covariant with respect to the chosen subtyping relation; and induction, the generic Mendler-style induction scheme.

The generic constructor and monotonicity. The datatype μF can be understood as the least fixedpoint of the type scheme F (a result known as *Lambek's lemma* [18]). It is well-known that unrestricted fixedpoint types in type theory lead to non-termination and inconsistency (when the theory is interpreted as a logic under the *Curry-Howard correspondence*), c.f. [20]. To avoid such issues, the formation, introduction, or elimination of fixedpoint types must be restricted somehow. Here, a restriction occurs in the introduction form in: when t is an F-collection of μF predecessors, and m is a proof that F is covariant, then in $\text{-}m\ t$ constructs a successor value of type μF. The type of m, $\mathsf{Mono} \cdot F$, is the definition of monotonicity in the partial order whose underlying set is the set of CDLE types and whose ordering relation is type inclusions, Cast.

Mendler-style induction. Mendler-style inductive types, first proposed by Mendler in [20, 21], provides an alternative to the conventional initial F-algebra semantics for inductive types for positive F (c.f. [28] for the categorical account). Roughly, the key difference between the conventional and Mendler-style formulation is the latter introduces higher-rank polymorphism and higher-order functions. Starting simply, compare the Mendler-style encoding of naturals below to the familiar Church encoding:

$$\mathsf{MNat} = \forall\, X{:}\star.\, X \to \underbrace{(\forall\, R{:}\star.\, (R \to X) \to R \to X)}_{\text{successor}} \to X$$

The intended reading is that the universally quantified type variable R "stands in" for recursive occurrences of the type MNat itself; thus, the interpretation of the successor function is as a polymorphic higher-order function taking a handle for making recursive calls ($R \to X$) on predecessors, a given predecessor of type R, and must return the appropriate result for the successor. The polymorphic typing ensures that the interpretation of successor *cannot make recursive calls on arbitrary terms of type MNat*, and helps to explain why Mendler-style recursion schemes are guaranteed to be terminating when general recursion (which has a similar shape) is not.

For the typing rule of induction, the Mendler-style is generalized further to dependent types. Given a type scheme F whose covariance is witnesses by m, a property $P : \mu F \to \star$ over the datatype, and a term t whose type is read as:

- *for all types R and witnesses c of a type inclusion of R into μF;* (we may think of R as a subtype of μF containing only the predecessors of the value being analyzed)
- *and assuming an inductive hypothesis stating that P holds for every term of type R (after inclusion into μF)*
- *given $xs : F \cdot R$, an F-collection of R predecessors, show that P holds for the value constructed from in of xs, after using monotonicity of F to include xs into the type $F \cdot \mu F$*

5.2 Signature containment

We now state a precise definition of the signature containment relation SigSub for first-order datatype signatures. This definition is a special case of a more general notion of containment used by Hinze [15] and Abel et al. [1] for higher-order schemes, formulated in terms of type inclusions.

Definition 5.1 (Signature containment). Given two type schemes F and G, we say that F *is contained in* G iff there is a witness of $\mathsf{SigSub} \cdot F \cdot G$, defined as

$$\mathsf{SigSub} \cdot F \cdot G = \forall\, X{:}\star.\, \forall\, Y{:}\star.\, \mathsf{Cast} \cdot X \cdot Y \to \mathsf{Cast} \cdot (F \cdot X) \cdot (G \cdot Y)$$

The signature containment relation is sufficient for establishing $\mathsf{Cast} \cdot \mu F \cdot \mu G$ for covariant F and G.

Theorem 5.2. *For two covariant type schemes $F, G : \star \to \star$, if $\mathsf{SigSub} \cdot F \cdot G$ then $\mathsf{Cast} \cdot \mu F \cdot \mu G$.*

Proof. The proof is formalized in Cedille in the code repository associated with this paper. It comes as a direct consequence of the reuse combinator `ifix2fix` of Diehl et al. [9] (their `Id` and `IdMapping` are equivalent to our Cast and Mono). □

5.3 Generic constructor packing

We now generalize our scheme for defining datatype signatures supporting flexible constructor subtyping so that we may instantiate the generic framework of Firsov et al.

Definition 5.3 (Sig, the generic datatype signature). Given $A : \star$ (the labeling type) and a type family $B : \star \to A \to \mathsf{Top} \to \star$, we define the type family of constructor arguments indexed over labels $a : A$, CtArgs, and the generic datatype signature, Sig, below as

$$\begin{aligned} \mathsf{CtArgs} \cdot A \cdot B &= \lambda\, R{:}\star.\, \lambda\, a{:}A.\, \sigma\, t{:}\mathsf{Top}.\, B \cdot R\ a\ t \\ \mathsf{Sig} \cdot A \cdot B &= \lambda\, R{:}\star.\, \Sigma\, a{:}A.\, \mathsf{CtArgs} \cdot A \cdot B \cdot R\ a \end{aligned}$$

The type family Sig over-generalizes the signatures of datatypes CSNat and CSInt: we do not assume that A is finite (even though the set of constructors for any datatype is), nor that B contains a finite set of conditional typing information for its Top argument. As discussed in Section 3.1, this is due to the lack of large eliminations in CDLE; instead, these requirements would be treated schematically in a specification of the elaboration of syntax for datatype declarations to impredicative encodings.

$$\frac{F : \star \to \star}{\mu F : \star} \qquad \frac{F : \star \to \star \quad m : \mathsf{Mono} \cdot F \quad t : F \cdot \mu F}{\mathsf{in}\ \text{-}m\ t : \mu F} \qquad \mathsf{Mono} \cdot F = \forall\, X{:}\star.\, \forall\, Y{:}\star.\, \mathsf{Cast} \cdot X \cdot Y \to \mathsf{Cast} \cdot (F \cdot X) \cdot (F \cdot Y)$$

$$\frac{F : \star \to \star \quad m : \mathsf{Mono} \cdot F \quad P : \mu F \to \star \quad t : \forall\, R{:}\star.\, \forall\, c{:}\mathsf{Cast} \cdot R \cdot \mu F.\, (\Pi\, x{:}R.\, P\ (\mathsf{cast}\ \text{-}c\ x)) \to \Pi\, xs{:}F \cdot R.\, P\ (\mathsf{in}\ \text{-}m\ (\mathsf{cast}\ (m\ c)\ xs))}{\mathsf{induction}\ \text{-}m\ t : \Pi\, x{:}\mu F.\, P\ x}$$

Figure 7: Interface for the generic encoding of Firsov et al. [11]

Next, in order to use the generic framework we must also establish that $\mathsf{Sig} \cdot A \cdot B$ is covariant. We have this when B is covariant in its type argument.

Lemma 5.4 (Covariance of $\mathsf{Sig} \cdot A \cdot B$). *Assume $A : \star$ and $B : \star \to A \to \mathit{Top} \to \star$. If, for all $a : A$ and $t : \mathit{Top}$, the type scheme $\lambda\, R{:}\star.\, B \cdot R\ a\ t$ is Mono, then so is $\mathsf{Sig} \cdot A \cdot B$.*

Proof. The proof is straightforward, as both Σ and σ are positive type constructors. See the Cedille code repository associated with this paper for details. □

5.4 Signature containment for Sig

The main result of this section is a sufficient condition for signature containment for type schemes defined using Sig. This, in combination with Thm. 5.2, in turns gives a sufficient condition for when datatypes whose signatures are given using Sig are in the subtyping relation.

Theorem 5.5. *Assume labeling types $A_1, A_2 : \star$ and branch type families $B_1 : \star \to A_1 \to \mathit{Top} \to \star$ and $B_2 : \star \to A_2 \to \mathit{Top} \to \star$ that are covariant in their resp. type arguments. If*

- *A_1 is a subtype of A_2, witnessed by c*
- *and for all $a_1 : A_1$ and $R : \star$, $\mathit{CtArgs} \cdot A_1 \cdot B_1 \cdot R\ a_1$ is a subtype of $\mathit{CtArgs} \cdot A_2 \cdot B_2 \cdot R\ (\mathit{cast}\ \text{-}c\ a_1)$, witnessed by $d_{(a_1,R)}$*

Then $\mathit{Sig} \cdot A_1 \cdot B_1$ and $\mathit{Sig} \cdot A_2 \cdot B_2$ are in the signature containment relation SigSub (Def. 5.1)

Proof. The proof is formalized in Cedille in the code repository associated with this paper; we give a corresponding proof in prose. We assume

- a witness $c : \mathsf{Cast} \cdot A_1 \cdot A_2$
- a family of witnesses $d_{(R,a_1)} : \mathsf{Cast} \cdot (\mathsf{CtArgs} \cdot A_1 \cdot B_1 \cdot R\ a_1) \cdot (\mathsf{CtArgs} \cdot A_2 \cdot B_2 \cdot R\ (\mathsf{cast}\ \text{-}c\ a_1))$ over all $R : \star$ and $a_1 : A_1$.
- arbitrary types X and Y, where $c' : \mathsf{Cast} \cdot X \cdot Y$ (Def. 5.1)

We must produce a proof of an inclusion of $\mathsf{Sig} \cdot A_1 \cdot B_1 \cdot X$ into $\mathsf{Sig} \cdot A_2 \cdot B_2 \cdot Y$, for which it suffices to show that there exists a function f between the two types such that, for all $t : \mathsf{Sig} \cdot A_1 \cdot B_1 \cdot X$, $f\ t$ is equal to t.

We define this function by induction on the given $\mathsf{Sig} \cdot A_1 \cdot B_1 \cdot X$ (whose outermost type constructor is Σ). We assume arbitrary $a_1 : A_1$ and $w : \mathsf{CtArgs} \cdot A_1 \cdot B_1 \cdot X\ a_1$ and must exhibit some $y : \mathsf{Sig} \cdot A_2 \cdot B_2 \cdot Y$ such that $(|a_1|, |w_1|) = |y|$.

Now, as an intermediate step we can prove the inclusion of $\mathsf{CtArgs} \cdot A_1 \cdot B_1 \cdot X\ a_1$ into $\mathsf{CtArgs} \cdot A_1 \cdot B_1 \cdot Y\ a_1$. Assuming an arbitrary w_1' of the first type (whose outermost type constructor is σ), we proceed by induction: assume arbitrary $t : \mathsf{Top}$ and an operationally irrelevant $b : B_1 \cdot X\ a_1\ t$. Appealing to monotonicity of B_1 and the assumed witness c', we have that the type of b is also $B_1 \cdot Y\ a_t\ t$. Produce the pair $(t, \text{-}b) : \sigma\, x : \mathsf{Top}.\, B_1 \cdot Y\ a_1\ x$, which is equal to the given weak pair (and b does not occur in an operationally relevant position).

Apply the inclusion c on a_1. Apply both the above derived type inclusion, then assumed inclusion $d_{(a_1,Y)}$ on w_1, to get $w_1 : \mathsf{CtArgs} \cdot A_2 \cdot B_2 \cdot Y\ (\mathsf{cast}\ \text{-}c\ a_1)$. We conclude by returning the pair $(a_1, w_1) : \mathsf{Sig} \cdot A_2 \cdot B_2 \cdot Y$, which is equal to the given pair. □

5.5 Example: naturals and integers

We now return to our earlier motivating example of the inclusion of natural numbers into integers to demonstrate the use of the generic development. Unlike the earlier formulation in which the constructors themselves were packaged together in a type family, the generic framework of Firsov et al. [11] provides a single generic constructor in, so we pack together just the different possible argument types.

For natural numbers, we have

$$\begin{aligned}
\mathsf{GNatPack} \cdot R\ l\ t =& (\{l \simeq \mathsf{lzero}\} \to \mathsf{View} \cdot \mathsf{Unit}\ t) \\
& \times (\{l \simeq \mathsf{lsucc}\} \to \mathsf{View} \cdot R\ t) \\
& \times (\neg \cdot \{l \simeq \mathsf{lzero}\} \times \neg \cdot \{l \simeq \mathsf{lsucc}\} \to \bot) \\
\mathsf{GNatSig} =& \mathsf{Sig} \cdot \mathsf{L} \cdot \mathsf{GNatPack} \\
\mathsf{GNat} =& \mu \mathsf{GNatSig}
\end{aligned}$$

where Unit is the singleton type, $\bot$ is the empty type, and $\neg \cdot T = T \to \bot$ – in the generic formulation, we require an additional explicit constraint that there are no other constructors. Also, it is clear that GNatPack is positive in its first type argument.

For integers

$$\begin{aligned}
\mathsf{GIntPack} \cdot R\ l\ t =& (\{l \simeq \mathsf{lzeroc}\} \to \mathsf{View} \cdot \mathsf{Unit}\ t) \\
& \times (\{l \simeq \mathsf{lsucc}\} \to \mathsf{View} \cdot R\ t) \\
& \times (\{l \simeq \mathsf{lpred}\} \to \mathsf{View} \cdot R\ t) \\
& \times (\neg \cdot \{l \simeq \mathsf{lzero}\} \times \neg \cdot \{l \simeq \mathsf{lsucc}\} \times \neg \cdot \{l \simeq \mathsf{lpred}\} \\
& \quad \to \bot) \\
\mathsf{GIntSig} =& \mathsf{Sig} \cdot \mathsf{L} \cdot \mathsf{GIntPack} \\
\mathsf{GInt} =& \mu \mathsf{GIntSig}
\end{aligned}$$

where again GIntPack is positive in its first argument.

Proposition 5.6. *There exists a cast from GNat to GInt*

Proof. The proof is formalized in the code repository associated with this paper. By Thms. 5.5 and 5.2, it suffices to show an inclusion of $\mathsf{CtArgs} \cdot \mathsf{L} \cdot \mathsf{GIntPack} \cdot R\ l$ into $\mathsf{CtArgs} \cdot \mathsf{L} \cdot \mathsf{GNatSig} \cdot R\ l$ for all $l : \mathsf{L}$ and $R : \star$ (notice that this does **not** require that a similar inclusion hold for GNatPack and GIntPack). This is given by a straightforward proof by cases on the label l for the assumed weak pair $(t, -(z, s, e_\perp))$

Case l = lzero: we have $(t, -(z, s, e'_\perp, e''_\perp))$: CtArgs·L·GIntPack·R lzero, where

- $e'_\perp$: {lzero ≃ lpred} ⟶ View · R t and
- $e''_\perp$: ¬ · {lzero ≃ lzero} × ¬ · {lzero ≃ lsucc} × ¬ · {lzero ≃ lpred} ⟶ ⊥

Case l = lsucc: we have $(t, -(z, s, e'_\perp, e''_\perp))$: CtArgs·L·GIntPack·R lsucc, where

- $e'_\perp$: {lsucc ≃ lpred} ⟶ View · R t, and
- $e''_\perp$: ¬ · {lsucc ≃ lzero} × ¬ · {lsucc ≃ lsucc} × ¬ · {lsucc ≃ lpred} ⟶ ⊥

Otherwise: impossible (from $e_\perp$ we have a proof of ⊥).

Recall that the second component of weak pairs are operationally erased, so in each case above the produced weak pair is equal to the assumed one. □

6 RELATED WORK

As previously mentioned, calculi with constructor subtyping and other desirable properties have been developed and explored by Barthe [3, 4]. However, there are many other approaches to subtyping that could enable similar features (i.e. function overloading) such as coercive subtyping [19, 29], algebraic subtyping [10], and semantic subtyping [5, 12, 13] to name a few. Research in Object Oriented Programming (OOP) has also extensively explored the idea of method overloading [6, 7, 24]. Indeed, method overloading is a common feature of almost all industry OOP languages. Ornaments have been used for proof reuse of inductive datatypes in Coq although they require the same inductive structure [23]. To the authors' knowledge there are no results about ornaments with respect to subtyping inductive datatypes with shared inductive substructure.

7 CONCLUSIONS

In this paper we have devised a way to derive inductive datatypes that support constructor subtyping where the subtyping relation is a cast. In particular, using casts as the subtyping relation allows for computationally efficient promotion of types and program reuse. We also proved that a similar technique does not work for subtyping of records. Beyond our derivation, we explored function overloading using constructor subtyping as originally proposed by Barthe and an indexed datatype example demonstrating language extension. Additionally, all of our developments and examples have been formalized in the Cedille programming language.

REFERENCES

[1] Andreas Abel, Ralph Matthes, and Tarmo Uustalu. 2005. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.* 333, 1-2 (2005), 3–66. https://doi.org/10.1016/j.tcs.2004.10.017

[2] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. 2006. Innovations in computational type theory using Nuprl. *J. Applied Logic* 4, 4 (2006), 428–469. https://doi.org/10.1016/j.jal.2005.10.005

[3] Gilles Barthe and Maria João Frade. 1999. Constructor subtyping. In *European Symposium on Programming*. Springer, 109–127.

[4] Gilles Barthe and Femke Van Raamsdonk. 2000. Constructor subtyping in the calculus of inductive constructions. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 17–34.

[5] Giuseppe Castagna and Alain Frisch. 2005. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 198–199.

[6] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A calculus for overloaded functions with subtyping. *Information and Computation* 117, 1 (1995), 115–135.

[7] Daniel KC Chan and Philip W Trinder. 1994. An object-oriented data model supporting multi-methods, multiple inheritance, and static type checking: A specification in Z. In *Z User Workshop, Cambridge 1994*. Springer, 297–315.

[8] Thierry Coquand. 1992. Pattern matching with dependent types. In *Informal proceedings of Logical Frameworks*, Vol. 92. Citeseer, 66–79.

[9] Larry Diehl, Denis Firsov, and Aaron Stump. 2018. Generic zero-cost reuse for dependent types. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–30.

[10] Stephen Dolan. 2017. *Algebraic subtyping*. BCS, The Chartered Institute for IT.

[11] Denis Firsov, Richard Blair, and Aaron Stump. 2018. Efficient Mendler-Style Lambda-Encodings in Cedille. In *International Conference on Interactive Theorem Proving*. Springer, 235–252.

[12] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2002. Semantic subtyping. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 137–146.

[13] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)* 55, 4 (2008), 1–64.

[14] Herman Geuvers. 2001. Induction Is Not Derivable in Second Order Dependent Type Theory. In *International Conference on Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer, Berlin, Heidelberg, 166–181.

[15] Ralf Hinze. 2002. Polytypic values possess polykinded types. *Sci. Comput. Program.* 43, 2-3 (2002), 129–159. https://doi.org/10.1016/S0167-6423(02)00025-4

[16] Christopher Jenkins and Aaron Stump. 2020. Monotone recursive types and recursive data representations in Cedille. arXiv:cs.PL/2001.02828 Under consideration for publication in J. Mathematically Structured Computer Science.

[17] Alexei Kopylov. 2003. Dependent Intersection: A New Way of Defining Records in Type Theory. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS '03)*. IEEE Computer Society, Washington, DC, USA, 86–.

[18] Joachim Lambek. 1968. A Fixpoint Theorem for Complete Categories. *Mathematische Zeitschrift* 103, 2 (1968), 151–161. https://doi.org/10.1007/bf01110627

[19] Zhaohui Luo. 1999. Coercive subtyping. *Journal of Logic and Computation* 9, 1 (1999), 105–130.

[20] N. P. Mendler. 1987. Recursive Types and Type Constraints in Second-Order Lambda Calculus. In *Proceedings of the Symposium on Logic in Computer Science ((LICS '87))*. IEEE Computer Society, Los Alamitos, CA, 30–36.

[21] Nax Paul Mendler. 1991. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic* 51, 1 (1991), 159 – 172. https://doi.org/10.1016/0168-0072(91)90069-X

[22] Alexandre Miquel. 2001. The Implicit Calculus of Constructions: Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications (TLCA'01)*. Springer-Verlag, Berlin, Heidelberg, 344–359.

[23] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2019. Ornaments for proof reuse in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[24] Francois Rouaix. 1989. Safe run-time overloading. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 355–366.

[25] Aaron Stump. 2017. The calculus of dependent lambda eliminations. *Journal of Functional Programming* 27 (2017), e14.

[26] Aaron Stump. 2018. From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic* 169, 7 (2018), 637–655. https://doi.org/10.1016/j.apal.2018.03.002

[27] Aaron Stump. 2018. Syntax and Semantics of Cedille. arXiv:cs.PL/1806.04709 https://arxiv.org/abs/1806.04709

[28] Tarmo Uustalu and Varmo Vene. 1999. Mendler-style Inductive Types, Categorically. *Nordic Journal of Computing* 6, 3 (Sep 1999), 343–361. http://dl.acm.org/citation.cfm?id=774455.774462

[29] Tao Xue. 2013. *Theory and implementation of coercive subtyping*. Ph.D. Dissertation. Royal Holloway, University of London, UK.

Heuristics-based Type Error Diagnosis for Haskell

The case of GADTs and local reasoning

Joris Burgers
Dept. of Information and Computing Sciences
Utrecht University
jorisburgers@live.nl

Jurriaan Hage
Dept. of Information and Computing Sciences
Utrecht University
j.hage@uu.nl

Alejandro Serrano
47 Degrees/Utrecht University
a.serranomena@uu.nl

Abstract

Helium is a Haskell compiler designed to provide programmer friendly type error messages. It employs specially designed heuristics that work on a type graph representation of the type inference process.

In order to support existentials and Generalized Algebraic Data Types (GADTs) in Helium, we extend the type graphs of Helium with facilities for local reasoning. We have translated the original Helium heuristics to this new setting, and define a number of GADT-specific heuristics that help diagnose Helium programs that employ GADTs.

Keywords type error diagnosis, generalized algebraic data types, type graphs, Haskell

1 Introduction

Haskell has always been a hotbed of language and type system innovation, contributing to the popularization of many such features. The advantage of a rich type system is that the programmer can obtain many guarantees about the correctness of an implementation without having to resort to testing. But advanced type system features come at a price. One price is that when type inconsistencies arise, it is noticeably harder for the compiler to explain to the programmer what the inconsistency is, where it arises, how it might be resolved, all without revealing internal implementation details of the compiler. This hinders the uptake of these advanced features, leading to programmers avoiding them, and settling for fewer guarantees.

One such language feature is that of Generalized Algebraic Datatypes (GADTs for short), that allows the programmer to encode type information in the data type constructors of an algebraic data type. It is a popular feature of Haskell, in particular for encoding type-like properties for deeply embedded domain-specific languages.
A simple but typical example is:

$$\begin{array}{l}\textbf{data}\ Expr\ a\ \textbf{where}\\ \quad LitInt\ \ :: Int \rightarrow Expr\ Int\\ \quad LitBool :: Bool \rightarrow Expr\ Bool\\ \quad Equals\ :: Eq\ a \Rightarrow Expr\ a \rightarrow Expr\ a \rightarrow Expr\ Bool\end{array}$$

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

where the *Equals* constructor encodes that it can only compare the equality of two subexpressions that have the same type *a*, that moreover is an instance of the *Eq* type class. The type inferencer will then forbid expressions such as *Equals* (*LitBool True*) (*LitInt* 1), because the arguments to *Equals* do not agree on the choice for *a*. Typical for GADTs, as compared to ordinary ADTs, is that the type variable *a* does not show up in the result of *Equals*, making it an *existential variable*.

Now, if we type check the following function (note that we have omitted the type signature),

$$\begin{array}{l}lit\ (LitInt\ \ \ x) = x\\ lit\ (LitBool\ x) = x\end{array}$$

then GHC, the standard Haskell compiler, returns the type error message

```
* Couldn't match expected type 'p' with actual type 'Int'
    'p' is untouchable
      inside the constraints: a ~ Int
      bound by a pattern with constructor: LitInt :: Int -> Expr Int,
               in an equation for 'lit'
      at <interactive>:18:6-13
  'p' is a rigid type variable bound by
    the inferred type of lit :: Expr a -> p
    at <interactive>:(18,1)-(19,19)
  Possible fix: add a type signature for 'lit'
* In the expression: x
  In an equation for 'lit': lit (LitInt x) = x
* Relevant bindings include
    lit :: Expr a -> p (bound at <interactive>:18:1)
```

What is wrong with this message? First of all, the message introduces type variables such as *p* that are not part of the input program. It uses terminology, e.g., ~, rigid and untouchable, that are involved in the type inference process but of which the programmer should not be aware, and it provides an inferred type for *lit*, namely *Expr a* → *p*, which is in fact not correct. Moreover, it produces a very similar message for the other branch of *lit*!

Our implementation, a branch of the Helium compiler [10], instead returns the following message in which it reports that the problem is that a type signature is missing, and moreover it produces a type signature for *lit* as a hint which is consistent with the rest of the code:

```
(6,1), (7,1): A type signature is necessary for this definition
 function : lit
 hint     : add a valid type signature, e.g. (X a) -> a
```

We achieve this by making the following *contributions*: we have extended type graphs in Helium to deal with local reasoning mirroring the behavior of the OutsideIn(X) system, the basis of the type inference process of GHC, and we have transferred the heuristics of Helium to the new setting. A number of heuristics have been designed to deal with type errors that involve GADTs, and our work has been implemented as a branch of a realistic Haskell compiler, Helium.

2 Constraint-Based Type Inference

Compilers for statically-typed programming languages must check that the input from the programmer conforms to the type system imposed by the language. We refer to this process as type *checking* – the compiler has to check that the program is well-typed according to the rules – and *inference* – the compiler may have to deduce some local type information. We use the term type inference to refer to both.

The earliest implementations of type inference for functional languages use a *direct* approach in which type inference is implemented by traversing the Abstract Syntax Tree (AST) and performing unifications on the fly, e.g., the classic $\mathcal{W}$ and $\mathcal{M}$ implementations of the Hindley-Milner type system [4, 12].

Later approaches often prefer a *constraint-based* approach, divided into two phases. In the first phase, the AST is traversed to gather *constraints* which must be satisfiable for the program to be well-typed. A dedicated *solver* then takes these constraints as input, checks their validity and returns types found for the inferred elements of the program. Pottier and Rémy [19] is the standard reference; many compilers like GHC [31] and Swift [28] have followed their lead.

Direct approaches to type inference usually have a *bias* with respect to type error reporting, due to the fixed order in which they traverse the AST. For example, if we are checking the expression $True \equiv$ *'a'* and we traverse arguments from left to right, the error is found in the second argument. For that reason, constraint-based approaches are often the preferred approach for type error diagnosis: we can more easily solve constraints in different orders, and it is easy to experiment with modified sets of constraints to figure out the best explanation for an error [5, 7, 23]. Given that the GHC dialect of Haskell has a constraint-based specification, constraint-based type inference is the natural choice for our work.

In the remainder of this section we give a high-level overview of constraint-based type inference. We describe type checking for the λ-calculus with pattern matching defined in Figure 1. Our presentation is heavily influenced by OutsideIn(X) [31]; we omit some details for the sake of conciseness. In particular, the described λ-calculus does not have a **let** construct for local bindings, but of course our implementation does.

As usual in Hindley-Damas-Milner-based type systems, the types of variables and data constructors in an environment Γ may quantify over some variables, and thus are assigned a *type scheme*. In addition to quantified variables, type schemes may also request some *constraints* to hold at each use of the corresponding variables. The syntax of constraints is left open by the framework – hence the X in OutsideIn(X) –, we only require X to have a notion of equality between types, $\tau_1 \sim \tau_2$. In the case of GHC, X includes the theory of type classes and type families, so we can form type schemes such as $\forall a. Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$.

The constraint gathering judgement takes the form $\Gamma \vdash e : \tau \rightsquigarrow Q$, which reads: in the environment Γ the expression e has type τ under the set of constraints Q. During constraint gathering some of the types are yet unknown, so we introduce *unification variables* α to represent them. Finding the types each of these unification variables stands for, corresponds to the inference part of the solver. The rules for the judgment, given in Figure 2, are unsurprising. In the VAR rule the rigid type variables quantified in a type scheme are *instantiated*, that is, replaced with fresh unification variables. Pattern matching is described by the CASE rule: we need to find both the particular instantiation of the type constructor $\mathsf{F}\,\overline{\gamma}$ used by the scrutinee e, and the common return type β of all the branches.

The next step of the process is constraint *solving*, which is formulated as a rewriting relation on constraints [26, 31], turning the original constraints into a simpler *solved* set of constraints. For reasons of space we provide two example rules:

$$\mathsf{F}\ \tau_1\ \ldots\ \tau_n \sim \mathsf{F}\ \rho_1\ \ldots\ \rho_n \rightsquigarrow \tau_1 \sim \rho_1 \wedge \cdots \wedge \tau_n \sim \rho_n$$
$$\mathsf{F}\ \tau_1\ \ldots\ \tau_n \sim \mathsf{G}\ \rho_1\ \ldots\ \rho_m \rightsquigarrow \bot, \quad \text{if } \mathsf{F} \not\equiv \mathsf{G}$$

The former rule shows how an equality check between two type constructors is decomposed (if they have the same name and the same number of arguments), while the latter show that if the heads do not match, then a type error results (modeled by rewriting to $\bot$). In Section 2.1, we shall refine $\bot$ to capture some additional information.

2.1 Type graphs

If the constraint solver, applying the rules of the rewrite relation, terminates without finding any inconsistencies among the gathered constraints, the compiler pipeline continues with further analyses and optimizations, to eventually reach

Rigid type variables		$\ni$	$a, b, \ldots$
Type constructors		$\ni$	$\mathsf{F}, \mathsf{G}, \ldots$
Monotypes	τ, ρ	$::=$	$a \mid \mathsf{F}\,\overline{\tau}$
Constraints	Q	$::=$	$\top \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid \ldots$
Type schemes	σ	$::=$	$\forall a. Q \Rightarrow \tau$
Term variables		$\ni$	$x, y, \ldots$
Data constructors		$\ni$	$K, \ldots$
Expressions	e	$::=$	$x \mid K \mid \lambda x \rightarrow e \mid e_1\, e_2$
			$\mathbf{case}\ e\ \mathbf{of}\ \overline{K\,\overline{x} \rightarrow e}$

Figure 1. Syntactic categories of λ-calculus with pattern matching

$$
\begin{array}{lrcl}
\text{Unification variables} & & \ni & \alpha, \beta, \ldots \\
\text{Monotypes} & \tau & ::= & \upsilon \mid \ldots
\end{array}
\qquad
\begin{array}{lrcl}
\text{Type variables} & \upsilon, \omega & ::= & a \mid \alpha \\
\text{Environments} & \Gamma & ::= & \epsilon \mid \Gamma, x : \sigma
\end{array}
$$

$$
\frac{x : \forall \overline{a}. Q \Rightarrow \tau \in \Gamma \qquad \overline{\alpha}\ \text{fresh}}{\Gamma \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow [\overline{a \mapsto \alpha}]Q}\ \text{VAR}
\qquad
\frac{\alpha\ \text{fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \rightsquigarrow Q}{\Gamma \vdash \lambda x \to e : \alpha \to \tau \rightsquigarrow Q}\ \text{ABS}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow Q_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow Q_2 \qquad \alpha\ \text{fresh}}{\Gamma \vdash e_1\, e_2 : \alpha \rightsquigarrow \tau_1 \sim (\tau_2 \to \alpha) \wedge Q_1 \wedge Q_2}\ \text{APP}
$$

$$
\frac{\Gamma \vdash e : \tau_0 \rightsquigarrow Q_0 \qquad \beta, \overline{\gamma}\ \text{fresh} \qquad K_i : \forall \overline{a}. \overline{\rho_i} \to \mathsf{F}\,\overline{a} \in \Gamma \qquad \overline{\Gamma, x_i : [\overline{a \mapsto \gamma}]\rho_i \vdash e_i : \tau_i \rightsquigarrow Q_i}}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ \overline{K_i\,\overline{x}_i \to e_i} : \beta \rightsquigarrow Q_0 \wedge \tau_0 \sim \mathsf{F}\,\overline{\gamma} \wedge \overline{Q_i \wedge \tau_i \sim \beta}}\ \text{CASE}
$$

Figure 2. Constraint gathering for λ-calculus with pattern matching

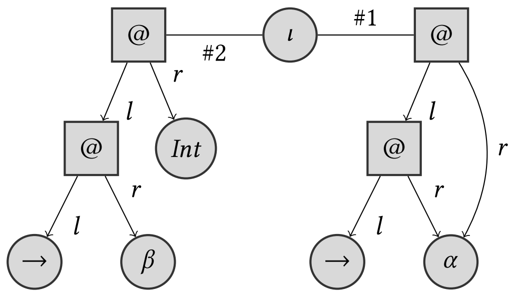


Figure 3. With type applications

code generation. If an inconsistency is detected, we should explain the problem to the programmer by means of a type error message. We aim to make this message as informative as possible, and at the same time as concise as possible to prevent the programmer from being overwhelmed [33]. In that case, we would like to know what are the *original* constraints which led to the problem; we can then link those constraints to the program positions in which they were generated to construct an informative error message. A naïve solution to the problem of finding the problematic constraints is to include every constraint which has ever taken part in the rewriting path to the constraint. However, we can easily end up with too many constraints. Consider for example the set of three constraints $\alpha \sim [\beta] \wedge \alpha \sim \mathit{Maybe}\,\gamma \wedge \gamma \sim \mathit{Int}$ (although we call them sets, we combine the separate constraints with $\wedge$). Since the order of solving is not set in advance, we can first make the second and third constraints interact, leading to $\alpha \sim \mathit{Maybe\ Int}$, and only then discover that we have inconsistent ideas of what α should stand for. The naïve approach would flag the three constraints as problematic, but it is clear that the third plays no real role in the type error.

Although alternative solutions exist to omit constraints that do not play a role in the type error (e.g., [5] to find all minimal unsatisfiable constraint sets), in our work we maintain a data structure with all the constraints obtained during the solving process, that we can process later to figure out the problem. Such a data structure must be able to represent not only consistent, but also *inconsistent* sets of constraints. *Type graphs* [7, 11] provide that functionality for the case of type equalities. Type graphs are part of the TOP framework, which is the type inference engine used by the Helium Haskell compiler [8, 9]. Figure 3 contains two examples of type graphs and the constraint sets they represent. Vertices can have two shapes: circular vertices are used for (unification and rigid) type variables and type constructors; the special square vertex tagged with @ is used for type application. Following the usual convention, type application associates to the left and the arrow constructor is written infix, so $\beta \to \gamma$ is equivalent to $((\to)\,\beta)\,\gamma$. Each type variable only appears *once* in a type graph, so different references to α in Figure 3 point to the same node. Edges are either directed edges marked with l and r outgoing from a type application node @ representing the two arguments of @, or undirected edges representing a type equality marked with the constraint they originated from.

During the solving phase, the type graph is saturated with *derived edges*, which represent those equalities which are implied by the original set. In Figure 3 two derived edges would be present once the solver is finished: one between β and α, and another between *Int* and α.

An inconsistency in the case of type equalities arises from a constraint which equates two distinct type constructors, such as $\mathit{Int} \sim \mathit{Bool}$, or fails the occurs check, such as $a \sim [a]$. In the type graph such a problem is represented by a path between the two problematic elements, we call them *error paths*. Figure 3 does not have error paths, but it would if we replace β by *Bool*.

Heuristics

An error path gives a set of constraints involved in an error, but in order to produce a concise error message we need to choose one of them as responsible. The choice should be made so that if the blamed constraint is removed, the type graph becomes consistent, as long as no other inconsistencies are present in the type graph. This is easy to check in the type graph by ensuring that no other path exists between the problematic vertices. However, we do not want to check every possible subset of constraints, and the choice may not be unique. For that reason, we define a set of *heuristics* to guide the search in the type graph.

Different heuristics work in different ways. Some of them filter out constraints which should not be blamed, other heuristics select a constraint and assign it a weight, and then the one assigned the highest weight will be blamed.

Heuristics tend to strongly differ in their specificity. Language-*independent* heuristics can be applied to any type graph, regardless of the programming language it represents. The participation heuristic assigns a higher weight to those constraints depending on how often they are part of an error path. Language-*dependent* heuristics employ knowledge of the underlying language, and which are the more plausible explanations for a programmer mistake. Because of their specificity and the subsequent specificity of the error messages they can generate, they typically assign higher weights. In the Helium compiler there are heuristics such as "missing argument in an application", "missing components in a tuple", or "mistook $(+)$ for $(+\!\!+)$ in a function call".

2.2 Type inference for GADTs

Generalized Algebraic Data Types (or GADTs, for short) extend ordinary ADTs, by allowing us to refine type information for particular constructors.

For the *Expr* datatype defined in the introduction, we can write a well-typed interpreter of type $Expr\ t \rightarrow t$.

$$\begin{aligned}
&eval :: Expr\ t \rightarrow t\\
&eval\ (LitBool\ b) \quad = b\\
&eval\ (LitInt\ \ \ i) \quad = i\\
&eval\ (Equals\ x\ y) = eval\ x \equiv eval\ y
\end{aligned}$$

Note that we do not have to check at every step that the returned expression has the correct type, because this is statically enforced.

Following the rules in Figure 2, this code is not well-typed: for one, the rule CASE requires that the types of all branches coincide, while in this case the first branch returns a *Bool* and the second an *Int*. Second, the type signature of *eval* requires the function to be *polymorphic* in *t*. However, each of the three branches fixes one *concrete t*.

The key difference with pattern matching over a GADT is that each constructor may bring in *local* information. For example, by matching on *LitBool* we know that *t* can only be *Bool* in that branch. But that only works if the solver avoids mixing information local to different branches.

The language of constraints from Figure 1 cannot encode local information, so we extend our constraint language with *existentials*, as done in Figure 4. A constraint of the form $\exists\overline{\alpha}.(Q_1 \supset Q_2)$ represents a local scope in which a substitution for unification variables $\overline{\alpha}$ should be obtained, and where the *wanted* constraints Q_2 may use information from the *given* constraints Q_1. For the *eval* function, the constraint set will then be something like:

$$\begin{aligned}
&\exists\overline{\alpha}.(t \sim Bool \supset \text{constraints from } LitBool \text{ branch})\\
&\wedge \exists\overline{\beta}.(t \sim Int \supset \text{constraints from } LitInt \text{ branch})\\
&\wedge \exists\overline{\gamma}.(t \sim Bool \supset \text{constraints from } Equals \text{ branch})
\end{aligned}$$

The modified CASE* rule is responsible for harvesting the given constraints $Q_i^\star$ in each existential from the types of the data constructors matched upon. One small detail is that the OUTSIDEIN(X) framework insists that the return type of each data constructor has the same form as for ADTs, that is, a type constructor applied to distinct type variables. The solution is to work around this restriction by using equality constraints. In other words, for the type checker the type of *LitBool* is actually:

$$LitBool :: \forall a.a \sim Bool \Rightarrow Bool \rightarrow Expr\ a$$

The constraint solver also has to be extended to deal with local constraints. In the case of OUTSIDEIN(X), this is done by moving from a simple rewriting relation $Q \rightsquigarrow Q'$ into a more complex form $Q_g; \overline{\alpha} \vdash Q_w \rightsquigarrow Q_r$, which represents that under local (given) information Q_g we can rewrite the (wanted) constraints Q_w into the simpler (residual) form Q_r, and only the variables $\overline{\alpha}$ should be treated as unifiable. Keeping track of the unifiable (or touchable) variables is important for maintaining scoping invariants that prevent information from one branch to infect the other. This rewriting relation is recursively called by the $\vdash^\star$ judgment from Figure 5: everytime we go inside an existential, the set of given constraints grows. As a technical detail, each type checker has to define a notion of *solved form*: a set of constraints which is completely solved. In the case of type equalities, that means that every constraint in the residual set is of the form $\alpha \sim \tau$.

The purpose of our work is to combine type graphs, a data structure that has been found useful for explaining type errors, with the ability to deal with local information. The heuristics can then work on such extended type graphs to analyze type inconsistencies in the presence of GADTs, and generate suitable type error messages.

3 Extended Type Graphs with Local Constraints

This section introduces our extensions to type graphs so that they can be used to represent a type inference process in OUTSIDEIN(X). From this point on we use the term OUTSIDEIN(X) to refer to the original design described by Vytiniotis et al. [31], TOP to refer to the older implementation in the Helium compiler based on type graphs, and *Rhodium* to refer to the extended type graphs introduced in this paper. It makes sense for Rhodium to be as backwards compatible as possible both with OUTSIDEIN(X) and Helium. There is one problem: the formulation of OUTSIDEIN(X) insists that local definitions are not implicitly generalized, while Helium follows the Hindley-Milner convention of generalizing every local binding as much as possible. We follow OUTSIDEIN(X) in this, so we sometimes reject programs that are accepted

$$\text{Constraints } Q ::= \exists\overline{\alpha}.\,(Q_g \supset Q_w) \text{ where } Q_g \text{ contains no existentials} \mid \dots$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau_0 \rightsquigarrow Q_0 \\ \beta, \overline{\gamma}\,\text{fresh}\end{array} \qquad \overline{\begin{array}{c} K_i : \forall \overline{a}\overline{b_i}.\, Q_i^\star \Rightarrow \overline{\rho_i} \to \mathsf{F}\,\overline{a} \in \Gamma \\ \Gamma, x_i : [\overline{a \mapsto \gamma}]\rho_i \vdash e_i : \tau_i \rightsquigarrow Q_i \end{array}} \qquad \overline{\delta}_i = \mathrm{fuv}(\tau_i, Q_i) - \mathrm{fuv}(\Gamma, \overline{\gamma})}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \overline{K_i\,\overline{x}_i \to e_i} : \beta \rightsquigarrow Q_0 \wedge \tau_0 \sim \mathsf{F}\,\overline{\gamma} \wedge \overline{\exists \overline{\delta}_i.\,([\overline{a \mapsto \gamma}]Q_i^\star \supset Q_i \wedge \tau_i \sim \beta)}}\ \textsc{case}^\star$$

Figure 4. Constraint gathering for λ-calculus with GADT pattern matching

$$\frac{\begin{array}{l} Q_s = \{Q \in Q_w \mid Q \text{ is not existential}\} \\ Q_g ; \overline{\alpha} \vdash Q_s \rightsquigarrow Q_r \\ \text{for each } \exists\overline{\beta}.\,(Q'_g \supset Q'_w) \text{ in } Q_w\text{:} \\ \quad Q_g \wedge Q_r \wedge Q'_g ; \overline{\beta} \vdash^\star Q'_w \rightsquigarrow Q'_r \\ \quad Q'_r \text{ is in solved form} \end{array}}{Q_g ; \overline{\alpha} \vdash^\star Q_w \rightsquigarrow Q_r}$$

Figure 5. Skeleton of a solver for existential constraints

by Helium, although all can be fixed by adding the right signatures in the right places.

3.1 Representation of extended type graphs

In this section we explain how constraints in OutsideIn(X) are translated into Rhodium type graphs. The main extension with respect to TOP is the need to represent *existential* constraints. Note that OutsideIn(X) is parametric, so each concrete implementation may add new sorts of vertices and edges to the type graph. In this section we focus on the parts shared by every possible X, namely types and equality constraints.

Variables, types, and constraints

There are multiple valid ways to represent a type in a type graph. Take for example the type *Either A B*. We can choose to represent type application as a binary operator, viewing the type as (*Either A*) *B*, or as an n-ary application in which the type constructor receives a list of argument types, hence viewing the type as *Either* [*A*, *B*]. In Rhodium, we follow the former design and use a special vertex for type application @, as depicted in Figure 6. Because Rhodium also supports type families, and these occur only in fully applied form, Rhodium does allow a vertex that represents a type family to have more than two children. For consistency reasons, the labels *r* and *l* that we saw in Section 2.1 have been replaced by the numbers 0 and 1. Apart from this detail, the treatment of type families in Rhodium follows [31].

Type variables and constructors inherit their representation from TOP. In the case of type variables we annotate the vertex with its *touchability*, which governs when a type variable can be unified. As depicted in Figure 6, a variable may be completely *untouchable* – also known as rigid or Skolem; these arise from checking polymorphic types – or

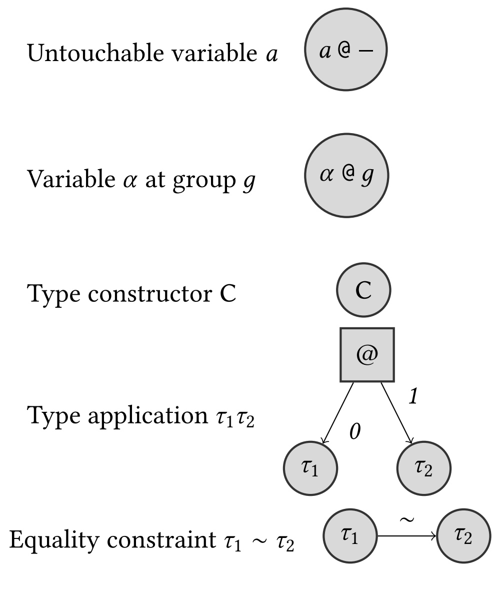


Figure 6. Representation of type graphs

touchable at a given group. As we shall discuss later, groups are used to track which constraints may interact with each other once existential constraints enter the picture.

The last element in our type graphs are *constraint edges*. This is an important design decision in Rhodium: every constraint in the system must be represented by an edge. In the simplest case of only type equalities, this representation is quite natural: we connect the two types which should be equal. But in contrast to TOP, type equalities in Rhodium are directional, that is, $\tau_1 \sim \tau_2$ is not represented in the same way as $\tau_2 \sim \tau_1$. The reason is that OutsideIn(X) requires an ordering to guarantee termination in a specific step of the solving process (more concretely, during *orientation*). Other than that, type equality edges are interpreted as undirected.

Relation to the type graphs of TOP

The original type graph implementation of Helium also deals with *instantiation* constraints of the form $\tau > \sigma$, representing that τ is an instantiation of a type scheme σ in order to deal with let-polymorphism. However, one of the design decisions in OutsideIn(X) is not to implicitly generalize **let** definitions. This makes instantiation constraints redundant, since we can generate new fresh instances of the programmer-provided type scheme during constraint gathering. In Rhodium we have taken an intermediate position: we do represent instantiation constraints explicitly in the type graph, but we readily turn them into equality constraints at the beginning of solving. Due to the invariants in OutsideIn(X) we can do this once and for all. The reason for this choice is two-fold. First, it opens the door to extensions of OutsideIn(X) such as GI [24], which introduce higher-rank and impredicative types. Second, future heuristics might want to return a different message depending on whether an inconsistent constraint arose from an instantiation constraint, or not.

Existentials

Pattern matching on GADTs introduces existential constraints during gathering, as described in Section 2.2. Supporting them leads to quite substantial changes to type graphs when compared to TOP's. The most important issue is that an existential constraint contains constraints nested into it, and we need to represent this nesting in our type graphs. We consider two possible choices and discuss their advantages and disadvantages.

The first possibility is to keep free of references to existentials and nesting. In this scenario, everytime we recurse using the $\vdash^\star$ judgment from Figure 5, we create a completely new type graph with the given constraints and the new simple constraints, and then proceed to solve it. This has the advantage of being simple, because we can be sure that all constraints in the graph may freely interact with each other. However, it makes type error diagnosis harder, since we cannot look at the interaction between different existential branches.

Consider the following example:

```
data Expr a where
  I :: Int  → X Int
  B :: Bool → X Bool
  A :: a    → X a

f :: Expr a → Bool
f (I x)  = x
f (B b)  = if b then 3 else 5
f (A _) = 7
```

This code is ill-typed. The most probable cause is that the type signature of f is not correct; we can fix the problem by replacing *Bool* by *Int*. If each branch of f would lead to a separate type graph, we would in fact find three errors, because neither branch is consistent with the type signature.

In the interest of good error diagnosis, we prefer a representation that allows a more holistic view. Therefore we have chosen to integrate all constraints into a *single* type graph. However, this means we have to provide a means to decide which pairs of constraints may interact, otherwise the local reasoning that we need to deal with existentials is lost.

For this reason, we assign to each type variable and each constraint edge a *group*, which tells us to which existential each constraint belongs, and whether a constraint is a given or wanted constraint. In this paper we use numbers to represent groups, starting with 0 for top-level given constraints, 1 for the top-level wanted ones, increasing these numbers as we go into existential constraints. We are careful to maintain two invariants: (1) if an existential constraint is part of another constraint, then its group identifier is higher than that of its parent, and (2) given constraints are always assigned an even number, and wanted constraint always have an odd identifier.

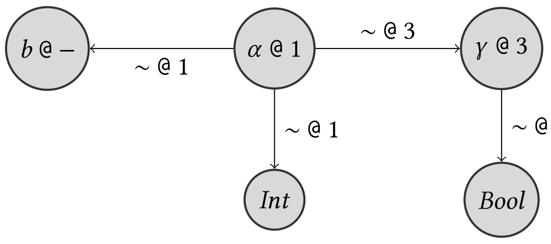


Figure 7. Rhodium type graph for $\alpha \sim Int \wedge \alpha \sim b \wedge \exists\gamma.(\gamma \sim Bool \supset \alpha \sim \gamma)$

Figure 7 depicts the extended type graph that represents

$$\alpha \sim Int \wedge \alpha \sim b \wedge \exists\gamma.(\gamma \sim Bool \supset \alpha \sim \gamma)\ .$$

At group 1, the top level wanted constraints, the only touchable variable is α, so it is marked as such. Each separate constraint outside the existential is represented as an edge with this group identifier. Inside the existential, $\gamma \sim Bool$ is a given constraints, and thus it is assigned an even group 2 (higher than 1). The innermost wanted constraint is assigned a higher, odd group, 3. Note that the group of a type variable is not related in general to the groups of the constraint edges that point to it, but rather to the specific existential in which the variable is introduced.

3.2 The solving of extended type graphs

The solving process of OutsideIn(X) in which solving processes are spawned recursively when an existential is encountered, has become a single iterative process in Rhodium. We employ groups attached to the constraints to ensure that only constraints that are allowed to, may interact with one another. Other than that, solving is performed using the usual rules that each implementation of OutsideIn(X) we know of uses. However, because we apply our rules to Rhodium type graphs instead of to constraints, below we provide some details of the rewriting process.

Groups and accessible sets

Recall that every constraint edge is assigned to a group, which represents the most deeply nested existential in which that constraint lives. To emulate local reasoning, we employ this information to decide when an interaction between two constraints may take place. Take for example the graph in

Figure 7: the constraint $\alpha \sim Int$ should always be allowed to interact with other constraints, since it resides at top-level. The given constraint $\gamma \sim Bool$ (with group 2) should be visible in the wanted part of that existential, in group 3.

To decide for a given (current) group g which constraints may be employed during solving, we introduce the notion of *accessible set*, the set of groups g may interact with. The accessible set for a constraint is built starting with its group, and then adding all the ancestor existential groups until we reach top level. Take for example the set of constraints, in which constraints in Q_n are assigned group n:

$$Q_1 \wedge \exists\overline{\alpha}_1.(Q_2 \supset Q_3) \wedge \exists\overline{\alpha}_2.(Q_4 \supset Q_5 \wedge \exists\overline{\alpha}_3.(Q_6 \supset Q_7))$$

The accessible set of Q_6 is $\{1, 4, 5, 6\}$: those are the other groups (including itself) it may interact with. Note that in particular the accessible set of Q_6 does *not* contain 2 or 3, since those constraints are in other existential branches. This mechanism is similar to the scoping mechanism described by Serrano [22].

The solving process traverses each group in a similar fashion to the one described in Section 2.2 for the OutsideIn(X) framework. We start by considering the top level constraints, and then recurse into the existentials. The use of increasing natural numbers as identifiers for groups gives us a simple method to know at every point which constraints may be considered. Since we maintain the invariant that the group of a constraint is always higher than that of its parents in the existential structure, it is enough to start with the constraints at group 0 (the top level given ones), and then increase the current group until all have been considered.

Translating solving rules to the setting of type graphs

Although organized somewhat differently, the Rhodium type graph solver follows OutsideIn(X) faithfully, using a rewriting relation like OutsideIn(X) does. However, since we work on Rhodium type graphs, and not on constraints, we must reflect the result of applying a rewrite rule back into the type graph.

In the case of a *canonicalization* rule, which rewrites a single constraint, the type graph solver first selects a constraint edge in the current group to which a canonicalization rule is applicable. Then it executes one step of the rewriting relation, producing a new set of constraints which should be added to the type graph. Special care should be taken here: the new constraints and new *touchable* variables have to be the same group as the considered constraint. The former ensures that canonicalization rules respect the nested existential structure, the latter are necessary to deal correctly with type families [31].

One important difference between the representation of a set of constraints in a purely syntactic manner, as done by the OutsideIn(X) formalization, and our type graphs, is that in the former case a rewritten constraint is removed from the current set, whereas in the latter all the constraints created

edge	constraint	created by
#0	$a \sim Int$	original
#1	$a \sim b$	original
#2	$a \sim Bool$	original
#3	$a \sim Int$	interact(#0, #1)
#4	$b \sim Int$	interact(#0, #1)
#5	$a \sim Bool$	interact(#2, #3)
#6	$Bool \sim Int$	interact(#2, #3)

Figure 8. Overly conservative error path

edge	constraint	created by
#0	$a \sim Int$	original, interact(#0, #1)
#1	$a \sim b$	original
#2	$a \sim Bool$	original, interact(#2, #0)
#7	$b \sim Int$	interact(#0, #1)
#8	$Bool \sim Int$	interact(#2, #0)

Figure 9. Modified error path

during the process are retained. To avoid infinite rewriting, once a rewriting rule has been applied to a constraint, that constraint is marked as *resolved*, and will not take part in further simplification.

In the case of *interaction* rules, two constraints interact with one another to create a new set of constraints. In order to guarantee correctness, we need to ensure that the constraints can interact safely. In particular, given a constraint Q in a group n, it may only interact with other constraints whose group belongs to its accessible set.

In general, an interaction rule has the form $Q_1, Q_2 \rightsquigarrow Q_3$. We insert the constraints Q_3 into the type graph the same way we did with canonicalization rules, assigning them to the current group, and mark both Q_1 and Q_2 as resolved. Due to the way in which solving proceeds, this means that we put the new constraints at the deepest existential level of the two, as it should be.

One common scenario in a rewriting system for type inference is that some of the constraints in Q_1 and Q_2 may be returned as part of Q_3. In that case we need to ensure that only the new constraints are introduced in the type graph, otherwise error reporting may suffer. Take, for example, the constraints $a \sim Int \wedge a \sim b \wedge a \sim Bool$. In Figure 8 all the constraints from an interaction are added to the type graph, whereas in Figure 9 only the new ones are added. The latter describes more precisely the solving process, and thus leads to more precise heuristics. As result, we may need to unmark some of the constraints as resolved, if they are present again in the new set produced by the rewriting rule.

Errors

If a constraint rewrite returns ⊥, no constraint is added to the type graph. Instead, the edge is marked as *inconsistent* preventing it from taking part in any further solving, although the solving process will continue. In addition, we may attach an *error label* to each inconsistent edge. For example, $Int \sim Bool$ may be labelled with INCORRECT CONSTRUCTORS, or $a \sim [a]$ with INFINITE TYPE. These labels can be employed by the heuristics used for type error diagnosis later on (Section 4).

Residual constraints

Once we have finished applying rewriting rules to the constraints in a group there might be some constraint edges which remain unmarked as resolved. However, a non-empty set of leftover constraints does not necessarily mean that the original program contains an error, we need some further post-processing. This additional process may be either performed at the end of the simplification of each group, or at the very end of the solving process.

First of all, there are constraints such as $Eq\ \alpha$ which we always expect to mark as resolved. In this case, not having done so means that an instance for α was not found in the given constraints or the axioms, and we should report this fact as a type error. The error label we assign to these constraints is RESIDUAL CONSTRAINT.

For the case of equality constraints like $\alpha \sim Int$ the distinction is subtler. Some of those equality constraints correspond to parts of the final *substitution* that the solving process produces; those are the ones of the form $\alpha \sim \tau$ which satify that (1) its group g correspond to a wanted set, and (2) the type variable α is also introduced in that same group. If condition (1) is not satisfied, the constraint is simply ignored, but if (1) holds but (2) does not, the constraint represents inconsistent information and it is marked as an error with label VARIABLE ESCAPE. Note this pair of conditions is a safe over-approximation of when a set of equality constraints represent a correct substitution; real implementations such as GHC implement a "variable floating" rule which is less strict yet still safe [24].

We close this section with an elaborate example to illustrate the solving process. Consider the wanted constraint

$$Num\ \alpha \wedge \alpha \sim Bool \wedge \exists\beta.\ (\beta \sim Int \supset \alpha \sim \beta) \wedge \\ \exists\gamma.\ (\gamma \sim Bool \supset \alpha \sim \gamma),$$

where the variable α is touchable at top level and no axioms or given constraints are present. (1) Rhodium makes a type graph of all the constraints, based on the constraint solver X that is specified. Groups are assigned as usual: even for given constraints, odd for wanted constraints. (2) We start the solving process for group 1. There we allow two constraints, $Num\ \alpha$ and $\alpha \sim Bool$, to interact with one another. This results in the constraints $Num\ Bool$ and $\alpha \sim Bool$, but only the former is added to the type graph, since the latter was already there. As these constraints can not be simplified further, we mark $Num\ Bool$ as residual, and we increase the current group to 2. (3) With a current group of 2, we consider the given constraints of the first existential. These constraints can interact with the constraints of group 1, but not with one another. Because of their particular shape of the constraints, no interaction rule applies, and we increase the current group to 3. (4) Within group 3, the constraint $\alpha \sim \beta$ is considered wanted. There are no more constraints in that group, but the constraint may interact with both $\alpha \sim Bool$ (group 1) and $\beta \sim Int$ (group 2), leading to $Bool \sim Int$. This is an inconsistent constraint, and it is marked as such. (5) We repeat the process with the other existential constraint. In this case the wanted constraint is first turned into $Bool \sim Bool$, which them disappears by a canonicalization rule. Thus, we have no residual constraints in this group.

4 Heuristics for GADTs

In this section, we focus on the heuristics defined specifically for diagnosing type incorrect code that involves GADTs, and provide examples of type error messages provided by our implementation. We have also re-implemented many heuristics that were present in Helium previously and that worked on the simpler type graphs in TOP [1].

4.1 How heuristics are applied

After constraint solving within Rhodium has terminated, some constraints may have been marked as an error (using a specific error label). For example, a constraint $Int \sim Bool$ will have the label INCORRECT CONSTRUCTORS, and $a \sim Int$ may have the label VARIABLE ESCAPE.

Given a single error constraint Q and the simplified type graph, we then determine the error slice associated with Q. This *error slice* consists of all the constraints that may have contributed to the problem. As mentioned previously, every constraint keeps track of how it was created: either it was generated from the program directly – an *original* constraint – or it is the result of a constraint solving step applied to some constraints, each of which keeps track of how it was created. By iteratively traversing the history of each constraint we construct the set all the constraints involved in the simplification process that led to the creation of Q. From this error slice we consider only those which were generated directly from the program, that is, the original ones. These constraints come with additional information obtained during gathering, e.g., the syntactic construct that generated the constraint, and the source location for that construct.

The input to the next step of this process is composed of pairs, where each pairs consists of an error constraint edge (which includes the error label attached to it) and the corresponding error slice. Each of these pairs is considered one by one. In each case, the goal is to reduce the error slice to a single original constraint, which is then *blamed* for the particular error. We do so by applying *heuristics* to the error slice. Even though heuristics consider only one error slice as target for reduction, they may query all the other error slices for additional information.

Rhodium provides quite a number of heuristics that are applied in sequence. Every application of a heuristic may reduce but never increase the error slice. If after running all heuristics more than one constraint remains, we choose the first constraint.

As in Helium, Rhodium supports two kinds of heuristic: filter heuristics and voting heuristics. A *filter heuristic* deletes constraints from the error slice, implying that those original constraints should *not* be blamed. An example of such a constraint is one that models that the condition of an **if**-expression should have type *Bool*. For the expression **if** 3 **then** 2 **else** 1 we expect a message that blames the use of 3 where a *Bool* is expected, and not a message that insists we should not demand an expression of type *Bool* in the condition.

A filter heuristic may delete any number of constraints from the slice, as long as the outcome is not the empty set, implying that no constraint can be blamed. Typically, if a filter heuristic observes that all constraints in the slice have the property it is designed to remove, it will in fact not delete any constraints in the hope that other heuristics can make a better choice.

The *voting heuristic* is essentially a collection of selectors. A *selector* is especially designed to recognize certain well-known error patterns, for example that the components of a pair occur in the wrong order. If it recognizes such a pattern, it returns the constraint to be blamed for the mistake, and a weight that indicates how likely it is that this is the cause of the inconsistency. If it does not recognize such a pattern, the heuristic will not participate in the voting heuristic.

After all selectors have made their choice, if any, all constraints with the highest weights assigned to them by a selector remain in the error slice and all others are deleted. The process then continues, if necessary, by considering any further heuristics.

The choice for a constraint to blame is not the only output of the process. Whenever a heuristic assigns the blame to a constraint, it also attaches a so-called *graph modifier* to that constraint that describes how the graph needs to be adapted to continue with the solving process. The default graph modifier is to delete the edge to which the blamed constraint was attached; this is the only graph modifier present in TOP, but we found we had to supply other options.

For example, a common type error is forgetting to add a particular constraint to the type signature of a function:

$g :: a \to a \to String$
$g\ x\ y = show\ x + show\ y$

In this case, we have two residual constraints of type *Show a*. If we may only remove constraints, we have to remove both *show x* and *show y* resulting in two very similar error messages. However, in Rhodium we employ a heuristic that blames a constraint that was found to be missing, and employs a graph modifier that *adds* the missing predicate *Show a* to the type signature of *g*, so that inference may continue. The type error message will come with a hint to the programmer to add the predicate to the type of *g*.

Our implementation provides a number of graph modifiers that we found useful. Beyond the default modifier, and the modifier that adds a residual constraint, Rhodium employs two others. Consider the example of *True* + 3. In that case, we have the constraints $\alpha \sim \beta \to \gamma \to \delta \wedge \alpha > \forall a.Num\ a \Rightarrow a \to a \to a$, where α represents the type of the function $(+)$. If we only remove $\alpha \sim \beta \to \gamma \to \delta$, we are still left with the instantiation constraint, which then causes an error as it has a residual constraint *Num a*. This graph modifier therefore removes both the application edge, as well as the accompanying type signature. The final modifier can add a type signature to a function. Indeed, every function that pattern matches on a GADT must have a type signature. When a type signature is missing, we produce a type error. In certain cases, we can recommend a type signature computed from the GADT pattern matches, and this modifier essentially allows us to add this recommended type signature to the type graph so that inference may continue.

4.2 Heuristics for GADTs

We now consider the type errors that can occur whenever GADTs are introduced. We describe a number of heuristics which deal with new error scenarios introduced by this language feature.

Missing constraint in GADT constructor

One of the main features of GADTs is the ability to introduce existential variables which do not exist outside of the scope of that constructor:

data X **where**
  $A :: b \to X$

$f :: X \to String$
$f\ (A\ x) = show\ x$

The type of the variable *x* is not mentioned in the data type *X*, so in this case we cannot add the constraint to type signature of the function. The *missing constraint heuristic* is aware of this fact, and produces the following error message:[1]

[1]Some error messages have been re-formatted to fit withing the page limits, but no text has been changed from the produced output of our implementation.

$$\begin{array}{l}
\textbf{data}\ \mathit{Expr}\ a\ \textbf{where} \\
\quad \mathit{LitInt}\ \ :: \mathit{Int}\ \ \rightarrow \mathit{Expr}\ \mathit{Int} \\
\quad \mathit{LitBool} :: \mathit{Bool} \rightarrow \mathit{Expr}\ \mathit{Bool} \\
g :: \mathit{Expr}\ \mathit{Int} \rightarrow \mathit{Int} \\
g\ (\mathit{LitInt}\ \ x) = x \\
g\ (\mathit{LitBool}\ y) = y
\end{array}$$

Figure 10. Unreachable pattern example

```
(5,11): Missing class constraint in type signature
 function          : show
   declared type   : Show a => a -> String
 class constraint  : Show b
 hint : add the class constraint to the type signature
        from the GADT constructor, defined at (2,4)
```

As part of the type error we provide the constraint that needs to be added, in this case the type class constraint *Show b*, and the location of the constructor to which the constraint should be added.

More generally, the "missing constraint" heuristic works in two phases. The heuristic tries first to introduce the missing constraint as part of the local definition, like the type signature. For example, a type signature of the function $Y\ a \rightarrow String$ would not be incorrect if the predicate *Show a* were to be added, so we prefer this over adding a constraint to the constructor. The main reason for this choice is that changing a constructor has arguably a larger impact than modifying a type signature, as the latter only requires the constraint to be satisfied whenever the *function* is called, not every single time the *constructor* is used. Only if the heuristic detects that it is impossible to add the constraint in a local definition, it suggests changing the constructor itself.

Unreachable pattern

Within a GADT, knowing the type of the scrutinee of a pattern match can make certain pattern matches inaccessible. Take for example the function *g* defined over a simplified version of the data type in the introduction in Figure 10. In this case, the type signature of *g* only allows values of type *Expr Int* as argument. As a result, the case of constructor *LitBool* can never happen, since it requires a value of type *Expr Bool*. This causes an inconsistent constraint of the shape $Int \sim Bool$ in the type inferencer.

The *unreachable pattern* heuristic detects that the inconsistency is caused due to a pattern match that does not match the provided type signature and provides an appropriate error message:

```
(7,4): Pattern is not accessible
 Pattern                 :  LitBool y
   constructor type      :  Bool -> Expr Bool
   defined at            :  (3,4)
   inferred type of      :  a -> Expr Int
   pattern
 hint :  change the type signature, remove the branch
        or change the branch
 possible type signature :  (Expr b) -> b
```

The error message specifies the type of the constructor, the inferred type of the branch, as well as the location of the definition of the constructor. Note that the heuristic also suggests a type signature that would allow the pattern match to be kept. This type signature is based on the most general type that can be derived from all of the individual branches. After this, the type signature is tested against the type graph to verify that it indeed resolves the error and does not introduce any other problems. Only when the type signature would resolve the error, it is recommended to the programmer. In all other cases, only the hint is provided, without mentioning the possible type signature.

Missing GADT type signature

As discussed by Vytiniotis et al. [31], once GADTs are introduced in the language, the principal types property is lost. This means that there could be *multiple* valid type signatures no two of which are instances of each other. As a result, functions dealing with GADTs require a type signature.

A very strict policy would require providing a type signature for *every usage* of a GADT, making the detection of not providing a GADT type signature a static check, but we decided against that. The reason is that in many cases we can use the information in the type graph to infer a possible type for the function. The process to determine this type signature is very similar to the process described for inferring type signatures for unreachable patterns.

If we take the code from Figure 10 and drop the type signature for *g*, then a type signature that would resolve the error is inferred and reported to the programmer:

```
(5,1), (6,1): A type signature is necessary for
              this definition
 function : g
 hint : add a valid type signature, e.g. (Expr a) -> a
```

The error message provides the possible type $(Expr\ a) \rightarrow a$ as a suggestion, but other type signatures might also be possible. Therefore, we keep the type signature as a hint, since we cannot guarantee it to be the programmer's intention.

Non-unifiable GADT variables

As discussed earlier, one key issue to sound checking and inference of code using GADTs is keeping track of which type variables can be unified at each moment. In fact, some of those are rigid and may *never* be unified with another type unless a given constraint assumes so.

Consider the following example, where we unify the variable *x* of type *b* with the type *Bool*, but the variable *b* is an existential introduced by the constructor *A*, hence forbidding *b* to unify with anything:

```
data X where
  A :: b → X
f :: X → Bool
f (A x) = x || True
```

Our implementation produces the following error message, stating that the variable cannot be unified. In addition to the error message itself, it also gives the original constructor, as well as the location at which it is defined:

```
(5,1): Cannot unify variable in function binding
 function binding         : f (A x) = x
   existential type       : b
   cannot be unified with : Bool
   constructor            : b -> X
   defined at             : (2,4)
```

This heuristic works on residual constraints of the shape $a \sim b$ where a is a non-touchable variable (be it rigid or coming from a different group) and b can be any type. We can tell from the type graph whether a is coming from a pattern match and whether that variable shows up in the result of the pattern match. For example, the variable d is not an existential in a constructor of type $c \rightarrow d \rightarrow Z\ d$, so in that case this heuristic does not apply.

4.3 Interaction between heuristics

Consider the following example, in which two errors are present: (||) is applied to 3 instead of a boolean, and the type signature of f is too general:

```
f :: a → b
f x = 3 || True
```

In the error message given by Rhodium, only the inconsistency is indicated:

```
(2,7): Type error in infix application
 expression       : 3 || True
 operator         : ||
   type           : Bool -> Bool -> Bool
 left operand     : 3
   type           : Int
   does not match : Bool
```

The "type signature is too general" heuristic did not contribute to the type error diagnosis process, as it could not do anything with the constraint $Bool \sim Int$. The inferencer detected also the residual constraint $b \sim Bool$, but this error was implicitly resolved by blaming $Bool \sim Int$, showing that the type inferencer, in combination with the heuristics, is capable of resolving multiple problems with a single message.

The following program exhibits two type errors, and since they are unrelated, two error messages are shown below:

```
f :: a → (Bool, a)
f x = let y = 3 || True
        in (y, "a")
```

Note that the original TOP only produces the second error, because the "type signature is too general" check is implemented in a post-processing phase, and not as part of the heuristics.

```
data Expr a where
  LitInt  :: Int              → Expr Int
  LitBool :: Bool             → Expr Bool
  Equals  :: Eq a ⇒ Expr a    → Expr a   → Expr Bool
  Max     ::        Expr Int  → Expr Int → Expr Int
eval :: Expr a → a
eval (LitInt  x)   = x
eval (LitBool b)   = b
eval (Equals x y)  = eval x y
eval (Max x y)     = maximum (eval x) (eval y)
```

Figure 11. A small expression language with its evaluation function

```
(1,1): Type signature is too general
 function           : f
   declared type    : a -> (Bool , a     )
   inferred type    : b -> (c    , String)
 hint               : try removing the type signature

(2,14): Type error in infix application
 expression         : 3 || True
 operator           : ||
   type             : Bool -> Bool -> Bool
 left operand       : 3
   type             : Int
   does not match   : Bool
```

For our next example, consider the code in Figure 11. There are two unrelated errors, one in the branch that checks the equality of expressions and the other is the confusion between the functions $max :: Ord\ a \Rightarrow a \rightarrow a \rightarrow a$, which takes two arguments, and $maximum :: Ord\ a \Rightarrow [a] \rightarrow a$, which takes one argument which is a list. The following error messages are reported by Rhodium:

```
(12,21): Type error in application
 expression         : eval x y
 term               : eval
   type             : Expr a -> Expr a -> Bool
   does not match   : Expr b -> b
 because            : too many arguments are given

(13,18): Type error in variable
 expression         : maximum
   type             : Ord a => [a] -> a
   expected type    :          Int -> Int -> Int
 probable fix       : use max instead
```

The error message identifies both errors correctly and is not confused about the presence of a predicate in the constructor of *Equals*. It also correctly identifies the return type of the incorrect usage of *eval* which is reported as *Bool*, due to the type signature of *eval*.

Compare these messages to those in Figure 12 that GHC produces. In the first message, GHC blames $a1 \sim (Expr\ a1 \rightarrow Bool)$. We would argue that this error message is worse than

```
Comparison2.hs:12:21: error:
    * Could not deduce: a1 ~ (Expr a1 -> Bool)
      from the context: (a ~ Bool, Eq a1)
        bound by a pattern with constructor:
                   Equals :: forall a. Eq a =>
                                 Expr a -> Expr a -> Expr Bool,
                 in an equation for 'eval'
        at Comparison2.hs:12:7-16
      'a1' is a rigid type variable bound by
        a pattern with constructor:
          Equals :: forall a. Eq a => Expr a -> Expr a -> Expr Bool,
        in an equation for 'eval'
        at Comparison2.hs:12:7-16
      Expected type: Expr a1 -> a
        Actual type: a1
    * The function 'eval' is applied to two arguments,
      but its type 'Expr a1 -> a1' has only one
      In the expression: eval x y
      In an equation for 'eval': eval (Equals x y) = eval x y
    * Relevant bindings include
        y :: Expr a1 (bound at Comparison2.hs:12:16)
        x :: Expr a1 (bound at Comparison2.hs:12:14)
   |
12 | eval (Equals x y) = eval x y
   |                     ^^^^^^^^
Comparison2.hs:13:27: error:
    * Couldn't match type 'Int' with 't0 (Int -> Int)'
      Expected type: t0 (Int -> a)
        Actual type: Int
    * In the first argument of 'maximum', namely '(eval x)'
      In the expression: maximum (eval x) (eval y)
      In an equation for 'eval':
          eval (Max x y) = maximum (eval x) (eval y)
   |
13 | eval (Max x y) = maximum (eval x) (eval y)
```

Figure 12. The type error message produced by GHC for the *eval* function

ours: it introduces new type variables, like *a1*, and mentions a context $(a \sim Bool, Eq\ a1)$ which we never had to provide. In the second error message, GHC says that it could not match *Int* with $t0\ (Int \to Int)$, and then goes on to say that the expected type is in fact $t0\ (Int \to a)$. Nowhere in the error message is the type variable *t0* introduced, neither is it mentioned that *maximum* should have gotten fewer arguments.

5 Related Work

Type error slicers present the programmer with information about *all* possible program points which contribute to the detected inconsistency. Skalpel [20] (a continuation of Haack and Wells [6]) implements type error slicers for Standard ML, supporting advanced SML features like modules, which are somewhat related to GADTs in Haskell. Schilling [21] adapts this idea to Haskell 98, but lacks support for local reasoning. The advantage of slicing is that the actual location that causes the problem is highlighted, a disadvantage is that many others locations are highlighted as well.

Because type error slices can be large, many researchers prefer to blame one or maybe a few constraints. For example, SHErrLoc [34] uses a graph-based structure to encode the solving process, and then ranks the likeness of a constraint being to blame using a Bayesian model. Their work considers type error reporting for modern Haskell, including local hypotheses. Chen and Erwig [2] explains type errors in Haskell programs using counter-factual typing, a version of variational typing in which they keep track of the different types that an expression may take. Although computationally somewhat costly, they can propagate type inconsistencies from one binding group to another. Pavlinovic et al. [16] achieves something similar by using an iterative deepening approach, in which the body of a binding is inlined in its usage site if a conflict is detected between both. This allows the inferencer to blame a location in the body of a (type correct) function if an application of that function is type incorrect, at the expense of repeatedly calling an SMT solver with a growing set of constraints. These papers perform only *error localization*.

In our work, we define specialized heuristics that recognize type error patterns, by examining a type graph. When we detect such a pattern, we not only know the location, but we can also explain about the pattern we detected, and for some patterns, even give a clue on how to fix the problem. A major influence on our work is [7] that introduces the type graphs we have extended in this paper, transplanting their heuristics and addding a number of GADT-specific ones.

Whenever the type system is extended, e.g., with type class information, extensions typically need to be made to the type graphs to represent these faithfully. The main technical contribution of this paper, is the design of a type graph structure that can represent constraint sets generated by OutsideIn(X), allowing us to represent local reasoning in type graphs. Type graphs were extended with type classes and row types in the setting of Elm [17], and Weijers et al. [32] uses heuristics to diagnose security type errors.

Some authors use a more complicated structure to diagnose type errors: [18] and [29] expose the trace of the type checker to the programmer (for Scala and OCaml, respectively), and Chitil [3] defines an explanation graph for Hindley-Miler type systems, which summarizes the information involved in type checking. LiquidHaskell [30] uses SMT solving as part of type checking. In those cases, *reverse interpolation* [15] can be used to derive a simpler explanation.

For the case that we have no control over the compiler infrastructure, Lerner et al. [13] presents an approach in which the compiler is iteratively queried for the well-typedness of modified versions the program, which are then ranked to present a solution to the type error. *Pointwise* GADTs [14] have been developed with better type error reporting in mind, by excluding pathological cases which are hard to explain. Others have used abduction to infer a common type for all branches in a GADT [25, 27]. In this case, reasoning is performed within a more complex framework, which is harder to explain to the programmer.

6 Conclusion and Future Work

We have extended Helium with GADTs and achieving good error diagnosis for a number of classes of inconsistent programs, as compared to GHC. We have extended Helium type graphs in order to model local reasoning in the type graph and defined GADT specific heuristics to help diagnose problems that involved GADTs. We have also transplanted all heuristics on vanilla type graphs to extended type graphs, so that for programs without GADTs we can expect to obtain the same type error messages [1]. This work is a major step in our endeavour to achieve good error diagnosis for advanced, but often used Haskell language extensions, including type class extensions, type families and higher-ranked types.

References

[1] Anon. [n. d.]. Reference omitted for anonymization.

[2] Sheng Chen and Martin Erwig. 2014. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2535838.2535863

[3] Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/507635.507659

[4] Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*. ACM, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

[5] María García de la Banda, Peter J. Stuckey, and Jeremy Wazny. 2003. Finding All Minimal Unsatisfiable Subsets. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (PPDP '03)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/888251.888256

[6] Christian Haack and J. B. Wells. 2004. Type Error Slicing in Implicitly Typed Higher-order Languages. *Sci. Comput. Program.* 50, 1-3 (March 2004), 189–224. https://doi.org/10.1016/j.scico.2004.01.004

[7] Jurriaan Hage and Bastiaan Heeren. 2007. Heuristics for Type Error Discovery and Recovery. In *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages (IFL'06)*. Springer-Verlag, Berlin, Heidelberg, 199–216. http://dl.acm.org/citation.cfm?id=1757028.1757040

[8] Jurriaan Hage and Bastiaan Heeren. 2009. Strategies for Solving Constraints in Type and Effect Systems. *Electron. Notes Theor. Comput. Sci.* 236 (April 2009), 163–183. https://doi.org/10.1016/j.entcs.2009.03.021

[9] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Constraint based type inferencing in Helium. In *Workshop Proceedings of Immediate Applications of Constraint Programming*, M.-C. Silaghi and M. Zanker (Eds.). Cork, 59 – 80.

[10] B. Heeren, D. Leijen, and A. van IJzendoorn. 2003. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*. ACM Press, New York, 62 – 71.

[11] Bastiaan J. Heeren. 2005. *Top Quality Type Error Messages*. Ph.D. Dissertation. Universiteit Utrecht, The Netherlands.

[12] Oukseh Lee and Kwangkeun Yi. 1998. Proofs About a Folklore Let-polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 707–723. https://doi.org/10.1145/291891.291892

[13] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-error Messages. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 425–434. https://doi.org/10.1145/1250734.1250783

[14] Chuan-kai Lin and Tim Sheard. 2010. Pointwise Generalized Algebraic Data Types. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1708016.1708024

[15] K. L. McMillan. 2004. An Interpolating Theorem Prover. In *Tools and Algorithms for the Construction and Analysis of Systems*, Kurt Jensen and Andreas Podelski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–30.

[16] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2015. Practical SMT-based Type Error Localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 412–423. https://doi.org/10.1145/2784731.2784765

[17] Falco Peijnenburg, Jurriaan Hage, and Alejandro Serrano. 2016. Type Directives and Type Graphs in Elm. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*. 2:1–2:12. https://doi.org/10.1145/3064899.3064907

[18] Hubert Plociniczak. 2013. Scalad: An Interactive Type-level Debugger. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, Article 8, 4 pages. https://doi.org/10.1145/2489837.2489845

[19] François Pottier and Didier Rémy. 2005. he Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489. http://cristal.inria.fr/attapl/

[20] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. 2017. Skalpel: A constraint-based type error slicer for Standard ML. *J. Symb. Comput.* 80, P1 (May 2017), 164–208. https://doi.org/10.1016/j.jsc.2016.07.013

[21] Thomas Schilling. 2012. Constraint-free Type Error Slicing. In *Proceedings of the 12th International Conference on Trends in Functional Programming (TFP'11)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-32037-8_1

[22] Alejandro Serrano. 2018. *Type Error Customization for Embedded Domain Specific Languages*. Ph.D. Dissertation. Universiteit Utrecht, The Netherlands.

[23] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag New York, Inc., New York, NY, USA, 672–698. https://doi.org/10.1007/978-3-662-49498-1_26

[24] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 783–796. https://doi.org/10.1145/3192366.3192389

[25] Vincent Simonet and François Pottier. 2007. A Constraint-based Approach to Guarded Algebraic Data Types. *ACM Trans. Program. Lang. Syst.* 29, 1, Article 1 (Jan. 2007). https://doi.org/10.1145/1180475.1180476

[26] Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. 2007. Understanding Functional Dependencies via Constraint Handling Rules. *J. Funct. Program.* 17, 1 (Jan. 2007), 83–129. https://doi.org/10.1017/S0956796806006137

[27] Martin Sulzmann, Tom Schrijvers, and Peter J Stuckey. 2008. Type inference for GADTs via Herbrand constraint abduction. https://lirias.kuleuven.be/retrieve/10888

[28] Swift Team. 2016. Type Checker Design and Implementation. https://github.com/apple/swift/blob/master/docs/TypeChecker.rst

[29] Kanae Tsushima and Kenichi Asai. 2013. An Embedded Type Debugger. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–206.

[30] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP ’14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

[31] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(x): Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

[32] Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. 2014. Security type error diagnosis for higher-order, polymorphic languages. *Science of Computer Programming* 95 (2014), 200 – 218. https://doi.org/10.1016/j.scico.2014.03.011 Selected and extended papers from Partial Evaluation and Program Manipulation 2013.

[33] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. 2000. Improved Type Error Reporting. In *In Proceedings of 12th International Workshop on Implementation of Functional Languages*. 71–86.

[34] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. Diagnosing Type Errors with Class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’15)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/2737924.2738009

A New Backend for Standard ML of New Jersey (Draft Paper)

Kavon Farvardin
Computer Science
University of Chicago
Chicago, IL, USA
kavon@farvard.in

John Reppy
Computer Science
University of Chicago
Chicago, IL, USA
jhr@cs.uchicago.edu

ABSTRACT

This paper describes the design and implementation of a new backend for the Standard ML of New Jersey (SML/NJ) system that is based on the LLVM compiler infrastructure. We first describe the history and design of the current backend, which is based on the MLRISC framework. While MLRISC has many similarities to LLVM, it provides a lower-level, policy agnostic, approach to code generation that enables customization of the code generator for non-standard runtime models (*i.e.*, register pinning, calling conventions, *etc.*). In particular, SML/NJ uses a stackless runtime model based on continuation-passing style with heap-allocated continuation closures. This feature, and others, pose challenges to building a backend using LLVM. We describe these challenges and how we address them in our backend.

KEYWORDS

Code Generation, Compilers, LLVM, Standard ML, Continuation-Passing Style

ACM Reference Format:
Kavon Farvardin and John Reppy. 2020. A New Backend for Standard ML of New Jersey (Draft Paper). In *Proceedings of 32nd Symposium on Implementation and Application of Functional Languages (IFL 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Standard ML of New Jersey is one of the oldest actively-maintained functional language implementations in existence [1, 7]. Much like the proverbial "Ship of Theseus," every part of the compiler, runtime system, and libraries has been reimplemented at least once, with some parts having been reimplemented half a dozen times or more.

The backend of the compiler is one such example. The original code generator translated a direct-style λ-calculus intermediate representation (IR) to Motorola 68000 and DEC VAX machine code [7]. Inspired by Kranz *et al.*'s work on the ORBIT compiler for Scheme [22, 23], Appel and Jim converted the backend of the compiler to use what they called a "Continuation-Passing, Closure-Passing Style" [3, 6].[1]

At the same time, additional machine-code generators were written for the MIPS and SPARC architectures, but with the proliferation of *Reduced-Instruction-Set Computers* (RISC) in the early 1990's, there was a need for more backends. These code generators also suffered from the problem that they did not share code, each was a standalone effort, and that they did not support many machine-code-level optimizations. These problems lead to the development of MLRISC [20] as a new, portable machine-code generator for SML/NJ. MLRISC defined an abstract load-store virtual-machine architecture that could sit between the language-specific parts of the code generator and the target-machine-specific parts, such as instruction selection, register allocation, and instruction scheduling. Over the past 25 years, MLRISC has been used to support roughly ten different target architectures in the SML/NJ system. It has also been used by several other compilers [14–16] and as a platform for research into advanced register allocation techniques [5, 19] and SSA-based optimization [27].

Unfortunately, MLRISC is no longer under active development,[2] so we need to consider alternatives. An obvious choice is the LLVM project, which provides a portable framework for generating and optimizing machine code [24, 25]. LLVM takes a language-centric approach to code generation by defining a low-level SSA-based [11] language, called LLVM IR, for describing code. LLVM IR has a textual representation, which we refer to as LLVM assembly code, as well as a binary representation, called *bitcode*, and a procedural representation in the form of a C++ API for generating LLVM IR in memory. The LLVM framework includes many analysis and optimization passes on both the target-independent LLVM IR and on machine-specific code. Most importantly, it supports the operating systems and architectures that SML/NJ supports, as well as some that we want to support in the future. While LLVM was originally developed to support C and C++ compilers, it has been used by a number of other functional-language implementations [12, 13, 26, 31, 36, 37].

Therefore, we are undertaking a project to migrate the backend of SML/NJ to use the LLVM infrastructure. This paper describes the challenges faced by this migration and how these challenges are being met. While there are many similarities between this effort and previous applications of LLVM to functional-language compilers,

IFL 2020, September 2–4, 2020, Online
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/1122445.1122456

[1]This CPS IR, with modifications to support multiple precisions of numeric types [17] and direct calls to C functions [9], continues to be used in the backend of the SML/NJ compiler.

[2]The last significant work was the addition of support for the amd64 (a.k.a., x86-64) architecture.

there are also a number of novel aspects driven by the SML/NJ runtime model and compiler architecture.

2 STANDARD ML OF NEW JERSEY

The Standard ML of New Jersey (SML/NJ) system provides both interactive compilation in the form of a *Read-Eval-Print Loop* (REPL) and batch compilation. In both cases, SML source code is compiled to binary machine code that is either loaded into a heap-allocated code object for execution or written to a file. Linking is handled in the elaborator, which wraps the compilation unit with a λ-abstraction that closes over its free variables; this code is then applied to the dynamic representation of the environment to link it. Dynamically, a compilation unit is represented as a function that takes a tuple of bindings for its free variables and returns a tuple representing the bindings that it has introduced. Thus, the SML/NJ system does not need to understand system-specific object-file formats or dynamic linking.

In the remainder of this section, we first describe SML/NJ's runtime conventions at an abstract level, then discuss the existing backend implementation, and the MLRISC-based machine-code generator.

2.1 Runtime Conventions

As described by Appel [2, 3], SML/NJ has a runtime model that can be described as a simple abstract machine (called the CMACHINE). The CMACHINE defines a small set of special *registers* to represent its state; these are:

- **alloc** is the allocation pointer, which points to the next word to allocate in the nursery.
- **limit** is the allocation-limit pointer, which points to the upper limit of the nursery minus a buffer of 1024 words. This buffer, which is called the *allocation slop*, allows most heap-limit tests to be implemented as a simple comparison.
- **store** is the store-list pointer, which points to a list of locations that have been modified since the last garbage collection (*i.e.*, it implements a write barrier).
- **exnptr** is the current-exception-handler pointer, which points to a closure representing the current exception handler.
- **varptr** is the var pointer, which is a global mutable location that can be used to implement features such as thread-local storage [28].
- **base** is the base-pointer register, which points to the beginning of the code object that holds the currently executing function. It is used to compute code addresses in a position-independent way.[3]

The **alloc** register is always mapped to a hardware register, the other special registers are either mapped to dedicated hardware registers or else represented by stack locations. For example, on the amd64 target, which has 16 general-purpose registers, the **alloc**, **limit**, and **store** registers are mapped to hardware registers, but the **exnptr** and **varptr** are represented by stack locations. The first five of these registers (**alloc**, **limit**, **store**, **exnptr**, and **varptr**) are live throughout the execution of SML code and, thus, are implicitly passed as parameters across calls. The **base** register is recomputed on entry to a function (since the caller and callee may be in different modules), and is threaded through the body of the function.

[3]Some architectures, such as the amd64, support PC-relative addressing, which can also be used for this purpose, but the SML/NJ backend currently does not take advantage of such addressing modes.

Table 1: CMACHINE general purpose registers

std-link	holds address of function for standard calls
std-clos	holds pointer to closure object for standard calls
std-cont	holds address of continuation
std-arg	first general-purpose argument register
misc$_i$	miscellaneous argument registers (including callee-save registers)

In addition, the compiler assumes that intermediate results, arguments to primitive operations, and arguments to function calls are always held in registers. The CMACHINE registers are assigned specific roles in the calling conventions as described in Table 1. Function calls come in three forms:

(1) *Standard function calls* are calls to "escaping" functions that use a standard calling convention; *i.e.*, functions where at least some call sites or targets are statically unknown.[4] The first three arguments of a standard function call are the function's address (**std-link**), its closure (**std-clos**), and return continuation address (**std-cont**). Following these arguments are k callee-save registers [8] (typically $k = 3$), which are assigned to the first k miscellaneous registers ($\mathbf{misc}_0, \ldots, \mathbf{misc}_{k-1}$). The remaining arguments correspond to the user arguments to the function and are mapped to registers by type; *i.e.*, pointers and integers are assigned to **std-arg**, $\mathbf{misc}_k$, $\mathbf{misc}_{k+1}$, *etc.*, and floating-point arguments are assigned to floating-point registers.
(2) *Standard continuation calls* are calls to "escaping" continuations. The first argument is the continuation's address and is assigned to the **std-cont** register; it is followed by the k callee-save registers, some of which are used to hold the continuation's free variables. The remaining arguments to the continuation are mapped to registers in the same way as for standard functions.
(3) *Known function calls* are "gotos with arguments" [34] that represent the internal control flow (loops and join points) in a standard function or continuation. Because the code generator knows both the entry and call sites for known functions, it is able to arrange for arguments to be passed in registers without unnecessary copying [19].

To illustrate how these conventions are realized in the CPS IR, consider the following trivial SML function:

```
fun f x = if (x < 1) then x else f (x-1);
```

The first-order CPS is a single cluster consisting of two CPS functions as shown below.

```
fun f (link, clos, k, cs1, cs2, cs3, arg) =
   lp (arg, k, cs1, cs2, cs3)

and lp (arg, k, cs1, cs2, cs3) =
```

[4]It should be noted that SML/NJ does not do any kind of sophisticated control-flow analysis, so escaping functions are quite common.

```
    if i63.>=(arg, 1) then
      let val tmp = isub63(arg, 1)
      in lp (tmp, k, cs1, cs2, cs3) end
    else
      k (k, cs1, cs2, cs3, arg)
```

Here we have taken the liberty of using meaningful variable names and an SML-like syntax for readability. The function `f` is a standard function, so its first three parameters are held in the **std-link**, **std-clos**, and **std-cont** CMACHINE registers. The next three parameters are the three callee-save registers followed by the function's actual argument (`arg`) in the **std-arg** register. The `lp` function is internal to the cluster, so the compiler is free to arrange its parameters in any order. The loop terminates by invoking the return continuation (`k`) using a standard continuation call. Here the first argument to the call (`k`) will be held in the **std-cont** register, then come the callee-saves, followed by the function's result in the **std-arg** register.

The code generator must support one other calling convention, which is the convention used to invoke the garbage collector (GC) [18]. This convention is a modified version of the standard function convention that uses a fixed set of registers (**link**, **clos**, **cont**, the callee-saves, and **arg**) as garbage-collection roots. Any additional live data, including all non-pointer register values (*e.g.*, untagged integer and floating-point registers), are packaged up in heap objects that are referred to by the **arg** register.

When a heap-limit check fails, control jumps to a block of code to invoke the GC. This code sets up the fixed set of root registers (as described above), fetches the address of an assembly-language shim from the stack and then does a standard call to the shim code, which, in turn, transfers control to the runtime system. After the GC finishes, control is returned back to the GC-invocation code, which restores the live variables and resumes execution of the SML code. Note that the return from the GC involves the exact same set of fixed registers that are passed as arguments, which is how the updated roots are communicated back to the program.

2.2 The Backend

The SML/NJ backend takes a higher-order continuation-passing-style (CPS) IR and, via a sequence of optimizing and lowering steps, produces a first-order CPS IR.[5] Unlike most other compilers, including other CPS-based compilers, SML/NJ foregoes use of a stack to manage calls and returns. Instead, all return continuations are represented by heap-allocated closures. The first-order CPS IR makes these closures explicit in the form of records and record selection operations. Because the runtime model uses heap-allocated continuation closures to represent function returns, the stack is not used in the traditional way. Instead, the runtime system allocates a single large frame that is used for register spilling and holding additional runtime values.

Along with this first-order IR, the compiler computes additional metadata about where heap-limit checks are needed and about which calling conventions should be used. This metadata is stored in auxiliary hash tables.

A program in the CPS IR is a collection of functions that represent both user functions and continuations. The body of a function is a CPS expression (`cexp`), where the leaves of an expression are applications. Thus, a `cexp` in the first-order CPS IR, where functions are not nested, can be viewed as an extended basic block [29].

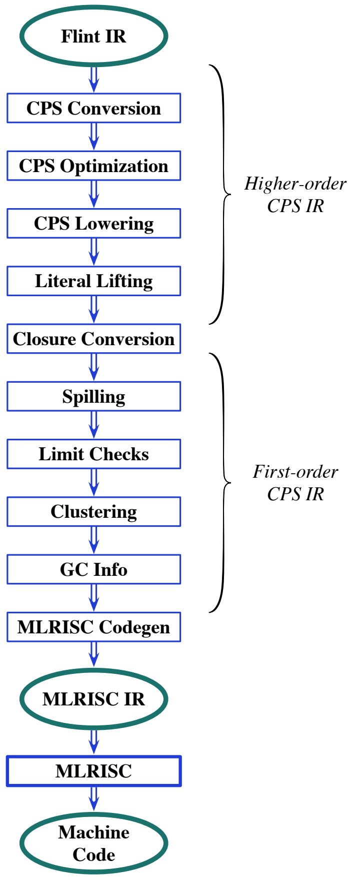


Figure 1: The existing backend

The phases of this backend are illustrated in Figure 1. We describe those passes that are directly affected by the design and implementation of the new backend.

- The *CPS Lowering* phase is responsible for expanding certain primitive operations (primops) into lower-level code.
- The *Literal Lifting* phase lifts floating-point and string literals (as well as data structures formed from these literals) out of the code and replaces them with references to a per-compilation-unit tuple of literal values.
- The *Spilling* phase ensures that the number of live variables never exceeds the fixed-size spill area (1024 words).[6]
- The *Limit Checks* and *GC Info* phases are responsible for determining where heap-limit checks should be added and determining the live variables at those points. Allocation checks are placed at the entry to functions (both escaping and known)

[5]Note that while the invariants for the IR change with lowering, the actual representation as SML datatypes does not.

[6]Appel's original code generator used the spilling phase to ensure that the number of live variables did not exceed the available machine registers [3], but the switch to MLRISC, which had a proper register allocator, relaxed this constraint.

and continuations. As discussed above, most functions allocate less than 1024 words, so the allocation slop allows us to simply compare the allocation and limit pointer for these checks.

- The *Clustering* phase groups CPS functions into clusters, which are connected graphs of CPS functions where the edges correspond to *known* function calls. The entry nodes for a cluster are escaping functions and continuations; note that a cluster may have more than one entry.

2.3 MLRISC

The final step of the backend is to generate machine code using the MLRISC framework. MLRISC was designed to address many of the same problems as LLVM; it provides an low-level virtual machine based on a load-store (*i.e.*, RISC-like) model. More so than LLVM, MLRISC is a "mechanism, not policy," design leaving ABI issues such as calling conventions, stack layout, register usage, *etc.*, up to the compiler writer.[7] It makes heavy use of SML's functors to support specialization for both the target architecture and the source language. For example, the register allocator is defined by a functor that is parameterized over the spilling mechanism, which gives the compiler writer control over stack layout.

MLRISC's policy agnostic approach was heavily influenced by the needs of SML/NJ's runtime model. SML/NJ's stackless execution model meant that calling conventions could not be baked into the design. Likewise, use of dedicated registers for the allocation pointer, *etc.*, and in the standard calling conventions meant that MLRISC had to support some form of register pinning. The MLRISC register allocator is also able to handle the multi-entry functions that can arise from the clustering phase. Lastly, the need to generate binary machine code meant that MLRISC required an integrated assembler to resolve local branch offsets, but that it did not require a direct mechanism for generating object files.

3 CHALLENGES TO USING LLVM

LLVM was originally designed to support C and C++ compilers and, as such, maintains a significant architectural bias toward conventional runtime models. Furthermore, because it embeds significant policy decisions about calling conventions, exception-handling mechanisms, garbage collection support, *etc.*, using it as a backend for a non-standard language runtime is challenging. In this section, we enumerate some of the mechanisms that our MLRISC backend uses that do not have direct analogues in LLVM. We also discuss the challenges of incorporating a code generator implemented in C++ into a compiler written in SML. In this discussion, we are focusing on the vanilla LLVM IR; as we describe in the next two sections, LLVM does provide ways to work around these limitations.

3.1 Comparing MLRISC and LLVM

MLRISC and LLVM are both designed to provide support for portable compilers. They are both based on a load-store model with an infinite supply of pseudo registers and a fairly standard set of basic instructions.A major difference, however, is that MLRISC abstracts over the instruction-set architecture, but not over the system ABI or runtime conventions. LLVM, on the other hand, has built in support for calling conventions, object-file formats, exception-handling mechanisms, garbage-collection metadata, and debugging information. Another major difference is in how they are used. While both systems define a virtual machine that a code generator can target, MLRISC only supports a procedural interface for code generation, whereas LLVM provides LLVM assembly, LLVM bitcode, and a procedural interface for code generation. The combination of builtin runtime conventions plus a textual representation of LLVM IR means that the only way to support different runtime models is to make changes to the LLVM implementation itself.

3.2 Limitations of the LLVM Model

Many of the issues that we face are a consequence of the fact that LLVM abstracts away from the runtime model to a much greater degree than MLRISC.

No direct access to hardware registers. Ths SML/NJ runtime model relies on being able to map key CMACHINE registers, such as the allocation pointer, to dedicated hardware registers for efficient execution. Unlike MLRISC, LLVM does not provide any mechanism for mapping variables to specific hardware registers.

No direct access to the stack. SML/NJ uses specific slots in the stack frame to communicate information from the runtime system to the SML execution (*e.g.*, the address of the `callGC` function). Some CMACHINE registers on some targets are also represented by stack locations. In LLVM, however, the layout of a function's stack frame is largely opaque at the LLVM IR level and there is no way to specify access to specific locations in the stack.

Builtin calling conventions. As described in Section 2.1, SML/NJ defines its own register-based calling conventions that do not involve the stack in any way, as well as a stack-based convention for invoking the garbage collector. The **`call`** instruction in LLVM is a heavyweight operation that embodies the policy defined by its calling convention. While LLVM has a number of predefined calling conventions, including several language-specific ones, there is not a good match for the SML/NJ runtime. Defining a different convention requires modifying the LLVM source and recompiling the LLVM libraries.

Multi-entry-point functions. The clustering phase of the SML/NJ backend produces clusters that can have multiple entry points. For example, compiling the following function that walks over a binary tree

```
fun walk Lf = ()
  | walk (Nd(l, r)) = (walk l; walk r)
```

will produce a cluster for `f` with two entries: a standard function for calling `f` on the root or left subtree and a second continuation entry for calling `f` on the right subtree. While it is natural to think of mapping clusters to LLVM functions; LLVM functions are restricted to a single entry point.

Tail-call overhead. Efficient tail calls are critical to performance, since all calls in CPS are tail calls. While LLVM provides a tail-call optimization (TCO), its primary purpose is to avoid stack growth. Even when TCO is applied to a function call, the resulting code

[7] It does provide some higher-level mechanisms, such as implementations of various C-language calling conventions;

incurs the overhead of deallocating the caller's frame and then allocating a fresh frame for the callee.

No trapping arithmetic. The semantics of integer arithmetic in SML require that the `Overflow` exception be raised when the result exceeds the representable range of the type [17]. MLRISC supports this requirement by defining *trapping* versions of the arithmetic operations, with the semantics that an appropriate trap is signaled on overflow. The runtime system handles this trap by mapping it to a control transfer to the exception handler continuation. While LLVM provides intrinsic functions for integer arithmetic with overflow, it does not provide a mechanism for generating an appropriate trap. While we could generate the control transfer to the exception handler in LLVM, we do not have access to the `Overflow` exception at that point.

Support for position-independent code. The machine code that SML/NJ uses must be position independent. We achieve this property by using the base pointer to compute absolute addresses from relative offsets, both for evaluating labels and for jump tables. While LLVM also supports position independent code, it does so by relying on a dynamic linker to patch code when it is loaded.

3.3 Integrating LLVM into the Compiler

There are two ways that one might use LLVM as a backend for a compiler. The first, which is most common, is to generate LLVM assembly code into a text file and then use the command-line toolchain to convert that to a target-machine object file.[8] This approach has the advantage that it does not require a foreign-function mechanism to communicate between the compiler and LLVM. The downside, however, is that it adds significant overhead in the form of formatting textual output, parsing said output, and running subprocesses. For an interactive compiler, such as SML/NJ's REPL, this approach also requires using system-specific dynamic linking to load and execute the code that was just generated.

The other way to use LLVM, which is used by industrial compilers like the **clang** C/C++ compiler, is to use LLVM's C++ APIs to construct a representation of the program directly, which can then be optimized and translated to machine code. This approach is similar to what we currently do with MLRISC, but it poses its own challenges. First of all, the C++ API for LLVM relies heavily on inline functions, which cannot be called from foreign languages. As an alternative, there is a C language wrapper for the C++ API that can be used, but it is less efficient than the C++ API and has a reputation of lagging behind changes in the C++ API. Another problem is the sheer volume of foreign calls that would be required for code generation. Given that foreign function calls in many functional-language implementations, including SML/NJ, are relatively expensive, this volume can add measurable overhead to code generation. Thus, the problem of efficient communication between the compiler and the code generator is a challenge for using LLVM as a library.

The last challenge to using LLVM for SML/NJ is that it produces object files (the specific object-file format depends on the system). For implementations that use traditional linking tools, this property is not an issue, but for a system like SML/NJ that works with raw code objects, it is necessary to extract the code from the object file.

[8]Typically, this toolchain involves using **llc** to generate native assembly code and then running an assembler to produce object code.

4 DESIGN OF THE NEW BACKEND

In order to use LLVM in the SML/NJ system, we need to solutions to the two broad challenges described above: how to support the SML/NJ runtime model in LLVM (Section 3.2) and how to integrate a LLVM-based backend into a compiler written in SML (Section 3.3).

4.1 Runtime conventions

Function entries and call sites are the key places where we need to guaranteed that our register conventions are being followed, elsewhere in the function we can let the register allocator dictate where information is held. Thus, by modifying LLVM to add a new calling convention, we can dictate the register usage at those places. In previous work for the Manticore system [12], we described a new calling convention for LLVM, called *Jump With Arguments* (JWA), that can be used to support the stackless, heap-allocated-closure runtime model used by both Manticore and SML/NJ. The JWA calling convention has the property that it uses almost all of the available hardware registers for general-purpose parameter passing.[9] The convention also has the properties that no registers are preserved across calls and that the return convention uses exactly the same register convention as calls.

We furthermore mark every function with the **`naked`** attribute, which tells LLVM to omit generating the function prologue and epilogue.[10] Thus the function will run in whatever stack frame exists when it is called, which fits the SML/NJ model of a single frame shared by all SML code.

There is one minor complication, which is that we actually have several different conventions to support (*i.e.*, escaping and known functions, continuations, and GC invocation). While we could define multiple LLVM conventions, we can make them all fit within the JWA convention by careful ordering of parameters and by using LLVM's undefined values for registers that are not part of a particular convention (*e.g.*, the **link** and **clos** registers when throwing to a `STD_CONT` fragment).

4.2 Integrating LLVM into SML/NJ

Replacing MLRISC with LLVM raises the question of how to connect the SML/NJ compiler, written in SML, with an LLVM code generator, written in C++. Previous functional-language implementations have generated LLVM assembly code and used a command-line toolchain to translate that into object code, but we decided that this approach was not a good fit for SML/NJ. Specifically, we were concerned about compilation latency, since the interactive REPL is a central part of the SML/NJ system, and about the extra dependencies on executables that we would have to manage. Therefore, we decided to integrate the LLVM libraries into the runtime system.

Having decided to directly generate LLVM code in memory, there was the question of how to do that efficiently. Fortunately,

[9]For SML/NJ, we use the same register convention that is used in the existing MLRISC backend. On the amd64, we omit the stack pointer and one scratch register from the convention, which leaves 14 registers available for parameter passing.
[10]The function prologue and epilogue is where the function's stack frame is allocated and deallocated.

the problem of how to connect compiler components that are implemented in different languages was addressed many years ago as part of the *Zepher Compiler Infrastructure Project* [38]. Zepher defined the *Abstract Syntax Description Language* (ASDL) for specifying compiler intermediate representations and a tool (**asdlgen**) for generating picklers and unpicklers in multiple languages. The original **asdlgen** tool does not support modern C++, so we built a new implementation of the tool that generates code that is compatible with LLVM.[11]

Our plan then was to use an **asdlgen** pickler to serialize the CPS IR, which would be passed to the runtime system to be the input to a LLVM-based code generator that would essentially be a C++ rewrite of the existing MLRISC code generator. The resulting machine code would then be returned to the SML code as an array of bytes. As we began work on this approach, however, we discovered that the CPS IR was not necessarily the right IR for connecting to LLVM. First, the MLRISC code generator depended heavily on metadata that was external to the CPS IR. Second, the CPS primops were designed to model the corresponding SML operations (*e.g.*, addition on tagged integers), which added a lot redundancy and extra work to the code generation process. Thus, we decided to introduce a new, lower-level IR, that would be the vehicle for communicating with the LLVM-based code generator. This new IR, which we call the CFG IR, is described in detail in the next section, but its key features are that it is self-contained and that its semantics are much closer to both the semantics of LLVM and MLRISC. The latter is important, because we decided to support a second code-generation path that uses MLRISC as both a way to validate the translation to CFG and to support legacy systems, such as the 32-bit x86, for which we do not plan to provide an LLVM-based backend.

4.3 The New Backend Pipeline

We conclude this section with a description of the new backend pipeline, which is illustrated in Figure 2. We have greyed out the labels of those passes from Figure 1 that are unchanged, but, for some passes, changes were required.

- The CPS Lowering phase is has been expanded to lower more CPS primops than before. These changes were made to avoid some primops that were difficult to translate directly to LLVM.
- The Clustering phase was modified to avoid multi-entry-point clusters, which requires introducing new CPS functions.
- The tracking of information about GC invocations was modified to work with the CFG code generator (discussed below in Section 6.5).
- The *CFG Codegen* phase replaces the old MLRISC Codegen phase.

Once we have produced the CFG IR, there are two paths to machine code. The *legacy* path (on the left) compiles the CFG to MLRISC and then uses the existing MLRISC backend to produce machine code.

The new code generation path first pickles the CFG IR and then passes the linearized representation to the runtime system where it

[11]The original implementation is still available at http://asdl.sourceforge.net; the new implementation, which currently only supports SML and C++ is included in the SML/NJ distribution.

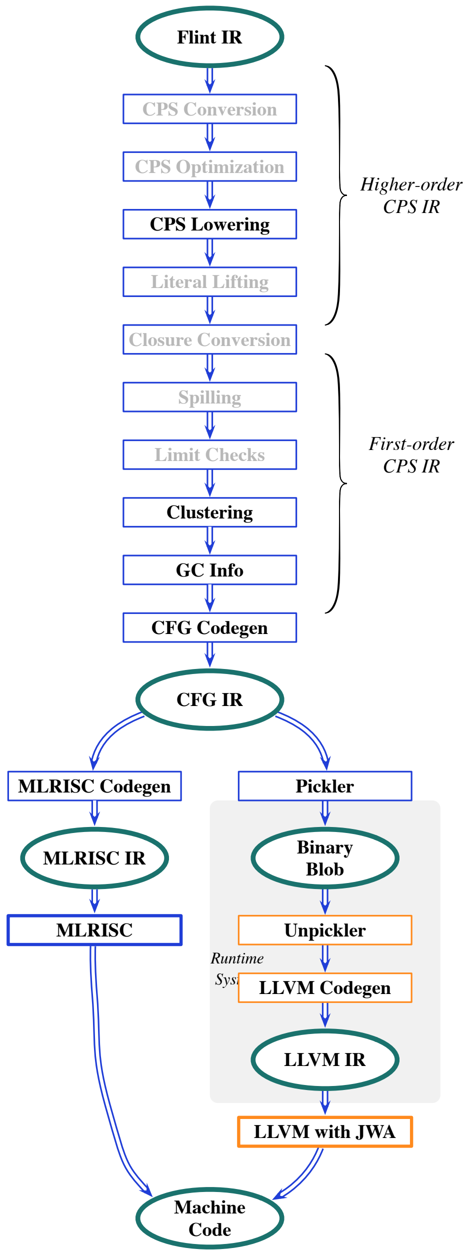


Figure 2: The new backend. Components represented by orange boxes are implemented in C++.

is unpickled into a C++ representation of the CFG IR. We then generate LLVM IR code using a version of LLVM (currently 10.0.1) extended with the JWA calling convention. For the new code generator, the GC Info pass is part of the LLVM Codegen pass, where we use the function's calling convention and parameter signature to determine the live variables. The next two sections describe the CFG and LLVM code generator in detail.

5 THE CFG REPRESENTATION

A major part of the new backend is the new CFG IR that sits between the existing first-order CPS IR and the MLRISC and LLVM code generators. The CFG IR encodes many of the invariants of the CPS IR into its representation and makes the metadata required for code generation explicit. The main datatypes used to represent the CFG IR are shown in Figure 3; we omit the primitive operators and have simplified the types slightly for space and presentation reasons.

Each unit of compilation (*e.g.*, declarations or expressions typed into the REPL or a source file) is mapped to a CFG compilation unit, which consists of a list of clusters. The first cluster in the list is the entry cluster, which handles linking the new code with the existing dynamic environment. CFG clusters roughly correspond to the clusters used in the MLRISC backend; each cluster consists of a list of *fragments*, which are extended basic blocks. Clusters also have attributes, which capture some basic information about the code in the cluster, such as does it require the base pointer register.

In the LLVM backend, clusters map to LLVM functions, which means that they must have a single entry point (unlike the clusters used in the MLRISC backend, which can have multiple entry points). Because of this restriction, we have modified the clustering phase to optionally split multi-entry-point clusters into several clusters.[12] The one complication for this splitting is that the new clusters may require access to the base pointer in order to compute label values. The original calls to these new clusters are unlikely to have the cluster's address as a parameter, since they are not standard calls. Thus, we have to change the calling convention slightly in these cases by adding the base pointer as an additional parameter. In the rare case that the original function uses all of the available general-purpose registers, we pass the base pointer using a dedicated stack location.

5.1 Expressions and Statements

CFG expressions (`exp`) and statements (`stm`) are used to define the computations that make up the body of fragments. While the constructors of these datatypes are in close correspondence to the CPS IR, there are some important differences.

First, pure expressions are represented as trees (the `exp` type), instead of having each primitive operation be bound to a `lvar`. Sharing of common expressions is made explicit by the `LET` constructor. Using expression trees has a couple of advantages: it reduces the size of CFG terms, which speeds pickling, and expression trees match the procedural code-generation interfaces of both LLVM and MLRISC.

Operations in the CFG IR are closer to machine level than those of the CPS IR. For example, the default integer type in SML is represented by a tagged value that has its lowest bit set (*i.e.*, the integer n is represented as $2n + 1$). Arithmetic on tagged integers requires various additional operations to remove and add tags. In the old backend, these were added when generating MLRISC code; we now generate these operations as part of the translation to CFG. The CFG IR also replaces many specialized CPS operations for memory allocation and access with a few lower-level mechanisms.

Figure 3 also shows the representation of types in the CFG IR. The types `LABt` (code addresses), `PTRt` (pointers or tagged values), and `TAGt` (tagged values) describe values that the garbage collector

```
datatype ty
  = LABt | PTRt | TAGt
  | NUMt of {sz : int} | FLTt of {sz : int}

type param = lvar * ty

datatype exp
  = VAR of {name : lvar}
  | LABEL of {name : lvar}
  | NUM of {iv : IntInf.int, sz : int}
  | LOOKER of {oper : looker, args : exp list}
  | PURE of {oper : pure, args : exp list}
  | SELECT of {idx : int, arg : exp}
  | OFFSET of {idx : int, arg : exp}

datatype stm
  = LET of exp * param * stm
  | ALLOC of alloc * exp list * lvar * stm
  | ARITH of arith * exp list * param * stm
  | SETTER of setter * exp list * stm
  | APPLY of exp * exp list * ty list
  | THROW of exp * exp list * ty list
  | GOTO of lvar * exp list
  | SWITCH of exp * stm list
  | BRANCH of branch * exp list * stm * stm
  | CALLGC of exp list * lvar list * stm

datatype frag_kind
  = STD_FUN | STD_CONT | KNOWN
  | INTERNAL

datatype frag = Frag of {
    kind : frag_kind,
    lab : lvar,
    params : param list,
    allocChk : word option,
    body : stm
  }

type attrs = { ... }

datatype cluster = Cluster of {
    attrs : attrs, frags : frag list
  }

type comp_unit = cluster list
```

Figure 3: The main CFG types

can parse and thus can be in a GC root. The other two types represent raw numeric data (integer and floating-point) of the specified size in bits. We map the `LABt` and `PTRt` types to the LLVM **`i64*`** type (**`i32*`** on 32-bit machines). The `TAGt` type is mapped to **`i64`**, while the `INTt` and `FLTt` types are mapped to the LLVM integer and float types of the specified size. We do not try to use LLVM's

[12]When using the MLRISC backend, this splitting is not necessary.

aggregate types to model heap allocated objects, since we usually only have that level of type information at the point of allocation.

5.2 Metadata

The other major difference between the CPS and CFG IRs is that the metadata for calling conventions and GC support has been incorporated into the CFG IR, instead of being held in external tables. This change makes transferring the information to the LLVM code generator much simpler, since we do not have to define a pickle format for the hash tables used to track the data.

The calling-convention metadata is represented by three aspects of the IR:

(1) Fragments are annotated with a `frag_kind`; `STD_FUN` for escaping functions, `STD_CONT`, for continuations, and `INTERNAL` for internal known function calls. The `KNOWN` kind is used for the functions that are introduced to avoid multiple entry-points during clustering.
(2) We use three different application forms: `APPLY` for functions, `THROW` for continuations, and `GOTO` for internal jumps. Calls to `KNOWN` functions are represented by an `APPLY` where the function is specified by a `LABEL` value.
(3) The `APPLY` and `THROW` constructs include the type signature of their arguments.

As seen in Figure 3, each fragment is annotated with an `allocChk` field that contains an optional unsigned integer. A value of `SOME` n signifies the need for a heap limit check at the beginning of the fragment. The most common case is where the fragment's allocation is less than the allocation slop, in which case $n = 0$. For fragments that can allocate more than the allocation slop amount, n is the upper bound on their allocation requirements.

5.3 C++ Representation

The CFG IR is defined using the ASDL specification language [30], which provides mechanisms for inductive types similar to those found in most functional programming languages. From this specification, we generate both the SML and C++ representations of the IR, as well as the pickling/unpickling code needed to communicate CFG values from SML to our LLVM code generator. As would be expected, the mapping from ASDL to SML types is straightforward. For C++, most types are represented as classes, but enumerations (*e.g.*, `frag_kind` in Figure 3) are mapped to C++ **`enum`** types. Sum types are represented with an abstract base class for the type and subclasses for the constructors.

6 IMPLEMENTATION DETAILS

In this section, we describe the LLVM code generator (*i.e.*, the orange boxes in Figure 2) in more detail. Our current prototype targets the amd64 architecture, but is almost entirely machine independent, so we expect that porting to other architectures will be straightforward.

6.1 LLVM Code Generation

As described above, the `exp` and `stm` types in the CFG IR are represented as abstract classes in C++, with each constructor its own subclass. Code generation is implemented as a two-pass walk over the CFG IR. The first pass collects information, such as a mapping from labels to clusters and fragments, and allocates placeholder objects, such as LLVM functions for clusters, LLVM ϕ-nodes for `INTERNAL` fragments, and LLVM basic blocks for the arms of `BRANCH` and `SWITCH` statements. The second pass walks the representation generating LLVM code.

ASDL provides a mechanism for adding methods to the generated classes. For the `cluster`, `frag`, and `stm` classes, we define a virtual `init` method for the initialization pass. We also define a virtual `codegen` method for these classes and for the `exp` and various primitive operator classes. Dispatching on the constructor of a sum type is implemented using the standard object-oriented pattern of virtual-method dispatch.

The code generation process requires keeping track of a significant amount of state, such as the current LLVM module, function, and basic block, and maps from `lvar`s to their LLVM representations. We define the `code_buffer` class to encapsulate the current state of the code generator as well as target-specific information. The `code_buffer` class also contains the implementation of various utility methods to support the calling conventions and GC invocation. We create a single object of this class, which is passed by reference to the `init` and `codegen` methods. Code generation for most of the CFG IR is straightforward, but we explain how we address the challenges of Section 3 in the sequel.

6.2 ϕ Nodes

LLVM's language is a Static-Single-Assignment (SSA) IR [11]. As the name suggests, variables (a.k.a. pseudo registers) in SSA are assigned to only once. When control flows into a block from multiple predecessors, it is necessary to introduce ϕ nodes, which make explicit the merging of values from multiple sources. Generating the SSA form from the CFG IR is quite straightforward.[13] During the initialization pass, we preallocate ϕ nodes for each `INTERNAL` fragment in a cluster. We define one ϕ node per fragment parameter plus additional nodes for those special registers that are mapped to hardware registers (*e.g.*, **alloc**, **limit**, *etc.*). When compiling a `GOTO` statement, we record the current values of the special registers and the values generated for the `GOTO`'s arguments in the ϕ nodes of the target fragment.

6.3 Stack References

As discussed in Section 3.2, we need to be able to generate references to specific locations in the stack frame. We have experimented with several possible mechanisms for accessing stack locations. Our first attempt was the **`@llvm.frameaddress`** intrinsic, but it requires using a frame pointer, which burns an additional register. We then took the approach of defining native inline assembly code for reading and writing the stack. This approach produced the desired code, but also introduced target-dependencies in the code generator. We finally settled on using the **`@llvm.read_register`** intrinsic to read the stack pointer.

One change that we had to make to our runtime model is the layout of the frame used to host SML execution. In the existing MLRISC code generator, the spill area is in the upper part of the

[13] As has been observed by others [4, 10, 21], there are strong similarities between λ-calculus IRs (especially CPS) and SSA form.

frame and the locations used to represent special registers, *etc.* are in the lower part of the frame. For the LLVM code generator, we have to swap this around to match LLVM's frame layout conventions. Fortunately, MLRISC makes it easy to specify the location of the spill area, so we can modify the MLRISC backend to be compatible with LLVM.

6.4 Position-independent Code

As described in Section 2.1, we generate code to explicitly maintain a pointer to the beginning of the current module as a mechanism to support position-independent code. For example, if the first function in a module has label l_0 and we have a standard function f with label f_l, then we can compute the base pointer by **base** = **link** − $(l_f - l_0)$, where $(l_f - l_0)$ is a compile-time constant. While at first glance, it seems easy to encode this computation in the LLVM code generator, but it turns out that LLVM, by default, leaves the computation of $(l_f - l_0)$ to link time. We were able to work around this problem by defining LLVM *aliases* for the compile-time constant expressions.

In practice, we only need to generate code for the base pointer when the cluster requires it (*i.e.*, when a `LABEL` is used in a non-application position or if the code contains a `SWITCH`). We record this information in the `attrs` record associated with each cluster.

6.5 Invoking GC

As described in Section 2.1, invoking the GC requires a fair amount of bookkeeping to preserve live data across the invocation. What makes it complicated is the combination of different cases that have to be managed. For example, a `STD_CONT` fragment does not use the **std-link** or **std-clos** registers, so these are either used to hold excess parameters or else must be nullified before the collection. Our original implementation put the generation of this bookkeeping code in the C++ code generator, but the resulting code was both lengthy and complicated. While the MLRISC code generator also dealt with this complexity, it is a problem that is much easier to solve in SML than C++. We subsequently realized that a better strategy is to encode the GC invocation code in the CFG IR. To this purpose, we added a heap-limit check as a `branch` primop and the `CALLGC` statement form. The translation from CPS to CFG handles the generation of code to invoke the GC, as well as inserting the limit checks into the IR. In addition to moving complexity out of the C++ code generator, this approach also allows us to share the implementation of the GC invocation protocol between the LLVM and legacy MLRISC machine-code generators.

We also implement a feature of the MLRISC code generator that shares implementations of the GC invocation code between multiple `STD_FUN` and `STD_CONT` fragments. Because the parameters of these fragments are in known locations and the code address of these fragments are in known registers (*i.e.*, **std-link** or **std-cont**), we can move the invocation code into a function that can then be shared. Measurements done when the GC API was originally designed show that over 95% of `STD_FUN` GC invocations can be covered by five different functions, while almost 95% of `STD_CONT` invocations can be covered by just one invocation function [18].

The actual invocation of the GC uses a non-tail JWA call. We use the JWA calling convention so that the GC roots are in predictable registers and we mark the call as a non-tail call so that the runtime can return to the GC invocation code. The return type of the call is a struct with fields for each of the GC roots (recall that the JWA call uses the same register assignment for calls and returns). These are then bound to the variables specified by the `CALLGC` statement.

6.6 Trapping arithmetic

To implement trapping arithmetic, we use LLVM's "arithmetic with overflow" intrinsic functions. These functions return a pair of their result and an overflow bit. In the generated LLVM code, we test the overflow bit and jump to a per-cluster piece of code that forces a hardware trap to signal the overflow. As with the MLRISC version, the runtime system handles the trap by dispatching an `Overflow` exception to the current exception handler. The need for this conditional control flow is one of the reasons why trapping arithmetic is represented as a `stm` in the CFG IR. LLVM does not provide a mechanism to generate the particular trap instruction that we need on the amd64, so we use its inline native assembly code mechanism to inject an "`int 4`" instruction into the generated code. For example, the SML function

```
fun f (x : Int64.int) = x + 42
```

results in the LLVM code shown in Figure 4.

6.7 Just-in-Time Compilation

LLVM provides rich support for just-in-time (JIT) compilation, but the JIT infrastructure is primarily focused on the problems of multi-threaded compilation, compilation on demand, and dynamic linking. While multi-threaded compilation is a feature that we might want to explore in the future, we already address the problems of compilation on demand and linking in SML/NJ. Therefore, we use the batch compilation infrastructure, but specify an in-memory output stream for the target of the machine-code generator, which produces an in-memory object file. While the actual format of the object file (*e.g.*, ELF *vs.* COFF *vs.* MACH-O), depends on how the LLVM `TargetMachine` object is configured, we can use generic operations provided by LLVM to identify and extract the code from the in-memory representation. We copy the code into a heap-allocated code object, which is returned to the SML side of the compiler.

7 EVALUATION

Since we are still in the process of shaking out the bugs in our implementation, we have not yet been able to evaluate the approach for either compile time, or the quality of the generated code. Based on the performance differenced between our LLVM and MLRISC backends for Manticore [12], we expect to see some improvement in the performance of generated code.

We will include a detailed evaluation of the new backend in the final version of the paper.

8 RELATED

The PURE programming language[14] appears to have been the first functional language to use LLVM in its implementation (starting in 2008). The implementation of the PURE interpreter is in C++ and LLVM is described in the documentation as being used as a JIT

[14] See https://agraef.github.io/pure-lang.

```
define private jwa void @fn207 (i64** %allocPtr, i64** %limitPtr, i64** %storePtr,
  i64* %0, i64* %1, i64* %2, i64* %3, i64* %4, i64* %5, i64 %6) naked
{
  %8 = call { i64, i1 } @llvm.sadd.with.overflow.i64 (i64 %6, i64 42)
  %9 = extractvalue { i64, i1 } %8, 1
  br i1 %9, label %13, label %10

10:
  %11 = extractvalue { i64, i1 } %8, 0
  %12 = bitcast i64* %2 to void (i64**, i64**, i64**, i64*, i64*, i64*, i64*, i64*, i64*, i64)*
  tail call jwa void %12 (i64** %allocPtr, i64** %limitPtr, i64** %storePtr,
    i64* undef, i64* undef, i64* %2, i64* %3, i64* %4, i64* %5, i64 %11)
  ret void

13:
  call void asm sideeffect "int␣$$4", ""() #2
  ret void
}
```

Figure 4: Example of LLVM code for trapping arithmetic

compiler, but there is no published description of the implementation.

Terei and Chakravarty's LLVM-based code generator for the Glasgow Haskell Compiler (GHC) [35, 36] is probably the earliest attempt to use LLVM for a language with a non-standard runtime model. As such, they were the first to confront and solve a number of the technical issues we describe here. In particular, they faced the problem of how to map logical registers in their runtime model to specific machine registers. It appears that Chris Lattner, the creator of LLVM, suggested defining a new calling convention to implement this mechanism.[15] The GHC calling convention is now a supported convention in LLVM.

The ErLLVM pipeline is an LLVM-based backend for the HiPE Erlang compiler [31]. As with GHC, and our system, the problem of targeting specific machine registers is solved with a new calling convention; the HiPE convention is also part of the official LLVM distribution. Unlike GHC and SML/NJ, ErLLVM uses, with some adaptation, LLVM's builtin mechanisms for garbage collection support and exception handling. The ErLLVM pipeline generates LLVM assembly and then uses the LLVM and system tools to produce an object file. They then parse the object file to extract a representation that is compatible with the HiPE loader, which is similar to what we do in SML/NJ.

We know of two other ML implementations that have LLVM backends. The SML# system generates fairly vanilla LLVM assembly code and uses LLVM's existing `fastcc` calling convention [37]. To ensure that tail recursion is efficient, they added loop detection to their compiler and generate branches in these cases, instead of relying on LLVM's tail-call optimization.[16]

The MLton SML compiler also has a LLVM backend [26]. Their LLVM compiler is modeled on their backend that generates C code, so they do not have the problems of mapping specialized runtime conventions onto LLVM. As with GHC and ErLLVM, they generate LLVM assembly code; one difference, however, is that they stack allocate all variables and then rely on LLVM's `mem2reg` pass to convert to SSA.

Our work reported here has as its roots the development of the JWA calling convention for use in Manticore's Parallel ML (PML) compiler [12]. As with the other examples above, the PML compiler generates LLVM assembly and uses the **llc** tool to generate native assembly code. Because PML programs are linked using standard tools, the compiler does not require special handling of position-independent code or global addresses, such as the code to invoke the GC. It also does not require access to specific locations in the stack. While PML is a dialect of SML, it has a different semantics for arithmetic (*i.e.*, no `Overflow` exceptions), so it was not necessary to use LLVM's arithmetic with overflow intrinsics.

Recently, we have used the PML compiler to explore performance and implementation tradeoffs between different runtime strategies for representing continuations and the call stack [13]. The implementation of heap-allocated continuations in that study was the version from our previous work [12], which lacks the more sophisticated closure optimizations implemented by the SML/NJ compiler [8, 32, 33]. It will be interesting to revisit the experiments using our new LLVM backend for SML/NJ.

9 CONCLUSION

We have described our ongoing effort to port the SML/NJ system to a new backend based on LLVM. The code generator that takes pickled CFG IR and generates LLVM code using the C++ API is complete and we are currently testing it as a standalone program that generates code for the 64-bit amd64 architecture. The other major components of the new backend are also complete and being tested.

For the final paper, we expect to have the code generator incorporated into the SML/NJ runtime system and plan to report on the

[15]See http://nondot.org/sabre/LLVMNotes/GlobalRegisterVariables.txt.
[16]Recall from Section 3 that LLVM's tail-call optimization does not avoid the overhead of allocating/deallocating stack frames.

compile and runtime performance of the new backend. We are also planning a 64-bit ARM port of the system, which would be a new architecture for SML/NJ. Since the code generator is largely machine independent, we expect that this port should be fairly smooth.

REFERENCES

[1] Andrew Appel and David B. MacQueen. 1991. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming (PLILP '91) (Lecture Notes in Computer Science, Vol. 528)*, J. Maluszynski and M. Wirsing (Eds.). Springer-Verlag, New York, NY, USA, 1–13. https://doi.org/10.1007/3-540-54444-5_83

[2] Andrew W. Appel. 1990. A Runtime System. *Lisp and Symbolic Computation* 3, 4 (Nov. 1990), 343–380. https://doi.org/10.1007/BF01807697

[3] Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, UK.

[4] Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Notices* 33, 4 (April 1998), 17–20. https://doi.org/10.1145/278283.278285

[5] Andrew W. Appel and Lal George. 2001. Optimal Spilling for CISC Machines with Few Registers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '01)* (Snowbird, UT, USA). Association for Computing Machinery, New York, NY, USA, 243–253. https://doi.org/10.1145/378795.378854

[6] A. W. Appel and T. Jim. 1989. Continuation-passing, Closure-passing Style. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA, 293–302. https://doi.org/10.1145/75277.75303

[7] Andrew W. Appel and David B. MacQueen. 1987. A Standard ML Compiler. In *Functional Programming Languages and Computer Architecture (FPCA '87)* (Portland, OR, USA) *(Lecture Notes in Computer Science, Vol. 274)*. Springer-Verlag, New York, NY, USA, 301–324. https://doi.org/10.1007/3-540-18317-5_17

[8] Andrew W. Appel and Zhong Shao. 1992. Callee-save Registers in Continuation-Passing Style. *Lisp and Symbolic Computation* 5 (Sept. 1992), 191–221. https://doi.org/10.1007/BF01807505

[9] Matthias Blume. 2001. No-Longer-Foreign: Teaching an ML compiler to speak C "natively.". In *First workshop on multi-language infrastructure and interoperability (BABEL '01)* (Firenze, Italy) *(Electronic Notes in Theoretical Computer Science, Vol. 59)*. Elsevier Science Publishers, New York, NY, USA, 16. Issue 1. https://doi.org/10.1016/S1571-0661(05)80452-9

[10] Manuel M.T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. 2004. A Functional Perspective on SSA Optimisation Algorithms. *Electronic Notes in Theoretical Computer Science* 82, 2 (2004), 347 – 361. https://doi.org/10.1016/S1571-0661(05)82596-4 Proceedings of Compiler Optimization Meets Compiler Verification (COCV '03).

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

[12] Kavon Farvardin and John Reppy. 2018. Compiling with Continuations and LLVM. In Proceedings *2016 ML Family Workshop / OCaml Users and Developers workshops* (Nara, Japan) *(Electronic Proceedings in Theoretical Computer Science, Vol. 285)*, Kenichi Asai and Mark Shinwell (Eds.). Open Publishing Association, Waterloo, NSW, Australia, 131–142. https://doi.org/10.4204/EPTCS.285.5

[13] Kavon Farvardin and John Reppy. 2020. From Folklore to Fact: Comparing Implementations of Stacks and Continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)* (London, England, UK). Association for Computing Machinery, New York, NY, USA, 75–90. https://doi.org/10.1145/3385412.3385994

[14] Kathleen Fisher, Riccardo Pucella, and John Reppy. 2001. A framework for interoperability. In *Proceedings of the First International Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01) (Electronic Notes in Theoretical Computer Science, Vol. 59)*, Nick Benton and Andrew Kennedy (Eds.). Elsevier Science Publishers, New York, NY, 17. Issue 1. https://doi.org/10.1016/S1571-0661(05)80450-5

[15] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07)* (Nice, France). Association for Computing Machinery, New York, NY, USA, 37–44. https://doi.org/10.1145/1248648.1248656

[16] Fermín Javier Reig Galilea. 2002. *Compiler Architecture using a Portable Intermediate Language*. Ph.D. Dissertation. University of Glasgow, Glasgow, Scotland, UK.

[17] Emden R. Gansner and John H. Reppy (Eds.). 2004. *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, UK.

[18] Lal George. 1999. SML/NJ: Garbage Collection API. (May 1999). https://smlnj.org/compiler-notes/gc-api.ps

[19] Lal George and Andrew W. Appel. 1996. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems* 18, 3 (May 1996), 300–324. https://doi.org/10.1145/229542.229546

[20] Lal George, Florent Guillame, and John H. Reppy. 1994. A Portable and Optimizing Back End for the SML/NJ Compiler. In *Proceedings of the 5th International Conference on Compiler Construction (CC '94)*. Springer-Verlag, New York, NY, USA, 83–97. https://doi.org/10.1007/3-540-57877-3_6

[21] Richard A. Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR '95)* (San Francisco, California, USA). Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/202529.202532

[22] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. 1986. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 Symposium on Compiler Construction (SIGPLAN '86)*. Association for Computing Machinery, New York, NY, USA, 219–233. https://doi.org/10.1145/12276.13333

[23] David A. Kranz. 1988. *ORBIT: An Optimizing Compiler for Scheme*. Ph.D. Dissertation. Computer Science Department, Yale University, New Haven, Connecticut. Research Report 632.

[24] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)* (Palo Alto, California). IEEE Computer Society, Washington, D.C., USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[25] Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Master's thesis. University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA.

[26] Brian Andrew Leibig. 2013. *An LLVM Back-end for MLton*. Master's thesis. Rochester Institute of Technology, Rochester, NY, USA. https://www.cs.rit.edu/~mtf/student-resources/20124_leibig_msproject.pdf

[27] Allen Leung and Lal George. 1999. Static Single Assignment Form for Machine Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99)* (Atlanta, GA, USA). Association for Computing Machinery, New York, NY, USA, 204–214. https://doi.org/10.1145/301618.301667

[28] J. Gregory Morrisett and Andrew Tolmach. 1993. Procs and Locks: A Portable Multiprocessing Platform for Standard ML of New Jersey. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '93)* (San Diego, California, USA). Association for Computing Machinery, New York, NY, USA, 198–207. https://doi.org/10.1145/155332.155353

[29] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[30] John Reppy. 2020. *ASDL 3.0 Reference Manual*. Included in the Standard ML of New Jersey distribution.

[31] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. 2012. ErLLVM: An LLVM Backend for Erlang. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang (ERLANG '12)* (Copenhagen, Denmark). Association for Computing Machinery, New York, NY, USA, 21–32. https://doi.org/10.1145/2364489.2364494

[32] Zhong Shao and Andrew W. Appel. 1994. Space-efficient Closure Representations. *SIGPLAN Lisp Pointers* VII, 3 (July 1994), 150–161. https://doi.org/10.1145/182590.156783

[33] Zhong Shao and Andrew W. Appel. 2000. Efficient and safe-for-space closure conversion. *ACM Transactions on Programming Languages and Systems* 22, 1 (2000), 129–161.

[34] Guy L. Steele Jr. 1977. *LAMBDA: The Ultimate GOTO*. Technical Report AI Memo 443. Massachusetts Institute of Technology, Cambridge, MA, USA.

[35] David A. Terei. 2009. Low Level Virtual Machine for Glasgow Haskell Compiler. , 73 pages. https://llvm.org/pubs/2009-10-TereiThesis.pdf Undergraduate Thesis.

[36] David A. Terei and Manuel M.T. Chakravarty. 2010. An LLVM Backend for GHC. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Haskell (HASKELL '10)* (Baltimore, MD). Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/1863523.1863538

[37] Katsuhiro Ueno and Atsushi Ohori. 2014. Compiling SML# with LLVM: a Challenge of Implementing ML on a Common Compiler Infrastructure. In *Workshop on ML*. 1–2. https://sites.google.com/site/mlworkshoppe/smlsharp_llvm.pdf

[38] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. 1997. The Zephyr Abstract Syntax Description Language. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL '97)* (Santa Barbara, California). USENIX Association, Berkeley, CA, USA, 15. https://www.usenix.org/legacy/publications/library/proceedings/dsl97/wang.html

A Compiler Approach Reconciling Parallelism and Dense Representations for Irregular Trees

Anonymous Author(s)

Abstract

Recent work showed that compiling functional programs to use dense, serialized memory representations for recursive algebraic datatypes can yield significant constant-factor speedups for sequential programs. Adding parallelism in such a scenario is an open problem which we address in this work. Serializing data in a *maximally* dense format consequently serializes the processing of that data, yielding a natural tension between density and parallelism. We show that a practical compromise is possible, presenting an extension of the Gibbon compiler that exceeds the performance of existing compilers for purely functional programs that process recursive algebraic datatypes (trees).

1 Introduction

Most modern programming languages and their compilers treat tree-like data in the same way: each node and leaf is an individual heap object, and nodes connect to sub-trees via fields containing pointers. This is a simple and effective representation that is appropriate for a wide range of use cases (applicable to both functional and object-oriented programming styles, and both dynamic and static typing), and it has not changed significantly since the days of early LISP systems. The rare deviations from this consensus are found mostly within limited high-performance scenarios where complete trees can be laid out using address arithmetic with no intermediate nodes.

Of course, as HPC programmers know, one cannot treat the numbers in an array as individual heap objects, and ideally the same should be true of programs that process trees in bulk, reading or writing them in one pass. Representing tree-like data as *pointer-less, serialized byte arrays* can be extremely efficient for such traversals, as it minimizes pointer-chasing and maximizes locality. Such a representation also has the benefit of unifying the on-disk and in-memory representation of tree data, allowing programs to rapidly process large recursive tree-like data without the overhead of deserialization. Prior work has explored this approach, and in particular the Gibbon compiler [Vollmer et al. 2019, 2017] automatically transforms functional programs to operate on serialized data.

While this data representation strategy works well for sequential programs, there is an intrinsic tension if we want to parallelize these tree traversals. As the name implies, efficiently serialized data must often be read serially. To change that, first, enough *indexing* data must be left in the representation in order for parallel tasks to "skip ahead" and process multiple subtrees in parallel. Second, the allocation areas must be bifurcated to allow allocation of outputs in parallel.

In this paper, we propose a solution to these challenges. We propose a strategy where form follows function: where data representation is random-access only insofar as parallelism is needed, and both data representation and control flow "bottom out" to sequential pieces of work. That is, granularity-control in the data mirrors traditional granularity-control in parallel task scheduling. We demonstrate our solution by extending the Gibbon compiler with support for parallel computation. We also extend *LoCal*, Gibbon's typed intermediate language, and give an updated formal semantics.

Ultimately, we believe that this shows one path forward for high-performance, purely-functional traversals of trees. Parallelism in functional programming has long been regarded as theoretically promising, but has a spottier track record in practice — due to problems in runtime systems, data representation, and memory management. The parallel version of Gibbon we demonstrate in this paper directly addresses these sore spots, showing how a purely functional program operating on fine-grained irregular data can also run fast and parallelize efficiently.

In this paper, we make the following contributions:

- We introduce the first compiler that combines parallelism with automatic dense data representations for trees. While dense data [Vollmer et al. 2019] and efficient parallelism [Westrick et al. 2019] have been shown to independently yield large speedups on tree-traversing programs, our system is the first to combine these sources of speedup, yielding the fastest known performance generated by a compiler for this class of programs.
- We formalize the semantics of a *parallel location calculus* (Section 3), which underpins this novel implementation strategy. To do so we extend prior work on formalizing LoCal [Vollmer et al. 2019], which in turn builds on work in region calculi [Tofte and Talpin 1997].
- On a single core, our implementation is 2.18× and 2.79× faster than MLton and GHC respectively—two of the most mature and performant implementations of general purpose typed functional programming. When utilizing 18 cores, our geomean speedup is 1.87× and

IFL'20, September 2–4, 2020, Virtual.
.

3.16× over parallel MLton and GHC, meaning that the use of dense representations to improve sequential processing performance coexists with scalable parallelism (Section 6).

2 Overview

We give a high-level overview of the ideas presented in this paper using a simple example program given in Figure 1. It constructs a small binary tree (N L (N L L)), and uses LoCal (short for location calculus) as its syntax. While Gibbon ultimately compiles regular functional programs (a subset of Haskell), LoCal is Gibbon's intermediate language that makes explicit the manipulation of memory regions and locations. We will use LoCal to introduce concepts and terminology that will be used in the rest of the paper.

```
data Tree = Leaf | Node Tree Tree

letregion r in
letloc lʳ = start r in
letloc l_aʳ = lʳ + 1 in
let a : Tree @ l_aʳ = (Leaf l_aʳ) in
letloc l_bʳ = after(Tree @ l_aʳ) in
let b : Tree @ l_bʳ =
    letloc l_cʳ = l_bʳ + 1 in
    let c : Tree @ l_cʳ = (Leaf l_cʳ) in
    letloc l_dʳ = after(Tree @ l_cʳ) in
    let d : Tree @ l_dʳ = (Leaf l_dʳ) in
    (Node l_bʳ c d)
in (Node lʳ a b)
```

Figure 1. A LoCal program that constructs a small binary tree, (N L (N L L)).

2.1 A Primer on Location-Calculus

LoCal is a type-safe language that represents programs operating on (mostly) serialized values. All serialized values live in regions, which are growable memory buffers that store the raw data, and all programs make explicit not only the region to which a value belongs to, but also a *location* at which that value is written, where locations are fine-grained indices into a region. Unlike pointers in languages like C, arbitrary arithmetic on locations is not allowed—locations are only introduced relative to other locations.

In the program given in Figure 1, the location l^r is at the start of the region r, $l_a{}^r$ is right after the location l^r, and $l_b{}^r$ is after every element of the value rooted at $l_a{}^r$. Any expression that allocates takes an extra argument: a location-region pair that specifies where the allocation should happen. The types of such expressions are decorated with these location-region pairs. For example, a (Leaf l^r 1) data constructor allocates at a location l in region r and has type (Tree@l^r). Functions may be polymorphic over any of their input or output locations, and the concrete locations are expected to be passed in at call-sites.

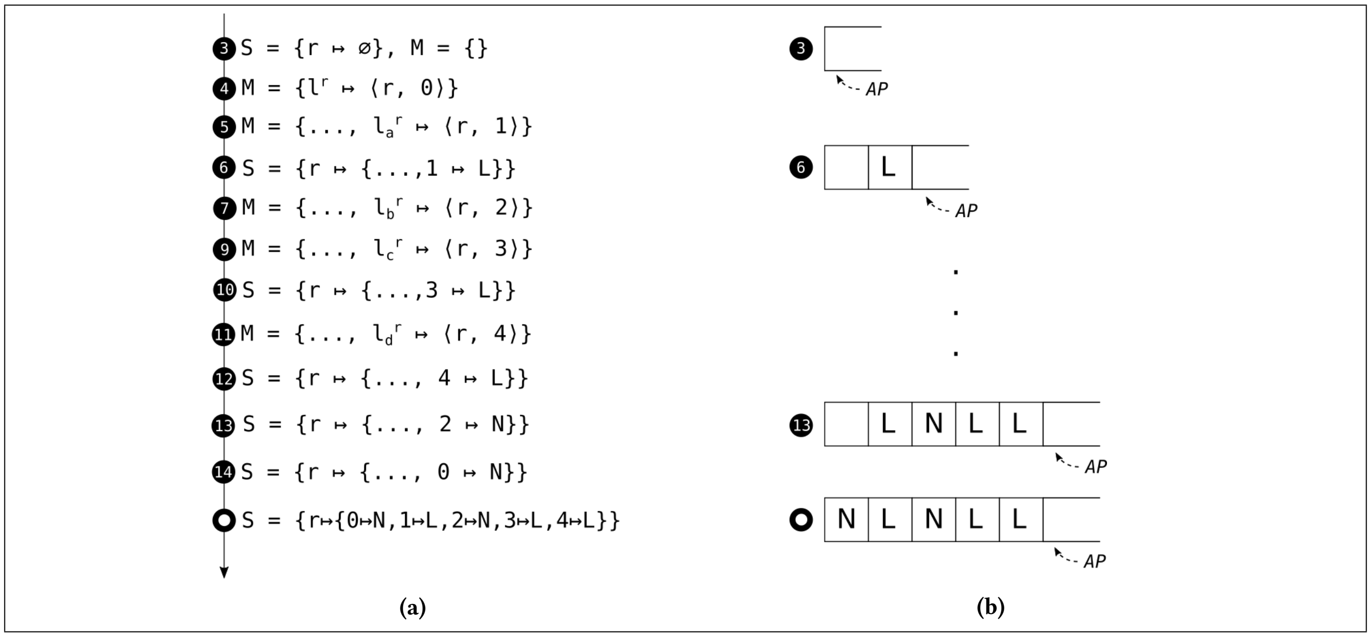
Only allowing fully-serialized values in a language means that they must be accessed in the same order in which they were serialized. While this restriction leads to efficient accesses when values are traversed in the order they are serialized, it can be inefficient in other cases because it takes away the random-access capabilities afforded by a pointer-based representation. In pointer-based C code, accessing b in (Node a b) is a constant time operation. But if all values are fully serialized, the only way to read the second value in a region is to scan over the first one; hence accessing b requires scanning over a first, which adds $O(n)$ amount of extra work! Vollmer et al. addressed this problem by allowing some offset information — such as pointers to some fields of a data constructor — to be included in the serialized representation [Vollmer et al. 2019]. Offsets can hence grant serialized datatypes random-access capabilities, but are only useful if the program consuming the data needs random access. The choice of how much or how little offset information to include is an optimization problem for the Gibbon compiler, or, at the level of LoCal, can be explicitly specified by annotating the datatype declarations.

2.2 Running LoCal Programs Sequentially

LoCal has a dynamic semantics which runs programs sequentially [Vollmer et al. 2019]. In this model, regions are represented as serialized heaps, where each heap is an array of cells that can store primitive values (data constructor tags, numbers, etc.) A write operation, such as the application of a data constructor, allocates to a fresh cell on the heap, and a read operation reads the contents of a cell. Performing multiple reads on a single cell is safe, but the type-system ensures that each cell is written to only once. At run time, *locations* in the source language translate to heap indices that specify the cells where reads/writes happen. And expressions that manipulate these locations allow a program to use different cells of the heap by performing limited arithmetic on the underlying indices. Such expressions are called "location expressions" in the language.

There are three different location expressions: (start r) returns the index of r's first cell, (l^r + 1) returns an index that points to a cell one after the cell pointed to by l^r, and (after τ@l^r) returns an index that is one after every cell occupied by the value rooted at l^r. An *end-witness* judgement is used to evaluate an after expression. A naive computational interpretation of this judgement is to simply scan over a value to compute its end, but in practice this linear scan can be avoided by tracking end-witnesses, for example, having every write return the index of the cell after it.

Intuitively, we can imagine there being a single allocation pointer that is used to perform all writes in the program. It always points to the next available cell on heap, and each write advances it by one. When the program starts executing,



```
3  S = {r ↦ ∅}, M = {}
4  M = {$l^r$ ↦ ⟨r, 0⟩}
5  M = {..., $l_a^r$ ↦ ⟨r, 1⟩}
6  S = {r ↦ {...,1 ↦ L}}
7  M = {..., $l_b^r$ ↦ ⟨r, 2⟩}
9  M = {..., $l_c^r$ ↦ ⟨r, 3⟩}
10 S = {r ↦ {...,3 ↦ L}}
11 M = {..., $l_d^r$ ↦ ⟨r, 4⟩}
12 S = {r ↦ {..., 4 ↦ L}}
13 S = {r ↦ {..., 2 ↦ N}}
14 S = {r ↦ {..., 0 ↦ N}}
○  S = {r↦{0↦N,1↦L,2↦N,3↦L,4↦L}}
```

Figure 2. (a) Sequential, step-by-step execution of the program from Figure 1, and (b) the corresponding heap operations. Each step is named after its line number in the program and only shows the changes relative to the previous step. *AP* is the allocation pointer.

the allocation pointer starts at the beginning of the heap and it chugs along in a continuous fashion performing writes along the way, as illustrated in Figure 2b. Consuming a serialized value can be thought of in a similar way—that there is a single read pointer that starts at the beginning of the heap and moves along its length performing reads. (Note that offsets eliminate the need for computing end-witnesses when performing reads, but not writes.)

Figure 2a gives a step-by-step trace of the sequential semantics executing the program from Figure 1. The store S maps regions to their corresponding heaps, and the location map M maps symbolic locations to their corresponding heap indices. In the first two steps, a fresh region r is created and location l^r is initialized to point to r's 0th cell. Then the location of the first sub-tree, ${l_a}^r$, is defined to be one after l^r. Step 6 constructs the first sub-tree by writing a tag L (short for leaf) on the heap. Then the location of the second sub-tree, ${l_b}^r$, is defined to be after every element of the first sub-tree. Since there is only a single leaf before it, ${l_b}^r$ gets initialized to point to the 2nd cell by the end-witness judgement. Note that the allocation pointer AP is already at the correct cell. Following similar steps, the second sub-tree is constructed at ${l_b}^r$. Finally, Step 12 writes the tag N (short for Node) which completes the construction of the full tree, `(N L (N L L))`.

2.3 Parallelism in LoCal

In this section, we outline the various opportunities for parallelism that exist in LoCal programs. The first kind of parallelism is available when LoCal programs access the store in a read-only fashion, such as the program that calculates the size of a binary tree.

```
size : ∀ $l^r$ . Tree @ $l^r$ → Int
size [$l^r$] t = case t of
                   Leaf → 1
                   Node (a : Tree @ ${l_a}^r$) (b : Tree @ ${l_b}^r$)
                     → (size [${l_a}^r$] a) + (size [${l_b}^r$] b)
```

However, even though the recursive calls in the `Node` case can safely evaluate in parallel, there is a subtelty: parallel evaluation is efficient only if the `Node` constructor stores offset information for its child nodes. If it does, then the address of `b` can be calculated in constant time, thereby allowing the calls to proceed immediately in parallel. If there is no offset information, then the overall tree traversal is necessarily sequential, because the starting address of `b` can be obtained only after a full traversal of `a`. As such, there is a tradeoff between space and time, that is, the cost of the space to store the offset in the `Node` objects versus the time of the sequential traversal (e.g., of `a`) forced by the absence of offsets.

Programs that write to the store also provide opportunities for parallelism. The most immediate such opportunity exists when the program performs writes that affect different regions. For example, the writes to construct the leaf nodes for `a` and `b` can happen in parallel because different regions cannot overlap in memory.

```
letregion ra in
letregion rb in
letloc $la^{ra}$ = start ra in
letloc $lb^{rb}$ = start rb in
```

```
let a : Tree @ la^ra = Leaf la^ra in
let b : Tree @ lb^ra = Leaf lb^rb in
...
```

There is another kind of parallelism that is more challenging to exploit, but is at least as important as the others: the parallelism that can be realized by allowing different fields of the same constructor to be filled in parallel. This is crucial in LoCal programs, where large, serialized data frequently occupy only a small number of regions, and yet there are opportunities to exploit parallelism in their construction. Consider the `buildtree` program, which creates a binary tree of a given size `n` in a given region `r`.

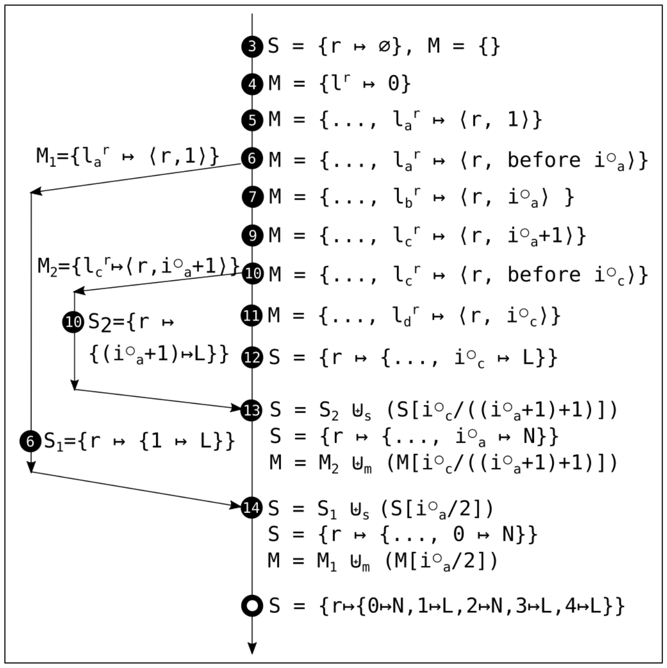
```
buildtree : ∀ l^r . Int → Tree @ l^r
buildtree [l^r] n =
  if n == 0 then (Leaf l^r 1)
  else letloc l_a^r = l^r + 1 in
       let left : Tree @ l_a^r =
           buildtree [l_a^r] (n - 1) in
       letloc l_b^r = after(Tree @ l_a^r) in
       let right : Tree @ l_b^r =
           buildtree [l_b^r] (n - 1) in
       (Node l^r left right)
```

If we want to access the parallelism between the recursive calls, we need to break the data dependency that the right branch has on the left. The starting address of the right branch, namely $l_b{}^r$, is assigned to be end witness of the left branch by the `letloc` instruction. But the end witness of the left branch is, in general, known only after the left branch is completely filled, which would effectively sequentialize the computation. One non-starter would be to ask the programmer to specify the size of the left branch up front, which would make it possible to calculate the starting address of the right branch. Unfortunately, this approach would introduce safety issues, such as incorrect size information, of exactly the kind that LoCal is designed to prevent. Instead, we explore an approach that is safe-by-construction and efficient, as we explain next.

2.4 Fully-Parallel Semantics

To address the challenges of parallel evaluation, we start by presenting a high-level execution model that can utilize all potential parallelism in LoCal programs. This execution model functions as a reference for the space of possible implementation strategies. In particular, the model formalizes all possible valid parallel schedules and all valid heap layouts. In this model, the surface language of LoCal is unchanged from the original, sequential language. That is, there are no new linguistic constructs needed to, e.g., spawn parallel tasks or synchronize on task completion. Parallelism in our fully-parallel model is generated implicitly, by allowing every let-bound expression to evaluate in parallel with the body.

To demonstrate the model, let us consider a trace of the fully-parallel evaluation of the program from Figure 1. We



3 $S = \{r \mapsto \varnothing\}$, $M = \{\}$
4 $M = \{l^r \mapsto 0\}$
5 $M = \{\ldots, l_a{}^r \mapsto \langle r, 1\rangle\}$
$M_1=\{l_a{}^r \mapsto \langle r,1\rangle\}$
6 $M = \{\ldots, l_a{}^r \mapsto \langle r, \text{before } i\circ_a\rangle\}$
7 $M = \{\ldots, l_b{}^r \mapsto \langle r, i\circ_a\rangle\}$
9 $M = \{\ldots, l_c{}^r \mapsto \langle r, i\circ_a+1\rangle\}$
$M_2=\{l_c{}^r\mapsto\langle r,i\circ_a+1\rangle\}$
10 $M = \{\ldots, l_c{}^r \mapsto \langle r, \text{before } i\circ_c\rangle\}$
10 $S_2=\{r \mapsto \{(i\circ_a+1)\mapsto L\}\}$
11 $M = \{\ldots, l_d{}^r \mapsto \langle r, i\circ_c\rangle\}$
12 $S = \{r \mapsto \{\ldots, i\circ_c \mapsto L\}\}$
13 $S = S_2 \uplus_s (S[i\circ_c/((i\circ_a+1)+1)])$
$S = \{r \mapsto \{\ldots, i\circ_a \mapsto N\}\}$
$M = M_2 \uplus_m (M[i\circ_c/((i\circ_a+1)+1)])$
6 $S_1=\{r \mapsto \{1 \mapsto L\}\}$
14 $S = S_1 \uplus_s (S[i\circ_a/2])$
$S = \{r \mapsto \{\ldots, 0 \mapsto N\}\}$
$M = M_1 \uplus_m (M[i\circ_a/2])$
$S = \{r\mapsto\{0\mapsto N,1\mapsto L,2\mapsto N,3\mapsto L,4\mapsto L\}\}$

Figure 3. Fully parallel, step-by-step execution of the program from Figure 1. Each step is named after its line number in the program and only shows the changes relative to the previous step.

are going to first examine the trace corresponding to the schedule shown in Figure 3, where the let expressions that bind `a` and for `c` are both parallelized. The parallel fork point for the first let expression (the one corresponding to `a`) occurs on the fourth step of the trace. At this point, the evaluation of the let-bound expression results in the creation of a new child task, and the continuation of the body of the let expression in the parent task. Each task has its own private view of memory, which is realized by giving the child and parent task copies of the store S and location map M. These copies differ in one way, however: each sees a different mapping for the starting location of `a`, namely $l_a{}^r$. The child task sees the mapping $l_a{}^r \mapsto \langle l_a, 1\rangle$, which is the ultimate starting address of `a` in the heap.

The parent task sees a different mapping for $l_a{}^r$, namely $\langle l_a, \texttt{before } i\circ_a\rangle$. This location is a *before index*: it behaves like an I-Var [Arvind et al. 1989], and, in our example, stands in for the completion of the memory being filled for `a`, by the child task. Any expression in the body of the let expression that tries to read from this location blocks on the completion of the child task. The reason this placeholder value is prefixed by "before" is that the variable $i\circ_a$ attached to it refers to the end witness of the object starting at `a`. The end witness of `a` is needed by the `letloc` expression at line 7, just after the parent continues after the fork point. At this point, the parent task uses the `letloc` expression to assign an appropriate location for the starting address of `b`, which is $l_b{}^r \mapsto \langle l_a, i\circ_a\rangle$. This *placeholder variable*, $i\circ_a$, is used by the parent task as a

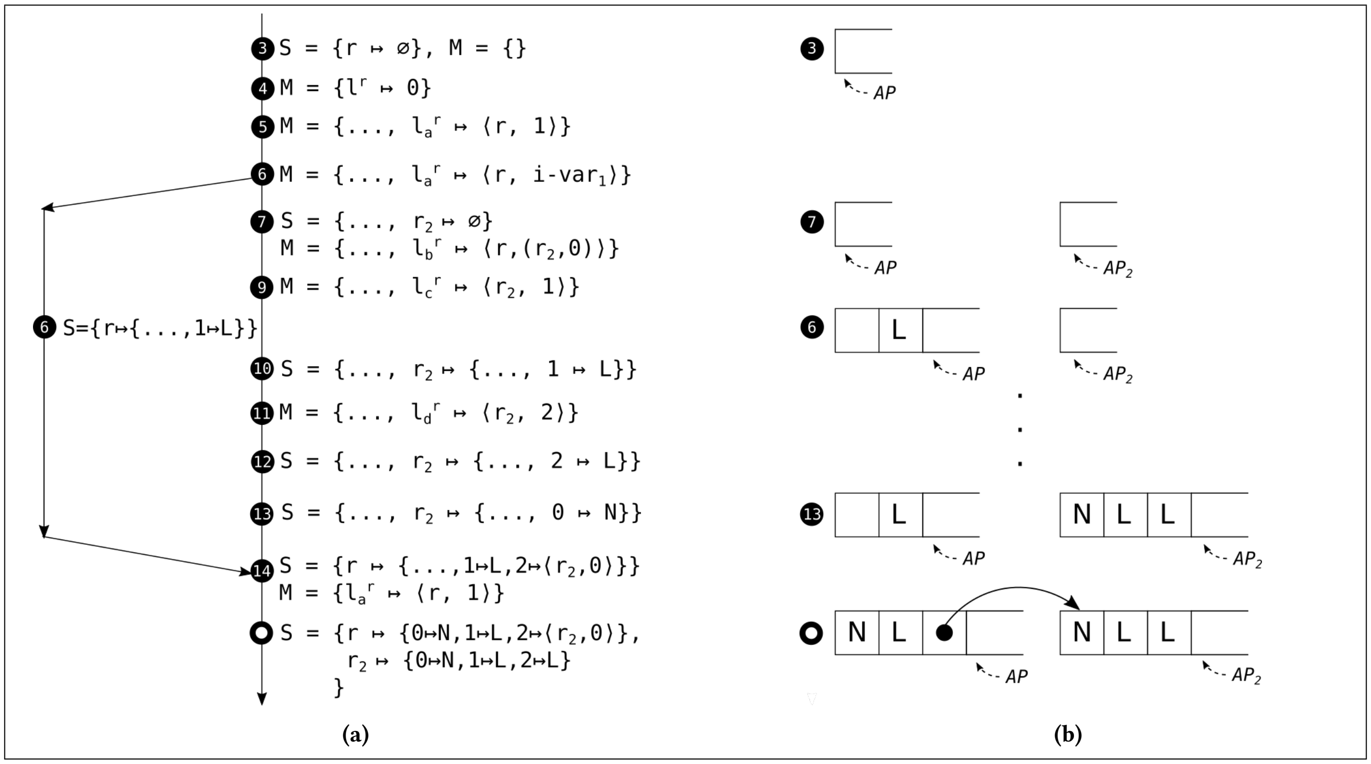

3 $S = \{r \mapsto \emptyset\}$, $M = \{\}$
4 $M = \{l^r \mapsto 0\}$
5 $M = \{\ldots, l_a^r \mapsto \langle r, 1\rangle\}$
6 $M = \{\ldots, l_a^r \mapsto \langle r, \texttt{i-var}_1\rangle\}$
7 $S = \{\ldots, r_2 \mapsto \emptyset\}$
$M = \{\ldots, l_b^r \mapsto \langle r, (r_2, 0)\rangle\}$
9 $M = \{\ldots, l_c^r \mapsto \langle r_2, 1\rangle\}$
6 $S=\{r \mapsto \{\ldots, 1 \mapsto L\}\}$
10 $S = \{\ldots, r_2 \mapsto \{\ldots, 1 \mapsto L\}\}$
11 $M = \{\ldots, l_d^r \mapsto \langle r_2, 2\rangle\}$
12 $S = \{\ldots, r_2 \mapsto \{\ldots, 2 \mapsto L\}\}$
13 $S = \{\ldots, r_2 \mapsto \{\ldots, 0 \mapsto N\}\}$
14 $S = \{r \mapsto \{\ldots, 1 \mapsto L, 2 \mapsto \langle r_2, 0\rangle\}\}$
$M = \{l_a^r \mapsto \langle r, 1\rangle\}$
$S = \{r \mapsto \{0 \mapsto N, 1 \mapsto L, 2 \mapsto \langle r_2, 0\rangle\},$
$r_2 \mapsto \{0 \mapsto N, 1 \mapsto L, 2 \mapsto L\}$
$\}$
(a)
3
AP
7
AP
AP_2
6
L
AP
AP_2
13
L
AP
N L L
AP_2
N L
AP
N L L
AP_2
(b)

Figure 4. Parallel, step-by-step execution of the program from Figure 1 such that parallel allocations happen only in separate regions (a), and the corresponding heap operations (b). Each step is named after its line number in the program and only shows the changes relative to the previous step. AP and AP_2 are the allocation pointers.

temporary allocation pointer, from which it can continue to allocate new objects. The next object allocated by the parent task is c, for which the starting address is $\langle l_a, i\circ_a +1\rangle$, the address one cell past the end of b in the parent task.

The use of a on line 14 forces the parent task to join with its child task. This particular join point eliminates both the before index and the placeholder variable in the parent task, thereby removing all occurrences of $i\circ_a$, and allowing the parent task to continue evaluating. In particular, the before index $\langle l_a, \texttt{before}\ i\circ_a\rangle$ is substituted for $\langle l_a, 1\rangle$, the starting address of a, and the addresses starting with the placeholder variable $i\circ_a$ in the store and location map of the parent task are substituted for $\langle l_a, 2\rangle$, the end witness of a. Finally, all the new entries in the location map M and store S of the child are merged into the corresponding environments in the parent task. Join points in LoCal are, in general, deterministic, because they only increase the information held by the parent task. Moreover, the layout of the heap after the join point is equivalent to the one that would be constructed in the sequential execution: all heap layouts, and the corresponding heap addresses in the environments, end up being the same for all schedules. This property is the main abstraction that is provided by the fully parallel semantics, but it does not lend itself well to efficient implementation. The problem is the complication of the addressing of objects in regions.

2.5 Region-Parallel Semantics

We now present a lower-level semantics that treats parallel allocations in the same region in a way that can be implemented efficiently, with simple, linear addressing for regions, while retaining the ability to take all possible parallel schedules. In this region-parallel semantics, unlike the fully parallel semantics, there can be at most one task allocating in a given region at a time. To realize single-region allocations, the semantics introduces fresh, intermediate regions as needed, that is, when the schedule takes a parallel evaluation step for a given let-bound expression, and the body expression tries to allocate in the same region.

Let us consider how our region-parallel semantics differs from our fully parallel version by following the trace in Figure 4 of our example program. After the first five steps, we reach the outer **let** binding, where the schedule forks a child task, as in our previous trace. The let-bound expression proceeds at this point to evaluate in a parallel task with the original region r. Like before, the parent and child tasks see a different mapping for $l_a{}^r$, $\texttt{i-var}_1$ and $\langle r, 1\rangle$ respectively. At step seven, the body of the let expression continues in the parent task, and uses a **letloc** expression to compute the end-witness of $l_a{}^r$. In such a situation the fully parallel semantics uses a placeholder index as the end witness. Here, instead of a placeholder index, a fresh region r_2 is created, and the starting address of b now becomes an indirection, $l_b{}^r \mapsto \langle r, (r_2, 0)\rangle$, and the parent task uses r_2 for allocations instead of r. The parent and child tasks have, in effect, two different allocation pointers for the same logical region. When the tasks reach the join point, the merging of their respective memories is handled by merging the stores with a simple set union operation, and then linking together the regions r and r_2 by pointers. To link the regions, the program writes an indirection pointer at the end of the region allocated by the child task, which points to beginning of the fresh region r_2 of the parent task. This linking is cheap, and in our implementation it replaces the merging of the store S and location map M performed at the join point in the fully parallel semantics.

3 Formal Semantics for Fully-Parallel LoCal

In this section, we present the formal semantics of our fully parallel LoCal. This semantics has also been mechanically tested in PLT Redex [Felleisen et al. 2009]. The grammar for the language is given in Figure 5. All parallelism in this model language is introduced implicitly, by evaluating `let` expressions. There is no explicit syntax for introducing parallelism in our fully parallel language, and consequently the language is, from the perspective of a client, exactly the same as the sequential language [Vollmer et al. 2019].

The parallel LoCal semantics does, however, differ from the sequential semantics, most notably from the introduction of a richer form of indexing in regions. Whereas in sequential LoCal a region index consists of a non-negative integer and a concrete location of a pair of a region identifier and an index, the region index and concrete location are more complex in parallel LoCal. The enriched forms support parallel construction of the fields of the same data constructor application by functioning as placeholders for heap indices that are not yet known. The region index i now generalizes to a *region-index expression*, which consists of either a concrete index $i\bullet$, a placeholder index $i\circ$, or an index, plus an offset $i + i\bullet$. A concrete index is a non-negative integer that specifies the final index of a position in a region. A placeholder index is a synchronization variable that is used to coordinate between parallel tasks. For example, the placeholder index $i\circ_a$ in the sample trace in Figure 3, is used by the child task to communicate to its parent task the end witness of the object starting at a, which is the final result generated by the child task. All indices allocated by a parent task allocate heap values at indices on an offset from the placeholder index. For example, the tree node c in the sample trace is allocated at the index $i\circ_a + 1$, that is, one cell past the end of a. A concrete location cl is enriched from its simpler definition in the sequential semantics to be a pair $\langle r, i\diamond\rangle$ of a region r and an extended region index $i\diamond$. The extended region index $i\diamond$ is either a region-index expression or a before index. A before index

$$
\begin{array}{l}
K \in \text{ Data Constructors}, \quad \tau_c \in \text{ Type Constructors}, \\
x, y, f \in \text{ Variables}, \quad l, l^r \in \text{ Symbolic Locations}, \\
r \in \text{ Regions}, \quad i\bullet, j\bullet \in \text{ Concrete Region Indices}, \\
i\circ, j\circ \in \text{ Placeholder Region Indices}
\end{array}
$$

$$
\begin{array}{rcl}
\text{Top-Level Programs} & top & ::= \overrightarrow{dd}\ ;\ \overrightarrow{fd}\ ;\ e \\
\text{Datatype Declarations} & dd & ::= \texttt{data}\ \tau_c = \overrightarrow{K\ \ \hat{\tau}} \\
\text{Function Declarations} & fd & ::= f : ts; f\ \overrightarrow{x} = e \\
\text{Located Types} & \hat{\tau} & ::= \tau @ l^r \\
\text{Types} & \tau & ::= \tau_c \\
\text{Type Scheme} & ts & ::= \forall_{\overrightarrow{l^r}} . \overrightarrow{\hat{\tau}} \to \hat{\tau} \\
\text{Region Indices} & i, j & ::= i\bullet \mid i\circ \mid i + i\bullet \\
\text{Extended Region Indices} & i\diamond, j\diamond & ::= i \mid \texttt{before}\ i\circ \\
\text{Concrete Locations} & cl & ::= \langle r, i\diamond\rangle^{l} \\
\text{Values} & v & ::= x \mid cl \\
\text{Expressions} & e & ::= v \\
 & & \mid f\ [\overrightarrow{l^r}]\ \overrightarrow{v} \\
 & & \mid K\ l^r\ \overrightarrow{v} \\
 & & \mid \texttt{let}\ x : \hat{\tau} = e\ \texttt{in}\ e \\
 & & \mid \texttt{letloc}\ l^r = le\ \texttt{in}\ e \\
 & & \mid \texttt{letregion}\ r\ \texttt{in}\ e \\
 & & \mid \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat} \\
\text{Pattern} & pat & ::= K\ \ \overrightarrow{(x : \hat{\tau})}\ \to\ e \\
\text{Location Expressions} & le & ::= \texttt{start}\ r \\
 & & \mid l^r + 1 \\
 & & \mid \texttt{after}\ \hat{\tau} \\
\text{Store} & S & ::= \{\, r_1 \mapsto h_1, \ \ldots\ , r_n \mapsto h_n \,\} \\
\text{Heap Values} & hv & ::= K \\
\text{Heap} & h & ::= \{\, i_1 \mapsto hv_1, \ \ldots\ , i_n \mapsto hv_n \,\} \\
\text{Location Map} & M & ::= \{\, l_1^{r_1} \mapsto cl_1, \ \ldots\ , l_n^{r_n} \mapsto cl_n \,\} \\
\text{Sequential States} & t & ::= S; M; e \\
\text{Parallel Tasks} & T & ::= (\hat{\tau}, cl, t)
\end{array}
$$

Figure 5. Grammar of LoCal$^{\text{par}}$.

`before` $i\circ$ denotes a field in some constructor application, such that the index $i\circ$ denotes the end witness of the field.

The state configurations of LoCal$^{\text{par}}$ appear at the bottom of Figure 5. Just like in the sequential LoCal, a sequential state of LoCal$^{\text{par}}$, t, contains a store, location map, and an expression. We generalize a sequential state to a parallel task T by adding two more fields: a located type and a concrete location, which together describe the type and location of the final result written by the task. A parallel transition in LoCal$^{\text{par}}$ takes the form of the following rule, where a number of tasks step together.

$$T_1, \ldots, T_n \Longrightarrow T_1', \ldots, T_n', \ldots, T_m$$

In each step, a given task may make a sequential transition, it may fork a new parallel task, it may join with another parallel task, or it may remain unchanged.

A subset of the sequential transition rules are given in Figure 6. The rules are close to the sequential rules, except for minor differences in three rules. For the rule D-LetLoc-Tag, we need to handle the case where a before index is assigned to the source symbolic location l'^r. For this purpose, we use the metafunction that either increments its parameter, if it is a region index, or advances to the end witness, if it is a before index.

$$\begin{array}{lcl} Incr(i) & = & i + 1 \\ Incr(\texttt{before } i\circ) & = & i\circ \end{array}$$

With respect to the rule D-Par-LetLoc-After, we now allow the concrete location assigned to the source location $l_1{}^r$ to hold a before index. The purpose of this relaxation is to allow an expression downstream from a parallelized `let` binding to continue evaluating in parallel with the task that is producing the value of the let-bound variable. The task evaluating the `letloc` expression continues by using a temporary allocation pointer based on the before index. That is, if the `letloc` encounters a before index in its source location, `before` $i\circ$, then the index j that results from our end witness judgment yields $i\circ$. The effect is to make $i\circ$ the setting for the allocation pointer for the task.

For the rule D-Case, there is a new metafunction $\hat{S}(r, i)$ that is needed to handle the complicated indexing in LoCal$^{\text{par}}$.

$$\hat{S}(r, i) \quad = \quad hv \text{ where } (i' \mapsto hv) \in S(r) \text{ and } Nf(i) = Nf(i')$$

$$\begin{array}{lcllcl} Nf(i\circ) & = & i\circ + 0 & & & \\ Nf(i\bullet) & = & i\bullet & & & \\ Nf(i + i\bullet) & = & i\bullet' + i\bullet & \text{where} & i\bullet' & = Nf(i) \\ Nf(i + i\bullet) & = & i\circ + (i\bullet' + i\bullet) & \text{where} & i\circ + i\bullet' & = Nf(i) \end{array}$$

The reason this metafunction is needed relates to the complicated indexing structure of LoCal$^{\text{par}}$. In order to resolve an index in the store, the store-lookup metafunction needs to resolve each index to a normal form, where a region index evaluates to either an integer value $i\bullet$ or to a placeholder index, plus an integer offset $i\circ + i$.

The parallel transition rules are given in Figure 7. In these rules, we model parallelism by an interleaving semantics. Any of the tasks that are ready to take a sequential step may make a transition in rule D-Par-Step. A parallel task can be spawned by the D-Par-Let rule, from which an in-flight `let` expression breaks into two tasks. The child task handles the evaluation of the `let`-bound expression e_1 and the parent the body e_2. To represent the future location of the `let`-bound expression, the rule creates a fresh placeholder index $i\circ_1$, and from it, builds a before index `before` $i\circ_1$, which is passed to the body of the `let` expression. A task can satisfy a data dependency in a rule, such as D-Par-Case-Join, where a `case` expression blocked on the value located at `before` $i\circ_c$ joins with the task producing the value. Although there are several other rules in addition to D-Par-Case-Join that handle joins, we omit them, because they are similar. Because each task has a private copy of the store and location map, the process of joining two tasks involves merging environments. Before

[D-LetLoc-Start]

$$S; M; \texttt{letloc } l^r = \texttt{start } r \texttt{ in } e \Rightarrow S; M'; e$$
$$\text{where } M' = M \cup \{ l^r \mapsto \langle r, 0 \rangle \}$$

[D-LetLoc-Tag]

$$S; M; \texttt{letloc } l^r = l'^r + 1 \texttt{ in } e \Rightarrow S; M'; e$$
$$\text{where } M' = M \cup \{ l^r \mapsto \langle r, Incr(i\diamond) \rangle \};\ \langle r, i\diamond \rangle = M(l'^r)$$

[D-LetLoc-After]

$$S; M; \texttt{letloc } l^r = \texttt{after } \tau @ l_1{}^r \texttt{ in } e \Rightarrow S; M'; e$$
$$\begin{array}{ll} \text{where } & M' = M \cup \{ l^r \mapsto \langle r, j \rangle \};\ \langle r, i\diamond \rangle = M(l_1{}^r) \\ & \tau; \langle r, i\diamond \rangle; S \vdash_{ew} \langle r, j \rangle \end{array}$$

[D-LetRegion]

$$S; M; \texttt{letregion } r \texttt{ in } e \Rightarrow S; M; e$$

[D-DataConstructor]

$$S; M; K\ l^r\ \overrightarrow{v} \Rightarrow S'; M; \langle r, i \rangle^{l^r}$$
$$\text{where } S' = S \cup \{ r \mapsto (i \mapsto K) \};\ \langle r, i \rangle = M(l^r)$$

[D-Case]

$$S; M; \texttt{case } \langle r, i \rangle^{l^r} \texttt{ of } [\ldots, K\ \ (\overrightarrow{x : \tau @ l^r})\ \rightarrow\ e, \ldots] \Rightarrow$$
$$S; M'; e[\langle r, \overrightarrow{w} \rangle^{\overrightarrow{l^r}} / \overrightarrow{x}]$$
$$\begin{array}{ll} \text{where } & M' = M \cup \{ \overrightarrow{l_1^r} \mapsto \langle r, i+1 \rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}} \rangle \} \\ & \overrightarrow{\tau_1}; \langle r, i+1 \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle \\ & \overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle \\ & K = \hat{S}(r, i);\ j \in \{ 1, \ldots, n-1 \};\ n = |\overrightarrow{x : \hat{\tau}}| \end{array}$$

[D-Let-Expr]

$$\frac{S; M; e_1 \Rightarrow S'; M'; e_1' \qquad e_1' \neq v}{S; M; \texttt{let } x : \hat{\tau} = e_1 \texttt{ in } e_2 \Rightarrow S'; M'; \texttt{let } x : \hat{\tau} = e_1' \texttt{ in } e_2}$$

[D-Let-Val]

$$S; M; \texttt{let } x : \hat{\tau} = v_1 \texttt{ in } e_2 \Rightarrow S; M; e_2[v_1 / x]$$

[D-App]

$$S; M; f\ [\overrightarrow{l^r}]\ \overrightarrow{v} \Rightarrow S; M; e[\overrightarrow{v} / \overrightarrow{x}][\overrightarrow{l^r} / \overrightarrow{l'^{r'}}]$$
$$\begin{array}{ll} \text{where } & fd = Function(f) \\ & f : \forall_{\overrightarrow{l'^{r'}}} . \overrightarrow{\hat{\tau}_f} \rightarrow \hat{\tau}_f; (f\ \overrightarrow{x} = e) = Freshen(fd) \end{array}$$

Figure 6. Dynamic semantics (sequential transitions).

[D-Par-Step]

$$\frac{S; M; e \Rightarrow S'; M'; e'}{\begin{array}{l} T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow \\ T_1, \ldots, (\hat{\tau}, cl, S'; M'; e'), \ldots T_n \end{array}}$$

[D-Par-Let]

$$\begin{array}{l} T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow \\ T_1, \ldots, (\hat{\tau}_1, cl'_1, S; M; e_1), \ldots T_n, (\hat{\tau}, cl, S; M_2; e'_2) \end{array}$$

$$\begin{array}{ll} \text{where} & e = (\texttt{let}\ x : \hat{\tau}_1 = e_1\ \texttt{in}\ e_2);\ \hat{\tau}_1 = \tau_1 @ {l_1}^{r_1} \\ & i\circ_1\ \text{fresh};\ cl'_1 = \langle r_1, \texttt{before}\ i\circ_1 \rangle \\ & M = \{\, {l_1}^{r_1} \mapsto cl_1 \,\} \cup M' \\ & M_2 = \{\, {l_1}^{r_1} \mapsto cl'_1 \,\} \cup M' \\ & e'_2 = e_2[cl'_1/x] \end{array}$$

[D-Par-Case-Join]

$$\begin{array}{l} T_1, \ldots, (\hat{\tau}_c, cl_c, S_c; M_c; e_c), \ldots, T_n \Longrightarrow \\ T_1, \ldots, (\hat{\tau}_c, cl_c, S'_c; M'_c; e'_c), \ldots T_n \end{array}$$

$$\begin{array}{ll} \text{where} & e_c = \texttt{case}\ \langle r, \texttt{before}\ i\circ_c \rangle^{l_c}\ \texttt{of}\ \overrightarrow{pat} \\ & T_p \in \{\, T_1, \ldots, T_n \,\} \\ & T_p = (\tau_p @ {l_p}^{r}, \langle r, \texttt{before}\ i\circ_c \rangle, S_p; M_p; \langle r, i_p \rangle) \\ & \tau_p; \langle r, i_p \rangle; S_p \vdash_{ew} \langle r, i_e \rangle \\ & S'_c = \mathit{MergeS}(S_p, S_c[i\circ_c / i_e]) \\ & M'_c = \mathit{MergeM}(M_p, M_c[i\circ_c / i_e]) \\ & e'_c = \texttt{case}\ \langle r, i_p \rangle^{l_p}\ \texttt{of}\ \overrightarrow{pat}[i_p/\texttt{before}\ i\circ_c] \end{array}$$

Figure 7. Dynamic semantics (parallel transitions).

merging the environments, all occurrences of the placeholder index $i\circ_c$ and the before index `before` $i\circ_c$ are eliminated in the location map and the continuation. These occurrences are replaced by the index i_p and the end witness i_e, that represent the starting index and the end witness produced by the task T_p respectively.

The merging of the task memories is performed by the metafunctions given in the Appendix A.1.

4 Formal Semantics for Region-Parallel LoCal

In this section, we present the formalism for the lower-level calculus, LoCal$^{\text{regpar}}$. Figure 8 shows the changes made to the grammar for the language. We make a return to the simpler, integer-based scheme for indexing the heap used in the sequential LoCal. Whereas in sequential LoCal and LoCal$^{\text{par}}$, only data constructor tags were allowed to be written to the heap, in LoCal$^{\text{regpar}}$ heap values are extended to support indirections. An indirection (q, j) that is written in the heap at $\langle r, i \rangle$, is a pointer from $\langle r, i \rangle$ to $\langle q, j \rangle$. Similar to LoCal$^{\text{par}}$, a concrete location is enriched to be a pair $\langle r, i\diamond \rangle$ of a region r and an extended region index $i\diamond$. But instead of having before indices, an extended region index is either a concrete region-index or an `i-var` i, which is used to synchronize between parent and child tasks. Like heap values, the concrete locations used in the location map are further enriched to support indirections.

Like in LoCal$^{\text{par}}$, our LoCal$^{\text{regpar}}$ machine transition steps a collection of parallel tasks using an interleaving semantics.

$$T_1, \ldots, T_n \Longrightarrow_{rp} T'_1, \ldots, T'_n, \ldots, T_m$$

The sequential transition steps are similar, except that since LoCal$^{\text{regpar}}$'s location map can also contain indirections, a map lookup function that can de-reference indirections, $\hat{M}$, has to be used.

$$\begin{array}{llll} \hat{M}(l) & = \langle r, i\diamond \rangle & \text{where} & (l \mapsto cl\diamond) \in M(l)\ \text{and} \\ & & & \langle r, i\diamond \rangle = \mathit{DerefM}(M, cl\diamond) \end{array}$$

$$\begin{array}{lcl} \mathit{DerefM}(M, \langle r, i\diamond \rangle) & = & \langle r, i\diamond \rangle \\ \mathit{DerefM}(M, \langle r, (q, i) \rangle) & = & \langle q, i \rangle \end{array}$$

Other meta functions operating on LoCal$^{\text{par}}$'s enriched region indices, namely Nf, Incr, and $\hat{S}$ are no longer required since LoCal$^{\text{regpar}}$ uses simple integer based region indices. Some parallel transitions are given in Figure 9. Others, and synchronization between parallel and child tasks is also similar to LoCal$^{\text{par}}$, but they use an I-Var instead of a placeholder index to manage the joining of parallel tasks. In the rest of the section we focus on the primary challenge in LoCal$^{\text{regpar}}$ which relates to computing end witnesses of I-Var's, and merging of memories at join points.

In order to efficiently compute the end-witness of an I-Var, we give a different treatment to the parallel transition for a letloc-after expression. If the `letloc`'s source location is not an I-Var, D-RegionPar-LetLoc-After computes the end witness just like sequential LoCal. If it is an I-Var, the D-RegionPar-LetLoc-After-New-Reg transition creates a fresh region r', and maps l^r to r''s 0th cell by adding an indirection to the location map, $l^r \mapsto \langle r, (r', 0) \rangle$. Now, the entity allocating at l^r will use the fresh region r'. Effectively there are two different allocation pointers for the same logical region, thus respecting the single-threaded-regions invariant. Since certain allocations use fresh regions, some fields of a data constructor may be written to different regions (depending on the schedule of parallel execution), and they have to be reconciled to simulate a single region.

In LoCal$^{\text{regpar}}$, merging of region memories occurs when a task T_b blocks at an I-Var, just like LoCal$^{\text{par}}$. The meta functions used to merge the task memories are similar (Appendix A.1), but are slightly modified since we don't need to compute normal forms of region indices, and the grammar uses `i-var`'s instead of `before` indices. However we still need to bring together into a single region the fields of data constructors which were written to different regions. The D-RegionPar-DataConstructor-Link transition accomplishes this. When a task is evaluating a constructor application and has already merged the memories of all its fields, it stitches together fields of the constructor with the help of indirections. This stitching together is achieved by attaching an indirection pointer to the end of a field, if it's neighboring

$$\begin{array}{rrl}
\text{Region Indices} & i, j & ::= i\bullet \\
\text{Extended Region Indices} & i\diamond, j\diamond & ::= i \mid \texttt{i-var}\ i \\
\text{Concrete Locations} & cl & ::= \langle r, i\diamond \rangle^{l} \\
\text{Types} & \tau & ::= \tau_c \mid \texttt{ind}\ \tau_c \\
\text{Indirections} & hr & ::= (r, i) \\
\text{Heap Values} & hv & ::= K \mid hr \\
\text{Extended Concrete Locations} & cl\diamond & ::= cl \mid \langle r, hr \rangle^{l} \\
\text{Location Map} & M & ::= \{\, l_1^{r_1} \mapsto cl\diamond_1,\ \ldots, \\
 & & \quad\ l_n^{r_n} \mapsto cl\diamond_n \,\}
\end{array}$$

Figure 8. Updated grammar for Region-Parallel Semantics. The syntactic forms not shown here remain the same as Figure 5.

field resides in a separate region. We use a meta function *LinkFields* for this purpose.

$$\begin{array}{lcl}
\mathit{LinkFields}(S, (\tau_1, \tau_2, \tau \ldots), (\langle r_1, i_1 \rangle, \langle r_2, i_2 \rangle, v \ldots)) & = & S'' \\
\quad \text{where}\ \ S' = \mathit{LinkFields}(S, (\tau_2, \tau \ldots), (\langle r_2, i_2 \rangle, v \ldots)) & & \\
\qquad\quad\ S'' = \mathit{Tie}(S', \tau_1, \langle r_1, i_1 \rangle, \langle r_2, i_2 \rangle) & & \\
\mathit{LinkFields}(S, (\tau_1), (v_1)) & = & S
\end{array}$$

$$\begin{array}{l}
\mathit{Tie}(S, \tau_1, \langle r_1, i_1 \rangle, \langle r_2, i_2 \rangle) = S \cup \{\, r_1 \mapsto (i_e \mapsto (r_2, i_2)) \,\} \\
\text{where}\ \ r_1 \neq r_2 \ \text{and}\ \tau_1; \langle r_1, i_1 \rangle; S \vdash_{ew} \langle r_1, i_e \rangle
\end{array}$$

Fortunately, in sequential LoCal, there is already an indirection pointer mechanism that is sufficient for our purposes. In sequential LoCal, indirection pointers support unbounded allocation in a region by representing a region as a linked list of byte arrays, linked by indirection pointers. We briefly discuss the aspects relevant to LoCal$^{\text{regpar}}$. For indirections, LoCal$^{\text{regpar}}$ uses a type-directed program transformation which adds a single indirection constructor `I` to every datatype. For example, the binary tree datatype becomes:

```
data Tree = Leaf | Node Tree Tree | I (Ind Tree)
```

where an `Ind Tree` is a pointer to a value of type `Tree`. Every case expression that operates on a `Tree` is updated during compilation to have an additional clause that dereferences the indirection pointer, and then re-executes the whole case expression with that value. This clause essentially introduces a loop since the dereferenced value can itself be an indirection. With this approach, the overall changes to the program are minimal, and, it offers maximum flexibility because any `Tree` value can now be written to a separate region and pointed to by an indirection.

Discussion A consequence of LoCal$^{\text{regpar}}$ introducing fresh regions is that the schedule of evaluation dictates the way a value is laid out on the heap. Every choice to parallelize a single-region allocation implies the creation of a new region and a new indirection, thereby introducing fragmentation. If a schedule is picked carelessly the heap might become very fragmented, similar to a full pointer-based representation, and the benefits of using a serialized representation will be lost. All the subsequent traversals will have to chase indirection pointers which will slow them down. In the implementation we study in the sequel, we give users control over picking a schedule suitable for the problem at hand by allowing them to perform manual granularity control. In future work, we plan to consider adopting automatic techniques for granularity control, such as Heartbeat Scheduling [Acar et al. 2018] or Oracle-Guided Scheduling [Acar et al. 2019].

5 Implementation

We implement our techniques in the open source Gibbon compiler. It serves as a good starting point since it provides us all the infrastructure required to compile LoCal programs to C code, and a small runtime system that handles memory management and garbage collection.

Gibbon is a whole-program micropass compiler that compiles a polymorphic, higher-order subset of Haskell. Using standard whole-program compilation and monomorphization techniques [Chlipala 2015], the Gibbon front-end lowers input programs to a first-order monomorphic representation. On this first-order representation, Gibbon performs *location inference* to convert it into a LoCal program, which has region and location annotations. Then a big middle end of the compiler is a series of LoCal->LoCal compiler passes that perform various transformations. Finally, it generates C code.

Our extension that adds parallelism operates in the middle end with minor extensions to the backend code generator. We add a collection of LoCal->LoCal compiler passes that transform the program so that reads and allocations can run in parallel. At run time, we make use of the Intel Cilk Plus language extension to realize parallel execution. Our implementation follows the design described in Section 4, but we make one important change. Instead of extracting parallelism from a program implicitly, we ask the programmers to provide explicit `spawn` and `sync` annotations, which mark a computation that can be executed in parallel and a computation that must be synchronized with respectively. As a result, unlike the semantics which can exploit all available parallelism, our implementation only supports nested fork/join parallelism. While this is restrictive than the models presented before, it is sufficiently expressive for writing a large number of parallel algorithms.

Explicit `spawn` and `sync` annotations enable a fundamental optimization in parallel programs — granularity control. Implicit parallelism is elegant in theory, but the overheads of parallelism often outweigh the benefits in practice. In our system, a schedule that parallelizes too many allocations also leads to fragmentation, and in the worst case the heap might

[D-RegionPar-Case-Join]

$$T_1,\ldots,T_c,\ldots,T_n \Longrightarrow_{rp} T_1,\ldots,T_c',\ldots T_n$$

$$\begin{aligned}\text{where }\ & T_c = (\hat{\tau}_c, cl_c, S_c; M_c; e_c);\ e_c = \texttt{case}\ \langle r, \texttt{i-var}\ i_c\rangle^{l_c}\ \texttt{of}\ \overrightarrow{pat}\\ & T_p \in \{\, T_1,\ldots,T_n \,\} = (\tau_p@{l_p}^{r}, \langle r, \texttt{i-var}\ i_c\rangle, S_p; M_p; \langle r, i_p\rangle)\\ & M_3 = MergeM(M_p, M_c);\ S_3 = MergeS(S_p, S_c)\\ & e_c' = \texttt{case}\ \langle r, i_p\rangle^{l_p}\ \texttt{of}\ \overrightarrow{pat}[i_p/\texttt{i-var}\ i_c];\ T_c' = (\hat{\tau}_c, cl_c, S_3; M_3; e_c')\end{aligned}$$

[D-RegionPar-LetLoc-After]

$$T_1,\ldots,(\hat{\tau}, cl, S; M; e),\ldots,T_n \Longrightarrow_{rp} T_1,\ldots,(\hat{\tau}, cl, S; M'; e),\ldots,T_n$$

$$\begin{aligned}\text{where }\ & e = \texttt{letloc}\ l^r = \texttt{after}\ \tau@{l_0}^{r}\ \texttt{in}\ e;\ \langle r, i\rangle = \hat{M}({l_0}^{r})\\ & \tau; \langle r, i\rangle; S \vdash_{ew} \langle r, j\rangle;\ M' = M \cup \{\, l^r \mapsto \langle r, j\rangle \,\}\end{aligned}$$

[D-RegionPar-LetLoc-After-new-reg]

$$T_1,\ldots,(\hat{\tau}, cl, S; M; e),\ldots,T_n \Longrightarrow_{rp} T_1,\ldots,(\hat{\tau}, cl, S'; M'; e'),\ldots,T_n$$

$$\begin{aligned}\text{where }\ & e = \texttt{letloc}\ l^r = \texttt{after}\ \tau@{l_0}^{r}\ \texttt{in}\ e_1;\ \langle r, \texttt{i-var}\ i\rangle^{l_0} = \hat{M}({l_0}^{r})\\ & r'\ \text{fresh}; S' = S \cup \{\, r' \mapsto \{\ \}\,\}; M' = M \cup \{\, l^r \mapsto \langle r, (r', 0)\rangle \,\}\end{aligned}$$

[D-RegionPar-DataConstructor-join]

$$T_1,\ldots,(\hat{\tau}, cl, S; M; e),\ldots,T_n \Longrightarrow_{rp} T_1,\ldots,T',\ldots,T_n$$

$$\begin{aligned}\text{where }\ & e = K\ l^r\ \overrightarrow{v};\ \langle r_j, \texttt{i-var}\ i_j\rangle \in \overrightarrow{v}\\ & T_p \in \{\, T_1,\ldots,T_n \,\} = (\tau_p@{l_p}^{r}, \langle r, \texttt{i-var}\ i_j\rangle, S_p; M_p; \langle r, i_p\rangle)\\ & M' = MergeM(M_p, M);\ S' = MergeS(S_p, S)\\ & e' = K\ l^r\ \overrightarrow{v}[i_p/\texttt{i-var}\ i_j];\ T' = (\hat{\tau}, cl, S'; M'; e')\end{aligned}$$

[D-RegionPar-DataConstructor-link]

$$T_1,\ldots,(\hat{\tau}, cl, S; M; e),\ldots,T_n \Longrightarrow_{rp} T_1,\ldots,(\hat{\tau}, cl, S''; M; \langle r, i\rangle),\ldots,T_n$$

$$\begin{aligned}\text{where }\ & e = K\ l^r\ \overrightarrow{v};\ \overrightarrow{\langle r, i\rangle} = \overrightarrow{v}\\ & \overrightarrow{\tau} = GetTypes(K);\ S' = LinkFields(S, \overrightarrow{\tau}, \overrightarrow{v})\\ & S'' = S' \cup \{\, r \mapsto (i \mapsto K)\,\};\ \langle r, i\rangle = \hat{M}(l^r)\end{aligned}$$

Figure 9. Dynamic semantics (region parallel transitions).

degenerate to a full pointer-based representation. To control these overheads, we let the programmers perform manual granularity control i.e. they can mark computations to run in parallel when they predict that the benefits (speedup) would outweigh the costs (overheads), and use a sequential variant of their algorithm on small sized inputs.

5.1 Parallel Reads

Using static analysis, the Gibbon compiler can infer if a dataype requires offsets, and it can transform the program to add offsets to datatypes that need them. In sequential LoCal, these are used to preserve asymptotic complexity of certain functions. For example, `rightmost` on a binary tree would be linear instead of logarithmic without offsets. In our implementation, we use these offsets to enable parallel reads. We update that static analysis and have it add offsets if a program performs parallel reads, i.e. via a clause in a case expression that accesses its fields in parallel.

5.2 Parallel Allocations

The implementation of single-region parallel allocations closely follows the design described in Section 4. Automatically generating code that creates fresh regions and writes indirections at appropriate places is accomplished by a program transformation pass. But there still exists an issue with fragmentation. Ideally, if a parallel program runs on a single core, the heap it constructs should be identical to one constructed by its sequential counterpart. But granularity control alone cannot accomplish this. It allows us to control the grain in order to restrict excessive creation of fresh regions, but the number of regions created will always be equal to the number of `spawn`'s in the program. This still causes fragmentation because all spawned tasks might not actually run in parallel. The key insight is to make the number of fresh region allocations equal to the number of *steals*, not spawns. That is if a work-stealing scheduler is being used, but the general idea applies to other schedulers as well.

```
buildtree : ∀ $l^r$ . Int → Tree @ $l^r$
buildtree [$l^r$] n =
  if n < GRAIN
  then buildtree_seq [$l^r$] n
  else if n == 0 then (Leaf $l^r$ 1)
  else letloc $l_a{}^r$ = $l^r$ + 1 in
       let left : Tree @ $l_a{}^r$ = spawn (buildtree [$l_a{}^r$] (n - 1)) in
       if continuation_stolen
       then letregion $r_2$
            letloc $l_p{}^{r_2}$ = start $r_2$ in
            let right : Tree @ $l_p{}^{r_2}$ = buildtree [$l_p{}^{r_2}$] (n - 1) in
            let _ : () = sync in
            letloc $l_b{}^r$ = after(Tree @ $l_a{}^r$) in
            let _ : Tree @ $l_b{}^r$ = tie $l_b{}^r$ $l_p{}^{r_2}$ in
            (Node $l^r$ left right)
       else letloc $l_b{}^r$ = after(Tree @ $l_a{}^r$) in
            let right : Tree @ $l_b{}^r$ = buildtree [$l_b{}^r$] (n - 1) in
            let _ : () = sync in
            (Node $l^r$ left right)
```

Figure 10. Parallel `buildtree` transformed by the compiler such that it allocates in parallel, and also avoids fragmentation.

Our implementation transforms `buildtree` as shown in Figure 10. In this version, a fresh region is created only if the let-bound expression runs in parallel. Otherwise the body expression allocates in the parent region, like a sequential `buildtree` would. This enables parallel allocations without excessive fragmentation. The `continuation_stolen` primitive is implemented in Gibbon's runtime system using the Cilk Plus API.

6 Evaluation

In this section we evaluate our implementation using a variety of benchmarks from existing literature. To measure the overheads of compiling parallel allocations using fresh regions and indirection pointers we compare our single-core performance against the original, sequential LoCal implementation. The original LoCal is also the best sequential baseline for performing speedup calculations since its programs operate on serialized heaps, and as shown in prior work, are significantly faster than their pointer-based counterparts. Note that prior work [Vollmer et al. 2017] compared sequential constant factor performance against a range of compilers including GCC and Java. Since Gibbon outperformed those compilers in sequential tree-traversal workloads, we focus here on comparing against Gibbon for sequential performance.

We also measure the scaling properties of Gibbon by comparing its performance to other programming languages and systems that support efficient parallelism for recursive, functional programs — MPL [1] [Westrick et al. 2019] and GHC. MPL is extension of MLton [2], which is a whole program optimizing compiler for the Standard ML [Milner et al. 1997] programming language. MPL supports nested fork/join parallelism, and generates extremely efficient code, and hence serves as a baseline for comparing against a system that is pointer-based. We compare against GHC as the most optimized existing implementation of a general purpose, purely functional language Haskell.

The experiments in this section are performed on a 18 core single socket Intel Xeon E5-2699 CPU with 64GB of memory. Each benchmark is run 5 times, and the median is reported. To compile the C programs generated by our implementation we use GCC 7.4.0 with all optimizations enabled (option `-O3`), and the Intel Cilk Plus extension (option `-fcilkplus`) to realize parallelism. To compile sequential LoCal programs, we use the Gibbon compiler but disable the changes that add parallelism with appropriate flags. For MPL, we use version `20200220.150446-g16af66d05` compiled from its source code. For GHC, we use its version 8.6.5 (with options `-threaded -O2`) along with the monad-par[Marlow et al. 2011] library (v 0.3.5) to realize parallelism.

[1]https://github.com/MPLLang/mpl
[2]http://www.mlton.org/

6.1 Benchmarks

We use the following set of of 10 benchmarks to evaluate performance. For GHC, we use strict datatypes in benchmarks which generally offers the same or better performance and avoids problematic interactions between laziness and parallelism.

- **fib**: Compute the 45th fibonacci number with a sequential cutoff at n=18.
- **buildFib**: This is an artificially designed benchmark that performs lot of parallel allocations, and has enough

	LoCal	Ours				MPL					GHC				
Benchmark	T_s	T_1	O	T_{18}	S	T_s	T_1	O	T_{18}	S	T_s	T_1	O	T_{18}	S
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
fib	4.3	3.7	-12.9	0.34	12.5	16	16.2	1	1.14	14	7	7.2	3	0.6	11.7
buildFib	6.8	5.9	-13.6	0.52	13.1	25	25.1	0.2	1.8	13.9	12.7	12.7	0	1	12.7
buildTree	0.77	0.78	0.54	0.11	7.1	1.4	1.9	31.3	0.4	3.6	4	4.4	9.2	0.57	7
add1Tree	0.91	1.1	25.8	0.11	8.1	2.2	2.9	30.5	0.58	3.8	4	4.5	9.7	0.67	6
sumTree	0.24	0.29	19.1	0.03	8.5	1.04	1.03	-0.3	0.07	14.1	0.54	0.6	11.1	0.07	7.9
buildKdTree	5.3	5.3	0	2.6	2	12.6	13.5	7.1	2.2	5.7	326.9	334	2.2	118.3	2.8
pointCorr	0.14	0.14	0	0.014	10.1	0.62	0.62	0	0.05	12.9	0.16	0.18	18.1	0.014	11.1
barnesHut	16.3	16.1	-1.4	1.4	11.7	41.8	30.6	-26.9	2.2	18.9	106.5	109.5	2.8	16.2	16.6
coins	10.3	9.3	-9.7	4.7	2.18	1.9	1.3	-30.7	0.96	2.03	0.89	0.9	12.5	0.74	4.8
countnodes	0.035	0.039	11.4	0.007	4.9	0.06	0.05	-16.7	0.006	10	0.16	0.18	12.5	0.033	4.8

Figure 11. Benchmark results. Column T_s shows the run time of a sequential program. T_1 is the run time of a parallel program on a single core, and O the percentage overhead relative to T_s, calculated as $((T_1 - T_s)/T_s) * 100$. T_{18} is the run time of a parallel program on 18 cores and S is the speedup relative to T_s, calculated as T_s/T_{18}. The overhead (Column 3) and speedup (Column 5) for Ours are computed relative to sequential LoCal (Column 1). For MPL and GHC, the overheads (Columns 8 and 13) and speedups (Columns 10 and 15) are self-relative — parallel MPL and GHC programs are compared to their sequential variants. All timing results are reported in seconds.

work to amortize their costs. It constructs a balanced binary tree of depth 18, and computes the 20th fibonacci number at each leaf. This benchmark is embarrassingly parallel, and it is included here to measure the overheads of parallel allocations under ideal conditions. The sequential cutoff is at depth=8.

- **buildTree** and **add1Tree** and **sumTree**: These benchmarks are taken from LoCal's benchmark suite. `buildTree` constructs a a balanced binary tree of depth 26 with an integer at the leaf, and `add1Tree` and `sumTree` operate on this tree. `add1Tree` is a mapper function which adds 1 to all the leaves and `sumTree` is a reducer which sums up all leaves in the tree. The sequential cutoff for each of these benchmarks is at depth=18.
- **buildKdTree** and **pointCorrelation**: `buildKDTree` constructs a kd-tree containing 4M 3-d points in the Plummer distribution. Each node in the tree stores the split axis, split value, the number of elements contained in all of its subtrees, and the minimum and maximum bounds on each dimension. `pointCorrelation` takes as input a kd-tree and then calculates the 2-point correlation for an arbitrary point in it. The sequential cuttoff for both these benchmarks is at a node which contains less than 500K elements.
- **barnesHut**: This benchmark is taken from the Problem Based Benchmark Suite [Shun et al. 2012]. It constructs a quad-tree containing 4M 2-d point-masses distributed uniformly within a square, and then uses it to run an nbody simulation over the given point-masses. The sequential cuttoff for constructing the tree is when the input list contains less than 65K elements. In this case, we implemented optimizations that go beyond the race-free, purely functional style of the other benchmarks. For all three compilers, we apply point forces by updating an array in parallel, using potentially-racy mutation operations. With library support these unsafe operations can be hidden behind a pure interface.
- **coins** This benchmark is taken from GHC's NoFib [3] benchmark suite. It is a combinatorial search problem that computes the number of ways in which a certain amount of money can be paid by using the given set of coins. It uses an append-list to store each combination of coins that adds up to the amount, and counts the number of non-nil elements in this list later. Only the time required to construct this list is measured. The input set of coins and their quantities are `[(250,55),(100,88),(25,88),(10,99),(5,122),(1,177)]`, and the amount to be paid is 999.
- **countNodes** This benchmark is also taken from LoCal's benchmark suite. It operates on ASTs used internally in the Racket compiler, and counts the number of nodes in them. The ASTs are a complicated datatype (9 mutually recursive types with 36 data constructors) and are stored on disk as text files. The GHC and MPL implementations parse these text files before operating on them. For our implementation, we store the serialized data on disk in its binary format, and the program reads this data using a single `mmap` call. To ensure an

[3] https://gitlab.haskell.org/ghc/nofib

apples-to-apples comparison, we do not measure the time required to parse the text files for GHC and MPL, and for our implementation, we run the `mmap`'d file through an identity function to ensure that it is loaded into memory. The size of the input file used for MPL and GHC is 150M, and that same file serialized for our implementation is 44M.

6.2 Results

Figure 11 shows the benchmark results. The quantities in both figures can be interpreted as follows. Column T_s shows the run time of a sequential program. T_1 is the run time of a parallel program on a single core, and O the percentage overhead relative to T_s, calculated as $((T_1 - T_s)/T_s) * 100$. T_{18} is the run time of a parallel program on 18 cores and S is the speedup relative to T_s, calculated as T_s/T_{18}. The overhead (Column 3) and speedup (Column 5) for Ours are computed relative to sequential LoCal (Column 1). For MPL and GHC, the overheads (Columns 8 and 13) and speedups (Columns 10 and 15) are self-relative — parallel MPL and GHC programs are compared to their sequential variants.

	Overhead (%)	Speedup (×)
Ours	-5.1	8.04
MPL	0.16	9.89
GHC	4.93	8.54

Figure 12. Average overheads and speedups.

	T_s	T_1	T_{18}
MPL	2.18×	2.58×	1.87×
GHC	2.79×	3.8×	3.16×

Figure 13. Geomean speedups of Ours relative to MPL and GHC. Higher is better for Ours.

Our experiments show that in most cases, parallelism in a serialized representation performs as well as in a pointer-based representation. As Figure 12 shows, the overheads and speedups for Ours are comparable to those of MPL and GHC. Moreover, if we compare absolute run times (Figure 13), our implementation is significantly faster than both MPL and GHC. When utilizing 18 cores, our geomean speedup is 2.13× and 3.6× over MPL (parallel MLton) and GHC, meaning that the use of dense representations to improve sequential processing performance coexists with scalable parallelism.

6.2.1 Overheads

To compare overheads, we inspect Columns 3, 8 and 13 in Figure 11. Across all the benchmarks that measure the performance of allocations, namely `buildFib`, `buildTree`, `add1Tree`,

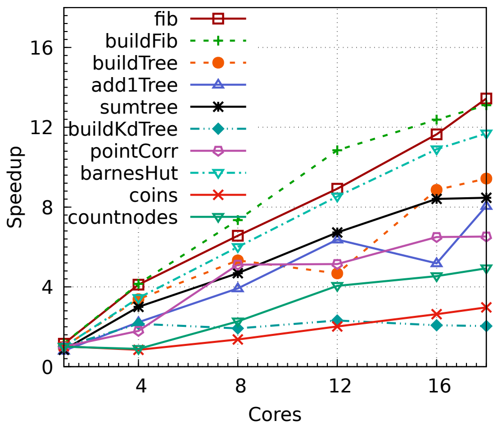


Figure 14. Speedups relative to sequential LoCal.

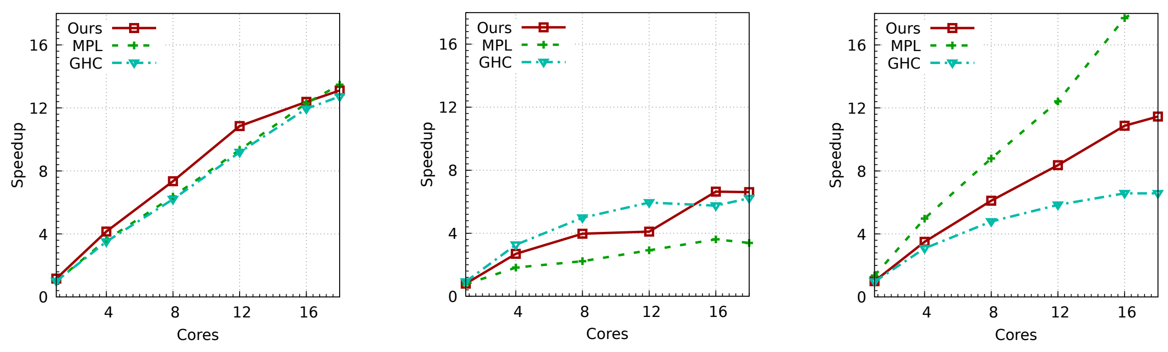
`buildKdTree`, and `coins`, only `add1Tree` has a high overhead of 25.8%; all others are under 1%.

6.2.2 Speedups

To compare speedups, we inspect Columns 5, 10 and 15 in Figure 11, and Figure 15 shows the scaling results for Ours on a 18 core machine for some selected benchmarks. For most benchmarks, speedup results for Ours are comparable to MPL's and GHC's. For `barnesHut`, our implementation's limited scaling is due to a reason unrelated to parallel allocations. While constructing each node in tree, the algorithm needs to pick point-masses that lie within a certain bounding box, and we use a standard `filter` function to implement this step. Unfortunately, the filter function in our vector library is not parallelized yet. We believe that parallelizing that will make this benchmark perform much better. If we leave out the time required to construct the tree and just measure the time required to run the nbody simulation, we observe that our implementation is 15× faster than sequential LoCal, which is much closer to MPL's scaling factor. `countnodes` is another benchmark for which both Ours and GHC don't scale very well. In our experiments, we observed that they both scale better on bigger inputs but we do not include those results here because SML/NJ's s-expression parsing library that we used for our MPL version runs out of memory while trying to parse those inputs.

7 Related Work

The most closely related work to this paper is, obviously, Vollmer et al.'s LoCal [Vollmer et al. 2019], which was summarized in Section 2.1. As discussed there, while LoCal's syntax is identical to Parallel LoCal, Vollmer et al.'s treatment only provided sequential semantics, while this paper extends those semantics to parallelism, both *fully* parallel semantics and *region parallel* semantics.



(a) buildFib **(b)** add1Tree **(c)** barnesHut

Figure 15. Self-relative scaling results for Ours on 18 cores. X axis is number of cores, Y axis is speedup.

This work, and LoCal, are related to several HPC approaches to serializing recursive trees into flat buffers for efficient traversal [Goldfarb et al. 2013; Makino 1990; Meyerovich et al. 2011]. Notably, these approaches *must* maintain the ability to access the serialized trees in parallel, despite the elimination of pointers internal to the data structure, or risk sacrificing their performance goals. The key distinction that makes enabling parallelism in the HPC setting "easier" than in our setting is that these approaches are application-specific. The serialized layouts are tuned for trees whose structure and size are known prior to serialization, and the applications that consume these trees are specially-written to deal with the application-specific serialization strategies. Hence, offsets are either manually included in the necessary locations, or are not necessary as tree sizes can be inferred from application-specific information.

Work on more general approaches for packing recursive structures into buffers include Cap'N Proto [Varda 2015], which attempts to unify on-disk and in-memory representations of data structures and Compact Normal Form (CNF) [Yang et al. 2015]. Neither of these approaches have the same design goals as LoCal and LoCal^{par}: both Cap'N Proto and CNF preserve internal pointers in their representations, eliding the problem of parallel access by invariably paying the cost (in memory consumption and lost spatial locality) of maintaining those pointers. We note that Vollmer et al. showed that LoCal's representations enable faster sequential traversal than either of those two approaches [Vollmer et al. 2019], and Section 6 shows that our approach is comparable in *sequential* performance to LoCal despite also supporting parallelism.

There is a long line of work on flattening and nested data parallelism, where parallel computations over irregular structures are *flattened* to operate over dense structures [Bergstrom et al. 2013; Blelloch 1992; Keller and Chakravarty 1998]. However, these works do not have the same goals as ours. They focus on generating parallel code and data layouts that promote data parallel access to the elements of the structure, rather than selectively trading off between parallel access to structures and efficient sequential access.

Efficient automatic memory management is a longstanding challenge for parallel functional languages. Recent work has addressed scalable garbage collection by structuring the heap in a hierarchy of heaps, enabling task-private collections of [Guatto et al. 2018], there is work proposing a split-heap collector that can handle a parallel lazy language [Marlow et al. 2009] and a strict one [Sivaramakrishnan et al. 2020], and there is work on a scalable, concurrent collector [Ueno and Ohori 2016]. All of these designs focus on a conventional object model for algebraic data types that, unlike LoCal, assumes a uniform, boxed representation. We plan to investigate how results in these collectors relate to the model used by LoCal, where objects may be laid out in a variety of different ways.

8 Conclusions and Future Work

We have shown how a practical form of task parallelism can be reconciled with dense data representations. We demonstrated this result inside a compiler designed to implicitly transform programs to operate on such dense representations. For a set of tree-manipulating programs we considered in Section 6, this experimental system yielded better performance than existing best-in-class compilers.

To build on what we have presented in this paper, we plan to explore automatic granularity control [Acar et al. 2019, 2018]; this would remove the last major source of manual programmer tuning in Gibbon programs (which already substantially automate data layout optimizations). A parallel Gibbon with automatic granularity control would represent the dream of implicitly parallel functional programming with good absolute wall-clock performance.

So far we have emphasized purely functional programs processing irregular data (trees). To continue to scale this approach up to a general purpose programming environment, we plan in the future to incorporate more well-studied data-parallel capabilities for sparse and dense multi-dimensional data. Finally, starting from the currently purely functional Gibbon language, we plan to incorporate efficient mutation of (dense) heap data, not by incorporating a standard, monadic approach, but through adding mutation primitives based on linear types, which we expect to mesh well with the implicitly parallel functional paradigm.

References

Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. ACM, New York, NY, USA, 214–228. http://mike-rainey.site/papers/oracle-ppop19-long.pdf

Umut A Acar, Arthur Charguéraud, , Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. (2018). http://mike-rainey.site/papers/heartbeat.pdf

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.* 11 (October 1989), 598–632. Issue 4. https://doi.org/10.1145/69558.69562

Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-Only Flattening for Nested Data Parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/2442516.2442525

Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language.* Technical Report. USA.

Adam Chlipala. 2015. An Optimizing Compiler for a Purely Functional Web-application Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 10–21. https://doi.org/10.1145/2784731.2784741

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex.* Mit Press.

Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing) (SC '13)*.

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut Acar, and Matthew Fluet. 2018. Hierarchical Memory Management for Mutable State. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3178487.3178494

Gabriele Keller and Manuel M. T. Chakravarty. 1998. Flattening Trees. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par '98)*. Springer-Verlag, Berlin, Heidelberg, 709–719.

Junichiro Makino. 1990. Vectorization of a treecode. *J. Comput. Phys.* 87 (March 1990), 148–160. https://doi.org/10.1016/0021-9991(90)90231-O

Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A Monad for Deterministic Parallelism. *SIGPLAN Not.* 46, 12 (Sept. 2011), 71–82. https://doi.org/10.1145/2096148.2034685

Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/1596550.1596563

Leo A. Meyerovich, Todd Mytkowicz, and Wolfram Schulte. 2011. Data Parallel Programming for Irregular Tree Computations, In HotPAR. https://www.microsoft.com/en-us/research/publication/data-parallel-programming-for-irregular-tree-computations/

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 68–70. https://doi.org/10.1145/2312005.2312018

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *arXiv preprint arXiv:2004.11663* (2020).

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Katsuhiro Ueno and Atsushi Ohori. 2016. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming.* 421–433.

Kenton Varda. 2015. Cap'n Proto. https://capnproto.org/

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 48–62. https://doi.org/10.1145/3314221.3314631

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.26

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2019. Disentanglement in Nested-Parallel Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 47 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371115

Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. 2015. Efficient Communication and Collection with Compact Normal Forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 362–374. https://doi.org/10.1145/2784731.2784735

A Metafunctions

This section contains definitions of metafunctions used in the operational semantics.

A.1 Merging task memories

$$\begin{array}{lcl}
MergeS(S_1, S_2) & = & \{\, r \mapsto MergeH(h_1, h_2) \mid (r \mapsto h_1) \in S_1, (r \mapsto h_2) \in S_2 \,\} \\
& \cup & \{\, r \mapsto h \mid (r \mapsto h) \in S_1, r \notin dom(S_2) \,\} \\
& \cup & \{\, r \mapsto h \mid (r \mapsto h) \in S_2, r \notin dom(S_1) \,\} \\
\\
MergeH(h_1, h_2) & = & \{\, Nf(i_1) \mapsto hv \mid (i_1 \mapsto hv) \in h_1, (i_2 \mapsto hv) \in h_2, Nf(i_1) = Nf(i_2) \,\} \\
& \cup & \{\, Nf(i) \mapsto hv \mid (i \mapsto hv) \in h_1, Nf(i) \notin \{\, Nf(i') | i' \in dom(h_2) \,\} \,\} \\
& \cup & \{\, Nf(i) \mapsto hv \mid (i \mapsto hv) \in h_2, Nf(i) \notin \{\, Nf(i') | i' \in dom(h_1) \,\} \,\} \\
\\
MergeM(M_1, M_2) & = & \{\, l^r \mapsto \langle r, Nf(i_1) \rangle \mid (l^r \mapsto \langle r, i_1 \rangle) \in M_1, (l^r \mapsto \langle r, i_2 \rangle) \in M_2, \\
& & \quad Nf(i_1) = Nf(i_2) \,\} \\
& \cup & \{\, l^r \mapsto \langle r, Nf(i_1) \rangle \mid (l^r \mapsto \langle r, i_1 \rangle) \in M_1, l^r \notin dom(M_2) \,\} \\
& \cup & \{\, l^r \mapsto \langle r, Nf(i_2) \rangle \mid (l^r \mapsto \langle r, i_2 \rangle) \in M_2, l^r \notin dom(M_1) \,\} \\
& \cup & \{\, l^r \mapsto \langle r, Nf(i_2) \rangle \mid (l^r \mapsto \langle r, i_2 \rangle) \in M_2, (l^r \mapsto \texttt{before}\ i\circ_1) \in M_1 \,\} \\
& \cup & \{\, l^r \mapsto \langle r, Nf(i_1) \rangle \mid (l^r \mapsto \langle r, i_1 \rangle) \in M_1, (l^r \mapsto \texttt{before}\ i\circ_2) \in M_2 \,\} \\
& \cup & \{\, l^r \mapsto \langle r, \texttt{before}\ i\circ_2 \rangle \mid (l^r \mapsto \langle r, \texttt{before}\ i\circ_2 \rangle) \in M_1, l^r \notin dom(M_2) \,\} \\
& \cup & \{\, l^r \mapsto \langle r, \texttt{before}\ i\circ_1 \rangle \mid (l^r \mapsto \langle r, \texttt{before}\ i\circ_1 \rangle) \in M_2, l^r \notin dom(M_1) \,\}
\end{array}$$

Figure 16. Metafunctions for merging task memories.

We merge two stores by merging the heaps of all the regions that are shared in common by the two stores, and then by combining with all regions that are not shared. We merge two heaps by taking the set of the all the heap values at indices whose normal forms are equal, and all the heap values at indices in only the first and only the second heap. The merging of location maps follows a similar pattern, but is slightly complicated by its handling of locations that map to before indices. In particular, for any location where one of the two location maps holds a before index and the other one holds a region index, we assign to the resulting location map the region index, because the region index contains the more recent information.

A.2 End-Witness judgement

The end witness provides a naive computational interpretation of the process for finding the index one past the end of a given concrete location, with its given type. This rule is mostly the same as the one given for the original, sequential LoCal, but additionally includes a new case for handling before indices.

Case (A) $\tau_c; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle$:

1. $\hat{S}(r, i_s) = K'$ such that
   $\texttt{data}\ \tau_c = \overrightarrow{K_1\ \vec{\tau}_1} \mid \ldots \mid K'\ \vec{\tau}' \mid \ldots \mid \overrightarrow{K_m\ \vec{\tau}_m}$
2. $\overrightarrow{w_0} = i_s + 1$
3. $\overrightarrow{\tau'_1}; \langle r, \overrightarrow{w_0} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle \wedge$
   $\overrightarrow{\tau'_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle$
   where $j \in \{\, 1, \ldots, n-1 \,\}; n = |\vec{\tau}'|$
4. $i_e = \overrightarrow{w_n}$

Case (B) $\texttt{ind}\ \tau_c; \langle r, i_s \rangle; S \vdash_{ew} \langle r, i_e \rangle$:

1. $i_e = i_s + 1$
2. $(r', i'_s) = \hat{S}(r, i_s)$
3. $\tau_c; \langle r', i'_s \rangle; S \vdash_{ew} \langle r', i'_e \rangle$

Case (C) $\tau; \langle r, \texttt{before}\ i\circ \rangle; S \vdash_{ew} \langle r, i\circ \rangle$

Figure 17. The end-witness rule.

Effective Host-GPU Memory Mangement Through Code Generation — DRAFT —

Hans-Nikolai Vießmann
Radboud University
Nijmegen, NL
h.viessmann@ru.nl

Sven-Bodo Scholz
Radboud University
Nijmegen, NL
svenbodo.scholz@ru.nl

Abstract

NVidia's CUDA programming environment provides several options on how to orchestrate the management of host and device memory as well as the transfers between them. This paper looks at how the choices between these options affect the performance of applications on a set of different GPU architectures.

We provide a code generation scheme that translates memory-agnostic programs into different CUDA variants and present some initial performance evaluations based on a concrete implementation in the context of the SaC compiler: for a simple compute kernel we see 30% runtime improvements when switching from the default options to a more suitable combination of allocations and memory transfers.

1 Introduction

NVidia's CUDA framework and CUDA-compatible GPUs are an industrial standard for most GPU-based computations. The favourable performance price ratio of GPUs combined with their suitability for many data intensive applications has led to a very quick evolution of new GPU hardware. Besides improvements in the GPU designs themselves, particular effort has been spent on improvements for managing the data transfers between hosts and GPUs. These novelties have led to extensions in the CUDA standard. While such extensions typically allow for a better utilisation of new hardware features, they pose challenges to code portability and code maintenance. Some of the newer features are not supported for older hardware, others introduce a vast overhead. Even if code is specifically constructed to be used on one particular hardware, figuring out which part of the CUDA standard is most suitable for a given task is not easy. NVidia's documentation at https://docs.nvidia.com/cuda/ alone provides different tuning guides for the latest five different architectures.

IFL '20, September 2–4, 2020, Online
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This need for architecture-specific tuning and its quickly evolving, volatile nature suggests that generative programming can provide the desired application stability while reducing the burden of rewrites for performance portability to a minimum. Indeed, several approaches exist [7, 8, 10, 15**?**] that demonstrate how systematic code re-writes can substantially improve GPU performance using either annotations, heuristics, or machine learning to guide the rewriting process. While the preexisting work mainly focuses on kernel construction and interplay, this paper is concerned with the memory management on host and device as well as the orchestration of the communication between them after the kernels have been decided upon. CUDA 10.1 offers several mechanisms to manage memory and to orchestrate data transfers between different memories:

- Memory on host and device can be allocated separately or in a unified fashion;
- Host memory allocations can be done through the operating system or CUDA itself;
- Transfers can be made synchronously or asynchronously;
- Depending on the choices above, transfers need to be explicit or can be triggered implicitly; synchronisations are being done implicitly or need to be inserted explicitly into the code.

The choices between these options depend on the capabilities of the executing architecture as well as the characteristics of any given application.

In this paper we present our experiences when investigating how to best leverage these options when compiling a functional language down into GPU kernels. Our main contributions are:

- An overview of the memory allocation and memory transfer mechanism currently available in CUDA and how they need to be orchestrated to avoid race conditions.
- A bandwidth comparison between CUDA's different memory transfer options for a set of different GPUs. It shows that there are differences of up to a factor of 2 in bandwidth between the different transfer methods. The comparison also shows that the differences in bandwidth are dependent on the hardware being used.
- A code generation scheme that enables the generation of five different CUDA code variants from a single source program. This includes provisions for safely

overlapping GPU activity and host code executions in the presence of asynchronous communications communications between them and asynchronous kernel launches.

- Some initial performance analyses of the different versions based on a full-fledged implementation in the context of the SaC compiler.

2 CUDA and its Memory Management

NVIDIA's CUDA is a software framework and driver to implement and run applications on NVIDIA's GPU devices. Like other many-core architectures, GPUs build on the following design: the GPU has its own memory, data needs to be transferred from the CPU-based host system to the GPU and back, and computations on the GPU are captured in so-called *kernels*. CUDA provides various API's to interact with the GPU device. Here, we only introduce those variants relevant to the presented work. For a full account we refer the reader to the most recent CUDA manual [14].

CUDA kernels are always launched from the host and are then executed asynchronously on the GPU [18]. For that purpose, the GPU has its own scheduler that non-preemptively executes the kernel [9].

In practice, there are three communication models provided by CUDA, synchronous communication, asynchronous communication, and managed communication. We use a simple example which can be seen as a canonical example for most GPU codes to explain the differences between these communication models. Listing 1 presents our canonical example using the standard, synchronous communication.

```
1  __global__ void increment_kernel(int *a)
2  {
3    int i = blockIdx.x * blockDim.x + threadIdx.x;
4    a[i] = a[i] + 1;
5  }
6
7  int main () {
8    int *a, *d_a;
9    a = (int *)malloc(1024*sizeof(int));
10   cudaMalloc(&d_a, 1024*sizeof(int));
11   ... // A
12   cudaMemcpy(d_a, a, 1024*sizeof(int),
13                        cudaMemcpyHostToDevice);
14   ... // B
15   increment_kernel <<<16, 64>>> (d_a);
16   ... // C
17   cudaMemcpy(a, d_a, 1024*sizeof(int),
18                        cudaMemcpyDeviceToHost);
19   ... // D
20   cudaFree(d_a);
21   free(a);
22   return 0;
23 }
```

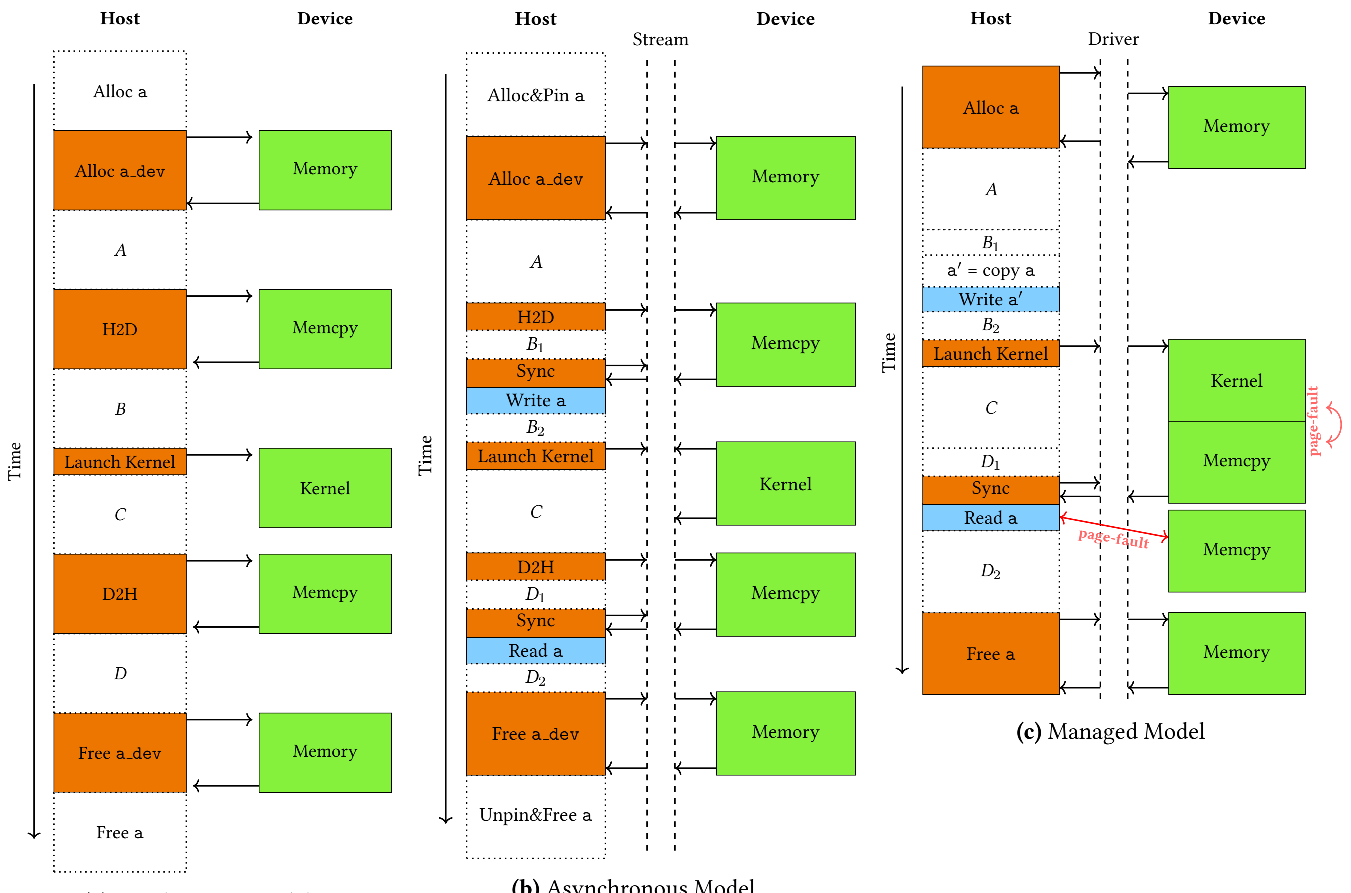
Listing 1. Canonical Example CUDA Code with Synchronous Communication

In lines 1-5 we define a kernel function `increment_kernel` which performs an element-wise increment of the argument array `a`. The `main` function essentially consists of five phases: memory allocation (lines 9 and 10) for the host and the device (GPU), transfer of the kernel argument from host to device (lines 12 and 13), kernel invocation (line 15), transfer of the result back from the device to the host (lines 17 and 18) and finally memory deallocation in lines 20 and 21. We assume further host code to exist between these phases as indicated by ...s. These code snippets perform the actual host operations including the initialisation of the host array and the interpretation of results that have come back from the GPU. However, since we are only interested in memory management and communication, we leave out these particulars here.

Figure 1 provides a comparison of how our canonical example is executed using the different memory and transfer options of NVIDIA's CUDA. For each model, we demonstrate how host and device interact over time. Our time axis evolves from top to bottom and each handshake between host and device is indicated by a horizontal arrow. In the sequel, we discuss each model separately and relate them to our canonical example from Listing 1. We discuss the required code adjustments for implementing the different models in the corresponding sub-sections.

2.1 Synchronous Communication

In Figure 1a we show the timeline of events that occur when running our code example with the default, *i.e.*, synchronous transfers. The first events perform an allocation of memory, both on the host and device. In the case of the device, the allocation is communicated through the CUDA driver and this operation blocks any further execution, causing the application to wait *until* the operation is completed. In block *A* we perform some host-side work on `a`, and continue to the next event. At this stage we now transfer the data from the host to the device using **`cudaMemcpy`**. Notice in the code example that among its parameters is a flag indicating the direction of the communication, in this instance we use `cudaMemcpyHostToDevice`. Each call to **`cudaMemcpy`** blocks further work from happening on the host, which must wait till the operation is done. In block *B* we do some more host-side work, and finally we launch our kernel. The kernel launch is the only host device interaction here which is asynchronous in nature. Since both, the host and the GPU operate on separate memory no further synchronisation is needed until the results of the kernel execution are needed, *i.e.*, the host can execute block *C* completely independent of the kernel execution on the GPU. When the host eventually calls **`cudaMemcpy`**



(a) Synchronous Model

(b) Asynchronous Model

(c) Managed Model

Figure 1. Diagram of CUDA Communication Models: Here we show the differences in synchronisations that happen between host and device for the three main communications models available through CUDA. Within each of the three models, we show host-side executions of CUDA operations in orange boxes on the left and GPU executions in green boxes on the right. Blue boxes indicate host side activities that require the insertion of explicit synchronisations or other host code modifications to avoid race conditions or erroneous results. Horizontal arrows indicate handshake events. Red lines indicate CUDA-initiated handshakes. *Note* that *H2D* stands for host-to-device and *D2H* stands for device-to-host.

with `cudaMemcpyDeviceToHost` in order to transfer the result back, this ensures synchronisation of the two activities: The host waits for the kernel and the memory transfer to complete before continuing with block *D* on the host side. Finally, we deallocate our host and device memories.

Overall, we can observe that the synchronous setup in this model ensures a tight synchronisation. Possible latency hiding that could be gained from the DMA (direct memory access) capabilities of modern GPUs can not be leveraged here.

2.2 Asynchronous Communication

Figure 1b shows the timeline of events for the canonical example when using asynchronous communication. It allows for overlapping device transfers with further operations on the host: While the GPU performs DMA operations on the host, the host itself can proceed. On the software side, CUDA implements this through the introduction of a queue-like structure into which device-side operations, like transfers and kernel launches, can be staged. This so-called *stream* moves the scheduling of device operations away from the host application and into the CUDA driver. In this way, the staged operations are being executed one after the other by the GPU, but independently of the execution on the host.

The main modification of the source code is the use of the asynchronous transfer function **`cudaAsyncMemcpy`** instead of its synchronous counterpart **`cudaMemcpy`** in lines 12 and 17 of our canonical example in Listing 1. As shown in Figure 1b, we now can overlap the execution of block *B* on the host with the memory transfer from the host to the device and the execution of block *D* with the memory transfer back. While this may seem an easy gain by simply replacing **`cudaMemcpy`** by **`cudaAsyncMemcpy`**, this can not be done without extra precautions. The challenge here is that it is no longer clear when a transfer is completed. For the transfer to the GPU, this means that we no longer know until when the reference to the host memory a is being needed for the transfer; for the transfer back it means that we actually do not know when we can start using the result. Consequently, If the host code within block *B* wants to re-use the memory of a in any which way, we need to inject an explicit synchronisation, *i.e.*, a call to **`cudaDeviceSynchronize`**, before that use. In Figure 1b, this is indicated by the upper blue box and the synchronisation that precedes it. Likewise the lower blue box indicates the first use of the result which needs to be preceeded by an explicit synchronisation as well. Failure to do so may result in a transfer of wrong data to the device or in erroneous result values.

Besides changing the transfer function, the asynchronous model also requires some extra provisions for the host memory to enable effective DMA transfers. CUDA offers two different ways of allocating such host memory. One variant builds on CUDA's own memory allocator for host memory, the other variant uses the system allocator via `malloc` but requires a special registration with the device driver.

CUDA Host Allocator. Through CUDA we have access to the **`cudaHostAlloc`** function, which is analogous to the system `malloc`. By using it, we allocate host memory that is marked as page-locked. Additionally we register it with the CUDA driver and provide properties that affect how the memory is used. The resulting pointer can be passed to either **`cudaMemcpy`** or **`cudaAsyncMemcpy`**. As the pointer is handled by CUDA, we can only free it by using **`cudaFreeHost`**. Use of this allocator comes with higher overheads as, unlike with the system allocator which can delay allocating physical memory, we need a physical memory address in order to page-lock the memory. Modifying our code example, we replace our `malloc` on line 10 and the `free` in line 21 as follows:

```
10  cudaHostAlloc(&a, 1024*sizeof(int),
11                cudaHostAllocDefault);
12  cudaMalloc(&d_a, 1024*sizeof(int));
 |  ...
19  cudaFree(d_a);
20  cudaFreeHost(a);
```

The pointer to the allocated memory is returned through the first parameter of the function `cudaHostAlloc`, which is followed by the number of bytes to allocate.

CUDA Host Register. Instead of using CUDA's host allocator, we can register memory allocated by the system allocator. The effect is identical to using CUDA's allocator, but provides one key advantage — we can delay the pinning of the memory. Furthermore, as the operation itself does not allocate physical memory, we can leverage the system allocators delayed allocation. This could reduce the overheads that happen with using CUDA's allocator. Staying with our example code, pinning by calling **`cudaHostRegister`** can be done at any point between the initial allocation and the transfer call. In our example we do this directly after `malloc`, and we unpin the memory after the last transfer. When calling the register function, we pass the allocated pointer and a flag indicating what properties the returned pointer should have. The `cudaHostRegisterDefault` is sufficient, ensuring that the pointer is treated the same in all contexts. This leads to the following changes of our example from Listing 1:

```
10  a = (int *)malloc(1024*sizeof(int));
11  cudaHostRegister(a, cudaHostRegisterDefault);
12  cudaMalloc(&d_a, 1024*sizeof(int));
 |  ...
19  cudaFree(d_a);
20  cudaHostUnregister(a);
21  free(a);
```

2.3 CUDA Unified Memory

With CUDA version 4.0, the memories of the host system and GPU device were combined into a single virtual address space, called Unified Virtual Addressing (UVA). This allows for pointers created by the CUDA API to be used on both the host and the device. Additionally, the concept of zero-copy memory was introduced, which allows the GPU to access pinned host memory without an explicit transfer operation.

Later in CUDA version 6.0, UVA was extended by the unified memory (UM) model, which introduced the concept of *managed* memory [13]. Managed memory departs from the idea of two explicit memories and explicit transfers between them completely. Memory on both sides, the host and the device, is being allocated in a single call to a CUDA specific memory allocator. The function **`cudaMallocManaged`** allocates memory using UM. The resulting pointer is tracked and if it is accessed from a non-local context (*e.g.* GPU device accessing host memory), the data is transferred implicitly.

Depending on what version of CUDA is used, and even what generation of CUDA device is used, the underlying behaviour of UM can vary. For versions of CUDA older than 8.0, and CUDA devices architectures before than Pascal, the implicit transfer of data happens as part of the kernel launch, where the entire memory associated with a managed pointer is transferred. Because of this, explicit synchronisations after the kernel launch are needed to keep the view of memory consistent in all contexts.

In versions of CUDA after 8.0, and device architectures like Pascal and newer, the transfers are initiated by demand-paging. Here an access to some host-based memory from the GPU device causes a page-fault, which the CUDA driver reacts to by sending the missing page. The driver actually send several consecutive pages, in varying quantities, whenever a page-fault occurs [12]. Here, host-side accesses to data located in device memory are resolved implicitly by the CUDA driver without an explicit synchronisation.

We depict this behaviour in the timeline for the canonical example when using the managed model in Figure 1c. The first observation is that there is only one allocation for a. It triggers allocations on the host and device. However, when exactly these are being actually performed is beyond the programmers control. The explicit transfers have been elided from the host. Transfers from the host to the device are being triggered through memory accesses in the kernel. This is indicated by the green "Memcpy" box after the "Kernel" box and the red double-sided arrows which indicate the interleaving of kernel executions and memory transfers. Similarly, the transfer back is happening implicitly when the host starts reading the results.

As in the asynchronous model, the managed model requires a synchronisation before the first read of the results so that we can make sure all computation on the GPU has terminated before data transfers back to the host happen.

To implement the managed model in our canonical example, we replace both memory allocations by a single call to **cudaMallocManaged**. Additionally we can elide our device memory allocations from line 11 as we no longer have a notion of host or device memory. In this way we update the parameter of our kernel to be a. We also remove the explicit transfer operations and insert a synchronisation before the results are being used, resulting in the following code:

```
10  cudaMallocManaged(&a, 1024*sizeof(int),
11                    cudaMemAttachGlobal);
 |  ...
15  increment_kernel <<<16, 64>>> (a);
16  ...
18  cudaDeviceSynchronize();
 |  ...
20  cudaFree(a);
```

One more aspect worth mentioning here is that the use of a unified view on a and a_dev has consequences if the canonical example makes further use of a while the data resides in the device. In the asynchronous model we already noticed that in such cases additional synchronisation is required to ensure that the data has been transferred completely to the GPU. In the managed model, such a synchronisation is not possible at all as there is no way to enforce the data to reside on either the host or the GPU only. If such a case arises, separate host memory needs to be allocated and the data of a may need to be copied. We indicate this in Figure 1c by the white box preceding the upper blue box.

Memory Prefetch. The UM system's reliance on demand driven transfers can make it less efficient in communication in comparison to the explicit communication orchestration described previously. Explicit prefetching can be triggered by using cudaMemPrefetchAsync function which in our example could be injected in those positions where the explicit transfers in the original example are placed.

2.4 Summary

With these different CUDA host memory models, we identify five distinct methods for performing transfers: (1) synchronous communication, (2) asynchronous communications using host allocation (which implicitly pins memory), (3) asynchronous communications with separately registered host memory, (4) implicit communication using CUDA managed memory and, finally, (5) implicit communication with explicit prefetch. For the rest of this paper we will refer to these respectively as **sync**, **async_alloc**, **async_reg**, **man**, and **man_prefetch**.

3 Memory Transfer Performance

From the previous section, we can see that switching the memory model of a CUDA application has the potential to lead to improved overlapping of host and GPU activity. We can also see that such a switch requires several subtle changes beyond just switching the allocation and transfer functions.

In this section, we investigate whether we can expect gains in transfer bandwidth when switching the memory model. We use a synthetic workload very similar to the canonical example of the previous section as test vehicle. We allocate and transfer a single array, of differing lengths, to the GPU device and perform a simple computation like element-wise incrementation. After this we transfer the array back to the host. In these workloads we intentionally use large arrays, taking up to half of GPU global memory, making the computation IO-bound.

Due to the simplistic nature of the benchmark we restrict ourselves to the models **sync**, **async_reg**, and **man**. We run these on two GPU devices, an NVidia K20 (Kepler architecture from 2012) and an NVidia RTX 2080 Ti (Turing architecture from 2018)[1] We use NVidia's profiling tool **nvprof** to measure the bandwidth of the memory transfers that is being achieved. The results of our experiments are shown in Figure 2 and 3.

For each memory model, we distinguish between host to device (*HtoD*) and device to host (*DtoH*) communication as the corresponding bandwidths differ significantly.

For the older Kepler architecture of the K20 in Figure 2, we can see that the asynchronous communication achieves the highest throughput at over 6 GB/s followed closely by managed memory communication. The default synchronous communication lags behind at just under 4 GB/s.

In the more recent Turing architecture of the RTX 2080 Ti in Figure 3 we have a different picture. Here, asynchronous communication is the best at a throughput of about

[1]Further details of these systems can be found in the table in Section 6.

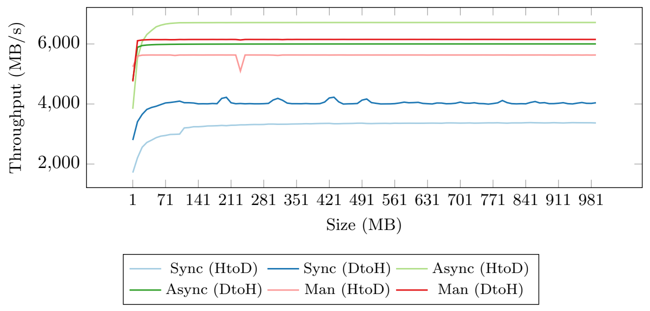


Figure 2. Data throughput to/from NVidia K20 GPU

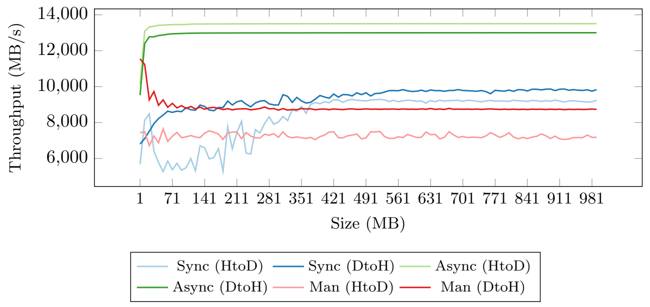


Figure 3. Data throughput to/from NVidia RTX 2080 Ti GPU

13 GB/s. Synchronous and managed memory communication lag significantly behind, with peak throughput of about 9 GB/s.

It is not surprising that asynchronous communication achieves the highest throughput on both systems, as it makes use of the DMA, avoiding CPU IO overheads. The behaviour of managed memory on both systems is different, with the measurements for the RTX 2080 Ti showing a large amount of variance in throughput as we change the size of the input array. Given that both synchronous and managed memory rely on the CPU for memory IO, the variance can be attributed to the dynamically changing clock-rate of the CPU for the given system. Additionally this can lead to slowdowns in general, explaining why managed memory communication in particular does not peak as high as asynchronous communication.

From these results, we draw several conclusions. Firstly, the differences in bandwidth can be significant. For both architectures, we see up to a factor of 2 difference between the smallest and largest bandwidth. Secondly, the relative behaviour depends not only on the GPU architecture but also on the host capabilities as well as on the host configuration (*e.g.* frequency scaling). Finally, while asynchronous transfer bandwidths seem to be almost agnostic to the amount of data that is being transferred, this is less so for the other two. In particular on the Turing architecture, it seems that the bandwidths for managed transfers outperform synchronous transfers while data less than 400MB is being transferred while this picture reverses for larger transfers.

With these results, it seems inevitable to adjust the memory model to a given combination of algorithm, host and GPU when trying to achieve the best possible overall runtime performance.

4 Generating CUDA from SaC

SaC is a functional array programming language that exposes no notion of hardware to the programmer: the use of GPUs, threads or even the notion of memory, be it on the host or the GPU-device, is hidden completely[2]. Our increment example from Section 2, in SaC, reduces to the purely computational aspects. Looking at the parts shown in Section 2 and inlining the increment function leads to a SaC code snippet of the form:

```
1  int main () {
2    ...
3    a = { iv -> a[iv] + 1;};
4    ...
5    return 0;
6  }
```

Note here, that not only are all memory related operations gone; the notion of a kernel has disappeared too, along with any indication that the variable a on the left hand side of line 3 can denote the same memory location as the variable a on the right hand side of that line.

This completely implicit notion of memory and memory transfers makes SaC an ideal starting point for generating CUDA code variants for the different memory models, adhering to all the synchronisation particulars as discussed in Section 2.

Several techniques have already been developed and implemented in the context of SaC which transform, optimise and eventually generate target architecture and resource-aware codes for efficient executions on a wide range of platforms [3, 7, 11, 16]. This includes a back-end for generating CUDA code from SaC programs.

In the sequel, we sketch the major stages of the compilation into CUDA that are relevant if we want to generate code for the different memory models explained in Section 2. As described in [7], the CUDA back-end during compile time introduces the notions of host-memory and device-memory, as well as explicit transfers between them. It also tries to minimize memory transfers between the two. For our given example, most likely, it would fuse the initial computation of the array a, *i.e.*, whatever happens in the code represented by the three dots in line 2, with the increment in line 3. Furthermore, it would also fuse that computation with whatever happens with the incremented version of the array a in the three dots of line 4.

[2]More details on SaC can be found elsewhere, e.g. in [4, 17].

For the sake of presentation, let us assume here that such a fusion is not performed and that there is no way of producing or consuming a directly on the GPU. Consequently, the code generator described in [7] would generate some intermediate code of the form:

```
1  int main () {
2    ...
3    a_dev = _host2device_(a);
4    b_dev = { i-> a_dev[i] + 1;}@CUDA;
5    b = _device2host_(b_dev);
6    ...
7    return 0;
8  }
```

Here, we see how the compiler has introduced the notion of two different memories (host and device), explicit transfers between them, and has identified the kernel itself as the array computation in line 3 (denoted by the postfix @CUDA). For readability, we have postfixed all device-allocated memories with _dev, and left all host allocated memories without postfixes. It should be noticed though, that, at this level of abstraction, the identifiers still do not refer to memory locations. The notion of memory is introduced at a later stage; it comes with the notion of references and operations for dynamic reference counting. At that stage, the code roughly looks like this:

```
1  int main () {
2    ...
3    a_dev = _dev_alloc_(1024, int);
4    a_dev = _host2device_(a);
5    b_dev = _dev_reuse_(a_dev);
6    b_dev = { i-> a_dev[i] + 1;}@CUDA;
7    b = _alloc_or_reuse_(1024, int, a);
8    b = _device2host_(b_dev);
9        _dev_decrc_(b_dev);
10   ...
11   return 0;
12 }
```

On this level of abstraction, we have explicit operations for allocating memory (_alloc_), reusing pointers (_reuse_), potentially reusing pointers (_alloc_or_reuse_), freeing memory (_free_) and potentially freeing memory (_decrc_). All these operations have two variants depending on whether they pertain to device memory (prefixed by _dev_) or to host memory. The uncertainty in some of the operations stems from the fact that aliasing analyses are undecidable in principle. As a consequence, dynamic inspections of reference counters are necessary, to determine whether some memory can be reused or needs to be freed. Details on reference counting in general and the specifics of the SaC compiler can be found in [1] and in [5], respectively.

Once explicit memory operations have been introduced, the SaC compiler moves to generating C code. Primitives like _alloc_ are transformed into intermediate code macros (ICMs), which are latter resolved by the C compiler. These allow for variants of code to materialise at compile-time, depending on parameters set by the SaC compiler (and the user). Additionally certain statically determined properties for our array variables are set, these include shape information and the reference count. This information is stored as adjacent variables that share the same name as the array but are postfixed indicating their purpose. This information is used by the runtime system to, for instance, determine if a variable can be freed, or even reused, at a particular point. With that we get the following C source code:

```
1  __global__
2  void sac_cuda_knl_1024(int * a) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      a[i] = a[i] + 1;
5  }
6  int main () {
7    ...
8    SAC_CUDA_ALLOC (a_dev, 1024, int)
9    SAC_CUDA_MEM_TRANSFER (a, a_dev, 1024, int,
10                          cudaMemcpyHostToDevice)
11   SAC_ND_REUSE (b_dev, a_dev);
12   dim3 block(16);
13   dim3 grid(1024/16);
14   sac_cuda_knl_1024<<<block, grid>>>(b_dev);
15   SAC_ND_ALLOC_OR_REUSE (b, 1024, int, a)
16   SAC_CUDA_MEM_TRANSFER (b_dev, b, 1024, int,
17                          cudaMemcpyDeviceToHost)
18   SAC_CUDA_DEC_RC_FREE (b_dev)
19   ...
20   return 0;
21 }
```

At this stage, the generated code now looks similar to our example code in Listing 1. All of the ICMs are direct translations from the SaC primitives, the only difference is the explicit computation of the grid and block sizes, where the compiler has set the block size to 16. When this source code is passed to the C compiler, the ICMs will resolve into a sequence of C function calls. For instance, SAC_ND_ALLOC_OR_REUSE will resolve into something similar to:

```
1  b = a_refcnt == 1 ? a : malloc(1024*sizeof(int));
```

Through the definition of the ICMs, we can change what code materialises, for instance the SAC_CUDA_MEM_TRANSFER in Jing's version resolves into a **cudaMemcpy**. If we want to change this into an asynchronous transfer, it suffices to change that macro expansion into **cudaAsyncMemcpy**. Similarly, we can change this expansion into an empty expansion when targeting managed memory.

In the next section we will present the compiler transformations we have developed to switch between five CUDA code variants. This also includes a transformation introduce managed memory and a transformation to add synchronisations at optimal positions in the code for the asynchronous code variants.

5 Generating Code for CUDA Transfer Mechanisms

In the previous section we introduced some parts of the SaC compiler code generation to synthesise different code variants from a single SaC source file. In this section we present a code generation scheme for creating program variants that make use of the one of the five CUDA memory models. Concretely, we present:

- an extension of the EMR optimisation [19] to determine memory reuse candidates for functions calls such as the CUDA transfer methods, *i.e.* **cudaMemcpy**,
- a compiler transformation that inserts at optimial positions in the code explicit synchronisations when using one of the asynchronous memory models,
- a compiler transformation to unify the memory models used by the compiler to generate CUDA managed memory code, and
- changes to ICMs definitions to make switching between the CUDA memory models at compile-time possible.

5.1 Extension to the Extended Memory Reuse Optimisation

The Extended Memory Reuse (EMR) optimisation [19] elides memory allocations for *with*-loops by reusing memory. It builds on top of other reuse techniques, such as in-place reuse [2] and reuse through polyhedral analysis [6], by inferring a pool of candidates from all preceding allocations, including those which are out of scope. These *extended* candidates need only be of the same type and shape, and must not be referenced after the *with*-loop being inspected. The effect of this on runtime is particularly effective when dealing with GPU device memory. Unlike memory allocated on the host, the allocation to GPU memory cannot be delayed till first write.

Though this is effective when generating CUDA code, it misses out on the additional memory operations that happen when performing transfers over the PCIe bus. In general, we must always allocate one buffer to store the data being transfers. This buffer may be allocated on the host or device, depending on the transfer direction. Once the buffer has been filled, its counterpart on the host or device is typically freed at this point. When dealing with an asynchronous memory model, we additionally need to pin the host-side memory before transferring. For a simple application like our example in Listing 1, this allocation and freeing of buffers does not have a large effect on runtime. If the code becomes more complex, for instance by iteratively on the host checking the status of some device computation, then memory operations and pinning before the transfer can significantly impact runtime.

In order to better explain this, we provide an example based on our working example in Listing 1. The difference here is that we launch the kernel iteratively and check on each iteration if the sum of the device-side array has reached some limit. This check occurs on the host, meaning on each iteration we transfer the current state of the array back to the host. The SaC code for this:

```
...
do {
    a = { i -> a[i] + 1 };
} while (sum (a) < LIMIT);
...
```

As it currently stands, the EMR optimisation would result in the following intermediate representation. Note that we have not included any memory operations:

```
...
a_dev = _host2device_(a);
do {
    a_dev = { i -> a_dev[i] + 1; }@CUDA;
    a_tmp = _device2host_(a_dev);
} while (sum (a_tmp) < LIMIT);
b = _device2host_(a_dev);
...
```

There are two operations here that we would like to remove. The first is the allocation (and free) due to the transfer in the loop. The other is the redundant transfer after the loop, which could equally well be replaced with an alias to `a_tmp`. There are no clear reuse candidates within scope. The solution here is to force an allocation before the loop, that is we allocate `a_tmp` outside the loop, avoiding the additional memory operations in the loop. As we are not dealing with memory yet within the compiler, we instead choose to create an assignment of `a` to `a_tmp`, which will eventually create a copy of `a`. The redundant transfer after the loop can now be updated to be an assignment of `a_tmp` to `b`. This results in our new code:

```
...
a_tmp = a;
a_dev = _host2device_(a);
do {
    a_dev = { i -> a_dev[i] + 1; }@CUDA;
    a_tmp = _device2host_(a_dev);
} while (abs (a_tmp) < LIMIT);
b = a_tmp;
...
```

5.2 Inserting Explicit Synchronisations

When generating multithreaded code, the orchestration of threads becomes critical in preventing race conditions and other unwanted behaviours. Similarly, when dealing with a heterogeneous platform, communication between host and device need to be managed. In the CUDA backend such synchronisation is not necessary when generating code for the default memory model as all host and device actions happen in sequence. By introducing the other models into the SaC compiler, explicit synchronisation is necessary. In particular, as is shown in Figure 1, we need to synchronise anytime we try to modify an array which is also being transferred, otherwise we may corrupt the data.

In the context of code generation, determining when such a situation might occur is non-trivial, due in part to aliasing of variables. We instead only know when we will initiate a transfer and where it needs to be *completed by*. If we have some transfer from the host to the device, we know the transfer with start at the call of the transfer function. It will need to be completed *before* the first reference of the transferred array. This gives us a kind of window in which we must at some point synchronise.

With this intuition, we designed a transformation which introduces the notion that there is a distinct start and end position within the syntax tree for a given transfer. The first stage of the transformation replaces all transfer primitives with a paired primitive, *e.g.*:

```
...
a_tmp_dev = _host2device_start_(a);
a_dev = _host2device_end_ (a_tmp_dev, a);
...
```

Notice that we have an intermediate variable `a_tmp_dev`. We do this to maintain static single-assignment form, and to also ensure that optimisations like dead-code removal don't elide the `_end_`. We do this by making the new variable a parameter of the `_end_` primitive.

From here, the optimisation then tries to create the synchronisation window, by pushing the transfer primitives apart. In general, we try to move **`_host2device_start_`** *up* and **`_device2host_end_`** *down*. Typically, the initial transfer primitives are placed before and after a kernel launch, but can be placed further up (or down) in the syntax tree depending on references to its parameters. Given this, we want to move a host to device transfer to just after the assignment of its host array. Similarly we want the device to host to be finished before its first reference of its host array.

With managed memory, there is no need to synchronise on each transfer. Instead we need to synchronise on all array references after a kernel launch were the array were parameters to the kernel. We do this by adding a synchronisation after the kernel launch immediately before such a reference.

5.3 Generating CUDA Managed Memory Code

With CUDA managed memory, there is no concrete distinction between host and GPU device memory any more. Pointers to memory are reachable in both the host and device context, meaning that the SaC compiler's use of explicit host and device types is redundant.

We implementation a transformation to elide all transfers and change all device types to host types. We define two variants of the transformation, one for the general case and other for managed memory with prefetching. The first one scans through the syntax tree and replaces all occurrences of a device type (postfixed with `_dev`) with its equivalent host type. Additionally, as managed memory implicitly moves data over the PCIe bus, explicit `memcpy`s are not needed so we remove these and replace them with assignments. The other compilcation scheme performs the same transformation, but additionally replaces `memcpy`s with calls to prefetch memory. When using managed memory with prefetching, instead of removing transfers we replace these with the intermediate representation from our code example in Listing 1. Based upon are example code in Listing 1, the transformation with prefetching would result into the follow:

```
...
a_tmp = _prefetch2device_(a);
b_tmp = { i-> a_tmp[i] + 1;}@CUDA;
b = _prefetch2host_(b_tmp);
...
```

Notice that in either case, once a device typed variable is replaced, we also change its name in order to maintain static single-assignment form. At the compilation stage where we generate C code, these assignments with be treated as aliases.

5.4 Extending the Runtime System

In Section 4 we described the code generation of the SaC compiler, resulting in C code with most SaC primitives replaced with intermediate code macros (ICMs). The ICMs are part of the runtime system of the compiler. Here variants of code are stored and at the time when the C compiler is called, the ICMs are expanded to actual code. We will use these to introduce the CUDA functions necessary to make use of the different transfer operations and memory models. Which CUDA transfer operations is ultimately generated is set by supplying the compiler "`-target`" flag with a particular target, for instance `cuda` uses synchronous transfers, `cuda_reg` and `cuda_alloc` use asynchronous transfers, and finally `cuda_man` uses the managed memory model. This commandline flag sets a macro flag, which affects what the ICMs resolve into.

We have through previous examples introduced a few of the ICMs that appear in our generated code. We now introduce ICMs which are used only for the CUDA backend of the compiler:

```
SAC_CUDA_ALLOC (var, size, type)
SAC_CUDA_FREE (var)
SAC_CUDA_DEC_RC_FREE (var, count)
SAC_CUDA_MEM_TRANSFER (src, dst, size,
                       type, direction)
```

The ALLOC and FREE ICMs resolve into the CUDA allocator and free functions for GPU device memory. The RC_DEC ICM extends on this by additionally checking the reference count, and if it is 1, freeing the device memory. The TRANSFER ICM resolves into a memory transfer functiona call, *e.g.* **cudaMemcpy**.

The three transfer methods are reasonably different that we must extend the existing runtime system in order to fully make use of them. We do this by introducing some new ICMs and changing the code generation to correctly place these. In general, we either need to extending existing allocation and free ICMs, or replace them entirely. The latter case is for instance important for using managed memory. The following are the new ICMs we introduce into the runtime system:

```
SAC_CUDA_HOST_ALLOC (var, size, btype)
SAC_CUDA_HOST_FREE (var)
SAC_CUDA_HOST_DEC_RC_FREE (var, count)
SAC_CUDA_REGISTER (var, size, btype)
SAC_CUDA_UNREGISTER (var)
```

The CUDA_HOST ICMs are used for both the CUDA alloc and managed methods, where we need to replace the normal host allocation ICM. Similarly we create an ICM to decrement the reference count and free if it's 1. Finally we have ICMs for explicitly pinning and unpinning host memory. We know go into further details for each of the transfer methods.

CUDA Registered Method. For the asynchronous case with explicit pinning, we only change the transfer ICM to use cuda-Async-Memcpy. As part of the code generation, whenever we're about to print an allocation of a pinned array, we append to the allocation ICM our new REGISTER ICM; similarity, at the point of freeing we prepend the UNREGISTER ICM. We need to take special care with the host DEC_RC_FREE and REUSE ICMs, as these are generically applicable to all arrays. In the first case we extend the ICM to additionally check if the array is pinned, and if we are freeing we unpin the memory before. In the latter case this becomes more tricky — in instances such as this it might not be clear if the *new* array is meant to be pinned or not. We could check its pinned status, as this is statically set, but as previously mentioned may not be accurate. We therefore conservatively assume that the new array is meant to be pinned as well. If the reference count is 1, then its a straight assignment; if we are allocating new memory, we pin the memory after allocating. Staying with our code example from Section 4, we get the following output:

```
SAC_ND_ALLOC(a, 2014, int)
SAC_CUDA_REGISTER (a)
...
SAC_CUDA_MEM_TRANSFER (a, a_dev, 1024, int,
                       cudaMemcpyHostToDevice)
SAC_ND_REUSE (b_dev, a_dev);
...
SAC_ND_ALLOC_OR_REUSE (b, 1024, int, a)
SAC_CUDA_MEM_TRANSFER (b_dev, b, 1024, int,
                       cudaMemcpyDeviceToHost)
SAC_CUDA_DEC_RC_FREE (b_dev)
...
SAC_CUDA_UNREGISTER (a)
SAC_ND_FREE (a)
```

CUDA Alloc and Managed Methods. Both the CUDA alloc and managed methods result in the same code generation at the level of ICMs, as such we group them together here. For CUDA alloc, we replace the SAC host ALLOC and FREE ICMs with the CUDA host ICMs whenever we have an allocation of a pinned array. For the ALLOC_OR_REUSE ICM we make the same assumption as before, and propagate the pinned state. As before, the transfer ICM resolves into cuda-AsyncMemcpy. With this we generate the following code:

```
SAC_CUDA_HOST_ALLOC (a, 2014, int)
...
SAC_CUDA_MEM_TRANSFER (a, a_dev, 1024, int,
                       cudaMemcpyHostToDevice)
SAC_ND_REUSE (b_dev, a_dev);
...
SAC_ND_ALLOC_OR_REUSE (b, 1024, int, a)
SAC_CUDA_MEM_TRANSFER (b_dev, b, 1024, int,
                       cudaMemcpyDeviceToHost)
SAC_CUDA_DEC_RC (b_dev)
...
SAC_CUDA_HOST_FREE (a)
```

CUDA Managed Method. For the managed method, we make use of the CUDA host allocation and free ICMs. A major difference through from the other methods is that we do not have to create GPU buffers to communicate data from or to the host. As such, ICMs for allocating and freeing GPU device memory resolve to no-operation. We still declare the GPU device memory array, but do not allocate. We use is as part of the transfer ICM, which performs an assignment. The same holds also for array REUSE ICMs, which does a simple assignment. The allocate or reuse ICM poses a challange as we cannot be sure that the new array is part of the managed memory model or not. Similarly with the CUDA registered method, we can check the reuse candidate arrays for the pinned attribute and decide this way. We conservatively choose to use managed memory to create the new array — even if it never is referenced by a kernel it can still be referenced by other host contexts. NOT GOOD!

```
SAC_CUDA_HOST_ALLOC (a, 2014, int)
...
SAC_CUDA_MEM_TRANSFER (a, a_dev, 1024, int,
                       cudaMemcpyHostToDevice)
SAC_ND_REUSE (b_dev, a_dev);
...
SAC_ND_ALLOC_OR_REUSE (b, 1024, int, a)
```

Table 1. Details of Systems used for Experiments

System	Hardware	Software
A	4× AMD Opteron 6376	Scientific Linux 7.6
	— 64-cores @ 2.3 GHz	GCC 7.2.0
	1 TB RAM	HWLOC 1.11.8
	NVIDIA K20 (driver v. 418.87)	CUDA 10.1
B	4× AMD Ryzen 7 2700	CentOS 7.6
	— 8-cores @ 1.5 GHz to 3.5 GHz	GCC 7.4.0
	32 GB RAM	HWLOC 1.11.13
	NVIDIA RTX 2080 Ti (driver v. 418.39)	CUDA 10.1

```
SAC_CUDA_MEM_TRANSFER (b_dev, b, 1024, int,
                       cudaMemcpyDeviceToHost)
SAC_CUDA_DEC_RC (b_dev)
...
SAC_CUDA_HOST_FREE (a)
```

6 Evaluation

In Section 3 we show our results of measuring data throughput using the three memory models on two GPU devices, with synthetic workloads. In this section we present further measurements, looking at FLOPs of two benchmarks. Both perform a relaxation of a 2-dimensional array, but one does this in a fixed number of iterations, and the other does this to some convergence point (*e.g.*, epsilon).

We use these benchmarks to showcases two communication scenarios. For fixed-iteration, we only perform communication immediately before and after the loop. This is similar to our synthetic workloads example. The other performs communication within the loop, in order to perform the convergence check.

Our systems setup is shown in Table 1. We use version 1.3.3-482 of the SaC compiler on both systems. We run our experiments with both the Extended Memory Reuse (EMR) optimisation [19] on and off, we do this in order to see the overhead of the memory models themselves when it comes to memory managed. Recall that asynchronous and managed memory affect host memory in addition to device memory, to for instance pin the memory. Our measurements are taken from running each benchmark five times on each platform for each memory model. Additionally we measure sequential execution on the CPU. Runtime measurements are derived from the median value of the five runs.

6.1 Results

Our results are show in Figures 4 to 7, with the left plot showing are measurements with the EMR optimisation activated and the right plot showing them with EMR off.

Our measurements in general show that for both benchmarks we achieve peak FLOPs with the EMR optimisation on. This is unsurprising as we avoid extra overheads through extra memory operations. If we look more closely, we can

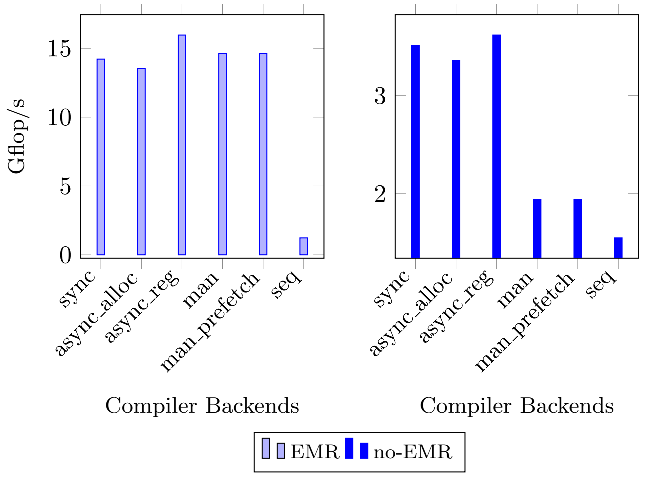


Figure 4. FLOP/s for Relaxation with Fixed Iteration on K20

see that for some memory models we achieve better FLOP then others. For instance, in left plot of Figure 4 we can see that for all the memory models we achieve about 15 Gflop/s. If we look to the right side plot this changes. For the synchronous and asynchronous models we factor 4 performance over sequential but for managed we only achieve a factor 2. This comes from the overhead of allocating further arrays within the loop, and immediately freeing these after their single reference.

In Figure 5 we can see that on the left plot the asynchronous model (using registered pinning) performs the best, and does so also in the right side plot as well. Asynchronous using host allocation suffers in both cases, especially in the right side plot where it is significantly less performant then sequential execution. Here the overheads of host allocation can be clearly seen. Similarly for managed memory, the right side plot shows that we are slightly slower then sequential execution. The communication within the loop of the relaxation adds an additional overhead, and in the case where EMR is off, also introduces further host memory operations. With asynchronous with host allocations this is deadly, and the managed memory case suffers as well. As mentioned in Section 2.2, the **`cudaHostAlloc`** function allocates physical memory immediately in order to pin it. This takes additional time and is compounded by the fact that the resulting pointer to memory is tracked by the CUDA device driver.

For the RTX 2080 Ti, we can see in Figure 6 shows the same performance pattern as in Figure 4, though with higher achieved FLOPs. For Figure 7 we see that in the left plot asynchronous with host allocations now performs better that with asynchronous with registered pinning. The results become even more divergent in the right side plot where synchronous trumps all other memory models and sequential execution. As before though, asynchronous with host allocation is the least performant together with managed

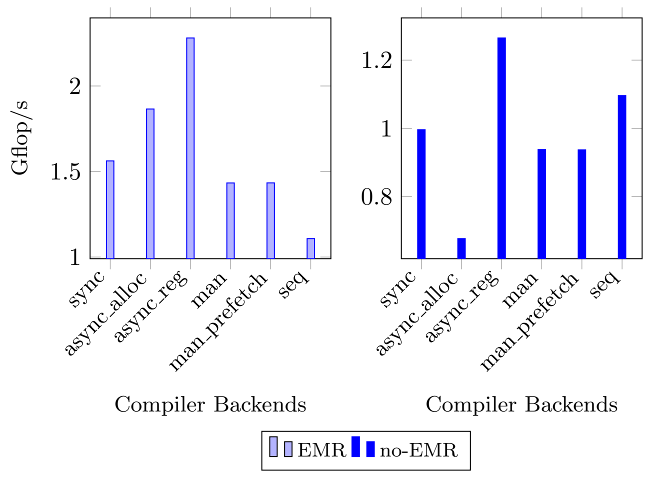


Figure 5. FLOP/s for Relaxation with Epsilon Conditional K20

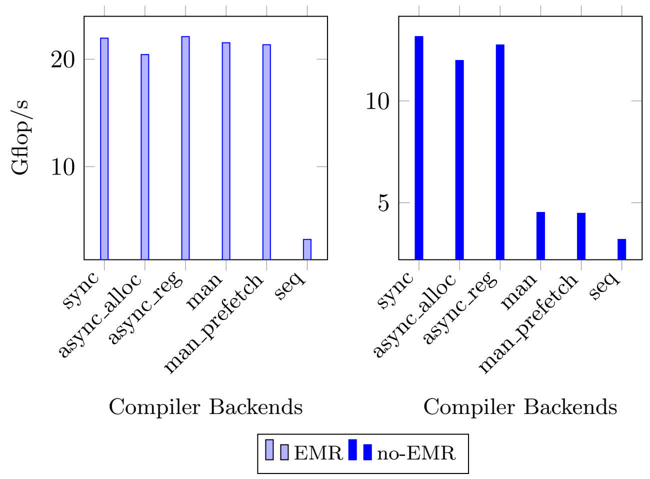


Figure 6. FLOP/s for Relaxation with Fixed Iteration on RTX 2080Ti

memory without prefetching. The demand driven transfer of the managed case is already inefficient and is made worse by the transfer within the loop. In the prefetched case, we avoid some of this overhead as the CUDA driver knows it must transfer the entire array back to the host.

A key observation is that the performance of one memory model is not the same for both systems. Given certain conditions, it is clear that one memory model is superior to another in peak performance. On the K20 system with EMR on or off, the asynchronous case with registered pinning performs best for both benchmarks. On the RTX 2080 Ti system, asynchronous with host allocation works better with EMR on, but where we don't have memory reuse, the synchronous memory model is preferred.

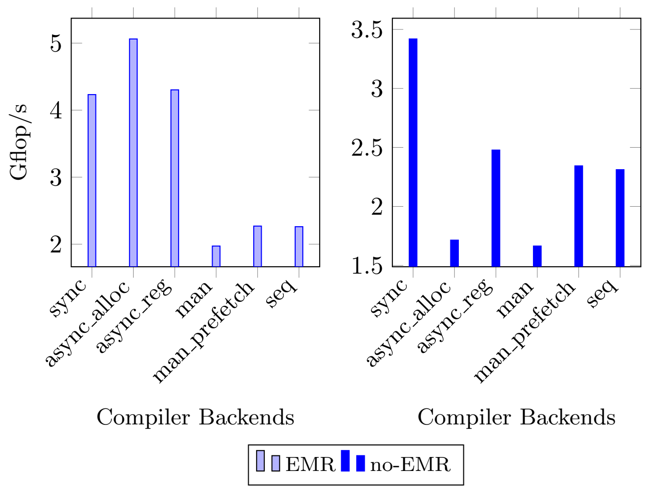


Figure 7. FLOP/s for Relaxation with Epsilon Conditional RTX 2080Ti

7 Related Work

To be done.

8 Conclusion

This paper looks into the performance potential that the different memory allocation and memory transfer options of CUDA have. Some memory transfer bandwidth investigations show that these can differ by a factor of two depending on the memory sizes being transferred, the GPU being used, and the host as well. Whether these bandwidths benefits can be translated into application performance depends on the structure of the code. In particular memory allocation frequencies but also the overall code structure can favour different memory transfer orchestrations on one and the same hardware.

We identify five different memory allocation and transfer models and show what it takes to adjust code generation from the functional high-level array language SaC into these models. Even for very simple relaxation kernels we can demonstrate that the choice between these models is non-trivial. The overall performance can easily differ by a factor of 2 between the lowest and the fastest choice. Unfortunately, different hardware setups require different choices. To make matters even more challenging, it turns out that the memory organisation introduced by the compiler can impact the overall performance very severely as well. If memory allocations are not carefully optimised away as much as possible, yet another factor of 2 in performance can be lost and the preferable choice may change from one memory model to another.

The lesson to be taken here is that a careful choice between the memory models is crucial for applications with frequent transfers if utmost performance is the goal. Whether this choice can be automated by some sophisticated performance model or requires some form of smart adaptation is left as future research.

References

[1] D.C. Cann. 1989. *Compilation Techniques for High Performance Applicative Computation.* Technical Report CS-89-108. Lawrence Livermore National Laboratory, LLNL, Livermore California.

[2] Steven M. Fitzgerald and Rodney R. Oldehoeft. 1996. Update-in-place Analysis for True Multidimensional Arrays. *Sci. Program.* 5, 2 (July 1996), 147–160. https://doi.org/10.1155/1996/493673

[3] Clemens Grelck. 2005. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 15, 3 (2005), 353–401. https://doi.org/10.1017/S0956796805005538

[4] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC: A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427. https://doi.org/10.1007/s10766-006-0018-x

[5] Clemens Grelck, Sven-Bodo Scholz, and Kai Trojahner. 2004. With-loop Scalarization: Merging Nested Array Operations. In *Implementation of Functional Languages, 15th International Workshop (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3145)*, Phil Trinder and Greg Michaelson (Eds.). Springer. https://doi.org/10.1007/978-3-540-27861-0_8

[6] Jing Guo, Robert Bernecky, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2014. Polyhedral Methods for Improving Parallel Update-in-Place. In *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Sanjay Rajopadhye and Sven Verdoolaege (Eds.). Vienna, Austria.

[7] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the Gpu Programming Barrier with the Auto-parallelising Sac Compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*. ACM Press, 15–24. https://doi.org/10.1145/1926354.1926359

[8] Tianyi David Han and Tarek S. Abdelrahman. 2009. HiCUDA: A High-Level Directive-Based Language for GPU Programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (Washington, D.C., USA) *(GPGPU-2)*. Association for Computing Machinery, New York, NY, USA, 52–61. https://doi.org/10.1145/1513895.1513902

[9] Christoph Hartmann and Ulrich Margull. 2019. GPUart - An application-based limited preemptive GPU real-time scheduler for embedded systems. *Journal of Systems Architecture* 97 (2019), 304—319. https://doi.org/10.1016/j.sysarc.2018.10.005

[10] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[11] T. Macht and C. Grelck. 2019. SAC Goes Cluster: Fully Implicit Distributed Computing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 996–1006.

[12] Nikolay Sakharnykh. 2017. Maximizing Unified Memory Performance in CUDA. https://devblogs.nvidia.com/maximizing-unified-memory-performance-cuda/. [Online; accessed 29-May-2019].

[13] Nikolay Sakharnykh. 2018. Everything You Need To Know About Unified Memory. http://on-demand.gputechconf.com/gtc/2018/presentation/s8430-everything-you-need-to-know-about-unified-memory.pdf. [Online; accessed 03-Nov-2019].

[14] NVIDIA Corporation. 2019. CUDA Toolkit Documentation v10.1.168. https://web.archive.org/web/20190523173815/https://docs.nvidia.com/cuda/archive/10.1/. [WayBack Machine; accessed 02-Nov-2019].

[15] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[16] Sven-Bodo Scholz. 1998. With-loop-folding in Sac — Condensing Consecutive Array Operations. In *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers (Lecture Notes in Computer Science, Vol. 1467)*, Chris Clack, Tony Davie, and Kevin Hammond (Eds.). Springer, 72–92. https://doi.org/10.1007/BFb0055425

[17] Sven-Bodo Scholz. 2003. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming* 13, 6 (2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[18] Steve Rennich. 2011. CUDA C/C++ Streams and Concurrency. http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf. [Online; accessed 03-Nov-2019].

[19] Hans-Nikolai Vießmann, Artjoms Šinkarovs, and Sven-Bodo Scholz. 2018. Extended Memory Reuse: An Optimisation for Reducing Memory Allocations. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages* (Lowell, MA, USA) *(IFL 2018)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3310232.3310242

Less Arbitrary waiting time

Short paper

Anonymous Author(s)

Abstract

Property testing is the cheapest and most precise way of building up a test suite for your program. Especially if the datatypes enjoy nice mathematical laws. But it is also the easiest way to make it run for an unreasonably long time. We prove connection between deeply recursive data structures, and epidemic growth rate, and show how to fix the problem, and make Arbitrary instances run in linear time with respect to assumed test size.

1 Introduction

Property testing is the cheapest and most precise way of building up a test suite for your program. Especially if the datatypes enjoy nice mathematical laws. But it is also the easiest way to make it run for an unreasonably long time. We show that connection between deeply recursive data structures, and epidemic growth rate can be easily fixed with a generic implementation. After our intervention the Arbitrary instances run in linear time with respect to assumed test size. We also provide a fully generic implementation, so error-prone coding process is removed.

2 Motivation

Typical arbitrary instance just draws a random constructor from a set, possibly biasing certain outcomes.

Generic arbitrary instance looks like this:

```
data Tree          a =
    Leaf           a
  | Branch [Tree a]
  deriving (Eq,Show,Generic.Generic)

instance Arbitrary        a
      => Arbitrary (Tree a) where
  arbitrary = oneof [Leaf   <$> arbitrary
                    ,Branch <$> arbitrary
                    ]
```

Assuming we run QuickCheck with any size parameter greater than 1, it will fail to terminate!

List instance is a wee bit better, since it tries to limit maximum list length to a constant option:

```
instance Arbitrary   a
      => Arbitrary [a] where
  lessArbitrary = sized $ \size do
    len  <- choose (1,size)
    vectorOf len lessArbitrary
```

Indeed QuickCheck manual[7], suggests an error-prone, manual method of limiting the depth of generated structure by dividing `size` by reproduction factor of the structure[1] :

```
data Tree = Leaf Int | Branch Tree Tree

instance Arbitrary Tree where
  arbitrary = sized tree'
    where tree' 0 = Leaf <$> arbitrary
      tree' n | n>0 =
        oneof [Leaf   <$> arbitrary,
           Branch <$> subtree <*> subtree]
        where subtree = tree' (n `div` 2)
```

Above example uses division of size by maximum branching factor to decrease coverage into relatively deep data structures, whereas dividing by average branching factor of ~2 will generate both deep and very large structures.

This fixes non-termination issue, but still may lead to unpredictable waiting times for nested structures. The depth of the generated structure is linearly limited by dividing the n by expected branching factor of the recursive data structure. However this does not work very well for mutually recursive data structures occuring in compilers[1], which may have 30 constructors with highly variable[2] branching factor just like GHC's HSExpr data types.

Now we have a choice of manual generation of these data structures, which certainly introduces bias in testing, or abandoning property testing for real-life-sized projects.

3 Complexity analysis

We might be tempted to compute average size of the structure. Let's use reproduction rate estimate for a single rewrite of `arbitrary` function written in conventional way.

We compute a number of recursive references for each constructor. Then we take an average number of references among all the constructors. If it is greater than 1, any **non-lazy** property test will certainly fail to terminate. If it is slightly smaller, we still can wait a long time.

What is an issue here is not just non-termination which is fixed by error-prone manual process of writing own instances that use explicit `size` parameter.

The much worse issue is unpredictability of the test runtime. Final issue is the poor coverage for mutually recursive data structure with multitude of constructors.

[1]We changed **liftM** and **liftM2** operators to <$> and <*> for clarity and consistency.
[2]Due to list parameters.

Given a *maximum size* parameter (as it is now called) to QuickCheck, would we not expect that tests terminate within linear time of this parameter? At least if our computation algorithms are linear with respect to input size?

Currently for any recursive structure like Tree a, we see some exponential function. For example $size^n$, where n is a random variable.

4 Solution

We propose to replace implementation with a simple state monad[4] that actually remembers how many constructors were generated, and thus avoid limiting the depth of generated data structures, and ignoring estimation of branching factor altogether.

```
newtype Cost = Cost Int
  deriving (Eq,Ord,Enum,Bounded,Num)

newtype CostGen                              a =
        CostGen {
          runCostGen :: State.StateT Cost QC.Gen a }
  deriving (Functor, Applicative, Monad, State.MonadFix)
```

We track the spending in the usual way:

```
spend :: Cost -> CostGen ()
spend c = CostGen $ State.modify (-c+)
```

To make generation easier, we introduce budget check operator:

```
($$$?) :: CostGen a
       -> CostGen a
       -> CostGen a
cheapVariants $$$? costlyVariants = do
  budget <- CostGen State.get
  if | budget > (0 :: Cost) -> costlyVariants
     | budget > -10000       -> cheapVariants
     | otherwise             -> error $
       "Recursive␣structure␣with␣no␣loop␣breaker."
```

In order to conveniently define our budget generators, we might want to define a class for them:

```
class LessArbitrary a where
  lessArbitrary :: CostGen a
```

Then we can use them as implementation of arbitrary that should have been always used:

```
fasterArbitrary :: LessArbitrary a => QC.Gen a
fasterArbitrary = sizedCost lessArbitrary

sizedCost :: CostGen a -> QC.Gen a
sizedCost gen = QC.sized (`withCost` gen)
```

Then we can implement Arbitrary instances simply with:

```
instance _
      => Arbitrary a where
  arbitrary = fasterArbitrary
```

Of course we still need to define LessArbitrary, but after seeing how simple was a Generic defintion Arbitrary we have a hope that our implementation will be:

```
instance LessArbitrary where
```

That is - we hope that the the generic implementation will take over.

5 Introduction to GHC generics

Generics allow us to provide default instance, by encoding any datatype into its generic Representation:

```
instance Generics (Tree a) where
  to   :: Tree a -> Rep (Tree a)
  from :: Rep (Tree a) -> Tree a
```

The secret to making a generic function is to create a set of **instance** declarations for each type family constructor.

So let's examine Representation of our working example, and see how to declare instances:

1. First we see datatype metadata D1 that shows where our type was defined:

```
type instance Rep (Tree a) =
  D1
    ('MetaData "Tree"
               "Test.Arbitrary"
               "less-arbitrary" 'False)
```

2. Then we have constructor metadata C1:

```
    (C1
       ('MetaCons "Leaf" 'PrefixI 'False)
```

3. Then we have metadata for each field selector within a constructor:

```
       (S1
          ('MetaSel
             'Nothing
             'NoSourceUnpackedness
             'NoSourceStrictness
             'DecidedLazy)
```

4. And reference to another datatype in the record field value:

```
          (Rec0 a))
```

5. Different constructors are joined by sum type operator:

```
     :+:
```

6. Second constructor has a similar representation:

```
     C1
           ('MetaCons "Branch" 'PrefixI 'False)
           (S1
              ('MetaSel
                 'Nothing
                 'NoSourceUnpackedness
                 'NoSourceStrictness
```

```
                              'DecidedLazy)
                              (Rec0 [Tree a])))
                               ignored
```

7. Note that Representation type constructors have additional parameter that is not relevant for our use case.

For simple datatypes, we are only interested in three constructors:

- :+: encode choice between constructors
- :*: encode a sequence of constructor parameters
- M1 encode metainformation about the named constructors, C1, S1 and D1 are actually shorthands for M1 C, M1 S and M1 D

There are more short cuts to consider: * U1 is the unit type (no fields) * Rec0 is another type in the field

5.1 Example of generics

This generic representation can then be matched by generic instances. Example of Arbitrary instance from [3] serves as a basic example[3]

1. First we convert the type to its generic representation:

```
genericArbitrary :: (Generic      a
                    ,Arbitrary (Rep a))
                 =>  Gen          a
genericArbitrary  = to <$> arbitrary
```

2. We take care of nullary constructors with:

```
instance Arbitrary G.U1 where
  arbitrary = pure G.U1
```

3. For all fields arguments are recursively calling Arbitrary class method:

```
instance Arbitrary c => Arbitrary (G.K1 i c) where
  gArbitrary = G.K1 <$> arbitrary
```

4. We skip metadata by the same recursive call:

```
instance Arbitrary             f
      => Arbitrary (G.M1 i c f) where
  arbitrary = G.M1 <$> arbitrary
```

5. Given that all arguments of each constructor are joined by :*:, we need to recursively delve there too:

```
instance (Arbitrary  a,
         ,Arbitrary         b)
      =>  Arbitrary (a G.:*: b) where
  arbitrary = (G.:*:) <$> arbitrary <*> arbitrary
```

6. In order to sample all constructors with the same probability we compute a number of constructor in each representation type with SumLen type family:

```
type family SumLen a :: Nat where
  SumLen (a G.:+: b) = (SumLen a) + (SumLen b)
  SumLen a           = 1
```

[3]We modified class name to simplify.

Now that we have number of constructors computed, we can draw them with equal probability:

```
instance (Arbitrary        a
         ,Arbitrary                b
         ,KnownNat (SumLen a)
         ,KnownNat (SumLen         b)
         )
      =>  Arbitrary        (a G.:+: b) where
  arbitrary = frequency
    [ (lfreq, G.L1 <$> arbitrary)
    , (rfreq, G.R1 <$> arbitrary) ]
    where
      lfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen a))
      rfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen b))
```

Excellent piece of work, but non-terminating for recursive types with average branching factor greater than 1 (and non-lazy tests, like checking **Eq** reflexivity.)

5.2 Implementing with Generics

It is apparent from our previous considerations, that we can reuse code from the existing generic implementation when the budget is positive. We just need to spend a dollar for each constructor we encounter.

For the Monoid the implementation would be trivial, since we can always use mempty and assume it is cheap:

```
genericLessArbitraryMonoid :: (Generic
a
                                ,GLessArbitrary (Rep a)
                                ,Monoid
a )
                            =>  CostGen
a
genericLessArbitraryMonoid  =
  pure mempty $$$? genericLessArbitrary
```

However we want to have fully generic implementation that chooses the cheapest constructor even though the datatype does not have monoid instance.

5.2.1 Class for budget-conscious

When the budget is low, we need to find the least costly constructor each time.

So to implement it as a type class GLessArbitrary that is implemented for parts of the Generic Representation type, we will implement two methods:

1. gLessArbitrary is used for normal random data generation
2. cheapest is used when we run out of budget

```
class GLessArbitrary datatype where
  gLessArbitrary :: CostGen (datatype p)
  cheapest       :: CostGen (datatype p)


genericLessArbitrary :: (Generic             a
                              ,GLessArbitrary (Rep a))
                           =>  CostGen            a
genericLessArbitrary = G.to <$> gLessArbitrary
```

5.2.2 Helpful type family

First we need to compute minimum cost of the in each branch of the type representation. Instead of calling it *minimum cost*, we call this function Cheapness.

For this we need to implement minimum function at the type level:

```
type family Min m n where
  Min m n = ChooseSmaller (CmpNat m n) m n

type family ChooseSmaller (o::Ordering)
                          (m::Nat)
                          (n::Nat) where
  ChooseSmaller 'LT m n = m
  ChooseSmaller 'EQ m n = m
  ChooseSmaller 'GT m n = n
```

so we can choose the cheapest^[We could add instances for :

```
type family Cheapness a :: Nat where
  Cheapness (a :*: b)  =
         Cheapness a + Cheapness b
  Cheapness (a :+: b)  =
    Min (Cheapness a) (Cheapness b)
  Cheapness  U1                    = 0
  <<flat-types>>
  Cheapness (K1 a other) = 1
  Cheapness (C1 a other) = 1
```

Since we are only interested in recursive types that can potentially blow out our budget, we can also add cases for flat types since they seem the cheapest:

```
Cheapness (S1 a (Rec0 Int        )) = 0
Cheapness (S1 a (Rec0 Scientific )) = 0
Cheapness (S1 a (Rec0 Double     )) = 0
Cheapness (S1 a (Rec0 Bool       )) = 0
Cheapness (S1 a (Rec0 Text.Text  )) = 1
Cheapness (S1 a (Rec0 other      )) = 1
```

5.2.3 Base case for each datatype

For each datatype, we first write a skeleton code that first spends a coin, and then checks whether we have enough funds to go on expensive path, or we are beyond our allocation and need to generate from among the cheapest possible options.

```
instance GLessArbitrary        f
      => GLessArbitrary (D1 m f) where
  gLessArbitrary = do
    spend 1
    M1 <$> (cheapest $$$? gLessArbitrary)
  cheapest = M1 <$> cheapest
```

5.2.4 Skipping over other metadata

First we safely ignore metadata by writing an instance:

```
instance GLessArbitrary           f
      => GLessArbitrary (G.C1 c f) where
  gLessArbitrary = G.M1 <$> gLessArbitrary
  cheapest       = G.M1 <$> cheapest

instance GLessArbitrary           f
      => GLessArbitrary (G.S1 c f) where
  gLessArbitrary = G.M1 <$> gLessArbitrary
  cheapest       = G.M1 <$> cheapest
```

5.2.5 Counting constructors

In order to give equal draw chance for each constructor, we need to count number of constructors in each branch of sum type :+: so we can generate each constructor with the same frequency:

```
type family SumLen a :: Nat where
  SumLen (a G.:+: b) = SumLen a + SumLen b
  SumLen  a          = 1
```

5.2.6 Base cases for GLessArbitrary

Now we are ready to define the instances of GLessArbitrary class.

We start with base cases GLessArbitrary for types with the same representation as unit type has only one result:

```
instance GLessArbitrary G.U1 where
  gLessArbitrary = pure G.U1
  cheapest       = pure G.U1
```

For the product of, we descend down the product of to reach each field, and then assemble the result:

```
instance (GLessArbitrary  a
         ,GLessArbitrary          b)
      =>  GLessArbitrary (a G.:*: b) where
  gLessArbitrary = (G.:*:) <$> gLessArbitrary
                           <*> gLessArbitrary
  cheapest       = (G.:*:) <$> cheapest
                           <*> cheapest
```

We recursively call instances of LessArbitrary for the types of fields:

```
instance  LessArbitrary          c
      => GLessArbitrary (G.K1 i c) where
  gLessArbitrary = G.K1 <$> lessArbitrary
  cheapest       = G.K1 <$> lessArbitrary
```

5.2.7 Selecting the constructor

We use code for selecting the constructor that is taken after[3].

```
instance (GLessArbitrary       a
         ,GLessArbitrary                  b
         ,KnownNat (SumLen     a)
         ,KnownNat (SumLen                b)
         ,KnownNat (Cheapness a)
```

```
        ,KnownNat (Cheapness                b)
        )
      => GLessArbitrary       (a Generic .:+: b) where
  gLessArbitrary =
    frequency
      [ (lfreq , L1 <$> gLessArbitrary)
      , (rfreq , R1 <$> gLessArbitrary) ]
    where
      lfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen a))
      rfreq = fromIntegral
            $ natVal (Proxy :: Proxy (SumLen b))
  cheapest =
      if lcheap <= rcheap
         then L1 <$> cheapest
         else R1 <$> cheapest
    where
      lcheap, rcheap :: Int
      lcheap = fromIntegral
             $ natVal (Proxy :: Proxy (Cheapness a))
      rcheap = fromIntegral
             $ natVal (Proxy :: Proxy (Cheapness b))
```

6 Conclusion

We show how to quickly define terminating test generators using generic programming. This method may be transferred to other generic programming regimes like Featherweight Go or Featherweight Java.

We recommend it to reduce time spent on making test generators.

7 Bibliography

[1] Day, L.E. and Hutton, G. 2013. Compilation à la Carte. *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages* (Nijmegen, The Netherlands, 2013).
[2] EnTangleD: A bi-directional literate programming tool: 2019. *https://blog.esciencecenter.nl/entangled-1744448f4b9f*.
[3] generic-arbitrary: Generic implementation for QuickCheck's Arbitrary: 2017. *https://hackage.haskell.org/package/generic-arbitrary-0.1.0/docs/src/Test-QuickCheck-Arbitrary-Generic.html#genericArbitrary*.
[4] Jones, M.P. and Duponcheel, L. 1993. *Composing monads*.
[5] Knuth, D.E. 1984. Literate programming. *Comput. J.* 27, 2 (May 1984), 97–111. DOI:https://doi.org/10.1093/comjnl/27.2.97.
[6] Pandoc: A universal document converter: 2000. *https://pandoc.org*.
[7] QuickCheck: An Automatic Testing Tool for Haskell: *http://www.cse.chalmers.se/~rjmh/QuickCheck/manual_body.html#16*.
[8] stack 0.1 released:.

Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing

Sólrún Halla Einarsdóttir
Nicholas Smallbone
slrn@chalmers.se
nicsma@chalmers.se
Chalmers University of Technology
Gothenburg, Sweden

ABSTRACT

We present a template-based extension of the Theory Exploration tool QuickSpec. QuickSpec uses testing to automatically discover equational properties about functions in a Haskell program. These properties can help the user understand the program or be used as a source of possible lemmas in proofs of the program's correctness.

In our extension, the user supplies templates, which describe families of laws such as associativity and distributivity, and we only consider properties that match the templates. This restriction limits the search space and ensures that only relevant properties are discovered. In this way, we sacrifice broad search for more direction towards desirable property patterns, which makes theory exploration tractable and scalable. We demonstrate theory exploration using our tool and compare it to the QuickSpec tool.

KEYWORDS

Theory exploration, QuickSpec, Functional programming, Algebraic properties, Program understanding, Property-based testing

ACM Reference Format:
Sólrún Halla Einarsdóttir and Nicholas Smallbone. SAMPLE. Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/SAMPLE

1 INTRODUCTION

One strength of functional programming is that programs are easy to reason about. Pure functions often obey simple formal specifications which, as long as the programmer writes them down, are a great help in programming. A formal specification can be proved correct, automatically tested with a tool such as QuickCheck [4] or SmallCheck [15], or simply read in order to understand a codebase.

Many functional programmers already specify their code, by writing e.g. QuickCheck properties, but many do not. Can those who do *not* specify their code also reap the benefits of formal specification? The answer is *yes*: given a piece of code, we can automatically infer properties about it.

A tool that infers properties from code is called a *theory exploration system.* Two theory exploration systems for Haskell are QuickSpec [16] and Speculate [1]. These tools take as input a collection of Haskell functions and, through testing, discover formal properties which can be expressed using those functions. For example, given the list functions `++`, `reverse`, and `map`, QuickSpec discovers a total of five laws, all of them well-known and useful:

```
reverse (reverse xs) = xs
map f (reverse xs) = reverse (map f xs)
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
reverse xs ++ reverse ys = reverse (ys ++ xs)
map f xs ++ map f ys = map f (xs ++ ys)
```

Both tools work in a similar way. Very roughly, they (1) consider *all* possible properties, up to some size limit, which can be built from the given functions (and some variables), (2) test which of those properties are true, (3) remove any redundant properties (a true property is redundant if it can be derived from other true properties), and (4) report all the non-redundant true properties. Because they explore all possible properties, the generated specification is *complete* (up to the size limit).

This approach works well on small sets of functions. Completeness means that we get an expressive specification, and discarding redundant properties keeps the specification short. When given only a few functions, QuickSpec and Speculate typically produce clear, crisp and useful specifications, like the one above. We have found that reading the output of QuickSpec is a great help in understanding an unfamiliar API.

Unfortunately, this approach breaks down when exploring large APIs: a *complete* theory exploration system simply finds too many laws. In a benchmark running QuickSpec on about 30 list functions [16], over 500 laws were found! The QuickSpec user is unlikely to bother reading all these laws. Many of them are unenlightening, for example:

```
map (f x) (take (succ 0) xs) = zipWith f (scanl g x []) xs
```

This law is found, not because it was interesting, but because it was true and because QuickSpec did not consider it to be redundant. When we explore large APIs, we often get huge numbers of uninteresting laws. Furthermore, the search space is huge so the tools often take a while to run: exploring the 30 list functions took about two hours. These problems arise because QuickSpec and Speculate are complete.

Conference'17, July 2017, Washington, DC, USA
© SAMPLE Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/SAMPLE

1.1 RoughSpec

We have developed a new theory exploration system, RoughSpec. Like QuickSpec and Speculate, it takes as input a set of Haskell functions (which we call the *signature*), and uses testing to find properties that seem to hold. The difference is that RoughSpec is *incomplete*: it does not try to find all true properties.

Instead, the user gives a set of *templates*, expressions which describe a family of laws such as associativity or distributivity. RoughSpec searches only for instances of these templates. In this way, the user can specify what kind of properties they would find interesting, and RoughSpec searches only for these properties.

A template is a Haskell equation containing functions, variables and *metavariables*. For example, here is a template which represents commutativity (note that in our syntax, variables are written in uppercase, and a metavariable is written as a variable with a leading question mark):

```
?F X Y = ?F Y X
```

When a template contains a metavariables, RoughSpec *instantiates* that metavariable with functions drawn from the signature, and reports any instances that make the equation hold. In this case, RoughSpec will search for functions `?F` such that `?F X Y = ?F Y X` for all `X` and `Y`—that is, for commutative functions.

Here are some more examples of templates. They describe: (1) associativity, (2) an invertible function; (3) distributivity; (4) and (5) a function having an identity element:

```
(1) ?F (?F X Y) Z = ?F X (?F Y Z)
(2) ?F (?G X) = X
(3) ?F (?G X) (?G Y) = ?G (?F X Y)
(4) ?F X ?E = X
(5) ?F ?E X = X
```

In (4) and (5), `?E` will be replaced by *constants* drawn from the signature.

When we run RoughSpec on a signature of five list functions `++`, `reverse`, `map`, `sort` and `nub`, using the templates (1)–(3) above as well as commutativity, we get the following output:

```
Searching for commutativity properties...
  1. sort (xs ++ ys) = sort (ys ++ xs)
Searching for associativity properties...
  2. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
  3. sort (sort (xs ++ ys) ++ zs) =
     sort (xs ++ sort (ys ++ zs))
  4. nub (nub (xs ++ ys) ++ zs) =
     nub (xs ++ nub (ys ++ zs))
Searching for inverse function properties...
  5. reverse (reverse xs) = xs
Searching for distributivity properties...
  6. map f xs ++ map f ys = map f (xs ++ ys)
  7. sort (sort xs ++ sort ys) = sort (xs ++ ys)
  8. nub (nub xs ++ nub ys) = nub (xs ++ ys)
```

Each property is tagged with the name of the template that generated it. For example, the first law is an instance of commutativity, `?F X Y = ?F Y X`, with `?F = \xs ys -> sort (xs ++ ys)`. (Section 2 describes how RoughSpec chooses how metavariables are instantiated.) We see that `++` is associative, that `reverse` is its own inverse, that `map` distributes over `++`, and that appending two lists and then `sort`ing or `nub`bing the result is a well-behaved operation in its own right.

By adding more templates, we can find more laws. For example, adding the template `?F (?G X) = ?G (?F X)` produces the law `map f (reverse xs) = reverse (map f xs)`. We have not found all of the important list laws (for example, the law `reverse (xs++ys) = reverse ys ++ reverse xs`), but have produced a useful and short subset.

The templates we have used so far represent well-known properties and apply to a wide range of APIs. The goal of RoughSpec is that the user can start with a "standard" set of templates, and find an incomplete, but useful set of properties for their program. Then they can find more detailed properties by adding templates that are tailored to their domain. By putting the user in charge of choosing templates, we aim to keep the output small and easy to understand.

In the next sections, we describe how RoughSpec works, and then show it in action on some larger examples.

2 HOW IT WORKS

To use RoughSpec, the user inputs the templates they are interested in, along with the functions they want to explore, in a *signature* [16]. See an example of a simple signature in Figure 1. As described in Section 1, the templates are expressed in a simple term language containing metavariables representing holes to be filled with a function symbol (written as a question mark followed by a string label), variables (written as names starting with a capital letter) and the function symbols occurring in the signature. In our current implementation, functions are written uncurried. For example the template `?F(?G(X,Y)) = ?F(?G(Y,X))` describes the nested composition of two functions (?*F* and ?*G*) being commutative in two variables.

```
simpleSig = [
  con "reverse" (reverse :: [A] -> [A]),
  con "++" ((++) :: [A] -> [A] -> [A]),
  con "length" (length :: [A] -> Int),
  template "nest-commute" "?F(?G(X,Y))=?F(?G(Y,X))"
  ]
```

Figure 1: A signature containing some list functions and a template for nest-commutative properties.

Candidate properties are generated by attempting to fill the holes in a template using the function symbols in scope of the exploration, making sure the generated equations are well typed. For example, filling the holes in the template above using functions `length`, `reverse`, and `++` on lists gives the candidate properties `length (xs ++ ys) = length (ys ++ xs)` (*cp*1) and `reverse (xs ++ ys) = reverse (ys ++ xs)` (*cp*2).

The generated candidate properties are then tested using QuickCheck [4]. If no counterexamples are found the property is presented to the user as a law. In our example, *cp*1 passes this phase and is presented to the user, while *cp*2 fails and is discarded.

2.1 Expanding templates

Note that in the algorithm described above, each hole in a template can be filled only with precisely one of the function symbols in scope. This is rather limiting and requires us to use multiple different templates to discover properties that we might intuitively want to place in the same category, as we shall see in the examples below.

We have implemented some automated "expansion" of user input templates in an attempt to make the results of exploration more general, and to help the user avoid the tedious work of typing up a set of nearly-identical templates.

2.1.1 Nested functions. Consider the property
`length (xs ++ ys) = length (ys ++ xs)` (*cp*1)
discovered in our example above. We discovered this property using a template that specifically described the composition of two function symbols being commutative. Suppose we had a more general template for commutativity, i.e. `?F X Y = ?F Y X`. What if we want such a template to cover properties like *cp*1, rather than having to type up more than one commutativity template?

In order to do this we have implemented an extension allowing a hole to be filled by a nested composition of two function symbols. We replace a given hole in our template with two holes representing an outer function applied to an inner function which is in turn applied to the original hole's arguments. That is, a hole of the form `?F e1...en` turns into `?G (?F e1...en)`. This allows us to discover the property *cp*1 using the commutativity template `?F X Y = ?F Y X`. It also allows us to use a general template for identity functions, `?F X = X`, to discover the property `reverse (reverse xs) = xs`.

2.1.2 Partial application. Suppose we extend our example signature from Figure 1 by adding the function `map` and a distributivity template
`?F (?G X Y) = ?G (?F X) (?F Y)` (*d*1),
describing a function `?F` distributing over a two-argument function `?G`.

We would like to discover the property
`map f (xs ++ ys) = map f xs ++ map f ys` (*dmap*),
describing how `map` distributes over `++`. However, since our template holes can only be filled using precisely one function symbol or two nested function symbols, this template does not cover the desired property. Instead we would need a more complex template like
`?F X (?G Y Z) = ?G (?F X Y) (?F X Z)`,
with an extra variable `X` for the function argument to map.

In order to avoid needing a variety of complicated templates when our signatures contain functions with varying numbers of arguments, we allow a template hole to be filled with a partially applied function. We replace a given hole in our template with a hole applied to a number of fresh variables, limited by the maximum arity of the functions in scope. By doing so our desired property *dmap* is now covered by the template *d*1.

In combination with our nested function expansion described above, this also allows us to discover properties such as

```
map f (concat (xss ++ yss)) =
map f (concat xss) ++ map f (concat yss)
```

using the same template *d*1 and adding the `concat` function to our signature.

This method considers all possible partially-applied functions when filling a hole. In practice we found this to give rise to some rather confusing properties when binary operators were involved. For instance, suppose we extend our example signature with a template `?F (?G X) = ?F X` meant to discover pairs of functions `?F` and `?G` where the result of `?F` is preserved when we apply `?G` to its argument. This gives rise to properties such as
`length (reverse xs) = length xs` and
`length (map f xs) = length xs`.

We also discover properties such as
`length (xs ++ reverse ys) = length (xs ++ ys)`,
where the hole `?F` has been filled by the function `length . (xs ++)`.

We find properties about partially applied functions such as `xs ++` rather confusing and uninteresting., and therefore decided to limit this expansion such that if a function is a binary operator (that is to say the function has two arguments and those arguments have the same type) we do not allow it to fill a hole.

2.1.3 Limiting expansion. Expanding templates automatically is a delicate balance. In moderation, it produces interesting properties that users want to see, and that intuitively match the given template. If we expand templates too much, we may generate irrelevant properties, overwhelm the user with output or increase the running time of our tool. As can be seen from the special treatment of binary operators in 2.1.2, we have implemented some ad hoc limitations to our expansions to prevent them from producing properties we found less interesting. Perhaps the appropriate expansions and when to use them most effectively is dependent on the context, what kinds of functions are being explored and the user's priorities. In order to make this expansion tractable we want to make the language for inputting functions and templates in the signature more expressive, for example, allowing the user to describe which functions they want to be partially applied and in how many arguments.

2.2 Pruning

Suppose we now run RoughSpec on our example signature containing some list functions and the templates for identity and preservation mentioned in 2.1 (see Figure 2).

```
simpleSig = [
  con "reverse" (reverse :: [A] -> [A]),
  con "++" ((++) :: [A] -> [A] -> [A]),
  con "length" (length :: [A] -> Int),
  con "map" (map :: (A -> B) -> [A] -> [B]),
  template "id" "?F(X)=X",
  template "preserve" "?F(?G(X))=?F(X)"
  ]
```

Figure 2: Our updated example signature.

We are presented with the following output:

```
== Laws ==
Searching for id properties...
  1. reverse (reverse xs) = xs
```

```
Searching for preserve properties...
  2. length (reverse xs) = length xs
  3. length (map f xs) = length xs
 4. length (reverse (reverse xs)) = length (reverse xs)
  5. length (reverse (map f xs)) = length (reverse xs)
  6. length (map f (reverse xs)) = length (map f xs)
  7. length (map f (map g xs)) = length (map f xs)
  8. reverse (reverse (reverse xs)) = reverse xs
  9. (++) (reverse (reverse xs)) = (++) xs
 10. length (reverse (reverse xs)) = length xs
 11. length (reverse (map f xs)) = length xs
 12. length (map f (reverse xs)) = length xs
 13. length (map f (map g xs)) = length xs
 14. map f (reverse (reverse xs)) = map f xs
```

Some of these properties appear to be redundant. For instance, property 4 is an instance of property 2, with `xs` replaced by `reverse xs`. Surely our user isn't interested in seeing a property that's just a more specific instance of a previously discovered property?

To solve this, RoughSpec includes a *pruning* phase, which discards any discovered properties that are instances of previous properties. In this case, properties 4 and 8 will be pruned away, as they are instances of properties 2 and 1, respectively. We also remove any properties that can be found by applying the same function to both sides of a previous property. For example, property 10 is equivalent to applying the `length` function to both sides of property 1. In our example, we will discard properties 4, 8, 9, 10, and 14, and will discover 9 properties in total.

This still leaves us with some rather redundant properties. Notice that property 5 above is a consequence of properties 2 and 3, and can be proved by rewriting using 2 and 3. If we were also to prune away properties that can be proved via rewriting using previous properties, we would be left with only three properties, namely properties 1, 2 and 3 above.

Through pruning we both avoid cluttering the output with redundant properties and avoid spending time testing such redundant properties. However, as our user has presumably input templates describing the exact shapes of properties they are interested in seeing output, we want to be careful not to go too far in pruning away properties matching those desired patterns. We therefore only use the pruning by rewriting in the case of properties that were found by expanding a given template and not properties that precisely match one of the input templates. For instance, property 11 is pruned away as it can be proved by rewriting and was generated from an expanded template. However, if we added the template `?F (?G (?H F X)) = ?F X` to our signature in Figure 2, we would no longer prune away property 11 as it would precisely match an input template, and can only be pruned by rewriting.

As properties discovered earlier are used to prune away ones that are discovered later, the order in which the templates are input makes a difference to which properties we output. To optimize pruning it seems good to start with smaller and/or more general templates and move on to larger and/or more specific ones, as smaller properties are more likely to be applicable to pruning larger ones, but our user can also toggle this and make sure to put the templates they find most relevant first.

3 CASE STUDIES

The following examples demonstrate theory exploration using our template-based approach and discuss what kinds of templates we've found to be useful. We compare our results to theory exploration with QuickSpec on the same sets of functions. The code is available at `https://github.com/solrun/quickspec`, in the `template-examples` directory.

3.1 Pretty Printing

This case study shows how RoughSpec can be useful in understand an unfamiliar library. Suppose we are using Hughes's pretty-printing library [9] for the first time. We are presented with an intimidating array of combinators:

```
empty :: Doc
text :: String -> Doc
nest :: Int -> Doc -> Doc
(<>) :: Doc -> Doc -> Doc
(<+>) :: Doc -> Doc -> Doc
($$) :: Doc -> Doc -> Doc
hcat :: [Doc] -> Doc
hsep :: [Doc] -> Doc
vcat :: [Doc] -> Doc
sep :: [Doc] -> Doc
fsep :: [Doc] -> Doc
```

The library documentation explains that `Doc` represents a pretty-printed document, `empty` is an empty document, `text` prints a string verbatim, and `nest` indents an entire document by a given number of spaces. The remaining functions combine multiple documents into one:

- <>, <+> and $$ typeset two documents beside one another, beside one another with a space in between, or one above the other, respectively.
- `hcat`, `hsep` and `vcat` are variants of <>, <+> and $$ that take a *list* of documents.
- `sep` and `fsep` choose whichever of <+> and $$ gives the prettiest output.

We may now feel happy going off and writing some pretty printers. But there are still questions unanswered:

- What is the difference between `empty` and `text ""`?
- If I am indenting a multi-line document, should I apply `nest` to each line individually or to the whole document?
- Does it matter if I use <> or `hcat`, <+> or `hsep`, $$ or `vcat`?
- Why is there no analogue of <> for `sep` and `fsep`?

These are the kinds of questions a formal specification of the pretty-printing library would answer. Let us see if RoughSpec can help us.

We start with the same list of ten templates as in 3.3. We reproduce RoughSpec's output verbatim. It finds the following 41 laws:

```
Searching for identity properties...
  1. hcat (unit x) = x
  2. hsep (unit x) = x
  3. vcat (unit x) = x
  4. sep (unit x) = x
  5. fsep (unit x) = x
```

```
Searching for fixpoint properties...
  6. nest x empty = empty
Searching for cancel properties...
  7. length (unit (nest x y)) = length (unit y)
Searching for left-id-elem properties...
  8. nest 0 x = x
  9. empty <> x = x
 10. empty $$ x = x
 11. empty <+> x = x
 12. hcat [] <> x = x
 13. hsep [] <> x = x
 14. vcat [] <> x = x
 15. sep [] <> x = x
 16. fsep [] <> x = x
 17. hcat [] $$ x = x
 18. hsep [] $$ x = x
 19. vcat [] $$ x = x
 20. sep [] $$ x = x
 21. fsep [] $$ x = x
 22. hcat [] <+> x = x
 23. hsep [] <+> x = x
 24. vcat [] <+> x = x
 25. sep [] <+> x = x
 26. fsep [] <+> x = x
Searching for right-id-elem properties...
 27. x <> empty = x
 28. x $$ empty = x
 29. x <+> empty = x
 30. x <> text [] = x
Searching for commutative properties...
Searching for commuting-functions properties...
 31. nest x (nest y z) = nest y (nest x z)
Searching for distributivity properties...
 32. nest x (y <> z) = nest x y <> nest x z
 33. nest x (y $$ z) = nest x y $$ nest x z
 34. nest x (y <+> z) = nest x y <+> nest x z
Searching for analogy-distributivity properties...
 35. text xs <> text ys = text (xs ++ ys)
 36. hcat xs <> hcat ys = hcat (xs ++ ys)
 37. vcat xs $$ vcat ys = vcat (xs ++ ys)
 38. hsep xs <+> hsep ys = hsep (xs ++ ys)
Searching for associative properties...
 39. (x <> y) <> z = x <> (y <> z)
 40. (x $$ y) $$ z = x $$ (y $$ z)
 41. (x <+> y) <+> z = x <+> (y <+> z)
```

Laws 12–26 are curious. They are all rather similar, and do not look very interesting. In fact, each of these laws contains a term (such as `hsep []` or `vcat []`) which is actually equal to `empty`. Once we know that, we see that these laws are trivial restatements of laws 9–11. The problem is that there was no template which allowed RoughSpec to discover laws such as `hsep [] = empty`.

To fix this, we add the template `?F ?X = ?Y`. This template finds 10 laws, including `hsep [] = empty` and its companions, and now laws 12–26 are pruned away as they follow from laws 9–11. We are left with a total of 26 laws: 1–11 and 27–41 above.

Together, these laws answer most of the questions we posed above. The difference between `empty` and `text ""` is that `empty` acts as an identity for the other operators:

```
empty <>  x = x     x <>  empty = x
empty <+> x = x     x <+> empty = x
empty $$  x = x     x $$  empty = x
```

On the other hand, `text ""` mostly does not, only satisfying one identity law:

```
x <> text "" = x
```

Of course, we could use QuickCheck (or indeed read Hughes [9]) to find out just *why* `text ""` is not an identity element.

As for whether one should indent each line separately or the whole document at once, it doesn't matter, because `nest` distributes over `$$`:

```
nest x (y $$ z) = nest x y $$ nest x z
```

Another distributivity law tells us that we can freely choose to typeset a long string in one go, or split it up into smaller pieces:

```
text xs <> text ys = text (xs ++ ys)
```

The `<>`, `<+>` and `$$` operators are associative:

```
(x <> y) <> z   = x <> (y <> z)
(x <+> y) <+> z = x <+> (y <+> z)
(x $$ y) $$ z   = x $$ (y $$ z)
```

and `hcat`, `vcat` and `hsep` appear to be those operators folded over a list:

```
hcat xs <> hcat ys = hcat (xs ++ ys)
vcat xs $$ vcat ys = vcat (xs ++ ys)
hsep xs <+> hsep ys = hsep (xs ++ ys)
```

Therefore, it doesn't matter whether one uses e.g. `<>` or `hcat`—they are equivalent.

Associativity of course means that we can write e.g. `x <> y <> z` without worrying about bracketing. We might wonder whether the same applies to sequences of mixed operators, e.g. `x <> y <+> z`. To find out we can add another template:

```
mixed-associativity: ?G (?F X Y) Z = ?F X (?G Y Z)
  -- in infix notation: (X `?F` Y) `?G` Z = X `?F` (Y `?G` Z)
```

This reveals that, indeed, a whole host of expressions can be freely rebracketed:

```
nest x y <> z = nest x (y <> z)
(x $$ y) <> z = x $$ (y <> z)
(x <+> y) <> z = x <+> (y <> z)
nest x y <+> z = nest x (y <+> z)
(x <> y) <+> z = x <> (y <+> z)
(x $$ y) <+> z = x $$ (y <+> z)
```

Finally, we come to the question of why there is no two-argument version of `sep` and `fsep`. Given what we learnt above, we might suspect that these operators are not associative. To test this, we can add two new functions to the signature:

```
sep2, fsep2 :: Doc -> Doc -> Doc
sep2 x y = sep [x, y]
fsep2 x y = fsep [x, y]
```

Indeed, no new associativity law appears.[1] Nor is it the case that e.g. `fsep2 (fsep xs) (fsep ys) = fsep (xs ++ ys)`. In fact, no interesting laws of any kind appear.

The laws that `hsep` and family satisfy are very useful when programming. When we want to typeset a list of documents horizontally, we can either use `hsep`, `<+>` or a mixture (e.g. we may write `hsep xs <+> hsep ys` instead of `hsep (xs++ys)`. By contrast, when using `sep` or `fsep`, we must carefully collect all documents into a list and only then combine them. In this case, the *lack of a nice specification* is itself useful information: it warns us that we should take care when using these combinators!

Summary. RoughSpec performed well on the pretty-printing library. It produced a manageable number of equations, all of them simple and easily understood. Despite their simplicity, they answered important questions about how to use the library—the questions listed at the top of this section. We believe that even simple properties, such as associativity and distributivity laws, are a great help in understanding how to use a new library. Finally, we got good results from a "standard" set of templates and were able to improve the output by adding our own.

The one hiccup in RoughSpec's performance was laws 12–26. We were forced to add a template specifically to prune away these laws. In fact, another instance of the same problem occurred: `sep` and `fsep` only differ on lists of at least three elements, which means that `sep2 = fsep2`. QuickSpec discovers this law instantly, but RoughSpec failed to find it as there was no template of the form `?X = ?Y`. Instead, laws about this function appear twice—once with `sep2` and once with `fsep2`.

In both cases, we have two laws containing syntactically different terms *that are actually equal*—for example, `hcat []` and `hsep []`. RoughSpec ought to detect that the terms are equal, and avoid generating duplicate laws. One option is to gather all the terms used to instantiate metavariables, divide them into equivalence classes by testing, and keep only the representative of each equivalence class.

Comparison with QuickSpec. As reported in [16], QuickSpec does well given the combinators `text`, `nest`, `<>`, `<+>` and `$$`, finding a complete specification that matches the one given by Hughes [9]. Unfortunately, when we add `hcat` and friends, QuickSpec finds many complicated, unimportant-looking laws, for example:

```
40. fsep (xs ++ [empty] ++ ys) = fsep (xs ++ ys)
41. hcat (xs ++ [empty] ++ ys) = hcat (xs ++ ys)
42. hsep (xs ++ [empty] ++ ys) = hsep (xs ++ ys)
43. hcat (xs ++ [hcat ys] ++ zs) = hcat (xs ++ ys ++ zs)
44. hsep (xs ++ [hsep ys] ++ zs) = hsep (xs ++ ys ++ zs)
45. fsep (xs ++ [x $$ (y $$ z)]) = fsep xs $$ (x $$ (y $$ z))
46. fsep (xs ++ [x $$ x] ++ ys) = fsep xs $$ ((x $$ x) $$ fsep ys)
```

3.2 Model-based properties

In [10], Hughes compares different methods of defining properties for QuickCheck testing, and finds that *model-based* testing is the most effective of the five methods he compares, revealing all the bugs in the test programs with a small number of properties to test.

[1]Exercise to the reader: reading the documentation of the `pretty` library, it seems reasonable that `fsep2` could be associative. Why is it not?

Model-based testing is based on the approach to proving the correctness of data representations introduced by Hoare in [8]. The data representation is related to an appropriate abstract representation using an *abstraction function.* For each operation both a concrete and an abstract implementation are defined and the following diagram is proven to commute:

$$\begin{tikzcd}
X \arrow[r, dashed, "\text{abstraction}"'] \arrow[d, "op_{concrete}"'] & X_{abstract} \arrow[d, "op_{abstract}"] \\
A_t \arrow[r, "\text{abstraction}"'] & A
\end{tikzcd}$$

We can then obtain correctness proofs for the data representation and operations in question based on (presumably simpler) correctness proofs for the abstract data and operations.

In model-based testing we define an abstract model of the data structure being tested and define test properties relating the concrete operations under test to the corresponding abstract ones using an abstraction function. In [10], bugs in the implementation of concrete operations are found to cause counterexamples to such properties.

Since we can include specific function symbols from the exploration scope in our templates, we can use RoughSpec to search only for properties that relate two operations via a given abstraction function, with a template along the lines of
`?F (abstraction X) = abstraction (?G X)`.

3.2.1 Binary trees. In [10], Hughes uses binary trees as an example and defines five model-based properties relating the tree operations to operations on a list of key-value pairs with *toList* as an abstraction function.

```
1. find x t = findList x (toList t)
2. insertList x (toList t) = toList (insert x t)
3. deleteKeyList x (toList t) = toList (delete x t)
4. toList nil = []
5. toList (union t t1) =
   sort (unionList (toList t) (toList t1))
```

Running RoughSpec on a signature containing the relevant functions and three templates describing model-based properties, we discover precisely these five properties in just under 0.3 seconds.

```
?F(Y,toList(X)) = ?G(Y,X)
toList(?X) = ?Y
toList(?H(X,Y)) = ?F(toList(X),toList(Y))
```

Due to the different shapes of the desired properties we need three different templates to discover them all. With a more expressive term language for our signatures, as discussed in 2.1, we may get away with using fewer such templates.

Comparison with QuickSpec. QuickSpec discovers 28 properties about the functions in our signature, among them the five model-based properties. This takes between 10 and 11 seconds, significantly longer than RoughSpec.

3.3 A large library of list functions

Section 4.2 in [16] describes a stress-test where QuickSpec was used to find properties about a set of 33 Haskell functions on lists. This took standard QuickSpec 42 minutes and resulted in 398 properties when limited to terms of size 7 or less, and hit a time limit of 2 hours when the size was increased to 8. As described in the Introduction, many of the laws found by QuickSpec were not interesting. This illustrates how running QuickSpec on larger theories scales poorly with regard to run-time and may produce an overwhelming amount of output. When we ran the most recent version of QuickSpec on this set of functions it ran out of memory and did not manage to produce any properties.

```
length    :: [A] -> Int
sort      :: [Int] -> [Int]
scanr     :: (A -> B -> B) -> B -> [A] -> [B]
(>>=)     :: [A] -> (A -> [B]) -> [B]
reverse   :: [A] -> [A]
(>=>)     :: (A -> [B]) -> (B -> [C]) -> A -> [C]
(:)       :: A -> [A] -> [A]
break     :: (A -> Bool) -> [A] -> ([A], [A])
filter    :: (A -> Bool) -> [A] -> [A]
scanl     :: (B -> A -> B) -> B -> [A] -> [B]
zipWith   :: (A -> B -> C) -> [A] -> [B] -> [C]
concat    :: [[A]] -> [A]
zip       :: [A] -> [B] -> [(A, B)]
usort     :: [Int] -> [Int]
sum       :: [Int] -> Int
(++)      :: [A] -> [A] -> [A]
map       :: (A -> A) -> [A] -> [A]
foldl     :: (A -> A -> A) -> A -> [A] -> A
takeWhile :: (A -> Bool) -> [A] -> [A]
foldr     :: (A -> A -> A) -> A -> [A] -> A
drop      :: Int -> [A] -> [A]
dropWhile :: (A -> Bool) -> [A] -> [A]
span      :: (A -> Bool) -> [A] -> ([A], [A])
unzip     :: [(A, B)] -> ([A], [B])
[]        :: [A]
partition :: (A -> Bool) -> [A] -> ([A], [A]))
take      :: Int -> [A] -> [A])
background [
(,)  :: A -> B -> (A, B),
fst  :: (A, B) -> A,
snd  :: (A, B) -> B,
(+)  :: Int -> Int -> Int,
0    :: Int,
succ :: Int -> Int]
```

Figure 3: A library of list functions.

In contrast, running RoughSpec on this set of functions we can tailor the templates we use to properties we are interested in discovering and produce a more manageable amount of output in a much shorter time. The list of functions is shown in Figure 3. The last six functions are declared as *background* functions. Background functions may appear in properties, but a discovered property must contain at least one non-background function.

We start with the following templates, all representing well-known patterns of laws:

```
identity: ?F X = X
fixpoint: ?F ?X = ?X
cancel: ?F (?G X) = ?F X
left-id-elem: ?F ?Y X = X
right-id-elem: ?F X ?Y = X
commutative: ?F X Y = ?F Y X
commuting-functions: ?F (?G X) = ?G (?F X)
distributivity: ?F (?G X Y) = ?G (?F X) (?F Y)
analogy-distributivity: ?F (?G X) (?G Y) = ?G (?H X Y)
associativity: ?F (?F X Y) Z = ?F X (?F Y Z)
```

Running RoughSpec on this set of functions with the above templates, we discover 164 properties in just under 4 minutes. The properties include many useful laws, such as distributivity-like properties:

```
length xs + length ys = length (xs ++ ys)
concat xss ++ concat yss = concat (xss ++ yss)
sum xs + sum ys = sum (xs ++ ys)
```

Template expansion results in more complex properties. The second property below has size 11, much larger than QuickSpec was able to discover:

```
take x (takeWhile p (zip xs ys)) =
  takeWhile p (zip (take x xs) (take x ys))
take x (zipWith f xs (zipWith g ys zs)) =
  zipWith f xs (zipWith g (take x ys) (take x zs))
```

These two properties are given as examples of distributivity (`take` is distributed over the rest of the expression). The user may not consider these laws interesting, which suggests that having a more expressive template language is important. Nonetheless, the laws discovered are better than those found by QuickSpec, and we are able to discover them in a fraction of the time. This demonstrates that RoughSpec is much better suited than QuickSpec to exploring large libraries of functions, and that it makes theory exploration tractable on such libraries that were previously infeasible to explore.

4 COMPARISON TO QUICKSPEC

RoughSpec and QuickSpec's approaches seem to be complementary. For large APIs, QuickSpec is slow, and often produces an overwhelming amount of output. By contrast, RoughSpec runs quickly, and produces a moderate number of laws. The laws it finds are easy to understand, because they follow standard patterns, and can be targeted to the user's interests.

On the other hand, RoughSpec does not usually find a complete specification. Even when testing lists, RoughSpec failed to find the law `reverse (xs ++ ys) = reverse ys ++ reverse xs`. This is by design but is nonetheless a weakness. We believe that a *hybrid* approach could work, where QuickSpec is used to find all *small* laws, running with a low size limit, and RoughSpec is used to find interesting laws beyond that.

We also ran into problems when our templates are too general, as in that case our premise of limiting the search space may no longer hold. For example, consider a template `?F(X) = ?G(X)` searching for equivalent functions. This template could produce interesting and useful properties, for instance stating that different sorting functions produce the same output for a given input. However, if our signature contains many functions that have the same type we will produce a large number of candidate properties and testing them will take a long time (and probably most will be falsified). Meanwhile, QuickSpec will discover relevant properties of this shape much more quickly. With a hybrid approach, we could leave QuickSpec to find properties of this shape.

5 RELATED WORK

Apart from QuickSpec [16] and Speculate [1], which we described in the Introduction, there are also theory exploration tools for mathematics. Below we describe several which support templates or schemas.

Buchberger [2] introduced the idea of schema-based theory exploration and his team implemented it in the Theorema [3] system. Theorema provides tools to assist the user in their theory exploration but does not automate the process. The user must provide the schemas (but can store them in a schema library for easier reuse), manually perform substitutions to instantiate the schemas with terms, and conduct proofs interactively.

IsaScheme [14] is a schema-based theory exploration system for Isabelle/HOL. Users provide the schemas as well as a set of terms to instantiate the schemas with, but the instantiation is performed automatically. The conjectures generated by instantiation are then automatically refuted using Isabelle/HOL's counter-example finders, or proved using the IsaPlanner [5, 6] prover.

MATHsAiD [13] is an automated theorem-discovery tool which has mainly been applied in the context of abstract algebra. It uses a combination of several exploration techniques, one of them being schema instantiation, which is used for a limited set of lemmas/theorems. The schemas used by MATHsAiD are predefined and built-in to the system and include, for example, reflexivity and transitivity.

6 FUTURE WORK

There are many avenues of future work we would like to explore.

Our tool could be made more user-friendly by not requiring the user to explicitly type up a signature. A default signature for a given set of functions could be automatically generated using Template Haskell.

QuickSpec has been used to discover lemmas in a theorem proving context, see [12], and we believe our extension could also be useful in such a context, using templates relevant for the theorem we would like to prove.

In the experiments described in this paper we have used handwritten templates provided by the user or by a library of default templates. We would like to further explore what kinds of templates are useful in a given context and how to automatically discover useful templates, using data-driven methods to learn good templates for a given context. We will explore using machine learning to extract common patterns from proof libraries, learning common lemma shapes given properties of the theorem we want to prove (c.f. [7]), as well as exploiting type-class laws and other algebraic properties. We will also investigate extracting templates from failed proof attempts, similar to critics in proof planning [11].

RoughSpec currently supports only equations as templates, but many applications require conditional equations. We are currently extending RoughSpec to discover conditional equations. In our approach, the user specifies a set of equational templates and a set of condition templates, and the tool discovers which conditions fit each equation. We believe this will make for a more practically useful tool.

As described in Section 4, QuickSpec is more efficient at discovering smaller properties with generic shapes while our tool can discover larger properties fitting more specific patterns much more quickly. A hybrid tool combining our extension with standard QuickSpec, i.e. using standard QuickSpec to discover properties up to a certain size and then switching to a template-based search, seems promising. This requires experiments to identify the "sweet spot" and develop a heuristic for when to switch approaches.

We currently use a set of heuristics to expand templates. Template expansion is important in order to capture a wide variety of laws, but it sometimes goes too far. For example, given the template `?F (?G X) = ?G (?F X)`, both `?F` and `?G` can be replaced by a nested function, resulting in laws of the form `f (g (h (i x))) = h (i (f (g x)))`. To reduce the use of heuristics, we would like to define an *expressive* template language, in which the user can say precisely what sort of laws they want, for example, to forbid the use of nested functions in the template above. As another example, it should be possible to define a template that capture a general distributivity law `f (g x1) (g x2)...(g xn)) = g (f x1...xn)` for *n*-ary functions, without specialising it to a particular *n*. Doing so requires designing a small set of combinators for building templates.

7 CONCLUSION

We have presented RoughSpec, a theory exploration tool in which the user specifies which properties are interesting. It generates specifications which are short, and easy to understand, but not complete. It can be used both to produce a rough specification of how a set of functions behaves, and to target specific families of laws that the user is interested in. It also scales well to large APIs. We believe that, together with QuickSpec, the two tools form a convincing theory exploration system for both small and large APIs.

ACKNOWLEDGMENTS

This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, and by the Swedish Research Council (VR) grant 2016-06204, *Systematic testing of cyber-physical systems (SyTeC)*.

REFERENCES

[1] Rudy Braquehais and Colin Runciman. 2017. Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell.* 40–51.
[2] Bruno Buchberger. 2000. Theory exploration with Theorema. *Analele Universitatii Din Timisoara, ser. Matematica-Informatica* 38, 2 (2000), 9–32.
[3] Bruno Buchberger, Adrian Craciun, Tudor Jebelean, Laura Kovács, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. 2006. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic* 4 (12 2006), 470–504. https://doi.org/10.1016/j.jal.2005.10.006
[4] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP.* 268–279.
[5] Lucas Dixon and Jacques D. Fleuriot. 2003. IsaPlanner: A Prototype Proof Planner in Isabelle. *LNCS (LNAI)* 2741, 279–283. https://doi.org/10.1007/978-3-540-45085-

6_22
[6] Lucas Dixon and Moa Johansson. 2007. IsaPlanner 2: A Proof Planner for Isabelle.
[7] Jonathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. 2013. Proof-Pattern Recognition and Lemma Discovery in ACL2. In *Proceedings of LPAR*. https://doi.org/10.1007/978-3-642-45221-5_27
[8] C. A. R. Hoare. 1976. Proof of correctness of data representations. In *Language Hierarchies and Interfaces*, Friedrich L. Bauer, E. W. Dijkstra, A. Ershov, M. Griffiths, C. A. R. Hoare, W. A. Wulf, and Klaus Samelson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 183–193.
[9] John Hughes. 1995. The Design of a Pretty-printing Library. In *Advanced Functional Programming*, J. Jeuring and E. Meijer (Eds.). Springer Verlag, LNCS 925, 53–96.
[10] John Hughes. 2020. How to Specify It!. In *Trends in Functional Programming*, William J. Bowman and Ronald Garcia (Eds.). Springer International Publishing, Cham, 58–83.
[11] Andrew Ireland and Alan Bundy. 1996. Productive Use of Failure in Inductive Proof. *Journal of Automated Reasoning* 16 (1996), 79–111.
[12] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. 2014. Hipster: Integrating Theory Exploration in a Proof Assistant. In *Proceedings of CICM*. Springer, 108–122.
[13] R. L. McCasland, A. Bundy, and P. F. Smith. 2017. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence* (23 Jun 2017). https://doi.org/10.1007/s10489-017-0954-8
[14] Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. 2012. Scheme-based theorem discovery and concept invention. *Expert systems with applications* 39, 2 (2012), 1637–1646.
[15] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*. 37–48.
[16] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. 2017. Quick specifications for the busy programmer. *Journal of Functional Programming* 27 (2017). https://doi.org/10.1017/S0956796817000090

Validating Formal Semantics by Comparative Testing

Péter Bereczky
Dániel Horpácsi
berpeti@inf.elte.hu
daniel-h@elte.hu
Eötvös Loránd University
Budapest, Hungary

Judit Kőszegi
Soma Szeier
koszegijudit@elte.hu
szeier529@inf.elte.hu
Eötvös Loránd University
Budapest, Hungary

Simon Thompson
S.J.Thompson@kent.ac.uk
University of Kent
Canterbury, UK
Eötvös Loránd University
Budapest, Hungary

Abstract

To describe the behaviour of programs in a programming language we can define a formal semantics for the language, formalising it in a proof assistant. From this semantics we can derive the behaviour of each particular program in the language. But there remains the question of validating the formal semantics: *have we got the semantics right?*

In this paper, we present our approach, property-based cross-testing of formal semantics, which is based on the combination of existing approaches to semantics validation. In particular, we present a prototype implementation for existing Erlang and Core Erlang formalisations. We describe the necessary adjustments needed to be made to execute these semantics, and then briefly summarise the technical details of the components of our prototype. Finally, we evaluate our preliminary results in the context of our short- and longer-term goals.

CCS Concepts: • **Theory of computation → Operational semantics**; **Program verification**; *Functional constructs*; • **General and reference → Validation**.

Keywords: formal semantics, validation, property-based testing, Coq, K framework

ACM Reference Format:
Péter Bereczky, Dániel Horpácsi, Judit Kőszegi, Soma Szeier, and Simon Thompson. 2018. Validating Formal Semantics by Comparative Testing. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

This work is part of a wider project that aims to reason about the correctness of refactoring. Our goal requires a rigorous, formal definition of the programming language under refactoring: in our case, Erlang. In earlier work, we have defined and implemented executable formal semantics for the sequential parts of both Erlang and Core Erlang. Initially we developed a reduction semantics for a subset of Erlang implemented in the $\mathbb{K}$ framework [19], while more recently we have defined a natural semantics for a subset of Core Erlang, implemented in the Coq proof assistant [1, 2].

In this paper, we investigate the validation of these semantic definitions by combining a number of techniques ranging from grammar-based and property-based testing to advanced proof tactics that are used to make big-step semantics executable. As Core Erlang is an intermediate language between Erlang and BEAM code [27], Erlang can be compiled to both Core Erlang and BEAM, and the semantics of these three languages can be contrasted. The presence of any discrepancies between these point to inconsistencies in the different semantics, whereas their absence provides evidence that the definitions are *valid relative to each other.*

There is not a complete, up-to-date and precise language specification available for any of the above languages. We therefore decided to take the Erlang/OTP compiler and the BEAM interpreter – i.e. the reference implementation – as the frame of reference for reasoning about correctness. This means that the compilation from Erlang to Core Erlang and from Core Erlang to BEAM, along with the BEAM interpretation, are trusted (Figure 2). The formal semantics of Erlang is said to be *correct* if and only if the BEAM code obtained by trusted translation from the Erlang program exhibits the same behaviour on the BEAM interpreter as the Erlang program exhibits according to the formal semantics; we investigate the correctness of the formal semantics of Core Erlang in a similar way.

Although the main idea is to test both semantics against the reference implementation on the BEAM, the cross-testing may come with extra benefits beyond the results of testing a single one, namely,

- If both formal semantics show the same (or similar) incorrect behaviour, that may indicate a generic misconception about the behaviour of a particular language feature, rather than an error in the formalisation,
- If one is correct and the other is incorrect, the correct definition can be used to assist the debugging of the incorrect one by exploiting the translation definition

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

used when transforming programs from Erlang to Core Erlang.

Beside hand-written test cases, we use *property-based testing* with randomly generated programs to validate the semantics. It is worth noting that the general idea of property-based cross-testing of (executable) semantics can be generalised for any two languages provided that one can be translated to the other.

The main contributions of this paper are:

- An approach to validation of formal semantics definitions by property-based random comparative testing.
- A validation architecture gluing both of the semantics (given in different systems) and the reference implementation.
- A method of making an inductive big-step semantics executable by means of advanced proof tactics in Coq.
- Extensive validation of a Core Erlang semantics implemented in Coq and an Erlang semantics given in the $\mathbb{K}$ framework.

The rest of the paper is structured as follows. In Section 2 we summarise the most common approaches to test a formal semantics, then in Section 3 we describe the general idea of our approach. In Section 4 we overview the semantics definitions to be validated, and in Section 5 we explain in detail how the prototype implementation performs the validation of the semantics. Section 6 presents and evaluates the findings, and finally Section 7 summarises future work and concludes.

2 Related Work

Although most programming languages lack a fully formal definition and are mainly defined by their reference implementation, there is an ever increasing effort on equipping mainstream languages with formal definitions. To mention but a few: C, Java, OCaml, Scheme, Haskell, PHP, EVM are being formalised in the $\mathbb{K}$ framework [18], while semantics for C [3], Javascript [5], R [6] and WebAssembly [17], among others, are being developed in the Coq proof assistant.

As other authors have pointed out [4, 15, 28], it is crucial to validate these formal definitions against the language specifications and the reference implementation; otherwise, the formal statements that hold in them could not be used to argue about the run-time behaviour of programs in the language. According to Blazy and Leroy [4], there are five basic methods to validate formal semantics:

1. Manual review and debugging
2. Proving properties of the semantics, such as type preservation and determinism
3. Using verified translations and trusted semantics
4. Validating executable semantics, e.g. testing against test suites and experimental testing
5. Using equivalent, alternate versions of the semantics

These methods, and the combinations thereof, are commonly used when a formal semantics definition is to be validated. The semantics of Lolisa [28] was validated with methods 2, 4 and 5, while CompCert [3, 4] apparently uses all five methods.

Yet, the most common way of validating a formal semantics is the 4th method: developing an executable version of the semantics and testing it agains the reference implementation. This method is used on the executable semantics for PHP [13], the semantics of SQL queries [15] and the semantics of Erlang [19], as well as in the work by Politz et al. on JavaScript [25] and in the work by Roessle et al. [26] on the big-step semantics of x86-64 binaries.

3 Formal Semantics Validation Approach

Our approach is a combination of the fundamental semantics validation techniques outlined by Blazy and Leroy [4]. In particular,

- We adapt method 3 by using verified translation (i.e. the official Erlang/OTP compiler) from Erlang to Core Erlang, and from Core Erlang to BEAM. Our trusted semantics component is the executable definition of BEAM (i.e. the official Erlang/OTP interpreter).
- We adapt method 4 by using a test suite as well as randomly generated programs to test our semantics against the reference implementation (i.e. the official Erlang/OTP interpreter). For this, we needed to make both the small-step semantics for Erlang and the big-step semantics for Core Erlang executable. Rather than investigating the definition of equivalent denotational semantics (or definitional interpreters), we sought to gather execution information from the big-step semantics, namely the final configurations and the corresponding proofs in the operational semantics. This approach is explained in detail in Section 4.
- Last but not least, we adapt method 5 by having semantics in two different styles (even though for two slightly different languages): the Erlang semantics is in small-step (reduction style with evaluation contexts), while the Core Erlang semantics is given as an inductive big-step (natural style) semantics.

We believe that this *combination* (as opposed to simple composition) of methods results in an even more effective formal semantics validation technique.

3.1 Property-based testing of formal semantics

In addition to the combination of well-understood techniques, our approach also proposes a novel feature: it employs property-based testing (PBT) for validating the formal semantics with randomized data (random programs executed with random parameters). For the testing of the Coq semantics we could have used QuickChick [11] as PBT implementation, but with the multiple semantics implemented in different systems, we opted for the Erlang QuickCheck [9] when designing our test bed. Note that PBT not only allows us to test with random data, but it can also control data distribution and it can assist comprehending errors by shrinking counterexamples.

Property-based testing of meta-programming tools (or programming language processors in general) requires a data generator for well-formed program terms. Horpácsi et al. [12] developed an attribute grammar based generator generator for Erlang QuickCheck (EQC), and they have formalised a subset of Erlang as an attribute grammar, which can be employed to synthesise a data generator for random Erlang programs. We took this result and tailored the generated programs (i.e. revised the grammar in order to modify the generated language) for semantics testing.

3.2 Architecture

Figure 1 shows an overview of the general idea. We consider two programming languages with reference implementations and executable formal semantics (possibly in different semantics frameworks), as well as a translator between the two languages. We use an EQC generator to synthesise random programs in the first language and translate it to the second language. Then we feed the original and translated programs into the corresponding implementations and semantics, and finally we compare the results. This latter step is of interest mainly from the technical point of view; in general, it is a structural equality check on the resulting values.

4 Executable Semantics for Erlang and Core Erlang

If an operational semantics (especially big-step semantics) is to be tested, therefore to be executed, one approach is to (re)define it in a computable style, such as the functional big-step semantics by Owens et al. [24]. Another option is to define a definitional interpreter as "equivalent alternate semantics" [4], but the denotational re-definition and the equivalence proof requires significant effort.

If one does not want to redefine the language, but the already defined operational semantics is not computable — either because it is not syntax-directed or it is not terminating — automatic execution is not trivial as it is essentially a proof search on the transition relation with existential variables. In Erlang and in Core Erlang, both exceptions and divergence are present, thus in our semantics definitions there can be several derivation rules applicable to a particular configuration.

In case of natural semantics, using pretty-big-step style [8] can reduce the number of applicable rules, but it cannot eliminate all decision points: for instance, executions may terminate either normally or with an exception, and even if the semantics is deterministic, we cannot tell in advance which branch leads to the normal form. The proof search is a depth-first search trying all of the evaluation paths one after another, which may have performance issues; in Section 4.2 we explain in detail how we managed to execute our traditional, inductive big-step semantics definition in Coq.

4.1 Erlang Semantics

The Erlang definition used in this project is given as reduction semantics with evaluation contexts. It is defined in the $\mathbb{K}$ framework[1], a language workbench that supports simple and effective syntax and semantics definitions, and generates various execution and analysis tools based on a single definition. One of the greatest features of this framework is that it has a reasonably effective search technique for finding small-step derivations, basically it synthesises an interpreter for the semantics definition. This means that the small-step semantics of Erlang is inherently executable with the help of $\mathbb{K}$ and does not need any special care in this regard. For the details of this language definition, we refer to previous work by Kőszegi [19].

4.2 Core Erlang Semantics

In our former work, we formalised sequential Core Erlang in Coq[2] considering exceptions and side effects too [1, 2, 20]. Unfortunately, this big-step semantics is an inductive type, which cannot be simply executed, as Blazy and Leroy also mentions [4].

Making it executable. In order to create an executable semantics for Core Erlang, we had make to some modifications in our description to enable simple pattern matching for the evaluation goals. Coq was not able to apply pattern matching on derivation rules which contained auxiliary function calls in their consequences (e.g. the derivation rule for variables and the use of append operation on side effect logs in our semantics [1]). This problem was avoided by introducing a new variable which replaced the auxiliary call, and the addition of a premise which states that this variable holds the result of the call in question.

On the other hand, in case of the side effect traces (and the mentioned append operations) to avoid the introduction of several new variables, we changed the use of these traces. Instead of handling only the additional side effects of an expression evaluation step, we rather consider using always the whole initial and final side effect traces (i.e. not only the difference). This way we could dispose of the append operations in the consequences of the derivation rules.

[1] $\mathbb{K}$ framework version 3.6
[2] Coq version 8.11.2

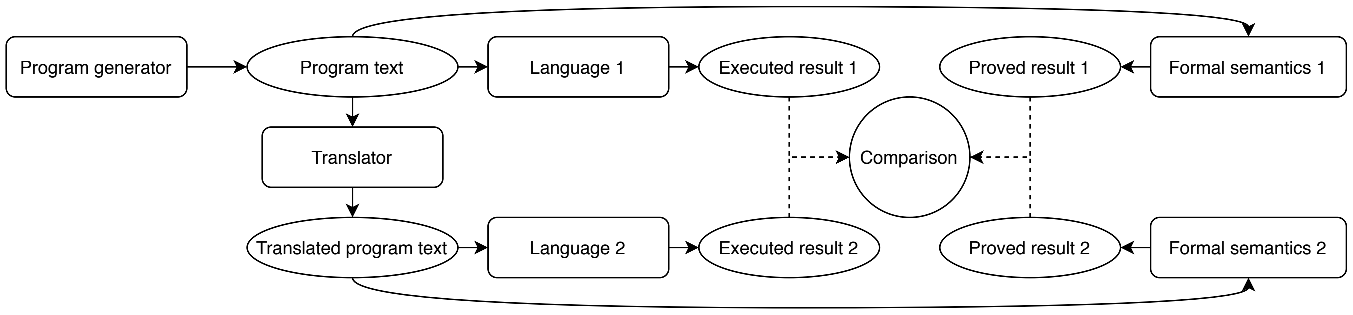


Figure 1. The general design of our approach

In our case, the evaluation of the Core Erlang semantics without exceptions is syntax-driven[3]. This means that a tactic [10] can be designed to evaluate any expression in any context based on pattern-matching on the expression to be evaluated.

However, after introducing exceptions, several derivation rules are applicable for evaluating an expression. We extended the evaluation tactic by applying one of the rules and if this fails, trying the next one. This can also be seen as a depth-first search for the successful evaluation path as mentioned before.

Thereafter, we introduced notations for the result in order to easily extract it to the corresponding Erlang value to enable comparison with the Erlang semantics and BEAM results.

Optimisation. Unfortunately, our evaluation tactic in Coq is quite slow and its memory usage is high. To speed up the execution of our semantics, we have also designed some helper functions and lemmas about specific expressions (e.g. the evaluation of tuple expressions which contain only literals), so that the evaluation tactic can apply these lemmas before trying to evaluate an expression with the mentioned depth-first search (if the helpers were not applicable). These lemmas can significantly speed up the evaluation of expressions which contain such specific sub-expressions.

In addition, it is also an interesting topic to compare this solution to other executable semantics styles. Our approach shares similarities with functional big-step semantics [24]; to ensure the termination of the tactic we use a time limit, which is similar to the "clock" in functional big-step semantics[4], moreover, the mentioned functions for optimisation can be seen as the functional big-step semantics of specific expressions.

[3]Our semantics is deterministic, however, Core Erlang itself is not [7], but we followed the footsteps of the reference implementation, which employs a leftmost-innermost evaluation strategy according to Neuhäußer and Noll [21].

[4]Alternatively, we could use the same concepts of recursion depth limit in the tactic too.

4.3 Notes on Language Coverage

In the setting of testing the two formal semantics with the same input, it is important to ensure that the language features covered by the Erlang definition translate to features covered by the Core Erlang definition. This is an issue to be taken account as our definitions do not cover the entire languages.

As a matter of fact, both the Erlang and Core Erlang formal definitions support most sequential constructs, such as arithmetic and boolean expressions, simple compound types (e.g. tuples, lists, maps), pattern matching, and control expressions (e.g. sequencing, `case`, `if`, subroutine calls). Beside these, both semantics define the behaviour of exceptional evaluation and tracing of simple side-effects (read and write to standard I/O).

Core Erlang has an official but out-of-date specification [7] against which we can measure the coverage, as well as both languages have formal syntax definitions ([22, 23]) which can be interpreted as a catalogue of language features. We have decent coverage of sequential language elements, although some parts were intentionally left out as we aimed at only formalising a representative set of basic constructs and types. Missing features include binaries, bitstrings, annotations, as well as float, char and string expressions. It should be noted too that the current definitions lack the definition of the concurrent programming features, but there is extensive literature on the definition thereof [14, 16] and we plan to extend our semantics in this regard.

Interestingly enough, full coverage of Erlang does not ensure full coverage on Core Erlang. In fact, Core Erlang is a richer language than that covered as the compiler is applied to Erlang, according to our testing. For instance, we could not generate `case` expressions with a non-empty "ValueList" [7]. Core Erlang language features not used by the object code of the Erlang compiler shall be validated separately.

5 Testing the Semantics of Erlang and Core Erlang

In this section, we give an overview on the structure and the behaviour of our prototype implementation of the semantics

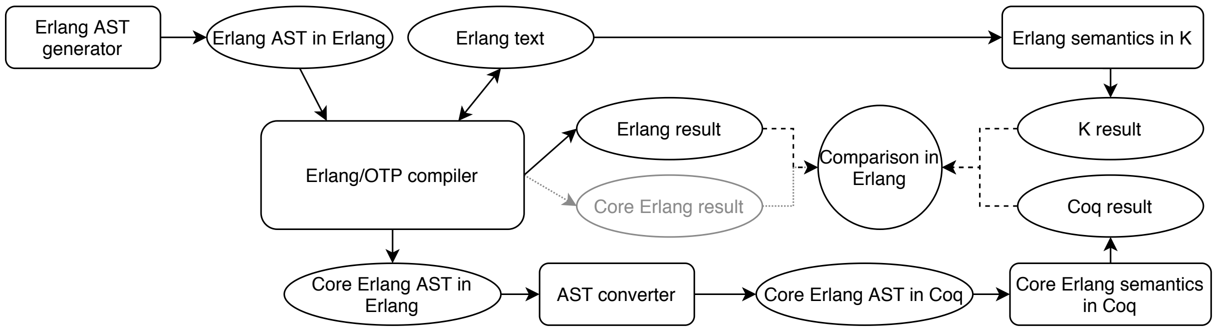


Figure 2. The components of our prototype

validation system. Basically, it compares the behaviour of the above-mentioned small-step semantics of Erlang implemented in the $\mathbb{K}$ framework and the big-step semantics of Core Erlang implemented in the Coq proof assistant with each other and with the behaviour of the reference implementation, by using randomly generated test programs. The structure of the prototype can be seen in Figure 2.

5.1 Random Program Generator

By default, the validation process uses a test suite, but it can also be instrumented to use random test data. For this, we use QuickCheck generators, which define a (weighted) set from which the testing chooses elements randomly. In previous work on validating refactoring tools [12] we implemented an attribute grammar based generator for syntactically and static semantically valid sequential Erlang programs. In order to use this for testing the Erlang semantics, we needed to match the generator grammar to the language coverage of the semantics, such that we only generate programs that we can evaluate in the formal definition. It is important to note that the generated programs are not only (syntactically) well-formed, but they adhere to the static semantics of the language (do not refer to unbound names) and are free of trivial type errors — this is supposed to dramatically improve the efficiency of fully randomised testing.

5.2 The Erlang/OTP Compiler

The Erlang/OTP compiler and interpreter (i.e. the reference implementation of Erlang[5]) is a trusted component and reference for reasoning in our solution. It plays four different roles:

- Pretty-prints randomly generated Erlang syntax trees
- Translates Erlang to Core Erlang and emits the abstract syntax tree (AST)
- Translates Erlang to BEAM and interprets the bytecode (i.e. executes the program to be tested and provides the result expected from the semantics definitions)
- Compares the results emitted by the semantics to the expected result

[5]Erlang/OTP version 22.0

Worth noting that in the Erlang to Core Erlang translation, we disable optimisation in order not to reduce the original code complexity. We plan to refine this solution and perform the validation with both the optimised and the unoptimised versions of the Core Erlang object code.

5.3 Conversions

Beside using the Erlang/OTP compiler for converting between abstract syntax trees (i.e. for parsing and pretty-printing), we needed to develop a glue component that helps feed the Core Erlang program into the Coq implementation of the semantics. As we ought to avoid developing a Core Erlang parser in Coq, we opted for pretty-printing the Core Erlang AST into Coq text defining the very same AST within Coq.

In particular, we have written an algorithm based on the official Core Erlang parser [22], which pretty-prints the Core Erlang AST (represented in Erlang) into a Coq proof goal and proof command that evaluates the AST and extracts the evaluation result.

While implementing this component, we encountered some difficulties when handling value lists and `try` expressions. In case of value lists, the Coq semantics needs adjustment, while in case of `try` we handle only three variable bindings in the `catch` clause, whereas the syntax allows the binding of any number of variables. This behaviour was based on informal semantics of `try` expressions described in the language specification [7]. Moreover, the official parser handles tuple and list expressions that contain only literals separately from other tuples and lists, which caused additional technical difficulty while implementing this component.

5.4 Orchestration

In our prototype implementation, the validation process is controlled by a shell script that coordinates and glues the rest of the components. In particular, it uses the QuickCheck generator to synthesise random programs, invokes the reference implementation to obtain the expected result, does the conversions to obtain representations to be fed into the formal semantics in $\mathbb{K}$ and Coq, invokes the semantics, and finally, uses the Erlang interpreter to compare the results. The test system can be parametrized to use hand-written tests or randomly generated tests, and produces statistics that characterise failing cases by labelling them (e.g. errors/incorrect results in either semantics). In the long term we want to implement the entire orchestration to provide full support for property-based testing, including thoroughly designed shrinking for random programs and a refined correctness property on the comparison of results.

6 Evaluation

This style of testing pointed out errors in the Core Erlang semantics, which were not encountered before by using only our test suite. Specifically, the most serious error we discovered, is that value lists are only partially supported (only in `let`, `case` and `try` expressions). This error was highlighted specifically by using unoptimised translated Core Erlang code (from Erlang).

Moreover, in case of `try` expressions in Core Erlang, although the language specification [7] explicitly states that three variables are bound in the `catch` clause in Erlang implementations, this was not always the case; sometimes only two handler variables were present. However, the language specification was written in 2004, so this information can be outdated, thus we need to investigate this issue.

In addition, we also found some minor faults in both of the semantics, e.g. some essential built-in functions were missing or their names were spelled wrongly, and some list operations worked on improper lists too, which they should not.

In terms of execution speed, while validating the semantics, the evaluation with our Coq tactic had the longest execution time (the three most complex examples were executed over four minutes in our test suite, even with the optimisation mentioned in Section 4.2). As Blazy and Leroy [4] mentioned, Coq is not the most efficient tool for executing specifications written using inductive types, even with our tactic. To simplify the Coq execution (i.e. the depth-first search), we could modify our semantics in a pretty-big-step way [8] to reduce the number of applicable constructors while we could design an equivalent interpreter or a functional big-step semantics [24] to increase the evaluation speed. Alternatively, to speed up our tactic, additional helper functions and theorems about evaluating specific expressions can be introduced, as mentioned in Section 4.2.

6.1 Coverage

The efficiency of our testing can be measured by the coverage of the semantics; the greater the code (rule) coverage, the more efficient the testing can be considered.

Currently, we measure the code coverage of our testing approach only informally with our hand-written test suites and the language elements supported by the random program generation (and the corresponding attribute grammar). Therefore, before writing a final paper about this research we will measure the line and rule coverage of our semantics with dedicated tools.

We also plan to investigate the coverage of the translated code from Erlang in the Core Erlang semantics, i.e. which Core Erlang expressions cannot be generated by the translation from Erlang. To tests these expressions, we plan to extend our test suite for the Core Erlang semantics.

7 Conclusion and Future Work

In conclusion, in this paper, we described an approach of validating formal semantics by testing them against each other and the reference implementation in a property-based way which is based on the combination of well-known semantics validation approaches. We also discussed our prototype implementation of testing Erlang and Core Erlang semantics including the necessary adjustments we made to execute our semantics (especially, the big-step semantics of sequential Core Erlang in Coq). Then we briefly summarised the technical details of our prototype, and evaluated our preliminary results.

In the near future, before submitting a full paper about this research, we will further increase and formally measure the coverage of our testing approach. We also plan to design an alternate semantics in Coq, which can be executed more efficiently.

Apart from these short-term goals, we also have some medium-term goals:

- Simplifying the evaluation tactic in Coq
- Shrinking incorrectly evaluated input programs
- Comparing the side effects produced by the semantics and the reference implementation beside the result values
- The adjustment of the value list concepts in the Core Erlang semantics
- Implementing the orchestration in a concurrent way, to shorten execution time

Our long term plans also include the formalisation of Erlang and the concurrent parts of Core Erlang in Coq.

Acknowledgments

The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, "Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications (3IN)").

Project no. ED_18-1-2019-0030 (Application domain specific highly reliable IT solutions subprogramme) has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme funding scheme.

References

[1] Péter Bereczky, Dániel Horpácsi, and Simon J. Thompson. 2020. Machine-Checked Natural Semantics for Core Erlang: Exceptions and Side Effects. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang* (Virtual Event, USA) *(Erlang 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3406085.3409008

[2] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2020. A Proof Assistant Based Formalisation of Core Erlang. (2020). arXiv:2005.11821

[3] Sandrine Blazy. 2007. Experiments in validating formal semantics for C. In *C/C++ Verification Workshop*. Oxford, United Kingdom, 95–102. https://hal.inria.fr/inria-00292043

[4] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (Jul 2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[5] Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. *SIGPLAN Not.* 49, 1 (Jan. 2014), 87–100. https://doi.org/10.1145/2578855.2535876

[6] Martin Bodin, Tomás Diaz, and Éric Tanter. 2018. A Trustworthy Mechanized Formalization of R. *SIGPLAN Not.* 53, 8 (Oct. 2018), 13–24. https://doi.org/10.1145/3393673.3276946

[7] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. 2004. *Core Erlang 1.0.3 language specification.* Technical Report. https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf

[8] Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 41–60. https://doi.org/10.1007/978-3-642-37036-6_3

[9] Koen Claessen and John Hughes. 2011. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 46, 4 (May 2011), 53–64. https://doi.org/10.1145/1988042.1988046

[10] Coq documentation 2020. *Ltac documentation.* Retrieved August 13rd, 2020 from https://coq.inria.fr/refman/proof-engine/ltac.html

[11] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*.

[12] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. 2010. Quickchecking Refactoring Tools. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang* (Baltimore, Maryland, USA) *(Erlang '10)*. Association for Computing Machinery, New York, NY, USA, 75–80. https://doi.org/10.1145/1863509.1863521

[13] Daniele Filaretti and Sergio Maffeis. 2014. An Executable Formal Semantics of PHP. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 567–592. https://doi.org/10.1007/978-3-662-44202-9_23

[14] Lars-Åke Fredlund. 2001. *A framework for reasoning about Erlang code.* Ph.D. Dissertation. Mikroelektronik och informationsteknik.

[15] Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 27–39. https://doi.org/10.14778/3151113.3151116

[16] Joseph R. Harrison. 2017. Towards an Isabelle/HOL Formalisation of Core Erlang. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang* (Oxford, UK) *(Erlang 2017)*. Association for Computing Machinery, New York, NY, USA, 55–63. https://doi.org/10.1145/3123569.3123576

[17] Xuan Huang. 2019. A Mechanized Formalization of the WebAssembly Specification in Coq. https://www.cs.rit.edu/~mtf/student-resources/20191_huang_mscourse.pdf

[18] K projects 2020. *K framework project catalogue.* Retrieved August 14th, 2020 from http://www.kframework.org/index.php/Projects

[19] Judit Kőszegi. 2018. KErl: Executable semantics for Erlang. *CEUR Workshop Proceedings* 2046 (2018), 144–160. http://ceur-ws.org/Vol-2046/koszegi.pdf

[20] Natural Semantics for Core Erlang 2020. *Core Erlang Formalization.* Retrieved August 17th, 2020 from https://github.com/harp-project/Core-Erlang-Formalization

[21] Martin Neuhäußer and Thomas Noll. 2007. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science* 176, 4 (2007), 147–163. https://doi.org/10.1016/j.entcs.2007.06.013 Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006).

[22] Official Core Erlang Parser 2018. *Core Erlang YECC Parser Grammar.* Retrieved August 13rd, 2020 from https://github.com/erlang/otp/blob/master/lib/compiler/src/core_parse.yrl

[23] Official Erlang Parser 2020. *Erlang YECC Parser Grammar.* Retrieved August 13rd, 2020 from https://github.com/erlang/otp/blob/master/lib/stdlib/src/erl_parse.yrl

[24] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

[25] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. *SIGPLAN Not.* 48, 2 (oct 2012), 1–16. https://doi.org/10.1145/2480360.2384579

[26] Ian Roessle, Freek Verbeek, and Binoy Ravindran. 2019. Formally Verified Big Step Semantics out of x86-64 Binaries. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 181–195. https://doi.org/10.1145/3293880.3294102

[27] The BEAM Book 2020. *The Erlang Runtime System.* Retrieved August 13rd, 2020 from https://github.com/happi/theBeamBook/releases/download/0.0.14.fix/beam-book.pdf

[28] Zheng Yang and Hang Lei. 2018. Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language. arXiv:1803.09885 [cs.PL]

An Adventure in Symbolic Execution (extended abstract)

Gergő Érdi
Standard Chartered Bank
gergo@erdi.hu

ABSTRACT

ScottCheck is a verifier for text adventure games based on symbolic execution. Its implementation is based on an idiomatic concrete interpreter written in Haskell. Even though Haskell is a general-purpose functional language, the changes required to transform it into a symbolic interpreter turned out to be fairly small.

ACM Reference Format:
Gergő Érdi. 2020. An Adventure in Symbolic Execution (extended abstract). In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL '20).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Interactive fiction is a format of computer programs that can broadly be described as a textual back-and-forth between a human player and an automated world simulation. A subset of them, text adventure games, are characterized by having explicit win and failure states, tracing their lineage back to 1976's *Colossal Cave Adventure*. The usual implementation strategy of text adventure games is to use a domain-specific language for describing the specifics of individual game worlds, and then create interpreters for this language, targeting whatever platforms the game is to be released on.

An adventure game is essentially a puzzle, and a puzzle that has no solution can be a frustrating experience for the player. Starting from the initial state, there should always be a way to get to a winning state.

We can use symbolic execution of the game world description to check if there is a sequence of player inputs that result in a winning end state. One approach is to take an off-the-shelf interpreter, and compile it into symbolically executed code: our interest in this topic was sparked by previous work[3] in which the `scottfree` interpreter, itself is written in C, is compiled with SymCC[6] into symbolic form. Another possible approach would be to implement the interpreter in an environment with ambient symbolic evaluation, such as Rosette[8].

Our work explores the low-tech approach of using the general-purpose functional programming language Haskell, implementing a concrete interpreter idiomatically, and then changing it just enough to be able to execute it symbolically and pass it to an SMT solver to find input that satisfies the winning condition.

Gergő Érdi is employed by Standard Chartered Bank. This paper has been created in a personal capacity and Standard Chartered Bank does not accept liability for its content. Views expressed in this paper do not necessarily represent the views of Standard Chartered Bank.

2 STRUCTURE AND INTERPRETATION OF ADVENTURE GAMES

Following previous work in [3], we focus on the format of Scott Adams's text adventure games, originating from his first game, 1978's *Adventureland*. The game world is modeled as a space of discrete *rooms*, connected with each other in the six cardinal directions. Each room comes with a textual description to present to the player. The rooms also contain *items*, which are objects the player can manipulate. Most notably, items can be moved around either directly by the player (by taking them, moving to another room and dropping them), or by various world simulation events.

Beside the data describing rooms, their connections, items, and their starting locations, the game files also contain *scripts* in a simple language. Each script line consists of a set of *conditions* (e.g. *is item #4 currently in the same room as the player character?*) and a sequence of *instructions* (e.g. *swap locations of items #5 and #2*).

Player input is processed by parsing against two small dictionaries of verbs and nouns. Script lines can either be automatic, executing in every turn regardless of user input; or keyed to some combination of a verb and a noun index.

Unlike more elaborate winning conditions in other games, the Scott Adams adventure games all uniformly use the concept of collecting *treasure* items as the goal. One room is marked as the *treasury*; the `SCORE` command shows the current number of treasures in the treasury, and finishes the game if it is equal to the number of all treasure items in the game.

3 MONAD TRANSFORMERS FOR CONCRETE INTERPRETERS

The concrete interpreter is based on the traditional stack of monad transformers[4]: a *Reader* giving access to the world description, a *Writer* collecting the output messages, and a *State* consisting of the current item locations, including the location of the player-controlled avatar:

```
type GameData = …
data St = St
  { currentRoom :: Int16
  , itemLocations :: Array Int16 Int16
  }
type Engine = ReaderT GameData (WriterT [String] (State S))
```

Each turn of the game takes three steps: world simulation, user input, then response to the player input. This means the interaction model itself is monadic as well: the player can see all previous output before deciding on their next input. We implement this structure by doing the first and the third step inside *Engine*. This means we have a purely functional core, with an external, thin layer of IO only to take care of showing output and getting input.

4 SYMBOLIC EXECUTION AND PUZZLE TESTING

To turn the interpreter into a solver, we change it from concrete to symbolic execution. SBV[2] is a Haskell library providing types that support symbolic evaluation. The resulting symbolic constraints are then passed to an SMT solver; in our case, we use the open-source solver Z3[1].

This code transformation is surprisingly straightforward and painless. The solver-specific parts begin only after the game data has been read and parsed; we can keep the parser as-is. The interpreter state is changed to use SBV's symbolic types (prefixed with an *S*):

```
data S = S
  { currentRoom :: SInt16
  , itemLocations :: Array Int16 SInt16
  } deriving (Generic, Mergeable)
```

Here, *SInt16* is SBV's 16-bit integer type. *itemLocations* is still a static array of symbolic values, since the set of items remains constant during play-through for a given game: only the locations of items (i.e. the elements of the array) change. We let data-generic instance deriving[5] write the instance for SBV's *Mergeable* typeclass; this typeclass enables branching in symbolic results, which is crucial when interpreting conditions that check item locations.

Arithmetic works without change, since SBV types implement the *Num* typeclass. Because in standard Haskell, operators like == are not overloaded in their return type, the Boolean operators have SBV-specific versions.

This takes care of data. For control, we can write *Mergeable* instances for *ReaderT*, *WriterT* and *State* since these are all just typed wrappers around bog-standard function types. This allows us to define symbolic versions of combinators like *when*, or **case** with literal matches. Thus, we can build up the kit that enables writing quite straightforward monadic code, just by replacing some combinators with their symbolic counterpart. Here's an example of the code that runs a list of instruction codes in the context of their conditions; even without seeing any other definitions, it should be fairly straightforward what it does:

```
execIf :: [SCondition] → [SInstr] → Engine SBool
execIf conds instrs = do
  (oks, args) ← partitionEithers ⟨$⟩ mapM evalCond conds
  let ok = sAnd oks
  sWhen ok (exec args instrs)
  return ok
```

5 NOTIONS OF ADVENTURING AND MONADS

At this point, we have a symbolic interpreter which can consume user input line by line:

```
stepPlayer :: (SInt16, SInt16) → Engine (SMaybe Bool)
stepPlayer (verb, noun) = do
  perform (verb, noun)
  isFinished
```

The question then is, how do we keep turning the crank of this and let the state evolve for more and more lines of symbolic input, until we get an *sJust sTrue* result, meaning the player has won the game? SBV's monadic *Query* mode provides a way to do this incrementally: at each step, fresh free symbolic variables standing for the next input line are fed to the state transition function, yielding a new symbolic state and return value. Then, satisfiability of this new return value being *sJust sTrue* is checked with the SMT solver; if there's no solution yet, we keep this process going, letting the next *stepPlayer* call create further constraints. Furthermore, since the *Query* monad allows IO, we can recover the behavior of our original, concrete interpreter. Instead of using free variables for the input at each step, we read and parse the player's input into *SInt16* variables containing concrete values. Since the only potentially symbolic arguments to the *Engine* are the player inputs, if those are concrete, everything further downstream will also be concrete. In particular, the output messages, while their type is *SString*, contain concrete values which can be extracted into the standard *String* type for printing. This allows the same interpreter implementation to be used for both solving and interactive playing.

6 CONCLUSION

The full code of our symbolic Scott Adams adventure game interpreter is available under the terms of the MIT license from https://github.com/gergoerdi/scottcheck.

The combination of Haskell, a general-purpose functional language, and SBV, a library for SMT-based verification, allowed rapid development of a symbolic interpreter with acceptable real-world performance: *ScottCheck* was written from scratch in a single week, by an author previously unfamiliar with symbolic execution techniques. In terms of performance, with the Z3 SMT solver backend, it can successfully find a solution (consisting of 14 steps) for the fourth tutorial adventure from the *ScottKit* suite[7] in three and a half minutes. Further testing with more complicated adventures remains future work.

REFERENCES

[1] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
[2] L. Erkök. SBV: SMT based verification in Haskell, 2011. URL https://leventerkok.github.io/sbv/.
[3] M. M. Lester. Program transformations enable verification tools to solve interactive fiction games. In *7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, 2020.
[4] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995.
[5] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for haskell. *ACM Sigplan Notices*, 45(11):37–48, 2010.
[6] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, 2020. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.
[7] M. Taylor. ScottKit - a toolkit for Scott Adams-style adventure games, 2009. URL https://rdoc.info/github/MikeTaylor/scottkit.
[8] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

Using OO Design Patterns in a Functional Programming Setting

The Implementation of the FSM Visualization Tool

Joshua Schappel
Seton Hall University
South Orange, NJ, USA
joshua.schappel@student.shu.edu

Sachin Mahashabde
Seton Hall University
South Orange, NJ, USA
sachin.mahashabde@student.shu.edu

Marco T. Morazán
Seton Hall University
South Orange, NJ, USA
morazanm@shu.edu

ABSTRACT

This article presents the implementation of a visualization tool for designing and debugging state machines in FSM—a domain specific language for the automata theory classroom. The FSM visualization tool is implemented in Racket. At the heart of the implementation is the use of object-oriented design patterns employing hallmarks of functional programming such as pattern matching and higher-order functions. The use of the Builder pattern to implement buttons and input fields, the use of the Factory Method pattern to implement scroll bars, and the use of the Builder and Adapter patterns to implement a foreign library interface are described. The implementation of each of these design patterns is summarized to enable their adoption by programmers at large.

CCS CONCEPTS

• Software and its engineering → Integrated and visual development environments; **• Theory of computation → Formal languages and automata theory**; **• General and reference → Design**.

KEYWORDS

Design Patterns, Functional Programming, Finite State Machine Visualization Tool

ACM Reference Format:
Joshua Schappel, Sachin Mahashabde, and Marco T. Morazán. 2020. Using OO Design Patterns in a Functional Programming Setting: The Implementation of the FSM Visualization Tool. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'20).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

It is not uncommon for Computer Science students to feel apathy towards the material covered in an Automata Theory and Formal Languages course. Computer Science students, trained to program, find such a course very challenging and sometimes even overwhelming. This occurs because Automata Theory courses are typically taught in a manner that goes against the grain of what students learned. That is, students are asked to solve problems without being able to test and get immediate feedback on their solutions–typically provided by a compiler or an interpreter. Such immediate feedback is not received when students are asked to develop a state machine by pencil and paper. More often than not, this leads to buggy solutions, low grades, and frustration.

To reduce apathy and frustration, a domain-specific language (DSL), FSM (**F**unctional **S**tate **M**achines), was developed [14]. This DSL (embedded in Racket) allows students to implement finite-state machines. It provides students with:

- Constructors for deterministic finite state machines, nondeterministic finite state machines, pushdown automatons, and Turing machines.
- Selectors to access the states, alphabet(s), starting state, final states, and transition rules.
- Random testing facilities that provide students (and instructors) with immediate feedback.
- A tailor-made error-messaging system that provides students with clear feedback messages [15].

By using FSM, a student is able to debug a machine before submitting it for grading. Furthermore, it allows a student to implement the machine-building algorithms they develop as part of their constructive proofs. In this manner, students can test their algorithms before attempting to complete a formal proof. The result has been that students experience less frustration and earn higher marks.

Although apathy towards Automata Theory is reduced, many students feel that they need a tool to visualize machine execution. Students quickly started using visualization tools like JFLAP [22], jFAST [26], and FSA [9]. They find these tools too distracting given that these tools require students to create their own state diagrams. Furthermore, they found themselves having to create two implementations: one in FSM and one for the foreign visualization tool. This led to the development of the FSM visualization tool. The FSM visualization tool is seamlessly integrated into the DSL and allows students to immediately visualize and edit any defined machine. Instead of focusing on developing state diagrams, the FSM visualization tool allows students to focus on the design of their machines.

The development of the FSM visualization tool proved to be an interesting exercise that led to the extensive use of design patterns typically associated with object-oriented (OO) programming. This article describes how design patterns are used in the implementation of the visualization tool. It is, however, not an implementation manual. Instead, this article describes how different design patterns were used and implemented aiming to avoid the need for future FSM developers to perform major code rewrites. The article is organized as follows. Section 2 provides a brief overview of the OO design patterns discussed in this article. Section 3 provides an overview of FSM and the FSM visualization tool. Section 4 describes the design of

IFL'20, September 2020, Cantebury, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

buttons using the Builder pattern. Section 5 describes the implementation of input fields using the Builder pattern. Section 6 describes the implementation of scroll bars using the Factory Method pattern. Section 7 describes the implementation of the interface with the Graphviz library (to automatically create state diagrams) using the Builder and Adapter patterns. Section 8 summarizes the pattern implementations developed. Section 9 discusses related work. Finally, Section 10 presents concluding remarks and directions for future work.

2 OVERVIEW OF OO DESIGN PATTERNS

Design patterns are used to write code that is easy to maintain and refine [10]. They are commonly associated with OO programming, but exist in other paradigms. In essence, a design pattern captures a recurring design problem and its solution. Design patterns are popular because they provide programmers with flexibility, reusability, a shared vocabulary, and capture best practices. In addition, using design patterns help improve program documentation and maintenance.

Design patterns are generally categorized as creational, structural, or behavioral. Creational patterns are used to design the creation of objects. Structural patterns use inheritance to compose implementations or interfaces. Behavioral patterns are used to design patterns of communication between objects. This article focuses on the use of 2 creational patterns and 1 structural pattern in a functional programming setting. The creational patterns are the Factory Method pattern and the Builder pattern. The structural pattern is the Adapter pattern.

The Builder pattern is used when object creation is complex and the objects created may have different representations. It separates object construction from its implementation. A class delegates the construction of an object to a Builder object, and each possible representation of an object is captured by a different Builder. For example, the Builder pattern is used to create an RTF (Rich Text Format) document converter. This design pattern allows the programmer to add new conversion types to the builder without affecting the original class structure [5]. In Scala, the Builder pattern is integrated into the language and is used to allow combiner methods to build new collections such as map [25].

The Factory Method pattern defines an interface for creating an object and defers instantiation to subclasses. In essence, it encapsulates the instantiation of concrete types. The Factory Method selects a class based on the application context and then instantiate the selected class. It returns this instantiation as an instance of the parent class type. The Factory Method pattern is used, for example, to create an abstraction over DAOs (Data Access Objects) in an ORM (Object Relational Mapping System) [18] to manage database connections. The Factory Method pattern is also used in java.net.URLStreamHandlerFactory, which abstracts over the protocol type (e.g., http, ftp) [16].

The Adapter pattern is used to convert what a class exposes to what is expected by another class. In essence, it adapts an interface into another (expected) interface. This allows classes to work together despite interface incompatibilities. These classes are able to work together without modifying the original classes [5]. Without taking the analogy too far, one may say that incompatible objects are fooled into thinking that they are directly working together. For example, an adapter is used to bridge a graphical-based program support and a third-party text program [5].

3 FSM OVERVIEW

FSM is a DSL for programming state machines and grammars. It is extensively used in Seton Hall's upper-level undergraduate automata theory and formal languages course. This section first briefly outlines the classical definitions of finite-state automatons, pushdown automatons, and Turing machines. After this, the language support for state-based machines is outlined. To make the use of FSM concrete a small example is presented. Finally, the FSM visualization tool is outlined.

A finite-state automaton (fsa), M, is a quintuple:

K_M: The set of states
Σ_M: The set of input symbols
S_M: The starting state $\in \mathsf{K}_M$
F_M: The set of final states $\subseteq \mathsf{K}_M$
δ_M: The set of transitions: (P σ Q),
  where $\sigma \in \{\Sigma_M \cup \{\epsilon\}\} \wedge$ P, Q $\in \mathsf{K}_M$

We say that M is deterministic if δ_M is a function. Otherwise, M is nondeterministic. Each transition rule, (P σ Q), moves M from state P to state Q by consuming σ from and moving right on the input tape.

A pushdown automata (pda), P, is a sextuple:

K_P: The set of states
Σ_P: The set of input symbols
Γ_P: The set of stack symbols
S_P: The starting state $\in \mathsf{K}_M$
F_P: The set of final states $\subseteq$ K_M
δ_P: The set of transitions: ((R σ ρ) (Q ϱ)),
  where $\sigma \in \{\Sigma_P \cup \{\epsilon\}\} \wedge$ R, Q $\in \mathsf{K}_P \wedge \rho, \varrho \in \Gamma^*$

Unlike an fsa, P has a stack that is used as memory. Each transition rule, ((R $\sigma\rho$) (Q ϱ)), moves P from state R to state Q by consuming σ, popping ρ, pushing ϱ, and moving right on the input tape.

A Turing machine (tm), T, is a quintuple:

K_T: The set of states
Σ_T: The set of input symbols
S_T: The starting state $\in \mathsf{K}_M$
F_T: The set of final states $\subseteq$ K_M
δ_T: The set of transitions: ((P σ) (Q υ)),
  where $\sigma \in \Sigma_T \cup \epsilon \wedge$ P, Q $\in \mathsf{K}_T \wedge \upsilon \in \{\sigma \mid \rightarrow \mid \leftarrow\}$

Unlike a pda, T does not have a stack. Each transition rule, ((P σ) (Q υ)), moves T from state P to state Q by consuming σ and performing action υ. The action is either moving left on the tape, moving right on the tape, or writing to the current position on the tape.

The input tape of a state machine, N, starts with, w, a word to process consisting of zero or more elements in Σ_N. We say that N accepts w if there exists a sequence of transitions that take N from S_N to f $\in \mathsf{F}_N$. For an fsa and a pda all the input must be consumed. In addition, for a pda the stack must be empty. Otherwise, N rejects w.

FSM uses the definitions displayed in Figure 1 to represent machines. Briefly, states are represented by symbols and letters are represented by the lowercase characters in [a..z]. Input and stack

```
     state → symbol

      lttr → [a..z]

  alphabet → (lttr*)

      word → (lttr*)

     trans → fsa-rule | pda-rule | tm-rule

  fsa-rule → (state {symbol | ϵ} state)

  pda-rule → ((state letter (lttr*))(state (lttr*)))

   tm-rule → ((state lttr) (state action))

    action → lttr | ← | →

    config → fsa-config | pda-config | tm-config

fsa-config → (word state)

pda-config → (word (word*) state)

 tm-config → (word natnum state)

    result → accept | reject | ϵ

     trace → (append (config*) (result))

        sm → fsa-interface | pda-interface | tm-interface
```

Figure 1: FSM Definitions for Machine Representation.

alphabets, as well as input words, are represented using a list of letters. A transition is any type of machine rule. A finite state automaton (deterministic or nondeterministic) rule is a triple with a source state, a consume item (a lttr or, ϵ, empty), and a destination state. A pushdown automaton rule is a triple and a double. The triple contains a source state, a consume item, and a list of letters to pop off the stack. The double contains a destination state and a list of letters to push onto the stack. A Turing machine rule consists of two doubles. The first double is a source state and a consume item. The second double is a destination state and an action. An action represents either a write, a move head left one space, or a move head right one space. A machine configuration, config, is a list representing a machine's state. For an fsa, it is a list containing the unconsumed input and a state. For a pda, it is a list containing the unconsumed input, the stack, and a state. For a Turing machine, it is a list containing the input tape, the head's position on the tape, and a state. The result of applying a machine to a word is either **accept**, **reject** or ϵ[1]. The **trace** of a computation is a list of configurations ending with a result. Finally, a state machine, sm, is an interface.

[1]The result is empty only for Turing machines that do not decide a language.

Based on these definitions, the FSM's state machine interface is described as follows[2]:

- **make-dfa**: (state$^+$) alphabet state (state*) transitions ['no-dead]→ dfa
  Purpose: To construct a deterministic finite-state automaton.
- **make-ndfa**: (state$^+$) alphabet state (state*) transitions → ndfa
  Purpose: To construct an nondeterministic finite-state automaton.
- **make-pda**: (state$^+$) alphabet alphabet state (state*)) transitions → pda
  Purpose: To construct a Pushdown Automaton.
- **make-tm**: (state$^+$) alphabet state (state*) transitions → tm
  Purpose: To construct a Turing machine.
- **sm-getstates**: sm → (state$^+$)
  Purpose: To access the given machine's set of states
- **sm-getalphabet**: sm → alphabet
  Purpose: To access the given machine's alphabet
- **sm-getstart**: sm → state
  Purpose: To access the given machine's starting state
- **sm-getfinals**: sm → (state*)
  Purpose: To access the given machine's set of final states
- **sm-getrules**: sm → transitions
  Purpose: To access the given machine's transitions
- **sm-apply**: sm Word → Result
  Purpose: To apply the given machine to the given word
- **sm-showtransitions**: sm Word → trace
  Purpose: To return the trace of applying the given sm to the given word
- **sm-test**: sm natnum → (word result)*
  Purpose: To return the results obtained from applying the given machine to the given number of randomly generated words
- **sm-visualize**: sm [(state predicate)*] → (void)
  Purpose: To visualize the execution of the given machine and the value of the optional invariant state-predicates as a computation progresses.

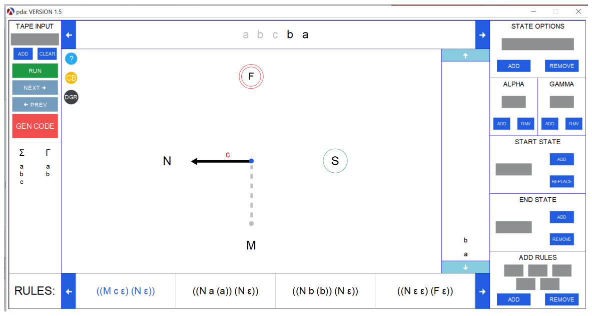
To illustrate the use of FSM consider implementing a pda to decide:

L = {wcw^r | w ∈ (a, b)*}

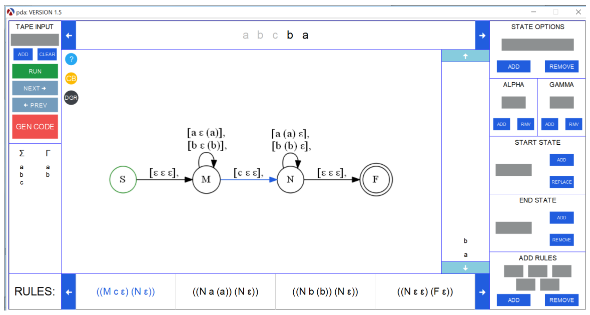
The FSM code for such a pda is[3]:

```
(define P (make-ndpda '(S M N F)
                      '(a b c)
                      '(a b)
                      'S
                      '(F)
                      `(((S ,EMP ,EMP) (M ,EMP))
                        ((M a ,EMP) (M (a)))
                        ((M b ,EMP) (M (b)))
                        ((M c ,EMP) (N ,EMP))
                        ((N a (a)) (N ,EMP)
                        ((N b (b)) (N ,EMP))
                        ((N ,EMP ,EMP) (F ,EMP)))))
```

[2]The dfa constructor takes an optional symbol, 'no-dead, to prevent the automatic addition of a dead state.
[3]EMP is FSM's constant for empty.



(a) Control View of P.



(b) Graph View of P.

Figure 2: Visualization Views for P.

```
(check-expect (sm-apply P '(c)) 'accept)
(check-expect (sm-apply P '(a b c b a)) 'accept)
(check-expect (sm-apply P '()) 'reject)
(check-expect (sm-apply P '(a b c a a)) 'reject)
```

P has four states: (S M N F). Its input alphabet is (a b c) and its stack alphabet is (a b). The starting state is S and the only final state is F. The transitions rules move the machine nondeterministically from S to M. In M, P pushes the read as and bs onto the stack until it encounters a c and moves to N. In N, P pops an element off the stack as long as it matches the read element. Nondeterministically, P moves from N to F. Upon reaching F, P accepts if all the input is consumed and the stack is empty. Otherwise, P rejects. The unit tests illustrate the expected behavior of P.

To invoke the FSM visualization tool on P, we may use:

```
(sm-visualize P)
```

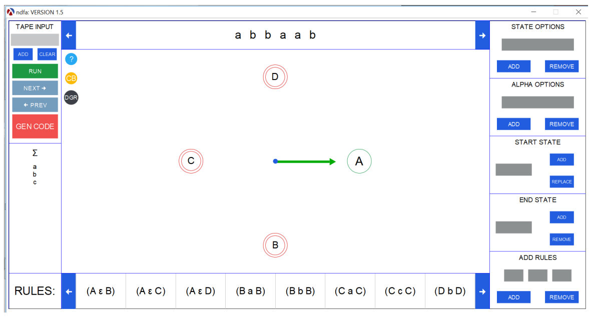
The FSM visualization tool is launched with P preloaded with no state invariants specified. Snapshots of P in the visualization tool are displayed in Figure 2. In Figure 2a, the control view of P is displayed. In Figure 2b, the state diagram view of P is displayed. Regardless of the view, the right column has input fields and buttons that allows the user to add to or remove elements from each component of the sextuple. The left column displays the input and stack alphabets and allows users to run the machine one step at a time using forward and backwards buttons, to render their edited machine as executable FSM code using code generation button, and to provide input to the machine using an input field and buttons to add to or clear the input tape. In the center, the top displays the input tape. The consumed input is faded out while the unconsumed input is not faded out. The bottom center displays the transition rules and highlights the last rule used. The center displays the machine and the stack. In Figure 2a, the states are organized in a circle, a solid arrow indicates the current state, and a dashed arrow indicates the previous state. The label of the solid arrow is the last consumed input element. The starting state is contained in a single circle while final states are contained in double circles. In Figure 2b, the states are organized as a graph or state diagram. The edges represent the transition relation. In both views the top-left corner has three circle buttons. The ? button takes the user to the FSM documentation page. The CB button toggles the colors for colorblind users. The DGR button flips the view from control view to graph view and vice versa.

Users find the Gen Code button extremely useful. This button generates the constructor code in FSM for that machine currently visualized. This constructor is saved in a separate file. In this manner, users can save the current state of their work and return to it later. This includes machines that do not build successfully. In this case, the constructor code contains a comment indicating that the defined machines does not successfully build.

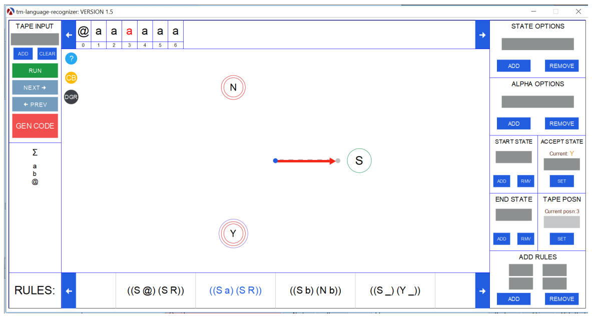
Depending on the type of machine being visualized, different features are added to the graphic. When visualizing a pushdown automaton, for example, the stack that is rendered on the right hand side of the screen and the stack alphabet, Γ, is displayed in the left column as shown in Figure 2b. Neither of these are displayed when visualizing a finite state automaton or a Turing machine. When a Turing machine is visualized, the tape displays the position of the head and an optional set tape position button and input field are made available to set the starting position on the tape.

Finally, in the graph view of a machine, each edge is an arrow that may have one or more labels. Each label represents a transition rule between the two nodes. For example, the arc on N in Figure 2b has two labels. The label [a (a) ϵ] corresponds to the rule ((N a (a)) (N ϵ)). If invariants are provided, in either view, an arrow indicating the current state turns green when the invariant holds and turns red when the invariant does not hold.

Figure 3a displays the control view of a finite state automaton. Observe that there is no stack nor stack alphabet displayed. The arrow indicating the current state, A, is green indicating that A's invariant holds in the current machine's configuration. Figure 3b displays the control view of a Turing machine. Observe that there is no stack nor stack alphabet displayed. Instead, the right column has the input field and the button to set the head's position on the tape in the TAPE POSN section. There is also an input field and a button to set the accept state when a Turing machine decides a language. Further observe, that the current position of the head is displayed by highlighting in red the contents in the input tape at the current position (an a in position 3 in this case). The tape position is also displayed TAPE POSN section. This is especially useful when the current tape position is blank. Finally, it is worth noting that when



(a) Control View of a Finite State Machine with A's Invariant Holding.



(b) Control View of a Turing Machine with S's Invariant Failing.

Figure 3: Finite State Automaton and Turing Machine Visualizations.

a Turing machine decides a language, the accept state is displayed inside a triple circle. When a Turing machine does not decide a language there is no final state enclosed in a triple circle.

4 BUTTONS

4.1 General Design

Buttons are an important aspect of many visualization GUIs because they allow the user to interact with the screen. Buttons in the FSM visualization tool are designed to behave like HTML5 buttons [12]. This means that buttons are responsive and have a color, text, size, position, and an on-click function. Being responsive means that the button must alert the user when an action is preformed. For instance, a button changes its shade on mouse events. There is an on-click function that defines the behavior of the button. This function is invoked when the button is clicked. For instance, an ADD button may add the contents of a input field to an internal data structure.

In the FSM visualization tool, a button is represented using the following structure:

```
(struct
  button (width  height     text     mode
          color  clickColor fontSize rounded?
          active location   onClick))
```

The structure definition automatically provides the programmer with a constructor, button, and with selectors for each field (e.g., button-onClick returns the function that is invoked when the button is clicked). The width and the height fields define the dimensions of the button. The text field is the label of the button, and mode is a symbol used to decide if the button is rendered outlined, solid, or transparent. The color field represents the color that is assigned to the button, while the clickColor is used to briefly highlight the button pressed, similar to how SCSS's[4] lighten function works [23]. The fontSize field specifies the size of the text displayed on the button, while the rounded? field is a Boolean that determines if the button should be a rectangle or a circle. The active field is a Boolean used to determine if the button is in an active state and the location field specifies the position on the screen at which to render the button. Last, the onClick field is the function that defines the behavior of the button. A button to add a state to a machine may (initially) be implemented as follows:

```
(define ADD-STATE
  (button 70
          25
          "Add"
          "solid"
          CONTROLLER-BUTTON-COLOR
          CONTROLLER-BUTTON-COLOR
          18
          #f
          #f
         (posn (- WIDTH 150) (- CONTROL-BOX-H 25))
          NULL-FUNCTION))
```

The NULL-FUNCTION performs no action and returns (void). This (default) value for the onClick field allows a programmer to experiment with the other features of a button before detailing its behavior.

A button to remove a state from a machine may be implemented as follows:

```
(define REMOVE-STATE
  (button 70
          25
          "Remove"
          "solid"
          CONTROLLER-BUTTON-COLOR
          CONTROLLER-BUTTON-COLOR
          18
          #f
          #f
         (posn (- WIDTH 110) (- CONTROL-BOX-H 25))
          NULL-FUNCTION))
```

Observe that many of the arguments to the constructor are the same as those used for the ADD-STATE button. This strongly suggests that an abstraction is needed.

[4]Sassy CSS or Sassy Cascading Style Sheets is a scripting language.

4.2 A Specialized Builder Pattern

In many GUIs, as exemplified above, the fields of different buttons are the same. This is a problem, because mundane repetitions are error-prone. This is a situation where an abstraction is ideally employed. The abstraction needs to identify the required and optional fields. When a button is constructed, the programmer only needs to provide the values for the required fields and for the optional fields to customize. A default value is used for every optional value not provided. This is precisely well-suited for the Builder pattern [5].

The classical builder pattern in an OO language creates a builder object. This object has a build method that allows you to create complex objects by separating the construction from its representation. Simplifications are achieved by allowing for the reduction of arguments that need to be provided. Default values are used for arguments not provided. The details of the default values are hidden by the implementation. Polymorphism allows to distinguish between different constructors to specialize different subsets of fields.

This section describes a variant of the Builder pattern developed for use with buttons and input fields. In contrast to the classical Builder pattern, this variant takes advantage of keywords, a Racket feature not present in many OO languages, to define a constructor that allows the programmer to choose which fields to specialize and which fields are initialized to default values.

A keyword argument is a function parameter that consists of an identifier followed by a expression [4]. One of the benefits of using keyword arguments is that they do not define a total ordering for the arguments provided. For instance, consider the following function:

```
     (define (builder #:param1 [param1 #t]
                      #:param2 [param2 #f])
       (and param1 param2))
```

This builder function has two keyword parameters: param1 and param2. Their default values, respectively, are true and false. A programmer may use builder, for example, in the following ways:

```
    (builder #:param2 #f
             #:param1 #f)
```

This expression provides false as the argument for both parameters and returns false.

```
    (builder #:param2 #t)
```

This expression provides true as the argument for param2 and uses the default value for param1. The expression evaluates to true.

```
    (builder)
```

This expression provides no arguments and both parameters are initialized to their default values. The expression evaluates to false.

Keyword arguments provide programmers with the ability to define constructors that only require values for fields that have to be specialized. This is useful to construct GUI buttons. In the FSM visualization tool, buttons only require the dimensions and the position of the button. All other button fields have default values that a programmer may customize. Using keyword arguments, the button builder may be defined as follows:

```
(define (button-builder
          width height loc
          #:text[text ""]
          #:color[color CTRL-BUTTON-COLOR]
          #:fntsize[size 18]
          #:round?[round #f]
          #:func[func NULL-FUNCTION]
          #:style[style "solid"])
  (button width height text  style
          color color  size  round
          #f    loc    func))
```

This definition states that the width, height, and loc are required and do not have a default value. All the other parameters are optional and have default values.

The job of an FSM developer is now simplified. For example, the ADD-STATE and REMOVE-STATE buttons above may now be defined as follows:

```
   (define ADD-STATE
           (button-builder
             70
             25
             (posn (- WIDTH 150) (- CONTROL-BOX-H 25))
             #:text "Add"))

   (define REMOVE-STATE
           (button-builder
             70
             25
             (posn (- WIDTH 110) (- CONTROL-BOX-H 25))
             #:text "Remove"))
```

Observe that only 1, not 6, customizable button characteristics need are provided.

5 INPUT FIELDS

Like buttons, input fields have a similar representation to input fields in HTML. This means that they have a background color, width, height, and position [13]. Like buttons, they are also reactive to allow for user interaction and contain two color fields in order to accommodate the tint factor. A textbox for an input field is defined as follows:

```
(struct textbox (width height     color orColor
                 text  charLength loc   active  func)
```

Using the builder pattern is a good design option for representing the above object in a OO language. Using our keyword-based builder pattern we can achieve the same effect. The textbox Builder is:

```
(define (textbox-builder
          width height loc
          #:text[text ""]
          #:color[color CTRL-TBOX-COLOR]
          #:orColor[orColor CTRL-TBOX-COLOR]
          #:limit[limit 18]
          #:active[round #f]
          #:func[func NULL-FUNCTION])
  (textbox width   height color
           orColor text   limit
           loc     active func))
```

A sample text box may be constructed as follows:

```
(make-textbox 150
              25
             (posn (- WIDTH 100) (- CONTROL-BOX-H 70))
           #:limit 5
           #:func addState)
```

Only two of six fields are customized: `limit` and `func`. The rest use default values for text boxes.

It is worth noting that input-field text boxes contain a procedure. This allows for an input field to respond to specified key strokes. For instance, a user may simple hit `Enter` when done typing in an input field.

6 SCROLL BARS

6.1 General Design

Scroll bars look deceptively simple, but do have some complexity behind them. The scroll bar code, for example, needs to create an appropriate rendering function. For example, the scroll bar for the rules of a machine needs to create a rendering function for either `fsa`, `pda`, or `tm` rules. This rendering function varies from one type of machine to another given that rule types vary among machine types.

Indeed, the most complex scroll bar in the `FSM` visualization tool is the one that displays the machines rules. The complexity rises from the varieties in machine rules. Recall that there are 3 types of machine rules:

```
FSA: (_ _ _)
PDA: ((_ _ _) (_ _))
 TM: ((_ _) (_ _))
```

The goal here is to build an interface that decouples the creation of the rendering function from the type of machine rules and that is scalable to new types of machines.

6.2 The Factory Pattern

The Factory Pattern is a good fit for this task. In an object oriented setting, a programmer creates a scroll bar rendering factory that returns the appropriate scroll bar rendering object. The appropriate object is dependent on the type of machine rule. By exploiting inheritance, the subclasses decide which type of scroll rendering object to create.

In a functional programming setting pattern matching may be used to achieve the same result. We may use pattern matching as a substitute to implement a factory method and functions as a substitute for child objects. Each branch in the pattern matching function is responsible for constructing the appropriate rendering function. Such a function looks like this:

```
(define (Scroll-Bar-factory lst-of-rules)
  (match (car lst-of-rules)
    [(list _ _ _)
     (FSA-Scroll-Bar lst-of-rules)]
    [(list (list _ _ _) (list _ _))
     (PDA-Scroll-Bar lst-of-rules)]
    [(list (list _ _) (list _ _))
     (TM-Scroll-Bar lst-of-rules)]
    [else (error "Invalid scroll bar factory")]))
```

Observe that the creation of the rendering function is decoupled from the type of rules being processed. A programmer may now call `SB-rendering-factory` regardless of the types of rules that may be displayed. Furthermore, this design is scalable. When a new machine type with a new transition rule type is added to `FSM` the above factory function is easily refined with a new pattern matching stanza.

To illustrate how our implementation mirrors a factory implementation in `Java`, the following is an outline of a scroll bar implementation:

```
abstract class ScrollBar {
  abstract void render(RuleList rules); }

class FsaScrollBar extends ScrollBar {
  void render(RuleList rules) { ... } }

class PdaScrollBar extends ScrollBar {
  void render(RuleList rules) { ... } }

class TmScrollBar extends ScrollBar {
  void render(RuleList rules) { ... } }

class ScrollBarFactory {
  enum mType { DFA, NDFA, PDA, TM, LR }
  public ScrollBar makeScrollBar(mType type) {
    switch (type) {
      case DFA:
      case NDFA:
        return new FsaScrollBar();
      case PDA:
        return new PdaScrollBar();
      case TM: return new TmScrollBar();
      default:
        throw new InvalidFactoryType(type);}}
```

To call the scroll bar factory the user writes code like this:

```
ScrollBarFactory factory = new ScrollBarFactory();
ScrollBar s = factory.makeScrollBar(mType.DFA);
s.render(rules);
```

This example shows how functions may be used in lieu of classes to achieve the same effect. In `FSM`, the factory returns a rule rendering function.

7 GRAPHVIZ LIBRARY

7.1 General Design

The creation of the graph-based rendering of a machine (i.e., a state diagram), as in Figure 2b, is implemented by interfacing with the `C`-based `Graphviz` library [1, 6, 24]. Interfacing with `Graphviz` is chosen because it is an open source visualization library that has been successfully used by other `DSLs` in `Racket` language family (e.g., [2]).

`Graphviz` uses the `DOT` language to represent graphs [8]. The following is a subset of the `DOT` language abstract grammar. Keywords are in bold font. Square brackets indicate optional items.

```
    graph ::= (graph | digraph) [ID] stmt-list
stmt-list ::= [ stmt [;] stmt-list ]
```

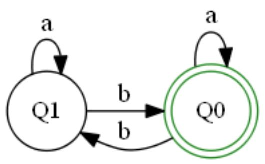


Figure 4: dfa for L = (bb)*.

```
     stmt ::= node-stmt | edge-stmt | attr-stmt
attr-stmt ::= (graph | node | edge) attr-list
```

The machine graphic displayed in Figure 4 is implemented in the DOT language as follows:

```
digraph G {
   rankdir="LR";
   Q1 [label="Q1", shape="circle", color="black"];
   Q0 [label="Q0", shape="doublecircle",
       color="forestgreen"];
   Q0 -> Q0 [label="a", fontsize=15];
   Q0 -> Q1 [label="b", fontsize=15];
   Q1 -> Q1 [label="a", fontsize=15];
   Q1 -> Q0 [label="b", fontsize=15];
}
```

The digraph's name is G and rankdir sets the direction of the graph layout: horizontally left to right. Nodes in the DOT language are represented as a symbol (e.g. Q0), while edges are represented as an arrow between two Nodes (e.g. Q0 -> Q1). Both nodes and edges have attributes in square brackets. The goal of FSM's interface with the Graphviz library is to generate the above DOT language representation of any machine built in FSM.

FSM requires specific formatting and customization in order to properly generate DOT code. For example, each machine type has a different syntax for transitions. Instead of having a custom DOT code generation routine for each new machine type added to FSM our goal is allow FSM developers to generate DOT code with little or no knowledge of the DOT language. The general design idea is to provide a graph generating function, graph->dot, that internally (hidden from the user) generates DOT code and interfaces with Graphviz.

In FSM, a graph is represented as a structure that has a name, a list of nodes, a list of edges, and a color (for color-blind mode). It is defined as follows:

```
(struct graph ([name]
               [node-list #:mutable]
               [edge-list #:mutable]
               [color-blind #:mutable]) #:transparent)
```

To make the FSM visualization tool more responsive the node and edge list are made mutable. This is required for faster rendering times. Every time the user presses the next and previous buttons to move forward or backward in the machine the graph needs to be recreated, converted to the dot-language, converted to a PNG file, and re-rendered on the screen. By using mutation we can essentially skip step 1 by just mutating the previous structure we have.

FSM represents a node as a structure that contains 4 fields: name, label, shape, and atb. The name and label represent the node name and its label. The shape field defines the geometric shape to render, and the atb is a map data structure that holds all the attributes for the node (e.g., color and shape). The structure definition is:

```
(struct node ([name]
              [label]
              [atb #:mutable]
              [type]) #:transparent)
```

A node may be defined as follows:

```
(define Q3
        (node 'Q3 'Q3 DEFAULT-NODE-ATTRS 'default))
```

The map structure allows to easily associate a Grahpviz attribute with a value. For example, this is the default map used when a node is created:

```
(define DEFAULT-NODE-ATTRS (hash
                              'color "black"
                              'shape "circle"))
```

The DOT code for Q3 is:

```
   Q3 [label="Q3", shape="circle", color="black"];
```

In FSM, an edge is represented as structure that has 3 fields: start-node (a symbol for the name of a node), end-node (a symbol for the name of a node), and atb (a map for the edge's attributes). The structure definition for a an edge is:

```
     (struct edge ([atb #:mutable]
                   [start-node #:mutable]
                   [end-node #:mutable]))
```

Constructing an edge labeled z between nodes A and B results in the following DOT code:

```
     A -> B [label="z", fontsize=15];
```

7.2 Builder Pattern

Section 3 discussed specialized Builder pattern used to create buttons and input fields. In this section, the classical Builder pattern is used to construct a graph. The use of the builder pattern is well-suited to hide the details of generating DOT code from a Racket graph structure. By doing so we are able to hide the logic behind generating DOT language code, allowing future developers to generate graphs without knowing the DOT language. Our builder interface for graph building (not image generation) only provides 4 functions to the programmer: graph-builder, add-edge, add-node, and graph->dot.

In Rust, for example, a template to instantiate such a Builder is:

```
     struct Graph {
       name: String,
       nodes: Vec<Node>,
       edges: Vec<Edge>, }

     impl Node {
      fn new(name: &str) -> Self {...}
      fn graph_to_dot(&mut self) -> PNG {...}
      fn add_edge(&mut self, ...) -> &mut self {...}
      fn add_node(&mut self, ...) -> &mut self {...} }
```

This allows the user, for example, to generate DOT code for a graph as follows:

```
let graph = Graph::new("Graph1")
    .add_node("A")
    .add_node("B")
    .add_edge("A", "a", "B")
    .graph_to_dot();
```

Observe that a key benefit obtained from using the builder pattern is readability. Even a reader not familiar with Rust can understand the above code.

In Racket, the builder pattern may be implemented using currying and message passing. The general skeleton for the graph builder is implemented as follows:

```
(define (graph-builder name)
  (define (add-node nname) ...)
  (define (add-edge from label to) ...)
  (define (graph->dot graph) ...)
  (define (graph-object message)
    (cond [(eq? 'add-node message) add-node]
          [(eq? 'add-edge message) add-edge]
          [(eq? 'gen-dot message) (graph->dot)]
          [else (error ...)]))
 graph-object)
```

Wrapper functions are written to present a cleaner interface to the user as follows:

```
(define add-node graph nname)
  ((graph 'add-node) nname))

(define add-edge graph from label to)
  ((graph 'add-edge) from label to))

(define (add-node graph nname)
  ((graph 'add-node) nname))

(define (graph->dot graph)
  (graph 'gen-dot))
```

The same graph generated using Rust above may now be generated in Racket in a remarkably similar manner:

```
(define graph (graph-builder 'dfa-graph))
(add-node graph 'A)
(add-node graph 'B)
(add-edge graph 'A 'a 'B)
(graph->dot graph)
```

The end result is that an FSM developer may now create a Graphviz graph without burdening themselves to learn the DOT language. This will reduce development time as support for new types of machines (e.g., finite state transducers) are added to FSM.

7.3 Adapter Pattern

The function graph->dot must convert any type of FSM machine into a DOT language representation. This means different FSM types must be converted to a single type that is used to generate the needed DOT syntax. This is a scenario that calls for using the adapter pattern. The adapter pattern is used to create an interface where the converters for each machine type are used together without modifying the code for any of the converters.

The FSM graph adapter converts any machine's rules into a string representation for the label above an edge in the graph image generated by Graphviz. In a functional programming setting, an adapter may be implemented using higher-order functions and pattern matching. The adapter takes as input any type of machine and returns the converted rules as follows:

```
(define (graph-adapter a-machine)
 (let ((rules (sm-getrules a-machine)))
  (match (car rules)
    [(list _ _ _)
     (map fsa-rule->string rules)]
    [(list (list _ _ _) (list _ _))
     (map pda-rules->string rules)]
    [(list _ _) (list _ _)
     (map tm-rule->string rules)]
    [else (error “Unsupported data type”)])))
```

Observe that a developer only needs to define how to generate a string from a single rule. For this, there is no knowledge of the DOT language required. Further observe that support for new types of machines are easily added without requiring a major code rewrite. All that is required is the addition of a new stanza in the match expression. If two rule types are the same then Racket's guard clauses may be used to distinguish between them. For example, consider a dfa variant where transitions consume numbers instead of symbols. To handle this special cause we use a guarded pattern as follows:

```
(define (graph-adapter a-machine)
 (let ((rules (sm-getrules a-machine)))
  (match (car rules)
    [(list _ t _) #:when (number? t)
     (map special-fsa-rule->string rules)]
    [(list _ _ _)
     (map fsa-rule->string rules)]
    ...)))
```

In a match clause #:when is used to guard a match. The expression after #:when must hold in order to match. Its important to note that the guarded match must be placed before an unguarded match. Otherwise, control will never reach the guarded case.

It is worth observing that the adapter pattern is used throughout the implementation of the FSM visualization tool. Another place where the Adapter pattern is used is in the implementation of the NEXT $\longrightarrow$ and $\longleftarrow$ PREV buttons to step through a computation. For example, when using the control view of a machine the image displayed (not generated using Graphviz) depends on the machine type. In this case, the adapter matches the machine type to create the image of the current machine configuration.

8 SUMMARY OF DESIGN PATTERN IMPLEMENTATIONS

The practical lessons to take away from this article are the implementation strategies for the Builder, Factory Method, and Adapter design patterns in a functional programming setting. This section summarizes the implementation strategies and provides corresponding function templates.

For the Builder pattern first identify the fields that a programmer may manipulate and provide wrapper methods for a clean interface. A template for the Builder pattern is:

```
(define (X-builder p_0 ... p_{n-1})
  (define (handler-message_0 ...) ...)
      ⋮
  (define (handler-message_{k-1} ...) ...)
  (define (get-message message)
     (cond [(eq? message message_0)
            handler-message_0]
                ⋮
           [(eq? message message_{k-1})
            handler-message_{k-1}]
           [else (error ...)]))
 get-message)

;; the wrapper functions
(define (wf_0 X ...) ((X message_0) ...))
     ⋮
(define (wf_{k-1} X ...) ((X message_{k-1}) ...))
```

For our specialized Builder, first define a structure that contains all the needed fields. Then identify all fields that have a default value and make them optional using keyword parameters. The template for our specialized Builder pattern is:

```
(struct K (field_0 ... field_{n-1}))

(define (K-builder ;; required fields
                   field_d field_c ... field_b
                   ;; optional fields
                   #:fieldj[fieldj default-val_j]
                   #:fielde[fielde default-val_e]
                       ⋮
                  #:fieldm[fieldm default-val_m])
      (K field_0 ... field_{n-1}))
```

For the Factory Method pattern write a function distinguishes between the varieties of the data to be processed. For each variety, develop an auxiliary function that constructs the required instance for the type. The template for the Factory Method pattern is:

```
(define (factory data)
  (match ...data)...
    [variety_0 (create-variety_1 ...data...)]
        ⋮
    [variety_{i-1} (create-variety_{i-1} ...data...)]
    [else (error ...)]))
```

For the Adapter first identify the data varieties that need to be converted and writes an adapter function for each. Then develop a main adapter function that dispatches on the variety that needs to be converted. The template for the Adapter pattern is:

```
(define (type_0-adapt ...) ...)
     ⋮
(define (type_{k-1}-adapt ...) ...)

(define (adapter data)
  (match data
    [type_0   (type_0-adapt ...)]
        ⋮
    [type_{k-1} (type_{k-1}-adapt ...)]
    [else (error ...)]))
```

9 RELATED WORK

Design patterns in functional programming have sometimes been categorized as unnecessary because they only exist due to missing features in a programming language [25]. Some functional programmers may even argue that native language features like higher-order functions, closures, and pattern matching are better alternatives to design patterns. This, of course, ignores that design patterns capture useful and recurring programming abstractions–just like higher-order functions, closures, and pattern matching. Whether polymorphism and inheritance or higher-order functions and pattern matching is used to implement a design pattern, the fact remains that an abstraction is always useful. First, it makes it easier to communicate to others how a problem is solved–a major goal of programming [3]. Second, as any abstraction, the use of a design pattern facilitates future refinements without requiring major code rewrites. In this article, three design patterns (Builder, Factory Method, and Adapter) have been used to highlight these advantages. Design patterns are not used for the sake of using design patterns just like higher-order functions are not used for the sake of using higher-order functions. They are used to improve readability and scalability and to make refinements easier. We exploit functional programming features to provide similar design pattern abstractions.

Many functional programmers, nonetheless, also argue that there are many functional design patterns. For example, Category Theory [11, 21] is considered a source of many design patterns in functional programming. One of the functional programming languages that has pioneered abstractions based on Category Theory is `Haskell` [25]. For example, the `Functor` class abstracts the `map` operation. For instance, the Functor class:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

may be used may be used to abstract `map` as follows:

```
instance Functor [] where
  fmap f [] = []
  fmap f (x:xs) = f x : fmap f xs
```

Observe that `fmap` is a *map pattern* that works on an arbitrary `Functor`, not just lists. That is, it implements polymorphism. In the same vein of abstraction, monads may also be used to implement design patterns. For example, the `remote monad design pattern` makes remote procedure calls more efficient [7]. Although the use of abstractions based on Category Theory are now common in many functional programming languages (e.g., `Haskell` [17], `ML` [19], `Racket`[20] and `Scala` [25]), their use is not universal. Many programmers find them too difficult to understand and maintain. We hypothesize that starting with `OO` design patterns, as described in this article, may serve as an effective stepping stone to abstractions based on Category Theory.

10 CONCLUDING REMARKS

This article describes how three OO design patterns are used in a functional programming setting to simplify and to make more readable code. The setting is the development of the FSM visualization tool. This tool assists users to design and implement state machines such as finite state machines, pushdown automata, and Turing machines. Button and input field implementation benefit from using a customized variant of the Builder pattern. The implementation of scroll bars benefit from using the Factory Method pattern. The implementation of an interface for the Graphviz library benefits from using the Builder and the Adapter patterns. These design patterns exploit hallmarks of functional programming like higher-order functions, pattern matching, and keyword parameters in lieu of objects, polymorphism, and inheritance. The result is an implementation that developers find straightforward to understand and refine. The article presents template for the design patterns discussed to ease their use by others.

Future work includes exploiting the implementation based on design patterns to extend FSM. Such extensions include support for finite state transducers and multitape Turing machines. Future work also includes extending the FSM visualization tool to support the derivation of words using regular, context-free, and context-sensitive grammars. This support will build on FSM's interface for grammars much like the current version of the FSM visualization tool builds on FSM's interface for state machines. Finally, future work also includes developing elegant implementations in a functional programming setting for all 23 OO design patterns.

ACKNOWLEDGMENTS

The authors thank Matthias Felleisen for suggesting that the control view of machines can coexists with the graph view of machines inside the FSM visualization tool. This led to the development of the interface with Graphviz. The authors also thank Isabella C. Felix and Sena N. Karsavran for their support as research assistants. The support provided by both The Department of Mathematics and Computer Science and The Office of the Dean of the College of Arts and Science of Seton Hall University that made the development of this work possible is also noted and appreciated.

REFERENCES

[1] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and Dynagraph–Static and DynamicGraph Drawing Tools. In *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

[2] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009.

[3] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barsilay, Jay McCarthy, and Sam Tobin-Hochstadt. A Programmable Programming Language. *Commun. ACM*, 61(13):62–71, March 2018.

[4] Matthew Flatt, Robert Bruce Findler, and PLT. The Racket Guide. https://docs.racket-lang.org/guide/lambda.html#%28part._lambda-keywords%29, last accessed 2020-08-10.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[6] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Sodtware Practice and Experience*, 30(11):1203–1233, 2000.

[7] Andy Gill, Neil Srulthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. The Remote Monad Design Pattern. *SIGPLAN Notices*, 50(12):59–70, August 2015.

[8] Graphviz - Graph Visualization Software. *The DOT Language*. https://graphviz.org/doc/info/lang.html, last accessed 2020-08-10.

[9] Michael T. Grinder. A preliminary empirical evaluation of the effectiveness of a finite state automaton animator. *SIGCSE Bull.*, 35(1):157–161, January 2003.

[10] Rohit Joshi. *Java Design Patterns*. Java Code Geeks, 1st edition, 2015.

[11] Tom Leinster. *Basic Category Theory*. Cambridge University Press, 2009.

[12] MDN web docs. *<button>: The Button element*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button, last accessed 2020-13-10.

[13] MDN web docs. *<input>: The Input (Form Input) element*. https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input, last accessed 2020-13-10.

[14] Marco T. Morazán and Rosario Antunez. Functional automata–formal languages for computer science students. In James Caldwell, Philip K. F. Hölzenspies, and Peter Achten, editors, *Proceedings 3^{rd} International Workshop on Trends in Functional Programming in Education*, volume 170 of *EPTCS*, pages 19–32, 2014.

[15] Marco T. Morazán and Josephine A. Des Rosiers. FSM error messages. *EPTCS*, 295:1–16, 2019.

[16] Oracle. Interface URLStreamHandlerFactory. https://docs.oracle.com/javase/8/docs/api/java/net/URLStreamHandlerFactory.html, 2020. last accessed 2020-08-10.

[17] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[18] Mukesh D. Parsana, Jayesh N. Rathod, and Jaladhi D. Joshi. Using Factory Design Pattern for Database Connection and Daos (Data Access Objects) With Struts Framework. *International Journal of Engineering Research and Development.*, 5(6):39–47, December 2012.

[19] Lawrence C. Paulson. *ML For the Working Programmer*. Cambridge University Press, USA, 2nd edition, 1996.

[20] PLT. Interfaces. https://docs.racket-lang.org/functional/interfaces.html, 2020. last accessed 2020-08-10.

[21] Emily Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2017.

[22] Susan H. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., USA, 2006.

[23] SCSS. *SCSS Implementation Guide*. https://sass-lang.com/documentation/modules/color, last accessed 2020-08-10.

[24] Mihalis Tsoukalos. An Introduction to Graphviz. *LINUX Journal*, 2004. https://www.linuxjournal.com/article/7275, last accessed 2020-08-10.

[25] Dean Wampler and Alex Payne. *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly Media, Inc., 2nd edition, 2014.

[26] Timothy M. White and Thomas P. Way. jFAST: A Java Finite Automata Simulator. *SIGCSE Bull.*, 38(1):384–388, March 2006.

Functional Programming and Interval Arithmetic with High Accuracy

Filipe Varjão
CIn/UFPE
frgv@cin.ufpe.br

Abstract

When working with floating-point numbers, the result is only an approximation of real value, and errors generated by rounding or by the instability of the algorithms can lead to incorrect results. We can?t affirm the accuracy of the estimated answer without the contribution of error analysis. Interval techniques compute an interval range, with the assurance the answer belongs to this range. Using intervals for the representation of real numbers, it is possible to control the error propagation of rounding or truncation, between others, in numerical computational procedures. Therefore, intervals results carry with them the security of their quality. In this paper, we describe a high accuracy tool ExInterval which provides types and functions for Maximum Accuracy Interval Arithmetic, following the standard convention IEEE 754 and 854 for single and double-precision, interval arithmetic is a mathematical tool to solve problems related to numerical errors.

ACM Reference Format:
Filipe Varjão. 2020. Functional Programming and Interval Arithmetic with High Accuracy. In *Proceedings of IFL 2020: Symposium on Implementation and Application of Functional Languages (IFL 2020).* ACM, New York, NY, USA, 1 page. https://doi.org/??

General Deforestation Using Fusion, Tupling and Intensive Redundancy Analysis

Anonymous Author(s)

Abstract

Fusion and Tupling are well-known optimizations that have been used by programmers, and also certain compilers over the years. While each of these transformations can improve performance when used independently, prior research has suggested that combining them together can be beneficial, as each transformation can help the other to optimize the program further. Despite this, we're not aware of any work that provides empirical evidence to demonstrate the benefits of this technique.

We propose a deforestation transformation that also combines fusion, tupling, but along with a novel *redundancy analysis*, and which is guaranteed to terminate and may be incorporated in compilers. Redundancy analysis cleans up some of the artifacts introduced by the fusion and tupling, and increases their effectiveness by exposing more optimization opportunities. We also provide a practical implementation of our deforestation transformation, and show that it is able to achieve significant speedup over unfused programs that contain some fairly complicated traversals.

1 Introduction

Fusion is a classic optimization that can eliminate intermediate structures that arise frequently in functional programs. When performed correctly, it reduces data traversal overhead and memory usage, resulting in significant speedup. In general, the goal of fusion is to take functions `f₁ :: A → B` and `f₂ :: B → C`, and produce a fused function `f₁₂ :: A → C`. The simplest illustrative example is a program like `(map f₂ (map f₁ ls))`, that maps functions f_1 and f_2 over a list `ls`. This program applies `(map f₁)` on its input, and generates an intermediate structure that is then consumed by `(map f₂)` to produce the final output. If we instead use a fused function, `(f₂ ∘ f₁)`, we can directly generate the output without creating an intermediate structure.

Fusion/deforestation transformations can be roughly classified into two groups: combinator based techniques, and general fusion. A combinator based technique relies on certain predefined *combinators* that have well defined compositional behavior, and a set of rewrite rules that can fuse the functions that use them. Such a technique is extremely effective when the target of fusion is a program that uses simple data structures such as lists or trees, for which many common operations can be expressed by composing fusable combinators. For example, shortcut fusion [12, 14, 19, 19] and stream fusion [10, 11] are both combinator based, and have been successfully used in modern compilers such as the Glasgow Haskell Compiler.

We classify these techniques under *shallow fusion*, because they do not reason about the definitions in the input program, or about the combinators themselves, and only rely on rules that are given by the programmer. They greatly simplify the problem, at the cost of generality.

General *deep* fusion techniques can directly fuse recursive functions without baking in knowledge about primitive combinators, but they have proved difficult to automate in a practical way [3], and, as such, they have remained comparatively unexplored for the last two decades. The most popular such approach is Wadler's deforestation [22], which guarantees programs in *treeless* form can be fused safely. Treeless form, however, is very restrictive: functions must be linear, and no intermediate data structures can be created during a single function evaluation—ensuring termination and complexity preservation. In his conclusion, Wadler states: "Further *practical* experience is needed to better assess the ideas in the paper".

Chin et al. [9] have refined Wadler's deforestation in an attempt to remove these syntactic restrictions. Their *extended-deforestation* algorithm also has some syntactic restrictions, but they are more fine grained that Wadler's. For example, consider a function `f :: List → List → List`, which uses its first argument non-linearly. Because of the non-linear argument, `f` is not in treeless form and Wadler's algorithm won't consider it for fusion at all. But the extended-deforestation algorithm will try to fuse `f` with the sub-terms passed it as its second argument. For example, given a call-site like `(f g h)`, `f` and `h` might be fused, as long as `h` obeys certain syntactic criteria. Thus, the extended-deforestation algorithm is applicable to a wider range of input programs than Wadler's deforestation, but certainly not all of them. In Section 7, Chin et al. [9] state: "The syntactic criteria proposed in this paper are based on safe approximations. They do not detect all possible opportunities for effective fusion, merely a sub-class of them".

Tupling is another well-known optimization [7, 8]. It eliminates multiple traversals of the same structure, each of which runs a different computation, by combining them into a single traversal that returns the results at once, using a tuple.

IFL'20, September 2–4, 2020, Virtual.
.

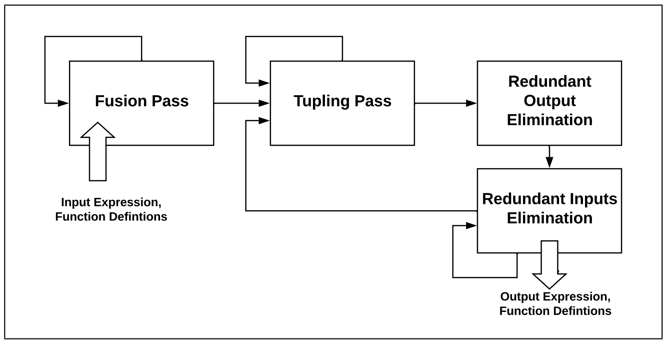

Fusion Pass
Tupling Pass
Redundant Output Elimination
Redundant Inputs Elimination
Input Expression, Function Defintions
Output Expression, Function Defintions

Figure 1. High-level structure of the deforestation transformation.

That is, it transforms functions such as `f₁ :: A → B` and `f₂ :: A → C` into a function `f₁₂ :: A → (B, C)`. Chin et al. studied the relationship between tupling and fusion, and suggested that tupling may increase the applicability of fusion [5]. Specifically, fusion may introduce multiple traversals of the same structure if it is performed on non-linear terms, but these additional traversals can be eliminated by a subsequent tupling. Two different transformations that use combinations of fusion and tupling [5, 6] have been suggested so far so far, but none of them have been implemented or evaluated yet.

Like Chin et al., we propose a deforestation transformation that uses a combination of fusion and tupling. Moreover, we also use a novel redundancy analysis which cleans up some of the artifacts introduced by the previous transformations, and increases their effectiveness by exposing more optimization opportunities. With respect to the syntactic restrictions, we attack the problem differently. Instead of having any restrictions on input programs, our transformation uses a fuel parameter to ensure termination.

This paper makes the following contributions:

- We propose a deforestation transformation that combines fusion, tupling, and intensive redundancy analysis, and which is guaranteed to terminate and may be incorporated in practical compilers.
- We implement and evaluate our transformation in a real compiler that operates on a first-order language with a Haskell backend, showing significant speedups on a large set of programs. This includes difficult-to-fuse examples such as rendering tree-structured documents (like HTML).
- We introduce a static analysis called "Intensive Redundancy Analysis" that is crucial for eliminating unnecessary work introduced by fusion and tupling for complicated programs.
- We show that the general, *deep* fusion is still a promising technique, one that—with good engineering—can fuse complicated programs that cannot be fused otherwise.

2 Overview

Figure 1 shows the high level structure of our deforestation transformation. It takes in a program consisting of data and function definitions as input, and optimizes it by using a combination of fusion, tupling and redundancy analysis. Consider the program given in Figure 2. It contains two functions, `prefixSum` and `shift`, that operate on a list of integers. `prefixSum` generates a new list in which each element at index `i` is the sum of elements at indices `i` to `n` in the old one. And `shift` moves all elements to the left by dropping the first one and adding a zero at the end of the list. This example might be somewhat contrived, but it highlights several properties of our transformation.

```
data List = Sing Int | Cons Int List

head :: List → Int
head ls = case ls of
            Sing x     → x
            Cons x xs  → x

shift  :: List → Int
shift ls = case ls of
             Sing x     → Sing 0
             Cons x xs  →
                 let x'  = head xs in
                 let xs' = shift xs in
                 Cons x' xs'

prefixSum :: List → List
prefixSum ls = case ls of
            Sing x     → Sing x
            Cons x xs  →
               let xs' = prefixSum xs in
               let x'  = head xs' in
               let x'' = x + x' in
               Cons x'' xs'

main = let ls  = MkList in
       let ls' = prefixSum ls in
       shift ls'
```

Figure 2

The transformation starts with the fusion step; it fuses a composition of two functions into a single one, and then continues to analyze this newly generated function. This process continues until a fix point is reached, or until the transformation runs out of fuel. For the example program, fusion first combines the functions `shift` and `prefixSum` to produce a function `shift_sum`. Next, it analyses `shift_sum` and observes that `head` and `sum` can be fused too. So it runs one more time and the generates the program shown in Figure 3a. Now there are no more opportunities for fusion, so it halts.

```
shift_sum :: List → List
shift_sum ls = case ls of
                 Sing x    → Sing 0
                 Cons x xs →
                   let hs = head_sum xs in
                   let ss = shift_sum xs in
                   Cons hs ss

head_sum :: List → Int
head_sum ls = case ls of
                Sing x    → x
                Cons x xs →
                  let hs = head_sum xs in
                  hs + x

main = let ls = MkList in
       shift_sum ls
```

(a)

```
shift_sum_T_head_sum :: List → (List, Int)
shift_sum_T_head_sum ls =
  case ls of
    Sing x→
      let o1 = Sing 0 in
      (o1, x)
    Cons x xs →
      let (p0,p1) = shift_sum_T_head_sum xs in
      let o1 = Cons p1 p0 in
      let o2 = p1 + x in
      (o1, o2)

main = let ls = MkList in
       let (ls', _) = shift_sum_T_head_sum ls in
       ls
```

(b)

Figure 3. The program on the left shows the result of fusion operating on the program given in Figure 2. The one on the right shows the result of running tupling on the program on the left.

Note that the fused function `shift_sum` calls `head_sum` for every element in the list, and `head_sum` traverses the complete list again. This has worse runtime complexity, $O(N^2)$, compared to the original $O(N)$! Fortunately, they both traverse the exact same list, `xs`, and tupling can combine these two functions! Fusion only eliminates intermediate structures in the computation; there still might be multiple functions that traverse the same structure, and combining them into a single traversal will further optimize the program.

Tupling analyses each function that is generated during the fusion step. Like fusion, it is also performed recursively. As mentioned before, `(shift_sum :: List → List)` and `(head_sum :: List → Int)` both traverse the same list. So these functions are tupled together into a function that traverses the list only once, and returns a tuple `(List, Int)`. Figure 3b shows the output of running tupling on the program generated by fusion in the previous step. Also, any repeated computation is eliminated using a simple common subexpression elimination (CSE) pass that is integrated with tupling. As it turns out, CSE can sufficiently simplify the program given in this example, but it is not always the case. Sometimes, an intensive redundancy analysis followed by several cycles of tupling might be needed.

As its name suggests, redundancy analysis eliminates redundant work. The process consists of two passes: eliminating output and input redundancy respectively. The first pass eliminates outputs of functions that appear at different indices in a tuple but always have the same value. In this case, one component of the tuple can be dropped and sometimes the tuple is eliminated completely. This is a step in the right direction by itself, but more importantly, it also enables more optimizations at the call sites of such functions where the fact that the two outputs are same can be leveraged to further eliminate redundant traversals and expressions. The next pass eliminates unused inputs of functions. It's unlikely for a programmer to have written functions that have such inputs, but functions generated during fusion and tupling often have this property, and this pass gets rid of them. Eliminating redundancy can allow more functions to be tupled, and hence tupling runs back-to-back with redundancy analysis until the process converges. Finally, a simplification pass runs several times during the transformation that performs common sub-expression and simple dead code elimination.

2.1 Non-linearity

In this paper, we borrow our notion of linearity from Wadler's work [22]: a term is said to be linear if no variable appears in it more than once. There's a special extension for case expressions: a variable that occurs in the scrutinee may not also appear in a branch, but a variable is allowed appear in more than one branch. For example, a function `foo` defined as `(foo x y = case MkFoo1 of MkFoo1 → y ; MkFoo2 → y)` is said to be linear even though it doesn't use `x` at all and `y` appears syntactically twice. The treeless form enforces that terms in the functions being fused are always linear, and this guarantees that no repeated work gets introduced during the fusion process. But our transformation cannot make this guarantee.

In our transformation, fusion generates programs in which all functions operate directly on the input tree to generate some part of the output tree. If the original functions are not linear it's possible that there would be multiple points in the fused program where the input tree is consumed, and each of them can become a traversal. In the example

$K \in$ Data Constructors, $\tau \in$ Types,

$f \in$ Function names $x, v \in$ Variables

Top-Level Programs	top	$::= \overrightarrow{dd}\ ; \overrightarrow{fd}\ ; e \mid e'$
Type Scheme	ts	$::= \overrightarrow{\tau} \to \tau$
Datatype Declarations	dd	$::= \texttt{data}\ \tau = \overrightarrow{K\ \overrightarrow{\tau}}$
Function Declarations	fd	$::= f : ts\ ; fb$
Function Definition	fb	$::= f\ \overrightarrow{x} = \texttt{case}\ \overrightarrow{x_1}\ \texttt{of}\ \overrightarrow{pat}$
Pattern	pat	$::= K\ (\overrightarrow{x : \tau}) \to e \mid e'$
Let Expression	e	$::= \texttt{let}\ x : \tau = e'\ \texttt{in}\ e \mid e'$
Leaf Expressions	e'	$::= f\ \overrightarrow{v} \mid K\ \overrightarrow{v} \mid v$

Figure 4. Language definition

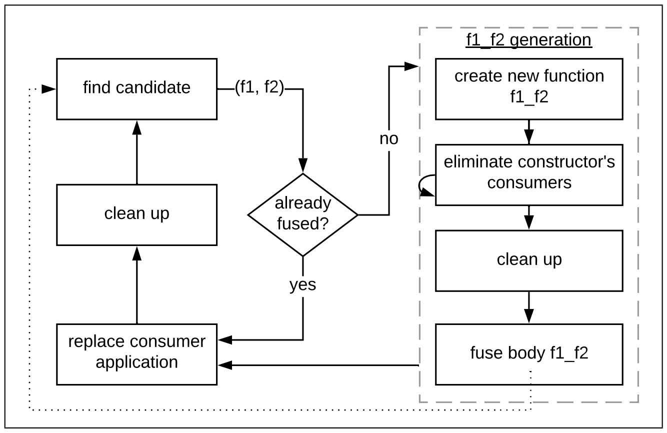


Figure 5. Fusion.

above, shift is non-linear—it consumes the tail of the list once in a recursive call, and again in a call to head. After fusion, those two points of consumption became separate traversals, shift_sum and head_sum, and this causes it's runtime complexity to become $O(N^2)$. As we show above, tupling cleans up cleans up any unnecessary work that fusion may have introduced. Furthermore, it brings work from different traversals closer to each and makes it easier to detect and eliminate redundancy. In the final tupled function, the list is consumed only once, and the runtime complexity is $O(N)$ again, with no intermediate structures.

2.2 Non-termination and Non-linearity, together

Our deforestation transformation can handle non-termination and non-linearity in isolation just fine. But when they occur in a program simultaneously, the program generated by our transformation can have worse runtime complexity compared to the original. In such cases, the fused program may contain redundant work, and the state of the program might be such that it makes it difficult for the subsequent tupling transformation to eliminate the redundant work. We plan to address this problem in the future.

3 Design

In this section we give details of all parts of the transformation: fusion, tupling and redundancy analysis. All of them operate on a monomorphic, first order, functional programming language described by the grammar shown in Figure 4. We use the notation $\overrightarrow{x}$ to denote a vector $[x_1, \ldots, x_n]$, and $\overrightarrow{x_i}$ to denote the item at position i. To simplify presentation, primitives are dropped from the formal language. It permits recursive data types, but since it is strict and side-effect free, it doesn't admit cyclic data structures.

A program consists of a set of data definitions, function definitions, and the main expression. Note that the function body has to be a case expression that destructs the first argument, which is assumed to be the dominant, traversed input.

The branches of the case expressions are sequences of flattened let expressions ending with leaf expressions – either a variable, a function application, or a constructor expression with variable arguments. This presentation is a simplified version of the actual language used in the implementation that supports literals and primitives, and expressions need not be already flattened. Also, the assumption that a function's first argument has to be the input that's traversed can be avoided by having the programmer provide annotations.

3.1 Fusion

The goal of the fusion pass is to eliminate intermediate structures in the program. Figure 5 shows the structure of the fusion pass. It takes an expression and function definitions as input and returns a new fused expression and a possibly-larger set of function definitions.

The pass starts by identifying a fusion candidate in the processed expression. To this end, it maintains a *definition-use* table that tracks variables that are bound to function applications, and their consumers. Specifically, a candidate for fusion (f_1, f_2) is a pair of functions that satisfies the following pattern:

```
let y = f1 x ⋯ in
 ⋯ f2 y ⋯
```

In such a case, a new function f_2_f_1 that represents the composition is generated. Generating the fused function draws on previous fusion techniques [9, 22]. However it's slightly altered to handle non-treeless expressions, and preserve the invariant that every function is a single case expression. This invariant makes the implementation of the optimization easier and more regular.

To illustrate the fusion process consider the previous example from Figure 2. Functions prefixSum and shift are candidates for fusion. In such cases, prefixSum is referred to as the producer, and shift as a consumer. As described in Figure 5

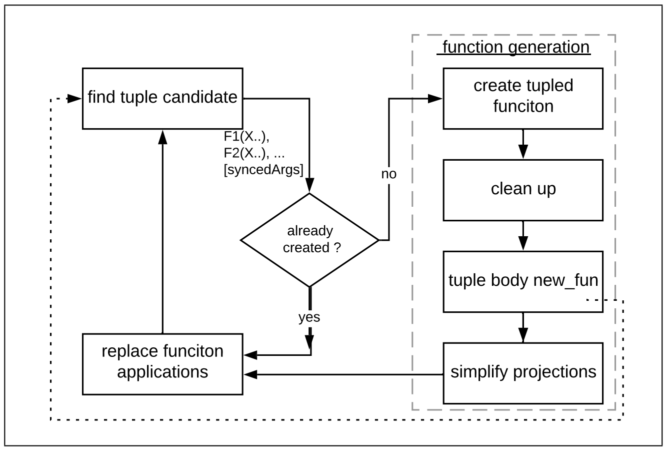


Figure 6. Tupling.

the first step is to create a new function `shift_sum` that represents the composition of `shift` and `prefixSum`. It's created according to the following rules:

1. The output type of fused function is the output type of the consumer function.
2. The input type of fused function is a concatenation of the inputs of producer and the consumer excluding the first input of the consumer function.
3. The body of the fused function is the body of the producer with the consumer function applied to the output of every branch in the producer.

Next we partially-evaluate the body of the generated function with a pass that eliminates constructor consumers (similar to "case of known constructor"). This pass uses its definition-use table to look for patterns of the form: `(let x = (K ..) in … f x)`

For each such pattern, the function application is replaced with the branch in f that corresponds to the constructor K after the appropriate instantiations. This sub-pass will keep running on the function until there are no further applications to known constructors. After the new, fused function is generated, a clean up pass will run, that removes common sub-expressions and unused let bindings. Fusion is then performed recursively on the body of the new function.

3.2 Tupling

Tupling combines traversals that traverse the same structure and bring computations closer to each other. Tupling is performed after fusion to eliminate redundant work that is introduced during fusion. For tupling, we extend the intermediate language to include operations on tuples. New expression forms are added for constructing tuples and projecting elements from tuples, plus a new product type.

Figure 6 summarizes the tupling pass, which begins by finding a tupling candidate. A candidate is a set of *independent* function applications that all traverse the same input (have the same first argument in our language).

By independent we mean that none of them directly nor indirectly consumes the other. For example in the code below, calls to f_1 and f_2 are not tupleable because f_2 indirectly consumes f_1 through the intermediate variables y.

```
let x = f₁ tree in
let y = x + 1 in
let z = f₂ tree y in
…
```

For each candidate, a tupled function is generated. The tupled function is generated according to the following rules:

1. The input type of the tupled function is the type of the traversed tree followed by the remaining inputs of each of the participating functions.
2. The output type of the tupled function is a tuple of the output types of the participating functions, with nested tuples flattened.
3. The body of the tupled function is a single case expression that destructs the traversed tree. For each case branch the body of the corresponding branch in each of the tupled functions is bound to a variable and a tuple of those variables is returned.

Next, this new function is optimized through a cleanup pass. At the end of the process, the original function applications that are tupled are eliminated by replacing the first application with the tupled function and the rest with projections to extract the corresponding output.

3.3 Redundancy Analysis

Following tupling, redundancy analysis is performed to further optimize the tupled functions. The optimizations performed during this pass are classified into two types; *redundant outputs* and *redundant inputs*. Each is described in detail in this section.

Note that as illustrated in Figure 1, tupling is performed again after redundancy analysis, since eliminating redundancy can enable more tupling to be done by eliminating some dependences that prohibit tupling.

3.3.1 Redundant outputs

The redundant outputs pass eliminates outputs of functions that appear at different indices in the tupled output but always have the same value.

Function f_t shown in Figure 7 illustrates such redundancy in its simplest form. The output of f_t is always the same for positions 0 and 1. We will use the notation $f_t^{0=1}$ to refer to that property throughout the section.

Different circumstances can cause such redundancy to originate. For example, consider tupling two fused functions, $f_x f_y$ and $f_x f_z$. If the result of f_x does not depend on f_z nor on f_y, then both functions would have the same output. Eliminating such redundancy is important for two reasons. First, if this function is called recursively, then the memory and runtime overhead of creating such a tuple is eliminated. The second

```
fₜ :: List → (Int, Int)
fₜ ls =
  case ls of
    Sing x → (0, 0)
    Cons x xs →
        let ret = depth xs in
        (ret, ret)
```

Figure 7. An example for which it is easy to syntactically eliminate redundant outputs.

important effect of such elimination is that it allows more optimizations on the caller side by leveraging the fact the the two outputs are the same to further eliminate redundant traversals and expressions.

Redundant output elimination consists of 3 steps:

1. Identify redundant outputs.
2. Create a new function with redundant outputs eliminated.
3. Fix callers to call the new function and optimize them.

For each tupled function, each output position is checked for redundancy, then the function is rewritten to eliminate any discovered redundancy. This is done by updating the function's return type and the tuple expressions in tail position of the function body. Next, all call sites of the functions are updated such that projections of the redundant element are switched to projections of the retained element. Steps two and three are straightforward rewrites. However, the first step, identifying the redundant outputs, is not always as trivial as it was in the previous example.

Inductive Redundant Output Analysis. In the previous example, it was easy to identify that the outputs at positions 0 and 1 of the return value are the same, by simply inspecting the output of each branch. Of course, the process is not always that simple; due to mutual recursion and complicated traversal patterns, a more rigorous inductive analysis is needed. Consider the example shown in Figure 8, which contains two mutually-recursive functions, f_1 and f_2.

Looking closely at those two functions, we observe that the second output of f_1 and f_2 is redundant and matches the first output, but how can we verify that soundly and systematically?

We want to check if f_1 always returns the same output at indices 0 and 1. In other words, if $f_1^{0=1}$ is satisfied. We can do that by checking the output at each branch. In this example the the following two equalities should be satisfied: `(Sing 0 == Sing 0)` and `(o1 == o2)`.

If the application of f_1 is a leaf function application (with respect to the execution call stack) then `(Sing 0 == Sing 0)` should hold. If it is a non leaf application, then `(o1 == o2)` should hold.

```
f₁ :: List → (List, List)
f₁ ls = case ls of
  Sing 0 → (Sing 0, Sing 0)
  Cons x xs →
    let p = f₂ tail in
    let o1 = Cons (v+1) (proj 0 p)in
    let o2 = Cons (v+1) (proj 1 p) in
    (o1, o2)

f₂ :: List → (List, List)
f₂ ls = case ls of
  Sing 0 → (Sing 0, Sing 0)
  Cons v xs →
    let p= f₁ xs in
    let y1 = Cons (v*2) (proj 0 p) in
    let y2 = Cons (v*2) (proj 1 p) in
    (y1, y2)
```

Figure 8. An example for which it is difficult to syntactically eliminate redundant outputs.

Verifying that `(o1 == o2)` is equivalent to verifying that `Cons (v+1) (proj 0 p) == Cons (v+1) (proj 1 p)`, which is true only if `(proj 0 p == proj 1 p)`—in other words, if $f_2^{0=1}$ is satisfied, since p is bound to f_2 function application.

More precisely, for $f_1^{0=1}$ to be satisfied during a non-leaf application at depth l, $f_2^{0=1}$ need to be satisfied for depth $l+1$. In a similar a way $f_2^{0=1}$ is satisfied if $f_1^{0=1}$ is satisfied.

We can use induction to show that $f_1^{0=1}$ is satisfied, under the assumption that the program terminates, as follows:

Base Case: $f_1^{0=1}$ and $f_2^{0=1}$ are satisfied during a leaf function application, since `(Sing 0 = Sing 0)`.

Induction hypothesis: Assume that $f_1^{0=1}$ and $f_2^{0=1}$ holds at depth $> l$.

Induction step: $f_1^{0=1}$ and $f_2^{0=1}$ are satisfied during non leaf application at depth l as a consequence of the induction hypothesis as discussed earlier.

We propose a process through which a compiler can conclude that two outputs of a given functions at two different locations are always the same. The process checks all the conditions that are needed to construct an inductive proof similar to the previous proof.

We will use the example above to illustrate the process, to verify $f_1^{0=1}$. The process tracks two sets of properties: $S1$ for properties that need to verified, and $S2$ for the properties that are already verified. A single property is of the form $f_1^{0=1}$. In our example, at the beginning of the process $S1 = \{ f_1^{0=1} \}$ and $S2 = \{\}$.

The process will keep pulling properties from $S1$ and checks for two things:

	natural termination	linear	treeless	lazy		strict	
				unfused	fused	unfused	fused
append (append ls)	✓	✓	✓	0.47s	**0.42s**	1.51s	**1.28s**
sum (square ls)	✓	✓	✓	0.37s	**0.25s**	0.99s	**0.36s**
shift (sum ls)	✓	✗	✗	**22.9ms**	31.8ms	11.4ms	**7.0ms**
mul2pd ls	✗	✓	✗	2.56s	**2.38s**	0.60s	**0.58s**
mul2pd tree	✗	✓	✗	1.56s	**0.42s**	0.89s	**0.32s**
seteven (sumup tree)	✓	✗	✗	1.11s	**1.07s**	0.95s	**0.52s**
sum (flatten tree)	✓	✓	✓	**0.69s**	0.8s	1.66s	**1.3s**
flip (flip tree)	✓	✓	✓	0.53s	**0.28s**	0.68s	**0.48s**
flipRec (flipRec tree)	✗	✓	✗	3.18s	**2ms**	0.75s	**1ms**
sum (flatten mtrx)	✓	✓	✓	**1.16s**	1.52s	1.37s	**1.35s**

Table 1. Comparison of the runtime of the fused and unfused programs under lazy and strict evaluation. Programs in this table are ported or inspired form previous work.

*Check*1**:** Whether the property is satisfied during a leaf application of the function (leaf with respect to to the call stack).

*Check*2**:** Wether the property is satisfied during a non-leaf application at level l under the assumption that all properties that need to be satisfied at depth $l + 1$ are satisfied.

If the two checks are satisfied, then the set of properties that need to be satisfied at depth $l + 1$ (the assumptions in *check2*) are then added to $S1$, and the condition that was checked will be moved to $S2$. If a condition already exists in $S2$, then it does not need to be added to $S1$ again since it is already verified.

3.3.2 Redundant inputs

The redundant inputs pass targets eliminating inputs of functions when they are not needed. Eliminating such inputs removes the overhead of passing them, especially in recursive functions. It also allows better optimization on the callee and the caller site by possibly eliminating related computations. Furthermore, it can eliminate dependences and allow more tupling. This section will describe several types of redundant inputs that are handled in our transformation.

Shared inputs Function applications that consume the same input at different input positions can be optimized by unifying such arguments into one argument. Although this optimization is performed during tupling, it is performed here again because the output redundancy pass can result in more inputs being shared.

Unconsumed inputs Unconsumed inputs are inputs that are not used in the body of the function that consumes it. Removing such input can eliminate false dependences and allow more tupling.

Non-recursively consumed inputs This pass eliminates inputs that are returned as output without being further consumed in the function. Thus the caller can be rewritten to use them directly.

4 Implementation

We implemented a prototype of the our deforestation algorithm as a program transformation pass in Gibbon [21], a compiler for a small subset of Haskell. Gibbon has a Haskell front-end and can be prompted to output the transformed program into Haskell output. Hence, we used Gibbon to perform Haskell source-to-source transformation. We plan to implement our transformation as a GHC plugin in the future.

5 Evaluation

We evaluated our transformation on a large set of programs showing its ability to fuse them, achieving better performance and lower memory usage.

We divided the benchmarks into two sets: a set of programs inspired by previous related work, and a set of more complicated programs that involve larger traversals. For each experiment, we evaluated the generated Haskell programs in both lazy and strict modes. Strict mode is achieved via the `Strict` pragma in GHC. We also report an experiment that measures the effect of each major pass in the transformation. Finally, we discuss a case in which our transformation was not able to consistently achieve a speedup (Section 5.3).

Experimental setup: We ran our experiments on a Intel Xenon E5-2699 CPU, with 65GB of memory running Ubuntu 18.04. All programs are compiled with GHC 8.8.1 using the `-O3` optimization level, and the runtime numbers are collected by taking the average of 10 program executions. To control termination, all cycles in the transformation are controlled by a maximum depth of 10 in all the reported experiments, unless otherwise noted.

	lazy		strict	
	unfused	**fused**	**unfused**	**fused**
4 render tree passes (1)	1.02s | 767MiB	**0.53s** | 551MiB	1.14s | 461MiB	**0.47s** | 228MiB
4 render tree passes (2)	3.79s |2.52GiB	**2.26s** | 1.85GiB	2.24s |1.53GiB	**1.23s** | 823MiB
5 render tree passes (1)	1.25s | 968MiB	**0.49s** | 590MiB	1.63s | 583MiB	**0.63s** | 291MiB
5 render tree passes (2)	4.97s | 3.16GiB	**2.16s** | 2.01GiB	4.24s |1.924GiB	**1.73s** | 1.04GiB
piecewise function f_1	5.06s | 6.37GiB	**0.88s** | 2GiB	4.99s | 4.71GiB	**1.65s** | 2.75GiB
piecewise function f_2	3.55s | 4.78GiB	**0.76s** | 1.9GiB	3.71s | 3.96GiB	**1.53s** | 2.65GiB
piecewise function f_3	15.0s | 28GiB	**5.56s** | 19GiB	12.1s | 11.28GiB	**6.00s** | 6.59GiB
5 binary tree traversals	4.12s | 2.5GiB	**3.10s**| 1.46GiB	2.08s | 864MiB	**0.77s** | 480MiB

Table 2. Comparison of the runtime and total memory allocated of different fused and unfused programs under lazy and strict evaluation. The two rows for render tree passes run on different inputs. The piecewise functions are defined as follows: $f_1 = x^3 + x^2 + x + 1,\quad f_2 = x^2 + x,\quad f_3 = (f_1)^2 + f_2$

5.1 Surveyed Simple Programs

Each program in this set is a composition of two functions, and is inspired by similar benchmarks from existing literature. These functions have been either shown before or are self-explanatory, and we briefly explain those which are not:

1. `mul2pd` multiplies each element in the input list by 2^i, where i is it's index in the list.
2. `sumup` and `seteven`: These benchmarks operate on a search tree defined as:
   ```
   data STr = Null | Leaf Int | Node Int Bool STr STr
   ```
   `sumup` stores the sum of all sub-trees of a `Node` in it, and `seteven` sets the boolean flag based on whether the sum is even or not.
3. `flipRec` flips each tree at depth `d`, `d` times.

We follow a convention that an argument named `ls` indicates that the input is a list, where `tree` indicates that the input is a tree. The last program, `sum (flatten mtrx)`, operates on a matrix represented as a list of lists.

Table 1 shows the results. For each program, the table contains times that correspond to the fused and the unfused versions in both lazy and strict modes. Three additional properties are shown: natural termination, linearity, and whether the program is in treeless form.

Under strict evaluation, the fusion improves performance for most programs and never introduces any slowdown, with speedups up to more than 5× . Conversely, under lazy evaluation, a runtime regression is caused by fusion for three programs. For some programs, something like fusion happens naturally during lazy evaluation. In such cases, the overhead due to tuples packing and unpacking, as well as the introduced coarser-grained traversals, is not justified.

`flipRec` is an interesting case; fusion does not terminate naturally on the program, however when truncated at depth 10 it eliminates all the additional traversal up to that level, and for a tree of depth 13 that is eliminating almost all of the work, achieving more than 100× speedup.

Overall, for programs in table 1, fusion achieves geomean speedups of 2.4× in lazy evaluation and 2.6× in strict evaluation.

5.2 Larger Programs

In this section we consider another set of programs that are larger, and closer to real-world programs one might encounter in the wild.

Render Tree: Render trees are used in render engines to represent the visual components of a document being rendered. A render tree is consumed by different functions to compute the visual attributes of elements of the document. We implemented a render tree for a document that consists of pages composed of nested horizontal and vertical containers with leaf elements (TextBox, Image, etc.). We implement five traversals that traverse the tree to compute height, width, positions and font style of the visual elements of the document. Each traversal consists of a set of mutually recursive functions. In total, the program consists of more than 40 functions with more than 400 lines of code. Table 2 shows four entries for the render tree, fusing 4 passes and fusing 5 passes with two different inputs. Fusion reduces memory usage and achieves speedups up to 3× for all programs under both lazy and strict evaluation. The suffixes (1) and (2) indicate the variant of the dataset used.

Piecewise Functions: Kd-trees can be used to compactly represent piecewise functions over a multi-dimensional domain. The inner nodes of the tree divide the domain of the function into different sub-domains, while leaf nodes store the coefficients of a polynomial that estimates the function within the node's sub-domain. In this program, we implemented a kd-tree for single variable functions, and different traversals to construct and perform computations on these functions such as adding a constant ($f_1 = x^3 + x^2 + x + 1$), multiplying with a variable ($f_2 = x^2 + x$), and adding the result of two functions ($f_3 = (f_1)^2 + f_2$). Table 2 shows the speedups for three different programs that are expressed using different compositions of those functions along with

	Unfused		Fusion		Fusion + Tupling		Fusion + Tupling + Redundancy Elimination	
	lazy	strict	lazy	strict	lazy	strict	lazy	strict
4 render tree passes (1)	3.79s	3.24s	2.23s	3.22s	4.07s	1.68s	**0.76s**	**0.46s**
5 render tree passes (1)	1.25s	1.63s	0.74s	1.15s	0.70s	1.05s	**0.49s**	**0.63s**
shift (sum ls)	**22.9ms**	11.4ms	58.6s	45.6s	31.8ms	**7.0ms**	31.8ms	**7.0ms**
5 binary tree traversals	4.12s	2.08s	**1.87s**	2.46	3.10	**0.77**	3.10	**0.77**
piecewise function f_3	15.02s	12.10s	**6.82s**	**5.64s**	**6.82s**	**5.64s**	**6.82s**	**5.64s**

Table 3. Runtime of the fused programs when the transformation is truncated at the its three main satges.

the corresponding equations. A binary tree of depth 22 is used to represent those functions.

Fusion achieves up to 5× speedups on those programs and significantly reduces the memory usage. The third program has a relatively lower speedup than the first two, and the reason is that the function that adds two piecewise functions consumes two trees, but our fusion performs fusion across one of them only.

Effect of different passes: Table 3 shows the runtime of the fused programs when the transformation is truncated at its major three stages: fusion, tupling, and redundancy analysis. Render tree is the most complicated program, and it utilizes both tupling and redundancy analysis to achieve speedups especially in strict mode. Simpler, non-linear programs need tupling only to eliminate redundancies and achieve speedups. Finally, although the piecewise functions program is large and not trivial, due to its linearity it only requires fusion to achieve its speedup.

5.3 Does it always work?

There is no guarantee that this transformation is always safe from a runtime perspective. Although for strict evaluation the transformation does not reduce the runtime for almost all the benchmarks, we encountered one case where the performance of the fused program varies between 2x speedup and 2x slowdown for different inputs.

We implemented a sequence of 7 functions that optimize and evaluate first-order lambda calculus expression. The program's traversals are complicated from a fusion perspective, and hard to fuse. Specifically because we are dealing with expressions only, not functions, fusion opportunities are less likely to be found at that level.

For this benchmark, a threshold of 10 for the depth of the transformation was too large for the transformation to terminate in a reasonable time. Furthermore, the code size grows very quickly since the number of different compositions of functions and traversed structures can get very large. In the future, we plan to do a more thorough investigation to analyze this benchmark, and determine the causes of slowdowns for some inputs, and whether it's something that can be handled by our transformation.

6 Related Work

In 1977, Burstall and Darlington [3] provide a calculation method to transform recursive equations so as to reach a fused program, however decisions for applying transformations are left to the programmer. More recent work [13] unified several previous fusion approaches under one theoretical and notational framework based on recursive coalgebras.

Domain specific languages [2, 16] and data-parallel libraries [1, 4, 15] typically include fusion rules that merge multiple data-parallel transformations of their data collections. For example, these systems frequently provide map and fold operations over multi-dimensional arrays (dense or sparse). These systems typically manipulate an explicit abstract syntax representation to perform fusion optimizations, and can generally be classified with the combinator-based approaches we discussed in Section 1.

In contrast, libraries that expose *iterator* or *generator* abstractions can often achieve fusion by construction, and avoid the necessity of fusion as a compiler optimization (which may not always succeed).

For example Rust (or C++) iterators[1] provide a stream of elements without necessarily storing them within a data structure; likewise a Rust (rayon) parallel map operation, simply returns a new parallel iterator without creating a new data structure. In functional contexts as well, libraries often provide data abstractions where a client can "pull" data, or where a producer pushes data to a series of downstream consumers (as in "push arrays" [20]. All these techniques amount to fusion-by-construction programming. However, in these approaches the programmer often needs to manually intervene if they *do* want to explicitly store a result in memory and share it between consumers.

Finally Grafter a [17, 18] is a fusion approach that operates on an imperative representation (where deforestation is not relevant because a tree is updated with no new intermediate result allocated). All the traversals in Grafter are assumed to traverse the same tree. While it might be possible to map functions that do not change the structure of the input into such a representation, Grafter allows limited structural mutations.

[1] https://doc.rust-lang.org/book/ch13-02-iterators.html

7 Conclusion

Deforestation is an important optimization in functional programs due to their stateless nature. Practical fusion optimizations that are adopted by compilers utilize combinator-based fusion techniques. While those are easy to implement, they address a narrow set of fusion opportunities, and require programs to be built using specific combinators.

In this work we propose and implement a practical fusion transformation that operates directly on general recursive functions. We utilize fusion, tupling and redundancy analysis to increase the applicability of such transformations and mitigate or eliminate any performance side effects. The proposed transformation shows significant speedup over GHC optimized Haskell code. We hope that this work will inspire and motivate more work to be done on practical, general deforestation techniques.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 265–283.

[2] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2011. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE, 89–100.

[3] Rod M Burstall and John Darlington. 1977. A transformation system for developing recursive programs. *Journal of the ACM (JACM)* 24, 1 (1977), 44–67.

[4] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Programming Language Design and Implementation*. ACM, New York, NY, USA, 363–375. https://doi.org/10.1145/1806596.1806638

[5] W Chin. 1995. Fusion and tupling transformations: Synergies and conflicts. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming, Susono, Japan*. World Scientific Publishing, 106–125.

[6] Weingan Chin, Zhenjiang Hu, and Masato Takeichi. 1999. A Modular Derivation Strategy via Fusion and Tupling. (12 1999).

[7] Wei-Ngan Chin. 1993. Towards an Automated Tupling Strategy. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '93)*. ACM, New York, NY, USA, 119–132. https://doi.org/10.1145/154630.154643

[8] Wei-Ngan Chin. 1993. Towards an Automated Tupling Strategy. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '93)*. ACM, New York, NY, USA, 119–132. https://doi.org/10.1145/154630.154643

[9] Wei-Ngan Chin. 1994. Safe fusion of functional expressions II: Further improvements. *Journal of Functional Programming* 4, 4 (1994), 515–555. https://doi.org/10.1017/S0956796800001179

[10] Duncan Coutts. 2011. Stream Fusion: Practical shortcut fusion for coinductive sequence types. (2011). https://doi.org/uuid:b4971f57-2b94-4fdf-a5c0-98d6935a44da

[11] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/1291151.1291199

[12] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

[13] Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and Practice of Fusion. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages (IFL'10)*. Springer-Verlag, Berlin, Heidelberg, 19–37. http://dl.acm.org/citation.cfm?id=2050135.2050137

[14] Patricia Johann. 2002. A Generalization of Short-Cut Fusion and its Correctness Proof. *Higher-Order and Symbolic Computation* 15, 4 (01 Dec 2002), 273–300. https://doi.org/10.1023/A:1022982420888

[15] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2012. Guiding parallel array fusion with indexed types. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 25–36.

[16] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ICFP: International Conference on Functional Programming*. ACM, 49–60.

[17] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. 2017. TreeFuser: A Framework for Analyzing and Fusing General Recursive Tree Traversals. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 76 (Oct. 2017), 30 pages. https://doi.org/10.1145/3133900

[18] Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, Fine-grained Traversal Fusion for Heterogeneous Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 830–844. https://doi.org/10.1145/3314221.3314626

[19] Josef Svenningsson. 2002. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP*, Vol. 2. 124–132.

[20] Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing push arrays. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 43–52.

[21] Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Peter Müller (Ed.), Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.26

[22] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231 – 248. https://doi.org/10.1016/0304-3975(90)90147-A

A Declarative Gradualizer with Lang-n-Change

Benjamin Mourad
University of Massachusetts Lowell
USA
benjamin_mourad@student.uml.edu

Matteo Cimini
University of Massachusetts Lowell
USA
matteo_cimini@uml.edu

Abstract

Language transformations are algorithms that take in input a language definition and return another language definition. They can be useful to automatically add features such as subtyping and pattern-matching to languages.

lang-n-change is a domain-specific language for expressing such language transformations algorithms. We have previously used lang-n-change to express simple transformations, which begs the question on whether lang-n-change can be applied to more sophisticated aspects of programming languages.

In this paper, we target the automatic transformation of functional languages into their gradual typed version. We formulate a significant part of the Gradualizer in lang-n-change. Our code is succinct, and shows that lang-n-change can, indeed, be applied to more sophisticated aspects.

CCS Concepts: • **Software and its engineering** → **General programming languages**.

ACM Reference Format:
Benjamin Mourad and Matteo Cimini. 2020. A Declarative Gradualizer with Lang-n-Change. In *Proceedings of Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages (IFL 2020).* ACM, New York, NY, USA, 7 pages.

1 Introduction

Programming language features such as subtyping, pattern-matching, type inference, and gradual typing, among several others, are often added to language definitions a posteriori. Some of these features can be thought of as transformations of a base language definition.

Consider the task of adding pattern-matching to a language, a task that language designers frequently undertake. The operational semantics of pattern-matching makes use of auxiliary relations to handle matches at compile-time and run-time. For example, one of these relations is the typing of patterns with a judgment of the form $\Gamma \vdash p : T \Rightarrow \Gamma'$. This relation ensures that the pattern is well-formed, and provides an output type environment Γ' with bindings (variable-type). In a language with lists, we must add the rules below on the right, derived from the typing rules of the language (on the left).

$$\Gamma \vdash \mathsf{nil} : \mathsf{List}\ T \quad \Longrightarrow \quad \Gamma \vdash \mathsf{nil} : \mathsf{List}\ T \Rightarrow \Gamma$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : \mathsf{List}\ T \Rightarrow \Gamma_2}{\Gamma \vdash \mathsf{cons}\ e_1\ e_2 : \mathsf{List}\ T} \quad \Longrightarrow \quad \frac{\Gamma \vdash p_1 : T \Rightarrow \Gamma_1 \qquad \Gamma \vdash p_2 : \mathsf{List}\ T \Rightarrow \Gamma_2 \qquad \Gamma' = \Gamma_1 \cup \Gamma_2}{\Gamma \vdash \mathsf{cons}\ p_1\ p_2 : \mathsf{List}\ T \Rightarrow \Gamma'}$$

This change can be described as an algorithm. Intuitively, such an algorithm must copy typing rules and insert ps in place of es. Furthermore, it must lift recursive calls to the shape of the typing judgement for patterns, which entails that we assign a new variable to accommodate the output of the call. Finally, all outputs of the recursive calls must be collected together to form the output of the overall rule.

To describe this, and others, transformations on languages, or *language transformations*, Mourad and Cimini have developed a domain-specific language called lang-n-change [Mourad and Cimini 2020a,b]. So far, lang-n-change has been applied to adding subtyping, pattern-matching, and to converting from small-step to big-step semantics, for (mostly functional) language definitions. In such a setting, these are rather simple aspects of programming languages, which begs the question:

Can lang-n-change language transformations be applied to more sophisticated aspects of PL?

In this paper, we show evidence that this is indeed the case by providing lang-n-change formulations that automatically add gradual typing to functional languages.

Gradual typing is an approach to integrating static and dynamic typing within the same language [Siek and Taha 2006]. The algorithms that we use to add gradual typing to languages are not novel. Indeed, we strictly follow the algorithms described in the Gradualizer papers [Cimini and Siek 2016, 2017]. This means that the gradualization process works only on functional languages. Ultimately, we could formulate most of the Gradualizer algorithms in roughly 300 lines of lang-n-change code.

The contributions of this paper are

- lang-n-change transformations to add gradual typing to functional languages, which implement the algorithms the Gradualizer papers in lang-n-change. Differently from the Gradualizer papers, lang-n-change transforms languages defined with a textual representation of operational semantics, while the Gradualizer takes in input logic programs.

IFL 2020, September 2020, Kent, UK.
2020.

```
Expression e ::= x | (abs T (x)e) | (app e e)
Type T ::= (arrow T T)
Value v ::= (abs T (x)e)
Context E ::= [] | (app E e) | (app v E)
TypeEnv Gamma ::= MAP(x, T)

(T-VAR)
member((x => T), Gamma)
--------------------------------------
Gamma |- x : T

(T-ABS)
Gamma, x : T1 |- e : T2
--------------------------------------
Gamma |- (abs T1 (x)e) : (arrow T1 T2)

(T-APP)
Gamma |- e1 : (arrow T1 T2),
Gamma |- e2 : T1
--------------------------------------
Gamma |- (app e1 e2) : T2

(R-BETA)
--------------------------------------
(app (abs T1 (x)e) v) --> e[v/x]

# variance arrow -> contra cov
# mode typeOf -> inp inp out | step -> inp out
```

Figure 1. The Simply-Typed λ-Calculus in LANG-N-CHANGE

- Our formulations show that the gradualization algorithms can be written succinctly in LANG-N-CHANGE, and that LANG-N-CHANGE language transformations can indeed be applied to more sophisticated aspects.

Section 2 describes our LANG-N-CHANGE code for generating the static semantics of gradually typed languages. Section 3 describes that for generating the dynamic semantics of gradually typed languages. Section 4 provides some discussion and concludes the paper.

The LANG-N-CHANGE tool is open source. Its repo contains language transformations algorithms, language definitions, and transformed languages, and can be found at [Mourad and Cimini 2019].

2 Static Semantics of Gradual Typing

LANG-N-CHANGE starts with a language in input. Fig. 1 shows the simply-typed lambda-calculus in LANG-N-CHANGE. The syntax for defining languages is essentially a textual representation of operational semantics. Then, LANG-N-CHANGE expresses a language transformation with a domain-specific language. We explain the operations of LANG-N-CHANGE as we encounter them in the remainder of the paper in the algorithms for adding gradual typing. These algorithms always start from the language definition in input, such as Fig 1, apply an instruction, and pass the modified language to the next instruction.

2.1 Adding the Dynamic Type

The gradually typed language augments the base language with a special type dyn that represents the dynamic type. The LANG-N-CHANGE code to do so is

```
1  Type T ::=  ... | (dyn [])
```

The notation ...| inside a grammar means that LANG-N-CHANGE takes the grammar Type of the language in input and add a new grammar item (dyn []) to it.

2.2 Split Type Equality

Next, we need to make the type variables that are used as output distinct. Since this operation is quite common, LANG-N-CHANGE provides a specific operation for doing this.

```
1  Rule(keep)[|-]:
2   uniquify(Premise[*]: self, mode, out) =>
        (mymap, newprems):
3    newprems
4    --------------
5    conclusion
```

Line 1 is a selector. It selects all the rules of the current language whose conclusion makes use of the relation $\vdash$; that is, it selects all the typing rules. The body of the selector is in lines 2-5, and is the body applied to the selected rule. For each of them, the body returns a new rule. uniquefy takes in in put three arguments. The first is a set of premises. In this case, we pass all the premises of the selected rule. We do so by using a selector, as well, with Premise[$*$] : self. Inside [..] is a pattern of the premises that we select, but in this case the pattern [*] selects all of them. uniquefy also takes in input a mode map that tells which arguments of relations are input and which are output, and takes the string "out" that instructs uniquefy to act only on the variables that are in output position according to mode.

uniquefy also returns a map that associates a variable just replaced with the variables that have been used to replace it. For the typing rule for application, it returns mymap $= \{T_1 \mapsto [T_{11}, T_{12}]\}$. The resulting language also needs the attribute keep, which tells LANG-N-CHANGE to simply keep the rules that do not match with $\vdash$.

2.3 Generating Consistency or Join

For the types that we split into unique variable names, we add the consistency ($\sim$) relation between them. The following code modifies the typing rule to reflect this.

```
1      Premise[*]: self,
2      concat(mymap[T]:
3        fold(~, mymap.[T])
4      )
5      -----------------------------------
6      conclusion
```

Line 1 uses a selector to preserve the current premises of the rule. Line 3 iterates over the keys of mymap, which are the types that were split into unique variable names. Line 4 uses the built-in fold operation to generate the new premises. It takes in input a predicate name for a binary relation and a list of terms (in this case, $\sim$ and `mymap.[T]`, respectively) and interleaves the terms from left to right in pairs, generating new premises for the given predicate name. For example, if `mymap.[T]` $= [T_1, T_2, T_3, T_4]$, then `fold(`$\sim$`, mymap.[T])` $= [T_1 \sim T_2, T_2 \sim T_3, T_3 \sim T_4]$.

As an example we show the above transformation on the typing rule for application:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \to T_2 \\ \Gamma \vdash e_2 : T_1\end{array}}{\Gamma \vdash e_1\ e_2 : T_2} \quad \Longrightarrow \quad \frac{\begin{array}{c}\Gamma \vdash e_1 : T_{11} \to T_2 \\ \Gamma \vdash e_2 : T_{12} \\ T_{11} \sim T_{12}\end{array}}{\Gamma \vdash e_1\ e_2 : T_2}$$

For application, the type T_{11} is contravariant, which allows for a consistency relation to be present. In the absence of such types, the variables in question are peers. Therefore, a join ($\sqcup$) between these types is required.

```
1     let contraT = concat(
2        Premise[Gamma |- e : (c TTs)]:
3          let vmap = makeMap(TTs, variance.[c]) in
4          vmap[T]: if vmap.[T] = contra then T
                  else nothing
5     ) in
6     Premise[*]: self,
7     mymap[T]:
8       if not(overlap(contraT, mymap.[T]))
9       then (join (T @ mymap.[T]))
10      else nothing
11    ---------------------------------------------
12    conclusion
```

Lines 1-5 make use of a let-binding to the variable `contraT`, which contains a list of type variables identified to be in contravariant positions in the typing premises. Lines 2-4 iterate over these premises whose output type matches with a constructor (`c` $T_1 \ldots T_n$). The constructor name `c` is then used in a lookup in `variance`, which is a mapping from constructor names to the variance of each position for its list of arguments. For example, in a language with the function ($\to$) type, `variance` $= \{$`arrow` $\mapsto$ `[contra, cov]`, $\ldots\}$, where the first argument is contravariant (`contra`) and the second argument is covariant (`cov`). Line 3 creates a mapping from each argument to its associated variance and binds it to `vmap`. Line 4 then iterates over each key in this map using a selector and filters out the types which map to `contra`. The expression `if vmap.[T] = contra then T else nothing` returns `nothing` if the type is not contravariant, which is equivalent to discarding the result from the list.

Lines 6-12 make use of `contraT` in deciding whether to compute the join and add it to the premises of the rule. Line 6 preserves the current premises, as done before. Line 7 iterates over the type variables which were split, since these are the ones relevant to computing the join. Line 8 uses the built-in operator `overlap`, which checks for overlapping terms between the lists `contraT` and `mymap.[T]`. If there are overlapping terms, then one or more of the types in `mymap.[T]` are contravariant, so we skip computing the join. Otherwise, line 9 adds the premise to compute the join of all the types in `mymap.[T]` and place the output in `T`.

In the case that there remain consistency relations which are subsumed by join relations, the following code will clean up the premises of the rule accordingly:

```
1     let consistencyPrems = Premise[T1 ~ T2]:
          self in
2     listDifference(Premise[*]: self,
          consistencyPrems),
3     consistencyPrems[T1 ~ T2]:
4       if isEmpty(Premise[(pred ts)]:
5         if pred = join then
6           if overlap(T1, ts) and overlap(T2, ts)
                then self else nothing
7         else nothing
8       ) then self else nothing
9     -----------------------------------------
10    conclusion
```

As an example, we show the transformation on the rule for the `if` operator:

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{Bool} \\ \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T\end{array}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T} \quad \Longrightarrow \quad \frac{\begin{array}{c}\Gamma \vdash e : \mathsf{Bool} \\ \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \\ T_1 \sim T_2 \quad T = T_1 \sqcup T_2\end{array}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T}$$

$$\Longrightarrow \quad \frac{\begin{array}{c}\Gamma \vdash e : \mathsf{Bool} \\ \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2 \\ T = T_1 \sqcup T_2\end{array}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T}$$

2.4 Compute Final Type and Fix the Conclusion

Previously we have made some output variables distinct. These have been only those in the premises of the rule. It may happen, then, that the conclusion of the rule still refers to the old name for the variable that have been split. This variable now is basically not bound to any output, and we therefore need to fix this. However, there is a question on what variable we should give to it because now there are several distinct names for the same variable. As explained in the first Gradualizer paper [Cimini and Siek 2016], in the case of the if-then-else, the conclusion takes the join type. If a contravariant variable was around instead, the original name of the variable should take that of the contravariant one. For each variable, then, we compute a *final type* [Cimini and Siek 2016], that is, the variable it should be replaced to when it occurs in the conclusion. Below is the code for computing the final type and for fixing the conclusion of the rule.

```
1     let finalType =
2       concat(mymap[Tk]:
3         concat([conclusion][Gamma |- e : Te]:
4           concat(varsOf(e)[Tf]:
5             if Tf in mymap.[Tk] then
6               makeMap(Tk, Tf)
7             else
8               let ov = getOverlap(mymap.[Tk],
                     contraT) in
9              if isEmpty(ov) then makeMap(Tk,
                   Tk) else makeMap(Tk, ov))))
10      in
11      substitute(self, finalType)
```

At line 3, the notation [conclusion][Gamma ⊢ e : Te] is a simple trick that makes use of the selector to pattern-match the conclusion with the form Gamma ⊢ e : Te. With that, we can extract the output type in the conclusion. The code above checks whether the are contravariant or not, The code builds a map between each original variable, which are in the keys of mymap, and the one variable it should be replaced to.

2.5 Pattern-matching

A subtlety in gradual typing occurs when the typing rule exactly expects to find a type constructor. For example in the typing rule for application we have that e_1 is expected to be typed at the function type. In gradual typing, instead, we need to accommodate the fact that that expression can simply be dynamically typed and we will check at run-time on whether it is of function type. To do so, we have to prevent the typing rule to exactly match the output of that type checker call with a function type. We instead modify that premise with two premises $\Gamma \vdash e_1 : T'$ and T' gradualMatch $T_1 \longrightarrow T_2$ where T' is a fresh new variable, and gradualMatch is specifically devoted to 1) match function types, if T happens to be an actual function type, and 2) also match the dynamic type dyn if instead it is dynamically typed. In the latter case, the typing rule consider T_1 and T_2 as dynamically typed too. The following LANG-N-CHANGE code performs this transformation:

```
1  concat(
2    Premise(keep)[Gamma |- e : (c Ts)]:
3      let V = newvar(V) in
4      [Gamma |- e : V, (gradualMatch [V, (c Ts)])]
5  )
6  -------------------------------------
7  conclusion
```

At line 2, we scan the typing rules of the language. However, we select only those whose output matches (c $T_1 \ldots T_n$), where c is a top level constructor. In that case, line 4 generates the two premises that we have discussed above.

2.6 Generate the Consistency Relation

The code that we have seen in the previous section generates the new typing rules for the gradually typed language. However, these rules make use of auxiliary relations that were not part of the base language that we started with. In particular, these rules make use of the consistency relation, gradualMatch, and the join. We show the code to generate consistency relation in this section. In the next section we show the code to generate gradualMatch. As computing the join follows similar lines we omit the code for generating its definition but it can be found in the repo of LANG-N-CHANGE [Mourad and Cimini 2020a].

We start with the consistency relation. The code is below.

```
1     T ~ (dyn []);
2
3     (dyn []) ~ T;
4
5     Type[(c Ts)]:
6       Ts[T]: let TT = unbind(T) in TT ~ TT'
7       ---------------------------------
8       (c Ts) ~ (c Ts')
```

Lines 1 and 3 mean that we simply add those two rules to the language. These two rules say that dyn is related to everything. Next, at lines 4-8, we generate the definition for type constructors. We scan every type in the grammar of types of the language. For each of these we generate one rule. The conclusion of this rule relates the selected type with other types with the same top level type constructor. Then, the premises are such to relate arguments pairwise. The relation is a congruence then, and is not driven by the variance of arguments. Of course, more sophisticated languages have a rather different treatment of the consistency relation (such as [Ahmed et al. 2011; Igarashi et al. 2017; Xie et al. 2019] among others) and those are out of the scope of this gradualization process.

2.7 Generate the Gradual Matching Relation

In this section we show the code to generate gradualMatch

```
1  Type[(c Ts)]:
2    if not(c = dyn)
3    then (gradualMatch [(c Ts), (c Ts)])
4    else nothing
5  ;
6  Type[(c Ts)]:
7    if not(c = dyn) then
8       let newTs = Ts[T]:
9        if isBinding(T) then
10         let X = boundOf(T) in
11         (X)(dyn [])
12       else (dyn [])
13      in (gradualMatch [(dyn []), (c newTs)])
14   else nothing
```

Lines 1-4 generate the rules to match a type constructor with itself. As we have seen in the case of function types, gradualMatch must be prepared to match a function type indeed. We scan every type. Notice however, that we are in a language in which the grammar of types has been augmented with dyn, we therefore skip dyn because the match operates for those type constructor that the typing rules of the original language where trying to match. For each of these types, then, we simply relate them with themselves.

Lines 6-14 instead, relate dyn with the types of the original language. In this case, we relate it to each type in which the top level constructor is applied to dyn for every argument.

2.8 Missing: Arbitrarily nested pattern-matching

The LANG-N-CHANGE code for generating the typing rules with `gradualMatch` of Section 2.5 (not the previous section) is less powerful than that of the Gradualizer paper [Cimini and Siek 2016]. Indeed, it works only for typing rules that solely match the top level constructor of a type applied to all variables, as in $\Gamma \vdash e_1 : T_1 \rightarrow T_2$. If the language was a little more complicated, for example if e_1 was a pair of functions with premise $\Gamma \vdash e_1 : (T_1 \rightarrow T_2) \times (T_3 \rightarrow T_4)$, the Gradualizer paper generates 4 premises: $\Gamma \vdash e_1 : X$, $\texttt{gradualMatch}\ X\ (X' \times X'')$, $\texttt{gradualMatch}\ X\ (T_1 \rightarrow T_2)$, and $\texttt{gradualMatch}\ X\ (T_3 \rightarrow T_4)$. That is, the matches are recursively expanded when nested matching are encountered. Unfortunately, LANG-N-CHANGE does not currently have recursion mechanisms and we can only generate `gradualMatch` for the top level type constructor. Extending LANG-N-CHANGE with recursion and capturing arbitrarily nested pattern matching is part of our future work.

3 Dynamic Semantics of Gradual Typing

In this section we describe our LANG-N-CHANGE transformation to automatically generate the dynamic semantics of a gradually typed language. In the standard approach to the dynamic semantics for gradual typing, programs are executed in a version of the language with a cast operator, known as the *Cast Calculus*. Here, casts have the form $(\texttt{cast}\ e\ T_1\ T_2)$, which means that the expression e is of type T_1 and is cast to the type T_2.

The dynamic semantics of the language with casts must be prepared to detect whether a cast fails or succeeds. For example, if an integer 4 is passed to a function that is dynamically typed and is then used in an operation which expects a boolean, we end up performing

$$(\texttt{cast}\ (\texttt{cast}\ 4\ \texttt{Int}\ \texttt{dyn})\ \texttt{dyn}\ \texttt{Bool})$$

which fails at run-time. We call the reduction rules that handle these cast scenarios *cast reduction rules*. However, extra difficulty arises for inductive types. For example, it is not clear how to perform a cast on a function, as in

$$(\texttt{cast}\ \lambda x.e\ (\texttt{Int} \rightarrow \texttt{dyn})\ (\texttt{Int} \rightarrow \texttt{Int}))$$

. How can we know that a dynamically typed function actually returns an integer at run-time? To solve this problem, we perform the cast only when the function is applied [Findler and Felleisen 2002]. Therefore the language is augmented with specific reduction rules. For functions, we have

$$(\texttt{cast}\ v_1\ (T'_1 \rightarrow T'_2)\ (T_1 \rightarrow T_2))\ \ v_2$$
$$\longrightarrow$$
$$(\texttt{cast}\ (v_1\ (\texttt{cast}\ v_2\ T_1\ T'_1))\ T'_2\ T_2)$$

Notice that casts are decomposed and distributed to the sibling arguments (here v_2 only) and also wrap the whole expression. The argument is cast before being passed, and the result of the function is also cast. Below, we call these types of reduction rules *operator-specific cast rules*.

The literature provides an algorithm to automatically generate the dynamic semantics of gradually typed languages [Cimini and Siek 2017]. The algorithm below implements most of the algorithm in LANG-N-CHANGE (we did not model blame tracking).

```
1   Type T ::= ... | (dyn []);
2   Expression e ::= ... | (cast e T T)
3     | (castError [])
4   ;
5   Error er ::= ... | (castError []);
6   Context E ::= ... | (cast E T T);
7   GroundType G ::=
8     Type[(c Ts)]: in (c Ts[*]: (dyn []))
9   ;
10  Value v ::= ... | (cast v G dyn) |
11    Type[(c Ts)]:
12      if not(isEmpty(Ts))
13      then (cast v (c Ts) (c Ts))
14      else nothing
15  ;
16
17  Gamma |- (castError []) : T;
18
19  Gamma |- e : T1, T1 ~ T2
20  --------------------------------
21  Gamma |- (cast e T1 T2) : T2
22  ;
23
24  (cast (cast v G (dyn [])) (dyn []) G) --> v;
25
26  G1 =/= G2
27  ---------------------------------------------
28  (cast (cast v G1 (dyn [])) (dyn []) G2) -->
        (castError [])
29  ;
30
31   ... the other cast reduction rules ...
32
33  Rule[Gamma |- (op es) : T]:
34   if isKindOp(op, Value) then nothing else
35    let castT = head(
36      Premise[G |- e : (c Ts)]: (c Ts)
37    ) in
38    let castMap = concat(
39      tail(premises)[Gamma |- e : Te]:
40        makeMap(e, Te)
41    ) in
42    let siblings =
43      tail(es)[e]:
44        (cast e castMap.[e]
             castMap.[e]'|(vars(castT)))
45    in
46    (op (cast v castT' castT) tail(es))
47                   -->
48    (cast (op v siblings) (T'|(vars(castT))) T)
```

Lines 1-6 augment the language with the dynamic type, cast operator and cast error. Lines 7-9 generate the grammar for the so-called *ground types*. In gradual typing, a cast from $\texttt{Int} \rightarrow \texttt{dyn}$ to $\texttt{dyn} \rightarrow \texttt{Int}$ is divided into two casts: one from

`Int` ⟶ `dyn` to `dyn` ⟶ `dyn` and another from `dyn` ⟶ `dyn` to `dyn` ⟶ `Int`. The type highlighted is a ground type, which can be checked in a dynamically typed fashion. Ground types comprise the basic types and the inductive types when applied to dynamic types only. We select all types and we replace the arguments in `Ts` with `dyn`, which has the effect of creating a list with as many `dyn` as arguments. Lines 11-15 generate the values. As we mentioned above a function cast from `Int` ⟶ `dyn` to `Int` ⟶ `Int` is unresolved until applied. Thus, this and similar casts are values in gradual typing. We augment the values of the language with cast values from inductive types to (the same) inductive types. We select every type and, if the arguments are not empty (as with inductive types), then we generate the cast value.

Lines 17-22 add the typing rules for the cast operator and cast error. Notice that the typing rule for the cast operator relies on the relation ∼. Lines 24-31 add the cast handling rules. These rules are standard from literature and we omit them. We only show the reduction rules for failing and succeeding casts.

Lines 33-48 are responsible for creating the operator-specific cast rules. Let us consider the case of function casts. We create a reduction rule for the application so that it handles casts on functions. This is because now function casts are values (see lines 10-15). The application of a function tries to remove the cast and expose the function underneath because this is the only value we can use. We therefore strive to go back to a place where we can use the β-reduction. However, once we remove the cast, the type of the function is different. This causes a mismatch with the types of the sibling arguments (the argument of the function, in this case) and with the type of the whole expression. Therefore we insert casts around the siblings and around the whole expression, and get back to matching the types. The mismatch happens when the types that the function uses are also used by the sibling arguments and are used for typing the whole expression (removing the cast exposes different types and creates the mismatch with all in the surrounding context that used those types). As shown in [Cimini and Siek 2017], this scenario is not particular to functions but generalizes to most common types. Line 33 selects all typing rules, while line 44 filters out rules that type values. For simplicity, we assume that the principal argument of an elimination form (the value being subject of the operation) is the first argument. We also assume that the first premise of the typing rule is the typing premise of this first argument. Then, lines 35-37 retrieve the type of the first argument from the first premise of the rule. Lines 38-41 create a map from the sibling arguments to their types. Lines 42-45 create the casts around the siblings. Casts are from their types to the ticked versions of their types. Also, the tick operation is restricted only to those types that appear in the type of the first/principal argument (`vars(castT)`). Lines 46-48 create the reduction rule (there is no horizontal line because we have no premises). In the source of the step we place a cast value in the first/principal argument position. The target of the step removes the cast at that position and leaves the value v. It also replaces the siblings with *siblings* and wraps the whole expression in a cast. The latter cast gets the type back to T from the type T in which the types of the first/principal argument (`castT`) are ticked.

4 Discussion and Conclusion

In this paper, we have used LANG-N-CHANGE to formulate a significant part of the Gradualizer by Cimini et al [Cimini and Siek 2016, 2017]. We believe that our formulations are rather declarative, and map well with the algorithms of the original papers. Furthermore, this paper could be a more accessible resource than the Gradualizer papers for newbies because 1) we work on a textual representation of pen&paper operational semantics. In contrast the Gradualizer takes in input and manipulates λ-prolog logic programs. 2) Also, our declarative LANG-N-CHANGE transformations may flesh out the intention of the Gradualizer paper in a clear way.

The Gradualizer implementation and our LANG-N-CHANGE formulation cannot be compared directly yet because they work on different representations, and because the Gradualizer also captures other features: blame tracking [Wadler and Findler 2009] and arbitrarily nested pattern-matching (discussed at the end of Section 2).

Nonetheless, for the parts that we cover with LANG-N-CHANGE we can provide a very succinct code in roughly 300 lines. We believe that this paper provides some evidence that language transformations can indeed be applied to sophisticated aspects of programming languages such as gradual typing.

In the future, we would like to cover the following features:

- Blame tracking,
- Arbitrarily nested pattern-matching. To add this feature, we will extend LANG-N-CHANGE with recursion.

References

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 201–214. https://doi.org/10.1145/1926385.1926409

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: a methodology and algorithm for generating gradual type systems. In *Symposium on Principles of Programming Languages (POPL)*.

Matteo Cimini and Jeremy G. Siek. 2017. Automatically Generating the Dynamic Semantics of Gradually Typed Languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 789–803.

Robert Bruce Findler and Matthias Felleisen. 2002. *Contracts for Higher-Order Functions*. Technical Report NU-CCS-02-05. Northeastern University.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017. On polymorphic gradual typing. *Proc. ACM Program. Lang.* 1, ICFP (2017), 40:1–40:29. https://doi.org/10.1145/3110284

Benjamin Mourad and Matteo Cimini. 2019. Lang-n-Change. Webpage of the tool. `http://cimini.info/LNC/index.html`.

Benjamin Mourad and Matteo Cimini. 2020a. A Calculus for Language Transformations. In *SOFSEM 2020: Theory and Practice of Computer Science - 46th International Conference on Current Trends in Theory and Practice of Informatics, SOFSEM 2020, Limassol, Cyprus, January 20-24, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12011)*, Alexander Chatzigeorgiou, Riccardo Dondi, Herodotos Herodotou, Christos A. Kapoutsis, Yannis Manolopoulos, George A. Papadopoulos, and Florian Sikora (Eds.). Springer, 547–555. https://doi.org/10.1007/978-3-030-38919-2_44

Benjamin Mourad and Matteo Cimini. 2020b. Lang-n-Change – A Tool for Transforming Languages (System Description). In *Proceedings of the 15th International Symposium on Functional and Logic Programming (FLOPS 2020)*. To appear.

Jeremy G. Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*. 81–92.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can't be blamed. In *European Symposium on Programming (ESOP)*. 1–16.

Ningning Xie, Xuan Bi, Bruno C. D. S. Oliveira, and Tom Schrijvers. 2019. Consistent Subtyping for All. *ACM Trans. Program. Lang. Syst.* 42, 1, Article 2 (Nov. 2019), 79 pages. https://doi.org/10.1145/3310339

Type- and Control-Flow Directed Defunctionalization

Maheen Riaz Contractor
Rochester Institute of Technology
Rochester, NY, United States of America
mc1927@rit.edu

Matthew Fluet
Rochester Institute of Technology
Rochester, NY, United States of America
mtf@cs.rit.edu

ABSTRACT

Defunctionalization is a program transformation that removes all first-class functions from a source program, leaving behind an equivalent target program that contains only first-order functions. As originally described by Reynolds, the defunctionalization transforms an untyped higher-order source language into an untyped first-order target language with a single, global dispatch function. In addition to being limited to untyped languages, another drawback of this approach is that obscures control flow, making it appear as though the code associated with every source function could be invoked at every call site of the target program. Subsequent work has extended defunctionalization to both simply-typed and polymorphically-typed languages, but the latter remains limited to a single, global dispatch function. Other work has extended defunctionalization of a simply-typed language to be guided by a control-flow analysis of the source program, where the types of the target program exactly capture the results of the flow analysis and makes it apparent which (limited) set of functions can be invoked at each call site. Our work draws inspiration from these previous approaches and proposes a novel flow-directed defunctionalization for a polymorphically-typed language. Guided by a *type- and control-flow analysis*, which exploits well-typedness of the source program to filter flows that are incompatible with static types, the transformation must construct evidence that filtered flows are impossible in order to ensure the well-typedness of the target program.

KEYWORDS

defunctionalization, control-flow analysis, type-flow analysis

1 INTRODUCTION

Defunctionalization is a program transformation that removes all first-class functions from a source program, leaving behind an equivalent target program that contains only first-order functions. In order to do so, each first-class-function value in the source program is represented by a first-order *closure* value, comprised of a distinct tag and a record of values; the tag is uniquely associated with a source λ-abstraction and the record of values corresponds to the free variables of the source λ-abstraction. Each application expression in the source program is transformed into an expression that performs a case analysis on the tag component of a closure (obtained as the result of evaluating the transformed function subexpression of the application) and dispatches to the transformed body of the corresponding source λ-abstraction passing the record of values component of the closure and the actual argument (obtained as the result of evaluating the transformed argument subexpression of the application). First discovered by Reynolds [12], a variety of techniques for [2, 3, 6, 7, 10, 11, 17, 18] and applications of defunctionalization have been proposed.

Consider the following source program, which we will use to illustrate the original defunctionalization transformation and our novel type- and control-flow directed defunctionalization:

```
let id = λx. x in
let app = λf. λz. let g = id f in g z in
let add = λa1.λa2. a1 + a2 in
let mul = λb1.λb2. b1 ∗ b2 in
let minc = λc1. λc2. if c1 then c2 + 1 else c2 in
let res1 = id add in
let res2 = id mul in
let res3 = app minc Tru in
...
```

Essence of Reynolds Defunctionalization. Defunctionalizing the source program yields the following target program which is comprised of (mutually recursive) algebraic data type definitions, (mutually recursive) first-order function definitions, and a "main" expression:

```
data Cls = {Id(), App(_), App′(_, _), Add(), Add′(_),
            Mul(), Mul′(_), MInc(), MInc′(_)} ;
fun idC(x) = x
fun appC(id, f) = App′(id, f)
fun app′C(id, f, z) = let g = apply(id, f) in apply(g, z)
fun addC(a1) = Add′(a1)
fun add′C(a1, a2) = a1 + a2
fun mulC(b1) = Mul′(b1)
fun mul′C(b1, b2) = b1 ∗ b2
fun mincC(c1) = MInc′(c1)
fun minc′C(c1, c2) = if c1 then c2 + 1 else c2
fun apply(fn, arg) = case fn of Id() ⇒ idC(arg)
                                App(id) ⇒ appC(id, arg)
                                App′(id, f) ⇒ app′C(id, f, arg)
                                Add() ⇒ addC(arg)
                                Add′(a1) ⇒ add′C(a1, arg)
                                Mul() ⇒ mulC(arg)
                                Mul′(b1) ⇒ mul′C(b1, arg)
                                MInc() ⇒ mincC(arg)
                                MInc′(c1) ⇒ minc′C(c1, arg) ;
let id = Id() in
let app = App(id) in
let add = Add() in
let mul = Mul() in
let minc = MInc() in
let res1 = apply(id, add) in
let res2 = apply(id, mul) in
let res3 = apply(apply(app, minc), Tru) in
...
```

IFL'20, September 2020, Virtual
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For each λ-abstractions in the source program, we introduce a variant of the Cls algebraic data type where the arity of the variant corresponds to the number of free variables of the λ-abstraction (e.g, Add′(_) for $\lambda a2.\ a1 + a2$). Also, for each λ-abstraction, we introduce a first-order "code" function that accepts the values of the free variables and the argument and executes the (transformed) body of the λ-abstraction (e.g, *add′C*). A distinguished *apply* function accepts a value of the Cls algebraic data type and an argument, examines the closure to determine the variant and extract the values of the free variables, and dispatches to the appropriate code function, passing the values of the free variables and the argument. Each λ-abstraction in the source program is transformed into a construction of the appropriate variant and each application expression in the source program is transformed into an application of the *apply* function.

Reynolds' original defunctionalization is similar to the above, except that we need to inline each of the first-order "code" functions into the *apply* function. Alternatively, we could inline the *apply* function at each call site.

There are two significant limitations with this original defunctionalization. First, the transformation is defined to target an untyped first-order language, which limits the amount of static checking that can be performed on the target program. Second, the transformation obscures the control flow by suggesting that any code function might be invoked from any call site in the target program. Moreover, these limitations are not entirely independent; indeed, due to the obscured control flow, in the target program it may appear that function can be called with an argument of an inappropriate type. To address the first limitation, defunctionalization has been extended to operate on simply-typed [2, 17, 18] and polymorphically-typed [10, 11] languages. To address the second limitation, defunctionalization of a simply-typed language has been extended to be guided by flow analyses [3], which more precisely captures the set of functions that may be invoked at a particular call site. But, no work has simultaneously addressed both limitations for polymorphically-typed languages.

Essence of Type- and Control-Flow Directed Defunctionalization. In this paper, we combine the benefits of flow-directed defunctionalization [3] and polymorphic typed defunctionalization [10, 11]. That is, we use a flow analysis to guide the defunctionalization of a polymorphic higher-order source program into a polymorphic first-order target program, where the results of the flow analysis are precisely reflected in (and verified by) the types of the target program. Consequently, each call site in the target program dispatches only among the functions that the flow analysis asserts may be invoked at the corresponding call site in the source program.

To guide our defunctionalization transformation, we use *type- and control-flow analysis* (TCFA), a flow analysis for System F (with recursion) that we developed in previous work [1, 5] as an extension of 0CFA [9, 16], the classic monovariant control-flow analysis that was formulated for the untyped lambda calculus. TCFA yields both *control-flow* information via a global context-insensitive environment that maps expression variables to sets of (abstract) values (e.g., λ- and Λ-expressions) that may be bound to the expression variable during evaluation and *type-flow* information via a global context-insensitive environment that maps type variables to sets of type expressions that may instantiate the type variable during evaluation. In addition, TCFA exploits well-typedness of the program to improve the precision of the analysis by allowing two flows to influence each other: control-flow information determines which Λ-expressions may be applied at a type-application expression (thereby determining which type expressions flow to which type variables) and type-flow information filters the (abstract) values that may flow to expression variables (by rejecting abstract values with static types that are incompatible according to the type-flow information with the static type of the receiving expression variable).

Consider the following polymorphically-typed version of our example source program:

```
let id = Λα. λx:α. x in
let app = Λβ. Λδ. λf:β→δ. λz:β.
            let g = id @(β→δ) f in g z in
let add = λa1:Int.λa2:Int. a1 + a2 in
let mul = λb1:Int.λb2:Int. b1 ∗ b2 in
let minc = λc1:Bool. λc2:Int. if c1 then c2 + 1 else c2 in
let res1 = id @(Int→Int→Int) add in
let res2 = id @(Int→Int→Int) mul in
let res3 = app @(Bool) @(Int→Int) minc Tru in
...
```

and the (partial) result of TCFA, given by an environment $\hat{\rho}$:

$$
\begin{array}{rlrl}
\hat{\rho}(\alpha) = & \{\texttt{Int}\to\texttt{Int}\to\texttt{Int}, \beta\to\delta\} & \hat{\rho}(id) = & \{\Lambda\alpha\} \\
\hat{\rho}(\beta) = & \{\texttt{Bool}\} & \hat{\rho}(x) = & \{\lambda a1,\ \lambda b1,\ \lambda c1\} \\
\hat{\rho}(\delta) = & \{\texttt{Int}\to\texttt{Int}\} & \hat{\rho}(f) = & \{\lambda c1\} \\
 & & \hat{\rho}(z) = & \{\hat{\$}\texttt{Bool}\} \\
 & & \hat{\rho}(g) = & \{\lambda c1\} \\
 & & \hat{\rho}(res1) = & \{\lambda a1, \lambda b1\} \\
 & & \hat{\rho}(res2) = & \{\lambda a1, \lambda b1\} \\
 & & \hat{\rho}(res3) = & \{\lambda c2\}
\end{array}
$$

(where $\hat{\$}\texttt{Bool}$ is the abstract value for the Bool base type).

As a monovariant analysis, TCFA conflates all functions that flow through the *id* function and (correctly) maps x to $\{\lambda a1, \lambda b1, \lambda c1\}$. Naïvely, it might appear that the flow analysis should map each variable bound to a call of *id* (*res1*, *res2*, and *g*) to this set. However, type soundness ensures that *res1* and *res2* may only be bound to values of type Int→Int→Int and therefore cannot be bound to $\lambda c1$. More subtly, g cannot be bound to $\lambda a1$ or $\lambda b1$, due to the static type of g ($\beta\to\delta$) and the type-flow information about the types at which β and δ may be instantiated. Note that failing to exploit the type-flow information when computing the control-flow information for g (i.e., by mapping g to $\{\lambda a1, \lambda b1, \lambda c1\}$) would result in *res3* being mapped to $\{\lambda a2, \lambda b2, \lambda c2\}$; furthermore, note that this mapping for *res3* could not be improved by post-processing, because both of $\lambda a2$ and $\lambda b2$ have static types that are compatible with that of *res3*.

The binding of g to the application $id\ @(\beta\to\delta)\ f$ highlights the key challenges to be addressed by our defunctionalization transformation. As noted above, three distinct functions flow through the *id* function; hence, the types of both the argument x and the result of the first-order function representing λx in the defunctionalized program will be a data type Val_3 with three constructors corresponding to $\{\lambda a1, \lambda b1, \lambda c1\}$. Meanwhile, the types of both the argument f and the local variable g of the first-order function representing λf in the defunctionalized program will be a data type

Val_4 with one constructor corresponding to $\{\lambda c1\}$. In order to pass the value of the actual argument f for the formal argument x, we will need to coerce from Val_4 to Val_3. In this case, it is a simple "up-cast", because the (single) constructor in Val_4 has a corresponding constructor in Val_3. However, in order to bind the result of the call to g, we will need to coerce from Val_3 to Val_4. In this case, it is a "down-cast", because only one of the constructors in Val_3 has a corresponding constructor in Val_4. For the other two constructors, the defunctionalization transformation must provide evidence that these matches are impossible.

The essence of our solution is to use Generalized Algebraic Data Types (GADTs) (first appearing in the literature under the names *guarded recursive data types* [19], *first-class phantom types* [4, 8], and *equality-qualified types* [15]) to give each constructor in a data type representing a set of abstract values a type equality that represents its static type from the source program. When introducing such a constructor, we must establish that the type equality holds; conversely, when eliminating such a constructor (in a match of a `case`-expression), we may assume that the type equality holds. Ultimately, the evidence required to justify that certain cases in a "down-cast" are "impossible" will arise from a contradiction (e.g., $\mathtt{Int} \sim \mathtt{Bool}$) derivable from the type equalities in scope. Because the static type of an abstract value or a receiving variable may be expressed in terms of type variables and it is sometimes necessary to reason about the types at which those type variables may be instantiated when filtering (as is the case in down-casting the result of the call of id @$(\beta{\to}\delta)$ f to g), we also use GADTs to represent the type-flow information. For each type variable in scope in the source program there is both a corresponding type variable in the target program and a corresponding "information" expression variable. The type of the "info" variable is a GADT that has a constructor for each of the type expressions at which the type variable may be instantiated; again, each constructor is given a type equality that represents the corresponding type from the source program. By performing a case analysis on the value of such an "info" expression variable, we can reason about each of the (source) types at which the type variable may be instantiated. Although this amounts to a form of run-time type passing, we emphasize that *no* dynamic type tests are performed during evaluation of the defunctionalized program; *every* case analysis of an "info" expression variable is on a code path that must lead to a contradiction — hence, the code path must never be executed during evaluation.

Figure 1 presents selected components of our type- and control-flow directed defunctionalization of the example program, emphasizing the first-order function $app'''C$ representing the first-class function λz from the source program.

Each distinct set of type expressions $\hat{T}$ that arises in the flow analysis becomes a distinct GADT declaration $\mathsf{Ty}_{\hat{T}}$. A source type variable v that is mapped by the flow analysis to the set $\hat{T}$ is translated by defunctionalization to a target type variable v and a target expression variable i_v of type $\mathsf{Ty}_{\hat{T}}(v)$; the latter is an explicit value representing the type at which the former has been instantiated. For example, the set $\{\mathtt{Int}{\to}\mathtt{Int}{\to}\mathtt{Int}, \beta{\to}\delta\}$ is represented by the GADT declaration:

$$\begin{aligned}\mathtt{data}\ \mathsf{Ty}_1(\alpha_1)\ \{&\mathsf{III}_1()\,[\alpha_1 \sim \mathsf{Arr}(\mathtt{Int}, \mathsf{Arr}(\mathtt{Int}, \mathtt{Int}))\,](),\\ &\mathsf{BD}_1(\beta,\delta)\,[\alpha_1 \sim \mathsf{Arr}(\beta,\delta)\,](\mathsf{Ty}_2(\beta), \mathsf{Ty}_3(\delta))\,\}\end{aligned}$$

(Note that a GADT declaration is comprised of a type constructor, zero or more universal (parametric) type variables and a set of zero or more constructors; each constructor is comprised of zero or more existential type variables, zero or more type equality constraints, and zero or more types of carried data.) In each constructor of the Ty_1 data type, corresponding to a type expression $\tau \in \{\mathtt{Int}{\to}\mathtt{Int}{\to}\mathtt{Int}, \beta{\to}\delta\}$, the type parameter α_1 is used in an equality constraint to assert that α_1 is equal to $[\![\tau]\!]_{\mathsf{R}}$, where $[\![\cdot]\!]_{\mathsf{R}}$ computes a type-level representation of the source type τ. The target program declares the (uninhabited) data types Arr, Forall, Z (zero), and S (successor) to represent function and universal types (using de Bruijn indices for $\forall$-bound type variables). When the type expression has free type variables, the corresponding constructor uses existential type variables and carries data that represents the type at which the free type variables have been instantiated. For example, the constructor BD_1 corresponding to $\beta{\to}\delta$ has existential type variables β and δ, an equality constraint $\alpha_1 \sim \mathsf{Arr}(\beta,\delta)$, and carries data of types $\mathsf{Ty}_2(\beta)$ and $\mathsf{Ty}_3(\delta)$. The type of the BD_1 constructor is essentially

$$\forall(\alpha_1).\ \forall(\beta,\delta).\ [\alpha_1 \sim \mathsf{Arr}(\beta,\delta)] \Rightarrow (\mathsf{Ty}_2(\beta), \mathsf{Ty}_3(\delta)) \to \mathsf{Ty}_1(\alpha_1)$$

with the caveat that constructors in the target language must always be fully applied and the type equalities must be satisfied at the point of application.

Similarly, each distinct set of abstract values $\hat{V}$ that arises in the flow analysis becomes a distinct GADT declaration $\mathsf{Val}_{\hat{V}}$. A source expression variable y of type τ that is mapped by the flow analysis to the set $\hat{V}$ will be translated by defunctionalization to a target expression variable y of type $\mathsf{Val}_{\hat{V}}([\![\tau]\!]_{\mathsf{R}})$. For example, the set of abstract values $\{\lambda a1, \lambda b1, \lambda c1\}$ is represented by the GADT declaration:

$$\begin{aligned}\mathtt{data}\ \mathsf{Val}_3(\alpha_3)\ \{&\mathsf{Add}_3()\,[\alpha_3 \sim \mathsf{Arr}(\mathtt{Int}, \mathsf{Arr}(\mathtt{Int}, \mathtt{Int}))\,](),\\ &\mathsf{Mul}_3()\,[\alpha_3 \sim \mathsf{Arr}(\mathtt{Int}, \mathsf{Arr}(\mathtt{Int}, \mathtt{Int}))\,](),\\ &\mathsf{MInc}_3()\,[\alpha_3 \sim \mathsf{Arr}(\mathtt{Bool}, \mathsf{Arr}(\mathtt{Int}, \mathtt{Int}))\,]()\,\}\end{aligned}$$

In each constructor of the Val_3 data type declaration, corresponding to an abstract value in $\{\lambda a1, \lambda b1, \lambda c1\}$, the type parameter α_3 is used in an equality constraint to assert that α_3 is equal to the representation of the static type of corresponding abstract value. For example, the constructor Add_3 has the equality constraint $\alpha_3 \sim \mathsf{Arr}(\mathtt{Int}, \mathsf{Arr}(\mathtt{Int}, \mathtt{Int}))$ because it corresponds to $\lambda a1$ with static type $\mathtt{Int}{\to}\mathtt{Int}{\to}\mathtt{Int}$. When the abstract value has free type and expression variables, the corresponding constructor uses existential type variables and carries data that represents the type at which the free type variables have been instantiated and the values for the free expression variables. For example, the set of abstract values $\{\lambda z\}$, which arises in the flow analysis as the result of the λf function, is represented by the GADT declaration:

$$\begin{aligned}\mathtt{data}\ \mathsf{Val}_7(\alpha_7)\ \{&\mathsf{App'''}_7(\beta,\delta)\\ &\quad[\alpha_7 \sim \mathsf{Arr}(\beta,\delta)\,]\\ &\quad(\mathsf{Ty}_2(\beta), \mathsf{Ty}_3(\delta),\\ &\quad\ \mathsf{Val}_1(\mathsf{Forall}(\mathsf{Arr}(\mathsf{Z}(), \mathsf{Z}()))), \mathsf{Val}_4(\mathsf{Arr}(\beta,\delta)))\,\}\end{aligned}$$

where λz has free type variables β (mapped to $\{\mathtt{Bool}\}$ by the flow analysis, which is represented by Ty_2) and δ (mapped to $\{\mathtt{Int}{\to}\mathtt{Int}\}$, represented by Ty_3) and free expression variables id (with type $\forall\alpha.\ \alpha{\to}\alpha$ and mapped to $\{\Lambda\alpha\}$, represented by Val_1) and f (with type $\beta{\to}\delta$ and mapped to $\{\lambda c1\}$, represented by Val_4).

With these GADT declarations, we can now examine the first-order function $app'''C$, representing the first-class function λz, with free type variables β and δ and free expression variables id

```
data Arr($\alpha_a, \alpha_r$) {}
data Forall($\alpha_r$) {}
data Z() {}
data S($\alpha_k$) {}
data Ty$_1$($\alpha_1$) {III$_1$()[$\alpha_1 \sim$ Arr(Int, Arr(Int, Int))](), BD$_1$($\beta, \delta$)[$\alpha_1 \sim$ Arr($\beta, \delta$)](Ty$_2$($\beta$), Ty$_3$($\delta$))}
data Ty$_2$($\alpha_2$) {B$_2$()[$\alpha_2 \sim$ Bool]()}
data Ty$_3$($\alpha_3$) {II$_3$()[$\alpha_3 \sim$ Arr(Int, Int)]()}
...
data Val$_1$($\alpha_1$) {Id$_1$()[$\alpha_1 \sim$ Forall(Arr(Z(), Z()))]()}
data Val$_2$($\alpha_2$) {Id$'_2$($\alpha$)[$\alpha_2 \sim$ Arr($\alpha, \alpha$)](Ty$_1$($\alpha$))}
data Val$_3$($\alpha_3$) {Add$_3$()[$\alpha_3 \sim$ Arr(Int, Arr(Int, Int))](), Mul$_3$()[$\alpha_3 \sim$ Arr(Int, Arr(Int, Int))](), MInc$_3$()[$\alpha_3 \sim$ Arr(Bool, Arr(Int, Int))]()}
data Val$_4$($\alpha_4$) {MInc$_4$()[$\alpha_4 \sim$ Arr(Bool, Arr(Int, Int))]()}
data Val$_5$($\alpha_5$) {BoolV$_5$()[$\alpha_5 \sim$ Bool](Bool)}
data Val$_6$($\alpha_6$) {MInc$'_6$()[$\alpha_6 \sim$ Arr(Int, Int)](Val$_5$(Bool))}
data Val$_7$($\alpha_7$) {App$'''_7$($\beta, \delta$)[$\alpha_7 \sim$ Arr($\beta, \delta$)](Ty$_2$($\beta$), Ty$_3$($\delta$), Val$_1$(Forall(Arr(Z(), Z()))), Val$_4$(Arr($\beta, \delta$)))}
... ;
fun idC($\alpha$)($i_\alpha$:Ty$_1$($\alpha$)):Val$_2$(Arr($\alpha, \alpha$)) = Id$'_2$(Arr($\alpha, \alpha$))($\alpha$)[Arr($\alpha, \alpha$) $\sim$ Arr($\alpha, \alpha$)]($i_\alpha$)
fun id$'$C($\alpha$)($i_\alpha$:Ty$_1$($\alpha$), x:Val$_3$($\alpha$)):Val$_3$($\alpha$) = x
...
fun app$'''$C($\beta, \delta$)($i_\beta$:Ty$_2$($\beta$), $i_\delta$:Ty$_3$($\delta$), id:Val$_1$(Forall(Arr(Z(), Z()))), f:Val$_4$(Arr($\beta, \delta$)), z:Val$_5$($\beta$)):Val$_6$($\delta$) =
  let g = case id of
            Id$_1$()[Forall(Arr(Z(), Z())) $\sim$ Forall(Arr(Z(), Z()))]() $\Rightarrow$
              let $i'_\alpha$ = BD$_1$(Arr($\beta, \delta$))($\beta, \delta$)[Arr($\beta, \delta$) $\sim$ Arr($\beta, \delta$)]($i_\beta, i_\delta$) in
              case idC(Arr($\beta, \delta$))($i'_\alpha$) of
                Id$'_2$($\alpha'$)[Arr(Arr($\beta, \delta$), Arr($\beta, \delta$)) $\sim$ Arr($\alpha', \alpha'$)]($i_{\alpha'}$) $\Rightarrow$
                  let x$'$ = case f of
                               MInc$_4$()[Arr($\beta, \delta$) $\sim$ Arr(Bool, Arr(Int, Int))]() $\Rightarrow$ MInc$_3$($\alpha'$)()[$\alpha' \sim$ Arr(Bool, Arr(Int, Int))]() in
                  case id$'$C($\alpha'$)($i_{\alpha'}$, x$'$) of
                    Add$_3$()[$\alpha' \sim$ Arr(Int, Arr(Int, Int)]() $\Rightarrow$ (case $i_\beta$ of B$_2$()[$\beta \sim$ Bool]() $\Rightarrow$ abort)
                    Mul$_3$()[$\alpha' \sim$ Arr(Int, Arr(Int, Int)]() $\Rightarrow$ (case $i_\beta$ of B$_2$()[$\beta \sim$ Bool]() $\Rightarrow$ abort)
                    MInc$_3$()[$\alpha' \sim$ Arr(Bool, Arr(Int, Int)]() $\Rightarrow$ MInc$_4$(Arr($\beta, \delta$))()[Arr($\beta, \delta$) $\sim$ Arr(Bool, Arr(Int, Int)]() in
  case g of
    MInc$_4$()[Arr($\beta, \delta$) $\sim$ Arr(Bool, Arr(Int, Int)) $\Rightarrow$ mincC(c1)
...
fun mincC()(c1:Val$_5$(Bool)):Val$_6$(Arr(Int, Int)) = MInc$'_6$(Arr(Int, Int))()[Arr(Int, Int) $\sim$ Arr(Int, Int)](c1)
... ;
let id = Id$_1$(Forall(Arr(Z(), Z())))()[Forall(Arr(Z(), Z())) $\sim$ Forall(Arr(Z(), Z()))]() in
...
```

Figure 1: Type- and Control-Flow Directed Defunctionalization (selected components)

and f. A first-order, polymorphic function in the target language is comprised of name, zero or more type variables, zero or more expression variables (with types), a result type, and a body expression. Note that the type and expression arguments of *app$'''$C* correspond exactly to the free type variables and free expression variables of λz plus the formal argument z. The first step is to compute *id* @($\beta{\to}\delta$); to do so in the defunctionalized program, a case analysis is performed on *id*. Since $\hat{\rho}(id) = \{\Lambda\alpha\}$, the exhaustive case analysis has exactly one match, indicating that the first-order function *idC* should be called. When performing a type application in the defunctionalized program, an explicit value representing the type used for instantiation is passed; the BD$_1$ constructor is used to build the representation of the type expression $\beta{\to}\delta$ in the set $\{$Int$\to$Int$\to$Int$, \beta{\to}\delta\}$, to which the flow analysis maps α.

The next step is to compute $\square\ f$, where $\square$ corresponds to the result of *id* @($\beta{\to}\delta$); again, to do so in the defunctionalized program, a case analysis is performed on the result of the call of *idC*. Once again, the exhaustive case analysis has exactly one match, indicating that the first-order function *id$'$C* should be called. Note that the Id$'_2$ constructor has an existential type variable α' that records the type at which *id* was instantiated and the type equality recovers that Arr(β, δ) $\sim \alpha'$. The actual argument in this call is f, where $\hat{\rho}(f) = \{\lambda c1\}$ (represented by Val$_4$), but the formal parameter in this call is x, where $\hat{\rho}(x) = \{\lambda a1, \lambda b1, \lambda c1\}$ (represented by Val$_3$). Thus, we must "up-cast" from f to x, which is easily achieved because the MInc$_4$ constructor of Val$_4$ can be trivially converted to the MInc$_3$ constructor of Val$_3$.

Next, the result of *id* @($\beta{\to}\delta$) f must be bound to g. Note that, in the source program, g has the type $\beta{\to}\delta$ and $\hat{\rho}(g) = \{\lambda c1\}$; hence, in the target program, g should have the type Val$_4$(Arr(β, δ)). However, flow analysis determines that the result of λx is $\{\lambda a1, \lambda b1, \lambda c1\}$ and, therefore, the result of *id$'$C*(α')($i_{\alpha'}, x'$) is Val$_3$(α'). Thus, we must "down-cast" from Val$_3$(α') to Val$_4$(Arr(β, δ). From above, we have the type equality Arr(β, δ) $\sim \alpha'$. In the MInc$_3$ match, we have the type equality $\alpha' \sim$ Arr(Bool, Arr(Int, Int); transitivity and injectivity establish that $\beta \sim$ Bool and $\delta \sim$ Arr(Int, Int), which suffices to construct a value of type $\mathsf{Val}_4(\mathsf{Arr}(\beta, \delta)$ with the MInc_4 constructor. In the Add_3 and Mul_3 matches, we have the type equality $\alpha' \sim \mathsf{Arr}(\mathtt{Int}, \mathsf{Arr}(\mathtt{Int}, \mathtt{Int})$; transitivity and injectivity establish that $\beta \sim \mathtt{Int}$ and $\delta \sim \mathsf{Arr}(\mathtt{Int}, \mathtt{Int})$. Performing an exhaustive case analysis on i_β of type $\mathsf{Ty}_2(\beta)$ requires a single B_2 match, which introduces the type equality $\beta \sim \mathtt{Bool}$; transitivity establishes that $\mathtt{Int} \sim \mathtt{Bool}$, which is a contradiction. Thus, in the Add_3 and Mul_3 matches, we use the `abort` expression, which is well-typed (with any type) only under inconsistent type equalities, to "prove" that this is unreachable, dead code.

The final step is to compute $g\ z$; in the defunctionalized program, the exhaustive case analysis of g has exactly one match, indicating that the first-order function *mincC* should be called.

1.1 Contributions

The full version of the paper will precisely define this type- and control-flow directed defunctionalization. The source language for our defunctionalization transformation is a variant of System F with integers and recursive functions; the static semantics of the language combines a type system and the type- and control-flow analysis as a single syntax-directed judgement. The target language for our defunctionalization transformation is comprised of (mutually recursive) GADT declarations, (mutually recursive) first-order, polymorphic, type-equality parameterized functions, and a main expression. The defunctionalization transformation is primarily defined by induction on the derivation of the source program's type system and type- and control-flow analysis judgement.

A subtle aspect of the translation, not illustrated by the example above, is that there may be "loops" in the type-flow information. For example, we may have $\hat{\rho}(\alpha) = \{\mathtt{Int}, \alpha{\rightarrow}\alpha\}$ (represented by Ty_a) and $\hat{\rho}(\beta) = \{\mathtt{Bool}, \beta{\rightarrow}\beta\}$ (represented by Ty_b). Given the type expressions at which α and β can be instantiated, the type equality $\alpha \sim \beta$ should imply a contradiction. In order to establish the contradiction, we require an *inductive* proof, which will be represented by a recursive function that examines arguments of type $\mathsf{Ty}_a(\alpha)$ and $\mathsf{Ty}_b(\beta)$. The soundness of this proof relies on the finiteness of the type-flow information and the decidability of type incompatibility.

REFERENCES

[1] Connor Adsit and Matthew Fluet. 2014. An Efficient Type- and Control-Flow Analysis for System F. In *IFL'14: Proceedings of the 26nd International Symposium on Implementation and Application of Functional Languages*, Sam Tobin-Hochstadt (Ed.). Association for Computing Machinery, Boston, MA, USA, Article 3, 14 pages.

[2] Jeffrey M. Bell, Françoise Bellegarde, and James Hook. 1997. Type-Driven Defunctionalization. In *ICFP'97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, Mads Tofte (Ed.). Association for Computing Machinery, Amsterdam, The Netherlands, 25–37.

[3] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-directed closure conversion for typed languages. In *ESOP'00: Proceedings of the Ninth European Symposium on Programming (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer-Verlag, Berlin, Germany, 56–71.

[4] James Cheney and Ralf Hinze. 2003. *First-class Phantom Types.* Technical Report TR2003-1901. Cornell University, Ithaca, NY, USA.

[5] Matthew Fluet. 2013. A Type- and Control-Flow Analaysis for System F. In *IFL'12: Post-Proceedings of the 24th International Symposium on Implementation and Application of Functional Languages (Lecture Notes in Computer Science)*, Ralf Hinze (Ed.). Springer-Verlag, Oxford, England, 122–139.

[6] Georgios Fourtounis and Nikolaos S. Papaspyrou. 2013. Supporting separate compilation in a defunctionalizing compiler. In *SLATE'13: Proceedings of the Second Symposium on Languages, Applications and Technologies (OASICS, Vol. 29)*, José Paulo Leal, Ricardo Rocha, and Alberto Simões (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[7] Georgios Fourtounis, Nikolaos S. Papaspyrou, and Panagiotis Theofilopoulos. 2014. Modular Polymorphic Defunctionalization. *Computer Science and Information Systems* 11, 4 (2014), 1417–1434.

[8] Ralf Hinze. 2003. Fun with Phantom Types. In *The Fun of Programming*, Jeremy Gibbons and Oege de Moor (Eds.). Palgrave Macmillan, 245–262.

[9] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer-Verlag.

[10] François Pottier and Nadji Gauthier. 2004. Polymorphic typed defunctionalization. In *POPL'04: Proceedings of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Xavier Leroy (Ed.). Association for Computing Machinery, Venice, Italy, 89–98.

[11] François Pottier and Nadji Gauthier. 2006. Polymorphic Typed Defunctionalization and Concretization. *Higher-Order and Symbolic Computation* 19, 1 (2006), 125–162.

[12] John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *ACM'72: Proceedings of 25th ACM National Conference*, Rosemary Shields (Ed.). Association for Computing Machinery, Boston, MA, USA, 717–740. Reprinted as [13], with a foreword [14].

[13] John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation* 11, 4 (1998), 363–397. Reprinting of [12].

[14] John C. Reynolds. 1998. Definitional Interpreters Revisited. *Higher-Order and Symbolic Computation* 11, 4 (1998), 355–361.

[15] Tim Sheard. 2004. Languages of the Future. In *OOPSLA'04: Proceedings of the 2004 ACM International Conference on Object Oriented Programming Systems, Languages, and Applications*, Doug Schmidt (Ed.). Association for Computing Machinery, Vancouver, BC, Cananda, 116–119.

[16] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Technical Report CMU-CS-91-145.

[17] Andrew Tolmach. 1997. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97).* Amsterdam, The Netherlands. Available as technical report BCCS-97-03, Computer Science Department, Boston College.

[18] Andrew Tolmach and Dino P. Oliva. 1998. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming* 8, 4 (1998), 367–412.

[19] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *POPL'03: Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Greg Morrisett (Ed.). Association for Computing Machinery, New Orleans, LA, USA, 224–235.

Towards a more perfect union type

Anonymous Author(s)

Abstract

We present a principled theoretical framework for inferring and checking the union types, and show its work in practice on JSON data structures.

The framework poses a union type inference as a learning problem from multiple examples. The categorical framework is generic and easily extensible.

1 Introduction

Typing dynamic languages has been long considered a challenge [3]. The importance of the task grown with the ubiquity cloud application programming interfaces (APIs) utilizing JavaScript object notation (JSON), where one needs to infer the structure having only a limited number of sample documents available. Previous research has suggested it is possible to infer adequate type mappings from sample data [2, 8, 14, 20].

In the present study, we expand on these results. We propose a modular framework for type systems in programming languages as learning algorithms, formulate it as equational identities, and evaluate its performance on inference of Haskell data types from JSON API examples.

1.1 Related work

1.1.1 Union type providers

The earliest practical effort to apply union types to JSON inference to generate Haskell types [14]. It uses union type theory, but it also lacks an extensible theoretical framework. F# type providers for JSON facilitate deriving a schema automatically; however, a type system does not support union of alternatives and is given shape inference algorithm, instead of design driven by desired properties [20]. The other attempt to automatically infer schemas has been introduced in the PADS project [8]. Nevertheless, it has not specified a generalized type-system design methodology. One approach uses Markov chains to derive JSON types [2][1]. This approach requires considerable engineering time due to the implementation of unit tests in a case-by-case mode, instead of formulating laws applying to all types. Moreover, this approach lacks a sound underlying theory. Regular expression types were also used to type XML documents [13], which does not allow for selecting alternative representation. In the present study, we generalize previously introduced approaches and enable a systematic addition of not only value sets, but inference subalgorithms, to the union type system.

[1]This approach uses Markov chains to infer best of alternative type representations.

GPCE, November, 2020, Illinois, USA
2020.

1.1.2 Frameworks for describing type systems

Type systems are commonly expressed as partial relation of *typing*. Their properties, such as subject reduction are also expressed relatively to the relation (also partial) of *reduction* within a term rewriting system. General formulations have been introduced for the Damas-Milner type systems parameterized by constraints [23]. It is also worth noting that traditional Damas-Milner type disciplines enjoy decidability, and embrace the laws of soundness, and subject-reduction. However these laws often prove too strict during type system extension, dependent type systems often abandon subject-reduction, and type systems of widely used programming languages are either undecidable [21], or even unsound [27].

Early approaches used lattice structure on the types [25], which is more stringent than ours since it requires idempotence of unification (as join operation), as well as complementary meet operation with the same properties. Semantic subtyping approach provides a characterization of a set-based union, intersection, and complement types [9, 10], which allows model subtype containment on first-order types and functions. This model relies on building a model using infinite sets in set theory, but its rigidity fails to generalize to non-idempotent learning[2]. We are also not aware of a type inference framework that consistently and completely preserve information in the face of inconsistencies nor errors, beyond using `bottom` and expanding to *infamous undefined behaviour* [5].

We propose a categorical and constructive framework that preserves the soundness in inference while allowing for consistent approximations. Indeed our experience is that most of the type system implementation may be generic.

2 Motivation

Here, we consider several examples similar to JSON API descriptions. We provide these examples in the form of a few JSON objects, along with desired representation as Haskell data declaration.

1. Subsets of data within a single constructor:
   a. *API argument is an email* – it is a subset of valid `String` values that can be validated on the client-side.

[2]Which would allow extension with machine learning techniques like Markov chains to infer optimal type representation from frequency of occuring values[2].

b. *The page size determines the number of results to return (min: 10, max:10,000)* – it is also a subset of integer values (Int) between 10, and 10,000
c. *The* date *field contains ISO8601 date* – a record field represented as a String that contains a calendar date in the format "2019-03-03"

2. Optional fields: *The page size is equal to 100 by default* – it means we expect to see the record like {"page_size": 50} or an empty record {} that should be interpreted in the same way as {"page_size": 100}
3. Variant fields: *Answer to a query is either a number of registered objects, or String* "unavailable" - this is integer value (Int) or a String (Int :|: String)
4. Variant records: *Answer contains either a text message with a user identifier or an error.* – That can be represented as one of following options:

```
{"message" : "Where can I submit my proposal?", "uid" : 1014}
{"message" : "Submit it to HotCRP",              "uid" :  317}
{"error"   : "Authorization failed",            "code":  401}
{"error"   : "User not found",                  "code":  404}
```

```
data Example4 = Message { message :: String, uid  :: Int }
              | Error   { error   :: String, code :: Int }
```

5. Arrays corresponding to records:

```
[ [1, "Nick",     null        ]
, [2, "George", "2019-04-11"]
, [3, "Olivia", "1984-05-03"] ]
```

6. Maps of identical objects (example from [2]):

```
{   "6408f5": { "size":       969709     , "height":     510599
              , "difficulty": 866429.732, "previous": "54fced" },
    "54fced": { "size":       991394     , "height":     510598
              , "difficulty": 866429.823, "previous": "6c9589" },
    "6c9589": { "size":       990527     , "height":     510597
              , "difficulty": 866429.931, "previous": "51a0cb" } }
```

It should be noted that the last example presented above requires Haskell representation inference to be non-monotonic, as an example of object with only a single key would be best represented by a record type:

```
data Example = Example { f_6408f5 :: O_6408f5, f_54fced :: O_6408f5
                       , f_6c9589 :: O_6408f5 }
data O_6408f5 = O_6408f5 { size, height :: Int, difficulty :: Double
                         , previous     :: String }
```

However, when this object has multiple keys with values of the same structure, the best representation is that of a mapping shown below. This is also an example of when user may decide to explicitly add evidence for one of the alternative representations in the case when input samples are insufficient. (like when input samples only contain a single element dictionary.)

```
data ExampleMap = ExampleMap (Map Hex ExampleElt)
data ExampleElt = ExampleElt { size :: Int, height :: Int
  , difficulty :: Double, previous :: String }
```

2.1 Goal of inference

Given an undocumented (or incorrectly labelled) JSON API, we may need to read the input of Haskell encoding and avoid checking for the presence of *unexpected* format deviations. At the same time, we may decide to accept all known valid inputs outright so that we can use types[3] to ensure that the input is processed exhaustively.

Accordingly, we can assume that the smallest non-singleton set is a better approximation type than a singleton set. We call it *minimal containing set principle.*

Second, we can prefer types that allow for a fewer number of *degrees of freedom* compared with the others, while conforming to a commonly occurring structure. We denote it as an *information content principle.*

Given these principles, and examples of frequently occurring patterns, we can infer a reasonable *world of types* that approximate sets of possible values. In this way, we can implement *type system engineering* that allows deriving type system design directly from the information about data structures and the likelihood of their occurrence.

3 Problem definition

As we focus on JSON, we utilize Haskell encoding of the JSON term for convenient reading(from Aeson package [1]); specified as follows:

```
data Value = Object (Map String Value) | Array [Value] | Null
           | Number Scientific         | String Text   | Bool Bool
```

3.1 Defining type inference

3.1.1 Information in the type descriptions

If an inference fails, it is always possible to correct it by introducing an additional observation (example). To denote unification operation, or **information fusion** between two type descriptions, we use a Semigroup interface operation <> to merge types inferred from different observations. If the semigroup is a semilattice, then <> is meet operation (least upper bound). Note that this approach is dual to traditional unification that *narrows down* solutions and thus is join operation (greatest lower bound). We use a neutral element of the Monoid to indicate a type corresponding to no observations.

```
class Semigroup ty              where (<>)   :: ty -> ty -> ty
class Semigroup ty => Monoid ty where mempty :: ty
```

In other words, we can say that mempty (or bottom) element corresponds to situation where **no information was accepted** about a possible value (no term was seen, not even a null). It is a neutral element of Typelike. For example, an empty array [] can be referred to as an array type with mempty as an element type. This represents the view that <> always **gathers more information** about the type, as opposed to the traditional unification that always **narrows**

[3]Compiler feature of checking for unmatched cases.

down possible solutions. We describe the laws described below as QuickCheck [4] properties so that unit testing can be implemented to detect apparent violations.

3.1.2 Beyond set

In the domain of permissive union types, a beyond set represents the case of **everything permitted** or a fully dynamic value when we gather the information that permits every possible value inside a type. At the first reading, it may be deemed that a beyond set should comprise of only one single element – the top one (arriving at complete bounded semilattice), but this is too narrow for our purpose of *monotonically gathering information*

However, since we defined **generalization** operator <> as **information fusion** (corresponding to unification in categorically dual case of strict type systems.), we may encounter difficulties in assuring that no information has been lost during the generalization[4]. Moreover, strict type systems usually specify more than one error value, as it should contain information about error messages and keep track from where an error has been originated[5].

This observation lets us go well beyond typing statement of gradual type inference as a discovery problem from incomplete information [22]. Here we consider type inference as a **learning problem** furthermore, find common ground between the dynamic and the static typing discipline. The languages relying on the static type discipline usually consider beyond as a set of error messages, as a value should correspond to a statically assigned and a **narrow** type. In this setting, mempty would be fully polymorphic type $\forall a.a$.

Languages with dynamic type discipline will treat beyond as untyped, dynamic value and mempty again is an entirely unknown, polymorphic value (like a type of an element of an empty array)[6].

```
class (Monoid t, Eq t, Show t) => Typelike t where beyond :: t -> Bool
```

Besides, the standard laws for a **commutative** Monoid, we state the new law for the beyond set: The beyond set is always **closed to information addition** by (<>a) or (a<>) for any value of a, or **submonoid**. In other words, the beyond set is an attractor of <> on both sides.[7] However, we do not require *idempotence* of <>, which is uniformly present in union type frameworks based on the lattice [25] and set-based approaches[8][9]. Concerning union types, the key property of the beyond set, is that it is closed to information acquisition:

In this way, we can specify other elements of beyond set instead of a single top. When under strict type discipline, like that of Haskell [21], we seek to enable each element of the beyond set to contain at least one error message[9].

We abolish the semilattice requirement that has been conventionally assumed for type constraints [24], as this requirement is valid only for the strict type constraint inference, not for a more general type inference considered as a learning problem. As we observe in example 5 in sec. 2, we need to perform a non-monotonic step of choosing alternative representation after monotonic steps of merging all the information.

When a specific instance of Typelike is not a semilattice (an idempotent semigroup), we will explicitly indicate that is the case. It is convenient validation when testing a recursive structure of the type. Note that we abolish semilattice requirement that was traditionally assumed for type constraints here [25]. That is because this requirement is valid only for strict type constraint inference, not for a more general type inference as a learning problem. As we saw on ExampleMap in sec. 2, we need non-monotonic inference when dealing with alternative representations. We note that this approach significantly generalized the assumptions compared with a full lattice subtyping [24, 25].

Time to present the **relation of typing** and its laws. In order to preserve proper English word order, we state that ty `Types` val instead of classical val:ty. Specifying the laws of typing is important, since we may need to separately consider the validity of a domain of types/type constraints, and that of the sound typing of the terms by these valid types. The minimal definition of typing inference relation and type checking relation is formulated as consistency between these two operations.

```
class Typelike ty => ty `Types` val where
  infer ::       val -> ty
  check :: ty -> val -> Bool
```

First, we note that to describe *no information*, mempty cannot correctly type any term. A second important rule of typing is that all terms are typed successfully by any value in the beyond set. Finally, we state the most intuitive rule for typing: a type inferred from a term, must always be valid for that particular term. The law asserts that the diagram on the figure commutes:

The last law states that the terms are correctly type-checked after adding more information into a single type. (For inference relation, it would be described as *principal type property*.) The minimal Typelike instance is the one that contains only mempty corresponding to the case of *no sample data received*, and a single beyond element for *all values permitted*. We will define it below as PresenceConstraint in sec. 3.3.3. These laws are also compatible with the strict, static type discipline: namely, the beyond set corresponds to a set of constraints with at least one type error, and a

[4]Examples will be provided later.
[5]In this case: beyond (Error _) = True | otherwise = False.
[6]May sound similar until we consider adding more information to the type.
[7]So both for ∀a(<> a) and ∀a.(a<>), the result is kept in the beyond set.
[8]Which use Heyting algebras, which have more assumptions that the lattice approaches.
[9]Note that many but not all type constraints are semilattice. Please refer to the counting example below.

$$\begin{tikzcd}
 & \text{Type}_1 \times \text{Type}_2 \arrow[dl, "\pi_1"'] \arrow[d, "<>"] \arrow[dr, "\pi_2"] & \\
\text{Type}_1 \arrow[r, "<>\text{Type}_2"] \arrow[dr] & \text{Type}_1 <> \text{Type}_2 \arrow[d, bend right, "\text{check value}_1"'] \arrow[d, bend left, "\text{check value}_2"] & \text{Type}_2 \arrow[l, "\text{Type}_1<>"'] \arrow[dl] \\
\text{Value}_1 \arrow[u, "\text{infer}"] \arrow[r, "\text{check with } Type_1"'] & \text{True} & \text{Value}_2 \arrow[u, "\text{infer}"'] \arrow[l, "\text{check with } Type_2"]
\end{tikzcd}$$

Figure 1. Categorical diagram for Typelike.

task of a compiler to prevent any program with the terms that type only to the beyond as a least upper bound.

3.2 Type engineering principles

Considering that we aim to infer a type from a finite number of samples, we encounter a *learning problem*, so we need to use *prior* knowledge about the domain for inferring types. Observing that a : false we can expect that in particular cases, we may obtain that a : true. After noting that b : 123, we expect that b : 100 would also be acceptable. It means that we need to consider a typing system to *learn a reasonable general class from few instances*. This observation motivates formulating the type system as an inference problem. As the purpose is to deliver the most descriptive[10] types, we assume that we need to obtain a broader view rather than focusing on a *free type* and applying it to larger sets whenever it is deemed justified.

The other principle corresponds to **correct operation**. It implies that having operations regarded on types, we can find a minimal set of types that assure correct operation in the case of unexpected errors. Indeed we want to apply this theory to infer a type definition from a finite set of examples. We also seek to generalize it to infinite types. We endeavour rules to be as short as possible. The inference must also be a **contravariant** functor with regards to constructors. For example, if AType x y types {"a": X, "b": Y}, then x must type X, and y must type Y.

3.3 Constraint definition

3.3.1 Flat type constraints

Let us first consider typing of flat type: String (similar treatment should be given to the Number.type.)

```
data StringConstraint = SCDate              | SCEmail
  | SCEnum (Set Text) {- non-empty set of observed values -}
  | SCNever {- mempty -} | SCAny {- beyond -}

instance StringConstraint `Types` Text where
  infer (isValidDate  -> True) = SCDate
  infer (isValidEmail -> True) = SCEmail
  infer  value                 = SCEnum $ Set.singleton value
  infer  _                     = SCAny

  check  SCDate      s = isValidDate  s
  check  SCEmail     s = isValidEmail s
  check (SCEnum vs)  s = s `Set.member` vs
  check  SCNever     _ = False
  check  SCAny       _ = True

instance Semigroup StringConstraint where
  SCNever    <>  a                                  = a
  SCAny      <>  _                                  = SCAny
  SCDate     <>  SCDate                             = SCDate
  SCEmail    <>  SCEmail                            = SCEmail
  (SCEnum a) <> (SCEnum b) | length (a `Set.union` b) <= 10 =
  _          <>  _                                  = SCAny
```

[10]The shortest one according to the information complexity principle.

3.3.2 Free union type

Before we endeavour on finding type constraints for compound values (arrays and objects), it might be instructive to find a notion of *free type*, that is a type with no additional laws but the ones stated above. Given a term with arbitrary constructors we can infer a *free type* for every term set T as follows: For any T value type Set T satisfies our notion of *free type* specified as follows:

```
data FreeType a = FreeType { captured :: Set a } | Full

instance (Ord a, Eq a) => Semigroup (FreeType a) where
  Full <> _     = Full
  _    <> Full = Full
  a    <> b    = FreeType $ (Set.union `on` captured) a b
instance (Ord a, Eq a, Show a) => Typelike (FreeType a) where
  beyond = (==Full)

instance (Ord a, Eq a, Show a) => FreeType a `Types` a where
  infer                  = FreeType . Set.singleton
  check Full        _term = True
  check (FreeType s)  term = term `Set.member` s
```

This definition is deemed sound and applicable to finite sets of terms or values. For a set of values: ["yes", "no", "error"], we may reasonably consider that type is an appropriate approximation of C-style enumeration, or Haskell-style ADT without constructor arguments. However, the deficiency of this notion of *free type* is that it does not allow generalizing in infinite and recursive domains! It only allows to utilize objects from the sample.

3.3.3 Presence and absence constraint

We call the degenerate case of Typelike a *presence or absence constraint*. It just checks that the type contains at least one observation of the input value or no observations at all. It is vital as it can be used to specify an element type of an empty array. After seeing true value, we also expect false, so we can say that it is also a primary constraint for pragmatically indivisible like the set of boolean values. The same observation is valid for null values, as there is only one null value ever to observe.

```
type BoolConstraint       = PresenceConstraint Bool
type NullConstraint       = PresenceConstraint ()
data PresenceConstraint a = Present | Absent
```

Variants It is simple to represent a variant of two *mutually exclusive* types. They can be implemented with a type related to Either type that assumes these types are exclusive, we denote it by :|:. In other words for Int :|: String type, we first control whether the value is an Int, and if this check fails, we attempt to check it as a String. Variant records are slightly more complicated, as it may be unclear which typing is better to use:

```
{"message": "Where can I submit my proposal?", "uid" : 1014}
{"error"  : "Authorization failed",            "code":  401}
```

```
data OurRecord = OurRecord { message, error :: Maybe String
                           , code,    uid   :: Maybe Int }
```

```
data OurRecord2 = Message  { message :: String, uid  :: Int }
                | Error    { error   :: String, code :: Int }
```

The best attempt here is to rely on the available examples being reasonably exhaustive. That is, we can estimate how many examples we have for each, and how many of them match. Then, we compare this number with type complexity (with options being more complex to process because they need additional case expression.) In such cases, the latter definition has only one Maybe field (on the toplevel optionality is one), while the former definition has four Maybe fields (optionality is four). When we obtain more samples, the pattern emerges:

```
{"error"  : "Authorization failed",            "code":  401}
{"message": "Where can I submit my proposal?", "uid" : 1014}
{"message": "Sent it to HotCRP",               "uid" :   93}
{"message": "Thanks!",                         "uid" : 1014}
{"error"  : "Missing user",                    "code":  404}
```

Type cost function Since we are interested in types with less complexity and less optionality, we will define cost function as follows:

```
class Typelike ty => TypeCost ty where
  typeCost ::  ty -> TyCost
  typeCost a = if a == mempty then 0 else 1
instance Semigroup TyCost where (<>)   = (+)
instance Monoid    TyCost where mempty = 0

newtype TyCost = TyCost Int
```

When presented with several alternate representations from the same set of observations, we will use this function to select the least complex representation of the type. For flat constraints as above, we infer that they offer no optionality when no observations occurred (cost of mempty is 0), otherwise, the cost is 1. Type cost should be non-negative, and non-decreasing when we add new observations to the type.

3.3.4 Object constraint

To avoid information loss, a constraint for JSON object type is introduced in such a way to **simultaneously gather information** about representing it either as a Map, or a record. The typing of Map would be specified as follows, with the optionality cost being a sum of optionalities in its fields.

```
data MappingConstraint = MappingNever -- mempty
  | MappingConstraint { keyConstraint   :: StringConstraint
                      , valueConstraint :: UnionType        }
instance TypeCost MappingConstraint where
  typeCost MappingNever            = 0
  typeCost MappingConstraint {..} = typeCost keyConstraint
                                  + typeCost valueConstraint
```

Separately, we acquire the information about a possible typing of a JSON object as a record of values. Note that RCTop never actually occurs during inference. That is, we could have represented the RecordConstraint as a Typelike with an empty beyond set. The merging of constraints would be simply merging of all column constraints.

```
data RecordConstraint =
    RCTop {- beyond -} | RCBottom {- mempty -}
  | RecordConstraint { fields :: HashMap Text UnionType }

instance RecordConstraint `Types` Object where
    infer = RecordConstraint    . Map.fromList
          . fmap (second infer) . Map.toList
    check RecordConstraint {fields} obj =
        all (`elem` Map.keys fields) -- all object keys
                (Map.keys  obj)       -- present in type
     && and (Map.elems $ Map.intersectionWith -- values check
                             check fields obj)
     && all isNullable (Map.elems $ fields `Map.difference` obj)
        -- absent values are nullable
```

Observing that the two abstract domains considered above are independent, we can store the information about both options separately in a record[11]. It should be noted that this representation is similar to *intersection type*: any value that satisfies ObjectConstraint must conform to both mappingCase, and recordCase. Also, this *intersection approach* in order to address alternative union type representations benefit from *principal type property*, meaning that a principal type serves to acquire the information corresponding to different representations and handle them separately. Since we plan to choose only one representation for the object, we can say that the minimum cost of this type is the minimum of component costs.

```haskell
data ObjectConstraint = ObjectNever -- mempty
  | ObjectConstraint { mappingCase :: MappingConstraint
                     , recordCase  :: RecordConstraint }

instance TypeCost ObjectConstraint where
  typeCost ObjectConstraint {..} = typeCost mappingCase
                            `min` typeCost recordCase
```

3.3.5 Array constraint

Similarly to the object type, ArrayConstraint is used to simultaneously obtain information about all possible representations of an array, differentiating between an array of the same elements, and a row with the type depending on a column. We need to acquire the information for both alternatives separately, and then, to measure a relative likelihood of either case, before mapping the union type to Haskell declaration. Here, we specify the records for two different possible representations:

```haskell
data ArrayConstraint = ArrayNever -- mempty
  | ArrayConstraint { rowCase :: RowConstraint, arrayCase :: UnionType }
```

Semigroup operation just merges information on the components, and the same is done when inferring types or checking them: For the arrays, we plan to choose again only one of possible representations, so the cost of optionality is the lesser of the costs of the representation-specific constraints.

```haskell
instance ArrayConstraint `Types` Array where
    infer vs = ArrayConstraint { rowCase = infer vs
      , arrayCase = mconcat (infer <$> Foldable.toList vs) }
    check ArrayNever            vs = False
    check ArrayConstraint {..} vs = check rowCase vs
      && and (check arrayCase <$> Foldable.toList vs)
```

3.3.6 Row constraint

A row constraint is valid only if there is the same number of entries in all rows, which is represented by escaping the beyond set whenever there is an uneven number of columns. Row constraint remains valid only if both constraint describe the record of the same length; otherwise, we yield RowTop to indicate that it is no longer valid. In other words, RowConstraint is a *levitated semilattice*[16][12] with a neutral element over the content type that is a list of UnionType objects.

```haskell
data RowConstraint = RowTop | RowNever | Row [UnionType]
```

3.3.7 Combining the union type

It should note that given the constraints for the different type constructors, the union type can be considered as mostly a generic Monoid instance [11]. Merging information with <> and mempty follow the pattern above, by just lifting operations on the component.

```haskell
data UnionType = UnionType {
    unionNull :: NullConstraint,    unionBool :: BoolConstraint
  , unionNum  :: NumberConstraint, unionStr  :: StringConstraint
  , unionArr  :: ArrayConstraint,  unionObj  :: ObjectConstraint }
```

The generic structure of union type can be explained by the fact that the information contained in each record field is *independent* from the information contained in other fields. It means that we generalize independently over different dimensions[13]

Inference breaks down disjoint alternatives corresponding to different record fields, depending on the constructor of a given value. It enables implementing a clear and efficient treatment of different alternatives separately[14]. Since union type is all about optionality, we need to sum all options from different alternatives to obtain its typeCost.

```haskell
instance UnionType `Types` Value where
  infer (Bool   b) = mempty { unionBool = infer b  }
  infer  Null      = mempty { unionNull = infer () }
  infer (Number n) = mempty { unionNum  = infer n  }
  infer (String s) = mempty { unionStr  = infer s  }
  infer (Object o) = mempty { unionObj  = infer o  }
  infer (Array  a) = mempty { unionArr  = infer a  }

  check UnionType { unionBool } (Bool   b) = check unionBool b
  check UnionType { unionNull }  Null      = check unionNull ()
  check UnionType { unionNum  } (Number n) = check unionNum  n
  check UnionType { unionStr  } (String s) = check unionStr  s
  check UnionType { unionObj  } (Object o) = check unionObj  o
  check UnionType { unionArr  } (Array  a) = check unionArr  a
```

3.3.8 Overlapping alternatives

The essence of union type systems have long been dealing with the conflicting types provided in the input. Motivated by the examples above, we also aim to address conflicting

[11]The choice of representation will be explained later. Here we only consider acquiring information about possible values.

[12]*Levitated lattice* is created by appending distinct bottom and top to a set that does not possess them by itself.

[13]In this example, JSON terms can be described by terms without variables, and sets of tuples for dictionaries, so generalization by anti-unification is straightforward.

[14]The question may arise: what is the *union type* without *set union*? When the sets are disjoint, we just put the values in different bins to enable easier handling.

alternative assignments. It is apparent that examples 4. to 6. hint at more than one assignment: in example 5, a set of lists of values that may correspond to Int, String, or null, or a table that has the same (and predefined) type for each values; in example 6 A record of fixed names or the mapping from hash to a single object type.

3.3.9 Counting observations

In this section, we discuss how to gather information about the number of samples supporting each alternative type constraint. To explain this, the other example can be considered:

```
{"history": [
   {"error"  : "Authorization failed",            "code":  401}
  ,{"message": "Where can I submit my proposal?", "uid" : 1014}
  ,{"message": "Sent it to HotCRP",               "uid" :   93}
  ,{"message": "Thanks!",                         "uid" : 1014}
  ,{"error"  : "Authorization failed",            "code":  401}]}
```

First, we need to identify it as a list of similar elements. Second, there are multiple instances of each record example. We consider that the best approach would be to use the multisets of inferred records instead. To find the best representation, we can a type complexity, and attempt to minimize the term. Next step is to detect the similarities between type descriptions introduced for different parts of the term:

```
{"history"      : [...]
,"last_message" : {"message": "Thanks!", "uid" : 1014} }
```

We can add the auxiliary information about a number of samples observed, and the constraint will remain a Typelike object. The Counted constraint counts the number of samples observed for the constraint inside so that we can decide on which alternative representation is best supported by evidence. It should be noted that Counted constraint is the first example that does not correspond to a semilattice, that is a<>a≠a. This is natural for a Typelike object; it is not a type constraint in a conventional sense, just an accumulation of knowledge.

```
data Counted a = Counted { count :: Int, constraint :: a }

instance Semigroup a => Semigroup (Counted a) where
  a <> b = Counted (count      a +  count      b)
                   (constraint a <> constraint b)
```

Therefore, at each step, we may need to maintain a **cardinality** of each possible value, and is provided with sufficient number of samples, we may attempt to detect[15]. To preserve efficiency, we may need to merge whenever the number of alternatives in a multiset crosses the threshold. We can attempt to narrow strings only in the cases when cardinality crosses the threshold.

[15]If we detect a pattern too early, we risk to make the types too narrow to work with actual API responses.

4 Finishing touches

The final touch would be to perform the post-processing of an assigned type before generating it to make it more resilient to common uncertainties. These assumptions may bypass the defined least-upper-bound criterion specified in the initial part of the paper; however, they prove to work well in practice[2, 14].

If we have no observations corresponding to an array type, it can be inconvenient to disallow an array to contain any values at all. Therefore, we introduce a non-monotonic step of converting the mempty into a final Typelike object aiming to introduce a representation allowing the occurrence of any Value in the input. That still preserves the validity of the typing. We note that the program using our types must not have any assumptions about these values; however, at the same time, it should be able to print them for debugging purposes.

In most JSON documents, we observe that the same object can be simultaneously described in different parts of sample data structures. Due to this reason, we compare the sets of labels assigned to all objects and propose to unify those that have more than 60% of identical labels. For transparency, the identified candidates are logged for each user, and a user can also indicate them explicitly instead of relying on automation. We conclude that this allows considerably decreasing the complexity of types and makes the output less redundant.

5 Future work

In the present paper, we only discuss typing of tree-like values. However, it is natural to scale this approach to multiple types in APIs, in which different types are referred to by name and possibly contain each other. To address these cases, we plan to show that the environment of Typelike objects is also Typelike, and that constraint generalization (*anti-unification*) can be extended in the same way.

It should be noted that many Typelike instances for non-simple types usually follow one the two patterns of (1) for a finite sum of disjoint constructors, we bin this information by each constructor during the inference (2) for typing terms with multiple alternative representations, we infer all constraints separately for each alternative representation. In both cases, Generic derivation procedure for the Monoid, Typelike, and TypeCost instances is possible [17]. This allows us to design a type system by declaring datatypes themselves and leave implementation to the compiler. Manual implementation would be only left for special cases, like StringConstraint and Counted constraint.

Finally, we believe that we can explain the duality of categorical framework of Typelike categories and use generalization (anti-unification) instead of unification (or narrowing) as a type inference mechanism. The beyond set would then correspond to a set of error messages, and a result of the inference would represent a principal type in Damas-Milner sense.

5.1 Conclusion

In the present study, we aimed to derive the types that were valid with respect to the provided specification[16], thereby obtaining the information from the input in the most comprehensive way. We defined type inference as representation learning and type system engineering as a meta-learning problem in which the **priors corresponding to the data structure induced typing rules**. We show how the type safety can be quickly tested as equational laws with QuickCheck, which is a useful prototyping tool, and may be supplemented with fully formal proof in the future.

We also formulated the **union type discipline** as manipulation of Typelike commutative monoids, that represented knowledge about the data structure. In addition, we proposed a union type system engineering methodology that was logically justified by theoretical criteria. We demonstrated that it was capable of consistently explaining the decisions made in practice. We followed a strictly constructive procedure, that can be implemented generically.

We hope that this kind of straightforward type system engineering will become widely used in practice, replacing less modular approaches of the past. The proposed approach may be used to underlie the way towards formal construction and derivation of type systems based on the specification of value domains and design constraints.

[16]Specification was given in the motivation section descriptions of JSON input examples, and the expected results given as Haskell type declarations.

Bibliography

[1] Aeson: Fast JSON parsing and generation: 2011. *https://hackage.haskell.org/package/aeson*.

[2] A first look at quicktype: 2017. *https://blog.quicktype.io/first-look/*.

[3] Anderson, C. et al. 2005. Towards type inference for javascript. *ECOOP 2005 - object-oriented programming* (Berlin, Heidelberg, 2005), 428–452.

[4] Claessen, K. and Hughes, J. 2000. QuickCheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.* 35, 9 (Sep. 2000), 268–279. DOI:https://doi.org/10.1145/357766.351266.

[5] C Standard undefined behaviour versus Wittgenstein: *https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/*.

[6] C Undefined Behavior - Depressing and Terrifying (Updated): 2018. *https://www.yodaiken.com/2018/05/20/depressing-and-faintly-terrifying-days-for-the-c-standard/*.

[7] EnTangleD: A bi-directional literate programming tool: 2019. *https://blog.esciencecenter.nl/entangled-1744448f4b9f*.

[8] Fisher, K. and Walker, D. 2011. The PADS Project: An Overview. *Proceedings of the 14th international conference on database theory* (New York, NY, USA, 2011), 11–17.

[9] Frisch, A. et al. 2002. Semantic subtyping. *Proceedings 17th annual ieee symposium on logic in computer science* (2002), 137–146.

[10] Frisch, A. et al. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM.* 55, 4 (Sep. 2008). DOI:https://doi.org/10.1145/1391289.1391293.

[11] Generics example: Creating monoid instances: 2012. *https://www.yesodweb.com/blog/2012/10/generic-monoid*.

[12] GHCID - a new ghci based ide (ish): 2014. *http://neilmitchell.blogspot.com/2014/09/ghcid-new-ghci-based-ide-ish.html*.

[13] Hosoya, H. and Pierce, B. 2000. XDuce: A typed xml processing language. (Jun. 2000).

[14] JSON autotype: Presentation for Haskell.SG: 2015. *https://engineers.sg/video/json-autotype-1-0-haskell-sg--429*.

[15] Knuth, D.E. 1984. Literate programming. *Comput. J.* 27, 2 (May 1984), 97–111. DOI:https://doi.org/10.1093/comjnl/27.2.97.

[16] Lattices: Fine-grained library for constructing and manipulating lattices: 2017. *http://hackage.haskell.org/package/lattices-2.0.2/docs/Algebra-Lattice-Levitated.html*.

[17] Magalhães, J.P. et al. 2010. A generic deriving mechanism for haskell. *SIGPLAN Not.* 45, 11 (Sep. 2010), 37–48. DOI:https://doi.org/10.1145/2088456.1863529.

[18] Michal J. Gajda, D.K. 2020. Fast XML/HTML tools for Haskell: XML Typelift and improved Xeno. Manuscript under review.

[19] Pandoc: A universal document converter: 2000. *https://pandoc.org*.

[20] Petricek, T. et al. 2016. Types from Data: Making Structured Data First-Class Citizens in F#. *SIGPLAN Not.* 51, 6 (Jun. 2016), 477–490. DOI:https://doi.org/10.1145/2980983.2908115.

[21] Peyton Jones, S. 2019. Type inference as constraint solving: How ghc's type inference engine actually works. Zurihac keynote talk.

[22] Siek, J. and Taha, W. 2007. Gradual typing for objects. *Proceedings of the 21st european conference on object-oriented programming* (Berlin, Heidelberg, 2007), 2–27.

[23] Sulzmann, M. and Stuckey, P.j. 2008. Hm(x) Type Inference is Clp(x) Solving. *J. Funct. Program.* 18, 2 (Mar. 2008), 251–283. DOI:https://doi.org/10.1017/S0956796807006569.

[24] Tiuryn, J. 1992. Subtype inequalities. *[1992] Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science.* (1992), 308–315.

[25] Tiuryn, J. 1997. Subtyping over a lattice (abstract). *Computational logic and proof theory* (Berlin, Heidelberg, 1997), 84–88.

[26] Undefined behavior in 2017: 2017. *https://blog.regehr.org/archives/1520.*
[27] 2019. *https://github.com/microsoft/TypeScript/issues/9825.*

6 Appendix: all laws of Typelike

$$
\begin{array}{llllllcll}
 & & & & \text{check} & \text{mempty} & v = & \textbf{False} & \textit{(mempty contains no terms)} \\
\text{beyond} & t & & \Rightarrow & \text{check} & t & v = & \textbf{True} & \textit{(beyond contains all terms)} \\
\text{check} & t_1 & v & \Rightarrow & \text{check} & (t_1 \diamond t_2) & v = & \textbf{True} & \textit{(left fusion keeps terms)} \\
\text{check} & t_2 & v & \Rightarrow & \text{check} & (t_1 \diamond t_2) & v = & \textbf{True} & \textit{(right fusion keeps terms)} \\
 & & & & \text{check} & (\text{infer } v) & v = & \textbf{False} & \textit{(inferred type contains the source term)} \\
 & & & & & t_1 \diamond (t_2 \diamond t_3) & = & t_1 \diamond (t_2 \diamond t_3) & \textit{(semigroup associativity)} \\
 & & & & & \text{mempty} \diamond t & = & t & \textit{(left identity of the monoid)} \\
 & & & & & t \diamond \text{mempty} & = & t & \textit{(right identity of the monoid)}
\end{array}
$$

7 Appendix: definition module headers

```haskell
{-# language AllowAmbiguousTypes        #-}
{-# language DeriveGeneric              #-}
{-# language DuplicateRecordFields      #-}
{-# language FlexibleInstances          #-}
{-# language GeneralizedNewtypeDeriving #-}
{-# language MultiParamTypeClasses      #-}
{-# language NamedFieldPuns             #-}
{-# language PartialTypeSignatures      #-}
{-# language ScopedTypeVariables        #-}
{-# language TypeOperators              #-}
{-# language RoleAnnotations            #-}
{-# language ViewPatterns               #-}
{-# language RecordWildCards            #-}
{-# language OverloadedStrings          #-}
{-# options_ghc -Wno-orphans            #-}
module Unions where

import           Control.Arrow(second)
import           Data.Aeson
import           Data.Maybe(isJust,catMaybes)
import qualified Data.Foldable as Foldable
import           Data.Function(on)
import           Data.Text(Text)
import qualified Data.Text as Text
import qualified Data.Text.Encoding  as Text
import qualified Text.Email.Validate(isValid)
import qualified Data.Set  as Set
import           Data.Set(Set)
import           Data.Scientific
import           Data.String
import qualified Data.HashMap.Strict as Map
import           Data.HashMap.Strict(HashMap)
import           GHC.Generics       (Generic)
import           Data.Hashable
import           Data.Typeable
import           Data.Time.Format   (iso8601DateFormat,parseTimeM,defaultTimeLocale)
import           Data.Time.Calendar (Day)
import           Missing
```

```haskell
<<freetype>>
<<typelike>>
<<basic-constraints>>
<<row-constraint>>
<<array-constraint>>
<<object-constraint>>
<<presence-absence-constraints>>
<<union-type-instance>>
<<type>>
<<counted>>
<<typecost>>
<<representation>>
```

8 Appendix: test suite

```haskell
{-# language FlexibleInstances     #-}
{-# language Rank2Types            #-}
{-# language MultiParamTypeClasses #-}
{-# language MultiWayIf            #-}
{-# language NamedFieldPuns        #-}
{-# language ScopedTypeVariables   #-}
{-# language StandaloneDeriving    #-}
{-# language TemplateHaskell       #-}
{-# language TypeOperators         #-}
{-# language TypeApplications      #-}
{-# language TupleSections         #-}
{-# language UndecidableInstances  #-}
{-# language AllowAmbiguousTypes   #-}
{-# language OverloadedStrings     #-}
{-# language ViewPatterns          #-}
{-# options_ghc -Wno-orphans       #-}
module Main where

import qualified Data.Set            as Set
import qualified Data.Text           as Text
import qualified Data.ByteString.Char8 as BS
import           Control.Monad(when, replicateM)
import           Control.Exception(assert)
import           Data.FileEmbed
import           Data.Maybe
import           Data.Scientific
import           Data.Aeson
import           Data.Proxy
import           Data.Typeable
import qualified Data.HashMap.Strict as Map
import           Data.HashMap.Strict(HashMap)
import           Test.Hspec
import           Test.Hspec.QuickCheck
import           Test.QuickCheck
import           Test.Validity.Shrinking.Property
import           Test.Validity.Utils(nameOf)
import qualified GHC.Generics as Generic
import           Test.QuickCheck.Classes
import           System.Exit(exitFailure)

import Test.Arbitrary
import Test.LessArbitrary as LessArbitrary
import Unions

instance Arbitrary Value where
  arbitrary = fasterArbitrary

instance LessArbitrary Value where
  lessArbitrary = cheap $$$? genericLessArbitrary
    where
      cheap = LessArbitrary.oneof [
                 pure       Null
               , Bool   <$> lessArbitrary
               , Number <$> lessArbitrary
               ]

instance LessArbitrary          a
      => LessArbitrary (Counted a) where

instance LessArbitrary a
      => Arbitrary     (Counted a) where
  arbitrary = fasterArbitrary

instance Arbitrary Object where
  arbitrary = fasterArbitrary

instance Arbitrary Array where
  arbitrary = fasterArbitrary

class Typelike         ty
   => ArbitraryBeyond ty where
  arbitraryBeyond :: CostGen ty

instance ArbitraryBeyond (PresenceConstraint a) where
  arbitraryBeyond = pure Present

instance ArbitraryBeyond StringConstraint where
  arbitraryBeyond = pure SCAny

instance ArbitraryBeyond IntConstraint where
  arbitraryBeyond = pure IntAny

instance ArbitraryBeyond NumberConstraint where
  arbitraryBeyond = pure NCFloat

instance ArbitraryBeyond RowConstraint where
  arbitraryBeyond = pure RowTop

instance ArbitraryBeyond RecordConstraint where
  arbitraryBeyond = pure RCTop

instance ArbitraryBeyond MappingConstraint where
  arbitraryBeyond =
    MappingConstraint <$$$> arbitraryBeyond
                       <*>  arbitraryBeyond

instance (Ord                    a
         ,Show                   a
         )
      => ArbitraryBeyond (FreeType a) where
  arbitraryBeyond = pure Full

instance ArbitraryBeyond ObjectConstraint where
  arbitraryBeyond = do
    ObjectConstraint <$$$> arbitraryBeyond
                     <*>  arbitraryBeyond

instance ArbitraryBeyond ArrayConstraint where
  arbitraryBeyond = do
    ArrayConstraint <$$$> arbitraryBeyond
                    <*>  arbitraryBeyond

instance ArbitraryBeyond UnionType where
  arbitraryBeyond =
    UnionType      <$$$> arbitraryBeyond
                    <*>  arbitraryBeyond
                    <*>  arbitraryBeyond
                    <*>  arbitraryBeyond
                    <*>  arbitraryBeyond
                    <*>  arbitraryBeyond

instance ArbitraryBeyond          a
      => ArbitraryBeyond (Counted a) where
  arbitraryBeyond = Counted <$> LessArbitrary.choose (0, 10000)
                          <*> arbitraryBeyond

arbitraryBeyondSpec :: forall          ty.
                      (ArbitraryBeyond ty
                      ,Typelike        ty)
                    => Spec
arbitraryBeyondSpec =
  prop "arbitrarybeyond returns terms beyond" $
    (beyond <$> (arbitraryBeyond :: CostGen ty))

instance LessArbitrary Text.Text where
  lessArbitrary = Text.pack <$> lessArbitrary

instance Arbitrary Text.Text where
  arbitrary = Text.pack <$> arbitrary

instance Arbitrary Scientific where
  arbitrary = scientific <$> arbitrary
                         <*> arbitrary

instance (LessArbitrary          a
         ,Ord                    a)
    => LessArbitrary (FreeType a) where

instance Arbitrary (FreeType Value) where
  arbitrary = fasterArbitrary
  {-shrink  Full           = []
  shrink (FreeType elts) = map FreeType
                         $ shrink elts-}

instance (Ord             v
       ,Show             v)
    => TypeCost (FreeType v) where
  typeCost  Full        = inf
  typeCost (FreeType s) = TyCost $ Set.size s

instance LessArbitrary (PresenceConstraint a) where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     (PresenceConstraint a) where
  arbitrary = fasterArbitrary

instance LessArbitrary IntConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     IntConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary NumberConstraint where
  lessArbitrary = genericLessArbitrary

instance Arbitrary     NumberConstraint where
  arbitrary = fasterArbitrary

listUpToTen :: LessArbitrary a
         => CostGen      [a]
listUpToTen  = do
  len <- LessArbitrary.choose (0,10)
  replicateM len lessArbitrary

instance LessArbitrary StringConstraint where
  lessArbitrary = LessArbitrary.oneof             simple
             $$$? LessArbitrary.oneof (complex <> simple)
    where
      simple  =  pure <$> [SCDate, SCEmail, SCNever, SCAny]
      complex = [SCEnum . Set.fromList <$> listUpToTen]
             <>  simple

instance Arbitrary     StringConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary ObjectConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     ObjectConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary RecordConstraint where
  lessArbitrary = genericLessArbitraryMonoid

instance Arbitrary     RecordConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary ArrayConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     ArrayConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary RowConstraint where
  lessArbitrary = genericLessArbitraryMonoid

instance Arbitrary     RowConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary MappingConstraint where
  lessArbitrary = genericLessArbitraryMonoid
instance Arbitrary     MappingConstraint where
  arbitrary = fasterArbitrary

instance LessArbitrary UnionType where
  lessArbitrary = genericLessArbitraryMonoid

instance Arbitrary     UnionType where
  arbitrary = fasterArbitrary


shrinkSpec :: forall    a.
              (Arbitrary a
              ,Typeable  a
              ,Show      a
              ,Eq        a
              )
           => Spec
shrinkSpec = prop ("shrink on " <> nameOf @a)
           $ doesNotShrinkToItself arbitrary (shrink :: a -> [a])

allSpec :: forall         ty v.
           (Typeable       ty
            ,Arbitrary      ty
            ,Show           ty
            ,Types          ty v
            ,ArbitraryBeyond ty
            ,Arbitrary          v
            ,Show               v
            ) => Spec
allSpec = describe (nameOf @ty) $ do
  arbitraryBeyondSpec @ty
  shrinkSpec    @ty

<<typelike-spec>>
<<types-spec>>
<<typecost-laws>>

-- * Unit tests for faster checking
```

```haskell
-- | Bug in generation of SCEnum
scEnumExample = label "SCEnum" $ s == s <> s
  where
    s = SCEnum $ Set.fromList $ [""] <> [Text.pack $ show i | i <- [0..8]]

-- | Bug in treatment of missing keys
objectExample = do
    print t
    quickCheck $ label "non-empty object" $ t `check` ob2
    quickCheck $ label "empty object"     $ t `check` ob
  where
    ob  :: Object = Map.fromList []
    ob2 :: Object = Map.fromList [("a", String "b")]
    t   :: RecordConstraint = infer ob2 <> infer ob

-- | Checking for problems with set.
freetypeExample = label "freetype" $ a <> b == b <> a
  where
    a = FreeType {captured = Set.fromList [Bool False,Bool True,Number (-3000.0),Number 0.6,Number (-1.1e11),Number (-9.0
    b = FreeType {captured = Set.fromList [Bool False,Bool True,Number 5.0e-6,Null,String "?",Number 1.1e9,Number 3.0e10]

-- * Run all tests
main :: IO ()
main  = do
  {-
  sample $ arbitrary @Value
  sample $ arbitrary @NullConstraint
  sample $ arbitrary @NumberConstraint
  sample $ arbitrary @RowConstraint
  sample $ arbitrary @RecordConstraint
  sample $ arbitrary @ArrayConstraint
  sample $ arbitrary @MappingConstraint
  sample $ arbitrary @ObjectConstraint
  -}
  quickCheck scEnumExample
  objectExample
  quickCheck freetypeExample

  lawsCheckMany
    [typesSpec (Proxy :: Proxy (FreeType Value) )
               (Proxy :: Proxy Value      ) True
    ,typesSpec (Proxy :: Proxy NumberConstraint )
               (Proxy :: Proxy Scientific) True
    ,typesSpec (Proxy :: Proxy StringConstraint )
               (Proxy :: Proxy Text.Text ) True
    ,typesSpec (Proxy :: Proxy BoolConstraint   )
               (Proxy :: Proxy Bool       ) True
    ,typesSpec (Proxy :: Proxy NullConstraint   )
               (Proxy :: Proxy ()         ) True
    ,typesSpec (Proxy :: Proxy RowConstraint    )
               (Proxy :: Proxy Array      ) True
    ,typesSpec (Proxy :: Proxy ArrayConstraint  )
               (Proxy :: Proxy Array      ) True
    ,typesSpec (Proxy :: Proxy MappingConstraint)
               (Proxy :: Proxy Object     ) True
    ,typesSpec (Proxy :: Proxy RecordConstraint )
               (Proxy :: Proxy Object     ) True
    ,typesSpec (Proxy :: Proxy ObjectConstraint )
               (Proxy :: Proxy Object     ) True
    ,typesSpec (Proxy :: Proxy UnionType         )
               (Proxy :: Proxy Value      ) True
    ,typesSpec (Proxy :: Proxy (Counted NumberConstraint))
               (Proxy :: Proxy Scientific      ) False
    ]
  representationSpec

typesSpec :: (Typeable  ty
             ,Typeable      term
             ,Monoid     ty
             ,ArbitraryBeyond ty
             ,Arbitrary ty
             ,Arbitrary     term
             ,Show      ty
             ,Show          term
             ,Eq         ty
             ,Eq             term
             ,Typelike   ty
             ,Types      ty term
             ,TypeCost   ty
             )
          =>  Proxy      ty
          ->  Proxy         term
          ->  Bool -- idempotent?
          -> (String, [Laws])
typesSpec (tyProxy   :: Proxy ty)
          (termProxy :: Proxy term) isIdem =
  (nameOf @ty <> " types " <> nameOf @term, [
     arbitraryLaws          tyProxy
   , eqLaws                 tyProxy
   , monoidLaws             tyProxy
   , commutativeMonoidLaws tyProxy
   , typeCostLaws           tyProxy
   , typelikeLaws           tyProxy
   , arbitraryLaws                  termProxy
   , eqLaws                         termProxy
   , typesLaws              tyProxy termProxy
   ]<>idem)
  where
   idem | isIdem     = [idempotentSemigroupLaws tyProxy]
        | otherwise = []

typesLaws :: (           ty `Types` term
             ,Arbitrary ty
             ,ArbitraryBeyond ty
             ,Arbitrary             term
             ,Show       ty
             ,Show                  term
             )
          => Proxy ty
          -> Proxy term
          -> Laws
typesLaws (_ :: Proxy ty) (_ :: Proxy term) =
  Laws "Types" [("mempty contains no terms"
              ,property $
                mempty_contains_no_terms        @ty @term)
             ,("beyond contains all terms"
              ,property $
                beyond_contains_all_terms       @ty @term)
             ,("fusion keeps terms"
              ,property $
                fusion_keeps_terms              @ty @term)
             ,("inferred type contains its term"
              ,property $
                inferred_type_contains_its_term @ty @term)
             ]

<<representation-examples>>

representationTest :: String -> [Value] -> HType -> IO Bool
representationTest name values repr = do
    if foundRepr == repr
      then do
        putStrLn $ "*** Representation test " <> name <> " succeeded."
        return True
      else do
        putStrLn $ "*** Representation test " <> name <> " failed: "
        putStrLn $ "Values         : " <> show values
        putStrLn $ "Inferred type : " <> show inferredType
        putStrLn $ "Representation: " <> show foundRepr
        putStrLn $ "Expected       : " <> show repr
        return False
  where
    foundRepr :: HType
    foundRepr = toHType inferredType
    inferredType :: UnionType
    inferredType = foldMap infer values

readJSON :: HasCallStack
         => BS.ByteString -> Value
readJSON = fromMaybe ("Error reading JSON file")
         . decodeStrict
         . BS.unlines
         . filter notComment
         .
         BS.lines
  where
    notComment (BS.isPrefixOf "//" -> True) = False
    notComment _                            = True

representationSpec :: IO ()
representationSpec  = do
  b <- sequence
    [representationTest "1a" example1a_values example1a_repr
    ,representationTest "1b" example1b_values example1b_repr
    ,representationTest "1c" example1c_values example1c_repr
    ,representationTest "2"  example2_values  example2_repr
    ,representationTest "3"  example3_values  example3_repr
    ,representationTest "4"  example4_values  example4_repr
    ,representationTest "5"  example5_values  example5_repr
    ,representationTest "6"  example6_values  example6_repr]
  when (not $ and b) $
    exitFailure
```

9 Appendix: package dependencies

```
name: union-types
version: '0.1.0.0'
category: Web
author: Anonymous
maintainer: example@example.com
license: BSD-3
extra-source-files:
- CHANGELOG.md
- README.md
dependencies:
- base
- aeson
- containers
- text
- hspec
- QuickCheck
- unordered-containers
- scientific
- hspec
- QuickCheck
- validity
- vector
- unordered-containers
- scientific
- genvalidity
- genvalidity-hspec
- genvalidity-property
- time
- email-validate
- generic-arbitrary
- mtl
- hashable
library:
  source-dirs: src
  exposed-modules:
  - Unions
tests:
  spec:
    main: Spec.hs
    source-dirs:
      - test/lib
      - test/spec
    dependencies:
      - union-types
      - mtl
      - random
      - transformers
      - hashable
      - quickcheck-classes
      - file-embed
      - bytestring
  less-arbitrary:
    main: LessArbitrary.hs
    source-dirs:
      - test/lib
      - test/less
    dependencies:
      - union-types
      - mtl
      - random
      - transformers
      - hashable
      - quickcheck-classes
      - quickcheck-instances
```

10 Appendix: representation of generated Haskell types

We will not delve here into identifier conversion between JSON and Haskell, so it suffices that we have an abstract datatypes for Haskell type and constructor identifiers:

```
newtype HConsId  = HConsId String
  deriving (Eq,Ord,Show,Generic,IsString)
newtype HFieldId = HFieldId String
  deriving (Eq,Ord,Show,Generic,IsString)
newtype HTypeId  = HTypeId String
  deriving (Eq,Ord,Show,Generic,IsString)
```

For each single type we will either describe its exact representation or reference to the other definition by name:

```
data HType =
    HRef HTypeId
  | HApp HTypeId [HType]
  | HADT [HCons]
  deriving (Eq, Ord, Show, Generic)
```

For syntactic convenience, we will allow string literals to denote type references:

```
instance IsString HType where
  fromString = HRef . fromString
```

When we define a single constructor, we allow field and constructor names to be empty strings (""), assuming that the relevant identifiers will be put there by post-processing that will pick names using types of fields and their containers [18].

```
data HCons = HCons {
              name ::  HConsId
            , args :: [(HFieldId, HType)]
            }
  deriving (Eq, Ord, Show, Generic)
```

At some stage we want to split representation into individually named declarations, and then we use environment of defined types, with an explicitly named toplevel type:

```
data HTypeEnv = HTypeEnv {
    toplevel :: HTypeId
  , env      :: HashMap HTypeId HType
  }
```

When checking for validity of types and type environments, we might need a list of predefined identifiers that are imported:

```
predefinedHTypes :: [HType]
predefinedHTypes = [
    "Data.Aeson.Value"
  , "()"
  , "Double"
  , "String"
  , "Int"
  , "Date" -- actually: "Data.Time.CalendarDay"
  , "Email" -- actually: "Data.Email"
  ]
```

Consider that we also have an htop value that represents any possible JSON value. It is polimorphic for ease of use:

```
htop :: IsString s => s
htop = "Data.Aeson.Value"
```

10.1 Code for selecting representation

Below is the code to select Haskell type representation. To convert union type discipline to strict Haskell type representations, we need to join the options to get the actual representation:

```
toHType :: ToHType ty => ty -> HType
toHType  = joinAlts . toHTypes

joinAlts     :: [HType] -> HType
joinAlts []    =  htop -- promotion of empty type
joinAlts alts =  foldr1 joinPair alts
  where
    joinPair a b = HApp ":|:" [a, b]
```

Considering the assembly of UnionType, we join all the options, and convert nullable types to Maybe types

```
instance ToHType UnionType where
  toHTypes UnionType {..} =
      prependNullable unionNull opts
    where
      opts = concat [toHTypes unionBool
                    ,toHTypes unionStr
                    ,toHTypes unionNum
                    ,toHTypes unionArr
                    ,toHTypes unionObj]

prependNullable :: PresenceConstraint a -> [HType] -> [HType]
prependNullable Present tys = [HApp "Maybe" [joinAlts tys]]
prependNullable Absent  tys =                tys
```

The type class returns a list of mutually exclusive type representations:

```
class Typelike ty
  => ToHType  ty where
  toHTypes :: ty -> [HType]
```

Conversion of flat types is quite straightforward:

```
instance ToHType BoolConstraint where
  toHTypes Absent  = []
  toHTypes Present = ["Bool"]
instance ToHType NumberConstraint where
  toHTypes NCNever = []
  toHTypes NCFloat = ["Double"]
  toHTypes NCInt   = ["Int"]
instance ToHType StringConstraint where
  toHTypes  SCAny      = ["String"]
  toHTypes  SCEmail    = ["Email"]
  toHTypes  SCDate     = ["Date"]
  toHTypes (SCEnum es) = [HADT $
                           mkCons <$> Set.toList es
                          ]
    where
      mkCons = (`HCons` [])
             .  HConsId
             .  Text.unpack
  toHTypes  SCNever    = []
```

For array and object types we pick the representation which presents the lowest cost of optionality:

```
instance ToHType ObjectConstraint where
  toHTypes ObjectNever            = []
  toHTypes ObjectConstraint {..} =
    if typeCost recordCase <= typeCost mappingCase
      then toHTypes recordCase
      else toHTypes mappingCase

instance ToHType RecordConstraint where
  toHTypes  RCBottom = []
  toHTypes  RCTop    = [htop] -- should never happen
  toHTypes (RecordConstraint fields) =
      [HADT
          [HCons "" $ fmap convert $ Map.toList fields]
      ]
    where
      convert (k,v) = (HFieldId $ Text.unpack k
                      ,toHType v)

instance ToHType MappingConstraint where
  toHTypes MappingNever = []
  toHTypes MappingConstraint {..} =
    [HApp "Map" [toHType keyConstraint
                ,toHType valueConstraint
                ]]

instance ToHType RowConstraint where
  toHTypes  RowNever  = []
  toHTypes  RowTop    = [htop]
  toHTypes (Row cols) =
    [HADT
        [HCons "" $ fmap (\ut -> ("", toHType ut)) cols]
    ]

instance ToHType ArrayConstraint where
  toHTypes ArrayNever            = []
  toHTypes ArrayConstraint {..} =
    if   typeCost arrayCase <= typeCost rowCase
      -- || count <= 3
      then [toHType arrayCase]
      else [toHType rowCase  ]
```

Appendix: Missing pieces of code

In order to represent FreeType for the Value, we need to add Ord instance for it:

```
deriving instance Ord Value
```

For validation of dates and emails, we import functions from Hackage:

```
isValidDate :: Text -> Bool
isValidDate = isJust
            . parseDate
            . Text.unpack
  where
    parseDate :: String -> Maybe Day
    parseDate  = parseTimeM True
                   defaultTimeLocale $
                   iso8601DateFormat Nothing

isValidEmail :: Text -> Bool
isValidEmail = Text.Email.Validate.isValid
           . Text.encodeUtf8

instance (Hashable        k
         ,Hashable        v)
      => Hashable (HashMap k v) where
  hashWithSalt s = hashWithSalt s
                 . Foldable.toList

instance Hashable         v
      => Hashable (V.Vector v) where
  hashWithSalt s = hashWithSalt s
                 . Foldable.toList
```

```
-- instance Hashable Scientific where
-- instance Hashable Value where
```

Then we put all the missing code in the module:

```
{-# language AllowAmbiguousTypes         #-}
{-# language DeriveGeneric               #-}
{-# language DuplicateRecordFields       #-}
{-# language FlexibleInstances           #-}
{-# language GeneralizedNewtypeDeriving #-}
{-# language MultiParamTypeClasses       #-}
{-# language NamedFieldPuns              #-}
{-# language PartialTypeSignatures       #-}
{-# language ScopedTypeVariables         #-}
{-# language StandaloneDeriving          #-}
{-# language TypeOperators               #-}
{-# language RoleAnnotations             #-}
{-# language ViewPatterns                #-}
{-# language RecordWildCards             #-}
{-# language OverloadedStrings           #-}
{-# options_ghc -Wno-orphans             #-}
module Missing where

import           Control.Arrow(second)
import           Data.Aeson
import           Data.Maybe(isJust,catMaybes)
import qualified Data.Foldable as Foldable
import           Data.Function(on)
import           Data.Text(Text)
import qualified Data.Text as Text
import qualified Data.Text.Encoding  as Text
import qualified Text.Email.Validate(isValid)
import qualified Data.Set  as Set
import           Data.Set(Set)
import           Data.Scientific
import           Data.String
import qualified Data.Vector         as V
import qualified Data.HashMap.Strict as Map
import           Data.HashMap.Strict(HashMap)
import           GHC.Generics        (Generic)
import           Data.Hashable
import           Data.Typeable
import           Data.Time.Format    (iso8601DateFormat,parseTimeM,defaultTimeLocale)
import           Data.Time.Calendar (Day)

<<missing>>
```

Container Unification for Uniqueness Types —DRAFT—

Folkert de Vries
Radboud University
Nijmegen, NL
Folkert.deVries@student.ru.nl

Sjaak Smetsers
Radboud University
Nijmegen, NL
Sjaak.Smetsers@ru.nl

Sven-Bodo Scholz
Radboud University
Nijmegen, NL
SvenBodo.Scholz@ru.nl

Abstract

This paper proposes a new approach towards representing uniqueness types as logic formulae. It introduces a notion of containerised uniqueness attributes which resolves the two key challenges of preexisting work on uniqueness types as logic formulae: The unification of such formulae becomes computationally tractable and the inferred types are more conducive for an interpretation by programmers.

1 Introduction

A key characteristic of pure languages is referential transparency. It guarantees that variables are placeholders whose values are fixed throughout the program execution. This property allows variables to be replaced by their definition at any time without effecting the overal result, enabling reasoning about programs in terms of equations and thus opening the door for a wide range of formal proofs.

Many implementations of pure functional languages make use of sharing to limit memory use: a value is stored just once in memory, and program variables are references to this memory. When there is more than one variable storing a reference to the same object, this object is *shared*. A destructive update to such values is *observable*: it can change the program output. A shared value will be used later in the evaluation of the program, at which point the original definition and the current value may be different because of an earlier destructive update: referential transparency is violated.

For this reason, variables in such languages cannot be used to denote memory that can be updated at will, in sharp contrast to procedural languages. Even small changes in large data structures, at least conceptually, require the creation of a completely new data structures. This copying causes huge overhead in both runtime, and space demand, and presents challenges for the efficient implementation of pure languages.

But crucially, a destructive update only violates referential transparency because the value is shared, and therefore used after it has been modified. If value is not shared, a destructive update to it is not *observable*: the original value is not used in the rest of the program, and the program output is unchanged. The challenge is then to determine whether a variable is shared, and thus whether it can be updated destructively.

Reference counting can determine at runtime whether a value is shared. Runtime values are extended with a counter, the *reference count*, that tracks how many live pointers exist to the value. A common usage of reference counting is garbage collection. When the reference count of a value is decremented to 0, there is no way to access the value and its memory can be reclaimed. But when the reference count is exactly 1, the value is not shared and can be safely destructively updated. Reference counting itself however has a runtime cost: values extended with space for the reference count may no longer fit in a cache line, and the manipulation of the reference count has overhead, especially in a multi-threaded scenario.

Uniqueness types are a mechanism to statically determine whether a value is non-shared, and thus whether a destructive update can be performed. While less precise than a runtime approach, the static nature enables reasoning about performance and memory usage. Furthermore, the type system can enforce that certain values are never shared, providing strong correctness guarantees.

Uniqueness types were initially developed for and implemented in the Clean programming language [15]. The semantics of Clean are based on a term rewrite system, and the uniqueness types constitute a non-trivial extension of the type system of Clean. They add a notion of uniqueness variables and constraints on them; subtyping is introduced for the use of unique objects in non-unique contexts. All these extensions are invasive in almost the entire type inference process.

In Uniqueness Typing Simplified (UTS) [7], Edsko de Vries et al. present an approach for inferring uniqueness types that is defined on the lambda calculus in a way that is orthogonal to the base type system. While this approach still contains the notion of uniqueness variables, subtyping and constraints

IFL '20, September 2–4, 2020, Online
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

on uniqueness variables in this approach are expressed as formulae in first-order logic.

While this approach very elegantly simplifies the expression of uniqueness types and their addition to other type systems, it does come with its own challenges. It requires the unification of logic formulae which can quickly turn into a performance bottleneck and which often leads to unnecessarily complex logic expressions. An automation of their simplification adds to the challenge and it seems to be impossible to guarantee a concise, easily understandable result.

In this work, we try to further improve on the UTS approach. We limit the expression of constraints between uniqueness variables to solely disjunctions. We establish upper and lower bounds on the uniqueness attribute of containers (e.g. tuples), and use a unification mechanism inspired by that of polymorphic records. The result is a unification of uniqueness attributes that is fast and infers succinct type signatures.

Chapter 2 first describes the core concepts involved in uniqueness types and their inference. Later sections look in detail at the inference of uniqueness types in the Clean programming language, and the approach of Uniqueness Typing Simplified [7]. Chapter 3 describes why inference with UTS is slow and finds large types. Chapter 4 describes container annotations, our proposed solution. Chapter 5 describes related work, and Chapter 6 concludes the thesis.

2 Background

2.1 Basic concepts

This section introduces the basic concepts behind uniqueness types, and is not yet specific to an implementation. The following sections look at the concrete systems of Clean and UTS in more detail.

2.1.1 Updating values. In procedural languages, variables can – and often do – denote memory that can be updated at will. Values are commonly destructively updated: the update modifies the original in-place in memory. In contrast, non-destructive updates never change the original. Conceptually, they first copy the original, and then change this new value in-place. Copying of data has a steep cost in terms of both runtime and memory use, and should therefore be avoided as much as possible.

An expensive copy can be partially avoided with tree-based data structures. When updating a tree, large subtrees will likely be untouched. *Structural sharing* is a strategy where only the the modified subtree and a path from it to the root need to be copied, all other parts of the tree can reference the original. The drawbacks of tree data structures are two-fold: access times are no longer constant and the number of memory allocations required for a single data structure typically increase linearly with the size of that overall data.

A flat array (a contiguous region of memory) does have constant access time and linear memory usage, but copying an array is expensive.
That presents a challenge for the usage of arrays in pure functional languages, because most of these languages cannot recognize in general when a destructive update is safe. There are pure functional languages for array-based programming, but they all must minimize copying to be performant. They use techniques like reference counting and uniqueness types to determine when in-place mutation is safe [12] [16]. In languages without such mechanisms, the usage of tree-based data structures instead of arrays is common because trees can benefit from structural sharing.

Destructive updates are however not just important for the efficient usage of arrays; the tree data structures and ADTs that are so ubiquitous in functional languages also benefit. For instance [17] uses reference counting to make linked-list and tree transformations use constant-space when the data structure is not shared.

Thus, the ability to use destructive updates in a functional language is desirable: it enables the efficient usage of arrays, an attractive data structure in many cases, and speeds up operations on other common data structures.

ADD: tradeoff of RC: more accurate but has runtime overhead h3ere we'll focus on uniqueness types (do things statically)

2.1.2 Statically marking sharing variables. To determine at compile time whether a destructive update to a value is safe, it must be known whether this value is potentially shared. We assume that constants (`1`, `[]`, `"foo"`, etc.) are never shared (e.g. $f\,[\,]\,[\,]$ would create two new empty lists). Furthermore we assume that the primary source of sharing is variables. A variable that is guaranteed not to be shared is an *exclusive* variable.

An accurate static marking of variables as shared or exclusive is undecidable, but the solution can be approximated. The general idea is to count how often a variable occurs in its scope. If it occurs once, the variable is exclusive. Otherwise it must be conservatively assumed the variable is shared.

For example, the x variable occurs only once in the body of the `identity` function, and therefore is marked as exclusive ($\odot$):

$$\texttt{identity x = x}^{\odot}$$

In constrast the x variable occurs twice in the body of *duplicate*, and thus is marked as shared ($\otimes$):

$$\texttt{duplicate x = (x}^{\otimes}\texttt{, x}^{\otimes}\texttt{)}$$

For implementation reasons these markings are conventionally written in the expression, rather than at the binding site.

The marking of variables solely by counting the number of occurences is conservative, because not every second occurence actually causes the variable to become shared. For this thesis we will assume it is possible to extract both values from a tuple without sharing the tuple: the two values could conceptually be extracted at the same time. Other refinements to improve the accuracy of markings are orthogonal to this thesis. Concretely, this annotation is correct:

$$\texttt{swap} = \lambda\, r.\ (\texttt{snd}\ r^{\odot}, \texttt{fst}\ r^{\odot})$$

In a more powerful language, this function can be implemented using pattern matching, in which case the `r` variable will only occur once.

2.1.3 Propagation of sharing. A variable marked as exclusive is not shared within it scope. However, it may be an alias for a variable that is shared, or its value may be shared. To guarantee the safety of destructive updates, local sharing information must be propagated to be able to give global guarantees about non-sharedness.

We call a variable *unique* if we can statically guarantee that it is globally non-shared. Otherwise it is *non-unique*. Variables marked as shared are certainly non-unique, but determining the uniqueness of exclusive-marked variables requires essentially a fixed-point calculation.

Functions can propagate the uniqueness between arguments and the output. For instance if the identity function is applied to a unique value, the output is still unique. When instead it is applied to a non-unique value, the output is still non-unique. The identity function $\lambda\, x.x$ should thus be able to handle both unique and non-unique values.

Intuitively it is possible to treat unique values as non-unique. The uniqueness property can be ignored: even though destructive updates are safe, one can choose to update non-destructively. But there is a subtle complication regarding function types. Consider the functions:

$$const = \lambda\, x.\, \lambda\, y.\ x$$
$$twice = \lambda\, f.\, \lambda\, x.\ (f(x), f(x))$$

Assume p is a unique value. The partial application $T = const\ p$ must hold onto this p value, so it can be returned when the partial application becomes fully applied. When T indeed becomes fully applied, the unique p value is returned. But p is unique, and therefore may not be shared. Therefore T may only become fully applied once! To enforce this constraint, a function that stores a unique value in its closure must itself become unique. Moreover, it is unique in a way that is unsafe to ignore: unique functions are *necesarilly unique*.

Therefore the argument to the the `duplicate` function cannot be any value of any uniqueness. Only non-necesarilly-unique values should be accepted.

$$\texttt{duplicate x = (x}^{\otimes}\texttt{, x}^{\otimes}\texttt{)}$$

Finally, containers (e.g. tuples, records, ADTs) put a demand on the relation between the uniqueness of the container and its elements. This is captured in the *container rule*:

To extract a unique value from a container,
the container must itself be unique

Concretely, given a pair (x, y), sharing the pair will also share the elements x and y, because they can be repeatedly extracted. We can now reason about the uniqueness demands and guarantees of `swap`:

$$\texttt{swap} = \lambda\, r.\ (\texttt{snd}\ r^{\odot}, \texttt{fst}\ r^{\odot})$$

If the input tuple r is unique, then the two elements can be extract uniquely, but also non-uniquely. If it is non-unique, then the elements can only by extracted non-uniquely. In other words, the uniqueness of the tuple r must be at least as unique as either of the elements. Thus it must be possible to express in a function type that certain parameters are at least as unique as others.

It turns out that a type system is well-suited for expressing the constraints between unique and non-unique values outlined above. A type checking and inference algorithm can perform the fixed-point calculation that determines for each expression whether it is globally non-shared. In the next subsections we will look at two concrete systems with uniqueness types. Based on this subsection, a point in the design space must be able to:

- distinguish between unique and non-unique values
- specify that two values are either both unique or both non-unique
- specify that a value is *at least as unique* as some other value

For practical programmer convenience, at least two other aspects are important:

- it should be possible to define functions generically for both unique and non-unique arguments when the implementation would be the same, thus preventing code duplication
- the inferred uniqeness properties need to be communicated with the programmer in a clear way

2.2 Uniqueness Types in Clean

2.2.1 Uniqueness attributes. In Clean every type comes in two variants: unique and non-unique. Standard Curry-style types, the *base types* are annotated with a uniqueness attribute:

- *unique*: denoted with a superscript bullet, $Int^{\bullet}$.
- *non-unique*: denoted with a superscript cross, $Int^{\times}$.

The attribute on function arrows is written above the arrow: $\overset{\times}{\rightarrow}, \overset{\bullet}{\rightarrow}$. A unique function $\overset{\bullet}{\rightarrow}$ is **necesarilly unique**. Values of a type annotated with $\bullet$ can be safely destructively updated, while values of a $\times$-annotated type cannot.

To express that the attribute on different (parts of) arguments is the same, *uniqueness polymorphism* is introduced. A uniqueness annotation can contain a *uniqueness variable*, for instance in the annotation `identity` $:: a^u \overset{\times}{\rightarrow} a^u$. Both the based type a and the uniqueness annotation u of the output must be the same as the input.

The uniqueness of constants is a free uniqueness variable: $1 :: \texttt{Int}^u$. A free uniqueness variable in the return type allows a caller to decide the uniqueness of a function's output, e.g. $(\lambda\, x.\, 1) :: a^u \overset{\times}{\rightarrow} \texttt{Int}^w$.

2.2.2 Subtyping. The container rule requires the ability to express that certain uniqueness attributes are more unique than others. To this end, Clean introduces two concepts: a subtyping rule on uniqueness attributes, and uniqueness constraints.

In the Clean type system, unique types are subtypes of their non-unique counterparts. This means that a unique value can be used in a non-unique context, e.g. when a function requires a non-unique argument, then it can be given a unique value. For most types, uniqueness is a property that can be ignored.

However, function types and type variables – which may be instantiated to function types – are *necesarilly unique*: their uniqueness cannot be ignored. The subtyping rule in Clean therefore exempts these types.

This exemption has consequences for the applicability of uniqueness type variables. Consider again the `duplicate` function:

$$\texttt{duplicate x = (x}^{\otimes}\texttt{, x}^{\otimes}\texttt{)}$$

This function duplicates its argument. Therefore the return type is a tuple with two non-unique elements. At first sight the subtyping rule seems to allow the signature:

$$\texttt{duplicate} :: a^u \overset{\times}{\rightarrow} (a^\times, a^\times)^w$$

That is, the input can be of any type a with any uniqueness attribute u. But the exemption of function types from the subtyping rule means this signature is incorrect. `duplicate` actually cannot be applied to function types with attribute $\bullet$. This function is therefore given the annotation:

$$\texttt{duplicate} :: a^\times \overset{\times}{\rightarrow} (a^\times, a^\times)^w$$

Now the argument can be any type a with an annotation that **is a subtype of** $\times$. That includes unique types that are not necesarilly-unique.

Equalities between uniqueness attributes are captured by parametric polymorphism over uniqueness variables. To express inequalities between uniqueness attributes, Clean uses *uniqueness constraints*. For instance:

$$\mathit{fst} :: (t^u, s^v)^w \rightarrow t^u, [w \leq u]$$

Here the syntax $[w \leq u]$ expresses that w must be a subtype of u. We can also interpret the constraint as u implies w: if u is unique, then w must be as well (and if u is not unique, w is unconstrained). Because the second element is not extracted, it's uniqueness annotation is not relevant for the signature of `fst`.

2.3 Uniqueness Typing Simplified

Uniqueness Typing Simplified [7] makes considerably different design decisions

A big accomplishment of UTS is that uniqueness types are orthogonal to other type system features. However the UTS approach has challenges of its own.

UTS combines base types and uniqueness attributes into one syntactical category, distinguishing the two with a kind system. Base types are of kind $\mathcal{T}$, and uniqueness attributes are of kind $\mathcal{U}$. A special constructor $\texttt{Attr} :: \mathcal{T} \rightarrow \mathcal{U} \rightarrow *$ combines a base type and uniqueness attribute into a type of kind $*$, the kind that is inhabited by values. The goal of this change is that a standard hindley-milner type checker with kind inference can be used to infer uniqueness types with minimal modification. The kind language is given in figure 1.

In UTS, uniqueness relations are represented as formulae in first-order logic. Uniqueness attributes, types of kind $\mathcal{U}$, are boolean expressions. The $\bullet$ type now stands for boolean True, $\times$ for boolean False, but attributes can also contain variables and the boolean connectives $\neg, \vee, \wedge$. We say that a value is unique when the uniqueness attribute of its type **evaluates** to $\bullet$. E.g. both of these are the type of unique integers: $\texttt{Int}^\bullet$ and $\texttt{Int}^{u \vee \bullet}$.

Like in Clean, uniqueness polymorphism can be used to express that the uniqueness of two types is the same. However, where Clean uses uniqueness constraints, UTS encodes implications between uniqueness attributes as disjunctions:

$$\mathit{fst} :: (t^u, s^v)^{u \vee w} \rightarrow t^u$$

To extract the first value uniquely (i.e. when $u = \bullet$), the tuple must also be unique. Indeed, the boolean expression $u \vee w$ evaluates to $\bullet$ if u is $\bullet$, and when $u = \times$, the free w variable still allows the annotation on the tuple to become $\bullet$.

An advantage of boolean attributes is that all information to determine the value of the annotation is in the annotation, rather than in a constraint in the environment. This is convenient when adding advanced type system features like higher-rank polymorphism and impredicativity [5] [14].

In UTS, all unique types are necesarilly unique: their uniqueness cannot be ignored. Therefore a value of type t^u cannot be implicatly converted into $t^\times$, and `duplicate` function cannot be given the type

$$\texttt{duplicate} :: t^u \overset{\times}{\rightarrow} (t^\times, t^\times)$$

Therefore we must assign it a type that is visually the same as Clean's, but semantically different:

$$\texttt{duplicate} :: t^\times \overset{\times}{\rightarrow} (t^\times, t^\times)$$

Because in UTS all unique types are necesarilly unique and there is no subtyping rule, this function can really only be applied to non-unique arguments.

At first sight, this change seems to severely limit the opportunities where a value is inferred as unique, and can thus

Kind language

κ ::=		kind
	$\mathcal{T}$	base type
	$\mathcal{U}$	uniqueness attribute
	$*$	base type together with uniqueness attribute
	$\kappa_1 \to \kappa_2$	type constructors

Type constants

`Int, Bool`	$:: \mathcal{T}$	base type
$\to$	$:: * \to * \to \mathcal{T}$	function space
$\bullet, \times$	$:: \mathcal{U}$	unique, non-unique
$\vee, \wedge$	$:: \mathcal{U} \to \mathcal{U} \to \mathcal{U}$	logical or, and
$\neg$	$:: \mathcal{U} \to \mathcal{U}$	logical negation
`Attr`	$:: \mathcal{T} \to \mathcal{U} \to *$	combine base type and attribute

Syntactic conventions

$$t^u \equiv \texttt{Attr}\ t\ u$$

$$a \xrightarrow{u} b \equiv \texttt{Attr}\ (a \to b)\ u$$

Figure 1. kind language

be destructively updated. However in practice, this problem can be overcome with careful API design.

A generic function $t^u \xrightarrow{\times} t^v$ cannot be implemented, and would be unsafe to expose as a primitive. But for specific types, e.g. arrays, a coercion primitive is perfectly safe:

$$\texttt{coerce} :: (\texttt{Array}\ \ t^u)^v \xrightarrow{\times} (\texttt{Array}\ \ t^u)^w$$

This function changes the uniqueness of the array from v to w, and may be instantiated at types $u = \bullet$ and $w = \times$. But explicit coercions are rarely needed if primitives return uniqueness-polymorphic values when possible. For instance:

$$\texttt{set} :: \texttt{Int}^z \xrightarrow{\times} t^u \xrightarrow{z} (\texttt{Array}\ \ t^u)^v \xrightarrow{z \vee u} (\texttt{Array}\ \ t^u)^w$$

If the input array is non-unique, the output array is newly allocated, otherwise the array is mutated in-place. In either case, the insertion always produces a unique array. But the output is not $(\texttt{Array}\ \ t^u)^\bullet$ to enable the output to actually be non-unique if the surrounding context demands it.

After type inference and checking, remaining polymorphic uniqueness attributes can be interpreted as $\bullet$: a polymorphic attribute at this stage means that either $\times$ or $\bullet$ can be handled, but we can pick $\bullet$ to potentially benefit from destructive updates.

2.3.1 Language.

To talk about the typing rules and uniqueness type inference, we must first define a language for them to operate on. Our language (2) is the lambda calculus extended with tuples, and usage markings exclusive ($\odot$) and shared ($\otimes$) on variables.

Expressions e	::=	$x^\odot$	variable (exlusive)
	|	$x^\otimes$	variable (shared)
	|	$\lambda\, x.\ e$	abstraction
	|	$e\ e$	application
	|	(e, e)	tuple construction
	|	**fst** e	tuple projection 1
	|	**snd** e	tuple projection 2

Figure 2. lambda calculus extended with tuples

Variable occurences are annotated as either exlusive or shared by a *usage analysis.* A variable x is marked exclusive if:

- it occurs freely exactly once in its scope, or
- it occurs freely exactly twice in its scope: once as an argument to the primitive function **fst**, and once as an argument to the primitive function **snd**

Otherwise, a variable is annotated as shared. Exclusive usage is denoted with a superscript $\odot$, shared usage is denoted with a superscript $\otimes$. Examples of annotations based on these rules are:

$$(\lambda\, x.x^\odot)$$
$$(\lambda\, x.(x^\otimes, x^\otimes))$$
$$(\lambda\, r.(\texttt{fst}\ r^\odot, \texttt{snd}\ r^\odot))$$
$$(\lambda\, r.(\texttt{fst}\ r^\otimes, \texttt{fst}\ r^\otimes))$$
$$(\lambda\, r.(\texttt{fst}\ ((\lambda\, x.x^\odot)r^\otimes), \texttt{snd}\ r^\otimes))$$

Note again that the markings are in the expressions, not at the binding site of a variable, even though in UTS the marking annotation has to be the same at all occurences of a variable. In the UTS system the annotation is used in the typing rule **var**. Having the annotation on the expression rather than in the environment makes the typing rules simpler.

2.3.2 Type Inference.

The typing relation (figure 3) is taken from [7], and extended with the rules for tuples. The typing relation consists of judgements of the form

$$\Gamma \vdash e : \tau|_{fv}$$

Which is read as "in environment Γ, expression e has type τ, where the uniqueness attributes on free variables in e are fv". The uniqueness attributes of free variables are used to determine whether a function must be unique (as free variables are captured in the closure, and closures must adhere to the container rule).

The **var**$^\otimes$ forces any shared variable to be of a non-unique type. **var**$^\odot$ assigns a free uniqueness attribute to exclusive variables.

When a function captures unique variables in its closure, the function must itself be unique. The variables captured by a closure are the free variables in the function body. Hence a map of free variables to their uniqueness attribute is maintained. Rule **abs** uses this map to constrain the type of the function arrow. The **app** rule enforces that the argument to a function has the same type as the function's parameter.

Next **pair** types the construction of tuples. Note that the uniqueness attribute on the result is the free variable w. The container rule is not enforced when creating containers, only when extracting values from them. Therefore w can be free in this rule. Finally **fst** and **snd** enforce the container rule: a unique element can only be extracted if the tuple is itself unique.

$$\textbf{var}^{\odot}\ \frac{}{\Gamma, x : \tau^{v} \vdash x^{\odot} : \tau^{v}|_{x:v}} \qquad \textbf{var}^{\otimes}\ \frac{}{\Gamma, x : \tau^{\times} \vdash x^{\otimes} : \tau^{\times}|_{x:\times}}$$

$$\textbf{abs}\ \frac{\Gamma, x : \tau \vdash e : \tau'|_{fv} \qquad fv' = fv \unrhd x}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{\bigvee fv'} \tau'|_{fv'}}$$

$$\textbf{app}\ \frac{\Gamma \vdash e : \tau \xrightarrow{v} \tau'|_{fv_1} \qquad \Gamma \vdash e' : \tau|_{fv_2}}{\Gamma \vdash e\ e' : \tau'|_{fv_1 \cup fv_2}}$$

$$\textbf{pair}\ \frac{\Gamma \vdash x : t^{u}|_{fv_1} \qquad \Gamma \vdash y : s^{v}|_{fv_2}}{\Gamma \vdash (x, y) : (t^{u}, s^{v})^{w}|_{fv_1 \cup fv_2}}$$

$$\textbf{fst}\ \frac{\Gamma \vdash r : (t^{u}, s^{v})^{u \vee w}|_{fv}}{\Gamma \vdash \textbf{fst}\ r : t^{u}|_{fv}}$$

$$\textbf{snd}\ \frac{\Gamma \vdash r : (t^{u}, s^{v})^{v \vee w}|_{fv}}{\Gamma \vdash \textbf{snd}\ r : s^{v}|_{fv}}$$

Figure 3. typing rules adapted from *Uniqueness typing simplified*[7] extended with products. In rule **abs**, $\unrhd$ is the domain subtraction operator. It removes x from the set of free variables because it is bound in the lambda body.

3 The problem with boolean attributes

The UTS approach of making uniqueness type inference orthogonal to the rest of the type system is impressive. However, we highlight two problems with UTS type inference already noted in [7]:

- unification of boolean attributes finds large unifiers, occurs often, and is computationally expensive
- inferred types are hard to interpret

These are serious problems in practice, because UTS relies on disjunctions for uniqueness propagation. We will first look at an example where a needlessly complex type is inferred, then discuss boolean unification and highlight why disjunctions in particular cause unifiers to be large.

3.1 An example

In [7], the function `swap` is given as an example where the inferred type is hard to interpret.

$$swap = \lambda\, t.(\textbf{snd}\ t^{\odot}, \textbf{fst}\ t^{\odot})$$

The desired inferred signature for this function in the UTS system is:

$$swap :: (s^{v}, t^{u})^{v \vee u \vee w} \to (t^{u}, s^{v})^{w'}$$

We ignore the attribute on the arrow. u and v can only be unique if the input tuple is. Note that the uniqueness on the output tuple is the unbound variable w', because tuple creation does not enforce the container rule. Unfortunately, the *inferred* type based on UTS is:

$$\begin{aligned} swap ::\ & (s^{(\neg\, v \wedge u) \vee (\neg\, v \wedge w) \vee (u1 \wedge u) \vee (u1 \wedge w)}, t^{u})^{u \vee w} \\ & \to (t^{u}, s^{(\neg\, v \wedge u) \vee (\neg\, v \wedge w) \vee (u1 \wedge u) \vee (u1 \wedge w)})^{v1} \end{aligned}$$

These two signatures are logically equivalent, but it's not at all trivial to see that they are. Additionally, [7] reports that type-checking `swap` with the succinct signature takes "a long time".

3.2 Unification

Type equivalence is important in type inference. For instance, in the application $f\ x$, the type of the first argument of f must be equivalent to the type of x for the application to be well-typed.

Equivalence of types is defined as *equality up to unification.* Unification of two terms T, T' aims to find a substitution or *unifier* S such that $B \vdash ST \doteq ST'$. Here the set B is the set of identities. To infer the most general type for functions, it is important to use not just any unifier, but a *most general unifier* (mgu). An mgu subsumes all other unifiers.

For the unification of base types, syntactical unification is used. With syntactical unification, the set of identities B is empty. Therefore for T and T' to unify, ST has to be syntactically the same as ST'. For example, the unification problem $\texttt{Int} \doteq a$ has the unifier $[a \mapsto \texttt{Int}]$. Syntactic unification is compositional: for instance, unification of two function types implies that the argument and result types individually must unify.

$$\frac{a \doteq c \qquad b \doteq d}{a \to b \doteq c \to d}$$

In contrast, UTS uses boolean unification to unify uniqueness attributes. For boolean unification the set of identities B contains Huntington's postulates, intuitively meaning that for S to be a unifier, the truth tables of ST and ST' must be the same. Boolean unification is decidedly non-compositional. For instance the unification problem $a \vee b \doteq \bullet$ has several unifiers: either a, or b, or both must be $\bullet$ for $a \vee b$ to unify with $\bullet$. The most general unifier must cover all of these options, and is therefore $[a \mapsto \neg b \vee a]$, and absolutely not $[a \mapsto \bullet, b \mapsto \bullet]$.

3.3 Unifying two disjunctions

The rest of our argument hinges on the observation that unification of two disjunctions produces a large unifier. Consider the unification problem $t^{u1 \vee u2} \doteq t^{v1 \vee v2}$. The intuition is that the annotation of this type is $\bullet$ if and only if at least one of u_1, u_2, v_1, v_2 is $\bullet$.

Boolean unificaton finds the unifier S:

$$u1 \mapsto (\neg u2 \wedge v1) \vee (\neg u2 \wedge v2) \vee (u1 \wedge v1) \vee (u1 \wedge v2)$$
$$u2 \mapsto (u2 \wedge v1) \vee (u2 \wedge v2)$$

Note that the variable names u_1, u_2 are present in their own subsitition, but they represent fresh variables. Furthermore, their assignment is irrelevant for the value of $S(u_1 \vee u_2)$, which is totally determined by the assignments of v_1, v_2. We define U_1, U_2 as shorthands, but with u_1, u_2 renamed to w_1, w_2 for clarity. Logically, $S(u_1 \vee u_2) = U_1 \vee U_2$.

$$U_1 = (\neg w2 \wedge v1) \vee (\neg w2 \wedge v2) \vee (w1 \wedge v1) \vee (w1 \wedge v2)$$
$$U_2 = (w2 \wedge v1) \vee (w2 \wedge v2)$$

To convince ourselves that this substitution is in fact a unifier, we must verify that:

1. the truth table of $U_1 \vee U_2$ is the same as the truth table of $v_1 \vee v_2$.
2. both $u_1 \doteq \bullet$ and $u_2 \doteq \bullet$ imply $v_1 \vee v_2 \doteq \bullet$.

The first point can easily be checked by hand. The second is harder, so we'll spell out the details for the case of u_1. If $u_1 \doteq \bullet$ that implies that $U_1 \doteq \bullet$. So we must solve the unification problem:

$$(\neg w2 \wedge v1) \vee (\neg w2 \wedge v2) \vee (w1 \wedge v1) \vee (w1 \wedge v2) \doteq \bullet$$

This gives the unifier $[w2 \mapsto (w2 \wedge w1), v1 \mapsto (\neg v2 \vee v1)]$. Now when we apply this unifier to $v_1 \vee v_2$, we get $\neg v_2 \vee v_1 \vee v_2$. By the law of the excluded middle, this expression always evaluates to $\bullet$.

To ensure the individual substitutions (e.g. U_1 and U_2 in the above example) are as small as possible, UTS proposes to use boolean simplification. But boolean simplification has exponential runtime complexity [18]. Because the substitutions are usually still larger than one variable (e.g. substitution of u_1 with U_1 grows the expression size), the types grow over the course of unification, to sizes where boolean unification's time complexity becomes a problem.

3.4 Nails in the coffin

In this section we will see that, besides being computationally expensive, unification of disjunctions occurs often. Additionally, inferred types are hard for the programmer to interpret.

Consider for instance the function:

$$choose : a \rightarrow a \rightarrow a$$

Inference of an application *choose x y* must unify the types assigned to x and y. We pick $x :: t^{u1 \vee u2}$ and $y :: t^{v1 \vee v2}$. What is the inferred return type of the application?

We can reason from first principles: the annotation on x is unique when either $u1$ or $u2$ is. Likewise the annotation on y is unique when either $v1$ or $v2$ is. Therefore the return value must be unique if at least one of $u1, u2, v1, v2$ is unique. We expect the inferred return type to be $t^{u1 \vee u2 \vee v1 \vee v2}$.

Unfortunately, the *at least one is unique* constraint is difficult to express in boolean logic. As we've seen, the boolean unification $u_1 \vee u_2 \doteq v_1 \vee v_2$ gives the rather large and obtuse unifier:

$$u1 \mapsto (\neg u2 \wedge v1) \vee (\neg u2 \wedge v2) \vee (u1 \wedge v1) \vee (u1 \wedge v2)$$
$$u2 \mapsto (u2 \wedge v1) \vee (u2 \wedge v2)$$

Thus, the type $t^{u1 \vee u2}$ will now be rendered as

$$t^{(\neg u2 \wedge v1) \vee (\neg u2 \wedge v2) \vee (u1 \wedge v1) \vee (u1 \wedge v2) \vee (u2 \wedge v1) \vee (u2 \wedge v2)}$$

This would be fine if this annotation is subsequently simplified, but [7] notes that the inferred types often cannot be sufficiently simplified to be easily interpretable by the programmer. The swap example of section 3.1 highlights this issue.

The unifier additionally introduces the boolean connectives $\wedge, \neg$ in annotations. These connectives never occur when translating Clean signatures into UTS, so they are not essential to express uniqueness types. The extra connectives further hinder programmer interpretation of inferred types.

Because any access to a container introduces a disjunctive annotation, unification of disjunctions will occur often. The situation is even worse with larger containers (e.g. records with many fields) that may be nested, because their annotations will be larger disjunctions and hence produce larger unifiers. In our experiments we found that even relatively simple functions can take in the order of seconds to typecheck. This is unacceptable in a modern compiler.

Altogether, whilst moving all of the uniqueness propagation complexity into boolean unification is elegant at first glance, it comes with problems of its own. Boolean unification introduces new non-essential connectives, and inferred types are needlessly large. Inferred signatures are hard interpret for the programmer. Moreover boolean unification and simplification cause unacceptable compile times.

We believe that the use of boolean annotations and unification is not a good approach *in practice*. In the next section, we propose an improved approach.

4 Container Unification

We have seen that unification of disjunctions produces prohibitively large unifiers, and unification of disjunctions occurs often. The types that UTS infers, and the time it takes to infer them, are not acceptable in a modern compiler. However, UTS does make substantial improvements over Clean. We want to preserve the orthogonality of uniqueness types and the absense of solving inequalities. Thus, we set out to find a better unification approach that can replace boolean unification, but otherwise preserves the advantages of UTS.

It is speculated in [7] that it may be possible to limit the type language to just disjunctions. The *choose* example (section 3.4) suggests that the unification of two disjunctions could be simplified: the unification of $u_1 \vee u_2$ with $v_1 \vee v_2$ should yield $u_1 \vee u_2 \vee v_1 \vee v_2$, a solution that contains only disjunctions. Unfortunately that does not quite work. First of all it's not obvious how to define this intuition as a unification: how would one create a substitution out of this idea? Secondly, a unification $u \vee v \doteq \bullet$ still introduces the problem of requiring at least one of u, v to be unique, necesarilly introducing the $\neg, \wedge$ connectives.

Therefore an annotation consisting of just a disjunction on variables is not sufficient. Slightly more structure is required to make unification of a collection of variables with $\bullet$ efficient. Our key idea is to exploit knowledge about how disjunctions arise. Disjunctions are introduced by the **abs** and **fst, snd** rules. In other words, disjunctions are only introduced on containers.

Thus, we replace boolean annotations with special container annotations of the form:

$$(w, \{u_1 \dots u_n \mid \alpha\})$$

In the notation, we explicitly distinguish between variables $u_1 \dots u_n$ occuring in the container, the *member variables*, and the *container variable* w that allows the container to be more unique than any of its elements. Member variables still conceptually constitute a disjunction of variables, but are equiped with a different unification approach inspired by polymorphic records. We define unification on container annotations, and show programmer-interpretable types can be inferred efficiently.

4.1 Comitting to disjunctions

We commit to only using disjunctions in annotations. In the previous section we noted that while the unification is intuitively obvious, it's hard to define what a substitution is in a boolean expression context. Therefore we chose a different approach inspired by polymorphic records [9].

have to add nw variables into containers? unification requires a substitution. we need to still be extensible after one unification. use trick from extensible records.

A disjunction $u_1 \vee \dots \vee u_n$ is written as $\{u_1 \dots u_n \mid \alpha\}$. The variable in the α position is the *extension variable*. A nested disjunction $\{u_1 \dots u_n \mid \{v_1 \dots v_m \mid \alpha\}\}$ is equal to $\{u_1 \dots u_n, v_1 \dots v_m \mid \alpha\}$.

Unification of two disjunctions is defined as:

$$\frac{\alpha \doteq \{v_1 \dots v_m \mid \gamma_1\} \qquad \beta \doteq \{u_1 \dots u_n \mid \gamma_2\} \qquad \gamma_1 \doteq \gamma_2}{\{u_1 \dots u_n \mid \alpha\} \doteq \{v_1 \dots v_m \mid \beta\}}$$

The γ variables enable further unifications. All hypotheses are simple unifications with a variable, resulting in the unifier:

$$[\alpha \mapsto \{v_1 \dots v_m \mid \gamma_1\}, \beta \mapsto \{u_1 \dots u_n \mid \gamma_2\}, \gamma_1 \mapsto \gamma_2]$$

The unification of two disjunctions is therefore efficient, and produces the minimal and desired result.

4.2 Container structure

As previously noted, the unification of a disjunction with $\bullet$ is problematic. The constraint that "at least one is unique" is no easier to express with the new notation.

To solve this problem, we must look at how annotations on containers arise in more detail. Consider the signature $(t^u, s^v)^{u \vee v \vee w}$. From a boolean expression perspective, all variables u, v, w in the $u \vee v \vee w$ disjunction are interchangable: they are just variables. But we know that this disjunction is associated with a tuple type. The annotation on the tuple actually encodes that the uniqueness of the tuple is at least as unique as u and v, and at most as unique as w. This distinction between u, v being a lower bound and w an upper bound on uniqueness cannot be exploited by boolean unification.

Another way to phrase the relation is to write $w \geq u, v$: the uniqueness of w is at least as unique as u and v. Indeed, this is exactly the constraint one would write in Clean (but with a $\leq$ instead of $\geq$ because we're here using the order $\bullet > \times$, not a subtyping relation).

Now suddenly unification with $\bullet$ is simple: If "any of u, v, w must be unique", then certainly $w = \bullet$: we can just pick w to be the unique variable. It may turn out that u or v also are unique, but that is no longer relevant for the uniqueness of the tuple. Note that this trick only works because w occurs freely, and we can thus pick any value for it so long as the container rule is not violated.

The typing rule **abs** also introduces a disjunction, but has no completely unrestricted variable. But we can generalize the **abs** rule slightly to include an extra free variable w in the $\bigvee fv'$ disjunction. This is in fact a generalization over the UTS system: functions can now be more unique than any of their captured variables.

With this insight, we have arrived at container annotations.

4.3 Container annotations

Recall our definition of a container annotation:

$$(w, \{u_1 \dots u_n \mid \alpha\})$$

where

- w is the *container variable.* It is at least as unique as the variables $u_1 \dots u_n$, and therefore equals the uniqueness of the whole container.
- $u_1 \dots u_n$ are the *member variables*: these occur as annotations on the elements of the container. If any of the member variables is unique, the whole container must be unique.
- α is the *extension variable.*

This is a combination of the ideas from the previous two sections: We combine the simple unification of two disjunctions with the simple unification of a disjunction with $\bullet$. The container unification rules are given in figure 4. Crucially, a container unification can still be turned into a boolean disjunction: $(w, \{u_1 \dots u_n \mid \alpha\}) = w \vee u_1 \vee \dots \vee u_n$.

$$\frac{w_1 \doteq w_2 \qquad \alpha \doteq \{v_1 \dots v_m \mid \gamma\} \qquad \beta \doteq \{u_1 \dots u_n \mid \gamma\}}{(w_1, \{u_1 \dots u_n \mid \alpha\}) \doteq (w_2, \{v_1 \dots v_m \mid \beta\})}\ \textbf{co-co}$$

$$\frac{w \doteq \times \qquad u_1 \doteq \times \dots u_n \doteq \times}{(w, \{u_1 \dots u_n \mid \alpha\}) \doteq \times}\ \textbf{non-unique}$$

$$\frac{w \doteq \bullet}{(w, \{u_1 \dots u_n \mid \alpha\}) \doteq \bullet}\ \textbf{unique}$$

Figure 4. container unification rules

4.4 Type system changes

Only minimal changes to the type language and inference rules are required, as shown in figures 5 and 6. The type language is extended with the container annotation, and boolean attributes are removed. Only three of the inference rules require changes:

- In **abs**, the arrow's uniqueness attribute changes from $\bigvee fv'$ to $(w, (range(fv') \mid \alpha))$. Because there is an extra variable w, this is a slight generalization. The function can now be more unique than any of its elements (values captured in the closure).
- In **fst, snd**, the tuple's uniqueness attribute changes from $w \vee u \vee v$ to $(w, \{u, v \mid \alpha\})$.

4.5 equivalence

We show that container unification is equivalent to boolean unification with respect to uniqueness types. The proof is based on the truth tables of annotations after unification. Unification of a container with a variable or $\times$ are straigtforward to prove equivalent to its boolean counterpart. We will look in detail at the other two cases: unification of a disjunction with $\bullet$, and unification of two disjunctions.

4.5.1 Disjunction with $\bullet$. The problem $u \vee v \doteq \bullet$ produces the unifier $[u \mapsto \neg v \vee u]$. In general, $u \vee v_1 \vee \dots \vee v_n$ has unifier $[u \mapsto u \vee (\neg v_1 \wedge \dots \wedge \neg v_n)]$. By the law of the excluded middle, any variable assignment will make the expression evaluate to $\bullet$.

Symmetrically, the problem $(u, \{v_1 \dots v_n \mid \alpha\}) \doteq \bullet$ finds the unifier $[u \mapsto \bullet]$. The container annotation is logically equivalent to the disjunction $u \vee v_1 \dots v_n$, and clearly $\bullet \vee v_1 \dots v_n = \bullet$ for all choices of $u, v_1 \dots v_n$.

4.5.2 Disjunction with Disjunction. This is a generalization of the argument from section 3.3. Consider the unification problem

$$t^{u_1 \vee \dots \vee u_n} \doteq t^{v_1 \vee \dots \vee v_m}$$

The intuition is that the annotation of this type is $\bullet$ if and only if at least one of $u_1, \dots, u_n, v_1, \dots, v_m$ is $\bullet$. The unification problem $u_1 \vee \dots \vee u_n \doteq v_1 \vee \dots \vee v_m$ has unifier S:

$$\begin{aligned} u_1 \mapsto\ & ((u_1 \wedge v_1) \vee (u_1 \wedge v_2) \vee \dots \vee (u_1 \wedge v_m)) \\ & \vee (\neg u_2 \wedge \dots \wedge \neg u_n \wedge v_1) \\ & \vee \dots \\ & \vee (\neg u_2 \wedge \dots \wedge \neg u_n \wedge v_m) \\ u_2 \mapsto\ & (u_2 \wedge v_1) \vee (u_2 \wedge v_2) \vee \dots \vee (u_2 \wedge v_m) \\ & \vdots \\ u_n \mapsto\ & (u_n \wedge v_1) \vee (u_n \wedge v_2) \vee \dots \vee (u_n \wedge v_m) \end{aligned}$$

It is hard to see that this is in fact a unifier. We show that it is in appendix A. Because S is a unifier, it makes the truth tables of the two disjunctions the same. Therefore, if and only if at least one of $u_1, \dots u_n, v_1, \dots v_m$ is $\bullet$, then both $u_1 \vee \dots \vee u_n = \bullet$ and $v_1 \vee \dots \vee v_m = \bullet$. If all variables are $\times$, then both disjunctions are also $\times$.

Now we must show that the truth table of the solution found by container unification is equivalent.

$$\frac{u_1 \doteq v_1 \qquad \alpha \doteq \{v_2, \dots, v_m \mid \gamma\} \qquad \beta \doteq \{u_2, \dots, u_n \mid \gamma\}}{(u_1, \{u_2, \dots, u_n \mid \alpha\}) \doteq (v_1, \{v_2, \dots, v_m \mid \beta\})}\ \textbf{co-co}$$

Rule **co-co** gives the unifier:

$$[u_1 \mapsto v_1, \alpha \mapsto \{v_2, \dots, v_m \mid \gamma\}, \beta \mapsto \{u_2, \dots, u_n \mid \gamma\}]$$

Applying this unifier to either side gives the annotation $(v_1, \{v_2, \dots v_m, u_2, \dots, u_n \mid \gamma\})$.

- if all variables are $\times$, then the container annotation is $\times$.
- if any of $v_1, \dots v_m, u_2, \dots u_n$ are $\bullet$, then the container annotation is $\bullet$.
- if $u_1 = \bullet$, then it must be that $v_1 = \bullet$, and the container annotation must be $\bullet$.

Thus, the truth tables of the uniqueness annotations found with either boolean unification or container unification are equivalent.

4.5.3 Induction. We have proven that after one unification, the truth table of the solution found with boolean unification is identical to the one found using container unification. Now we can write the container annotation as a boolean disjunction again, to get boolean expressions equivalent to the ones found by boolean unification, but consisting solely of disjunctions. Then for subsequent unifications we repeat the argument.

4.6 Loss of generality

Finally, are container annotations less expressive than boolean expressions? Certainly, fewer programs can be typed with container annotations than with boolean annotations. There is no way to express $t^{u \wedge v}$ as a container annotation.

The question is rather whether any useful expressivity is lost. [7] poses the same question. It notes that it is occasonally useful to write an implication as a conjunction, but draws no firm conclusions.

We argue however that no usefull expressivity is lost, because container annotations are just as expresive as Clean's uniqueness implications. For the purposes of expressing uniqueness types, the full expressivity of boolean expressions is not required.

5 Related Work

Uniqueness Types UTS was preceded by Uniqueness Typing Redefined [6], and more background is given in the lead author's PhD thesis [5]. The system – specifically uniqueness inference – of Clean is described in [1]. A more general introduction to Clean can be found in [15].

Usage Analysis we have left refinements to the usage analysis – marking which variables are shared in their scope – to future work. A potential starting point in this area are the counting analyses presented in [11] and [19].

Reference counting has a long tradition, although the focus is commonly on garbage collection. The Sisal project [8] is an early example of using reference counting effectively for inserting destructive updates. More recent examples include SAC [10] and Lean [17].

Array Programming There are several approaches for fast array manipulation in functional languages, but all have to minimize copying of data [13]. Futhark [12] and SAC [16] are functional languages with a big focus on array manipulation. Futhark uses uniqueness types, while SAC performs reference counting to allow safe destructive updates. The Haskell Accelerate library [4] uses an embedded domain-specific language to specify array computations that can be executed on the GPU. A reference counting scheme is used to copy only when needed, and mutate destructively when values are non-shared.

Linear types [20] are a very active area of research, and are increasingly implemented in functional languages, e.g. Haskell [2] and Idris [3]. Where intuitively uniqueness types can guarantee that a value has not been shared in the past, linear types guarantee that a value will not be shared in the future. Both of these guarantees are useful, and there is overlap in their applications.

6 Conclusion & Future Work

We have discussed two approaches to infer uniqueness types: Clean introduces a subtyping rule to use unique values in non-unique contexts. Relations between uniqueness attributes are expressed as constraints. Uniqueness Typing Simplified (UTS) [7] encodes relations between uniqueness attributes

Kind language

$\kappa ::=$		kind
	$\mathcal{T}$	base type
	$\mathcal{U}$	uniqueness attribute
	$\mathcal{R}$	member variable set
	$*$	base type together with uniqueness attribute
	$\kappa_1 \rightarrow \kappa_2$	type constructors

Type constants

`Int, Bool`	$:: \mathcal{T}$	base type
$\rightarrow$	$:: * \rightarrow * \rightarrow \mathcal{T}$	function space
$\bullet, \times$	$:: \mathcal{U}$	unique, non-unique
$(-\{-\mid-\})$	$:: \mathcal{U} \rightarrow \mathcal{U}^* \rightarrow \mathcal{R} \rightarrow \mathcal{U}$	container annotation
`Attr`	$:: \mathcal{T} \rightarrow \mathcal{U} \rightarrow *$	combine base & attribute

Syntactic conventions

t^u	$\equiv$ `Attr` $t\ u$
$a \xrightarrow{u} b$	$\equiv$ `Attr` $(a \rightarrow b)\ u$

Figure 5. kind language

$$\textbf{var}^{\odot}\ \frac{}{\Gamma, x : \tau^{v} \vdash x^{\odot} : \tau^{v}|_{x:v}} \qquad \textbf{var}^{\otimes}\ \frac{}{\Gamma, x : \tau^{\times} \vdash x^{\otimes} : \tau^{\times}|_{x:\times}}$$

$$\textbf{abs}\ \frac{\Gamma, x : \tau \vdash e : \tau'|_{fv} \qquad fv' = fv \mathrel{\rhd} x}{\Gamma \vdash \lambda\, x.e : \tau \xrightarrow{(w,(range(fv')\,|\alpha))} \tau'|_{fv'}}$$

$$\textbf{app}\ \frac{\Gamma \vdash e : \tau \xrightarrow{v} \tau'|_{fv_1} \qquad \Gamma \vdash e' : \tau|_{fv_2}}{\Gamma \vdash e\ e' : \tau'|_{fv_1 \cup fv_2}}$$

$$\textbf{pair}\ \frac{\Gamma \vdash x : t^{u}|_{fv_1} \qquad \Gamma \vdash y : s^{v}|_{fv_2}}{\Gamma \vdash (x, y) : (t^{u}, s^{v})^{w}|_{fv_1 \cup fv_2}}$$

$$\textbf{fst}\ \frac{\Gamma \vdash r : (t^{u}, s^{v})^{(w,\{u,v|\alpha\})}|_{fv}}{\Gamma \vdash \textbf{fst}\ r : t^{u}|_{fv}}$$

$$\textbf{snd}\ \frac{\Gamma \vdash r : (t^{u}, s^{v})^{(w,\{u,v|\alpha\})}|_{fv}}{\Gamma \vdash \textbf{snd}\ r : s^{v}|_{fv}}$$

Figure 6. container annotation typing rules. In rule **abs**, $\rhd$ is the domain subtraction operator. It removes x from the set of free variables because it is bound in the lambda body.

in boolean formulae. Uniqueness polymorphism enables the generic treatment of unique and non-unique values.

An attractive feature of UTS is that uniqueness types are orthogonal to the rest of the type system. However, we highlight two limitations that hinder practical adoption: inferred types are needlesly complex, and unification of uniqueness attributes is a performance bottleneck.

We present container annotations as a modification to UTS that maintains orthogonality of uniqueness types. Container unification is computationally simpler than boolean unification, and infers succinct uniqueness attributes.

References

[1] Erik Barendsen and Sjaak Smetsers. 1995. Uniqueness type inference. In *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 189–206.

[2] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–29.

[3] EDWIN BRADY. [n.d.]. Idris 2: Quantitative Type Theory in Action. ([n. d.]).

[4] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 3–14.

[5] Edsko de Vries. 2008. *Making Uniqueness Typing Less Unique.* Ph.D. Dissertation. Trinity College Dublin.

[6] Edsko De Vries, Rinus Plasmeijer, and David M Abrahamson. 2006. Uniqueness typing redefined. In *Symposium on Implementation and Application of Functional Languages*. Springer, 181–198.

[7] Edsko De Vries, Rinus Plasmeijer, and David M Abrahamson. 2007. Uniqueness typing simplified. In *Symposium on Implementation and Application of Functional Languages*. Springer, 201–218.

[8] John T Feo, David C Cann, and Rodney R Oldehoeft. 1990. A report on the Sisal language project. *J. Parallel and Distrib. Comput.* 10 (1990), 349–366.

[9] Benedict R Gaster and Mark P Jones. 1996. *A polymorphic type system for extensible records and variants.* Technical Report. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University ....

[10] Clemens Grelck and Kai Trojahner. 2004. Implicit memory management for SAC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL*, Vol. 4. 335–348.

[11] Jurriaan Hage, Stefan Holdermans, and Arie Middelkoop. 2007. A generic usage analysis with subeffect qualifiers. *ACM SIGPLAN Notices* 42 (2007), 235–246.

[12] Troels Henriksen. 2017. *Design and implementation of the Futhark programming language.* Ph.D. Dissertation. Department of Computer Science, Faculty of Science, University of Copenhagen.

[13] Paul Hudak and Adrienne Bloss. 1985. The aggregate update problem in functional programming systems. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 300–314.

[14] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of functional programming* 17, 1 (2007), 1–82.

[15] Rinus Plasmeijer, Marko van Eekelen, and John van Groningen. 2011. Clean language report version 2.2.

[16] Sven-Bodo Scholz. 2003. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming* 13, 6 (2003), 1005–1059.

[17] Sebastian Ullrich and Leonardo de Moura. 2019. Counting immutable beans: Reference counting optimized for purely functional programming. *arXiv preprint arXiv:1908.05647* (2019).

[18] Christopher Umans, Tiziano Villa, and Alberto L Sangiovanni-Vincentelli. 2006. Complexity of two-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 7 (2006), 1230–1246.

[19] HL Verstoep. 2013. *Counting analyses.* Master's thesis.

[20] Philip Wadler. 1990. Linear types can change the world!. In *Programming concepts and methods*, Vol. 3. Citeseer, 5.

A Unifier of two disjunctions

The unification problem $u_1 \vee \ldots \vee u_n \doteq v_1 \vee \ldots \vee v_m$ has unifier S:

$$\begin{aligned} u_1 \mapsto\ & ((u_1 \wedge v_1) \vee (u_1 \wedge v_2) \vee \ldots \vee (u_1 \wedge v_m)) \\ & \vee (\neg u_2 \wedge \ldots \wedge \neg u_n \wedge v_1) \\ & \vee \ldots \\ & \vee (\neg u_2 \wedge \ldots \wedge \neg u_n \wedge v_m) \\ u_2 \mapsto\ & (u_2 \wedge v_1) \vee (u_2 \wedge v_2) \vee \ldots \vee (u_2 \wedge v_m) \\ & \vdots \\ u_n \mapsto\ & (u_n \wedge v_1) \vee (u_n \wedge v_2) \vee \ldots \vee (u_n \wedge v_m) \end{aligned}$$

We will now show that this is in fact a unifier. In particular it is hard to see why some $u_i = \bullet$ causes that $v_1 \vee \ldots \vee u_m = \bullet$.

First define $SU = S(u_1 \vee \ldots \vee u_n)$. There are 4 cases:

Case 1: if any $u_i = \bullet, 1 \leq i \leq n$, then $v_1 \vee \ldots \vee v_m = \bullet$:

If $i = 1$, then $S(u_i) \doteq \bullet$, which expands to:

$$\begin{aligned} & (u_1 \wedge v_1) \vee \ldots \vee (u_1 \wedge v_m) \\ & \quad \vee (\neg u_2 \wedge \ldots \wedge \neg u_n \wedge v_1) \\ & \quad \vee \ldots \\ & \quad \vee (\neg u_2 \wedge \ldots \wedge \neg u_n \wedge v_m) \\ & \quad \doteq \bullet \end{aligned}$$

This problem has the unifier

$$[u_1 \mapsto u_2 \vee \ldots \vee u_n \vee u_1, v_1 \mapsto v_1 \vee (\neg v_2 \wedge \ldots \wedge \neg v_m)]$$

Applying the substitution to $v_1 \vee \ldots \vee v_m$ gives:

$$v_1 \vee (\neg v_2 \wedge \ldots \wedge \neg v_m) \vee v_2 \vee \ldots v_m$$

If any $v_1 \ldots v_m = \bullet$, then this expression is $\bullet$. If all v are $\times$, then the second disjunct makes the disjunction evaluate to $\bullet$ anyway.

Otherwise if $i > 1$, then $S(u_i) \doteq \bullet$ expands to:

$$(u_i \wedge v1) \vee (u_i \wedge v_2) \vee \ldots \vee (u_i \wedge v_m)) \doteq \bullet$$

This problem has the unifier:

$$[u_i \mapsto \bullet, v_1 \mapsto v_1 \vee (\neg v_2 \wedge \ldots \wedge \neg v_m)]$$

Applying the substitution to $v_1 \vee \ldots \vee v_m$ gives the same expression as in the case above.

Case 2: if any $v_j = \bullet, 1 \leq j \leq m$, then $u_1 \vee \ldots \vee u_n = \bullet$:

Either:

- Some $u_i = \bullet$. SU contains the disjunct $u_i \wedge v_j$, therefore $SU = \bullet$.
- Otherwise all $u_i = \times$, then SU contains the disjunct $\neg u_2 \wedge \ldots \wedge \neg u_n$, which evaluates to $\bullet$, therefore $SU = \bullet$.

Case 3: if all $u_i = \times, 1 \le i \le n$, then $v_1 \vee \ldots \vee v_m = \times$:

If all $u_i = \times, 1 \le i \le n$, then it must be that $SU \doteq \times$. This problem has unifier

$$[v_1 \mapsto \times, v_2 \mapsto \times, \ldots, v_m \mapsto \times]$$

Therefore $v_1 \vee \ldots \vee v_m = \times$.

Case 4: if all $v_j = \times, 1 \le j \le m$, then $u_1 \vee \ldots \vee u_n = \times$:

In SU, every disjunct is a conjunct containing some v_j. Therefore if all $v_j = \times, 1 \le j \le m$, all disjuncts of SU are $\times$, and thus $SU = \times$.

Polymorphic System I

Alejandro Díaz-Caro[*]
Dpto. de Ciencia y Tecnoloía.
Universidad Nacional de Quilmes.
CONICET–Universidad de Buenos Aires. Instituto de Investigación en Ciencias de la Computación (ICC).
Bernal, Buenos Aires, Argentina
adiazcaro@icc.fcen.uba.ar

Pablo E. Martínez López[*]
Dpto. de Ciencia y Tecnología.
Universidad Nacional de Quilmes.
Bernal, Buenos Aires, Argentina
fidel@unq.edu.ar

Cristian F. Sottile[*]
CONICET–Universidad de Buenos Aires. Instituto de Investigación en Ciencias de la Computación (ICC).
Buenos Aires, Argentina
csottile@icc.fcen.uba.ar

ABSTRACT

System I is a simply-typed lambda calculus with pairs, extended with an equational theory obtained from considering the type isomorphisms as equalities. In this work we propose an extension of System I to polymorphic types, adding the isomorphisms corresponding to the universal quantifier. This is a work in progress proving only subject reduction. For the final version we expect to include a non-standard proof of strong normalisation, extending that of System I.

CCS CONCEPTS

• **Theory of computation** → **Lambda calculus**; **Type theory**; **Proof theory**.

KEYWORDS

Lambda calculus, Type theory, Type isomorphisms, Polymorphism

ACM Reference Format:
Alejandro Díaz-Caro, Pablo E. Martínez López, and Cristian F. Sottile. 2020. Polymorphic System I. In *Proceedings of IFL 2020: The 32nd Symposium on Implementation and Application of Functional Languages (IFL '20).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Two types A and B are considered isomorphic if there exist two functions $f : A \Rightarrow B$ and $g : B \Rightarrow A$ such that the composition $g \circ f$ is semantically equivalent to the identity in A and the composition $f \circ g$ is semantically equivalent to the identity in B. Di Cosmo et al. characterized the isomorphic types in different systems: simple types, simple types with pairs, polymorphism, etc. (cf. [9] for reference). Using such a characterization, System I has been defined [12], a simply-typed lambda calculus with pairs, where isomorphic types are considered equal. In this way, if A and B are isomorphic, every term of type A can be used as a term of type B. For example, the currying rule $(A \wedge B) \Rightarrow C \equiv A \Rightarrow B \Rightarrow C$ allows passing arguments one by one to a function expecting a pair. Normally, this would imply for a function $f : (A \wedge B) \Rightarrow C$ to be transformed through a term t into $tf : A \Rightarrow B \Rightarrow C$. System I goes further, by considering that f has both types $(A \wedge B) \Rightarrow C$ and $A \Rightarrow B \Rightarrow C$, and so, the transformation occurs without the need for the term t. To make this idea work, System I includes an equivalence between terms; for example: $t\langle r, s\rangle \rightleftarrows trs$, since if t expects a pair, it can also take each component at a time. Also, β-reduction have to be parametrized by the type: if the expected argument is a pair, then $t\langle r, s\rangle$ β-reduces; otherwise, it does not β-reduce, but trs does. For example, $(\lambda x^{A\wedge B}.u)\langle r, s\rangle$ β-reduces if r has type A and s has type B. Instead, $(\lambda x^{A}.u)\langle r, s\rangle$ does not reduce directly, but since it is equivalent to $(\lambda x^{A}.u)rs$, which does reduce, then it also reduces.

The idea of identifying some propositions has already been investigated, for example, in Martin-Löf's type theory [20], in the Calculus of Constructions [6], and in Deduction modulo theory [16, 17], where definitionally equivalent propositions, for instance $A \subseteq B$, $A \in \mathcal{P}(B)$, and $\forall x\ (x \in A \Rightarrow x \in B)$ can be identified. But definitional equality does not handle isomorphisms. For example, $A \wedge B$ and $B \wedge A$ are not identified in these logics. Besides definitional equality, identifying isomorphic types in type theory is also a goal of the univalence axiom [22]. From the programming perspective, isomorphisms capture the computational meaning correspondence between types. Taking currying again, for example, we have a function $f : A \wedge B \Rightarrow C$ that can be transformed, because of the fact that there exists an isomorphism, into a function $f' : A \Rightarrow B \Rightarrow C$. These two functions differ in how they can be combined with other terms, but they share a computational meaning: they both computes C given two arguments of types A and B. In this sense, System I's proposal is to allow a programmer to focus on the computational meaning of programs and combining any term with the ones that are combinable with its isomorphic counterparts (e.g. $fx^{A}y^{B}$ and $f'\langle x^{A}, y^{B}\rangle$), ignoring its rigid syntax within the safe context provided by type isomorphisms. From the logic perspective, isomorphisms make proofs more natural. For instance, to prove $(A \wedge (A \Rightarrow B)) \Rightarrow B$ in natural deduction we need to introduce the conjunctive hypothesis $A \wedge (A \Rightarrow B)$ which has to be decomposed into A and $A \Rightarrow B$, while using currying allows to transform the goal to $A \Rightarrow (A \Rightarrow B) \Rightarrow B$ and to directly introduce the hypotheses A and $A \Rightarrow B$, completely eliminating the need for the conjunctive hypotheses.

An interpreter of a preliminary version of System I extended with a recursion operator has been implemented in Haskell [15]. Such

[*]All authors have contributed equally to this research.

$$A \wedge B \equiv B \wedge A \quad (1)$$
$$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C \quad (2)$$
$$A \Rightarrow (B \wedge C) \equiv (A \Rightarrow B) \wedge (A \Rightarrow C) \quad (3)$$
$$(A \wedge B) \Rightarrow C \equiv A \Rightarrow B \Rightarrow C \quad (4)$$
$$\text{if } X \notin FTV(A) \qquad \forall X.(A \Rightarrow B) \equiv A \Rightarrow \forall X.B \quad (5)$$
$$\forall X.(A \wedge B) \equiv \forall X.A \wedge \forall X.B \quad (6)$$

Table 1: Isomorphisms considered in PSI

a language have peculiar characteristics. For example, using the existing isomorphism between $A \Rightarrow (B \wedge C)$ and $(A \Rightarrow B) \wedge (A \Rightarrow C)$, we can project a function computing a pair of elements, and obtain, through evaluation, a simpler function computing only one of the elements of the pair, discarding the unused code that computes the output that is not of interest to us.

In this work in progress we propose an extension of System I to polymorphism, considering some of the isomorphisms corresponding to polymorphic types.

Plan of the paper. The paper is organized as follows: Section 2 introduces the proposed system, and Section 3 gives examples to better clarify the constructions. Section 4 proves the Subject Reduction property, which is the main theorem in the paper. Finally, Section 5 discusses some design choices, as well as possible directions for future work.

2 INTUITIONS AND DEFINITIONS

We define Polymorphic System I (PSI) as an extension of System I [12] to polymorphic types. The syntax of types coincides with System F with pairs [9]:

$$A \quad := \quad X \mid A \Rightarrow A \mid A \wedge A \mid \forall X.A$$

where $X \in \mathcal{TV}ar$, a set of type variables.

However, the extension with respect to System F with pairs consists of adding the following typing rule:

$$\frac{\Gamma \vdash t : A \quad A \equiv B}{\Gamma \vdash t : B}\ (\equiv)$$

valid for every pair of isomorphic types A and B. This non-trivial addition induces the modification of the operational semantics of the calculus.

There are eight isomorphisms characterizing all the valid isomorphism of System F with pairs (cf. [9]). From those eight, we only consider the six given as a congruence in Table 1, where $FTV(A)$ is the set of free type variables defined as usual.

The two not listed isomorphisms are the following:

$$\forall X.A \equiv \forall Y.[X := Y]A \quad (7)$$
$$\forall X.\forall Y.A \equiv \forall Y.\forall X.A \quad (8)$$

The isomorphism (7) is in fact an α-equivalence, and we indeed consider terms and types modulo α-equivalence. We simply do not make this isomorphism explicit in order to avoid confusion.

The isomorphism (8) on the other hand is not treated on this paper because PSI is presented in Church style (as System I), and so, being able to swap the arguments of a type abstraction would imply swapping the typing arguments, and so it would carry a cumbersome notation, with little gain. We will discuss this in Section 5.2.3.

The added typing rule $(\equiv)$ induces certain equivalences between terms. In particular, the isomorphism (1) implies that the pairs $\langle t, r\rangle$ and $\langle r, t\rangle$ are indistinguishable, since both are typed as $A \wedge B$ and also as $B \wedge A$, independently of which term have type A and which term have type B. Then, we consider that those two pairs are equivalent. In the same way, as a consequence of isomorphism (2), $\langle t, \langle r, s\rangle\rangle$ is equivalent to $\langle\langle t, r\rangle, s\rangle$.

Such an equivalence between terms implies that the usual projection, which is defined with respect to the position (i.e. $\pi_i\langle t_1, t_2\rangle \hookrightarrow t_i$), is not well-defined in this system. Indeed, $\pi_1\langle t, r\rangle$ would reduce to t, but since $\langle t, r\rangle$ is equivalent to $\langle r, t\rangle$, it would also reduce to r. Therefore, PSI (as well as System I), defines the projection with respect to a type: If $\Gamma \vdash t : A$ then $\pi_A\langle t, r\rangle \hookrightarrow t$.

This rule turns PSI into a non-deterministic (and therefore non-confluent) system. Indeed, if both t and r have type A, then $\pi_A\langle t, r\rangle$ reduces non-deterministically to t or to r. This non-determinism, however, can be argued not to be of a mayor problem: if we think of PSI as a proof system, then the non-determinism, as soon as we have type preservation, implies that the system identify different proofs of isomorphic propositions (as a form of proof-irrelevance). On the other hand, if PSI is thought as a programming language, then the determinism can be recovered by the following encoding: if t and r have the same type, it suffices to encode the deterministic projection of $\langle t, r\rangle$ into t as $\pi_{B\Rightarrow A}\langle \lambda x^B.t, \lambda x^C.r\rangle s$ where $B \not\equiv C$ and s has type B. Hence, the non-determinism of System I (inherited in PSI) is considered a feature and not a flaw (cf. [12] for a longer discussion).

Therefore, PSI (as well as System I) is one of the many non-deterministic calculi in the literature, e.g. [4, 5, 7, 8, 21] and so our pair-construction operator can also be considered as the parallel composition operator of a non-deterministic calculus.

In non-deterministic calculi, the non-deterministic choice is such that if r and s are two λ-terms, the term $r \oplus s$ represents the computation that runs either r or s non-deterministically, that is such that $(r \oplus s)t$ reduces either to rt or st. On the other hand, the parallel composition operator $\mid$ is such that the term $(r \mid s)t$ reduces to $rt \mid st$ and continue running both rt and st in parallel. In our case, given r and s of type $A \Rightarrow B$ and t of type A, the term $\pi_B(\langle r, s\rangle t)$ is equivalent to $\pi_B\langle rt, st\rangle$, which reduces to rt or st, while the term $\langle rt, st\rangle$ itself would run both computations in parallel. Hence, our pair-constructor is equivalent to the parallel composition while the non-deterministic choice $\oplus$ is decomposed into the pair-constructor followed by its destructor.

In PSI and System I, the non-determinism comes from the interaction of two operators, $\langle, \rangle$ and π. This is also related to the algebraic calculi [1–3, 11, 14, 23], some of which have been designed to express quantum algorithms. There is a clear link between our pair constructor and the projection π, with the superposition constructor $+$ and the measurement π on these algebraic calculi. In these cases, the pair $s + t$ is not interpreted as a non-deterministic choice, but as a superposition of two processes running s and t, and the operator π is the projection related to the measurement, which

$$\dfrac{}{\Gamma, x : A \vdash x : A}\ (ax) \qquad \dfrac{\Gamma \vdash t : A \quad A \equiv B}{\Gamma \vdash t : B}\ (\equiv)$$

$$\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \Rightarrow B}\ (\Rightarrow_i) \qquad \dfrac{\Gamma \vdash t : A \Rightarrow B \quad \Gamma \vdash r : A}{\Gamma \vdash tr : B}\ (\Rightarrow_e)$$

$$\dfrac{\Gamma \vdash t : A \quad \Gamma \vdash r : B}{\Gamma \vdash \langle t, r\rangle : A \wedge B}\ (\wedge_i) \qquad \dfrac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_A(t) : A}\ (\wedge_e)$$

$$\dfrac{\Gamma \vdash t : A \quad X \notin FTV(\Gamma)}{\Gamma \vdash \Lambda X.t : \forall X.A}\ (\forall_i) \qquad \dfrac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t[B] : [X := B]A}\ (\forall_e)$$

Table 2: Typing rules

is the only non-deterministic operator. In such calculi, the distributivity rule $(r + s)t \rightleftarrows rt + st$ is seen as the point-wise definition of the sum of two functions.

The syntax of terms is then similar to that of System F with pairs, but with the projections depending on types instead of position, as discussed:

$$t \quad := \quad x^A \mid \lambda x^A.t \mid tt \mid \langle t, t\rangle \mid \pi_A t \mid \Lambda X.t \mid t[A]$$

where $x^A \in \mathcal{V}ar$, a set of variables. We omit the type of variables when it is evident from the context. For example, we write $\lambda x^A.x$ instead of $\lambda x^A.x^A$.

The type system of PSI is standard, with only two modifications with respect to that of System F with pairs: the projection $(\wedge_e)$, and the added rule $(\equiv)$. The full system is shown in Table 2.

In the same way as isomorphisms (1) and (2) induce the commutativity and associativity of pairs, as well as a modification in the elimination of the pairs (i.e. the projection), the isomorphism (3) induces that some terms must be identified: an abstraction of type $A \Rightarrow (B \wedge C)$ can be considered as a pair of abstractions of type $(A \Rightarrow B) \wedge (A \Rightarrow C)$, and so it can be projected. Therefore, an abstraction returing a pair is identified with a pair of abstractions, and a pair applied, distributes its argument: That is, $\lambda x^A.\langle t, r\rangle \rightleftarrows \langle \lambda x^A.t, \lambda x^A.r\rangle$, and $\langle t, r\rangle s \rightleftarrows \langle ts, rs\rangle$, where $\rightleftarrows$ is a symmetric symbol (and $\rightleftarrows^*$ its transitive closure).

In addition, isomorphism (4) induces the following: $t\langle r, s\rangle \rightleftarrows trs$. However, this equivalence produces an ambiguity with the β-reduction. For example, if t has type A and r has type B, the term $(\lambda x^{A\wedge B}.s)\langle t, r\rangle$ can β-reduce to $[x := \langle t, r\rangle]s$, but also, since this term is equivalent to $(\lambda x^{A\wedge B}.s)tr$, which β-reduces to $([x := t]s)r$, reduction would not be stable by equivalence. To ensure the stability of reduction by equivalence, the β-reduction must be performed only when the type of the argument is the same as the type of the abstracted variable: if $\Gamma \vdash r : A$ then $(\lambda x^A.t)r \hookrightarrow [x := r]t$.

The two added isomorphisms for polymorphism ((5) and (6)) also add several equivalences between terms. Two induced by (5), and four induced by (6).

Summarizing, the operational semantics of PSI is given by the relation $\hookrightarrow$ modulo the symmetric relation $\rightleftarrows$. That is, we consider the relation $\to \ := \ \rightleftarrows^* \circ \hookrightarrow \circ \rightleftarrows^*$. As usual, we write $\to^*$ the reflexive and transitive closure of $\to$. Both relations for PSI are given in Table 3.

$$
\begin{array}{lrl}
 & \langle r, s\rangle \rightleftarrows \langle s, r\rangle & (\textsc{comm}) \\
 & \langle r, \langle s, t\rangle\rangle \rightleftarrows \langle\langle r, s\rangle, t\rangle & (\textsc{asso}) \\
 & \lambda x^A.\langle r, s\rangle \rightleftarrows \langle \lambda x^A.r, \lambda x^A.s\rangle & (\textsc{dist}_\lambda) \\
 & \langle r, s\rangle t \rightleftarrows \langle rt, st\rangle & (\textsc{dist}_{\text{app}}) \\
 & r\langle s, t\rangle \rightleftarrows rst & (\textsc{curry}) \\
\text{if } X \notin FTV(A) & \Lambda X.\lambda x^A.r \rightleftarrows \lambda x^A.\Lambda X.r & (\textsc{p-comm}_{\forall_i\Rightarrow_i}) \\
\text{if } X \notin FTV(A) & (\lambda x^A.t)[B] \rightleftarrows \lambda x^A.t[B] & (\textsc{p-comm}_{\forall_e\Rightarrow_i}) \\
 & \Lambda X.\langle r, s\rangle \rightleftarrows \langle \Lambda X.r, \Lambda X.s\rangle & (\textsc{p-dist}_{\forall_i\wedge_i}) \\
 & \langle r, s\rangle[A] \rightleftarrows \langle r[A], s[A]\rangle & (\textsc{p-dist}_{\forall_e\wedge_i}) \\
 & \pi_{\forall X.A}(\Lambda X.r) \rightleftarrows \Lambda X.\pi_A r & (\textsc{p-dist}_{\forall_i\wedge_e}) \\
\text{if } t : \forall X.(B \wedge C) & (\pi_{\forall X.B} t)[A] \rightleftarrows \pi_{[X:=A]B}(t[A]) & (\textsc{p-dist}_{\wedge_e\forall_e})
\end{array}
$$

$$
\begin{array}{rr}
\text{If } \Gamma \vdash s : A,\ (\lambda x^A.r)s \hookrightarrow [x := s]r & (\beta_\lambda) \\
(\Lambda X.r)[A] \hookrightarrow [X := A]r & (\beta_\Lambda) \\
\text{If } \Gamma \vdash r : A,\ \pi_A\langle r, s\rangle \hookrightarrow r & (\pi)
\end{array}
$$

$$\dfrac{t \rightleftarrows r}{\lambda x^A.t \rightleftarrows \lambda x^A.r} \quad \dfrac{t \rightleftarrows r}{ts \rightleftarrows rs} \quad \dfrac{t \rightleftarrows r}{st \rightleftarrows sr} \quad \dfrac{t \rightleftarrows r}{\langle t, s\rangle \rightleftarrows \langle r, s\rangle}$$

$$\dfrac{t \rightleftarrows r}{\langle s, t\rangle \rightleftarrows \langle s, r\rangle} \quad \dfrac{t \rightleftarrows r}{\pi_A t \rightleftarrows \pi_A r} \quad \dfrac{t \rightleftarrows r}{t[A] \rightleftarrows r[A]} \quad \dfrac{t \rightleftarrows r}{\Lambda X.t \rightleftarrows \Lambda X.r}$$

$$\dfrac{t \hookrightarrow r}{\lambda x^A.t \hookrightarrow \lambda x^A.r} \quad \dfrac{t \hookrightarrow r}{ts \hookrightarrow rs} \quad \dfrac{t \hookrightarrow r}{st \hookrightarrow sr} \quad \dfrac{t \hookrightarrow r}{\langle t, s\rangle \hookrightarrow \langle r, s\rangle}$$

$$\dfrac{t \hookrightarrow r}{\langle s, t\rangle \hookrightarrow \langle s, r\rangle} \quad \dfrac{t \hookrightarrow r}{\pi_A t \hookrightarrow \pi_A r} \quad \dfrac{t \hookrightarrow r}{t[A] \hookrightarrow r[A]} \quad \dfrac{t \hookrightarrow r}{\Lambda X.t \hookrightarrow \Lambda X.r}$$

Table 3: Relations defining the operational semantics of PSI

3 EXAMPLES

In this Section we present some examples to discuss the need for the rules presented.

The first example shows the use of term equivalence to allow applications that are not possible to build in Simple Types. In particular, the function $apply = \lambda f^{A\Rightarrow B}.\lambda x^A.fx$ can ben applied to a pair, e.g. $\langle g, t\rangle$ with $\Gamma \vdash g : A \Rightarrow B$ and $\Gamma \vdash t : A$, because, due to isomorphism (4), the following type derivation is valid:

$$\dfrac{\dfrac{\Gamma \vdash \lambda f^{A\Rightarrow B}.\lambda x^A.fx : (A \Rightarrow B) \Rightarrow A \Rightarrow B}{\Gamma \vdash \lambda f^{A\Rightarrow B}.\lambda x^A.fx : ((A \Rightarrow B) \wedge A) \Rightarrow B}\ (\equiv) \quad \dfrac{\Gamma \vdash g : A \Rightarrow B \quad \Gamma \vdash t : A}{\Gamma \vdash \langle g, t\rangle : (A \Rightarrow B) \wedge A}\ (\wedge_i)}{\Gamma \vdash (\lambda f^{A\Rightarrow B}.\lambda x^A.fx)\langle g, t\rangle : B}\ (\Rightarrow_e)$$

and we have $(\lambda f^{A\Rightarrow B}.\lambda x^A.fx)\langle g, t\rangle \rightleftarrows (\lambda f^{A\Rightarrow B}.\lambda x^A.fx)gt \hookrightarrow^2_\beta gt$.

The second example shows that the same application can be used in other ways. The term $(\lambda f^{A\Rightarrow B}.\lambda x^A.fx)tg$ is also well-typed, using isomorphisms (1) and (4), and reduces to gt: $(\lambda f^{A\Rightarrow B}.\lambda x^A.fx)tg \rightleftarrows (\lambda f^{A\Rightarrow B}.\lambda x^A.fx)\langle t, g\rangle \rightleftarrows (\lambda f^{A\Rightarrow B}.\lambda x^A.fx)\langle g, t\rangle \to^* gt$.

The uncurried function *apply'* $= \lambda x^{(A\Rightarrow B)\wedge A}.\pi_{A\Rightarrow B}(x)\pi_A(x)$ can be applied to $\Gamma \vdash g : A \Rightarrow B$ and $\Gamma \vdash t : A$ as if it was curried:

$$\begin{aligned}&(\lambda x^{(A\Rightarrow B)\wedge A}.\pi_{A\Rightarrow B}(x)\pi_A(x))gt\\ &\rightleftarrows (\lambda x^{(A\Rightarrow B)\wedge A}.\pi_{A\Rightarrow B}x\pi_A x)\langle g,t\rangle\\ &\hookrightarrow_\beta \pi_{A\Rightarrow B}\langle g,t\rangle\pi_A\langle g,t\rangle\\ &\hookrightarrow_\pi^2 gt\end{aligned}$$

Another example of interest is the one mentioned in Section 2: a function returning a pair can be projected. Consider the term $\pi_{A\Rightarrow B}(\lambda x^A.\langle t,r\rangle)$, where $\Gamma, x : A \vdash t : B$ and $\Gamma, x : A \vdash r : C$. This term is typable, using isomorphism (3), since $A \Rightarrow (B \wedge C) \equiv (A \Rightarrow B) \wedge (A \Rightarrow C)$. The reduction goes as follows: $\pi_{A\Rightarrow B}(\lambda x^A.\langle t,r\rangle) \rightleftarrows \pi_{A\Rightarrow B}\langle \lambda x^A.t, \lambda x^A.r\rangle \hookrightarrow_\pi \lambda x^A.t$. Note that the function is projected even while not being applied, returning another function.

Rule ($\textsc{p-comm}_{\forall_i\Rightarrow_i}$) is a consequence of isomorphism (5). The term $(\Lambda X.\lambda x^A.\lambda f^{A\Rightarrow X}.fx)t$, for instance, is well-typed assuming $\Gamma \vdash t : A$ and $X \notin FTV(A)$:

$$\dfrac{\dfrac{\Gamma \vdash \Lambda X.\lambda x^A.\lambda f^{A\Rightarrow X}.fx : \forall X.(A \Rightarrow (A \Rightarrow X) \Rightarrow X)}{\Gamma \vdash \Lambda X.\lambda x^A.\lambda f^{A\Rightarrow X}.fx : A \Rightarrow \forall X.((A \Rightarrow X) \Rightarrow X)}(\equiv) \qquad \Gamma \vdash t : A}{\Gamma \vdash (\Lambda X.\lambda x^A.\lambda f^{A\Rightarrow X}.fx)t : \forall X.((A \Rightarrow X) \Rightarrow X)}(\Rightarrow_e)$$

and we have $(\Lambda X.\lambda x^A.\lambda f^{A\Rightarrow X}.fx)t \rightleftarrows (\lambda x^A.(\Lambda X.\lambda f^{A\Rightarrow X}.fx))t \hookrightarrow_{\beta_\lambda} (\Lambda X.\lambda f^{A\Rightarrow X}.ft)$.

Rule ($\textsc{p-comm}_{\forall_e\Rightarrow_i}$) is also a consequence of isomorphism (5). Consider the term $(\lambda x^{\forall X.(X\Rightarrow X)}.x)[A]\Lambda X.\lambda x^X.x$. Let $\mathbb{X} = \forall X.(X \Rightarrow X)$. Since $\mathbb{X} \Rightarrow \mathbb{X} \equiv \forall Y.(\mathbb{X} \Rightarrow (Y \Rightarrow Y))$ (renaming the variable for readability), then, $\vdash (\lambda x^{\mathbb{X}}.x)[A]\Lambda X.\lambda x^X.x : A \Rightarrow A$.

The reduction goes as follows: $(\lambda x^{\forall X.(X\Rightarrow X)}.x)[A]\Lambda X.\lambda x^X.x \rightleftarrows (\lambda x^{\forall X.(X\Rightarrow X)}.x[A])\Lambda X.\lambda x^X.x \hookrightarrow_{\beta_\lambda} (\Lambda X.\lambda x^X.x)[A] \hookrightarrow_{\beta_\Lambda} \lambda x^A.x$.

Rules ($\textsc{p-dist}_{\forall_i\wedge_i}$) and ($\textsc{p-dist}_{\forall_i\wedge_e}$) are both consequences of the same isomorphism: (6). Consider the term $\pi_{\forall X.(X\Rightarrow X)}(\Lambda X.\langle \lambda x^X.x, t\rangle)$. Since $\forall X.((X \Rightarrow X) \wedge A) \equiv (\forall X.(X \Rightarrow X)) \wedge (\forall X.A)$, we can derive $\Gamma \vdash \pi_{\forall X.(X\Rightarrow X)}(\Lambda X.\langle \lambda x^X.x, t\rangle) : \forall X.(X \Rightarrow X)$. A possible reduction is:

$$\begin{aligned}\pi_{\forall X.(X\Rightarrow X)}(\Lambda X.\langle \lambda x^X.x, t\rangle) &\rightleftarrows \pi_{\forall X.(X\Rightarrow X)}\langle \Lambda X.\lambda x^X.x, \Lambda X.t\rangle\\ &\hookrightarrow_\pi \Lambda X.\lambda x^X.x\end{aligned}$$

Another consequence of isomorphism (6) is the rule ($\textsc{p-dist}_{\forall_e\wedge_i}$). Consider $\langle \Lambda X.\lambda x^X.\lambda y^A.t,\ \Lambda X.\lambda x^X.\lambda z^B.r\rangle[C]$ where t has type D and r has type E. Since $\forall X.(X \Rightarrow A \Rightarrow D) \wedge \forall X.(X \Rightarrow B \Rightarrow E) \equiv \forall X.((X \Rightarrow A \Rightarrow D) \wedge (X \Rightarrow B \Rightarrow E))$, we have $\langle \Lambda X.\lambda x^X.\lambda y^A.t, \Lambda X.\lambda x^X.\lambda z^B.r\rangle[C] : (C \Rightarrow A \Rightarrow D) \wedge (C \Rightarrow B \Rightarrow E)$. It reduces as follows: $\langle \Lambda X.\lambda x^X.\lambda y^A.t,\ \Lambda X.\lambda x^X.\lambda z^B.r\rangle[C] \rightleftarrows \langle (\lambda x^X.\lambda y^A.t)[C], (\lambda x^X.\lambda z^B.r)[C]\rangle \hookrightarrow_{\beta_\Lambda} \langle \lambda x^C.\lambda y^A.t,\ \lambda x^C.\lambda z^B.r\rangle$.

Finally, rule ($\textsc{p-dist}_{\wedge_e\forall_e}$) is also a consequence of isomorphism (6). Consider the term $(\pi_{\forall X.(X\Rightarrow X)}(\Lambda X.\langle \lambda x^X.x, r\rangle))[A]$, with type $A \Rightarrow A$, which reduces as follows:

$$\begin{aligned}(\pi_{\forall X.(X\Rightarrow X)}(\Lambda X.\langle \lambda x^X.x, r\rangle))[A] &\rightleftarrows \pi_{A\Rightarrow A}((\Lambda X.\langle \lambda x^X.x, r\rangle)[A])\\ &\hookrightarrow_{\beta_\Lambda} \pi_{A\Rightarrow A}\langle \lambda x^A.x, [X := A]r\rangle\\ &\hookrightarrow_\pi \lambda x^A.x\end{aligned}$$

4 SUBJECT REDUCTION

In this section we prove the preservation of typing through reduction. First we need to characterize the equivalences between types, for example, if $\forall X.A \equiv B \wedge C$, then $B \equiv \forall X.B'$ and $C \equiv \forall X.C'$, with $A \equiv B' \wedge C'$ (Lemma 4.9). Due to the number of isomorphisms, this kind of lemmas are not trivial. To prove these relations, we first define the multiset of prime factors of a type (Definition 4.1). That is, the multiset of types that are not equivalent to a conjunction, such that the conjunction of all its elements is equivalent to a certain type. This technique has already been used in System I [12], however, it has been used with simply types with only one basic type τ. In PSI, instead, we have an infinite number of variables acting as basic types, hence the proof becomes more complex.

We write $\forall\vec{X}.A$ for $\forall X_1.\forall X_2.\ldots.\forall X_n.A$, for some $n \geq 0$ (where if $n = 0$, $\forall\vec{X}.A = A$).

Definition 4.1 (Prime factors).

$$\begin{aligned}PF(X) &= [X]\\ PF(A \Rightarrow B) &= [\forall\vec{X}_i.((A \wedge B_i) \Rightarrow Y_i)]_{i=1}^n\\ &\qquad \text{where } PF(B) = [\forall\vec{X}_i.(B_i \Rightarrow Y_i)]_{i=1}^n\\ PF(A \wedge B) &= PF(A) \uplus PF(B)\\ PF(\forall X.A) &= [\forall X.\forall\vec{Y}_i.(A_i \Rightarrow Z_i)]_{i=1}^n\\ &\qquad \text{where } PF(A) = [\forall\vec{Y}_i.(A_i \Rightarrow Z_i)]_{i=1}^n\end{aligned}$$

Lemma 4.2 and Corollary 4.3 state the correctness of Definition 4.1. We write $\bigwedge([A_i]_i)$ for $\bigwedge_i A_i$.

Lemma 4.2. *For all A, there exist $\vec{X}, n, B_1, \ldots, B_n, Y_1, \ldots, Y_n$ such that $PF(A) = [\forall\vec{X}_i.(B_i \Rightarrow Y_i)]_{i=1}^n$.*

Proof. Straightforward induction on the structure of A. ☐

Corollary 4.3. *For all A, $A \equiv \bigwedge(PF(A))$.*

Proof. By induction on the structure of A.

- Let $A = X$. Then $PF(X) = [X]$, and $\bigwedge([X]) = X$.
- Let $A = B \Rightarrow C$. By Lemma 4.2, $PF(C) = [\forall\vec{X}_i.(C_i \Rightarrow Y_i)]_{i=1}^n$. Hence, by definition, $PF(A) = [\forall\vec{X}_i.(B \wedge C_i \Rightarrow Y_i)]_{i=1}^n$. By the induction hypothesis, $C \equiv \bigwedge(PF(C)) = \bigwedge_{i=1}^n \forall\vec{X}_i.(C_i \Rightarrow Y_i)$. Therefore, $A = B \Rightarrow C \equiv B \Rightarrow \bigwedge_{i=1}^n \forall\vec{X}_i.(C_i \Rightarrow Y_i) \equiv \bigwedge_{i=1}^n \forall\vec{X}_i.((B \wedge C_i) \Rightarrow Y_i) = \bigwedge([\forall\vec{X}_i.(B \wedge C_i \Rightarrow Y_i)]_{i=1}^n) = \bigwedge(PF(A))$.
- Let $A = B \wedge C$. By the induction hypothesis, $B \equiv \bigwedge(PF(B))$ and $C \equiv \bigwedge(PF(C))$. Hence, $A = B \wedge C \equiv \bigwedge(PF(B)) \wedge \bigwedge(PF(C)) \equiv \bigwedge(PF(B) \uplus PF(C)) = \bigwedge(PF(A))$.
- Let $A = \forall X.B$. By Lemma 4.2, $PF(B) = [\forall\vec{Y}_i.(B_i \Rightarrow Z_i)]_{i=1}^n$. Hence, by definition, $PF(A) = [\forall X.\forall\vec{Y}_i.(B_i \Rightarrow Z_i)]_{i=1}^n$. By the induction hypothesis, $B \equiv \bigwedge(PF(B)) = \bigwedge_{i=1}^n \forall\vec{Y}_i.(B_i \Rightarrow Z_i)$. Therefore, $A = \forall X.B \equiv \forall X.\bigwedge_{i=1}^n \forall\vec{Y}_i.(B_i \Rightarrow Z_i) \equiv \bigwedge_{i=1}^n \forall X.\forall\vec{Y}_i.(B_i \Rightarrow Z_i) = \bigwedge([\forall X.\forall\vec{Y}.(B_i \Rightarrow Z)]_{i=1}^n) = \bigwedge(PF(A))$. ☐

Lemma 4.5 states the stability of prime factors through equivalence and Lemma 4.6 states a sort of reciprocal result.

Definition 4.4. $[A_1, \dots, A_n] \sim [B_1, \dots, B_m]$ if $n = m$ and $A_i \equiv B_{p(i)}$, for $i = 1, \dots, n$ and p a permutation on $\{1, \dots .n\}$.

Lemma 4.5. *For all A, B, if $A \equiv B$, then $PF(A) \sim PF(B)$.*

Proof. First we check that $PF(A \wedge B) \sim PF(B \wedge A)$ and similar for the other five isomorphisms. Then we prove by structural induction that if A and B are equivalent in one step, then $PF(A) \sim PF(B)$. We conclude by an induction on the length of the derivation of the equivalence $A \equiv B$. □

Lemma 4.6. *For all R, S, if $R \sim S$, then $\bigwedge(R) \equiv \bigwedge(S)$.* □

Lemma 4.7. *For all $\vec{X}, \vec{Z}, A, B, Y, W$, if $\forall\vec{X}.(A \Rightarrow Y) \equiv \forall\vec{Z}.(B \Rightarrow W)$, then $\vec{X} = \vec{Z}$, $A \equiv B$, and $Y = W$.*

Proof. By simple inspection of the isomorphisms. □

Lemma 4.8. *For all A, B, C_1, C_2, if $A \Rightarrow B \equiv C_1 \wedge C_2$, then there exist B_1, B_2 such that $C_1 \equiv A \Rightarrow B_1$, $C_2 \equiv A \Rightarrow B_2$ and $B \equiv B_1 \wedge B_2$.*

Proof. By Lemma 4.5, $PF(A \Rightarrow B) \sim PF(C_1 \wedge C_2) = PF(C_1) \uplus PF(C_2)$.

By Lemma 4.2, let $PF(B) = [\forall\vec{X}_i.(D_i \Rightarrow Z_i)]_{i=1}^n$, $PF(C_1) = [\forall\vec{Y}_j.(E_j \Rightarrow Z'_j)]_{j=1}^k$, and $PF(C_2) = [\forall\vec{Y}_j.(E_j \Rightarrow Z'_j)]_{j=k+1}^m$. Hence, $[\forall\vec{X}_i.((A \wedge D_i) \Rightarrow Z_i)]_{i=1}^n \sim [\forall\vec{Y}_j.(E_j \Rightarrow Z'_j)]_{j=1}^m$. So, by definition of $\sim$, $n = m$ and for $i = 1, \dots, n$ and a permutation p, we have $\forall\vec{X}_i.((A \wedge D_i) \Rightarrow Z_i) \equiv \forall\vec{Y}_{p(i)}.(E_{p(i)} \Rightarrow Z'_{p(i)})$. Hence, by Lemma 4.7, we have $\vec{X}_i = \vec{Y}_{p(i)}$, $A \wedge D_i \equiv E_{p(i)}$, and $Z_i = Z'_{p(i)}$.

Thus, there exists I such that $I \cup \bar{I} = \{1, \dots, n\}$, such that

$$PF(C_1) = [\forall\vec{Y}_{p(i)}.(E_{p(i)} \Rightarrow Z'_{p(i)})]_{i \in I}$$
$$PF(C_2) = [\forall\vec{Y}_{p(i)}.(E_{p(i)} \Rightarrow Z'_{p(i)})]_{i \in \bar{I}}$$

Therefore, by Corollary 4.3, $C_1 \equiv \bigwedge_{i \in I} \forall\vec{Y}_{p(i)}.(E_{p(i)} \Rightarrow Z'_{p_i}) \equiv \bigwedge_{i \in I} \forall\vec{X}_i.((A \wedge D_i) \Rightarrow Z_i)$ and $C \equiv \bigwedge_{i \in \bar{I}} \forall\vec{X}_i.((A \wedge D_i) \Rightarrow Z_i)$.

Let $B_1 = \bigwedge_{i \in I} \forall\vec{X}_i.(D_i \Rightarrow Z_i)$ and $B_2 = \bigwedge_{i \in \bar{I}} \forall\vec{X}_i.(D_i \Rightarrow Z_i)$. So, $C_1 \equiv A \Rightarrow B_1$ and $C_2 \equiv A \Rightarrow B_2$. In addition, also by Corollary 4.3, we have $B \equiv \bigwedge_{i=1}^n \forall\vec{X}_i.(D_i \Rightarrow Z_i) \equiv B_1 \wedge B_2$. □

The proofs of the following two lemmas are similar to the proof of Lemma 4.8. Full details are given in the appendix.

Lemma 4.9. *For all X, A, B, C, if $\forall X.A \equiv B \wedge C$, then there exist B', C' such that $B \equiv \forall X.B'$, $C \equiv \forall X.C'$ and $A \equiv B' \wedge C'$.* □

Lemma 4.10. *For all X, A, B, C, if $\forall X.A \equiv B \Rightarrow C$, then there exists C' such that $C \equiv \forall X.C'$ and $A \equiv B \Rightarrow C'$.* □

Since the calculus is presented in Church-style, excluding rule $\equiv$, the PSI is syntax directed. Therefore, the generation lemma (Lemma 4.12) is straightforward, and we have the following unicity lemma (whose proof is given in the appendix):

Lemma 4.11 (Unicity modulo). *For all Γ, r, A, B, if $\Gamma \vdash r : A$ and $\Gamma \vdash r : B$, then $A \equiv B$.* □

Lemma 4.12 (Generation). *For all Γ, x, r, s, X, A, B:*

1. *If $\Gamma \vdash x : A$ and $\Gamma \vdash x : B$, then $A \equiv B$.*
2. *If $\Gamma \vdash \lambda x^A.r : B$, then there exists C such that $\Gamma, x : A \vdash r : C$ and $B \equiv A \Rightarrow C$.*
3. *If $\Gamma \vdash rs : A$, then there exists C such that $\Gamma \vdash r : C \Rightarrow A$ and $\Gamma \vdash s : C$.*
4. *If $\Gamma \vdash \langle r, s \rangle : A$, then there exist C, D such that $A \equiv C \wedge D$, $\Gamma \vdash r : C$ and $\Gamma \vdash s : D$.*
5. *If $\Gamma \vdash \pi_A r : B$, then $A \equiv B$ and there exists C such that $\Gamma \vdash r : B \wedge C$.*
6. *If $\Gamma \vdash \Lambda X.r : A$, then there exists C such that $A \equiv \forall X.C$, $\Gamma \vdash r : C$ and $X \notin FTV(\Gamma)$.*
7. *If $\Gamma \vdash r[A] : B$, then there exists C such that $[X := A]C \equiv B$ and $\Gamma \vdash r : \forall X.C$.* □

The detailed proofs of Lemma 4.13 (Substitution) and Theorem 4.14 (Subject Reduction) are given in the appendix.

Lemma 4.13 (Substitution).

1. *For all Γ, x, r, s, A, B, if $\Gamma, x : B \vdash r : A$ and $\Gamma \vdash s : B$ then $\Gamma \vdash [x := s]r : A$.*
2. *For all Γ, r, X, A, B, if $\Gamma \vdash r : A$, then $[X := B]\Gamma \vdash [X := B]r : [X := B]A$.* □

Theorem 4.14 (Subject reduction). *For all Γ, r, s, A, if $\Gamma \vdash r : A$ and $r \hookrightarrow s$ or $r \rightleftarrows s$, then $\Gamma \vdash s : A$.* □

5 CONCLUSION, DISCUSSION AND FUTURE WORK

System I is a proof system for propositional logic, where isomorphic propositions have the same proofs. In this paper we have defined PSI, a polymorphic extension of System I where two of the isomorphisms corresponding to the universal quantifier were added. This is a step towards obtaining a system that identifies all the isomorphisms (which have been characterized by Di Cosmo [9]).

5.1 Termination (work in progress)

The strong normalisation of System I has been proved [12], using a non-trivial reformulation of Tait's classical proof for Simple Types. Indeed, in System I we cannot define a notion of neutral terms [19], which are usually defined being the elimination terms (i.e. application, projection). In System I, and so in PSI, being neutral is not stable through equivalence $\rightleftarrows$. For instance, $\langle r, s \rangle t$ is an application, thus it is neutral, but its equivalent term $\langle rt, st \rangle$ is a pair, which is not neutral. Therefore, our proof does not rely on the definition of neutral terms and the so called CR3 property. We claim that it is possible to extend such a proof technique to PSI, and it is ongoing work.

5.2 Other future work

5.2.1 Implementation and fix point. As mentioned in the previous section, we have already proposed an implementation of an early version of System I, extended with a fix point operator [15]. We plan to extend such an implementation for polymorphism, following the design of PSI.

5.2.2 Towards more connectives. It is a subtle question how to add a neutral element of the conjunction, which would imply more isomorphisms, e.g. $A \wedge \top \equiv A$, $A \Rightarrow \top \equiv \top$ and $\top \Rightarrow A \equiv A$ [9]. Adding the equation $\top \Rightarrow \top \equiv \top$ would make it possible to derive $(\lambda x^\top.xx)(\lambda x^\top.xx) : \top$; however, this term is not the classical Ω, since it is typed by $\top$, and by imposing some restrictions on the

β-reduction, it could be forced not to reduce to itself but to discard its argument. For example: "If $A \equiv \top$, then $(\lambda x^A.r)s \hookrightarrow r[\star/x]$", where $\star : \top$ is the introduction rule of $\top$.

5.2.3 Swap. As mentioned in Section 2, two isomorphisms for System F with pairs, as defined by Di Cosmo [9], are not considered explicitly: isomorphisms (7) and (8). However, the isomorphism (7) is just the α-equivalence, which has been given implicitly, and so it has indeed been considered. The isomorphism that actually was not considered is (8), which allows to swap the type abstractions: $\forall X.\forall Y.A \equiv \forall Y.\forall X.A$. This isomorphism is the analogous to the isomorphism $A \Rightarrow B \Rightarrow C \equiv B \Rightarrow A \Rightarrow C$ at the first order level, which is a consequence of isomorphisms (4) and (1). At this first order level, the isomorphism induces the following equivalence: $(\lambda x^A.\lambda y^B.r)st \rightleftarrows (\lambda x^A.\lambda y^B.r)\langle s,t\rangle \rightleftarrows (\lambda x^A.\lambda y^B.r)\langle t,s\rangle \rightleftarrows (\lambda x^A.\lambda y^B.r)ts$

An alternative approach would have been to introduce an equivalence between $\lambda x^A.\lambda y^B.r$ and $\lambda y^B.\lambda x^A.r$. However, in any case, to keep subject reduction, the β_λ reduction must verify that the type of the argument matches the type of the variable before reducing. This solution is not easily implementable for the β_Λ reduction, since it involves using the type as a labelling for the term and the variable, to identify which term corresponds to which variable (leaving the posibility for non-determinism if the "labellings" are duplicated), but at the level of types we do not have a natural labelling.

Another alternative solution, in the same direction, is the one implemented by the selective lambda calculus [18], where only arrows, and not conjunctions, were considered, and so only the ismorphism $A \Rightarrow B \Rightarrow C \equiv B \Rightarrow A \Rightarrow C$ is treated. In the selective lambda calculus the solution is indeed to include external labellings (not types) to identify which argument is being used at each time. We could have added a labelling to type applications, $t[A_X]$, together with the following rule: $r[A_X][B_Y] \rightleftarrows r[B_Y][A_X]$ and so modifying the β_Λ to $(\Lambda X.r)[A_X] \hookrightarrow [X := A]r$.

Despite that such a solution seems to work, we found that it does not contribute to the language in any aspect, while it does make the system less readable. Therefore, we have decided to exclude the isomorphism (8) for PSI.

5.2.4 Eta-expansion rule. An extended fragment of an early version of System I [10] has been implemented in Haskell [15]. In such an implementation, we have added some ad-hoc rules in order to have a progression property (that is, having only introductions as normal forms of closed terms). For example, "If $s : B$ then $(\lambda x^A.\lambda y^B.r)s \hookrightarrow \lambda x^A.((\lambda y^B.r)s)$". Such a rule, among others introduced in this implementation, is a particular case of a more general η-expansion rule. Indeed, with the rule $t \hookrightarrow \lambda x^A.tx$ we can derive

$$\begin{aligned}(\lambda x^A.\lambda y^B.r)s &\hookrightarrow \lambda z^A.(\lambda x^A.\lambda y^B.r)sz \\ &\rightleftarrows^* \lambda z^A.(\lambda x^A.\lambda y^B.r)zs \\ &\hookrightarrow \lambda z^A.((\lambda y^B.r[z/x])s)\end{aligned}$$

In [13] we have showed that it is indeed the case that all the ad-hoc rules from [10] can be lifted by adding extensional rules.

We left as a future work to add these extensional rules to PSI and show a progression property for it.

REFERENCES

[1] Pablo Arrighi and Alejandro Díaz-Caro. 2012. A System F Accounting for Scalars. *LMCS* 8, 1:11 (2012), 1–32.
[2] Pablo Arrighi, Alejandro Díaz-Caro, and Benoît Valiron. 2017. The Vectorial Lambda-Calculus. *Inf. and Comp.* 254, 1 (2017), 105–139.
[3] Pablo Arrighi and Gilles Dowek. 2017. Lineal: A linear-algebraic lambda-calculus. *LMCS* 13, 1:8 (2017), 1–33.
[4] Gérard Boudol. 1994. Lambda-Calculi for (Strict) Parallel Functions. *Inf. and Comp.* 108, 1 (1994), 51–127.
[5] Antonio Bucciarelli, Thomas Ehrhard, and Giulio Manzonetto. 2012. A Relational Semantics for Parallelism and Non-Determinism in a Functional Setting. *APAL* 163, 7 (2012), 918–934.
[6] Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Inf. and Comp.* 76, 2–3 (1988), 95–120.
[7] Ugo de'Liguoro and Adolfo Piperno. 1995. Non Deterministic Extensions of Untyped λ-calculus. *Inf. and Comp.* 122, 2 (1995), 149–177.
[8] Mariangiola Dezani-Ciancaglini, Ugo de'Liguoro, and Adolfo Piperno. 1998. A filter model for concurrent λ-calculus. *SIAM JComp.* 27, 5 (1998), 1376–1419.
[9] Roberto Di Cosmo. 1995. *Isomorphisms of types: from λ-calculus to information retrieval and language design.* Birkhauser, Switzerland.
[10] Alejandro Díaz-Caro and Gilles Dowek. 2013. Non determinism through type isomorphism. *EPTCS (LSFA'12)* 113 (2013), 137–144.
[11] Alejandro Díaz-Caro and Gilles Dowek. 2017. Typing quantum superpositions and measurement. *LNCS (TPNC'17)* 10687 (2017), 281–293.
[12] Alejandro Díaz-Caro and Gilles Dowek. 2019. Proof Normalisation in a Logic Identifying Isomorphic Propositions. *LIPIcs (FSCD'19)* 131 (2019), 14:1–14:23.
[13] Alejandro Díaz-Caro and Gilles Dowek. 2020. Extensional proofs in a propositional logic modulo isomorphisms. Draft at `arXiv:2002.03762`.
[14] Alejandro Díaz-Caro, Mauricio Guillermo, Alexandre Miquel, and Benoît Valiron. 2019. Realizability in the Unitary Sphere. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2019).* IEEE, Vancouver, BC, Canada, 1–13.
[15] Alejandro Díaz-Caro and Pablo E. Martínez López. 2015. Isomorphisms considered as equalities: Projecting functions and enhancing partial application through an implementation of λ^+. *ACM IFL* 2015, 9 (2015), 1–11.
[16] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. 2003. Theorem proving modulo. *JAR* 31, 1 (2003), 33–72.
[17] Gilles Dowek and Benjamin Werner. 2003. Proof normalization modulo. *JSL* 68, 4 (2003), 1289–1316.
[18] Jacques Garrigue and Hassan Aït-Kaci. 1994. The Typed Polymorphic Label-Selective λ-Calculus. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94).* Association for Computing Machinery, New York, NY, USA, 35–47.
[19] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types.* Cambridge U.P., UK.
[20] Per Martin-Löf. 1984. *Intuitionistic type theory.* Bibliopolis, Napoli, Italy.
[21] Michele Pagani and Simona Ronchi Della Rocca. 2010. Linearity, non-determinism and solvability. *Fund. Inf.* 103, 1–4 (2010), 173–202.
[22] The Univalent Foundations Program. 2013. *HoTT: Univalent Foundations of Mathematics.* Institute for Advanced Study, Princeton, NJ, USA.
[23] Lionel Vaux. 2009. The algebraic lambda calculus. *MSCS* 19, 5 (2009), 1029–1059.

A DETAILED PROOFS

Lemma 4.9. *If $\forall X.A \equiv B \wedge C$, then $B \equiv \forall X.B'$, $C \equiv \forall X.C'$ and $A \equiv B' \wedge C'$.*

Proof. By Lemma 4.5, $PF(\forall X.A) \sim PF(B \wedge C) = PF(B) \uplus PF(C)$. By Lemma 4.2, let

$$PF(A) = [\forall \vec{Y_i}.(A_i \Rightarrow Z_i)]_{i=1}^n$$
$$PF(B) = [\forall \vec{W_j}.(D_j \Rightarrow Z'_j)]_{j=1}^k$$
$$PF(C) = [\forall \vec{W_j}.(D_j \Rightarrow Z'_j)]_{j=k+1}^m$$

Hence, $[\forall X.\forall \vec{Y_i}.(A_i \Rightarrow Z_i)]_{i=1}^n \sim [\forall \vec{W_j}.(D_j \Rightarrow Z'_j)]_{j=1}^m$. So, by definition of $\sim$, $n = m$ and for $i = 1, \dots, n$ and a permutation p, we have

$$\forall X.\forall \vec{Y_i}.(A_i \Rightarrow Z_i) \equiv \forall \vec{W}_{p(i)}.(D_{p(i)} \Rightarrow Z'_{p(i)})$$

Thus, by Lemma 4.7, we have $X, \vec{Y_i} = \vec{W}_{p(i)}$, $A_i \equiv D_{p(i)}$, and $Z_i = Z'_{p(i)}$. Therefore, there exists I such that $I \cup \bar{I} = \{1, \dots, n\}$, such that

$$PF(B) = [\forall \vec{W}_{p(i)}.(D_{p(i)} \Rightarrow Z'_{p(i)})]_{i \in I}$$
$$PF(C) = [\forall \vec{W}_{p(i)}.(D_{p(i)} \Rightarrow Z'_{p(i)})]_{i \in \bar{I}}$$

Hence, by Corollary 4.3, we have, $B \equiv \bigwedge_{i \in I} \forall \vec{W}_{p(i)}.(D_{p(i)} \Rightarrow Z'_{p_i}) \equiv \bigwedge_{i \in I} \forall X.\forall \vec{Y_i}.(A_i \Rightarrow Z_i)$, and $C \equiv \bigwedge_{i \in \bar{I}} \forall X.\forall \vec{Y_i}.(A_i \Rightarrow Z_i)$.

Let $B' = \bigwedge_{i \in I} \forall \vec{Y_i}.(A_i \Rightarrow Z_i)$ and $C' = \bigwedge_{i \in \bar{I}} \forall \vec{Y_i}.(A_i \Rightarrow Z_i)$. So, $B \equiv \forall X.B'$ and $C \equiv \forall X.C'$. Hence, also by Corollary 4.3, we have $A \equiv \bigwedge_{i=1}^n \forall \vec{Y_i}.(A_i \Rightarrow Z_i) \equiv B' \wedge C'$. □

Lemma 4.10. *If $\forall X.A \equiv B \Rightarrow C$, then $C \equiv \forall X.C'$ and $A \equiv B \Rightarrow C'$.*

Proof. By Lemma 4.5, $PF(\forall X.A) \sim PF(B \Rightarrow C)$. By Lemma 4.2, let

$$PF(A) = [\forall \vec{Y_i}.(A_i \Rightarrow Z_i)]_{i=1}^n$$
$$PF(C) = [\forall \vec{W_j}.(D_j \Rightarrow Z'_j)]_{j=1}^m$$

Hence, $[\forall X.\forall \vec{Y_i}.(A_i \Rightarrow Z_i)]_{i=1}^n \sim [\forall \vec{W_j}.((B \wedge D_j) \Rightarrow Z'_j)]_{j=1}^m$. So, by definition of $\sim$, $n = m$ and for $i = 1, \dots, n$ and a permutation p, we have

$$\forall X.\forall \vec{Y_i}.(A_i \Rightarrow Z_i) \equiv \forall \vec{W}_{p(i)}.((B \wedge D_{p(i)}) \Rightarrow Z'_{p(i)})$$

Hence, by Lemma 4.7, we have $X, \vec{Y_i} = \vec{W}_{p(i)}$, $A_i \equiv B \wedge D_{p(i)}$, and $Z_i = Z'_{p(i)}$. Hence, by Corollary 4.3, $C \equiv \bigwedge_{j=1}^n \forall \vec{W_j}.(D_j \Rightarrow Z'_j) \equiv \bigwedge_{i=1}^n \forall \vec{W}_{p(i)}.(D_{p(i)} \Rightarrow Z'_{p(i)}) \equiv \bigwedge_{i=1}^n \forall X.\forall \vec{Y_i}.(D_{p(i)} \Rightarrow Z_i)$.

Let $C' = \bigwedge_{i=1}^n \forall \vec{Y_i}.(D_{p(i)} \Rightarrow Z_i)$. So, $C \equiv \forall X.C'$.

Hence, also by Corollary 4.3, we have $A \equiv \bigwedge_{i=1}^n \forall \vec{Y_i}.(A_i \Rightarrow Z_i) \equiv \bigwedge_{i=1}^n \forall \vec{Y_i}.((B \wedge D_{p(i)}) \Rightarrow Z_i) \equiv B \Rightarrow \bigwedge_{i=1}^n \forall \vec{Y_i}.(D_{p(i)} \Rightarrow Z_i) \equiv B \Rightarrow C'$. □

Lemma 4.11 (Unicity modulo). *If $\Gamma \vdash r : A$ and $\Gamma \vdash r : B$, then $A \equiv B$.*

Proof.

- If the last rule of the derivation of $\Gamma \vdash r : A$ is $(\equiv)$, then we have a shorter derivation of $\Gamma \vdash r : C$ with $C \equiv A$, and, by the induction hypothesis, $C \equiv B$, hence $A \equiv B$.
- If the last rule of the derivation of $\Gamma \vdash r : B$ is $(\equiv)$ we proceed in the same way.
- All the remaining cases are syntax directed. □

Lemma 4.13 (Substitution).

1. *If $\Gamma, x : B \vdash r : A$ and $\Gamma \vdash s : B$ then $\Gamma \vdash [x := s]r : A$.*
2. *If $\Gamma \vdash r : A$, then $[X := B]\Gamma \vdash [X := B]r : [X := B]A$.* □

Proof.

1. By structural induction on r.
   - Let $r = x$. By Lemma 4.12, $A \equiv B$, thus $\Gamma \vdash s : A$. Since $[x := s]x = s$, we have $\Gamma \vdash [x := s]x : A$.
   - Let $r = y$, with $y \neq x$. Since $[x := s]y = y$, we have $\Gamma \vdash [x := s]y : A$.
   - Let $r = \lambda x^C.t$. We have $[x := s](\lambda x^C.t) = \lambda x^C.t$, so $\Gamma \vdash [x := s](\lambda x^C.t) : A$.
   - Let $r = \lambda y^C.t$, with $y \neq x$. By Lemma 4.12, $A \equiv C \Rightarrow D$ and $\Gamma, y : C \vdash t : D$. By the induction hypothesis, $\Gamma, y : C \vdash [x := s]t : D$, and so, by rule $(\Rightarrow_i)$, $\Gamma \vdash \lambda y^C.[x := s]t : C \Rightarrow D$. Since $\lambda y^C.[x := s]t = [x := s](\lambda y^C.t)$, using rule $(\equiv)$, $\Gamma \vdash [x := s](\lambda x^C.t) : A$.
   - Let $r = tu$. By Lemma 4.12, $\Gamma \vdash t : C \Rightarrow A$ and $\Gamma \vdash u : C$. By the induction hypothesis, $\Gamma \vdash [x := s]t : C \Rightarrow A$ and $\Gamma \vdash [x := s]u : C$, and so, by rule $(\Rightarrow_e)$, $\Gamma \vdash ([x := s]t)([x := s]u) : A$. Since $([x := s]t)([x := s]u) = [x := s](tu)$, we have $\Gamma \vdash [x := s](tu) : A$.
   - Let $r = \langle t, u \rangle$. By Lemma 4.12, $\Gamma \vdash t : C$ and $\Gamma \vdash u : D$, with $A \equiv C \wedge D$. By the induction hypothesis, $\Gamma \vdash [x := s]t : C$ and $\Gamma \vdash [x := s]u : D$, and so, by rule $(\wedge_i)$, $\Gamma \vdash \langle [x := s]t, [x := s]u \rangle : C \wedge D$. Since $\langle [x := s]t, [x := s]u \rangle = [x := s]\langle t, u \rangle$, using rule $(\equiv)$, we have $\Gamma \vdash [x := s]\langle t, u \rangle : A$.
   - Let $r = \pi_A t$. By Lemma 4.12, $\Gamma \vdash t : A \wedge C$. By the induction hypothesis, $\Gamma \vdash [x := s]t : A \wedge C$, and so, by rule $(\wedge_e)$, $\Gamma \vdash \pi_A([x := s]t) : A$. Since $\pi_A([x := s]t) = [x := s](\pi_A t)$, we have $\Gamma \vdash [x := s](\pi_A t) : A$.
   - Let $r = \Lambda X.t$. By Lemma 4.12, $A \equiv \forall X.C$ and $\Gamma \vdash t : C$. By the induction hypothesis, $\Gamma \vdash [x := s]t : C$, and so, by rule $(\forall_i)$, $\Gamma \vdash \Lambda X.[x := s]t : \forall X.C$. Since $\Lambda X.[x := s]t = [x := s](\Lambda X.t)$, using rule $(\equiv)$, we have $\Gamma \vdash [x := s](\Lambda X.t) : A$.
   - Let $r = t[C]$. By Lemma 4.12, $A \equiv [X := C]D$ and $\Gamma \vdash t : \forall X.D$. By the induction hypothesis, $\Gamma \vdash [x := s]t : \forall X.D$, and so, by rule $(\forall_e)$, $\Gamma \vdash ([x := s]t)[C] : [X := C]D$. Since $([x := s]t)[C] = [x := s](t[C])$, using rule $(\equiv)$, we have $\Gamma \vdash [x := s](t[C]) : A$.
2. By induction on the typing relation.
   - (ax): Let $\Gamma, x : A \vdash x : A$. Then, using rule (ax), we have $[X := B]\Gamma, x : [X := B]A \vdash [X := B]x : [X := B]A$.
   - $(\equiv)$: Let $\Gamma \vdash r : A$, with $A \equiv C$. By the induction hypothesis, $[X := B]\Gamma \vdash [X := B]r : [X := B]C$. Since $A \equiv C$, $[X := B]A \equiv [X := B]C$. Using rule $(\equiv)$, we have $[X := B]\Gamma \vdash [X := B]r : [X := B]A$.
   - $(\Rightarrow_i)$: Let $\Gamma \vdash \lambda x^C.t : C \Rightarrow D$. By the induction hypothesis, $[X := B]\Gamma, x : [X := B]C \vdash [X := B]t : [X := B]D$. Using rule $(\Rightarrow_i)$, $[X := B]\Gamma \vdash \lambda x^{[X:=B]C}.[X := B]t : [X := B]C \Rightarrow [X := B]D$. Since $\lambda x^{[X:=B]C}.[X := B]t = [X := B](\lambda x^C.t)$, we have $[X := B]\Gamma \vdash [X := B](\lambda x^C.t) : [X := B](C \Rightarrow D)$.

- ($\Rightarrow_e$): Let $\Gamma \vdash ts : D$. By the induction hypothesis, $[X := B]\Gamma \vdash [X := B]t : [X := B](C \Rightarrow D)$ and $[X := B]\Gamma \vdash [X := B]s : [X := B]C$. Since $[X := B](C \Rightarrow D) = [X := B]C \Rightarrow [X := B]D$, using rule ($\Rightarrow_e$), we have $[X := B]\Gamma \vdash ([X := B]t)([X := B]s) : [X := B]D$. Since $([X := B]t)([X := B]s) = [X := B](ts)$, we have $[X := B]\Gamma \vdash [X := B](ts) : [X := B]D$.
- ($\wedge_i$): Let $\Gamma \vdash \langle t, s\rangle : C \wedge D$. By the induction hypothesis, $[X := B]\Gamma \vdash [X := B]t : [X := B]C$ and $[X := B]\Gamma \vdash [X := B]s : [X := B]D$. Using rule ($\wedge_i$), $[X := B]\Gamma \vdash \langle [X := B]t, [X := B]s\rangle : [X := B]C \wedge [X := B]D$. Since $\langle [X := B]t, [X := B]s\rangle = [X := B]\langle t, s\rangle$, and $[X := B]C \wedge [X := B]D = [X := B](C \wedge D)$, we have $[X := B]\Gamma \vdash [X := B]\langle t, s\rangle : [X := B](C \wedge D)$.
- ($\wedge_e$): Let $\Gamma \vdash t : C \wedge D$. By the induction hypothesis, $[X := B]\Gamma \vdash [X := B]t : [X := B](C \wedge D)$. Since $[X := B](C \wedge D) = [X := B](C) \wedge [X := B](D)$, using rule ($\wedge_e$)we have $[X := B]\Gamma \vdash \pi_{[X:=B]C}([X := B]t) : [X := B](C)$. Since $\pi_{[X:=B]C}[X := B]t = [X := B]\pi_C t$, we have $[X := B]\Gamma \vdash [X := B]\pi_C t : [X := B](C)$.
- ($\forall_i$): Let $\Gamma \vdash \Lambda Y.t : \forall Y.C$, with $X \notin FTV(\Gamma)$. By the induction hypothesis, $[X := B]\Gamma \vdash [X := B]t : [X := B]C$. Since $X \notin FTV(\Gamma)$, $X \notin FV([X := B]\Gamma)$. Using rule ($\forall_i$), we have $[X := B]\Gamma \vdash \Lambda Y.[X := B]t : \Lambda Y.[X := B]C$. Since $\Lambda Y.[X := B]t = [X := B]\Lambda Y.t$, and $\forall Y.[X := B]C = [X := B]\forall Y.C$, we have $[X := B]\Gamma \vdash [X := B]\Lambda Y.t : [X := B]\forall Y.C$.
- ($\forall_e$): Let $\Gamma \vdash t[D] : [Y := D]C$. By the induction hypothesis, $[X := B]\Gamma \vdash [X := B]t : [X := B]\forall Y.C$. Since $[X := B]\forall Y.C = \forall Y.[X := B]C$, using rule ($\forall_e$), we have $[X := B]\Gamma \vdash ([X := B]t)[[X := B]D] : [Y := [X := B]D][X := B]C$.
  Since $([X := B]t)[[X := B]D] = [X := B](t[D])$, and $[Y := [X := B]D][X := B]C = [X := B][Y := D]C$, we have $[X := B]\Gamma \vdash [X := B](t[D]) : [X := B][Y := D]C$. □

Theorem 4.14 (Subject reduction). *If $\Gamma \vdash r : A$ and $r \hookrightarrow s$ or $r \rightleftarrows s$, then $\Gamma \vdash s : A$.*

Proof. By induction on the rewrite relation.

- (COMM): $\langle t, r\rangle \rightleftarrows \langle r, t\rangle$

$(\rightharpoonup)$(1) $\Gamma \vdash \langle t, r\rangle : A$ (Hypothesis)
(2) $A \equiv B \wedge C$
$\Gamma \vdash t : B$
$\Gamma \vdash r : C$ (1, Lemma 4.12)
(3) $B \wedge C \equiv C \wedge B$ (Iso. (1))
(4)

$$[2]\ \dfrac{[3]\ \dfrac{\dfrac{\Gamma \vdash r : C \qquad \Gamma \vdash t : B}{\Gamma \vdash \langle r, t\rangle : C \wedge B}\,(\wedge_i)}{\Gamma \vdash \langle r, t\rangle : B \wedge C}\,(\equiv)}{\Gamma \vdash \langle r, t\rangle : A}\,(\equiv)$$

$(\leftharpoondown)$ analogous to $(\rightharpoonup)$.

- (ASSO): $\langle t, \langle r, s\rangle\rangle \rightleftarrows \langle\langle t, r\rangle, s\rangle$

$(\rightharpoonup)$(1) $\Gamma \vdash \langle t, \langle r, s\rangle\rangle : A$ (Hypothesis)
(2) $A \equiv B \wedge C$
$\Gamma \vdash t : B$
$\Gamma \vdash \langle r, s\rangle : C$ (1, Lemma 4.12)
(3) $C \equiv D \wedge E$
$\Gamma \vdash r : D$
$\Gamma \vdash s : E$ (2, Lemma 4.12)
(4) $B \wedge (D \wedge E) \equiv (B \wedge D) \wedge E$ (Iso. (2))
(5) $A \equiv B \wedge (D \wedge E)$ (2, 3, congr. ($\equiv$))
(6)

$$[5]\ \dfrac{[4]\ \dfrac{\dfrac{\dfrac{\Gamma \vdash t : B \qquad \Gamma \vdash r : D}{\Gamma \vdash \langle t, r\rangle : B \wedge D}\,(\wedge_i) \qquad \Gamma \vdash s : E}{\Gamma \vdash \langle\langle t, r\rangle, s\rangle : (B \wedge D) \wedge E}\,(\wedge_i)}{\Gamma \vdash \langle\langle t, r\rangle, s\rangle : B \wedge (D \wedge E)}\,(\equiv)}{\Gamma \vdash \langle\langle t, r\rangle, s\rangle : A}\,(\equiv)$$

$(\leftharpoondown)$ analogous to $(\rightharpoonup)$.

- (DIST$_\lambda$): $\lambda x^A.\langle t, r\rangle \rightleftarrows \langle \lambda x^A.t, \lambda x^A.r\rangle$

$(\rightharpoonup)$(1) $\Gamma \vdash \lambda x^A.\langle t, r\rangle : B$ (Hypothesis)
(2) $B \equiv A \Rightarrow C$
$\Gamma, x : A \vdash \langle t, r\rangle : C$ (1, Lemma 4.12)
(3) $C \equiv D \wedge E$
$\Gamma, x : A \vdash t : D$
$\Gamma, x : A \vdash r : E$ (2, Lemma 4.12)
(4) $A \Rightarrow (D \wedge E) \equiv (A \Rightarrow D) \wedge (A \Rightarrow E)$ (Iso. (3))
(5) $B \equiv A \Rightarrow (D \wedge E)$ (2, 3, congr. ($\equiv$))
(6)

$$[5]\ \dfrac{[4]\ \dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash t : D}{\Gamma \vdash \lambda x^A.t : A \Rightarrow D}\,(\Rightarrow_i) \qquad \dfrac{\Gamma, x : A \vdash r : E}{\Gamma \vdash \lambda x^A.r : A \Rightarrow E}\,(\Rightarrow_i)}{\Gamma \vdash \langle \lambda x^A.t, \lambda x^A.r\rangle : (A \Rightarrow D) \wedge (A \Rightarrow E)}\,(\wedge_i)}{\Gamma \vdash \langle \lambda x^A.t, \lambda x^A.r\rangle : A \Rightarrow (D \wedge E)}\,(\equiv)}{\Gamma \vdash \langle \lambda x^A.t, \lambda x^A.r\rangle : B}\,(\equiv)$$

$(\leftharpoondown)$(1) $\Gamma \vdash \langle \lambda x^A.t, \lambda x^A.r\rangle : B$ (Hypothesis)
(2) $B \equiv C \wedge D$
$\Gamma \vdash \lambda x^A.t : C$
$\Gamma \vdash \lambda x^A.r : D$ (1, Lemma 4.12)
(3) $C \equiv A \Rightarrow C'$
$\Gamma, x : A \vdash t : C'$ (2, Lemma 4.12)
(4) $D \equiv A \Rightarrow D'$
$\Gamma, x : A \vdash r : D'$ (2, Lemma 4.12)
(5) $(A \Rightarrow C') \wedge (A \Rightarrow D') \equiv A \Rightarrow (C' \wedge D')$ (Iso. (3))
(6) $B \equiv (A \Rightarrow C') \wedge (A \Rightarrow D')$ (2, 3, 4, congr. ($\equiv$))
(7)

$$[6]\ \dfrac{[5]\ \dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash t : C' \qquad \Gamma, x : A \vdash r : D'}{\Gamma, x : A \vdash \langle t, r\rangle : C' \wedge D'}\,(\wedge_i)}{\Gamma \vdash \lambda x^A.\langle t, r\rangle : A \Rightarrow (C' \wedge D')}\,(\Rightarrow_i)}{\Gamma \vdash \lambda x^A.\langle t, r\rangle : (A \Rightarrow C') \wedge (A \Rightarrow D')}\,(\equiv)}{\Gamma \vdash \lambda x^A.\langle t, r\rangle : B}\,(\equiv)$$

- (DIST$_{\text{app}}$): $\langle t, r\rangle s \rightleftarrows \langle ts, rs\rangle$

$(\rightharpoonup)$(1) $\Gamma \vdash \langle t, r\rangle s : A$ (Hypothesis)
(2) $\Gamma \vdash \langle t, r\rangle : B \Rightarrow A$
$\Gamma \vdash s : B$ (1, Lemma 4.12)
(3) $B \Rightarrow A \equiv C \wedge D$
$\Gamma \vdash t : C$
$\Gamma \vdash r : D$ (2, Lemma 4.12)

(4) $C \equiv B \Rightarrow C'$
$D \equiv B \Rightarrow D'$
$A \equiv C' \wedge D'$ (3, Lemma 4.8)

(5)
$$\dfrac{[4]\ \dfrac{\Gamma \vdash t : C}{\Gamma \vdash t : B \Rightarrow C'}\,(\equiv) \qquad \Gamma \vdash s : B}{\Gamma \vdash ts : C'}\,(\Rightarrow_e)$$

(6)
$$\dfrac{[4]\ \dfrac{\Gamma \vdash r : D}{\Gamma \vdash r : B \Rightarrow D'}\,(\equiv) \qquad \Gamma \vdash s : B}{\Gamma \vdash rs : D'}\,(\Rightarrow_e)$$

(7)
$$[4]\ \dfrac{\dfrac{\dfrac{(5)}{\Gamma \vdash ts : C'} \qquad \dfrac{(6)}{\Gamma \vdash rs : D'}}{\Gamma \vdash \langle ts, rs\rangle : C' \wedge D'}\,(\wedge_i)}{\Gamma \vdash \langle ts, rs\rangle : A}\,(\equiv)$$

$(\leftarrow)$(1) $\Gamma \vdash \langle ts, rs\rangle : A$ (Hypothesis)
(2) $A \equiv B \wedge C$
$\Gamma \vdash ts : B$
$\Gamma \vdash rs : C$ (1, Lemma 4.12)
(3) $\Gamma \vdash t : D \Rightarrow B$
$\Gamma \vdash s : D$ (2, Lemma 4.12)
(4) $\Gamma \vdash r : E \Rightarrow B$
$\Gamma \vdash s : E$ (2, Lemma 4.12)
(5) $D \equiv E$ (3, 4, Lemma 4.11)
(6) $D \Rightarrow (B \wedge C) \equiv (D \Rightarrow B) \wedge (D \Rightarrow C)$ (Iso. (3))
(7) $E \Rightarrow C \equiv D \Rightarrow C$ (6, congr. $(\equiv)$)
(8)
$$[2]\ \dfrac{\dfrac{[5]\ \dfrac{\dfrac{\Gamma \vdash t : D \Rightarrow B \qquad [7]\ \dfrac{\Gamma \vdash r : E \Rightarrow C}{\Gamma \vdash r : D \Rightarrow C}\,(\equiv)}{\Gamma \vdash \langle t, r\rangle : (D \Rightarrow B) \wedge (D \Rightarrow C)}\,(\wedge_i)}{\Gamma \vdash \langle t, r\rangle : D \Rightarrow (B \wedge C)}\,(\equiv)}{\Gamma \vdash \langle t, r\rangle s : B \wedge C}\,(\Rightarrow_e)}{\Gamma \vdash \langle t, r\rangle s : A}\,(\equiv)$$

- (CURRY): $t\langle r, s\rangle \rightleftarrows trs$

$(\rightarrow)$(1) $\Gamma \vdash t\langle r, s\rangle : A$ (Hypothesis)
(2) $\Gamma \vdash t : B \Rightarrow A$
$\Gamma \vdash \langle t, r\rangle : B$ (1, Lemma 4.12)
(3) $B \equiv C \wedge D$
$\Gamma \vdash r : C$
$\Gamma \vdash s : D$ (2, Lemma 4.12)
(4) $B \Rightarrow A \equiv (C \wedge D) \Rightarrow A$ (3, congr. $(\equiv)$)
(5) $(C \wedge D) \Rightarrow A \equiv C \Rightarrow (D \Rightarrow A)$ (Iso. (4))
(6)
$$\dfrac{[5]\ \dfrac{[4]\ \dfrac{\Gamma \vdash t : B \Rightarrow A}{\Gamma \vdash t : (C \wedge D) \Rightarrow A}\,(\equiv)}{\Gamma \vdash t : C \Rightarrow (D \Rightarrow A)}\,(\equiv) \qquad \Gamma \vdash r : C}{\Gamma \vdash tr : D \Rightarrow A}\,(\Rightarrow_e)$$

(7)
$$\dfrac{\dfrac{(6)}{\Gamma \vdash tr : D \Rightarrow A} \qquad \Gamma \vdash s : D}{\Gamma \vdash trs : A}\,(\Rightarrow_e)$$

$(\leftarrow)$(1) $\Gamma \vdash trs : A$ (Hypothesis)
(2) $\Gamma \vdash tr : B \Rightarrow A$
$\Gamma \vdash s : B$ (1, Lemma 4.12)
(3) $\Gamma \vdash t : C \Rightarrow (B \Rightarrow A)$
$\Gamma \vdash r : C$ (2, Lemma 4.12)
(4) $C \Rightarrow (B \Rightarrow A) \equiv (C \wedge B) \Rightarrow A$ (Iso. (4))
(5)
$$\dfrac{[4]\ \dfrac{\Gamma \vdash t : C \Rightarrow (B \Rightarrow A)}{\Gamma \vdash t : (C \wedge B) \Rightarrow A}\,(\equiv) \qquad \dfrac{\Gamma \vdash r : C \qquad \Gamma \vdash s : B}{\Gamma \vdash \langle r, s\rangle : C \wedge B}\,(\wedge_i)}{\Gamma \vdash t\langle r, s\rangle : A}\,(\Rightarrow_e)$$

- (P-COMM$_{\forall_i \Rightarrow_i}$): $\Lambda X.\lambda x^A.t \rightleftarrows \lambda x^A.\Lambda X.t$

$(\rightarrow)$(1) $X \notin FTV(A)$ (Hypothesis)
(2) $\Gamma \vdash \Lambda X.\lambda x^A.t : B$ (Hypothesis)
(3) $B \equiv \forall X.C$
$\Gamma \vdash \lambda x^A.t : C$
$X \notin FTV(\Gamma)$ (2, Lemma 4.12)
(4) $C \equiv A \Rightarrow D$
$\Gamma, x : A \vdash t : D$ (3, Lemma 4.12)
(5) $\forall X.(A \Rightarrow D) \equiv A \Rightarrow \forall X.D$ (1, Iso. (5))
(6) $\forall X.C \equiv \forall X.(A \Rightarrow D)$ (4, congr. $(\equiv)$)
(7)
$$[3]\ \dfrac{[6]\ \dfrac{[5]\ \dfrac{\dfrac{[1\ 3]\ \dfrac{\Gamma, x : A \vdash t : D}{\Gamma, x : A \vdash \Lambda X.t : \forall X.D}\,(\forall_i)}{\Gamma \vdash \lambda x^A.\Lambda X.t : A \Rightarrow \forall X.D}\,(\Rightarrow_i)}{\Gamma \vdash \lambda x^A.\Lambda X.t : \forall X.(A \Rightarrow D)}\,(\equiv)}{\Gamma \vdash \lambda x^A.\Lambda X.t : \forall X.C}\,(\equiv)}{\Gamma \vdash \lambda x^A.\Lambda X.t : B}\,(\equiv)$$

$(\leftarrow)$(1) $X \notin FTV(A)$ (Hypothesis)
(2) $\Gamma \vdash \lambda x^A.\Lambda X.t : B$ (Hypothesis)
(3) $B \equiv A \Rightarrow C$
$\Gamma, x : A \vdash \Lambda X.t : C$ (2, Lemma 4.12)
(4) $C \equiv \forall X.D$
$\Gamma, x : A \vdash t : D$
$X \notin FTV(\Gamma) \cup FTV(A)$ (3, Lemma 4.12)
(5) $\forall X.(A \Rightarrow D) \equiv A \Rightarrow \forall X.D$ (1, Iso. (5))
(6) $A \Rightarrow C \equiv A \Rightarrow \forall X.D$ (4, congr. $(\equiv)$)
(7)
$$[3]\ \dfrac{[6]\ \dfrac{[5]\ \dfrac{[4]\ \dfrac{\dfrac{\Gamma, x : A \vdash t : D}{\Gamma \vdash \lambda x^A.t : A \Rightarrow D}\,(\Rightarrow_i)}{\Gamma \vdash \Lambda X.\lambda x^A.t : \forall X.(A \Rightarrow D)}\,(\forall_i)}{\Gamma \vdash \Lambda X.\lambda x^A.t : A \Rightarrow \forall X.D}\,(\equiv)}{\Gamma \vdash \Lambda X.\lambda x^A.t : A \Rightarrow C}\,(\equiv)}{\Gamma \vdash \Lambda X.\lambda x^A.t : B}\,(\equiv)$$

- (P-COMM$_{\forall_e \Rightarrow_i}$): $(\lambda x^A.t)[B] \rightleftarrows \lambda x^A.t[B]$

$(\rightarrow)$(1) $X \notin FTV(A)$ (Hypothesis)
(2) $\Gamma \vdash (\lambda x^A.t)[B] : C$ (Hypothesis)
(3) $C \equiv [X := B]D$
$\Gamma \vdash \lambda x^A.t : \forall X.D$ (2, Lemma 4.12)
(4) $\forall X.D \equiv A \Rightarrow E$
$\Gamma, x : A \vdash t : E$ (3, Lemma 4.12)
(5) $E \equiv \forall X.E'$
$D \equiv A \Rightarrow E'$ (4, Lemma 4.10)
(6) $A \Rightarrow [X := B]E' = [X := B](A \Rightarrow E')$ (1, Def.)
(7) $[X := B](A \Rightarrow E') \equiv [X := B]D$ (5, congr. $(\equiv)$)
(8)

$$[3]\ \dfrac{[7]\ \dfrac{[6]\ \dfrac{\dfrac{\dfrac{[5]\ \dfrac{\Gamma, x:A\vdash t:E}{\Gamma, x:A\vdash t:\forall X.E'}\ (\equiv)}{\Gamma, x:A\vdash t[B]:[X:=B]E'}\ (\forall_e)}{\Gamma\vdash\lambda x^A.t[B]:A\Rightarrow[X:=B]E'}\ (\Rightarrow_i)}{\Gamma\vdash\lambda x^A.t[B]:[X:=B](A\Rightarrow E')}\ (\equiv)}{\Gamma\vdash\lambda x^A.t[B]:[X:=B]D}\ (\equiv)}{\Gamma\vdash\lambda x^A.t[B]:C}\ (\equiv)$$

$(\leftharpoondown)$(1) $X\notin FTV(A)$ (Hypothesis)
(2) $\Gamma\vdash\lambda x^A.t[B]:C$ (Hypothesis)
(3) $C\equiv A\Rightarrow D$
$\Gamma, x:A\vdash t[B]:D$ (1, Lemma 4.12)
(4) $D\equiv[X:=B]E$
$\Gamma, x:A\vdash t:\forall X.E$ (2, Lemma 4.12)
(5) $A\Rightarrow\forall X.E\equiv\forall X.(A\Rightarrow E)$ (Iso. (5))
(6) $[X:=B](A\Rightarrow E)=A\Rightarrow[X:=B]E$ (1, Def.)
(7) $A\Rightarrow[X:=B]E\equiv A\Rightarrow D$ (4, congr. $(\equiv)$)
(8)

$$[3]\ \dfrac{[7]\ \dfrac{[6]\ \dfrac{\dfrac{[5]\ \dfrac{\dfrac{\Gamma, x:A\vdash t:\forall X.E}{\Gamma\vdash\lambda x^A.t:A\Rightarrow\forall X.E}\ (\Rightarrow_i)}{\Gamma\vdash\lambda x^A.t:\forall X.(A\Rightarrow E)}\ (\equiv)}{\Gamma\vdash(\lambda x^A.t)[B]:[X:=B](A\Rightarrow E)}\ (\forall_e)}{\Gamma\vdash(\lambda x^A.t)[B]:A\Rightarrow[X:=B]E}\ (\equiv)}{\Gamma\vdash(\lambda x^A.t)[B]:A\Rightarrow D}\ (\equiv)}{\Gamma\vdash(\lambda x^A.t)[B]:C}\ (\equiv)$$

- (P-DIST$_{\forall_i\wedge_i}$): $\Lambda X.\langle t,r\rangle\rightleftarrows\langle\Lambda X.t,\Lambda X.r\rangle$

$(\rightharpoonup)$(1) $\Gamma\vdash\Lambda X.\langle t,r\rangle:A$ (Hypothesis)
(2) $A\equiv\forall X.B$
$\Gamma\vdash\langle t,r\rangle:B$
$X\notin FTV(\Gamma)$ (1, Lemma 4.12)
(3) $B\equiv C\wedge D$
$\Gamma\vdash t:C$
$\Gamma\vdash r:D$ (2, Lemma 4.12)
(4) $\forall X.(C\wedge D)\equiv\forall X.C\wedge\forall X.D$ (Iso. (6))
(5) $\forall X.B\equiv\forall X.(C\wedge D)$ (3, congr. $(\equiv)$)
(6)

$$[2]\ \dfrac{[5]\ \dfrac{[4]\ \dfrac{\dfrac{[2]\ \dfrac{\Gamma\vdash t:C}{\Gamma\vdash\Lambda X.t:\forall X.C}\ (\forall_i)\qquad[2]\ \dfrac{\Gamma\vdash r:D}{\Gamma\vdash\Lambda X.r:\forall X.D}\ (\forall_i)}{\Gamma\vdash\langle\Lambda X.t,\Lambda X.r\rangle:\forall X.C\wedge\forall X.D}\ (\wedge_i)}{\Gamma\vdash\langle\Lambda X.t,\Lambda X.r\rangle:\forall X.(C\wedge D)}\ (\equiv)}{\Gamma\vdash\langle\Lambda X.t,\Lambda X.r\rangle:\forall X.B}\ (\equiv)}{\Gamma\vdash\langle\Lambda X.t,\Lambda X.r\rangle:A}\ (\equiv)$$

$(\leftharpoondown)$(1) $\Gamma\vdash\langle\Lambda X.t,\Lambda X.r\rangle:A$ (Hypothesis)
(2) $A\equiv B\wedge C$
$\Gamma\vdash\Lambda X.t:B$
$\Gamma\vdash\Lambda X.r:C$ (1, Lemma 4.12)
(3) $B\equiv\forall X.D$
$\Gamma\vdash t:D$
$X\notin FTV(\Gamma)$ (2, Lemma 4.12)
(4) $C\equiv\forall X.E$
$\Gamma\vdash r:E$
$X\notin FTV(\Gamma)$ (2, Lemma 4.12)
(5) $\forall X.(D\wedge E)\equiv\forall X.D\wedge\forall X.E$ (Iso. (6))
(6) $\forall X.D\wedge\forall X.E\equiv B\wedge C$ (3, 4, congr. $(\equiv)$)
(7)

$$[2]\ \dfrac{[6]\ \dfrac{[5]\ \dfrac{[3]\ \dfrac{\dfrac{\Gamma\vdash t:D\qquad\Gamma\vdash r:E}{\Gamma\vdash\langle t,r\rangle:D\wedge E}\ (\wedge_i)}{\Gamma\vdash\Lambda X.\langle t,r\rangle:\forall X.(D\wedge E)}\ (\forall_i)}{\Gamma\vdash\Lambda X.\langle t,r\rangle:\forall X.D\wedge\forall X.E}\ (\equiv)}{\Gamma\vdash\Lambda X.\langle t,r\rangle:B\wedge C}\ (\equiv)}{\Gamma\vdash\Lambda X.\langle t,r\rangle:A}\ (\equiv)$$

- (P-DIST$_{\forall_e\wedge_i}$): $\langle t,r\rangle[B]\rightleftarrows\langle t[B],r[B]\rangle$

$(\rightharpoonup)$(1) $\Gamma\vdash\langle t,r\rangle[B]:A$ (Hypothesis)
(2) $A\equiv[X:=B]C$
$\Gamma\vdash\langle t,r\rangle:\forall X.C$ (1, Lemma 4.12)
(3) $\forall X.C\equiv D\wedge E$
$\Gamma\vdash t:D$
$\Gamma\vdash r:E$ (2, Lemma 4.12)
(4) $D\equiv\forall X.D'$
$E\equiv\forall X.E'$
$C\equiv D'\wedge E'$ (3, Lemma 4.9)
(5) $[X:=B](D'\wedge E')=[X:=B]D'\wedge[X:=B]E'$ (Def.)
(6) $[X:=B]C\equiv[X:=B](D'\wedge E')$ (4, congr. $(\equiv)$)
(7)

$$[2]\ \dfrac{[6]\ \dfrac{[5]\ \dfrac{\dfrac{[4]\ \dfrac{\Gamma\vdash t:D}{\Gamma\vdash t:\forall X.D'}\ (\equiv)}{\Gamma\vdash t[B]:[X:=B]D'}\ (\forall_e)\qquad\dfrac{[4]\ \dfrac{\Gamma\vdash r:E}{\Gamma\vdash r:\forall X.E'}\ (\equiv)}{\Gamma\vdash r[B]:[X:=B]E'}\ (\forall_e)}{\Gamma\vdash\langle t[B],r[B]\rangle:[X:=B]D'\wedge[X:=B]E'}\ (\wedge_i)}{\Gamma\vdash\langle t[B],r[B]\rangle:[X:=B](D'\wedge E')}\ (\equiv)}{\Gamma\vdash\langle t[B],r[B]\rangle:[X:=B]C}\ (\equiv)}{\Gamma\vdash\langle t[B],r[B]\rangle:A}\ (\equiv)$$

$(\leftharpoondown)$(1) $\Gamma\vdash\langle t[B],r[B]\rangle:A$ (Hypothesis)
(2) $A\equiv C\wedge D$
$\Gamma\vdash t[B]:C$
$\Gamma\vdash r[B]:D$ (1, Lemma 4.12)
(3) $C\equiv[X:=B]C'$
$\Gamma\vdash t:\forall X.C'$ (2, Lemma 4.12)
(4) $D\equiv[X:=B]D'$
$\Gamma\vdash r:\forall X.D'$ (2, Lemma 4.12)
(5) $\forall X.(C'\wedge D')\equiv\forall X.C'\wedge\forall X.D'$ (Iso. (6))
(6) $[X:=B](C'\wedge D')=[X:=B]C'\wedge[X:=B]D'$ (Def.)
(7) $[X:=B]C'\wedge[X:=B]D'\equiv C\wedge D$ (3, 4, congr. $(\equiv)$)
(8)

$$[2]\ \dfrac{[7]\ \dfrac{[6]\ \dfrac{\dfrac{[5]\ \dfrac{\dfrac{\Gamma\vdash t:\forall X.C'\qquad\Gamma\vdash r:\forall Y.D'}{\Gamma\vdash\langle t,r\rangle:\forall X.C'\wedge\forall X.D'}\ (\wedge_i)}{\Gamma\vdash\langle t,r\rangle:\forall X.(C'\wedge D')}\ (\equiv)}{\Gamma\vdash\langle t,r\rangle[B]:[X:=B](C'\wedge D')}\ (\forall_e)}{\Gamma\vdash\langle t,r\rangle[B]:[X:=B]C'\wedge[X:=B]D'}\ (\equiv)}{\Gamma\vdash\langle t,r\rangle[B]:C\wedge D}\ (\equiv)}{\Gamma\vdash\langle t,r\rangle[B]:A}\ (\equiv)$$

- (P-DIST$_{\forall_i\wedge_e}$): $\pi_{\forall X.B}(\Lambda X.t)\rightleftarrows\Lambda X.\pi_B t$

$(\rightharpoonup)$(1) $\Gamma\vdash\pi_{\forall X.B}(\Lambda X.t):A$ (Hypothesis)
(2) $A\equiv\forall X.B$
$\Gamma\vdash\Lambda X.t:(\forall X.B)\wedge C$ (1, Lemma 4.12)

(3) $(\forall X.B) \wedge C \equiv \forall X.D$

$\Gamma \vdash t : D$

$X \notin FTV(\Gamma)$ (2, Lemma 4.12)

(4) $C \equiv \forall X.C'$

$D \equiv B \wedge C'$ (3, Lemma 4.9)

(5)

$$[2]\,\dfrac{[3]\,\dfrac{\dfrac{[4]\,\dfrac{\Gamma \vdash t : D}{\Gamma \vdash t : B \wedge C'}\,(\equiv)}{\Gamma \vdash \pi_B t : B}\,(\wedge_e)}{\Gamma \vdash \Lambda X.\pi_B t : \forall X.B}\,(\forall_i)}{\Gamma \vdash \Lambda X.\pi_B t : A}\,(\equiv)$$

($\leftarrow$)(1) $\Gamma \vdash \Lambda X.\pi_B t : A$ (Hypothesis)

(2) $A \equiv \forall X.C$

$\Gamma \vdash \pi_B t : C$

$X \notin FTV(\Gamma)$ (1, Lemma 4.12)

(3) $B \equiv C$

$\Gamma \vdash t : C \wedge D$ (2, Lemma 4.12)

(4) $\forall X.(C \wedge D) \equiv \forall X.C \wedge \forall X.D$ (Iso. (6))

(5)

$$[2]\,\dfrac{\dfrac{[4]\,\dfrac{[2]\,\dfrac{\Gamma \vdash t : C \wedge D}{\Gamma \vdash \Lambda X.t : \forall X.(C \wedge D)}\,(\forall_i)}{\Gamma \vdash \Lambda X.t : \forall X.C \wedge \forall X.D}\,(\equiv)}{\Gamma \vdash \pi_{\forall X.B}(\Lambda X.t) : \forall X.C}\,(\wedge_e)}{\Gamma \vdash \pi_{\forall X.B}(\Lambda X.t) : A}\,(\equiv)$$

- ($\text{P-DIST}_{\wedge_e\forall_e}$): $(\pi_{\forall X.B} t)[C] \rightleftarrows \pi_{[X:=C]B}(t[C])$

($\rightarrow$)(1) $\Gamma \vdash t : \forall X.(B \wedge D)$ (Hypothesis)

(2) $\Gamma \vdash (\pi_{\forall X.B} t)[C] : A$ (Hypothesis)

(3) $A \equiv [X := C]E$

$\Gamma \vdash \pi_{\forall X.B} t : \forall X.E$ (2, Lemma 4.12)

(4) $\forall X.E \equiv \forall X.B$

$\Gamma \vdash t : \forall X.E \wedge F$ (3, Lemma 4.12)

(5) $E \equiv B$ (4)

(6) $[X := C](B \wedge D) = [X := C]B \wedge [X := C]D$ (Def.)

(7) $[X := C]B \equiv [X := C]E$ (5, congr. ($\equiv$))

(8)

$$[3]\,\dfrac{[7]\,\dfrac{\dfrac{[6]\,\dfrac{\dfrac{\Gamma \vdash t : \forall X.B \wedge D}{\Gamma \vdash t[C] : [X := C](B \wedge D)}\,(\forall_e)}{\Gamma \vdash t[C] : [X := C]B \wedge [X := C]D}\,(\equiv)}{\Gamma \vdash \pi_{[X:=C]B}(t[C]) : [X := C]B}\,(\wedge_e)}{\Gamma \vdash \pi_{[X:=C]B}(t[C]) : [X := C]E}\,(\equiv)}{\Gamma \vdash \pi_{[X:=C]B}(t[C]) : A}\,(\equiv)$$

($\leftarrow$)(1) $\Gamma \vdash t : \forall X.(B \wedge D)$ (Hypothesis)

(2) $\Gamma \vdash \pi_{[X:=C]B}(t[C]) : A$ (Hypothesis)

(3) $A \equiv [X := C]B$

$\Gamma \vdash t[C] : A \wedge E$ (2, Lemma 4.12)

(4) $\forall X.(B \wedge D) \equiv \forall X.B \wedge \forall X.D$ (Iso. (6))

(5)

$$[3]\,\dfrac{\dfrac{\dfrac{[4]\,\dfrac{\Gamma \vdash t : \forall X.(B \wedge D)}{\Gamma \vdash t : \forall X.B \wedge \forall X.D}\,(\equiv)}{\Gamma \vdash \pi_{\forall X.B} t : \forall X.B}\,(\wedge_e)}{\Gamma \vdash (\pi_{\forall X.B} t)[C] : [X := C]B}\,(\forall_e)}{\Gamma \vdash (\pi_{\forall X.B} t)[C] : A}\,(\equiv)$$

- (β_λ): If $\Gamma \vdash s : A$, $(\lambda x^A.r)s \hookrightarrow [x := s]r$

(1) $\Gamma \vdash s : A$ (Hypothesis)

(2) $\Gamma \vdash \lambda x^A.r : B$ (Hypothesis)

(3) $\Gamma \vdash \lambda x^A.r : A \Rightarrow B$ (2, Lemma 4.12)

(4) $A \Rightarrow B \equiv A \Rightarrow C$

$\Gamma, x : A \vdash r : C$ (3, Lemma 4.12)

(5) $B \equiv C$ (4, congr. ($\equiv$))

(6) $\Gamma \vdash [x := s]r : C$ (1, 4, Lemma 4.13)

(7) $\Gamma \vdash [x := s]r : B$ (5, 6, rule ($\equiv$))

- (β_Λ): $(\Lambda X.r)[A] \hookrightarrow [X := A]r$

(1) $\Gamma \vdash (\Lambda X.r)[A] : B$ (Hypothesis)

(2) $B \equiv [X := A]C$

$\Gamma \vdash \Lambda X.r : \forall X.C$ (1, Lemma 4.12)

(3) $\forall X.C \equiv \forall X.D$

$\Gamma \vdash r : D$

$X \notin FTV(\Gamma)$ (2, Lemma 4.12)

(4) $C \equiv D$ (3)

(5) $\Gamma \vdash r : C$ (4, rule ($\equiv$))

(6) $[X := A]\Gamma \vdash \Gamma \vdash [X := A]r : [X := A]C$ (5, Lemma 4.13)

(7) $\Gamma \vdash [X := A]r : B$ (2, 3, 7, rule ($\equiv$))

- (π): If $\Gamma \vdash r : A$, $\pi_A\langle r, s\rangle \hookrightarrow r$

(1) $\Gamma \vdash r : A$ (Hypothesis)

(2) $\pi_A\langle r, s\rangle : B$ (Hypothesis)

(3) $B \equiv A$

$\Gamma \vdash \langle r, s\rangle : A \wedge C$ (2, Lemma 4.12)

(4) $\Gamma \vdash \pi_A\langle r, s\rangle : A$ (2, 3, rule ($\equiv$))

□

Schema-driven mutation of datatype with multiple representations

Work-in-progress report

Anonymous Author(s)

Abstract

We attempt to make change gradual, and commute unnecessary updates in a functional language. To do this, instead of using state monads, we utilise semigroup right action instead. Finding that diffing is left inverse of mutation, we recover an alternative algebra of change that allows modifying the local state in a similar way as updating state distributed on multiple remote servers, or database relations.

1 Introduction

While the pure functional view of programs as transformations allow us to reach unprecedented robust systems, sometimes we miss the simplicity of record update in the presence of large schema. Even more, we would sometimes like to treat complex APIs as implementations of a data structure.

Functional languages have used van Laarhoven-style lens[9] and optics[5] to provide a more complex way of doing the same thing. However, there remain practical issues: (1) each lens-based update requires allocation of record nodes across the whole data structure. While this is acceptable for small changes, it is rather inefficient for amassed updates that touch significantly part of the data structure. (2) lens objects can be hardly used on derived representations of the same schema: we might want to make a mutable record to allow a fast update, where each substructure is represented by IORef a instead of a (3) We might make a database record where each reference is a foreign key of another structure like in beam: ForeignKey a (4) We might want to have a generic data structure that represents a change between two values for showing a changelog (5) We might want to compress multiple updates into a single update and execute it at once (6) Finally, we might want to use a schema to represent objects represented by remote API like in GraphQL, and push updates for it generically.

We can divide functional change management into subproblems:

compositional path where we want to assemble fragments of the path in order to indicate that a small update should be applied somewhere deep in the data structure. Lens and optics solve this one.

update consolidation where we have an algorithm that affects many little updates to the structure, and we want to make sure that the total cost of them does not break the complexity of the algorithm. Zippers solve this problem.

change virtualisation where we have an algorithm affecting change using one schema of the data structure, but want to change the representation to improve asymptotic complexity: in Haskell lens [9] solve this problem, while object-oriented languages like Python and Java use attribute getters and setters.

representation change problem, where we want to use the same change description to affect the change in different representations of the same schema: pure versus mutable data structure, or local memory data structure versus the cloud.

We argue that solving multiple problems from the above list will give us synergistic effects, and allow better programming. We attempt to solve all of this *schema-oriented programming* challenges, where data structure content is separated solving issues related to its representation and location. That is, we call for separation schema and other qualities of the datatype: (1) representation (2) location, local or remote (by allowing one to update remote datatype without the need to ever materialise it locally) (3) efficiency optimisations: update consolidation, data structure implementation, strictness or laziness or partial materialisation.

2 Solution

In this work, we plan our solution on higher kinded data families in order to accommodate multiple representations of the same schema with a single data type declaration which was done before in limited contexts [1–3, 10, 11].

However, we also go an extra mile to generically derive class instances[1], while allowing for overriding to customise data structures with special laws. Usually, we customise the treatment of data structure that have non-free terms, like mappings or sets.] derive change protocols for these types, which allows us to generalise lens [9] and optics [5] to handle all derived representations.

, ,
.

[1]For data structures corresponding to closed data terms without additional laws.

In all, our solution provides compatible treatment of all these requirements with simple and easy to understand interface. It also allows natural expansion to large schemas, and composing of both lenses, and commuting of **changes** on the massive data structures.

2.1 Example schema

We are using higher-kinded datatypes to allow for multiple representations based on the same schema [2, 3, 10, 11]. Let us consider a set of files. If the contents are in memory, we can use a different representation of the same schema than when we consider files stored in the cloud:

```
newtype FileSet f = FileSet {
    unFileSet :: f (Map.Map FilePath (f Content)) }

type FilesInMemory = FileSet Identity

type family ContentInfo a where
  ContentInfo Content = ContentHeader Identity
  ContentInfo a       = a

data ContentHeader f = ContentHeader {
    chETag      :: f ETag
  , chLLength   :: f Integer
  , chVersion   :: f ObjectVersionId
  , chExpires   :: f UTCTime }

type FilesInS3Bucket = FileSet ContentInfo
```

Our running example may be reading a set of files from the filesystem, then minification[2] of those files that are HTML or CSS, and then synchronising them to an AWS S3 bucket [4]. For efficiency, we would like to minify the files by making imperative updates on their contents:

```
type MutableFileSet = FileSet IORef
```

2.2 Change representation

2.3 Finding the change description

```
class Monoid (Diff  a)
   => Diffy         a where
   data family Diff a
   diff :: a -> a -> Diff a
```

The mempty of the Monoid corresponds to an empty diff:

$$diff\ a\ a \equiv mempty$$

In order to apply the change to different objects based on the same schema, we want to use a single description of change c. We also use a basic tool for describing differences between two snapshots of the same object: diff :: a -> a -> c. Then we apply this to our state, by running it in a monad: patch :: c -> m (). From the laws of both operations, we can infer that c is a semigroup right action on an object state hidden in the monad (as indicated by the categorical diagram).

[2]By removing unnecessary spaces and comments that do not change semantics.

```
class (Diffy    c a
      ,Monad  m)
  =>  Change m c a where
  settle :: c -> m ()
  see    ::      m a
```

2.4 Finding diff

There are many families of generic diff algorithms presented in literature [8], so we are satisfied that we may easily derive a simple case with Generic types.

We expand on this scheme, by *generic* diffing, where we override a default implementation to implement diff on a non-free data type that permits a better change representation:

```
data family Diff a
```

In the case of flat datatypes the implementation is straightforward:

```
instance Diffy String where
  type Diff String = String
  diff _ new = new
  patch new _ = Right new

instance Diffy Int where
  type Diff Int = Int
  diff _ new = new
  patch new _ = Right new
```

That means that we can use Haskell Generic to derive differences automatically for free datatypes. However, when implementing dictionaries, we can override the default and give a better change representation:

```
data Diff (Map.Map k v) =
  ByKey { added   :: Map.Map k v
        , deleted :: Set.Set k
        , updated :: Map.Map k (Diff v) }

newtype FileSetChange = FileSet Diff

instance (Diffy          v
        , Ord           k
        , Show          k  )
      => Diffy (Map.Map k v) where
  diff old new = ByKey {
      added   = new Map.\\ old
    , deleted = Set.fromList $ Map.keys $ old Map.\\ new
    , updated = Map.intersectionWith diff new old
    }
  patch  Same       v = Right v
  patch (Set v)     _ = Right v
```

$$\begin{tikzcd}
& \text{Snapshot}_1 \times \text{Snapshot}_2 \arrow[dl, "\pi_1"'] \arrow[d, "\text{diff}"] \arrow[dr, "\pi_2"] & \\
\text{Snapshot}_1 & \text{Action}_{1\to 2} & \text{Snapshot}_2 \\
\text{State}_1 \arrow[u, "\mathit{see}"] & \text{State}_1 \times \text{Action}_{1\to 2} \arrow[l, "\pi_1"'] \arrow[u, "\pi_2"] \arrow[r, "\text{act}"] & \text{State}_2 \arrow[u, "\text{see}"]
\end{tikzcd}$$

$$\begin{array}{lcll}
a \diamond (b \diamond c) & = & (a \diamond b) \diamond c & \text{(associativity of actions)} \\
\text{mempty} \diamond a & = & a & \text{(left identity of action)} \\
a \diamond \text{mempty} & = & a & \text{(right identity of action)} \\
\text{settle } a >\!> \text{settle } b & = & \text{settle } (a \diamond b) & \text{(semigroup action on a monad)} \\
\text{settle mempty} & = & \text{return } () & \text{(monoid action on a monad)} \\
\text{diff } a\ a & = & \text{mempty} & \text{(no change)} \\
\text{see} >\!> \text{return } () & = & \text{return } () & \text{(querying changes nothing)} \\
\text{see} = \text{return } a \Rightarrow \text{settle } (\text{diff } a\ b) >\!> \text{see} & \approx_{\text{is}} & \text{return } b & \text{(observation of settled difference)}
\end{array}$$

Figure 1. Laws of the change, when a state is only partially accessible for making a snapshot with see. We are using $\approx_{\text{is}}$ to indicate equivalence modulo ignoring state.

```
patch (ByKey {..}) v = updates
                     $ additions
                     $ deletions v
  where
    additions = Map.union        additions
    deletions = Map.withoutKeys deleted
    updates   = Merge.mergeA  failedHunk
                         Merge.preserveMissing
                        (Merge.zipWithAMatched)
                         applyHunk updated
    applyHunk hunk m = m>>=patch hunk
    failedHunk = Merge.dropMissing
```

3 Summary

We exhibit the current status of our work-in-progress to use schema-driven programming in order to facilitate updates of the standard schema. This schema can represent just a directory full of files, or a remote configuration of a cloud service. Schema-driven programming with generic derivations[3] and higher-kinded datatypes allows us to reduce boilerplate code significantly, while still benefitting from type-safety of keeping the same virtual information on different representations.

4 Bibliography

[1] Beam: 2018. *http://travis.athougies.net/projects/beam.html.*

[2] Fumiaki Kinoshita 2019. Barbies-th: Create strippable HKD via TH. Hackage.

[3] Gorín, D. 2018. Barbies: Classes for working with types that can change clothes. Hackage.

[4] Guides and API References: *https://docs.aws.amazon.com/#user_guides.*

[5] Gundry, A. 2019. Announcing the optics library. (Sep. 2019).

[6] Lempsink, E. et al. 2009. Type-safe diff for families of datatypes. *Proceedings of the 2009 acm sigplan workshop on generic programming* (New York, NY, USA, 2009), 61–72.

[7] Miraldo, V.C. et al. 2017. Type-directed diffing of structured data. *Proceedings of the 2nd acm sigplan international workshop on type-driven development* (New York, NY, USA, 2017), 2–15.

[8] Miraldo, V.C. and Swierstra, W. 2019. An efficient algorithm for type-safe structural diffing. *Proc. ACM Program. Lang.* 3, ICFP (Jul. 2019). DOI:https://doi.org/10.1145/3341717.

[9] O'Connor, R. 2011. Functor is to lens as applicative is to biplate: Introducing multiplate. *CoRR.* abs/1103.2841, (2011).

[10] Penner, C. 2019. Higher kinded option parsing. Blog post.

[11] Swierstra, W. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (Jul. 2008), 423–436. DOI:https://doi.org/10.1017/S0956796808006758.

[3]Supported for free datatypes only.

On Structuring Pure Functional Programs Using Monoidal Profunctors

Alexandre Garcia de Oliveira
FATEC - Rubens Lara
Santos, São Paulo, Brazil

Mauro Jaskelioff
CIFASIS - CONICET
Rosario, Santa Fe, Argentina

Ana Cristina Vieira de Melo
University of São Paulo
São Paulo, São Paulo, Brazil

Abstract

We study monoidal profunctors as a tool to reason and compose pure functional programs. We present a formalization of this structure, and we show the free monoidal profunctor construction, some primary instances, and some applications in a Haskell context such as optics and type-safe lists. The relationship between monoidal profunctor optics and other existent optics is also discussed.

CCS Concepts: • **Theory of computation** → Categorical semantics.

Keywords: Monoidal Profunctors, Category Theory, Functional Programming.

ACM Reference Format:
Alexandre Garcia de Oliveira, Mauro Jaskelioff, and Ana Cristina Vieira de Melo. 2018. On Structuring Pure Functional Programs Using Monoidal Profunctors. In *IFL '20: The 32nd Symposium on Implementation and Application of Functional Languages, September 02–04, 2018, ONLINE.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

It is well-known that pure functional programming views programs as pure mathematical functions without computational side-effects. Compositionality is a powerful tool for structuring such programs [19] and leads us to write clean, efficient, and easy to reason code.

Category theory [13] has inspired many tools to achieve compositional programs. Monads [18] allow composition by making distinctions between values and computations. Applicative functors [15], are similar to monads and gain compositionality at the cost of only dealing with static computations. Arrows [6] are focused on compositional processes that model machine-like constructions.

IFL '20, September 02–04, 2020, online
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

A comparison among monads, applicatives, and arrows [12] shows that a theory of idioms (applicative functors) can be embedded into static arrows and monads into high-order arrows. In the chain of abstractions of unary type constructors, applicative functors lie between functors (the weakest) and monads (the strongest).

Monoidal profunctors are a categorical structure with two components: an identity computation and a generic parallel composition. Being a profunctor, they may lift pure computations into its structure. Arrows are, in a sense, a generalisation of monads from unary type constructors to binary type constructors [7, 25], where the first type parameter is contravariant and the second covariant. In this analogy, profunctors play the role of functors. This work studies whether a monoidal profunctor is the applicative equivalent for such binary type constructors.

This work's primary motivation is to investigate if monoidal profunctors can be used to structure pure functional programs: Can monoidal profunctors be used to structure and reason about pure functional programs in the same manner as applicative functors? Can the gap in the following table be filled with monoidal profunctors?

functor	applicative	monad
profunctor	????	arrow

Table 1. Structure relations

Therefore, with this paper we aim to gather the knowledge about monoidal profunctors and study their application in the context of functional programming, helping its use in the Haskell ecosystem.

Possible applications for monoidal profunctors are in parallel programming, as a tool for reasoning about contexts, and even optics [1]. We present an application of monoidal product profunctors in the optics area and observe connections with well-known structures such as traversals and grates [20]. The category-theoretic framework around this structure is also provided. This work presents some useful and primary instances for the monoidal product profunctor type-class and discusses some applications seen in the Haskell ecosystem.

Many works propose categorical structures to reason about pure functional programs. We are not aware of any other work that investigates the use of monoidal profunctors to do this. Hughes introduced arrows as a generalized interface for computations [6], which has a sequential composition interface alongside a parallel one. Monoidal product profunctors exposes only a parallel composition interface, and hence are weaker than Arrows.

This work provides another instance of the use of categorical monoids to model computations and follows the same approach as in the work of Rivas and Jaskelioff [25].

In the optics area, works such as [24] discuss the uses of profunctors to achieve the same results at a higher level of abstraction than the original work by Van Laarhoven [28]. Using Doubles [22] and Tambara Modules [17] one can build a plethora of profunctor optics [1, 26]. Using a construction similar to a representation theorem for second-order functionals [8], and a profunctor version of it [21], monoidal profunctor optics can be built with a different approach than the aforementioned investigations.

An application of monoidal product profunctors is present on the packages `product-profunctors` [5] and `opaleye` [4]. The former presents a way to generate type-safe lists using a type-class that holds default computations in joint work with such profunctors. We discuss this technique in Section 5.

This work is divided as follows. Section 2 presents the notion of a monoidal category and its laws, describes profunctors, and defines the Day convolution. Section 3 introduces monoidal profunctors together, the notion of a monoid on top of it, and a free monoidal profunctor together with a representation theorem for profunctors. Section 4 discusses instances and examples of the type-class `MonoPro`, and Section 5 applications such as type-safe lists and monoidal product profunctor optics.

2 Category theory background

2.1 Monoidal Categories

The definition of a monoidal category gives us a minimal framework for defining a monoid in a category.

Definition 1. *A monoidal category is a sextuple* $(C, \otimes, I, \alpha, \rho, \lambda)$ *where*

- C *is a category;*
- $\otimes : C \times C \to C$ *is a bifunctor;*
- I *is an object called unit;*
- $\rho_A : A \otimes I \to A$, $\lambda_A : I \otimes A \to A$ *and* $\alpha_{ABC} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$ *are three natural isomorphisms such that the diagrams below commute.*

$$\begin{tikzcd}
A \otimes (B \otimes (C \otimes D)) \arrow[r, "\alpha"] \arrow[dd, "id\otimes\alpha"'] & (A \otimes B) \otimes (C \otimes D) \arrow[d, "\alpha"] \\
 & ((A \otimes B) \otimes C) \otimes D \\
A \otimes ((B \otimes C) \otimes D) \arrow[r, "\alpha"'] & (A \otimes (B \otimes C)) \otimes D \arrow[u, "\alpha\otimes id"']
\end{tikzcd}$$

$$\begin{tikzcd}
A \otimes (I \otimes B) \arrow[rr, "\alpha"] \arrow[dr, "id\otimes\lambda"'] & & (A \otimes I) \otimes B \arrow[dl, "\rho\otimes id"] \\
 & A \otimes B. &
\end{tikzcd}$$

If the isomorphisms ρ, λ and α are equalities then the monoidal category is called strict, if there is a natural isomorphism $\gamma_{AB} : A \otimes B \to B \otimes A$ the monoidal category is called symmetric.

A monoidal category is closed if there is an additional functor, called the internal hom, $\Rightarrow: C^{op} \times C \to Set$ such that $C(A \otimes B, C) \cong C(A, B \Rightarrow C)$ for every A, B and C objects of C. The witnesses of this isomorphism are called currying and uncurrying. In Set, $A \Rightarrow B$ is just the hom-set $A \to B$.

A symmetric closed monoidal category [13] is the main categorical tool for reasoning about pure functional programs in this work.

Definition 2. *A monoid in a monoidal category* C *is the tuple* (M, e, m) *where* M *is an object of* C, $e : I \to M$ *is the unit morphism and* $m : M \otimes M \to M$ *is the multiplication morphism, satisfying*

1. *Right unit:* $m \circ (id \otimes e) = \rho$
2. *Left unit:* $m \circ (e \otimes id) = \lambda$
3. *Associativity:* $m \circ (id \otimes m) = m \circ (m \otimes id) \circ \alpha$

The following commuting diagrams represent those laws.

$$\begin{tikzcd}
M \otimes I \arrow[r, "id\otimes e"] \arrow[dr, "\rho"'] & M \otimes M \arrow[d, "m"] & I \otimes M \arrow[r, "e\otimes id"] \arrow[dr, "\lambda"'] & M \otimes M \arrow[d, "m"] \\
 & M & & M
\end{tikzcd}$$

$$\begin{tikzcd}
M \otimes (M \otimes M) \arrow[r, "id\otimes m"] \arrow[dd, "\alpha"'] & M \otimes M \arrow[dr, "m"] & \\
 & & M \\
(M \otimes M) \otimes M \arrow[r, "m\otimes id"] & M \otimes M \arrow[ur, "m"] &
\end{tikzcd}$$

2.2 Profunctors

A profunctor generalizes the notion of function relations and bimodules [11].

Definition 3. *Given two categories* C *and* $\mathcal{D}$, *a profunctor is a functor* $P : C^{op} \times \mathcal{D} \to Set$, *written* $P : C \nrightarrow \mathcal{D}$, *consists of:*

- *for each* a *object of* C *and* b *object of* $\mathcal{D}$, *a set* $P(a, b)$;
- *for each* a *object of* C *and* b, d *objects of* $\mathcal{D}$, *a function (left action)* $\mathcal{D}(d, b) \times P(a, d) \to P(a, b)$;
- *for each* a, c *objects of* C *and* b *object of* $\mathcal{D}$, *a function (right action)* $P(a, b) \times C(c, a) \to P(c, b)$.

This definition is also known as a Bimodule or a $(C,\mathcal{D})$-module.

Since a profunctor is a functor from the product category $\mathcal{C}^{op} \times \mathcal{D}$ to Set, it must satisfy the functor laws.

$$P(1_C, 1_D) = 1_{P(C,D)}$$
$$P(f \circ g, h \circ i) = P(g, h) \circ P(f, i)$$

Note that the units 1_C and 1_D are identity morphisms on objects C and D of the categories $\mathcal{C}$ and $\mathcal{D}$, while $1_{P(C,D)}$ is an identity morphism in Set. The second law tells us that a profunctor preserve the composition of morphisms of any morphisms f, g from $\mathcal{C}$ and h, i of $\mathcal{D}$.

An example of a profunctor is the hom functor $Hom : \mathcal{C}^{op} \times \mathcal{C} \to Set$, written as $A \to B$ when $\mathcal{C} = Set$. the profunctor actions are pre-composition and post-compostion of set valued functions.

Definition 4. *Let $\mathcal{C}$ and $\mathcal{D}$ small categories, $Prof(\mathcal{C}, \mathcal{D})$ is the profunctor category consisting of profunctors as objects, natural transformations as morphisms, and vertical composition to compose them.*

The profunctor category inherits some structure from the functor category $Set^{\mathcal{C}}$ such as binary products given by $(P \times Q)(S, T) = P(S, T) \times Q(S, T)$ and binary coproducts given by $(P + Q)(S, T) = P(S, T) + Q(S, T)$, where $\times, +$ are the respective universal constructions from Set. There is also terminal and initial profunctors given by $1_p(S, T) = \{*\}$ and $0_p(S, T) = \emptyset$, i.e., constant profunctors on initial and terminal objects in Set. If the target of a profunctor that is not Set, but some other category, say $\mathcal{E}$, with binary products and coproducts, initial and terminal objects, the profunctor category based on top of $\mathcal{E}$ will also have these constructs.

2.3 Day Convolution

Definition 5. *Let $\mathcal{C}$ be a small monoidal category and $F, G : \mathcal{D} \to Set$, the Day convolution [2] of F and G is another functor (in T) given by*

$$(F \star G)T = \int^{X,Y \in Ob(\mathcal{D})} FX \times GY \times Hom_{\mathcal{D}}(X \otimes Y, T). \quad (1)$$

The co-end (or an end when present) in this definition can have a notational reduction to

$$\int^{XY} FX \times GY \times Hom_{\mathcal{D}}(X \otimes Y, T)$$

whenever the context is clear.

We instantiate this convolution in the category $Prof$ of profunctors letting $\mathcal{D} = \mathcal{C}^{op} \times \mathcal{C}$ be the described product category. For this definition we use the calculus of ends and coends. For any object (S, T) in this category:

$$\begin{aligned}
&(F \star G)(S, T) \\
&= \int^{ABCD} F(A, B) \times G(C, D) \times [\mathcal{C}^{op} \times \mathcal{C}]((A, B) \otimes (C, D), (S, T)) \\
&\cong \int^{ABCD} F(A, B) \times G(C, D) \times [\mathcal{C}^{op} \times \mathcal{C}]((A \otimes C, B \otimes D), (S, T)) \\
&\cong \int^{ABCD} F(A, B) \times G(C, D) \times \mathcal{C}^{op}(A \otimes C, S) \times \mathcal{C}(B \otimes D, T) \\
&\cong \int^{ABCD} F(A, B) \times G(C, D) \times \mathcal{C}(S, A \otimes C) \times \mathcal{C}(B \otimes D, T)
\end{aligned}$$

The profunctor $J(A, B) = \mathcal{C}(A, I) \times \mathcal{C}(I, B)$ is a unit for $\star$. When $I = 1$, where 1 is a terminal object, $J(A, B) \cong B$.

Proposition 1. *Let $\mathcal{C}$ be a monoidal category, the profunctor $J(A, B) = \mathcal{C}(A, I) \times \mathcal{C}(I, B)$ is the right and left unit of $\star$.*

The associativity of $\star$ is required to define a monoidal profunctor category.

Proposition 2. *Let $(\mathcal{C}, \otimes, I)$ be a monoidal category and S, T two objects of $\mathcal{C}$, the Day convolution for profunctors is an associative tensor product $(P \star Q) \star R \cong P \star (Q \star R)$*

In order to be able to define monoids in a monoidal profunctor category, one needs to check that when $\mathcal{C}$ and $\mathcal{D}$ are monoidal categories then $(Prof(\mathcal{C}, \mathcal{D})), \star, J)$ is a monoidal category.

Proposition 3. *Let $\mathcal{C}$ and $\mathcal{D}$ are monoidal small categories. Then $(Prof(\mathcal{C}, \mathcal{D})), \star, J)$ is a monoidal category.*

Proof. *Since $\mathcal{C}$ and $\mathcal{D}$ are monoidal categories, $\star$ is a bifunctor by construction, and by Proposition 1 and 2 gives the desired morphisms, it follows that $(Prof(\mathcal{C}, \mathcal{D})), \star, J)$ is a monoidal category.*

It is now possible to define a monoid in this category by showing that a morphism going out of Day convolution of profunctors is in one-to-one correspondence with a morphism not using this tensor, as in the work of Rivas and Jaskelioff [25].

Proposition 4. *Let $\mathcal{D} = \mathcal{C}^{op} \otimes \mathcal{C}$, there is a one-to-one correspondence defining morphisms going out of a Day convolution for profunctors*

$$\int_{XY} (P \star Q)(X, Y) \to R(X, Y)$$
$$\cong \int_{ABCD} P(A, B) \times Q(C, D) \to R(A \otimes C, B \otimes D)$$

which is natural in P, Q and R.

Whenever $P = Q$ in the equation of Proposition 4 we get $\int_{ABCD} P(A, B) \times P(C, D) \to P(A \otimes C, B \otimes D)$ useful to define a monoid in the profunctor category $Prof$ with Day convolution as its tensor.

2.4 Yoneda lemma

The famous Yoneda Lemma [3], in its covariant and contravariant, needs to be stated in order to proceed.

Lemma 1 (Yoneda). *Let $\mathcal{C}$ be a* locally small category *and $F : \mathcal{C} \to Set$ a covariant functor. There is an isomorphism*

$$FX \quad \cong \quad Nat(\mathcal{C}(X, -), F)$$

natural in X. The Meaning is that there is a natural isomorphism between the set FX and the set of natural transformations involving the hom functor $\mathcal{C}(X, -)$ and F [25]. The same lemma holds when considering a contravariant functor $G : \mathcal{C} to Set$. There is also an isomorphism

$$GY \quad \cong \quad Nat(\mathcal{C}(-, Y), G)$$

natural in Y.

Using ends and coends, one can rewrite [3] the above lemma as :

$$FX \quad \cong \quad \int_A C(X,A) \to FA \quad \cong \quad \int^A FY \times C(A,Y).$$

The rightmost term is the well-known co-Yoneda lemma, which holds by the duality principle.

3 Monoidal Profunctors

This section aims to provide the essential categorical tool to derive a Haskell representation for a monoid on a monoidal category of profunctors. This section also discusses the free monoidal profunctor construction and a representation theorem for profunctors.

3.1 A monoid on monoidal profunctors

The unit and the multiplication of this monoid are a direct consequence of Yoneda's lemma and Proposition 4.

Proposition 5. *Let $(C, \otimes, I)$ be a small monoidal category, $P : C^{op} \times C \to Set$ be a profunctor, and S, T two objects of C. Then $C(J(S,T), P(S,T)) \cong P(I,I)$.*

Proof.

$$\begin{aligned} C(J(S,T), P(S,T)) &\cong C(S,I) \times C(I,T) \to P(S,T) \\ &\cong C(S,I) \to C(I,T) \to P(S,T) \\ &\cong C(S,I) \to P(S,I) \\ &P(I,I) \end{aligned}$$

□

With all categorical tools in hand, the central notion of this works emerges from the category of monoidal profunctors.

Definition 6. *Let $(C, \otimes, I)$ be a small monoidal category. A monoid in the monoidal profunctor category is a profunctor P, a unit given by the natural transformation between the profunctors J and P, $e : J \to P$, equivalent to $e : P(I,I)$ by Proposition 5, and the multiplication is $m : P \star P \to P$ which is isomorphic to the family of morphisms $V(m)_{ABCD} = P(A,B) \times P(C,D) \to P(A \otimes C, B \otimes D)$. Such a monoid is called a* monoidal profunctor.

As an example, consider $(C, \otimes, I)$ any monoidal category and the *Hom* profunctor $P(A,B) = A \to B$, a monoid in the monoidal profunctor category $Prof(C^{op}, C)$ is obtained if we set

$$e : I \to I$$
$$e(x) = I$$

$$V(m)_{ABCD} : (A \to B) \times (C \to D) \to ((A \otimes C) \to (B \otimes D))$$
$$V(m)_{ABCD}(f, g) = f \otimes g$$

Internal homs exists in the monoidal profunctor category $Prof(C^{op}, C)$ and can be calculated:

Proposition 6. *Let $(C, \otimes, I)$ be a small monoidal category, and P, Q monoidal profunctors, then*

$$(P \Rightarrow Q)(X,Y) = \int_{CD} P(C,D) \to Q(X \otimes C, Y \otimes D)$$

defines an internal hom on the monoidal profunctor category.

This proposition means that the monoidal category of profunctors is closed.

3.2 Free monoidal profunctor

The notion of a fixpoint of an initial algebra enables a definition of the free structure for a monoidal profunctor.

Definition 7. *Let C be a category, given an endofunctor $F : C \to C$, a F-algebra consists of an object A of C, the carrier of the algebra, and an arrow $\alpha : F(A) \to A$. A morphism $h : (A, \alpha) \to (B, \beta)$ of F-algebras is an arrow $h : A \to B$ in C such that $h \circ \alpha = \beta \circ F(h)$.*

$$\begin{tikzcd} F(A) \arrow[r, "F(h)"] \arrow[d, "\alpha"'] & F(B) \arrow[d, "\beta"] \\ A \arrow[r, "h"] & B \end{tikzcd}$$

The category of F-algebras and its morphisms on a category C are called $F - Alg(C)$.

The existence of a free monoidal profunctor is guaranteed by the following proposition [25].

Proposition 7. *Let $(C, \otimes, I)$ be a monoidal category with exponentials. If C has binary coproducts, and for each $A \in ob(C)$ the initial algebra for the endofunctor $I + A \otimes -$ exists, then for each A the free monoid A^* exists and its carrier is the carrier of the initial algebra.*

$Prof(C^{op}, C)$, when C is a small monoidal category, is monoidal with the Day convolution $\star$ and the profunctor I as its unit, and also have binary products and exponentials. The least fixed point of the endofunctor $Q(X) = J + P \star X$ in $Prof(C^{op}, C)$ gives the free monoidal profunctor.

3.3 Representation Theorem

In the work of O'Connor and Jaskelioff [8], a representation theorem was derived that helps to obtain optics. In this work, the unary version of this representation theorem is needed.

Theorem 1. *Theorem 3.1 (Unary representation) Consider an adjunction $-^* \vdash U : \mathcal{E} \to \mathcal{F}$, where $\mathcal{F}$ is small and $\mathcal{E}$ is a full subcategory of Set^{Set} such that the family of functors $R_{A,B}(X) = A \times (B \to X)$ is in $\mathcal{E}$. Then, we have the following isomorphism natural in A, B, and X.*

$$\int_F (A \to U(F(B))) \to U(F(X)) \cong U(R^*_{A,B})(X)$$

This isomorphism ranges over any structure upon small functors F, such as pointed functors and applicatives, and is used to change representations from ends involving functors to simpler ones. It is possible to obtain the same unary representation for profunctors [21].

Theorem 2. *(Unary representation for profunctors) Consider an adjunction between profunctors $-^* \vdash U : \mathcal{E} \to \mathcal{F}$, where $\mathcal{F}$ is small and $\mathcal{E}$ is a full subcategory of $Prof(Set, Set)$, the family of profunctors $Iso_{A,B}(S,T) = (S \to A) \times (B \to T)$ gives the following isomorphism natural in A, B, and dinatural in S, T.*

$$\int_P UP(A,B) \to UP(S,T) \cong Iso^*_{A,B}(S,T)$$

Where Iso^* is the a free profunctor generated by Iso.

Since the free monoidal profunctor exists and is in the form

$$P^*(S,T) = (J + P \star P^*)(S,T),$$

this theorem helps us to find the unary representation for monoidal functors.

Proposition 8. *The unary representation for monoidal profunctors is given by the isomorphism:*

$$\int_P P(A,B) \to P(S,T) \cong \sum_{n \in \mathbb{N}} (S \to A^n) \times (B^n \to T)$$

where P ranges over all monoidal profunctors.

4 Programming examples

We now turn to Haskell code and show how to implement the ideas of the previous section.

4.1 Profunctor typeclass

In Haskell, a profunctor is an instance of the following class

```
class Profunctor p where
  dimap :: (a → b) → (c → d) → p b c → p a d
```

As we know that a profunctor is a functor, *dimap* needs to satisfy the functor laws as well.

$$dimap\ id\ id = id$$
$$dimap\ (f \circ g)\ (h \circ i) = dimap\ g\ h \circ dimap\ f\ i$$

Note that *dimap* has the left and right actions definitions of a profunctor together. In the `profunctors` library [10] there are two functions name *lmap* and *rmap* corresponding to those actions. The profunctor interface lifts pure functions into both type arguments, the first in a contravariant manner, and the second in a covariant way. A morphism in the $Prof$ category can be represented in Haskell as the type below.

```
type (⇝) p q = ∀x y.p x y → q x y
```

The hom-functor, in Haskell (→), is the most basic example of a profunctor.

```
instance Profunctor (→) where
  dimap ab cd bc = cd ∘ bc ∘ ab
```

One notion captured by a Profunctor is that of a structured input and output of a function (Kleisli arrow allows a pure input and a structured output, for example). A type representing these functions will be called, *SISO*.

```
data SISO f g a b = SISO {unSISO :: f a → g b}
instance (Functor f, Functor g) ⇒
  Profunctor (SISO f g) where
  dimap ab cd (SISO bc) = SISO (fmap cd ∘ bc ∘ fmap ab)
```

Two specializations of *SISO* are known in the Haskell's profunctor library, *Star* when f is the identity functor and *Costar* when g is.

Another profunctor example is a fold.

```
data Fold m a b = Fold ((b → m) → a → m)
instance Profunctor (Fold m) where
  dimap ab cd (Fold bc) = Fold (λdm → bc (dm ∘ cd) ∘ ab)
```

This amounts to *foldMap* when m is a monoid and the type $a{\sim}f\ b$ for *Foldable* f.

4.2 The Day convolution type

In Haskell, the Day convolution is represented by the existential type

```
data Day p q s t = ∀a b c d.
  Day (p a b) (q c d) (s → (a, c)) (b → d → t)
```

Since $\mathcal{C}(A, I)$ is isomorphic to a singleton set (unit of the cartesian product ×), and $\mathcal{C}(I, B) \cong B$, one can write, in Haskell, the type

```
data I a b = I {unI :: b}
```

as the unit of the Day convolution. The following functions are representations of the right and left units.

```
ρ :: Profunctor p ⇒ Day p I ⇝ p
ρ (Day pab (I d) sac bdt) =
  dimap (fst ∘ sac) (λb → bdt b d) pab
λ :: Profunctor q ⇒ Day I q ⇝ q
λ (Day (I b) qcd sac bdt) =
  dimap (snd ∘ sac) (λd → bdt b d) qcd
```

The associativity of the Day convolution and its symmetric map also can be represented in Haskell as the functions below.

```
α :: (Profunctor p, Profunctor q, Profunctor r) ⇒
  Day (Day p q) r ⇝ Day p (Day q r)
α (Day (Day p q s1 f) r s2 g) =
  Day p (Day q r f1 f2) f3 f4
  where
    f1            = first′ (snd ∘ s1) ∘ s2
    f2 d1 d2      = (d2, λx → f x d1)
    f3            = first′ (fst ∘ s1 ∘ (fst ∘ s2)) ∘ diag
    f4 b1 (d2, h) = g (h b1) d2
γ :: (Profunctor p, Profunctor q) ⇒ Day p q ⇝ Day q p
γ (Day p q sac bdt) = Day q p (swap ∘ sac) (flip bdt)
  where swap (x, y) = (y, x)
```

Since ρ, λ, and α are natural isomorphisms its inverses exist and are represented by the following Haskell functions.

$$\begin{array}{l}
\rho^{-1} :: \mathit{Profunctor}\ p \Rightarrow p \rightsquigarrow \mathit{Day}\ p\ I \\
\rho^{-1}\ \mathit{pab} = \mathit{Day}\ \mathit{pab}\ (I\ ())\ \mathit{diag}\ (\mathit{curry}\ \mathit{fst}) \\[4pt]
\lambda^{-1} :: \mathit{Profunctor}\ p \Rightarrow p \rightsquigarrow \mathit{Day}\ I\ p \\
\lambda^{-1}\ \mathit{pcd} = \mathit{Day}\ (I\ ())\ \mathit{pcd}\ \mathit{diag}\ (\mathit{curry}\ \mathit{snd}) \\[4pt]
\alpha^{-1} :: (\mathit{Profunctor}\ p, \mathit{Profunctor}\ q, \mathit{Profunctor}\ r) \Rightarrow \\
\quad \mathit{Day}\ p\ (\mathit{Day}\ q\ r) \rightsquigarrow \mathit{Day}\ (\mathit{Day}\ p\ q)\ r \\
\alpha^{-1}\ (\mathit{Day}\ p\ (\mathit{Day}\ q\ r\ s1\ f)\ s2\ g) = \\
\quad \mathit{Day}\ (\mathit{Day}\ p\ q\ f1\ f2)\ r\ f3\ f4 \\
\quad \mathbf{where} \\
\qquad f1 \qquad\quad = \mathit{second}'\ (\mathit{fst} \circ s1) \circ s2 \\
\qquad f2\ d1\ d2 \quad = (d1, \lambda x \rightarrow f\ d2\ x) \\
\qquad f3 \qquad\quad = \mathit{second}'\ (\mathit{snd} \circ s1 \circ (\mathit{snd} \circ s2)) \circ \mathit{diag} \\
\qquad f4\ (d1, h)\ b1 = g\ d1\ (h\ b1)
\end{array}$$

4.3 MonoPro typeclass

As a consequence, the type $p\ ()\ ()$ is a representation in Haskell of $P(I, I)$ and the Proposition 4 gives the multiplication $\int_{ABCD} P(A, B) \times P(C, D) \rightarrow P(A \otimes C, B \otimes D)$ allowing to write the following class in Haskell.

$$\begin{array}{l}
\mathbf{class}\ \mathit{Profunctor}\ p \Rightarrow \mathit{MonoPro}\ p\ \mathbf{where} \\
\quad \mathit{mpempty} :: p\ ()\ () \\
\quad (\star) :: p\ b\ c \rightarrow p\ d\ e \rightarrow p\ (b, d)\ (c, e)
\end{array}$$

satisfying the monoid laws

- Left identity:
$$\mathit{dimap}\ \mathit{diag}\ \mathit{snd}\ (\mathit{mpempty} \star f) = f$$
- Right identity:
$$\mathit{dimap}\ \mathit{diag}\ \mathit{fst}\ (f \star \mathit{mpempty}) = f$$
- Associativity:
$$\mathit{dimap}\ \mathit{assoc}^{-1}\ \mathit{assoc}\ (f \star (g \star h)) = (f \star g) \star h$$

where *assoc*, assoc^{-1} and *diag* are given by the Haskell functions below.

$$\begin{array}{l}
\mathit{diag} :: x \rightarrow (x, x) \\
\mathit{diag}\ x = (x, x) \\[4pt]
\mathit{assoc}^{-1} :: ((x, y), z) \rightarrow (x, (y, z)) \\
\mathit{assoc}^{-1}\ ((x, y), z) = (x, (y, z)) \\[4pt]
\mathit{assoc} :: (x, (y, z)) \rightarrow ((x, y), z) \\
\mathit{assoc}\ (x, (y, z)) = ((x, y), z)
\end{array}$$

If one focus on the second argument, i.e., fixing a profunctor p and an type s, $\mathit{MonoPro}\ p\ s$ inherits the applicative functor behavior naturally represented by the function

$$\begin{array}{l}
\mathit{appToMonoPro} :: \mathit{MonoPro}\ p \Rightarrow \\
\quad p\ s\ (a \rightarrow b) \rightarrow p\ s\ a \rightarrow p\ s\ b \\
\mathit{appToMonoPro}\ \mathit{pab}\ \mathit{pa} = \\
\quad \mathit{dimap}\ \mathit{diag}\ (\mathit{uncurry}\ (\$))\ (\mathit{pab} \star \mathit{pa})
\end{array}$$

with *pure* being *mpempty*.

The *MonoPro* class provides an abstraction of parallel composition and inherits the "zippy" nature of an Applicative (Monoidal) functor.

Another way to understand *MonoPro* is that it lifts pure functions with many inputs to a binary constructor type, while a profunctor only lifts functions with one type as input parameter. That fact is easily seen by comparing the two functions below.

$$\begin{array}{l}
\mathit{lmap} :: \mathit{Profunctor}\ p \Rightarrow (a \rightarrow b) \rightarrow p\ b\ c \rightarrow p\ a\ c \\[10pt]
\mathit{lmap2} :: ((b, \mathit{bb}) \rightarrow b') \rightarrow p\ a\ b \rightarrow p\ c\ \mathit{bb} \rightarrow p\ (a, c)\ b' \\
\mathit{lmap2}\ f\ \mathit{pa}\ \mathit{pc} = \mathit{dimap}\ \mathit{id}\ f\ \$\ \mathit{pa} \star \mathit{pc}
\end{array}$$

A monoidal profunctor has a straightforward instance for the Hom profunctor

$$\begin{array}{l}
\mathbf{instance}\ \mathit{MonoPro}\ (\rightarrow)\ \mathbf{where} \\
\quad \mathit{mpempty} = \mathit{id} \\
\quad f \star g = \lambda(a, b) \rightarrow (f\ a, g\ b)
\end{array}$$

A pratical use for this instance is writing expressions in a pointfree manner, one can write an *unzip′* function, for example, for any functor containing a pair type.

$$\begin{array}{l}
\mathit{unzip}' :: \mathit{Functor}\ f \Rightarrow f\ (a, b) \rightarrow (f\ a, f\ b) \\
\mathit{unzip}' = (\mathit{fmap}\ \mathit{fst} \star \mathit{fmap}\ \mathit{snd}) \circ \mathit{diag}
\end{array}$$

The datatype *SISO* is another example of a monoidal profunctor.

$$\begin{array}{l}
\mathbf{instance}\ (\mathit{Functor}\ f, \mathit{Applicative}\ g) \Rightarrow \\
\qquad\quad \mathit{MonoPro}\ (\mathit{SISO}\ f\ g)\ \mathbf{where} \\
\quad \mathit{mpempty} = \mathit{SISO}\ (\lambda\_ \rightarrow \mathit{pure}\ ()) \\
\quad \mathit{SISO}\ f \star \mathit{SISO}\ g = \mathit{SISO}\ (\mathit{zip}' \circ (f \star g) \circ \mathit{unzip}')
\end{array}$$

where *zip′* is the applicative functor multiplication given by

$$\begin{array}{l}
\mathit{zip}' :: \mathit{Applicative}\ f \Rightarrow (f\ a, f\ b) \rightarrow f\ (a, b) \\
\mathit{zip}'\ (\mathit{fa}, \mathit{fb}) = \mathit{pure}\ (,) \otimes \mathit{fa} \otimes \mathit{fb}
\end{array}$$

as one can observe, the most basic notion of a monoidal profunctor is represented by this instance. It tells us that the input needs to be a functor instance because of *unzip′*, the functions f and g are composed in a parallel manner using the monoidal profunctor instance for $(\rightarrow)$ and then regrouped together using the applicative (monoidal) behavior of *zip′*.

4.4 Free MonoPro

By expanding [16], the free monoidal profunctor is represented, in Haskell, by the following Generalized Abstract Data Type

$$\begin{array}{l}
\mathbf{data}\ \mathit{FreeMP}\ p\ s\ t\ \mathbf{where} \\
\quad \mathit{MPempty} :: t \rightarrow \mathit{FreeMP}\ p\ s\ t \\
\quad \mathit{FreeMP} :: (s \rightarrow (x, z)) \rightarrow ((y, w) \rightarrow t) \\
\qquad\qquad \rightarrow p\ x\ y
\end{array}$$

```
        → FreeMP p z w
        → FreeMP p s t
```

where *MPempty* is the equivalent of *mpempty*, and *FreeMP* is the multiplication expanding the definition of Day convolution for P and P^*. This interface stacks profunctors, and in each layer, it provides pure functions to simulate the parallel composition nature of a monoidal profunctor.

The following functions provide the means to build the free construction on monoidal profunctors, *toFreeMP* insert a single profunctor into the free structure, and *fromFreeMP* provides a way of evaluating the structure, collapsing into a single monoidal profunctor.

```
toFreeMP :: Profunctor p ⇒ p s t → FreeMP p s t
toFreeMP p = FreeMP diag fst p (MPempty ())


fromFreeMP :: MonoPro p ⇒ FreeMP p s t → p s t
fromFreeMP (MPempty t) =
  dimap (λ_ → ()) (λ() → t) mpempty
fromFreeMP (FreeMP f g p mp) =
  dimap f g (p ⋆ fromFreeMP mp)
```

A free construction behaves like list and, of course, *MonoPro* should provide a way to embed a plain profunctor into the free context.

```
consMP :: Profunctor p ⇒ p a b → FreeMP p s t
                           → FreeMP p (a, s) (b, t)
consMP pab (MPempty t) = FreeMP id id pab (MPempty t)
consMP pab (FreeMP f g p fp) =
  FreeMP (id ⋆ f) (id ⋆ g) pab (consMP p fp)
```

and with it, an instance of *MonoPro* for the free structure can be defined as

```
instance Profunctor p ⇒ MonoPro (FreeMP p) where
  mpempty = MPempty ()
  MPempty t          ⋆ q                  =
    dimap snd (λx → (t, x)) q
  q                  ⋆ MPempty t          =
    dimap fst (λx → (x, t)) q
  (FreeMP f g p fp) ⋆ (FreeMP k l pp fq) = dimap t1 t2 t3
    where
      t1 = (assoc′ ∘ (f ⋆ k))
      t2 = (sw ∘ (l ⋆ g) ∘ associnv)
      t3 = (consMP p (consMP pp (fp ⋆ fq)))
```

where $assoc :: ((x, z), c) \rightarrow (z, (x, c))$ and $associnv' :: (y, (w, d)) \rightarrow ((w, y), d)$. Hence, a free monoidal profunctor is indeed a monoidal profunctor.

A free monoidal profunctor *FreeMP p*, when *p* is an arrow, also can be derived. To achieve this instance, one needs to colapse all parallel profunctors in order to make the sequential composition as one can observe in the following functions.

```
instance (MonoPro p, Arrow p) ⇒
        K.Category (FreeMP p) where
  id = FreeMP (λx → (x, ())) fst (arr id) (MPempty ())
  mp ∘ mq = toFreeMP (fromFreeMP mpK. ∘ fromFreeMP mq)
instance (MonoPro p, Arrow p) ⇒
        Arrow (FreeMP p) where
  arr f = FreeMP (λx → (x, ())) fst (arr f) (MPempty ())
  (∗∗∗) = (⋆)
```

5 Applications

5.1 Type-safe lists

An application for the monoidal profunctor is to handle tuples instead of lists which give type-safety concerning its size. This techinique is found in the packages `opaleye` [4] and `product-profunctors` [5].

The monoidal profunctor interface lacks a function lifting like in *arr*, from Arrow type-class. One can understand *Default* as a type-class that picks a distinguished computation of the form *p a b* representing a lifted function basing on the structure of *p*.

```
class Default p a b where
  def :: p a b
```

Given two default computations, *p a b* and *p c d*, it is possible to overload *def* with the help of the GHC extension `MultiParamTypeClasses` to derive an instance for $p\ (a, c)\ (b, d)$.

```
instance (MonoPro p, Default p a b, Default p c d) ⇒
        Default p (a, c) (b, d) where
  def = def ⋆ def
```

If one has more than two computations, it is possible to overload it with the monoidal profunctor product and flattening functions like *flat3i*, *flat3l*, *flat4i*, *flat4l*, and so on (see Appendix). Those boilerplate codes can also be derived with the help of generics, template Haskell and quasi-quotations.

```
instance (MonoPro p,
        Default p a b,
        Default p c d,
        Default p e f) ⇒
        Default p (a, c, e) (b, d, f) where
  def = dimap flat3i flat3l (def ⋆ def ⋆ def)
instance (MonoPro p,
        Default p a b,
        Default p c d,
        Default p e f,
        Default p j k) ⇒
        Default p (a, c, e, j) (b, d, f, k) where
  def = dimap flat4i flat4l (def ⋆ def ⋆ def ⋆ def)
```

As examples, the functions *replicate* [5], *iterate*, and *zipWith* can have type-safe versions using this technique.

A *Replicator* is a type that enables the type-safe version of *replicate*.

```
newtype Replicator r f a b = Replicator (r → f b)
```

A profunctor instance for *Replicator r f*, noting that *a* is a phantom type argument since this, amounts to a functor applied to a type *b*. The phantom type *a* argument is needed to match the desired kind.

```
instance Functor f ⇒ Profunctor (Replicator r f) where
  dimap _ h (Replicator f) =
    Replicator ((fmap ∘ fmap) h f)
```

Whenever $r \sim f\ b$, one can choose *Replicator id* as its default value.

```
instance Applicative f ⇒
        Default (Replicator (f b) f) b b where
  def = Replicator id
```

A *Replicator* is a *MonoPro* when *f* is applicative; its monoidal profunctor product is just zip.

The function *replicateT* does the trick. It uses *def′*, which is deconstructed to *Replicator f*, to overload the monoidal product basing on a type given in runtime.

```
replicateT :: Default (Replicator r f) b b ⇒ r → f b
replicateT = f
  where Replicator f = def′
        def′ :: Default p a a ⇒ p a a
        def′ = def
```

For example, we may get three integers from the command line by

```
replicateT (readLn :: IO Int) :: IO (Int, Int, Int)
```

The number of integers varies with the type. In the case of iterators, it is important to note that this implementation differs slightly from the original *iterate* from `Data.List`, since the first element here is ignored.

```
data It a z b = It ((a → a) → a → (a, b))
```

An *It a* is a profunctor on *b* and has a trivial instance omitted here. A monoidal profunctor instance for *It a* works with the return type *a*, the first component of the tuple, acting as a state.

```
instance MonoPro (It a) where
  mpempty = It $ λh x → (h x, ())
  It f ⋆ It g = It $ λh x →
    let (y, b) = f h x
        (z, c) = g h y
    in (z, (b, c))
```

A default computation for *It* is one step iteration, and this will keep the iteration happening when computed the monoidal product.

```
instance Default (It a) z a where
  def = It $ λf a → (f a, f a)
```

Using the overloaded *def* again and deconstructing its type with the help of *itExplicit*, the function *iterT* is the type-safe version of *iterate*.

```
iterT :: Default (It a) b b ⇒
       (a → a) → a → b
iterT = itExplicit def
  where
    itExplicit :: It a b b → (a → a) → a → b
    itExplicit (It h) f a = snd $ h f a
```

Evaluating

```
iterT (2∗) 3 :: (Integer, Integer, Integer, Integer),
```

gives $(6, 12, 24, 48)$ which is exactly four iterations.

It is also possible to construct a type-safe version of the function *zipWith* relying on the type *Grate*. This example shows a connection with this technique and optics (more details in the next section).

```
data Grate a b s t = Grate (((s → a) → b) → t)
```

The datatype *Grate a b* is a profunctor on *s* and *t* and relies on a continuation-like style.

```
instance Profunctor (Grate x y) where
  dimap f g (Grate h) =
    Grate (λk → g (h (λt → k (t ∘ f))))
```

Its monoidal profunctor product instance unzips the input function and passes it to the monoidal product of *f* and *g*.

```
instance MonoPro (Grate x y) where
  mpempty = Grate $ λ_ → ()
  Grate f ⋆ Grate g =
    Grate (λh → (f ⋆ g) (k (unzip′ (Aux h))))
    where
      k = unAux ⋆ unAux
```

The type *Aux* is just a helper type that makes the definition of ⋆ easier.

```
data Aux x y a = Aux {unAux :: (a → x) → y}
```

Applying id to an input function is the default computation for a *Grate* whenever *s a* and $t \sim b$.

```
instance Default (Grate a b) a b where
  def = Grate (λf → f id)
```

The same pattern of *Replicator* and *It* also occurs with *Grate*.

```
grateT :: Default (Grate a b) s t ⇒ (((s → a) → b) → t)
grateT = grateExplicit def
  where
    grateExplicit :: Grate a b s t → (((s → a) → b) → t)
    grateExplicit (Grate g) = λf → g f
```

A type-safe *zipWith*, called *zipWithT*, can be constructed using the *grateT*.

$$
\begin{aligned}
&zipWithT :: (Int \to Int \to Int)\\
&\qquad\quad \to (Int, Int, Int)\\
&\qquad\quad \to (Int, Int, Int)\\
&\qquad\quad \to (Int, Int, Int)\\
&zipWithT\ op\ s1\ s2 = grateT\ (\lambda f \to op\ (f\ s1)\ (f\ s2))
\end{aligned}
$$

This connection with optics has an obvious limitation that it can only generate functions with explicit types like *zipWithT* to avoid ambiguous types. It is interesting to note that the same construction can be used to create type-safe traversals (which is also an optic). One needs to consider the above type *Traverse*, and *Traverse* ($) as default computation.

$$\textbf{data}\ Traverse\ f\ r\ s\ a\ b = Traverse\ ((r \to f\ s) \to a \to f\ b)$$

5.2 Monoidal profunctor optics

Data accessors are an essential part of functional programming. They allow reading and writing a whole data structure or parts of it [24]. In Haskell, one needs to deal with Algebraic Data Types (ADTs) such as products (fields), sums, containers, function types, to name a few. For each of these structures, the action of handling can be a hard task and not compositional at all. To circumvent this problem, the notion of modular (composable) data acessors [24] helps to tackle this problem with the help of some tools category-theoretic constructions such as profunctors.

An optic is a general denotation to locate parts (or even the whole) of a data structure in which some action needs to be performed. Each optic deals with a different ADT, for example, the well-known lenses deal with product types, prisms with sum types, traversals with traversable containers, grates with function types, isos deals with any type but cannot change its shape, and so on.

The idea of an optic is to have an in-depth look into get/set operations, for example, if one has a "big" data structure s, it is possible to extract a piece of it, say a, which can be written as a function $get :: s \to a$. Whereas, if one focus in a "big" structure s providing a value of b (part of f) it can turn in another "big" structure t (this may not change, and the data can still be s), a good manner to represent that is via the function $set :: s \to b \to t$.

Both functions can be amalgamated in terms of a binary type constructor p giving the type $\forall p.p\ a\ b \to p\ s\ t$, an optic amount in a suitable type class to constrain the polymorphic type p, for example, if p is *Strong*, $p\ a\ b \to p\ s\ t$ is a lens. If one plugs for p, the contravariant hom-functor which is *Strong* (also known as the data constructor $Forget :: (a \to r) \to Forget\ r\ a\ b$ in the Haskell ecosystem), and use $first' :: p\ a\ b \to p\ (a, x)\ (b, x)$ as a lens. One can see that gives the projection of the first component from a product type, producing, in this case, the function $get :: (a, x) \to a$.

Lenses help to give the intuition behind this profunctorial optics machinery, but this work will solely focus on the mixed optic derived from a monoidal profunctor with $\otimes = \times$, which combines grates and traversals. It will be called a mono.

Those two optics have the following types.

$$
\begin{aligned}
&\textbf{type}\ Iso\ s\ t\ a\ b = \forall p.Profunctor\ p \Rightarrow p\ a\ b \to p\ s\ t\\
&\textbf{type}\ Mono\ s\ t\ a\ b = \forall p.MonoPro\ p \Rightarrow p\ a\ b \to p\ s\ t
\end{aligned}
$$

Every *Mono* is an *Iso*. The latter provides us the necessary tool for handling isomorphisms between types.

$$
\begin{aligned}
&swap :: Profunctor\ p \Rightarrow p\ (b, a)\ (c, d) \to p\ (a, b)\ (d, c)\\
&swap = dimap\ sw\ sw\\
&associate :: Profunctor\ p \Rightarrow\\
&\qquad p\ ((w, y), d)\ ((x, z), c) \to p\ (y, (w, d))\ (z, (x, c))\\
&associate = dimap\ associnv\ assoc
\end{aligned}
$$

The *swap* iso represents the isomorphism $A \times B \cong B \times A$. It takes a profunctor and reverses the order of all product types involved, and *associate* iso represents an associative rule of product types. The units () can be treated as well but will be omitted.

A *Mono* locates every position from a product (tuple) type (which can be generalized to a finite vector [8]).

$$
\begin{aligned}
&each2 :: MonoPro\ p \Rightarrow p\ a\ b \to p\ (a, a)\ (b, b)\\
&each2\ p = p \star p\\
&each3 :: MonoPro\ p \Rightarrow p\ a\ b \to p\ (a, a, a)\ (b, b, b)\\
&each3\ p = dimap\ flat3i\ flat3l\ (p \star p \star p)\\
&each4 :: MonoPro\ p \Rightarrow p\ a\ b \to p\ (a, a, a, a)\ (b, b, b, b)\\
&each4\ p = dimap\ flat4i\ flat4l\ (p \star p \star p \star p)
\end{aligned}
$$

As one can observe, *each2* deals with parallel composition with the argument p with itself using the *monoPro* interface. The focus id on tuples of size 2. The monos *each3* and *each4* deal with tuples of size 3 and 4 and depends on the functions flattening functions defined earlier.

Actions can be performed on a mono, given the desired location; one can read/write any product (tuple) type.

$$
\begin{aligned}
&foldOf :: Monoid\ a \Rightarrow Mono\ s\ t\ a\ b \to s \to a\\
&foldOf\ \ mono = runForget\ (mono\ (Forget\ \ id))
\end{aligned}
$$

This action tells that given a Mono (location) one can monoidally collect many parts a from the big structure s (in this case, tuples). It is nice to remember that *Forget* is just the contravariant hom-functor, an instance of a *SISO*, when $f = Id$, and $g = Const\ r$ the constant applicative functor, whenever r (the covariant part of the *SISO*) is a monoid. For example,

$$foldOf\ \ each3 :: Monoid\ a \Rightarrow (a, a, a) \to a$$

behaves in the same way as the function *fold* do with lists, its evaluation on the value (`"AA"`, `"BB"`, `"CC"`) gives `"AABBCC"` as expected. A mono called *foldMapOf* can also behave like its list counterpart *foldMap*,

$$\begin{aligned}&\mathit{foldMapOf} :: \mathit{Monoid}\ r \Rightarrow \\ &\qquad \mathit{Mono}\ s\ t\ a\ b \to (a \to r) \to s \to r \\ &\mathit{foldMapOf}\ \ \mathit{lens}\ f = \mathit{runForget}\ (\mathit{lens}\ (\mathit{Forget}\ f))\end{aligned}$$

locating all elements of a 3-element tuple gives

$$\mathit{foldMapOf}\ \ \mathit{each3} :: \mathit{Monoid}\ r \Rightarrow (a \to r) \to (a, a, a) \to r$$

as mentioned.

Every profunctorial optic has a van Laarhoven [21], functorial representation, the base of the whole lens package [9]. Such representation can be extracted from a mono, obtained by the function

$$\begin{aligned}&\mathit{convolute} :: (\mathit{Applicative}\ g, \mathit{Functor}\ f) \Rightarrow \\ &\qquad \mathit{Mono}\ s\ t\ a\ b \to (f\ \ a \to g\ \ b) \to f\ \ s \to g\ \ t \\ &\mathit{convolute}\ \ \mathit{mono}\ f = \mathit{unSISO}\ (\mathit{mono}\ (\mathit{SISO}\ f))\end{aligned}$$

following the same pattern as in *foldMapOf* changing the *Forget* by a *SISO*. This representation was found in [20] and is called *FiniteGrate* relying on a typeclass called *Power* which is similar to *MonoPro* but without the monoidal profunctor semantics.

If we specialize *convolute* to the identity functor $f = \mathit{Id}$,

$$\begin{aligned}&\mathit{traverseOf} :: \mathit{Applicative}\ g \Rightarrow \\ &\qquad \mathit{Mono}\ s\ t\ a\ b \to (\mathit{Id}\ \ a \to g\ \ b) \to (\mathit{Id}\ \ s \to g\ \ t) \\ &\mathit{traverseOf}\ \ \mathit{mono} = \mathit{convolute}\ \ \mathit{mono}\end{aligned}$$

one gets the definition of a *Traversal*, which is a member of the lens package. Specializing *convolute* to the applicative functor $g = \mathit{Id}$,

$$\begin{aligned}&\mathit{zipFWithOf} :: \mathit{Functor}\ f \Rightarrow \\ &\qquad \mathit{Mono}\ s\ t\ a\ b \to (f\ \ a \to \mathit{Id}\ \ b) \to (f\ \ s \to \mathit{Id}\ \ t) \\ &\mathit{zipFWithOf}\ \ \mathit{mono} = \mathit{convolute}\ \ \mathit{mono}\end{aligned}$$

gives the van Laarhoven representation for grates (which depends on a Closed typeclass of Profunctors) [20].

$$\begin{aligned}&\textbf{class}\ \mathit{Profunctor}\ p \Rightarrow \mathit{Closed}\ p\ \textbf{where} \\ &\quad \mathit{closed} :: p\ \ a\ \ b \to p\ (x \to a)\ (x \to b)\end{aligned}$$

Monoidal profunctors with $\otimes = \times$ captures the essence of a grate and a traversal. Grates have a structured contravariant part (input) while traversals, the covariant one (output). A structured input and structured output function *SISO* played a significant role in this construction.

6 Conclusion

Although not providing specific syntactic tools like do notation, arrow notation [23], and applicative do [14], this work centralized many studies related to monoidal profunctors, some applications and derived connections to optics. A step further towards the use of such a structure is made. An investigation towards using other monoidal profunctors (when varying the tensor products) with distributive laws is needed. A study in this direction can provide another way to reason about mixed optics [1] and fruitful applications such as static parser [27]. Monoidal alternative profunctors, and its free version, could be derived in the same way as this work does provide an interesting tool to be used alongside with monoidal profunctors.

References

[1] Bryce Clarke, Derek Elkins, Jeremy Gibbons, Fosco Loregiàn, Bartosz Milewski, Emily Pillmore, and Mario Román. 2020. Profunctor optics, a categorical update. *ArXiv* abs/2001.07488 (2020).

[2] Brian Day. 1970. On closed categories of functors. In *Reports of the Midwest Category Seminar IV*, S. MacLane, H. Applegate, M. Barr, B. Day, E. Dubuc, Phreilambud, A. Pultr, R. Street, M. Tierney, and S. Swierczkowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–38.

[3] Brian Day and Max Kelly. 1969. Enriched functor categories. *Reports of the Midwest Category Seminar III (Lecture Notes in Mathematics)* Volume 106 (1969).

[4] Tom Ellis. [n.d.]. opaleye: An SQL-generating DSL targeting PostgreSQL. https://hackage.haskell.org/package/opaleye. Accessed: 2019-05-28.

[5] Tom Ellis. [n.d.]. Product-profunctors. https://hackage.haskell.org/package/product-profunctors. Accessed: 2019-05-20.

[6] John Hughes. 2005. Programming with Arrows. In *Proceedings of the 5th International Conference on Advanced Functional Programming* (Tartu, Estonia) *(AFP'04)*. Springer-Verlag, Berlin, Heidelberg, 73–129. https://doi.org/10.1007/11546382_2

[7] Bart Jacobs, Chris Heunen, and Ichiro Hasuo. 2009. Categorical semantics for arrows. *Journal of Functional Programming* 19, 3-4 (2009), 403–438. https://doi.org/10.1017/S0956796809007308

[8] Mauro Jaskelioff and Russell O'Connor. 2014. A representation theorem for second-order functionals. *ArXiv* abs/1402.1699 (2014).

[9] Edward Kmett. [n.d.]. lens: Lenses, Folds and Traversals. https://hackage.haskell.org/package/lens. Accessed: 2019-05-28.

[10] Edward Kmett. [n.d.]. Profunctors. https://hackage.haskell.org/package/profunctors. Accessed: 2019-03-16.

[11] Tom Leinster. 2003. Higher Operads, Higher Categories. arXiv:math/0305049 [math.CT]

[12] Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. *Electronic Notes in Theoretical Computer Science* 229, 5 (2011), 97 – 117. https://doi.org/10.1016/j.entcs.2011.02.018 Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008).

[13] Saunders MacLane. 1971. *Categories for the Working Mathematician.* Springer-Verlag, New York. ix+262 pages. Graduate Texts in Mathematics, Vol. 5.

[14] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's Do-Notation into Applicative Operations. *SIGPLAN Not.* 51, 12 (Sept. 2016), 92–104. https://doi.org/10.1145/3241625.2976007

[15] Conor Mcbride and Ross Paterson. 2008. Applicative Programming with Effects. *J. Funct. Program.* 18, 1 (Jan. 2008), 1–13. https://doi.org/10.1017/S0956796807006326

[16] Bartosz Milewski. [n.d.]. Free Monoidal Profunctors. https://bartoszmilewski.com/2018/02/20/free-monoidal-profunctors. Accessed: 2019-10-20.

[17] Bartosz Milewski. [n.d.]. Tambara. https://bartoszmilewski.com/2016/01/21/tambara-modules/. Accessed: 2020-03-20.

[18] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (July 1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[19] Davor Obradovic. 1998. Structuring Functional Programs By Using Monads.

[20] RusselL O'Connor. [n.d.]. Grate: A new kind of Optic. https://r6research.livejournal.com/28050.html. Accessed: 2019-02-02.

[21] RusseLl O'Connor. [n.d.]. A Representation Theorem for Second-Order Pro-functionals. https://r6research.livejournal.com/27858.html. Accessed: 2019-02-01.

[22] Craig A. Pastro and Ross Street. 2008. Doubles for monoidal categories. *arXiv: Category Theory* (2008).

[23] Ross Paterson. 2003. Arrows and Computation. In *The Fun of Programming*, Jeremy Gibbons and Oege de Moor (Eds.). Palgrave, 201–222. http://www.soi.city.ac.uk/~ross/papers/fop.html

[24] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor Optics: Modular Data Accessors. *The Art, Science, and Engineering of Programming* 1, 2 (Apr 2017). https://doi.org/10.22152/programming-journal.org/2017/1/7

[25] Exequiel Rivas and Mauro Jaskelioff. 2017. Notions of Computation as Monoids (extended version).

[26] Mario Román. 2020. Profunctor optics and traversals. *ArXiv* abs/2001.08045 (2020).

[27] S. Swierstra and L. Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming*.

[28] T van Laarhoven. [n.d.]. Where do I get my non-regular types? http://twanvl.nl/blog/haskell/non-regular2. Accessed: 2020-08-08.

Resource Analysis for Lazy Evaluation with Polynomial Potential

Sara Moreira
Pedro Vasconcelos
Mário Florido
Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
Portugal

ABSTRACT

Operational properties of lazily-evaluated programs are hard to predict at compile-time. This is an obstacle to a broad adoption of non-strict programming languages. In 2012 it was introduced a novel type-and-effect analysis for predicting upper-bounds on memory allocation costs for programs in a simple lazily-evaluated functional language [17]. This analysis was successfully applied to several programs, but limited to bounds that are linear in the size of the input. Here we overcome that shortcoming by extending this system to polynomial resource bounds.

CCS CONCEPTS

• **Theory of computation → Program analysis**; *Type theory*; • **Software and its engineering → Functional languages**.

KEYWORDS

Resource analysis, Amortised analysis, Type-based analysis, Lazy evaluation

ACM Reference Format:
Sara Moreira, Pedro Vasconcelos, and Mário Florido. 2020. Resource Analysis for Lazy Evaluation with Polynomial Potential. In . ACM, New York, NY, USA, 10 pages. https://doi.org/XXXX/XXXX

1 INTRODUCTION

Lazy evaluation offers known advantages in terms of modularity and higher abstraction [10]. However, operational properties of programs (such as time and space behaviour) are more difficult to predict than for strict languages. This can be an obstacle to a more widespread use of non-strict programming languages, such as Haskell.

Previous work on type-based amortised analysis for lazy languages has enabled the automatic prediction of resource bounds for lazy higher-order functional programs with linear costs on the number of (co)data constructors [12, 17]. While this system is an important contribution, it is limited to linear bounds, which means that functions with polynomial costs can not be typed. Because many functions fall under this category, it is important to overcome this limitation.

As a motivating example, consider the two functions *attach* and *pairs* (adapted to Haskell from [6]):

```
pairs :: [a] -> [(a, a)]
pairs [] = []
pairs (x:xs) = attach x xs ++ pairs xs

attach :: a -> [a] -> [(a, a)]
attach _ [] = []
attach y (x:xs) = (x,y): attach x ys
```

The function *pairs* takes a list and computes a list of pairs that are two-element sub-lists of the given list; this uses an auxiliary definition *attach* that pairs a single element to every element of the argument list.

It is straightforward that *attach* consumes time and space that is linear on the length *n* of the input list. Moreover, a precise bound can be derived by the type system in [12] through a type annotated with a constant *potential* associated with each list node of the input list. Function *pairs*, however, exhibits quadratic time and space on the length its input. Hence, it does not admit a type derivation in the mentioned system.

In this paper we extend type-based amortised analysis of non-strict languages to *polynomial* resource bounds by following the approach of Hoffman for the strict setting [2, 7]. The analysis is presented for a small lazy functional language with higher-order functions, pairs, lists and recursion. Finally, we give examples of the application of our analysis to programs exhibiting polynomial resource behaviour.

The rest of the paper is organised as follows. Section 2 surveys relevant background and related work about amortised analysis. Section 3 presents the language and its annotated operational semantics. Section 4 presents the main contribution of this paper: a type and system for resource analysis of lazy evaluation with polynomial bounds. In Section 5, we show several worked examples of the analysis. Finally, we conclude and present some future work.

2 BACKGROUND AND RELATED WORK

2.1 Type-based Analysis

Type-based analysis [14] is an approach to static analysis that attaches static analysis information to types.

One main advantage of this approach is the fact that it facilitates modular analysis since types allow the expression of interfaces between components. It also helps the communication with the programmer by extending an already-known notation, namely, types.

Other advantages revolve around efficiency and completeness. Types provide an infrastructure from which the analysis can be done. For example, in a type and effect system, each typing rule provides a localised setting for the analysis. Furthermore, the correctness of the analysis is subsumed by the correctness of the type system, which means that the correctness of the analysis can be formulated and proven using the well-studied methods in type systems. Overall, these systems improve the information given by types by decorating them with annotations so that they express more about the program being analysed.

2.2 Classic Amortisation

Amortised analysis [15, 18] is a method for analysing the complexity of a sequence of operations. While worst-case analysis considers the worst case for each operation, and average-case analysis considers the average cost over all possible inputs, amortised analysis is concerned with the overall worst-case cost over a sequence of operations. The motivation for this type of analysis arises from the fact that some operations can be costly, while others can be faster or "cheaper", and in the end, they can even each other out. In some cases, analysing the worst-case per operation may be too pessimistic.

In an amortised analysis we define a notation of "amortised cost" for each operation that satisfies the following equation:

$$\sum_{n=1}^{m} a_n \geq \sum_{n=1}^{m} t_n$$

With a as the amortised cost and t as the actual cost, this means that, for each sequence of operations, the total amortised cost is an upper bound of the total actual cost. As a consequence, in each intermediate step of the sequence, the accumulated amortised cost is an upper bound of the accumulated actual cost. This allows for the existence of operations with an actual cost that exceeds their amortised cost, these are called *expensive operations. Cheap operations* are operations with a cost lower than their amortised cost. Expensive operations can only occur when the difference between the accumulated amortised cost and the accumulated actual cost (*accumulated savings*) is enough to cover the "extra" cost.

There are three different methods for amortised analysis: the *aggregate method (total cost)*, the *accounting method (banker's view)* and the *potential method* (physicist's view). The choice of which to use depends on how convenient each is to the situation.

Potential method. This method defines a function Φ that maps each state of the data structure d_i to a real number (*potential of* d_i). This function should be chosen such that the potential of the initial state is 0 and never becomes negative, that is, $\Phi(d_0) = 0$ and $\Phi(d_i) \geq 0$, for all i. This potential represents a lower bound to the accumulated savings.

The amortised cost of an operation is defined as its actual cost (t_i), plus the change in potential between d_{i-1} and d_i, where d_i is the state of data structure before operation i:

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

This means that:

$$\begin{aligned}\sum_{i=1}^{j} t_i &= \sum_{i=1}^{j} (a_i + \Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \sum_{i=1}^{j} (\Phi(d_{i-1}) - \Phi(d_i)) \\ &= \sum_{i=1}^{j} a_i + \Phi(d_0) - \Phi(d_j)\end{aligned}$$

Note that the sequence of potential function values forms a *telescoping series* and thus all terms except the initial and final values cancel in pairs. And because $\Phi(d_j)$ is always equal or greater than $\Phi(d_0)$, then $\Sigma(a_i) \geq \Sigma(t_i)$.

Note that with the right choice of a potential function, the amortised analysis gives a tighter bound for a sequence of operations than simply analysing each operation individually.

2.3 Automatic Amortised Analysis

In 2003, Hofmann and Jost [8] proposed a system for static automatic analysis of heap space usage for a strict first-order language. This system was able to obtain linear bounds on the heap space consumption of a program by using a type system refined with resource annotations. This annotated type system allowed the analyser to predict the amount of heap space needed to evaluate the program by keeping track of the memory resources available. This form of analysis would later be recognised as automatic amortised resource analysis (AARA).

Further work has since then been done using this approach, which is, more specifically, based on the potential method of amortised analysis. The main idea behind this method is the association of *potential* to data structures. This potential is assigned using type annotations, where the annotations serve as coefficients for the potential functions. The key to a successful analysis is the choice of a ''good" potential function, ''good" being a potential function that simplifies the amortised costs. Because the inference of suitable annotations can be reduced to a linear optimisation problem, it is possible to automatically infer the potential function.

Following work by the same authors [9] used the same approach to obtain heap space requirements for Java-like programs with explicit deallocations. The data is assigned a potential related to its input and layout, and the allocations are then paid with this potential. This way, the potential provides an upper bound on the heap space usage for the given input. Whereas in the previous work a refined type consisted of a simple type together with a number, object-oriented languages require a more complex approach due to aliasing and inheritance, and so a refined type in this context consists of a number together with refined types for the attributes and methods.

Later, Atkey [1] presented a system that extends AARA to pointer-manipulation languages by embedding a logic of resources based on intuitionistic logic of bunched implications within separation logic.

In 2010 [7], the same authors address the biggest limitation on previous article [8]: restriction to linear bounds. Their new system infers polynomial upper bounds on resource usage for first-order programs as a function of their input and is generic in terms of resources. This extension is done without losing expressiveness. The inferred polynomial bounds result in linear constraints, meaning that the inference of polynomial bounds can still be reduced to a linear optimisation problem.

Jost et al [11] presented the first automatic amortised analysis able to determine linear upper-bounds on the use of quantitative resources for strict, higher-order recursive programs.

In [4] it is studied how AARA can be used to derive worst-case resource usage for procedures with several arguments, and the previous inference of bounds is generalised for arbitrary multivariate polynomials (with limits like $m * n$). The drawbacks of an univariate analysis are the fact that many functions have multivariate characteristics, and the fact that, if data from different sources is interconnected in a program, multivariate bounds like $(m+n)^2$ will appear.

In 2016, Hoffman et al. [5] presented a resource analysis system based on AARA that derives worst-case polynomial bounds for higher-order programs with user-defined inductive types, which was integrated into Inria's OCaml compiler.

In [17], AARA is extended to compute linear bounds for lazily evaluated functional languages. This is an important extension because it tries to remove an obstacle to the broader use of lazy languages: the fact that resource usage for their execution is very hard to predict. This system improves the precision of the analysis for co-recursive data by combining two previous analyses that considered the allocation costs of recursive and co-recursive programs. The system was subsequently extended to a parametric cost model and for tracking self-references in co-recursive definitions [12, 19], which is essential to model the graph reduction techniques that are typically used in lazy functional language implementations.

2.4 Polynomial potential

In this section, we briefly explain Hoffman's approach to polynomial potential [7]. We go over the main contributions of this system and what influenced our approach.

This article presents a technique for inferring polynomial bounds, that still relies only on linear constraints. This is a very important feature because, until then, it was considered that the dependence on linear programming imposed a limitation to linear bounds.

One key aspect of this work is the use of binomial coefficients as a basis for polynomials, rather than the more common monomial basis x^n for $n >= 0$

First, let us consider a list of type $L^{\vec{p}}(A)$. This is a simple list type, refined with a resource annotation $\vec{p} = (p_1, \ldots, p_k)$, where $(p_1, \ldots, p_k)$ represents a vector of coefficients that will be used to calculate the potential of the list. We can translate this annotated type to: the number q_1 is the potential assigned to every element of the list, q_2 is the potential assigned to every element of every suffix of the list, q_3 is the potential assigned to every element of every suffix of the suffixes of the list, and so on.

The main advantage of using binomial coefficients is the fact that it simplifies the definition of the *additive shift*. The additive shift is an operation on the coefficients represented by a resource annotation, that corresponds to the change in potential for typing branches of a pattern match. Let us consider a vector of coefficients

$$\frac{\vec{p} = (p_1, \ldots, p_k)}{\Sigma; x_h{:}A, x_t{:}L^{\triangleleft(\vec{p})} \left|\frac{p_1 + K^{cons}}{0}\right. \mathtt{cons}(x_h, x_t){:}L^{(\vec{p})}(A)}\ \text{T:Cons}$$

$$\frac{\begin{array}{c}\vec{p} = (p_1, \ldots, p_k)\\ \Sigma; \Gamma; x_h{:}A, x_t{:}L^{\triangleleft(\vec{p})}(A) \left|\frac{q + p_1 - K_1^{matC}}{q' + K_2^{matC}}\right. e_1{:}B \\ \Sigma; \Gamma \left|\frac{q - K^{nil}}{q' - K^{nil}}\right. e_2{:}B\end{array}}{\Sigma; x{:}L^{(\vec{p})} \left|\frac{q}{q'}\right. \mathtt{match}\ x\ \mathtt{with}\ \mathtt{cons}(x_h, x_t)\ \texttt{->}\ e_1\ \mathtt{|}\ \mathtt{nil}\ \texttt{->}\ e_2{:}B}\ \text{T:MatL}$$

Figure 1: Rules T:Cons and T:MatL

$\vec{p} = (p_1, p_2, \ldots, p_k)$, the additive shift of vector $\vec{p}$ is

$$\triangleleft\vec{p} = (p_1 + p_2, p_2 + p_3, \ldots, p_{k-1} + p_k, p_k)$$

The idea is that the potential assigned to the tail $xs{:}L^{\triangleleft\vec{p}}$ of a list $x :: xs{:}L^{\vec{p}}$ is used to pay for recursive calls, calls to auxiliary functions and constant costs before and after recursive calls.

Similarly to the other works on AARA, the inference of constraints on the resource annotations is done during type inference, so it is also important to explain how these concepts were introduced in the type rules and why. As mentioned, the additive shift allows the typing of the branches of a pattern match, so naturally, we see these concepts arise in match rules and constructor rules. In his analysis, Hoffman works with list and tree data structures, but because we only consider lists in our analysis, we are only interested in the rules written for lists. We can see them in Fig. 1.

Some things to mention before explaining the particularities of these rules, note how the turnstile is annotated with values, one above and another below. Those are the values that keep track of resource usage during type inference. To be more specific, a judgement of the form $\Gamma \left|\frac{z'}{z}\right. e{:}C$ can be read as: considering a typing environment Γ and with z resource units available, we can infer the type C for the expression e and infer that the evaluation of e consumes $z - z'$ resource units.

T:Cons infers the type of a list constructor and illustrates the fact that one has to pay for the potential that is assigned to the new list. To do so, they require that the tail of the list x_t is typed with the additive shift of the potential of the new list and that there are p_1 resource units available. The parameter K^{cons} is a parametric constant, it is there to formalise the fact that we need to pay for the cost of allocating space for the new list. The rule T:MatL complements T:Cons, and shows how to use the potential of a list to pay for resource usage, particularly in the "cons" branch. The tail of the list is annotated with the additive shift of the potential of the list, allowing recursive calls (with annotation $\vec{p}$) and calls to auxiliary functions (with annotation $(p_2, p_3, \ldots)$), furthermore, p_1 resource units become directly available.

To summarise, we have explained the idea behind the additive shift and described how Hoffmann introduced it in the type inference rules. The way it is inserted into the type system through a vector of coefficients, and the way the type rules use these values during inference is used in our system in a mostly identical manner.

2.5 Lazy evaluation

In [12], Jost et al. approach the problem of inferring strict cost bounds for lazy functional languages by taking advantage of an AARA system to keep track of resource usage. In this section, much like in the previous one, we briefly explain this approach, focusing mainly on the key points that we took advantage of for our system.

The main contributions of this system deal with the particularities of the mechanics that define lazy evaluation, namely, how it delays the evaluation of arguments and uses references to prevent multiple evaluations of the same terms.

One very important contribution is the introduction of an annotated *thunk* structure to the type system. This structure essentially denotes a delayed evaluation of a term and maintains the cost of evaluating the delayed term. $\mathsf{T}^p(A)$ means: to evaluate the delayed expression of type A, we need p resource units available.

The use of resource annotations is also crucial, much like in other AARA systems. They are used during type inference to keep track of the resource usage of an expression, and attached to the types of functions to denote the overall cost evaluating the function.

$$\Gamma \,\Big|\!\frac{z'}{z}\; e{:}C$$

This judgement means, under the environment Γ and with z resource unit available, the evaluation of e has type C and leaves z' resource units available.

Finally and possibly the most important contribution, the type rule Prepay. This is a structural rule that allows the cost of a thunk to be paid in advance, preventing that same cost to be accounted in further uses of the same thunk, "simulating" this way the memoization of a call-by-need evaluation.

$$\frac{\Gamma, x{:}\mathsf{T}^{q_0}(A) \,\Big|\!\frac{p}{p'}\; e{:}\,C}{\Gamma, x{:}\mathsf{T}^{q_0+q_1}(A) \,\Big|\!\frac{p+q_1}{p'}\; e{:}\,C} \quad (\textsc{Prepay})$$

These are the main points that we considered to understand how we could handle lazy evaluation in our analysis. Supplementary to these elements, we also took advantage of most syntactic and semantic choices of this article to write our system and the language that supports it. We will come back to these choices next when we explain our language and operational semantics.

3 LANGUAGE AND OPERATIONAL SEMANTICS

In this section, we present the language and operational semantics against which our analysis is done.

We start by introducing a *simple lazy functional language* (SLFL) composed by the syntactical terms e and w, presented in Fig. 2. Our expressions e include variables, lambda expressions, list constructors, let-expressions, and pattern matching. The values w are in weak head normal form and include constant values, pairs, list constructors and lambda expressions. To simplify the presentation of our expressions, sometimes we will be using a semicolon instead of in in let-expressions.

As mentioned, our syntax and cost model are largely based on Jost et al.'s semantics [12], which in its turn, is based on Sestof's revision [16] of Launchbury's operational semantics for lazy evaluation [13]. The main difference is the restriction to list and pairs constructors rather than more general recursive types. This was done to simplify the presentation, and we believe it would be a straightforward task to extend this system to more general data structures.

$$\begin{array}{rcl}
e & ::= & c \;\mid\; \lambda x.\, e \;\mid\; e\ y \;\mid\; \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
 & \mid & (x_1, x_2) \;\mid\; \mathsf{cons}(x_h, x_t) \;\mid\; \mathsf{nil} \\
 & \mid & \mathsf{match}\ e_0\ \mathsf{with}\ (x_1, x_2)\ \text{->}\ e_1 \\
 & \mid & \mathsf{match}\ e_0\ \mathsf{with}\ \mathsf{cons}(x_h, x_t)\ \text{->}\ e_1 \mid \mathsf{nil}\ \text{->}\ e_2 \\
 & & \\
w & ::= & c \;\mid\; \lambda x.\, e \;\mid\; (x_1, x_2) \;\mid\; \mathsf{cons}(x_h, x_t) \;\mid\; \mathsf{nil}
\end{array}$$

Figure 2: Syntax for SLFL expressions and normal forms

3.1 Operational semantics

In this section, we present the rules that define the operational semantics for SLFL. Before we explore the rules in more detail, it is important to explain the structure of our judgements and its meaning:

$$\mathcal{H}, \mathcal{S}, \mathcal{L} \,\Big|\!\frac{m}{m'}\; e \Downarrow w, \mathcal{H}'$$

The relation can be read as follows: under a heap $\mathcal{H}$, a set of bound variables $\mathcal{S}$ and a set of locations $\mathcal{L}$, an expression e is evaluated to value w, in weak head normal form, consuming $m - m'$ resource units and producing a new heap $\mathcal{H}$'. The semantic rules in Fig. 3 illustrate how an expression is evaluated.

A heap $\mathcal{H}$ is a mapping from variables to *thunks*. As was mentioned in Section 2, a *thunk* is a delayed evaluation of an expression, meaning that our heap saves expressions that are possibly not yet evaluated. A set of locations $\mathcal{L}$ is used to keep track of the locations of the expressions that are being evaluated (See rule $\textsc{Var}_{\Downarrow}$), this is done to prevent cyclic evaluation. We also use a set of variables $\mathcal{S}$ to keep track of bound variables.

The operational semantics is instrumented by a counting mechanism that keeps track of resource usage for each expression. The resource usage tracked in these rules is the target of our cost analysis. For simplicity, we decided that our analysis would only be interested in calculating cost bounds on the number of allocations used in an expression. Note that, however, the system could easily be extended to consider multiple cost parameters, such as the number of steps, number of applications, and others. This could be done by assigning different constants to each reduction rule to specify how many resource units should be available when considering a specific cost parameter. We can see this parametrization be used in Hoffman's [7] and Jost et al.'s [12] analyses. In our system we consider only one constant, 1, in the reduction rules Let and Letcons.

Discussing the evaluation rules. As mentioned above, these rules are largely based on the semantics from [12], their construction and meaning are mostly identical. The main differences can be seen in the definition for rules $\textsc{Match-L}_{\Downarrow}$, $\textsc{Match-P}_{\Downarrow}$ and $\textsc{Letcons}_{\Downarrow}$.

$$\frac{}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m} w \Downarrow w, \mathcal{H}} \quad (\textsc{whnf}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \cup \{l\} \vdash^{m}_{m} e \Downarrow w, \mathcal{H}'}{\mathcal{H}[l \rightarrow e], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} l \Downarrow w, \mathcal{H}'[l \rightarrow w]} \quad (\textsc{Var}_{\Downarrow})$$

$$\frac{l \text{ is fresh} \qquad \mathcal{H}[l \rightarrow e_1[l/x]], \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e_2[l/x] \Downarrow w, \mathcal{H}'}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m+1}_{m'} \text{let } x = e_1 \text{ in } e_2 \Downarrow w, \mathcal{H}'} \quad (\textsc{Let}_{\Downarrow})$$

$$\frac{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m'} e \Downarrow \lambda x.\, e', \mathcal{H}' \qquad \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e'[y/x] \Downarrow w, \mathcal{H}''}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} e\ y \Downarrow w, \mathcal{H}''} \quad (\textsc{App}_{\Downarrow})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup (\{x_1, x_2\} \cup \text{BV}(e_1) \cup \text{BV}(e_2)), \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow \text{cons}(l_1, l_2), \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e_1[l_1/x_1, l_2/x_2] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} \text{match } e_0 \text{ with cons}(x_1, x_2) \text{ -> } e_1 \mid \text{nil -> } e_2 \Downarrow w, \mathcal{H}''} \quad (\textsc{Match-L}_{\Downarrow})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup (\{x_1, x_2\} \cup \text{BV}(e_1) \cup \text{BV}(e_2)), \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow \text{nil}, \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e_2 \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} \text{match } e_0 \text{ with cons}(x_1, x_2) \text{ -> } e_1 \mid \text{nil -> } e_2 \Downarrow w, \mathcal{H}''} \quad (\textsc{Match-N}_{\Downarrow})$$

$$\frac{\begin{array}{c}\mathcal{H}, \mathcal{S} \cup (\{x_1, x_2\} \cup \text{BV}(e_1) \cup \text{BV}(e_2)), \mathcal{L} \vdash^{m}_{m'} e_0 \Downarrow (l_1, l_2), \mathcal{H}' \\ \mathcal{H}', \mathcal{S}, \mathcal{L} \vdash^{m'}_{m''} e_1[l_1/x_1, l_2/x_2] \Downarrow w, \mathcal{H}''\end{array}}{\mathcal{H}, \mathcal{S}, \mathcal{L} \vdash^{m}_{m''} \text{match } e_0 \text{ with } (x_1, x_2) \text{ -> } e_1 \Downarrow w, \mathcal{H}''} \quad (\textsc{Match-P}_{\Downarrow})$$

Figure 3: Operational semantics for SLFL

Rule $\textsc{whnf}_{\Downarrow}$: A lambda expression, a constructor and a constant are already final values so they evaluate to themselves and leave the heap unmodified. This incurs no cost.

Rule $\textsc{Var}_{\Downarrow}$: A variable l that is linked to an expression e in the initial heap, evaluates to a value w if the evaluation of e reaches that same value. The final heap will have the expression e that is linked to l, replaced by the value w, this way we avoid re-evaluations of e, obtaining *lazy evaluation*. This means that the cost of evaluating a variable is the cost of evaluating the expression that is associated with it.

Rule $\textsc{Let}_{\Downarrow}$: the expression e_1 bound to x is not evaluated, instead a thunk is allocated and associated with a fresh location l in the heap. The rules proceed to evaluate the expressions e_2. Because the purpose of our analysis is to infer cost bounds on the number of allocations, the evaluation of these rules needs to cost at least 1 resource unit, plus the cost of evaluating e_2.

Rule $\textsc{Match-P}_{\Downarrow}$ and $\textsc{Match-L}_{\Downarrow}$: In both these rules, the variables bound by the pattern matching are replaced in each branch by the respective locations that result from the evaluation of e_0 and are stored in the heap. The final value and heap are the result of evaluating the branch taken.

Example 3.1. Consider the term:

$$\begin{array}{c}\text{let } f = \text{let } z = z;\ (\lambda x.\lambda y.y)\ z \\ \text{in let } i = \lambda x.x;\ \text{let } v = f\ i\ ;\ f\ v\end{array}$$

.

We can see how this term evaluates to $\lambda x.\, x$ under the rules of Fig. 3, leaving a heap $\Theta = [l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow \lambda x.x]$.

$$[l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow l_1\ l_2\ ] \vdash^{0}_{0} \lambda x.x_{\Downarrow}\lambda x.x, [l_1 \longrightarrow \lambda y.y] \qquad \textsc{Whnf}_{\Downarrow} \quad (1)$$

$$[l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow l_1\ l_2\ ] \vdash^{0}_{0} l_{2\Downarrow}\lambda x.x, [l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x] \qquad \textsc{Var}_{\Downarrow}\ (1) \quad (2)$$

$$[l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow l_1\ l_2\ ] \vdash^{0}_{0} \lambda y.y_{\Downarrow}\lambda y.y, [l_1 \longrightarrow \lambda y.y] \qquad \textsc{Whnf}_{\Downarrow} \quad (3)$$

$$[l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow l_1\ l_2\ ] \vdash^{0}_{0} l_1 \Downarrow \lambda y.y, [l_1 \longrightarrow \lambda y.y] \qquad \textsc{Var}_{\Downarrow}\ (3) \quad (4)$$

$$[l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow l_1\ l_2\ ] \vdash^{0}_{0} l_1\ l_2\ \Downarrow \lambda x.x, [l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x] \qquad \textsc{App}_{\Downarrow}\ (4,2) \quad (5)$$

$$[l_1 \longrightarrow \lambda y.y, l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow l_1\ l_2\ ] \vdash^{0}_{0} l_{3\Downarrow}\lambda x.x, [\ldots, l_3 \longrightarrow \lambda x.x] \qquad \textsc{Var}_{\Downarrow}\ (5) \quad (6)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots, l_4 \longrightarrow z] \vdash^{0}_{0} \lambda y.y_{\Downarrow}\lambda y.y \qquad \textsc{Whnf}_{\Downarrow} \quad (7)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots, l_4 \longrightarrow z] \vdash^{0}_{0} (\lambda x.\lambda y.y)_{\Downarrow}\lambda x.\lambda y.y \qquad \textsc{Whnf}_{\Downarrow} \quad (8)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots, l_4 \longrightarrow z] \vdash^{0}_{0} (\lambda x.\lambda y.y)\ l_4\ {}_{\Downarrow}\lambda y.y \qquad \textsc{App}_{\Downarrow}\ (8,7) \quad (9)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots] \vdash^{1}_{0} \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ {}_{\Downarrow}\lambda y.y \qquad \textsc{Let}_{\Downarrow}\ (9) \quad (10)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ , \ldots] \vdash^{1}_{0} l_{1\Downarrow}\lambda y.y, [l_1 \longrightarrow \lambda y.y] \qquad \textsc{Var}_{\Downarrow}\ (10) \quad (11)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ , l_2 \longrightarrow \lambda x.x, l_3 \longrightarrow l_1\ l_2\ ] \vdash^{1}_{0} l_1\ l_3\ {}_{\Downarrow}\lambda x.x, \Theta \qquad \textsc{App}_{\Downarrow}\ (11,6) \quad (12)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ , l_2 \longrightarrow \lambda x.x] \vdash^{2}_{0} \text{let } v = l_1\ l_2\ ;\ l_1\ v\ {}_{\Downarrow}\lambda x.x, \Theta \qquad \textsc{Let}_{\Downarrow}\ (12) \quad (13)$$

$$[l_1 \longrightarrow \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ ] \vdash^{3}_{0} \text{let } i = \lambda x.x;\ \text{let } v = l_1\ i\ ;\ l_1\ v\ {}_{\Downarrow}\lambda x.x, \Theta \qquad \textsc{Let}_{\Downarrow}\ (13) \quad (14)$$

$$\vdash^{4}_{0} \text{let } f = \text{let } z = z;\ (\lambda x.\lambda y.y)\ z\ ;\ \text{let } i = \lambda x.x;\ \text{let } v = f\ i\ ;\ f\ v\ {}_{\Downarrow}\lambda x.x, \Theta \qquad \textsc{Let}_{\Downarrow}\ (14) \quad (15)$$

4 LAZY EVALUATION WITH POLYNOMIAL POTENTIAL

In this section, we present our type system to analyse resource usage and provide a detailed description of how the analysis works using some illustrating examples.

4.1 Annotated Types

Here, we present the syntax for the annotated types of our language and the type rules used to perform the cost analysis. Types include primitives, function types, thunks, pairs and lists.

$$A,\ B \quad ::= \quad \mathsf{int} \mid A \xrightarrow{q} B \mid \mathsf{T}^{q}(A) \mid A \times B \mid \mathsf{L}^{q}(\vec{p}, A)$$

The variables q and $\vec{p}$ stand for *cost annotations*. More precisely, $\vec{p}$ stands for *list potential* and actually represents a vector of cost annotations, $\vec{p} = (p_1, \ldots, p_n)$.

The annotation q on function types is an upper bound on the cost of applying that function. Thunk types represent a delayed evaluation of an expression of type A and are also annotated with an upper bound on the cost of evaluating the delayed expression. List types are annotated with a simple annotation q, representing the cost of evaluating one constructor of the list, and a vector annotation $\vec{p}$, which represents the potential associated with that list. The primitive type int is free of cost annotations and type pairs is a pair of any type.

We define the additive shift of a vector of coefficients $\vec{p}$ as Hoffmann (see section 2.4):

$$\triangleleft(p_1, p_2, \ldots, p_n) = (p_1 + p_2,\ p_2 + p_3,\ \ldots,\ p_{n-1} + p_n,\ p_n)$$

We also define an addition operation on vectors of coefficients of equal length:

$$(p_1,\ \ldots,\ p_n) + (q_1,\ \ldots,\ q_n) = (p_1 + q_1,\ p_2 + q_2,\ \ldots,\ p_n + q_n)$$

In Fig. 4 and Fig. 5 we present the type rules used to derive these types and their cost annotations.

4.2 The sharing relation

Before we go on to explain how the type system works, it is important to explain the concept of *sharing*: $A \curlyvee \{B_1, \ldots, B_n\}$. In short, sharing allows the potential of a type A to be distributed amongst other types $\{B_1, \ldots, B_n\}$. The rules presented in Fig. 6 illustrate how the sharing relation applies depending on the types it is used on, and they follow very strictly the construction and explanation of the sharing rules presented in [12]. The main difference is present in the rule regarding list types (because Jost et al. system deals with possibly recursive algebraic data types, and not only lists). In our sharing relation, the Sharelist rule allows for the potential of a certain list A, to be shared amongst types B_i.

4.3 Subtyping

The subtyping relation is a particular case of sharing. It allows us to relax the annotations associated to a type by requiring them to be greater or equal than those of that type. We say a type A_1 is a subtype of a type A when $A_1 <: A$, this relation could also be represented as $A_1 \curlyvee \{A, A'\}$, where A' is a type with annotations

$$\frac{}{\vdash^{0}_{0} n : \mathsf{int}} \quad (\textsc{Const})$$

$$\frac{}{x{:}\mathsf{T}^{p}(A) \vdash^{p}_{0} x : A} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash^{z}_{z'} e : A \xrightarrow{p} C}{\Gamma, y : A \vdash^{z+p}_{z'} e\ y : C} \quad (\textsc{App})$$

$$\frac{\Gamma, x{:}A \vdash^{p}_{0} e : C \qquad x \notin \Gamma \qquad \Gamma \curlyvee \{\Gamma, \Gamma\}}{\Gamma \vdash^{0}_{0} \lambda x.e : A \xrightarrow{p} C} \quad (\textsc{Abs})$$

$$\frac{\begin{array}{c} A \curlyvee \{A, A'\} \qquad x \notin \{\Gamma, \Delta\} \qquad e_1 \text{ is not a constructor} \\ \Gamma, x : \mathsf{T}^{0}(A') \vdash^{p}_{0} e_1 : A \qquad \Delta, x : \mathsf{T}^{p}(A) \vdash^{z}_{z'} e_2 : C \end{array}}{\Gamma, \Delta \vdash^{z+1}_{z'} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : C} \quad (\textsc{Let})$$

$$\frac{\begin{array}{c} \vec{q} = (q_1, \ldots, q_k) \quad A = \mathsf{L}^{p}(\vec{q}, B) \quad A \curlyvee \{A,\ A'\} \\ \Gamma, x : \mathsf{T}^{0}(A') \vdash^{0}_{0} \mathsf{cons}(x_h, x_t) : A \qquad \Delta, x : \mathsf{T}^{0}(A) \vdash^{z}_{z'} e : C \end{array}}{\Gamma, \Delta \vdash^{z+1+q_1}_{z'} \mathsf{let}\ x = \mathsf{cons}(x_h, x_t)\ \mathsf{in}\ e : C} \quad (\textsc{Letcons})$$

$$\frac{}{x_1{:}A_1, x_2{:}A_2 \vdash^{0}_{0} (x_1, x_2) : A_1 \times A_2} \quad (\textsc{Pair})$$

$$\frac{}{\vdash^{0}_{0} \mathsf{nil} : \mathsf{L}^{q}(\vec{p}, A)} \quad (\textsc{Nil})$$

$$\frac{}{x_h{:}B, x_t{:}\mathsf{T}^{p}(\mathsf{L}^{p}(\triangleleft\vec{q}, B)) \vdash^{0}_{0} \mathsf{cons}(x_h, x_t) : \mathsf{L}^{p}(\vec{q}, B)} \quad (\textsc{Cons})$$

$$\frac{\Gamma \vdash^{z}_{z'} e_0 : A_1 \times A_2 \qquad \Delta,\ x_1 : A_1,\ x_2 : A_2 \vdash^{z'}_{z''} e_1 : C}{\Gamma, \Delta \vdash^{z}_{z''} \mathsf{match}\ e_0\ \mathsf{with}\ (x_1, x_2)\ \texttt{->}\ e_1 : C} \quad (\textsc{Match-P})$$

$$\frac{\begin{array}{c} \vec{q} = (q_1, \ldots, q_k) \\ \Gamma \vdash^{z}_{z'} e_0 : \mathsf{L}^{p}(\vec{q}, A) \\ \Delta, x_h : A, x_t : \mathsf{T}^{p}(\mathsf{L}^{p}(\triangleleft\vec{q}, A)) \vdash^{z'+q_1}_{z''} e_1 : C \\ \Delta \vdash^{z'}_{z''} e_2 : C \end{array}}{\Gamma, \Delta \vdash^{z}_{z''} \mathsf{match}\ e_0\ \mathsf{with}\ \mathsf{cons}(x_h, x_t)\ \texttt{->}\ e_1 \mid \mathsf{nil}\ \texttt{->}\ e_2 : C} \quad (\textsc{Match-L})$$

Figure 4: Syntax directed type rules

$$\frac{\Gamma, x{:}\mathsf{T}^{q_0}(A) \vdash^{p}_{p'} e\colon C}{\Gamma, x{:}\mathsf{T}^{q_0+q_1}(A) \vdash^{p+q_1}_{p'} e\colon C} \quad (\textsc{Prepay})$$

$$\frac{\Gamma \vdash^{p}_{p'} e{:}C}{\Gamma, x{:}A \vdash^{p}_{p'} e\colon C} \quad (\textsc{Weak})$$

$$\frac{\Gamma, x{:}A_1, x{:}A_2 \vdash^{p}_{p'} e\colon C \quad A \curlyvee \{A_1, A_2\}}{\Gamma, x{:}A \vdash^{p}_{p'} e\colon C} \quad (\textsc{Share})$$

$$\frac{\Gamma \vdash^{p}_{p'} e\colon A \quad q \geq p \quad q - p \geq q' - p'}{\Gamma \vdash^{q}_{q'} e\colon A} \quad (\textsc{Relax})$$

$$\frac{\Gamma \vdash^{p}_{p'} e\colon A \quad A <: B}{\Gamma \vdash^{p}_{p'} e\colon B} \quad (\textsc{Subtype})$$

$$\frac{\Gamma, x{:}B \vdash^{p}_{p'} e\colon C \quad A <: B}{\Gamma, x{:}A \vdash^{p}_{p'} e\colon C} \quad (\textsc{Supertype})$$

Figure 5: Structural type rules

$$\frac{}{A \curlyvee \emptyset} \quad (\textsc{ShareEmpty})$$

$$\frac{A \curlyvee \{A_1, \ldots, A_n\} \qquad B \curlyvee \{B_1, \ldots, B_n\}}{A \times B \curlyvee \{A_1 \times B_1, \ldots, A_n \times B_n\}} \quad (\textsc{SharePair})$$

$$\frac{\begin{array}{c} B_i = \mathsf{L}^{p_i}(\vec{q}_i, A_i) \qquad A \curlyvee \{A_1, \ldots, A_n\} \\ \vec{q} \geq \sum_{i=1}^{n} \vec{q}_i \qquad p_i \geq p \end{array}}{\mathsf{L}^{p}(\vec{q}, A) \curlyvee \{B_1, \ldots, B_n\}} \quad (\textsc{ShareList})$$

$$\frac{A_i \curlyvee \{A\} \quad C \curlyvee \{C_i\} \quad q_i \geq p \quad (1 \leq i \leq n)}{A \xrightarrow{p} C \curlyvee \{A_1 \xrightarrow{q_1} C_1, \ldots, A_n \xrightarrow{q_n} C_n\}} \quad (\textsc{ShareFun})$$

$$\frac{A \curlyvee \{A_1, \ldots, A_n\} \quad q_i \geq p \quad (1 \leq i \leq n)}{\mathsf{T}^{p}(A) \curlyvee \{\mathsf{T}^{q_1}(A_1), \ldots, \mathsf{T}^{q_n}(A_n)\}} \quad (\textsc{ShareThunk})$$

$$\frac{}{\Gamma \curlyvee \emptyset} \quad (\textsc{ShareEmptyCtx})$$

$$\frac{A \curlyvee \{B_1, \ldots, B_n\} \quad \Gamma \curlyvee \Delta}{x\colon A, \Gamma \curlyvee (x\colon B_1, \ldots, x\colon B_n,\ \Delta)} \quad (\textsc{ShareCtx})$$

Figure 6: Sharing rules

greater than or equal to zero. We can say that subtyping has the following properties:

$$\begin{array}{ll}
\mathsf{int} <: \mathsf{int} & \\
\mathsf{T}^{q_1}(A_1) <: \mathsf{T}^{q_2}(A_2) & \text{if } q_1 \geq q_2 \text{ and } A_1 <: A_2 \\
A_1 \times A_2 <: B_1 \times B_2 & \text{if } A_1 <: B_1 \text{ and } B_1 <: B_2 \\
A_1 \xrightarrow{q_1} B_1 <: A_2 \xrightarrow{q_2} B_2 & \text{if } q_1 \geq q_2 \text{ and } A_1 <: A_2 \text{ and } B_2 <: B_1 \\
\mathsf{L}^{q_1}(\vec{p}_1, A_1) <: \mathsf{L}^{q_2}(\vec{p}_2, A_2) & \text{if } q_1 \geq q_2 \text{ and } \vec{p}_1 \geq \vec{p}_2 \text{ and } A_1 <: A_2
\end{array}$$

We say $\vec{q} \geq \vec{p}$ if, $|\vec{q}| = |\vec{p}| = n$ and $\forall_{1 \leq i \leq n}, q_i \geq p_1$.

4.4 Type System

The type rules required for our analysis are presented in Fig. 4. These rules are complemented with the structural rules in Fig. 5, which introduce some flexibility to our analysis in ways that we will later explain. Our judgements have the form $\Gamma \vdash^{p'}_{p} e : A$ and can be read as follows: considering a typing context Γ, and with p resource units available, we can derive the annotated type A for expression e, leaving p' resource units available. These rules result from combining the ones presented in the two previous systems [7, 12] While many rules are identical to previous work, there are important differences in rules that concern the use of potential, namely, Letcons, Cons and Match.

We now describe each rule informally, focusing on on how type annotations express resource usage. Recall that we consider cost bounds for the number of allocations, i.e. the number of let-expressions evaluated.

Rule Const does not consume any resources as evaluating a primitive value incurs no additional allocations.

Rule Var deals with the elimination of a thunk type, so it is necessary to pay for the cost associated with that thunk.

Rules Let and Letcons deal with the allocation of a thunk for subexpressions. Both rules require at least 1 unit to be available (corresponding to the newly allocated thunk) and recursive use of the bound variable x is allowed. Note also that the side condition $A \curlyvee \{A, \ldots, A'\}$ that guarantees that the type A' does not have potential is required to ensure soundness (so that self-referencing structures are assigned zero potential [12]). Rule Let allows the cost of e_1 to be paid for only once, even in the case of self-reference; the intuition for this is that any productive uses of the bound variable in self-referencing definitions must be to an evaluated form [19].

Rule Letcons formalises the fact that one has to pay for the allocation of a new list constructor, which requires paying for the potential associated with the new list. We do so by requiring q_1 units to be available and complementing it with rule Cons, to be applied on the first expression e_1, which must be a list constructor.

Rules Cons and Pair are simple references to a constructor so they do not consume any resources. In rule Cons we do require the tail of the list to be annotated with the additive shift of its potential, complementing rule Letcons.

Rule App requires that the cost associated with a function is paid for each time the function is applied.

Rule Abs captures the cost of the expression in the type annotation of the function.

Rule Match-L shows how to use the potential of a list to pay for resource consumption. To do so, we require that the branch matching with the list constructor gains the excess potential q_1. We also annotate the tail of the list with the additive shift of the list potential, to allow future recursive calls or calls to auxiliary functions. This rule requires that both branches are of the same type C and that the amount of resources z'', available after the evaluation of each branch, is the same, which may require *relaxation* of the costs (See structural rule Relax in Fig. 5).

Rule Match-P deals with pattern matching against a pair constructor. Like in Match-P, we require that both branches are of the same type C and that the amount of resources z'' is the same.

5 WORKED EXAMPLES

To better understand how the analysis works, let us take a look at some examples.

Example 5.1. Let us consider function *pairs* in Fig. 7. This function is a translation into SLFL of the example from Section 1. Function *pairs* takes a list as an argument and computes a list of pairs that are two-element sub-lists the given list, while function *attach* combines each element of a list with the first argument. Note that the auxiliary function *app′* is the translation of list append with the argument order flipped, i.e. *app′* = `flip (++)`; this is done so that recursion is over the second argument and the type rules allow assigning potential to this argument.[1].

To facilitate the presentation of annotated type assignments, we have added potential annotations to list variables in Fig. 7: $l^{\vec{q}}$ means that variable l has type $\mathsf{L}^0(\vec{q}, B)$ for some B, i.e. l is a list with potential $\vec{q}$ and zero thunk cost for the spine. Since we expect function *pairs* has quadratic cost on the argument list length, we annotate it with pair of coefficients $\vec{q} = (q_1, q_2)$. Conversely, we expect functions *attach* and *app′* to have linear cost, hence we annotated these with a single coefficient.

Function *app′* is defined by structural recursion on the second argument l_2 and uses a single let-expression for each constructor in the argument; this means that l_2 should have a potential of at least 1 resource unit for each constructor. In *attach* we can see two let-expressions being used, which means the input potential should be at least 2. However, when analysing the body of function *pairs*, we can see that the output of *attach* is also the second input of app'. This means that to be able to type *pairs*, the output of *attach* must be compatible with the input of app', and because of that, its potential should be at least 1. Because the output potential needs to be accounted for in the input, we need to add it to the potential 2 we mentioned before.

[1]In particular, the side condition for rule Abs requires that the typing context Γ has no potential.

```
attach =  λn. λl. match l^{k_1} with
              nil->nil
              cons(x, xs^{j_1})-> let p = (x, n); f = attach n xs^{n_1}
                                  in cons(p, f)

app′ =  λl_1. λl_2. match l_2^{v_1} with
            nil->l_1
            cons(x, xs^{w_1})-> let f = app′ l_1 xs^{m_1}
                                in cons(x, f)

pairs =  λl. match l^{(q_1,q_2)} with
             nil->nil
             cons(x, xs^{(r_1,r_2)})-> let  f_1 = pairs xs^{(s_1,s_2)};
                                            f_2 = attach x xs^{(p_1,p_2)}
                                       in app′ f_1 f_2
```

Figure 7: Translation of the *pairs* function and auxiliary definitions into SLFL.

Using the annotations for *attach* and *app′* in Fig. 7, we derive the following constraints:

$$j_1 = k_1 \qquad \text{(additive shift)}$$
$$j_1 = n_1 \qquad \text{(share)}$$
$$n_1 = k_1 \qquad \text{(recursive call)}$$
$$k_1 \geq 2 + v_1$$

(two let-expressions plus the potential of the output of *attach*/input of app')

$$w_1 = v_1 \qquad \text{(additive shift)}$$
$$w_1 = m_1 \qquad \text{(share)}$$
$$m_1 = v_1 \qquad \text{(recursive call)}$$
$$v_1 \geq 1 \qquad \text{(single let-expression)}$$

We can solve this system of equations with $v_1 = m_1 = w_1 = 1$ and $q_1 = r_1 = s_1 = 3$ and derive the following annotated types:

$$app' : \mathsf{T}^0(\mathsf{L}^0(0, B \times B)) \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)$$
$$attach : B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(3, B)) \xrightarrow{0} \mathsf{L}^0(1, B \times B)$$

To better understand how the analysis works, we are going to illustrate the inference steps with more detail. The rules are applied in a very straightforward way, but it is important to pay attention to how resource usage is passed from and onto the judgements. Let us start by assuming:

$$\Gamma = app' : \mathsf{T}^0(\mathsf{L}^0(0, B \times B)) \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(1, B \times B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)$$
$$\Sigma = attach : B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(3, B)) \xrightarrow{0} \mathsf{L}^0(1, B \times B)$$

We will derive a type for *pairs* as follows:

$$\Theta = pairs : \mathsf{T}^0(\underbrace{\mathsf{L}^0((q_1, q_2), B))}_{L_{In}} \xrightarrow{p} \underbrace{\mathsf{L}^0((0,0), B \times B)}_{L_{Out}}$$

For simplicity, sometimes we omit certain elements of the type context that are not needed for the derivation in question. We also divide the definition of *pairs* into two sub-expressions as shown:

$$pairs = \lambda l.\ \overbrace{\texttt{match}\ l\ \texttt{with}\ \texttt{nil->nil}\ \ \texttt{cons}(x,xs)\texttt{->}\ \overbrace{\texttt{let}\ f_1 = pairs\ xs;\ f_2 = attach\ x\ xs\ \texttt{in}\ app'\ f_1\ f_2}^{e_2}}^{e_1}$$

We start by stating the typing obligation for the outer part of the recursive definition:

$$\Gamma, \Sigma \left|\frac{1}{0}\right. \texttt{let}\ pairs = \lambda l.\ e_1\ \texttt{in}\ pairs{:}\mathsf{T}^0(L_{In}) \xrightarrow{p} L_{Out} \tag{1}$$

By rule Let, we need to prove:

$$\Gamma, \Sigma, \Theta \left|\frac{0}{0}\right. \lambda l.\ e_1 : \mathsf{T}^0(L_{In}) \xrightarrow{p} L_{Out} \tag{2}$$

The later follows from rule Abs if we prove:

$$\Gamma, \Sigma, \Theta,\ l{:}\mathsf{T}^0(L_{In}) \left|\frac{p}{0}\right. e_1 : L_{Out} \tag{3}$$

By rule Match-L we get three new obligations; the first two correspond to the scrutinised list and the right-hand side of nil-case:

$$l{:}\mathsf{T}^0(L_{In}) \left|\frac{0}{0}\right. l{:}L_{In} \tag{Var}$$

$$\left|\frac{0}{0}\right. \texttt{nil}{:}L_{Out} \tag{Nil}$$

The remaining case for non-empty lists is:

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^0((q_1+q_2, q_2), B)) \left|\frac{q_1}{0}\right. e_2{:}L_{Out} \tag{4}$$

We now apply the Share rule to distribute the potential of the tail *xs* for the two uses in right-hand side expression e_2. The side condition is:

$$\mathsf{L}^0((q_1+q_2, q_2), B) \curlyvee \{\mathsf{L}^0((p_1,p_2),B),\ \mathsf{L}^0((s_1,s_2),B)\} \tag{5}$$

for some annotations p_1, p_2, s_1, s_2 such that $q_1 + q_2 \geq p_1 + s_1 \wedge q_2 \geq p_2 + s_2$. The two contexts are:

$$\Delta_1 = xs{:}\mathsf{T}^0(\mathsf{L}^0((s_1,s_2),B)) \quad \text{(for the recursive call to } pairs\text{)}$$

$$\Delta_2 = xs{:}\mathsf{T}^0(\mathsf{L}^0((p_1,p_2),B)) \quad \text{(for the call to } attach\text{)}$$

We can now type the recursive right-hand side e_2:

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ \Delta_1, \Delta_2 \left|\frac{2}{0}\right. \begin{array}{l}\texttt{let}\ \ f_1 = pairs\ xs;\\ \quad\ \ \ f_2 = attach\ x\ xs\\ \texttt{in}\ app'\ f_1\ f_2\end{array} : L_{Out} \tag{6}$$

The cost annotation on the turnstile correspond to the two uses of *let* for f_1 and f_2, as will be confirmed from the remaining derivation. We continue by typing the bound sub-expressions:

$$\Theta, \Delta_1 \left|\frac{0}{0}\right. pairs\ xs : \mathsf{L}^0((0,0), B\times B) \tag{7}$$

$$\Sigma, \Delta_2, x{:}B \left|\frac{0}{0}\right. attach\ x\ xs : \mathsf{L}^0(0, B\times B) \tag{8}$$

Judgments (7) and (8) follow immediately from Var and App. Note that, while the annotations on the turnstile are zero, the uses of App impose constraints on the annotations in Δ_1 and Δ_2: $p_1 = 3$, $p_2 = 0, s_1 = q_1$ and $s_2 = q_2$. It remains to type the inner expression:

$$\Delta_2, \Gamma, \Theta, f_1{:}\mathsf{T}^0(L_{Out}) \left|\frac{1}{0}\right. \texttt{let}\ f_2 = attach\ x\ xs\ \texttt{in}\ app'\ f_1\ f_2{:}L_{Out} \tag{9}$$

This follows from the rules Var and App twice:

$$\Gamma, f_1{:}\mathsf{T}^0(L_{Out}), f_2{:}\mathsf{T}^0(\mathsf{L}^0(1, B\times B)) \left|\frac{0}{0}\right. app'\ f_1\ f_2{:}L_{Out} \tag{10}$$

With this detailed illustration it is easy to see where the constraints mentioned before come from. From (7), (8) and (9) we get $p_1 = 3$, $p_2 = 0, s_1 = q_1$ and $s_2 = q_2$. From (4) and (6) we get $q_1 \geq 2$. From (5) we get that $q_1 + q_2 = s_1 + p_1$ and $q_2 = s_2 + p_2$. These constraints admit the solution $p_1 = s_2 = q_2 = 3$, $s_1 = q_1 = 2, p_2, p = 0$, giving us the following typing:

$$pairs : \mathsf{T}^0(\mathsf{L}^0((2,3),B)) \xrightarrow{0} \mathsf{L}^0(0, B\times B)$$

This typing ensures that *pairs* can be applied to an input list l with potential $2 \times |l| + 3 \times \binom{|l|}{2}$ leaving no leftover potential. This corresponds to a quadratic cost bound of $2 \times n + 3 \times \binom{n}{2} + 0 = 2 \times n + \frac{3}{2} \times n \times (n-1)$ expressed as a function of the input list length $n = |l|$.

Example 5.2. In the previous derivation we choose zero annotations for the thunks in the list spine; this corresponds to deriving a cost bound for the case where the spine of the input list is fully evaluated. Let us now consider the case where the input list l is annotated with $\mathsf{L}^1((q_1,q_2),B)$, i.e., evaluating each list successive constructor costs 1.

Because of the rule Match, when we introduce the tail element of the list to our environment it will be associated with a unitary cost thunk. We can use the structural rule Prepay to pay for its thunk cost only once, rather than for each use, before using Share to duplicate it. Because the rule Prepay is structural, we could have chosen not to use it and the inference would still have obtained an acceptable but less precise type.

Again, we are going to illustrate the inference steps with more detail. Note that, again, we omit certain elements of the type context that are not needed for the derivation in question. The expression is divided into 3 sub-expressions as illustrated before.

As before we assume annotated type for the auxiliary functions:[2]

$$\Gamma = app' : \mathsf{T}^0(\mathsf{L}^0(0, B\times B)) \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^0(1, B\times B)) \xrightarrow{0} \mathsf{L}^0(0, B\times B)$$

$$\Sigma = attach : B \xrightarrow{0} \mathsf{T}^0(\mathsf{L}^1(4,B)) \xrightarrow{0} \mathsf{L}^0(1, B\times B)$$

let us derive a type for *pairs* as follows:

$$\Theta = pairs : \mathsf{T}^p(\underbrace{\mathsf{L}^1((q_1,q_2),B)}_{L_{In}}) \xrightarrow{a} \underbrace{\mathsf{L}^0((0,0), B\times B)}_{L_{Out}}$$

The derivation is very similar to the previous example. It is when we reach the point of sharing the potential of the list that the main difference appears.

$$\Gamma, \Sigma, \Theta,\ x{:}B,\ xs{:}\mathsf{T}^p(\mathsf{L}^1((q_1+q_2,q_2),B)) \left|\frac{q_1}{0}\right. e_2{:}L_{Out} \tag{11}$$

Because this time the list is associated with a unitary cost thunk rather than a 0 annotated thunk, if we applied the rule Share as before, that cost would be replicated for both lists, meaning that we would have to pay for both uses. To prevent this from happening, we use the structural rule Prepay right before we use Share. We can see how the lists that result from sharing end up associated with a 0 annotated thunk:

[2]Note that we need a slightly different annotation for the input list of *attach*.

$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^1(\mathsf{L}^1((q_1+q_2,q_2),B)) \left|\frac{3}{0}\right. e_2{:}L_{Out} \qquad \text{(Prepay)}$$

$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^1((q_1,q_2),B)) \left|\frac{2}{0}\right. e_2{:}L_{Out} \qquad \text{(Share)}$$

The use of Share creates the following condition:

$$\mathsf{T}^0(\mathsf{L}^1((q_1+q_2,q_2),B)) \curlyvee \{\mathsf{T}^0(\mathsf{L}^1((p_1,p_2),B)),\ \mathsf{T}^0(\mathsf{L}^1((s_1,s_2),B))\} \tag{12}$$

Note that, although the outermost thunks have been reduced by the use of Prepay, the list spine thunks still cost 1. This is because sharing distributes list potential but not thunk costs (See Fig. 6).

The remaining derivation is:

$$\Gamma, \Sigma, x{:}B,\ xs{:}\mathsf{T}^0(\mathsf{L}^1((p_1,p_2),B)),\ xs{:}\mathsf{T}^0(\mathsf{L}^1((s_1,s_2),B)) \left|\frac{2}{0}\right. e_2{:}L_{Out} \tag{13}$$

The main constraints that result from this derivation are very similar to the ones from the example above, with the exception of $p_1 = 4$ (because of the different type assumption for *attach*) and $q_1 \geq 3$ (because of the use of Prepay after (13)). These constraints can be solved by $p_1 = s_2 = q_2 = 4, s_1 = q_1 = 3, p_2 = 0, p = 0$, giving us the type

$$pairs : \mathsf{T}^0(\mathsf{L}^1((3,4),B)) \xrightarrow{0} \mathsf{L}^0(0, B \times B)$$

This type corresponds to a cost bound of $3 \times n + 4 \times \binom{n}{2} + 0 = 3 \times n + 2 \times n \times (n-1)$ for list of length n.

Comparing this result the bound obtained for the previous example, we note an over-estimation of the cost: we would expect paying only extra n units for evaluating a list spine of length n; instead the difference between the bounds is $3 \times n + 2 \times n \times (n-1) - (\frac{3}{2} \times n \times (n-1)) = n + \frac{1}{2} \times n \times (n-1)$.

The overestimation results from the sharing of the list tail xs between *pairs* and *attach*: the two uses do not account for the repeated evaluation of xs. Note, however, that simply changing the sharing rule to distribute the list spine costs, i.e. sharing $xs{:}\mathsf{T}^0(\mathsf{L}^1(\ldots, B))$ to $xs_1{:}\mathsf{T}^0(\mathsf{L}^0(\ldots, B))$ and $xs_2{:}\mathsf{T}^0(\mathsf{L}^1(\ldots, B))$ would, in general, be unsound because we may discard the variable xs_2 and use only xs_1, thus underestimating the cost.

6 FINAL REMARKS AND FURTHER WORK

In this paper, we present a first extension of amortised resource analysis for higher-order lazy functional programs from linear to polynomial bounds. We show how we combine main concepts from previous systems in order to reach this goal: the usage of thunk types and prepaying for lazy evaluation and the additive shift for polynomial potential. Although our type system has been successfully applied to some small examples. We are developing a prototype implementation, which we believe will be an advantage in the analysis of larger examples.

We do not have a formal proof of soundness yet, thus, an obvious next step would be to develop a soundness proof; previous work in [12] could be adapted to the polynomial potential case.

Another limitation of our analysis as presented here is the fact that it does not allow resource polymorphic recursion, i.e., recursive calls with different resource annotations; as in the strict setting, we expect that this will cause many programs that are not in tail-recursive form to fail to admit an annotated type [3, 7]. For example, if we consider our definition of *pairs* and change the order in which the arguments are sent to app', the inference of annotations eventually reaches some inconsistency. This problem was addressed by Hoffmann in the strict setting by using a cost-free resource metric that assigns zero costs for each evaluation step and extending the algorithmic type rules with resource polymorphic recursion. We believe that the same approach could be used in our system.

Example 5.2 illustrated a cost overestimation caused by duplication of thunk costs inside data structures. We leave investigating mitigations for this issue as future work.

REFERENCES

[1] Robert Atkey. 2010. Amortised resource analysis with separation logic. In *European Symposium on Programming*. Springer, 85–103.

[2] Jan Hoffmann. 2011. *Types with potential: Polynomial resource bounds via automatic amortized analysis.* epubli.

[3] Jan Hoffmann. 2011. *Types with potential: polynomial resource bounds via automatic amortized analysis.* Ph.D. Dissertation. Ludwig Maximilians University Munich.

[4] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 357–370.

[5] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 359–373.

[6] Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In *Asian Symposium on Programming Languages and Systems*. Springer, 172–187.

[7] Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polynomial potential. In *European Symposium on Programming*. Springer, 287–306.

[8] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 185–197.

[9] Martin Hofmann and Steffen Jost. 2006. Type-based amortised heap-space analysis. In *European Symposium on Programming*. Springer, 22–37.

[10] John Hughes and Chalmers Hogskola. 1999. Why Functional Programming Matters. (05 1999).

[11] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *ACM Sigplan Notices*, Vol. 45. ACM, 223–236.

[12] Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning* 59, 1 (2017), 87–120.

[13] John Launchbury. 1993. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 144–154.

[14] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. Springer.

[15] Chris Okasaki. 1999. *Purely functional data structures.* Cambridge University Press.

[16] Peter Sestoft. 1997. Deriving a lazy abstract machine. *Journal of Functional Programming* 7, 3 (1997), 231–264.

[17] Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*. 165–176.

[18] Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318.

[19] Pedro Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. 2015. Type-Based Allocation Analysis for Co-recursion in Lazy Functional Languages. In *Programming Languages and Systems*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 787–811.

Building an Integrated Development Environment (IDE) on top of a Build System

The tale of a Haskell IDE

Neil Mitchell
Facebook
ndmitchell@gmail.com

Moritz Kiefer
Digital Asset
moritz.kiefer@purelyfunctional.org

Pepe Iborra
Facebook
pepeiborra@gmail.com

Luke Lau
Trinity College Dublin
luke_lau@icloud.com

Zubin Duggal
Chennai Mathematical Institute
zubin.duggal@gmail.com

Hannes Siebenhandl
TU Wien
hannes.siebenhandl@posteo.net

Matthew Pickering
University of Bristol
matthewtpickering@gmail.com

Alan Zimmerman
Facebook
alan.zimm@gmail.com

Abstract

When developing a Haskell IDE we hit upon an idea – why not base an IDE on an build system? In this paper we'll explain how to go from that idea to a usable IDE, including the difficulties imposed by reusing a build system, and those imposed by technical details specific to Haskell. Our design has been successful, and hopefully provides a blue-print for others writing IDEs.

1 Introduction

Writing an IDE (Integrated Development Environment) is not as easy as it looks. While there are thousands of papers and university lectures on how to write a compiler, there is much less written about IDEs ([1] is one of the exceptions). We embarked on a project to write a Haskell IDE (originally for the GHC-based DAML language [4]), but our first few designs failed. Eventually, we arrived at a design where the heavy-lifting of the IDE was performed by a *build system*. That idea turned out to be the turning point, and the subject of this paper.

Over the past two years we have continued development and found that the ideas behind a build system are both applicable and natural for an IDE. The result is available as a project named *ghcide*[1], which is then integrated into the *Haskell Language Server*[2].

In this paper we outline the core of our IDE §2, how it is fleshed out into an IDE component §3, and then how we build a complete IDE around it using plugins §4. We look at where the build system both helps and hurts §5. We then look at the ongoing and future work §6 before concluding §7.

[1]https://github.com/digital-asset/Ghcide
[2]https://github.com/haskell/haskell-language-server

IFL'20, September 2–4, 2020, Online
2020.

2 Design

In this section we show how to implement an IDE on top of a build system. First we look at what an IDE provides, then what a build system provides, followed by how to combine the two.

2.1 Features on an IDE

To design an IDE, it is worth first reflecting on what features an IDE provides. In our view, the primary features of an IDE can be grouped into three capabilities, in order of priority:

Errors/warnings The main benefit of an IDE is to get immediate feedback as the user types. That involves producing errors/warnings on every keystroke. In a language such as Haskell, that involves running the parser and type checker on every keystroke.

Hover/goto definition The next most important feature is the ability to interrogate the code in front of you. Ways to do that include hovering over an identifier to see its type, and clicking on an identifier to jump to its definition. In a language like Haskell, these features require performing name resolution.

Find references Finally, the last feature is the ability to find where a symbol is used. This feature requires an understanding of all the code, and the ability to index outward.

The design of Haskell is such that to type check a module requires to get its contents, parse it, resolve the imports, type check the imports, and only then type check the module itself. If one of the imports changes, then any module importing it must also be rechecked. That process can happen once per user character press, so is repeated incredibly frequently.

Given the main value of an IDE is the presence/absence of errors, the way such errors are processed should be heavily optimised. In particular, it is important to hide/show an error as soon as possible. Furthermore, errors should persist until they have been corrected.

2.2 Features of a build system

The GHC API is a Haskell API for compiling Haskell files, using the same machinery as the GHC compiler [17]. Therefore, to integrate smoothly with the GHC API, it is important to choose a build system that can be used as a Haskell library. Furthermore, since the build graph is incredibly dynamic, potentially changing on every key stroke, it is important to be a monadic build system [12, §3.5]. Given those constraints, and the presence of an author in common, we chose to use Shake [11].

The Shake build system is fully featured, including parallelism, incremental evaluation and monadic dependencies. While it has APIs to make file-based operations easy, it is flexible enough to allow defining new types of rules and dependencies which do not use files. At its heart, Shake is a key/value mapping, for many types of key, where the type of the value is determined by the type of the key, and the resulting value may depend on many other keys.

2.3 An IDE on a build system

Given the IDE and build system features described above, there are some very natural combinations. The monadic dependencies are a perfect fit. Incremental evaluation and parallelism provide good performance. But there are a number of points of divergence which we discuss and overcome below.

2.3.1 Restarting. A Shake build can be interrupted at any point, and we take the approach that whenever a file changes, e.g. on every keystroke, we interrupt the running Shake build and start a fresh one. While that approach is delightfully simple, it has some problems in practice, and is a significant divergence from the way Shake normally works.

Firstly, we interrupt using asynchronous exceptions [14]. Lots of Haskell code isn't properly designed to deal with such exceptions. We had to fix a number of bugs in Shake and other libraries and are fairly certain some still remain.

Secondly, when interrupting a build, some things might be in progress. If type checking a big module takes 10 seconds, and the user presses the key every 1 second, it will keep aborting 1 second through and never complete. In practice, interrupting hasn't been a significant hurdle, although we discuss possible remedies in §5.3.

2.3.2 Errors. In normal Shake execution an error is thrown as an exception which aborts the build. However, for an IDE, errors are a common and expected state. Therefore, we want to make errors first class values. Concretely, instead of the result of a rule such as type checking being a type checked module, we use:

```
([Diagnostic], Maybe TcModuleResult)
```

Where `TcModuleResult` is the type checked module result as provided by the GHC API. The list of diagnostics stores errors and warnings which can occur even if type checking succeeded. The second component represents the result of the rule with `Nothing` meaning that the rule could not be computed either because its dependencies failed, or because it failed itself.

In addition, when an error occurs, it is important to track which file it belongs to, and to determine when the error goes away. To achieve that, we make all Shake keys be a pair of a phase-specific type alongside a `FilePath`. So a type-checked value is indexed by:

```
(TypeCheck, FilePath)
```

where `TypeCheck` is isomorphic to `()`.

The second component of the key determines the file the error will be associated with in the IDE. We cache the error per `FilePath` and phase, and when a `TypeCheck` phase for a given file completes, we overwrite any previous type checking errors that file may have had. By doing so, we can keep an up-to-date copy of what errors are known to exist in a file, and know when they have been resolved.

2.3.3 Performance. Shake runs rules in a random order [11, §4.3.2]. But as rule authors, we know that some steps like type checking are expensive, while others like finding imports (and thus parsing) cause the graph to fan out. Using that knowledge, we can deprioritise type checking to reduce latency and make better use of multicore machines. To enable that deprioritisation, we added a `reschedule` function to Shake, that reschedules a task with a lower priority.

2.3.4 Memory only. Shake usually operates as a traditional build system, working with files and commands. As standard, it stores its central key/value map in a journal on disk, and rereads it afresh on each run. That caused two problems:

1. Reading the journal each time can take as long as 0.1s. While that is nearly nothing for a traditional build, for an IDE that is excessive. We solved this problem by adding a `Database` module to Shake that retains the key/value map in memory.
2. Shake serialises all keys and values into the journal, so those types must be serializable. While adding a memory-only journal was feasible, removing the serialisation constraints and eliminating all serialisation would require more significant modifications. Therefore we wrote serialisation methods for all the keys. However, values are often GHC types, and contain embedded types such as `IORef`, making it difficult to serialise them. To avoid the need to use value serialisation, we created a shadow map containing the actual values, and stored dummy values in the Shake map.

The design of Shake is for keys to accumulate and never be removed. However, as the IDE is very dynamic, the relevant set of keys may change regularly. Fortunately, the Shake portion of the key/value is small enough not to worry about, but the shadow map should have unreachable nodes removed in a garbage-collection like process (see §5.6).

2.4 Layering on top of Shake

In order to simplify the design of the rest of the system, we built a layer on top of Shake, which provides the shadow map, the keys with file names, the values with pairs and diagnostics etc. By building upon this layer we get an interface that more closely matches the needs of an IDE. Using this layer, we can define the type checking portion of the IDE as:

```
type instance RuleResult TypeCheck =
    TcModuleResult

typeCheck = define $ \TypeCheck file -> do
    pm <- use_ GetParsedModule file
    deps <- use_ GetDependencies file
    tms <- uses_ TypeCheck $
        transitiveModuleDeps deps
    session <- useNoFile_ GhcSession
    liftIO $ typecheckModule session tms pm
```

Reading this code, we use the `RuleResult` type family [2] to declare that the `TypeCheck` phase returns a value of type `TcModuleResult`. We then define a rule `typeCheck` which implements the `TypeCheck` phase. The actual rule itself is declared with `define`, taking the phase and the filename. First, it gets the parsed module, then the dependencies of the parsed module, then the type checked results for the transitive dependencies. It then uses that information along with the GHC API session to call a function `typecheckModule`. To make this code work cleanly, there are a few key functions we build upon:

- We use `define` to define types of rule, taking the phase and the filename to operate on.
- We define `use` and `uses` which take a phase and a file (or lists thereof) and return the result.
- On top of `use` we define `use_` which raises an exception if the requested rule failed. In `define` we catch that exception and switch it for `([], Nothing)` to indicate that a dependency has failed.
- Some items don't have a file associated with them, e.g. there is exactly one GHC session, so we have `useNoFile` (and the underscore variation) for these.
- Finally, the GHC API can be quite complex. There is a GHC provided `typecheckModule`, but it throws exceptions on error, prints warnings to a log, returns too much information for our purposes and operates in the GHC monad. Therefore, we wrap it into a "pure" API (where the output is based on the inputs), with the signature:

```
typecheckModule
    :: HscEnv
    -> [TcModuleResult]
    -> ParsedModule
    -> IO ([Diagnostic], Maybe TcModuleResult)
```

2.5 Error tolerance

An IDE needs to be be tolerant to errors in the source code, and must continue to aid the developer while the source code is incomplete and does not parse or typecheck, as this state is the default while source code it is being edited. We employ a variety of mechanisms to achieve this goal:

- GHC's `-fdefer-type-errors` and `-fdefer-out-of-scope-variables` flags turn type errors and out of scope variable errors into warnings, and let it proceed to typecheck and return usable artifacts to the IDE. This flag leads to GHC downgrading the errors produced to warnings, so we must promote such warnings back into errors before reporting them to the user.
- If the code still fails to typecheck (for example due to a parse error, or multiple declarations of a function etc.), we still need to be able to return results to the user. Therefore, we define the `useWithStale` function to get the most recent, *successfully* computed value of a key, even if it was for a older version of the source.
- The function `useWithStale` has return type `Maybe (v, PositionMapping)` where `v` is the return type of the rule, and the type `PositionMapping` is a set of functions that help us convert source locations in the current version of a document back into the version of the document for which the rule was last computed successfully, and vice versa. For example, if the user inserts a line at the beginning of the file, the reported source locations of all the definitions in the file need to be moved one line down. Similarly, when we are querying the earlier version of the document for the symbol under a cursor, we must remember to shift the position of the cursor up by one line. We maintain this mapping between source locations for all versions of a file for which we have artifacts older than the current version of the document.

2.6 Responsiveness

An IDE needs to return results quickly in order to be helpful. However, we found that running all the Shake rules to check for freshness and recompute results on every single request was not satisfactory with regards to IDE responsiveness. This problem was particularly evident for completions, which need to show up quickly in order to be useful. However, each keystroke made by a user invalidates the Shake store, which needs to be recomputed.

For this reason, we added an alternative mechanism to directly query the computed store of results without rerunning all the Shake rules. We defined a function `useWithStaleFast` for this purpose, with a signature like `useWithStale`. This function first asynchronously fires a request to refresh the Shake store. Immediately afterwards, it checks to see if the result has already been computed in the store. If it has, it immediately returns this result, along with the `PositionMapping` for the version of the document this result was computed for, as described in the previous section. If the result has never been computed before, it waits for recomputation request to Shake to finish, and then returns its result.

This technique provides a significant improvement in the responsiveness of requests like hovering, go to definition, and completions, in return for a small sacrifice in correctness.

3 Integration

To go from the core described in §2 to a fully working IDE requires integrating with lots of other projects. In this section we outline some of the most important.

3.1 The GHC API

The GHC API provides access to the internals of GHC and was not originally designed as a public API. This history leads to some design choices where `IORef` values (mutable references) hide alongside huge blobs of state (e.g. `HscEnv`, `DynFlags`). With careful investigation, most pieces can be turned into suitable building blocks for an IDE. Over the past few years the Haskell IDE Engine [18] project has been working with GHC to upstream patches to make more functions take in-memory buffers rather than files, which has been very helpful.

One potentially useful part of the GHC API is the "downsweep" mechanism. In order to find dependencies, GHC first parses the import statements, then sweeps downwards, adding more modules into a dependency graph. The result of downsweep is a static graph indicating how modules are related. Unfortunately, this process is not very incremental, operating on all modules at once. If it fails, the result is a failure rather than a partial success. This whole-graph approach makes it unsuitable for use in an IDE. Therefore, we rewrote the downsweep process in terms of incremental dependencies. The disadvantage is that many things like preprocessing and plugins are also handled by the downsweep, so they had to be dealt with specially. We hope to upstream our incremental downsweep into GHC at some point in the future.

3.1.1 Separate type-checking. In order to achieve good performance in large projects, it's important to cache the results of type-checking individual modules and to avoid repeating the work the next time they are needed, or when loading them for the first time after restarting the IDE. Our IDE leverages two features of GHC that, together, enable fully separate typechecking while preserving all the IDE features mentioned in §2.1

1. Interface files (so called `.hi` files) are a by-product of module compilation and have been in GHC since the authors can remember. They contain a plethora of information about the associated module. When asking the GHC API to type-check a module M that depends on a module D, one can load a previously obtained `D.hi` interface file instead of type-checking D, which is much more efficient and avoids duplicating work. Using this file is only correct when D hasn't changed since `D.hi` was produced, but happily GHC performs recompilation checks and complains when this assumption isn't met.
2. Extended interface files (so called `.hie` files) are also a by-product of module compilation, recently added to GHC in version 8.8. Extended interface files record the full details of the type-checked AST of the associated module, enabling tools to provide hover and go-to reference functionality without the need to use the GHC API at all. Our IDE mines these files to provide hover and go-to reference for modules that have been loaded from an interface file, and thus not typechecked in the current session.

3.2 Setting up a GHC Session

When using the GHC API, the first challenge is to create a working GHC session. This involves setting the correct `DynFlags` needed to load and type-check the files in a project. These typically include compilation flags like include paths and what extensions should be enabled, but they also include information about package dependencies, which need to be built beforehand and registered with *ghc-pkg*. Furthermore, these details are all entirely dependent on the build tool: The flags that *Stack* passes to GHC to build a project will be different from what *Cabal* passes, because each builds and stores package dependencies in different locations and package databases.

Because this whole process is so specific to the build tool, setting up the environment and extracting the flags for a Haskell project has traditionally been a very fickle process. A new library *hie-bios* [19] was developed to tackle this problem, consolidating efforts into one place. The name comes from the idea that it acts as the first point of entry for setting up the GHC session, much like a system BIOS is the first point of entry for hardware on a computer. Its philosophy is to delegate the responsibility of setting up a session entirely to the build tool — whether that be Cabal, Stack, Hadrian [13], Bazel [7] or any other build system that invokes GHC.

hie-bios is based around the idea of *cradles* which describe a specific way to set up an environment through a specific build tool. For instance, hie-bios comes with cradles for Stack projects, Cabal projects and standalone Haskell files, but it can interface with other build tools by invoking them and reading the arguments to GHC via `stdout`. These cradles are essentially functions that call the necessary functions on the build tool to build and register any dependencies, and return the flags that would be passed to GHC for a specific file or component. For Cabal and Stack, this information is currently obtained through the `repl` commands. The cradle that should be used for a specific project can be inferred through the presence of build-tool specific files like `cabal.project` and `stack.yaml`. For more complex projects which comprise of multiple directories and packages, the cradles used can be explicitly configured through a `hie.yaml` file to describe exactly what build tool should be used, and what component should be loaded for the GHC session, for each file or directory.

3.3 Handling multiple components in one session

Haskell projects are often separated into multiple packages, and when using Cabal [9], a package consists of multiple components. These components might be a library, executable, test-suite or a benchmark. Each of the components might require a different set of compilation options and they might depend on each other. Ideally, we want to be able to use the IDE on all components at the same time, so that features like goto-definition and refactoring work sensibly. Consequentially, using the IDE on a big project with multiple subprojects should work as expected.

However, the GHC API is designed to only handle a single component at a time. This limitation is hard-coded in multiple locations within the GHC code-base. As it can only handle a single component, GHC only checks whether any modules have changed for this single component, assumes that any dependencies are stored on disk and won't change during the compilation. However, in our dynamic usage, local dependencies might change!

The same problematic behaviour can be found in everyday usage of an interactive GHC session. Loading an executable into the interactive session, and applying changes to the library the executable depends on, will not cause any recompilation in the interactive session. For any of the changes to take effect, the user needs to entirely shut-down the interactive GHC session and reload it. In the IDE context, if the library component changes the executable component will not be recompiled, as GHC does not notice that a dependency has changed and diagnostics for the executable component become stale. To work around these limitations, we handle components in-memory and modify the GHC session adhoc. Whenever the IDE encounters a new component, we calculate the global module graph of all components that are in-memory. With this graph, we can handle module updates ourselves and load multiple components in a single GHC session.

3.4 Language Server Protocol (LSP)

In order to actually work as an IDE, we need to communicate with a text editor. We use the Language Server Protocol (LSP) [10] for this, which is supported by most popular text editors and clients, either natively or through plugins and extensions. LSP is a JSON-RPC based protocol that works by sending messages between the editor and a *language server*. Messages are either *requests*, which expect a *response* to be sent back in reply, or *notifications* which do not expect any. For example, the editor (client) might send notifications that some file has been updated, or requests for code completions to display to the user at a given source location. The language server may then send back responses answering those requests and notifications that provide diagnostics.

To bridge the gap between messages and the build graph, *ghcide* deals with the types of incoming messages differently:

- When a notification arrives from LSP that a document has been edited, we modify the nodes that have changed, e.g., the content of the modified files, and immediately start a rebuild in order to produce diagnostics.
- When a request for some specific language feature arrives, we append a target to the ongoing build asking for whatever information is required to answer that request. For example, if a hover request arrives, we ask for the set of type-checked spans corresponding to that file. Importantly, this does not cause a rebuild.
- When the graph computes that the diagnostics for a particular file have changed, we send a notification to the client to show updated diagnostics.

3.5 Testing

Our IDE implements a large part of the LSP specification, and has to operate on a large range of possible projects with all sorts of edge cases. We protect against regressions from these edge cases with a functional test suite built upon *lsp-test*, a testing framework for LSP servers. *lsp-test* acts as a client which language servers can talk to, simulating a session from start to finish at the transport level. The library allows tests to specify what messages the client should send to the server, and what messages should be received back from the server.

Functional testing turns out to be rather important in this scenario as the RPC-based protocol is in practice, highly asynchronous, something which unit tests often fail to account for. Clients can make multiple requests in flight and Shake runs multiple worker threads, so the order in which messages are delivered is non-deterministic. Because of this fact, a typical test might look like:

```
test :: IO ()
test = runSession "ghcide" fullCaps "test" $ do
  doc <- openDoc "Foo.hs" "haskell"
  skipMany anyNotification
  let prms = DocumentSymbolParams doc
  rsp <- request TextDocumentDocumentSymbol prms
  liftIO $ rsp ^. result `shouldNotSatisfy` null
```

In this session, *lsp-test* tells *ghcide* to open up a document, and then ignore any notifications it may send with `skipMany` `anyNotification`. A session is actually a parser combinator [8] operating on incoming messages under the hood, which allows the expected messages from the server to be specified in a flexible way that can handle non-deterministic ordering. It then sends a request to the server to retrieve the symbols in a document, waits for the response and finally makes some assertion about the response.

An additional benefit of having testing at the transport level is that we can reuse much of the test suite in IDEs building on top of Ghcide for free, since we only need to swap out what server the tests should be run on. *lsp-test* is also used not only for testing, but also for automating benchmarks (See §5.7).

4 Plugins

The IDE described in §3 corresponds to the Haskell library Ghcide, which is currently used in at least four different roles:

- With a thin wrapper as a stand alone IDE for GHC.
- As the engine powering the IDE for DAML.
- As the foundation of a compiler for DAML.
- As the GHC layer for a more full-featured Haskell IDE (Haskell Language Server, HLS).

The key to supporting all these use cases is a rich plugin mechanism.

4.1 LSP extensibility

The Language Server Protocol is extensible, in that it provides sets of messages that provide a (sub) protocol for delivering IDE features. Examples include:

- Context aware code completion
- Hover information. This is context-specific information provided as a separate floating window based on the cursor position. Additional analysis sources should be able to seamlessly add to the set of information provided.
- Diagnostics. The GHC compiler provides warnings and errors. It should be possible to supplement these with any other information from a different analysis tool. Such as `hlint`, or `liquid haskell`.
- Code Actions. These are context-specific actions that are provided based on the current cursor location. Typical uses are to provide actions to fix simple compiler errors reported, e.g. adding a missing language pragma or import. But they can also provide more advanced functionality, like suggesting refactorings of the code.
- Code Lenses. These operate on the whole file, and offer a way to display annotations to a given piece of code, which can optionally be clicked on to trigger a code action to perform some function. In ghcide these are used to display inferred type signatures for functions, and allow you to add them to the code with one click.

The standardised messaging allows uniform processing on the client side for features, but also means new features should be easy to add on the server side.

4.2 Ghcide plugins

Internally, *ghcide* is two things, a rule engine, and an interface to the Language Server Protocol (§3.4).

So to be extensible, there must be a way to add rules to the rule database, and additional message handlers to the LSP message processing.

A plugin in *ghcide* is thus defined as a data structure having `Rules` and `PartialHandlers`

A *Monoid* class is provided for these, meaning they can be freely combined. There is one caveat, in that order matters for the `PartialHandlers`, so the last message handler for a particular message wins.

In practical terms the plugin uses these features as follows

- It provides rules to generate additional artefacts and add them to the Shake graph if needed. For most plugins this is unnecessary, as the full output of the underlying compiler is available. Typical use-cases for this would be to trigger additional processing for diagnostics, such as for `hlint` or similar external analysis. Note that care must be taken with adding rules, as it affects both memory usage and processing time.
- It provides handlers for the specific LSP messages needed to provide its feature(s).

This is a fairly low-level capability, but it is sufficient to provide the plugins built in to *ghcide*, and serve as a building block for the *Haskell Language Server*.

4.3 Haskell Language Server plugins

The *Haskell Language Server* makes use of *ghcide* as its IDE engine, relying on it to provide fast, accurate, up to date information on the project being developed by a user.

Where *ghcide* is intended to do one thing well, *Haskell Language Server* is targeted at being the "batteries included" starter IDE for any Haskell user. *HLS* is the family car where *ghcide* is the sports model.

We will describe here its approach to plugins.

Firstly, a design goal for *HLS* is to be able mix and match any set of plugins. The current version (0.3) has a set built in, but the road map calls for the ability to provide a tiny custom `Main` module that imports a set of plugins, puts them in a structure and passes them in to the existing main programme.

To enable this, it has a plugin descriptor which looks like

```
data PluginDescriptor =
  PluginDescriptor
  { pluginId
      :: !PluginId
  , pluginRules
      :: !(Rules ())
  , pluginCommands
```

```
      :: ![PluginCommand]
  , pluginCodeActionProvider
      :: !(Maybe CodeActionProvider)
  , pluginCodeLensProvider
      :: !(Maybe CodeLensProvider)
  , pluginHoverProvider
      :: !(Maybe HoverProvider)
  ...
  }
```

The pluginId is used to make sure that if more than one plugin provides a *Code Action* with the same command name, HLS can choose the right one to process it.

The [PluginCommand] is a possibly empty list of commands that can be invoked in code actions.

The rest of the fields can be filled in with just the capabilities the plugin provides.

So a plugin providing additional hover information based on analysis of the existing GHC output would only fill in the pluginId and pluginHoverProvider fields, leaving the rest at their defaults.

4.4 Haskell Language Server plugin processing

The *HLS* engine converts the *HLS*-specific plugin structures to a single *ghcide* plugin.

It simply combines the Rules monoidally, but does some specific processing for the other message handlers.

The key difference is that *HLS* processes the entire set of PluginHandlers at once, rather than using the pairwise mappend operation.

This means that when a hover request comes in, it can call *all* the hover providers from all the configured plugins, combine the results and send a single combined reply to the original request.

The same technique is used as appropriate for each of the message handlers.

5 Evaluation

We released our IDE and it has become an important part of the Haskell tools ecosystem. When it works, the IDE provides fast feedback with increasingly more features by the day. Building on top of a build system gave us a suitable foundation for expressing the right things easily. Building on top of Shake gave us a well tested and battle hardened library with lots of additional features we didn't use, but were able to rapidly experiment with. However, the interesting part of the evaluation is what *doesn't* work.

5.1 Asynchronous exceptions are hard

Shake had been designed to deal with asynchronous exceptions, and had a full test suite to show it worked with them. However, in practice, we keep coming up with new problems that bite in corner cases. Programming defensively with asynchronous exceptions is made far harder by the fact that even finally constructions can actually be aborted, as there are two levels of exception interrupt. We suspect that in time we'll learn enough tricks to solve all the bugs, but it's a very error prone approach, and one where Haskell's historically strong static checks are non-existent.

5.2 Session setup

The majority of issues reported by users are come from the failure to setup a valid GHC session — this is the first port of call for ghcide, so if this step fails then every other feature will fail. The diversity of project setups in the wild is astounding, and without explicit configuration hie-bios struggles to detect the correct cradles for the correct files (see §3.2). It is a difficult problem, and the plethora of Haskell build tools out there only exacerbates it further. Tools such as Nix [5] are especially common and problematic.

Work is currently underway to push the effort upstream from hie-bios into the build tools themselves, to expose more information and provide a more reliable interface for setting up sessions: Recently a show-build-info command has been developed for *cabal-install* that builds package dependencies and returns information about how Cabal would build the project in a machine readable format.

In addition, some projects require more than one GHC session to load all modules — we are still experimenting with solutions for this problem.

5.3 Cancellation

While regularly cancelling builds doesn't seem to be a problem in practice, it would be better if the partial work started before a cancellation could be resumed. A solution like FRP [6] might offer a better foundation, but we were unable to identify a suitable existing library for Haskell (most cannot deal with parallelism). Alternatively, a build system based on a model of continuous change rather than batched restarts might be another option. We expect the current solution using Shake to be sufficient for at least another year, but not another decade.

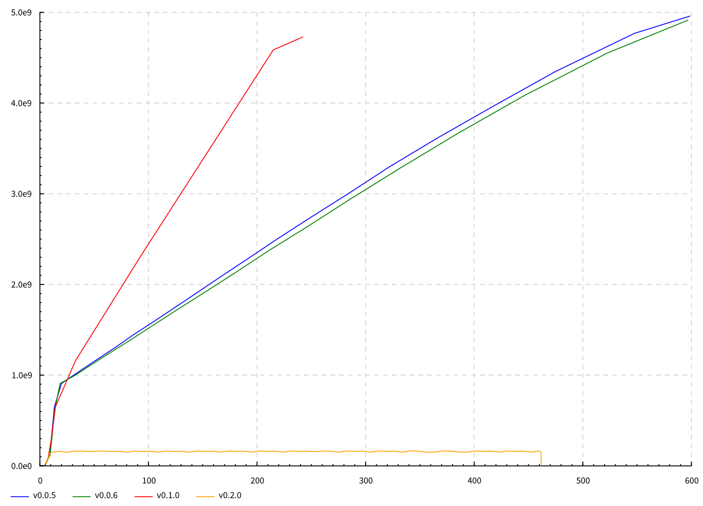
5.4 Runtime evaluation

Some features of Haskell involve compiling and running code at runtime. One such culprit is Template Haskell [15]. The mechanisms within GHC for runtime evaluation are improving with every release, but still cause many problems.

5.5 References

As stated in §2.1, an IDE offers three fundamental features – diagnostics, hover/goto-definition and find references. Our IDE offers the first two, but not the third. If the IDE was aware of the roots of the project (e.g. the Main module for a program) we could use the graph to build up a list of references. However, we have not yet done so.

Figure 1. Heap usage over successive versions of Ghcide



5.6 Garbage collection

Currently, once a file has been opened, it remains in memory indefinitely. Frustratingly, if a temporary file with errors is opened, those errors will remain in the users diagnostics pane even if the file is shut. It is possible to clean up such references using a pass akin to garbage collection, removing modules not reachable from currently open files. We have implemented that feature for the DAML Language IDE [4], but not yet for the Haskell IDE.

5.7 Memory leaks

A recurring complaint of our users is the amount of memory used. Indeed one of the authors witnessed >70GB resident set sizes on multiple occasions on medium/large codebases. This memory consumption was not only ridiculously inefficient but also a source of severe responsiveness issues while waiting [3] for the garbage collector to waddle through the mud of an oversized heap.

Our initial efforts focused on architectural improvements like separate type-checking and a frugal discipline on what gets stored in the Shake graph. But it wasn't until a laziness related space leak was identified and fixed in the Haskell library `unordered-containers` library that we observed a material improvement. Figure 1 shows the heap usage of a replayed Ghcide session over time, for various versions of Ghcide, where we can see that for versions prior to 0.2.0 it would grow linearly and without bounds until running out of memory.

Given how much effort and luck it took to clear out the space leak, and the lack of methods or tooling for diagnosing leaks induced by laziness, we have installed mechanisms to prevent new leaks from going undetected:

1. A benchmark suite that replays various scenarios while collecting space and time statistics.
2. An experiment tool that runs benchmarks for a set of commits and compares the results, highlighting regressions.

Monitoring and preventing performance regressions is always a good practice, but absolutely essential when using a lazy language due to the rather unpredictable dynamic semantics.

6 Future work

Since the IDE was released, a number of volunteer contributors have been developing and extending the project in numerous directions. In addition, some teams in commercial companies have starting adopting the IDE for their projects. Some of the items listed in this section are currently under active development, while other are more aspirational in nature.

6.1 hiedb

`hiedb`[4] is a tool to index and query GHC extended interface (`.hie`) files. It reads `.hie` files and extracts all sorts of useful information from them, such as references to names and types, the definition and declaration spans, documentation and types of top level symbols, storing it in a SQLite database for fast and easy querying.

Integrating hiedb with Ghcide has many obvious benefits. For example, we can finally add support for "find references", as well as allowing you to search across all the symbols defined in your project.

In addition, the hiedb database serves as an effective way to persist information across Ghcide runs, allowing greater responsiveness, ease of use and flexibility to queries. hiedb works well for saving information that is not local to a particular file, like definitions, documentation, types of exported symbols and so on.

A branch of Ghcide integrating it with hiedb is under active development.

Ghcide acts as an indexing service for hiedb, generating .hi and .hie files which are indexed and saved in the database, available for all future queries, even across restarts. A local cache of .hie files/type-checked modules is maintained on top of this to answer queries for the files the user is currently editing, while non-local information about other files in the project is accessed through the database.

6.2 Replacing Shake

As we observed in §5, a build system is a good fit for an IDE, but not a perfect fit. Using the abstractions we built for our IDE, we have experimented with replacing Shake

[3]By default the GHC runtime will trigger a major collection after 0.3 seconds of idleness; thankfully this can be customized along with many other GC settings.

[4]https://github.com/wz1000/HieDb

for a library based on Functional Reactive Programming [6], specifically the Haskell library `Reflex`. Initial results are promising in some dimensions (seems to be lower overhead), but lacking (no parallelism). We continue to experiment in this space.

6.3 Multiple Home Unit in GHC

As described in §3.3, there are limitations in the GHC API that force us to handle the module graph in-memory. This is error-prone and complicates the IDE quite a lot. Moving this code into GHC improves the performance and simplify support for multiple GHC versions. Moreover, it might prove useful for follow up contributions to enable GHC to work as a build server. As such, it can compile multiple units in parallel without being restarted, while using less memory in the process.

7 Conclusion

We implemented an IDE for Haskell on top of the build system Shake. The result is an effective IDE, with a clean architectural design, which has been easy to extend and adapt. We consider both the project and the design a success.

The idea of using a build system to drive a compiler is becoming more widespread, e.g. in Stratego [16] and experiments with replacing GHC `--make` [20]. By going one step further, we can build the entire IDE on top of a build system. The closest other IDE following a similar pattern is the Rust Analyser IDE [3], which uses a custom recomputation library, not dissimilar to a build system. Build systems offer a powerful abstraction whose use in the compiler/IDE space is likely to become increasingly prevalent.

Acknowledgments

Thanks to everyone who contributed to the IDE. The list is long, but includes the Digital Asset team (who did the initial development), the Haskell IDE engine team (who improved the GHC API and lead the trail), and the hie-bios team (who made it feasible to target real Haskell projects). In addition, many open source contributors have stepped up with bug reports and significant improvements. Truly a team effort.

References

[1] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: a language server for OCaml (experience report). *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–15.

[2] Manuel MT Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–13.

[3] Rust IDE Contributors. 2020. Three Architectures for a Responsive IDE. (20 July 2020). https://rust-analyzer.github.io/blog/2020/07/20/three-architectures-for-responsive-ide.html.

[4] Digital Asset. 2020. DAML Programming Language. (2020). https://www.daml.com/.

[5] Eelco Dolstra, Merijn De Jonge, Eelco Visser, et al. 2004. Nix: A Safe and Policy-Free System for Software Deployment. In *LISA*, Vol. 4. 79–92.

[6] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *International Conference on Functional Programming*.

[7] Google. 2020. Bazel. (2020). http://bazel.io/.

[8] Graham Hutton and Erik Meijer. 1996. Monadic Parser Combinators.

[9] Isaac Jones. 2005. The Haskell Cabal: A Common Architecture for Building Applications and Libraries, Marko van Eekelen (Ed.). 340–354.

[10] Microsoft. 2020. Language Server Protocol. (2020). https://microsoft.github.io/language-server-protocol/.

[11] Neil Mitchell. 2012. Shake before building: Replacing Make with Haskell. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 55–66.

[12] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proceedings ACM Programing Languages* 2, Article 79, 79:1–79:29 pages.

[13] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive Make Considered Harmful - Build Systems at Scale. In *Haskell 2016: Proceedings of the ACM SIGPLAN symposium on Haskell*. 55–66.

[14] Simon Peyton Jones. 2001. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*. IOS Press, 47–96.

[15] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 Haskell Workshop, Pittsburgh*. 1–16.

[16] Jeff Smits, Gabriël D. P. Konat, and Eelco Visser. 2020. Constructing Hybrid Incremental Compilers for Cross-Module Extensibility with an Internal Build System. *CoRR* (2020). arXiv:2002.06183

[17] The GHC Team. 2020. The GHC Compiler, Version 8.8.3. (2020). https://www.haskell.org/ghc/.

[18] The haskell-ide-engine Team. 2020. haskell-ide-engine. (2020). https://github.com/haskell/haskell-ide-engine.

[19] The hie-bios Team. 2020. hie-bios. (2020). https://github.com/mpickering/hie-bios.

[20] Edward Yang. 2016. ghc –make reimplemented with Shake. (2016). https://github.com/ezyang/ghc-shake.

Functional Programming Application for Digital Synthesis Implementation

Evan Sitt, Xiaotian Su, Beka Grdzelishvili, Zurab Tsinadze, Zongpu Xie
Hossameldin Abdin, Giorgi Botkoveli, Nikola Cenikj, Tringa Sylaj, Viktória Zsók
{sitt.evan,suxiaotian31,bekagrdzelishvili0,zukatsinadze,szumixie}@gmail.com
{hossamabdeen17,botko.gio,nicola.cenic,tringasylaj}@gmail.com,zsv@inf.elte.hu
Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
Budapest, Hungary

Abstract

Digital synthesis is a cross-discipline application used in fields such as music, telecommunication, and others. Digital synthesis involving multiple tracks, as well as parallel post-processes, lends itself naturally to the functional programming paradigm. The paper demonstrates this by creating a fully functional, cross-platform, standalone synthesizer application framework implemented in a pure lazy functional language. The application handles MIDI input and produces WAV output played by any multimedia player. Therefore, it can serve as a preprocessor for users who intend to create digital signals before transcribing them into digital or physical media. Sufficient background and implementation techniques were explored for building software solutions for digital synthesis in functional programming. We demonstrate that functional programming concepts such as lazy evaluation using arrays are efficient for processing digital audio signals, and they are intuitive for working with music, using programming language as *Clean*.

Keywords: Functional Programming, Digital Synthesis, Waveforms, MIDI, WAV

1 Introduction

Digital synthesis is a *Digital Signal Processing (DSP)* technique for creating musical sounds. In contrast to *analog synthesizers*, digital synthesis processes discrete bit data to replicate and recreate a continuous waveform. The digital signal processing techniques used are relevant in many disciplines and fields including telecommunications, biomedical engineering, seismology, and others. Digital synthesis is an application typically implemented in C++ with many frameworks provided [8, 14]; however, their algorithms and methods are less intuitive.

Our project proposes to explore the applications of functional programming and to demonstrate its features in a framework implementation that can be used in multiple disciplines, such as broadcasting, mathematics education, physics education, application-oriented programming, and many more.

Due to the parallel nature of processing multiple tracks of audio, the project is designed to replicate synthesis techniques by utilizing all the features and advantages of a purely lazy functional paradigm. While some algorithms were referenced, we implemented it from scratch. This is important as the algorithms used by typical frameworks are not made to be recursive, and as such, lack the optimizations from recursion. In addition, an algorithm built from scratch for a functional paradigm can avoid the many possible side effects that accompany the procedural algorithms.

This application extends and supports the applications for the Clean programming community by developing a Functional Programming paradigm framework for digital signal synthesis based upon analog Fourier series leveraging the capabilities of the Clean programming language. Additional contributions made by this project are the LFO, Flanger, Saturation, MusicXML parsing, and Stereo separation.

In this paper, after briefly presenting a general background of digital synthesis (section 2), the details of each project component are provided (section 3). These include: intuitive description of the workflow (section 3.1), diving into the explanation of the methods used for implementing the digital synthesis (section 3.2), clarification of amplitude modulation (section 3.3), providing the details of the MIDI input file format (section 3.6), description of the methods used for processing the data (section 3.9) and finally analyzing the .WAV output file format (section 3.10). These are, later on, followed by the summary of the results (section 4), by the related work (section 5), by the conclusions (section 6), and by the future plans (section 7).

2 Background

Digital synthesis is a field that was pioneered in the 1970s, and it is still continuously innovated by the music industry. The purpose of digital synthesizers is to use the power of microprocessors to replicate analog synthesis. Among the techniques used are additive synthesis, *wavetable lookup synthesis* (see subsection 3.2 for details), and physical modeling.

Additive synthesis is a technique for creating *waveforms* (see subsection 3.2) via the summation of sine waves. A sine wave is a waveform of pure single-frequency value. By summing multiple sine waves at various frequencies, amplitudes, and phase shifts, it is theoretically possible to generate all types of sound waves. The reference [?] gives more helpful information about this simple but commonly used concept, as a general base of generating sounds. Similarly, subtractive synthesis is a technique for creating *waveforms* via the subtraction of sine waves.

Our application utilizes harmonic additive synthesis to create the basic waveforms commonly used to generate more complex synths. Harmonic additive synthesis involves using the Fourier series of a waveform to determine the weighted summation of sine waves to generate the target waveform. In other words, it is an expansion of those waves using their relationship and the concept of orthogonality. The usage of it is breaking up an arbitrarily long and periodical sequence into smaller, simple chunks that can be processed individually. Also, the Fourier series, when used with appropriate weights, can be applied as a function approximator [?]. These sine waves are called *harmonics*, so-called because their frequencies are integer multiples of a standard fundamental frequency.

In order to generate the waveforms efficiently, digital waveform synthesis is typically implemented using wavetable lookup synthesis. In contrast to calculating a specific value of a waveform at a specific point of time, a waveform table is used to store one duty cycle of a waveform. The value of the waveform can be accessed by using the frequency to modify the access point of the waveform table and then multiply by the appropriate amplitude. With this method, it is far more efficient to generate a waveform by use of constant time array access instead of repeated calculations. In section 3.2 the details of each waveform are given.

3 Project details

3.1 The Process Flow from MIDI Input to WAV Output

Figure 1 depicts the structure of the application's process flow. The important phases of the digital synthesis are as in the following:

MIDI Input: it opens the MIDI input file. Reads notation information and stores within a list.

Digital Signal Process Chain (DSP Chain): it handles the signal generation and the data processing.

- *Sine Wavetable*: The Sine Wavetable contains a hard-coded array of values corresponding to amplitude values of one cycle of a fundamental sine wave.
- *Waveforms*: Using the data of the wavetable and user-specified Fourier series, the Waveforms module does weighted summation to generate new waveforms.
- *Envelope*: Using an envelope profile, the Envelope module applies an ADSR envelope to the signal (see section 3.3 for details).
- *Render*: The Render module generates signals from data passed from the MIDI Input module and it outputs a final render for writing to file.

Transcode: it transcodes the render data into the proper encoding for 8, 16, or 32 bit PCM WAV format.

WAV Output: it opens the WAV output file and writes the transcoded render data to the final WAV output file.

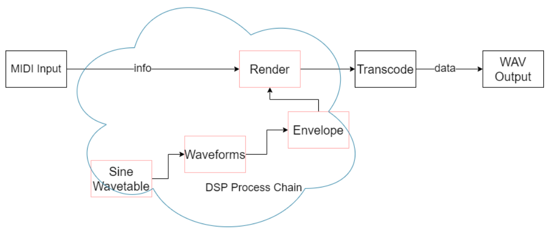


Figure 1. The Signal Flow Modules

3.2 Wavetable Lookup Synthesis

A periodic waveform can be decomposed into an infinite sum of varying frequencies and amplitudes of sine waves. This is the foundation of a technique called *Wavetable Lookup Synthesis*, in which a specific waveform is stored in a wavetable, and it exploits the relation between frequency and sampling rate to quickly build new waveforms. A waveform is the shape of a signal's graph, which shows the changes in amplitude over a certain amount of time. Sine wave is the simplest of all waveforms, and it contains only a single frequency and no harmonics. We used other simple waveforms (generated from the sine wave), which will make building complex sounds much more straightforward. We had to figure out how to sample from the stored wavetable and how to build new waveforms efficiently. In the following section, we discuss our approach of making sampling wavetable as efficient as possible.

3.2.1 Implementation. Based on the methods for designing wavetables of [17], our implementation chooses to set the size of the table as 2205, i.e., we store a table of 2205 real numbers, representing consecutive amplitudes within one single vibration of the sound wave. Thus, achieving the minimum sound frequency that humans can hear, which is 20 Hz. The single cycle sine wavetable, shown in Figure 2, is the basis for our additive and subtractive synthesis. As mentioned above, all the other waveforms can be efficiently generated from the sine wave by utilizing Fourier series [12].

For each waveform, we generate a list of indices, which we need to sample from the wavetable, using the `getIndexes` function (listing 3.2.2). These indices depend on the frequency and harmonic, and they are not necessarily integers. The `getValues` function (listing 1) takes wavetable, frequency, harmonic, and duration as parameters, and it uses generated indices, while linear interpolation solves the complication caused by real indices.

```
getValues :: {Real} Frequency Int Samples → [Real]
getValues waveTable frequency harmonic dur =
    [(getValue i waveTable) \\ i ← indexes]
where
    indexes = getIndexes frequency harmonic dur
```

Listing 1. Function `getValues`

```
getIndexes :: Frequency Harmonic Samples → [Real]
getIndexes frequency harmonic dur =
    map (λx = realRem x (toReal tableSize))
        (take dur [0.0, rate..])
where
    newRate = toReal SAMPLING_RATE
              / ((toReal harmonic)*frequency)
    rate = toReal(tableSize)/newRate
```

Listing 2. Function `getIndexes`

The wavetable is implemented as an array. Even though lists offer much more functionality, they are actually linked-lists, and they do not give us access to the elements in constant time.

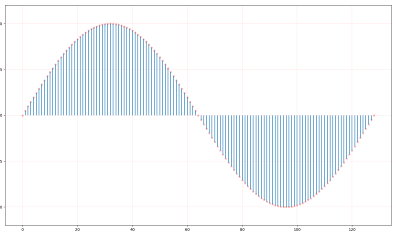


Figure 2. Sine wavetable

3.2.2 Wave Forms.

Besides sine wave, we use square, triangle and sawtooth as our building blocks. The project also includes parameters to generate pulse, silence, and noise waves [?]. In the implementation, a waveform type is represented as an algebraic data structure:

```
:: Wave = Sine | Square | Triangle | Noise
        | Pulse | Sawtooth | Silence
```

This is a parameter of our interface function, which generates waves as a list of `Real` numbers. Each waveform has a list of harmonics and a list of amplitudes. In the case of square, triangle, sawtooth, and silence, these lists are easily defined, while for pulse and noise, there is a need for more sophisticated techniques, such as phase-shifting for noise and subtracting sawtooth wave from a phase-shifted version of itself for a pulse wave.

Sawtooth: Frequency components are all harmonics. Relative amplitudes are inverses of harmonic numbers and all harmonics are in-phase

Square: Frequency components are odd-numbered harmonics, relative amplitudes are inverses of the squares of the harmonic numbers, and all harmonics are in phase

Triangle: Frequency components are odd-numbered harmonics, relative amplitudes are inverse harmonic numbers and every second harmonic is 180 degrees out of phase

Pulse: For the Pulse wave generation, a Sawtooth wave, and phase-shifted version of itself are subtracted from it. For this, an efficient helper function, `shiftLeft`, is defined. It moves every element of a list by the given number to the left.

Noise: For generating the Noise wave, all amplitudes are equal to 1, and harmonics are random numbers. Again using the `shiftLeft` function, lists are shifted by a random number of places before summing them up. Clean provides functions to generate pseudo-random numbers using Mersenne Twister Algorithm [9] in the module `Math.Random`.

```
harmonics_amplitudes :: Wave → ([Real], [Real])
harmonics_amplitudes Sine =
    ([1.0],[1.0])
harmonics_amplitudes Sawtooth =
    ([1.0,2.0..50.0],
     [(-1.0)^(k+1.0)*(1.0/k) \\ k←[1.0,2.0..50.0]])
harmonics_amplitudes Square =
    ([1.0,3.0..100.0],
     [1.0/x \\ x←[1.0,3.0..100.0]])
harmonics_amplitudes Triangle =
    ([1.0,3.0..100.0],
     [(-1.0)^(i+1.0)*(1.0/(k^2.0))
     \\ k←[1.0,3.0..100.0] & i←[1.0..]])
```

Listing 3. Function that returns respective lists of harmonics and amplitudes for different waves

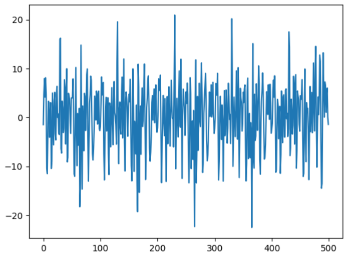


Figure 3. Noise waveform

3.3 Envelopes

The rendering process handles the data extracted from the MIDI files. This process converts the given data to a sequence of numbers, which represents the sum of all waves after normalization. Before the summation, envelopes modify each waveform to get the actual sound of the musical instrument. In music, an envelope describes the varying level of a sound wave over time. It is the envelope of a wave that establishes the sound's uniqueness, and it has a significant influence on how we interpret music. Classic envelopes consist of 4 main phases: Attack, Decay, Sustain, and Release. Sustain refers to a level, while the other phases represent time intervals. The attack phase is the period during which a sound needs to reach its peak from zero after the key is pressed. After the attack, the decay phase starts when the sound level decreases to its sustain level. The sound level stays unchanged during the sustain phase until the key is released. The final phase of the envelope is the Release phase, which continues until the sound fades to silence. Almost every musical instrument has a distinct envelope. For example, a quick attack with little decay makes a sound similar to an organ, while a longer decay is characteristic of a guitar. This application includes an envelope generator, which is a common feature of synthesizers and electronic musical instruments used to control the different stages of sound.

3.3.1 ADSR Envelope.

The first type of envelope implemented at the beginning of the project is ADSR. It is the simplest form of an envelope, which provides good bases for other more complex types, which are introduced later. The `getADSR` function is used to generate an ADSR envelope, Figure 4. It has only four basic steps: Attack, Decay, Sustain, and release. This function gets a beat, time signature, tempo, and ADSR record as parameters. At first thet are used to calculate the duration of the note. `noteToSamples`, one of the utility functions, is used to convert these parameters to the number of samples in this time interval. After that, the number of samples for each step of the envelope is calculated.

Since the release is independent of the note duration, it is enough to convert the given release duration to samples directly, but the other three steps need different approaches. Instead of directly using the given duration of each step independently, the number of samples is calculated based on the offset from the starting time and subtracting the sum of samples of the previous steps. This approach is essential to avoid losing samples during flooring of real numbers, and it makes sure that the number of total samples is equal to the sum of each step's samples. After calculating the list of samples, each step of the envelope is calculated independently. Concatenating these results produces the entire envelope excluding the release tail, however as the key may be released any time during the first three steps, including attack or decay, it might be necessary to shorten it. Finally, the release tail list is generated in the same way as the attack and decay, and it is concatenated to the others to get a complete envelope.

```
getADSR :: Beat TimeSignature Tempo ADSR → [Real]
getADSR beat timeSig tempo adsr
    = shortenedEnv
    ++ [...\\ x ← [1,2..releaseSamples]]
where
    noteDur = noteToSamples beat timeSig tempo
    attackSamples = secondsToSamples adsr.att
    decaySamples = secondsToSamples (adsr.att+adsr.dec)
                  - attackSamples
    ...
    wholeEnv =
        [toReal x / (adsr.att * (toReal SAMPLING_RATE))
        \\ ...
        ++ [adsr.sus \\ x ← [1,2..sustainSamples]]
    shortenedEnv = take noteDur wholeEnv
    endValue | noteDur  =  0 = 0.0
             | noteDur ≤ attackSamples
                = toReal noteDur
                  / (adsr.att * toReal SAMPLING_RATE)
             ...
             = adsr.sus
```

Listing 4. The `getADSR` implementation

3.3.2 DAHDSR Envelope.

The `getDAHDSR` function generates another type of envelope, which has two more steps than the ADSR envelope: delay and hold. Delay is the time interval before the attack, when the sound stays silent, while the hold phase comes after the attack and indicates the duration of the sound maintaining its peak. The implementation is similar to `getADSR` function and the data is stored as the `::DAHDSR` record. Each step is generated using list comprehensions, and they are concatenated. The whole envelope is generated, then we take its prefix to make sure that the key can be released at any time.

3.3.3 Casio, 8 step Envelope.

Casio, Figure 4, is a more modern type of envelope which allows more flexibility and a wide variety of configurations. It is different from the types mentioned above. It has eight steps, each consisting of a rate of change value and a target level value. The rate is used instead of duration. The rate and level pairs make it possible for the same phase to be ascending or descending depending on the needs of the users. Implementation of the Casio envelope differs from the other two envelopes, as the structure is different. The `CasioCZ` record provides data necessary for creating it. The first five steps represent the front part of the envelope, while the last three steps are used to generate the release tail after sustain. The `generateLine` function is used to generate point values of a line between two levels using the current rate. This function returns not a list of points, but a tuple of the list and real value. The second return value plays an essential role in interpolation. The last value of the line may not have an integer index. Hence, it cannot be included in the list. Due to the reason mentioned above, instead of directly using the previous endpoint as the beginning for the current line, we need to recalculate it based on the second value of the `generateLine` function using the formula: casio.level1 − rt2 ∗ (snd line1). In the end, similarly to other envelopes, we need to take the exact amount of samples according to the note duration.

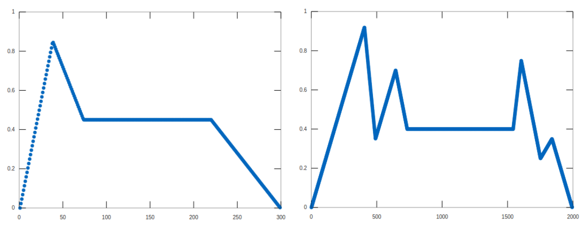


Figure 4. ADSR and 8-Step Casio Envelopes

3.3.4 Generalized Envelope. The last type of envelope data structure is a Generalized Envelope, which is similar to Casio but provides even more flexibility during sound synthesizing. Both of them use rate and level values to describe each step, but generalized envelopes do not have a fixed number of steps like the other previous structures. The `GenEnv` record uses a list to store data, where each element is an `EnvLevel` record type, containing rate and level values. Also, as generalized envelopes do not have a fixed number of steps before the release tail, the `GenEnv` record contains a value for the index indicating sustain level. Generating data for each step is done similarly to the Casio envelope, but the rate and starting value cannot be recalculated manually, so data preprocessing is needed before using it. Therefore, the implementation, which is shown below (listing **??**), is a bit different. `parseData` recursively traverses the initial list and generates a new one, which can be directly used to generate lines for each step using a similar method as in the Casio envelope.

Four data structures were created to support the different types of envelopes: ADSR, DAHDSR, Casio, and the Generalized envelope. A demonstration of DAHDSR followed by ADSR being applied to a sine wave is shown in Figure 5. Several types of envelopes provide a flexible environment for sound synthesizing and generate more sophisticated and better sounds.

3.3.5 Low-frequency Oscillator. Low-frequency oscillator (LFO) produces regularly repeating wave-forms at a low frequency. Those waves can be applied to generated waveforms during rendering. To encapsulate and handle parameters more easily several ADTs were implemented. LFOs are similar to envelopes. Both of them are used to modify sound to add some effects. But, they have different applying procedures. While envelopes are applied to each chunk independently, LFO is appllied to final list at the end of the rendering. `applyLFO`and `applyD ualLFO` functions are used to modify sound based on the LFO Profiles and they give capabilities to create effects, such as tremelo, vibrato, or ripple.

3.3.6 Rendering waves and applying envelope. The rendering process consists of several steps. The first step is to calculate the whole length of the sound, as each wave can start at a different moment and can have distinct lengths. This value is used later to generate a silent track, which acts as the base during the summing of all wave samples. The next step is to process data stored in each `NoteChunk` to generate sound waves and sum all of them up. Each `NoteChunk` stores wave type, time signature, tempo, envelope, and other data extracted from MIDI files, which are needed for generating wave and applying an envelope to it (Figure 5). Values for each wave can be calculated using already programmed functions for envelopes and sound synthesizing. After generating all the waves, we need to sum them up into a single list. If we use arrays, each wave's starting time is an index offset, but the same approach is not useful with lists. To easily sum up lists, they need to be of the same size. Therefore, the appropriate amount of silent samples should be appended on both sides of the list. The last step is the normalization, i.e. converting values to the $[-1.0, 1.0]$ range. After summing the lists up, some samples might go out of those bounds, which is why the final list needs to be normalized at the end of the process. After normalization, the sound rendering is finished, and it can be used for later processing.

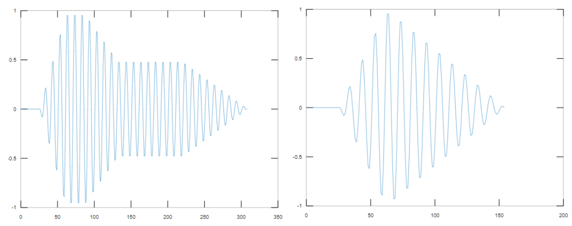


Figure 5. Sine wave modified by different envelopes

3.4 Sound reverberation

Reverb is a result of sound being reflected of some surface creating delayed sounds waves with lower amplitudes that interacting with each other enrich the sound quality by adding tone. The reverberation process implemented in this project depends on few parameters: *delay* (time interval, the difference between the starting wave and its first reflection), *decay* (real number between zero and one, representing the ratio between the amplitude of one wave and the amplitude of its preceding one) and the *number of bounces* (number of newly generated reflections which interfere with the starting sound). The implementation is based on repeating the same step, which is interfering single reflection with the resulting wave. Each new reflection is generated by shifting to the right and scaling down the last reflection (or the original wave in the case of generating first reflection) depending on the *delay* and *decay* respectively.

3.5 Stereo Separation and Audio Panning

The method of sound reproduction with the purpose of creating an illusion of multi-directional audible perspective, is called Stereophonic sound or, simply, stereo. This is accomplished by utilizing a minimum two autonomous sound channels to form the impression of sound being heard from various directions.

Stereo panning intents to adapt to the natural human way of accepting audio signals from two focal points, the left and the right ear. It modifies the digital signal and then distributes it to the appropriate channel, achieving a perception that a particular sound might come from the left, from the right, or from the center, hence producing a desired sound effect.

There are three main types of panning:

- **Amplitude Panning**: Tends to make a sound appear on the left/right by mimicking the natural phenomenon of a sound appearing louder on the side of the source and quieter on the opposite direction. This is done by manipulating with the volume of the channels which is represented as the amplitude of waves. To be more specific, if we want to pan a sound on a direction, i.e. on the left, by a specific amount, then we increase the amplitude on the left and decrease it by the same amount on the right. The right case would be similarly.
- **Delay Panning**: Intents to make a sound appear as coming from the left/right by relying in the fact that the sound arrives faster on the side of the sound source but is delayed by a small amount on the opposite direction. Specifically, if a sound is to be panned on one direction by a delay then we would shift the wave values on the opposite direction by the panning value.
- **Mixed Panning**: Utilizes both amplitude and delay panning. However, in order to avoid awkward silences we also consider a mix value.

```
seperation :: [Real] Real → ([Real],[Real])
seperation origSamples amt = result
where

  c_dist =sqrt( 2.0*115.5625 - 2.0*115.5625
  *cos( (1.0-amt) * 90.0))
  b_dist = sqrt( 2.0*115.5625 - 2.0*115.5625
  *cos( (1.0+amt) * 90.0))
  c_loudness = 0.5 + 0.5*(1.0- (  b_dist/10.75 ) )
  c_phaseOff = SAMPLING_RATE * (c_dist/soundSpeed)

  b_loudness = 0.5 + 0.5*(1.0 - (  c_dist/10.75 ) )
  b_phaseOff = SAMPLING_RATE * (b_dist/soundSpeed)
  c_samples =  repeatn (toInt((c_phaseOff -
  min(c_phaseOff, b_phaseOff)))) 0.0
  ++ (map (λx = x*c_loudness) origSamples)

  b_samples =  repeatn (toInt((b_phaseOff -
  min(c_phaseOff, b_phaseOff)))) 0.0
  ++ (map (λx = x*b_loudness) origSamples)
  result = (c_samples, b_samples)
```

Listing 5. The `seperation` implementation

The method "separation" takes a list of real numbers, that is the signal sent from the sound source, the second argument is real which denotes the direction of the given sound source. The method returns the tuple of a list of real numbers, one of them is the signal that is modified from the original one according to the direction, angle, and so on. So we get two different signals one of them is for the left ear and the second one is for the right one. There are some trigonometric calculations to get the angles, cosines, and sines, that's to calculate the distance between the sound source and left-right ears. If the sound source is closer to the left ear than the left ear will receive earlier than the right ear, so to have such a difference, phase shifting adds some zeroes in the list which is sent to the right ear. This makes a little time difference. To calculate the right amount of zeroes for shifting, distance difference should be divided to the sound speed and multiplied to the rate of the signal. The time complexity of the function is O(N) because it iterates over the list.

3.6 Parsing MIDI file Input

MIDI is short for *Musical Instrument Digital Interface*, which is related to audio devices for playing, editing, and recording music. The byte order is big-endian (as in [11]).

MIDI files are the standard format across all computing platforms for transferring MIDI data amongst users. MIDI files contain the standard channel-based MIDI messages, along with sequencer-related data (e.g. tempo, time and key signature, track names, etc.) and System Exclusive messages. Each message (also referred to as an event) is time-stamped. Any decent MIDI sequencer should allow MIDI files to be loaded and saved, in addition to the use of any proprietary file format. MIDI files differ from most other types of music files in that they do not contain encoded sound (e.g., as in a WAV file). Consequently, compared with WAV or even MP3 files, MIDI files are extremely compact.

The content of a MIDI file is structured as a series of blocks of data referred to as *chunks*. Each chunk begins with a 4-character ASCII type. It is followed by a 32-bit length, most significant byte first. There are two main types of chunks defined in MIDI, as illustrated in the table below.

structure / **type**	type (4 bytes)	length (4 bytes)	data (variable length of bytes)
Header Chunk	MThd	6	<format><tracks><division>
Track Chunk	MTrk	<length>	<delta_time><events>...

Table 1. Three types of MIDI Events

3.6.1 Information in MIDI file.

The following introduces the most critical information in a MIDI file.

Format types of MIDI files: This describes the chunk structure of a MIDI file and determines how the following MTrk chunks relate to one another (as in [5]).

- *format 0*: MIDI files, there should only be one MTrk chunk, and this can, therefore, contain any valid event - i.e., all timing-related and note (potentially multi MIDI channel) data.
- *format 1*: The first MTrk chunk is a global tempo track and should contain all timing-related events and no note data. The second and subsequent MTrk chunks contain the actual note data, and should not contain any timing related events. As all tracks are played together, they all follow the tempo map described in the global tempo track (the first MTrk chunk).
- *format 2*: Each track is a separate entity (like drum patterns within a drum machine), and can each contain any type of event. There is no global tempo track - each track may have its tempo map. Any timing-related events are specific to the track in which they occur.

tracks: Describe the number of track chunks contained in the MIDI file. Track chunks (identifier = MTrk) contain a sequence of time-ordered events (MIDI and/or sequencer-specific data), each of which has a delta time value associated with it - i.e., the amount of time (specified in tickdiv units) since the previous event.

delta_time: The delta-time specifies the number of tickdiv intervals since the previous event (or from the nominal start of the track if this is the first event). If an event is to occur at the very start of a track, or simultaneously with the previous event, then it will have a delta-time of 0. The delta-time is a variable-length quantity in that it is specified using 1, 2, 3, or 4 bytes, as necessary.

events: There are three main different kinds of events that can occur in track chunk; each type has a different number of bytes to store the information. We are not able to know the length of each specific event until we reach its status byte, which stores the information indicating what type it is.

- *MIDI events* (status bytes 0x8n - 0xEn) Corresponding to the standard Channel MIDI messages, i.e., where 'n' is the MIDI channel (0 - 15). This status byte will be followed by 1 or 2 data bytes, as is usual for the particular MIDI message. Any valid Channel MIDI message can be included in a MIDI file.
- *SysEx events* (status bytes 0xF0 and 0xF7) There are a couple of ways in which system exclusive messages can be encoded - as a single message (using the 0xF0 status), or split into packets (using the 0xF7 status). The 0xF7 status is also used for sending escape sequences.
- *Meta events* (status byte 0xFF) These contain additional information that would not be in the MIDI data stream itself. E.g., TimeSig, KeySig, Tempo, TrackName, Text, Marker, Special, EOT (End of Track) events being some of the most common.

structure / **event type**	status byte	byte2	byte3	byte4
MIDI events	0x8n - 0xEn	data	(data)	–
sysex events	0xF0 and 0xF7	length	data	–
meta events	0xFF	type	length	data

Table 2. Three types of MIDI Events

3.6.2 Challenges for parsing the MIDI file. Unlike regular audio files like MP3 or WAV files, MIDI files do not contain actual audio data; therefore, it is much smaller in size and more compact, which makes it more difficult to parse it since there are much information to extract and store.

3.6.3 Challenges for parsing Delta Time. Delta time is represented by a time value, which is a measurement of the time to wait before playing the next message in the stream of MIDI file data. Time values are stored as Variable-Length Values (VLV: a number with a variable width) [16]. Each byte of delta time consists of two parts: 1 continuation bit and 7 data bits. The highest-order bit is set to 1 if it needs to read the next byte, set to 0 if this byte is the last one in variable-length value.

Solution: To get an integer number represented by a variable length value.

i. convert the first byte in VLV to integer
   - if it is greater than 128, put it into a list and read the next bytes recursively
   - if it is not greater than 128, add this byte into a list and end the recursion
ii. convert the list of bytes into an integer number
iii. return an integer representing delta time and the length of the track chunk in bytes

3.6.4 Challenges for parsing Running Status of MIDI Events. While reading bytes coming from a MIDI message, the STATUS byte can in fact be omitted (except in the first message of that type).In such a case, we can receive a message that only has DATA bytes. The STATUS byte is then supposed to be the same as the last STATUS byte received. This is called MIDI RUNNING STATUS. It is useful for instance to optimize transmission when a long series of the same messages are sent.

If the first (status) byte is less than 128 (hex 80), this implies that the running status is in effect and that this byte is the first data byte (the status is carried over from the previous MIDI event). This can only be the case if the immediately previous event is also a MIDI event because system exclusive events and meta events will interrupt (clear) the running status.

Solution: The length of the track chunk is useful since it does not only help for processing standard chunks but also makes it easier to deal with unexpected chunk types – just by skipping that amount of bytes, then we can continue to process next chunk.

3.6.5 The structure of MIDI processing. The following is a general structure of functions that deal with the information contained in MIDI file (see figure 6 and listing 6).

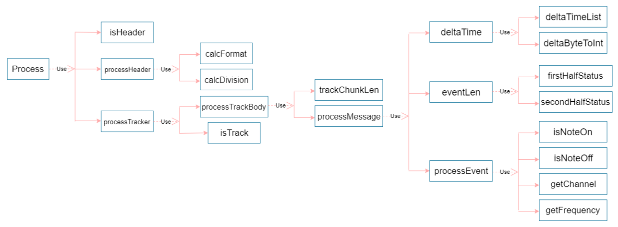


Figure 6. The MIDI processing functions

```
process :: [Char] → Info
process l | length l > 14 && isHeader (take 4 l)
            = { headerInfo = processHeader (drop 8 l),
                trackInfo  = processTrack (drop 14 l) }
          = abort "not enough information"

processHeader :: [Char] → HeaderInfo
processHeader l =
    { format = calcFormat (take 2 l),
      division = calcDivision (take 2 (drop 4 l))}

processTrack :: [Char] → [TrackInfo]
processTrack [] = []
processTrack l | isTrack l = processTrackBody (drop 4 l)
               = processTrackBody l
```

Listing 6. The `process`, `processHeader`, `processTrack` functions

`process`: This function parses a MIDI file from scratch, accepting a list of Char (i.e., bytes) and returning an `Info` record, which contains information about header and track chunks.
`isHeader`: This function takes the first four elements from a list of bytes to see if it is the type of header chunk(`MThd`). The first six bytes of the list gives information about the format, the number of track chunks in total and division.
`processHeader`: This function stores the first and third value in the `HeadInfo` record.
`processTrack`: This function uses `isTrack` function to see if the beginning of a track chunk is currently being processed, and if so, it drops the first four elements which contain information of chunk type and continues processing the remaining information.

3.7 Parsing MusicXML file Input

MusicXML is an XML-based file format for representing Western musical notation. It is a digital sheet music interchange and distribution format. The goal is to create a universal format for common Western music notation, similar to the role that the MP3 format serves for recorded music. The musical information is designed to be usable by notation programs, sequencers and other performance programs, music education programs, and music databases.

3.7.1 Target type for parsing. The tree data structure for storing final information that being extracted from the MusicXML file using the parsers. String contains the name of a tag and the list of ElementAttribute has the attribute information for the correspondence tag, the XML list contains the child elements of the parent.

```
:: XML = Text String
       | Element String [ElementAttribute] [XML]

:: ElementAttribute = { name :: String,
                        value :: String }
```

3.7.2 Target type for extracting information. MusicXML is a simplified version of XML which left out useless information and store data in a more useable structure. In this parser of Clean, we only focus on the part-wise score type of MusicXML where measures are nested within parts.We use the record to store the data of measure. For each measure, it contains several attributes and notes, and the information in attributes and note are as shown above.

```
:: MusicXML := [Measure]
:: Measure = { attributes :: [Attributes],
               notes :: [Note] }
:: Attributes = { divisions :: Divisions,
                  key :: Key,
                  time :: TimeSignature }
:: Note = { pitch :: Pitch,
            duration :: Duration,
            type :: Note_type }
```

3.7.3 Parsing process. Monadic parsing: A minimal monadic parser combinator library was written in Clean for XML parsing.

Process for parsing: There are two different kinds of tags, one is self-closing tag the other one has a starting tag and an ending tag. With the aid of parsers in the library we have separate parsers for that two types of tags accordingly.

Process for extracting information: After geting the XML type, we can now try to get the needed

3.8 Saturation Type Distortions in Digital Signal

Saturation is defined as a condition upon which additional amplitude gain is restricted past a set threshold amount.

In the book *DAFX - Digital Audio Effects* when the authors started talking about their work with tape saturation, they said *" They prefer doing multi-track recordings with analog tape-based machines and use the special physics of magnetic tape recording as an analog effects processor for sound design. One reason for their preference for analog recording is the fact that magnetic tape goes into distortion gradually Ear761 (pp. 216-218) and produces those kinds of harmonics which help special sound effects on drums, guitars and vocals."*

3.8.1 Implementation. All the implementations that will be shown in this section can be obtained in any of the wave forms to produce a saturated sound.

Hard clipping, the implementation of it is very straight forward as we are following the math function:

$$f(x, max) = \begin{cases} \text{max} & : \text{where x > max} \\ \text{-max} & : \text{where x < max} \\ \text{x} & : \text{where -max} \le \text{x} \le \text{max} \end{cases}$$

Soft Clipping, there are many implementation for it and to have many choices for the user since every clipping will give different "character" for the sound, there will be more than one implementation for the soft clipping. **Customized DAFX approach** The solution offered in the DAFX for distortion is using the formula :

$$f(x) = x/\mid x \mid (1 - \exp(x^2/\mid x \mid))$$

But looking through the formula and considering our goal, the formula that was used in the project and that had the desirable effect was:

$$f(x) = x/\mid x \mid (1 - \exp(-x^2/\mid x \mid))$$

As it is shown in the graph, the new wave doesn't hit the threshold which is **1** anymore, and the transition was very smooth as expected form the sigmoid function. The only part that was not satisfying in that distortion that the formula is giving us limited options in shaping the wave which is counted as disadvantage.

3.9 Digital Sample Transcoding and Normalization

3.9.1 General information about Transcoding. After the synthesis part, the signal needs to be converted into a form that can be recorded onto physical or digital media. This process is also known as *transcoding*. In days of analogous signal synthesis, recording equipment transcoded the electrical signals using various mechanical or electromagnetic methods. With digital synthesis, the applications have to transcode the digital waveforms into bits in order to store them into the appropriate file.

3.9.2 Finding Suitable Form Challenge. Similarly to this concept, in this implementation, once the program obtains the wavetable examined in Section 3.2, the next step is to write this sound data to a WAV file.

As discussed in Section 3.10, three main components separate the WAV file: the `RIFF` chunk, the `fmt` sub-chunk and the `data` sub-chunk. The `data` sub-chunk contains the sound information, which is stored in bits. In consequence of that, it was necessary to find a way to convert the result of the wavetable into appropriate data for the file; hence, there are transforming functions implemented.

Solution: Initially, only the 8-bit version was created, which takes the list of output sample values and its maximum value and converts the values to fit the 8 bits range. In other words, the values of the samples are converted into an interval from 0 to 255. Later on, as a precondition for increasing the quality of the generated sounds, the function 16-bit version functionality was added, which alters the values to 16 bits samples stored into the interval 0 to $2^{16} - 1$, and to maximize the quality of the generated sound 32-bit version, which alters the values to 32 bits samples stored into the interval 0 to $2^{32} - 1$.

3.9.3 Multiple Channels Challenge. In a physical aspect, a *channel* is the passage through which a signal or data is transported. In the case of audio files, it is the passage or communication channel in which a sound signal is transferred from the player source to the speaker. Since humans evolved to hear binaurally, in order to deliver more depth and spaciousness for enhancing the audio, at least two channels are needed. That is why the creation of a multiple-channel version implementation was introduced as our next challenge.

Solution: To make the project more flexible with the number of channels in the received input, two versions were made for the transform function. In the default case, the sound data obtained as the input will represent only one channel, meaning that the waveform could be correctly represented as a list of `Reals`. On the other hand, in case the data has two or more channels, then a better representation would be a list of lists of `Real`.

3.9.4 Implementation and Mathematical Background of Transcoding. After some research regarding the opportunities Clean offers, the best approach proved to be the concept of vertical graph shifting and multiplying. The most straightforward vertical graph transformation involves adding a positive or negative constant to a function. For example, by adding the same constant k to the output value of the function regardless of the input, the function shifts the graph of the function vertically by k units.

```
transform_one_channel :: [Real] Real BitVersion → [Byte]
transform_one_channel list max bitVersion
= flatten
    (map (toBytes Signed LE
                (translated_bit_version/BYTE_SIZE))
         (map (λx = moving_wave x max bitVersion) list))
where
```

```
    translated_bit_version
        = translating_bit_version bitVersion
```

Listing 7. The `transform_one_channel` function

To give a more detailed explanation of the implementation (Listing 7), it is a good idea to handle the 8, 16, and 32 bits cases separately. Regarding the 8 bit case, the first step is applying the `moving_wave` (Listing 8) function to each element in the list. The `moving_wave` function takes three parameters: the `targeted_number`, the `max` and the `bitVerson`. When passing 8 as a `bitVerson` parameter to the `moving_wave`, this function divides the `targeted_number` with the `max`, which in this implementation denotes the maximal possible limit of the values in the input list. In the case of real numbers from $[-0.5, 0.5]$ this value is 0.5. Following this, the function `moving_wave` is adding 0.5 to `targeted_number` over `max`. This step represents vertical shifting. After that, this sum is multiplied by 255 (which is $2^8 - 1$) in order to get real numbers in $[0, 255]$ interval. But as the expected output is of type `Int`, applying the function `toInt` (build-in function in Clean) is needed as the last operation performed by the `moving_wave` function. After the `moving_wave` is performed to each number from the given list the next step in the transformation is mapping the function `toBytes` that converts an `Int` into a list of binary digits of length 8. The last step is flattening the list of `Bytes` into a list of bits that can be later written into the WAV file. If the input is a list of lists instead of a single list (in case of multiple channels), mapping the same transformation to every input sub-list is the proper conversion.

```
moving_wave :: Real Real BitVersion → Int
moving_wave targeted_number max bitVersion
| translated_bit_version = 8
    = toInt (255.0*(targeted_number/max+0.5))
= moving_wave_aux targeted_number
                  max
                  translated_bit_version
where
    translated_bit_version
        = translating_bit_version bitVersion
```

Listing 8. The `moving_wave` function

In the 16 bit case, in the `moving_wave` function instead of using `toInt` as a last operation, it is more appropriate to create and apply the function `moving_wave_aux` which takes three parameters (the `targeted_number`, the `max` and the `bitVersion`). The `moving_wave_aux` function then returns $2^{15} - 1$ if the `targeted_number` equals to `max` or otherwise the lower integer part of `targeted_number` multiplied by 2^{15} and divided by `max`. Following that, similar to the 8-bit version, the `toByte` mapping is performed. The transformation is concluded by concatenating the sub-lists of bits of length 16 into a single list. If the input is a list of lists, mapping the same transformation to each sub-list of the input gives the expected output.

The 32-bit version is almost the same as the 16-bit one, the only difference is that the `moving_wave` multiplies by 2^{31} instead of 2^{15} and `toByte` returns a list of length 32 instead of 16.

3.9.5 Evolution to the Interface. As stated previously, we gradually created three individual functions to cover the 8-bit, 32-bit and 64-bit cases. However, in time, it was established that the creation of an interface was much more generic, and hence more reliable. As observed before, the function `transform_one_channel` takes the bit version as a parameter, which makes it much more flexible in case further changes need to be introduced.

3.10 WAV Output File Format

The WAV file is an instance of a *Resource Interchange File Format* (RIFF) defined by IBM and Microsoft. Many audio coding formats are derived from the RIFF format (i.e., AVI, ANI, and CDR) [10]. The most common WAV audio format is uncompressed audio in the linear pulse code modulation (LPCM). However, a WAV file can contain compressed audio, on Microsoft Windows, any Audio Compression Manager codec can be used to compress a WAV file. LPCM audio is a choice for professional users and audio experts in order to acquire maximum audio quality. [7].

3.10.1 The WAV File Structure. A RIFF file is a tagged file format. It consists of a specific container format called chunk, and the chunk has four character tag (FourCC) and the size (number of bytes) of itself. The tag specifies how the data within the chunk should be interpreted, and there are several standard FourCC tags. Tags consisting of all capital letters are reserved tags. The outermost chunk of a RIFF file has a RIFF form tag; the first four bytes of chunk data are a FourCC that specifies the form type and is followed by a sequence of subchunks. In the case of a WAV file, those four bytes are the FourCC WAVE. The remainder of the RIFF data is a sequence of chunks describing the audio information [10]. The ability to extend the format later is a massive advantage for a tagged file format, as the mentioned format will not confuse the file reader. The rules of RIFF reader specifies that it should ignore all irrelevant tagged chunk and treat it as valid input.

```
<WAVE-form> →
    RIFF('WAVE'
         <fmt-ck>              // Format
         [<fact-ck>]           // Fact chunk
         [<cue-ck>]            // Cue points
         [<playlist-ck>]       // Playlist
         [<assoc-data-list>]   // Associated data list
```

```
        <wave-data> )          // Wave data
```

Listing 9. RIFF header

The definition has a few interesting points. The format chunk is necessary since it describes the format of the sample data that follows. The cue points chunk is optional and identifies some significant sample numbers in the wave file, the playlist and fact chunks are also optional. Finally, the mandatory wave data chunk contains the actual samples. Unfortunately, the definition of the WAV file is foggy regarding the place of INFO chunk, as well as the CSET chunk, but the PCM format generated omits this chunk since the functionality does not depend on it.

The `<wave-data>` contains the waveform data. It is defined as in the following [10]:

```
<wave-data>  → { <data-ck> | <data-list> }
<data-ck>    → data( <wave-data> )
<wave-list>  →
  LIST( 'wavl' { <data-ck> |           // Wave samples
                 <silence-ck> } ... )  // Silence
<silence-ck> →
  slnt( <dwSamples:DWORD> ) // Count of silent samples
```

Listing 10. Wave data

In the wave data chuck (Listing 10) produced by the application, the implementation changes `<data-list>` to `<wave-list>` in line 1 and `<wave-data>` to `<bSampleData:BYTE>` in line 2. These changes are done in order to avoid any possible recursion of `<wave-data>` contained in a `<data-ck>`. WAV files can contain embedded IFF `lists`, which can contain several `sub-chunks`.

3.10.2 WAV File Format Limitation. The RF64 format specified by the European Broadcasting Union has been created to solve the limited file size issue of the WAV format since the WAV format can only handle files less than 4 GB because of its use of a 32-bit unsigned integer to record the file size header although this is equivalent to about 6.8 hours of CD-quality audio (44.1 kHz, 16-bit stereo).

WAV format suffers from duplicated information between chunks. Also, 8-bit data is unsigned, which differs with 16-bits data which is signed, such inconsistency can be puzzling. Based on the file specification of the WAV file format, a set of functions were implemented to create a framework for writing the data into a file. These functions have been enumerated in detail in the following pages. In the process of making the framework for writing to a WAV file, we had to take into consideration some points, including the functional way in handling IO operations and language-specific features of Clean.

3.10.3 Purity in Clean. Due to Clean being a purely functional language, side effects such as writing to a file is not as straightforward as in imperative languages. Clean deals with this by using uniqueness typing to preserve referential transparency [1, 2].

3.10.4 Handling binary data in Clean. The Clean `StdEnv` supports basic file manipulation in the `StdFile` module. It provides operations for the `File` type, which can also be a unique type. There are several operations for writing data, though most of them are not easy to work with for binary data. The smallest unit we can write is a `Char`. We assume a `Char` in Clean is a byte, we denote it with a type synonym (`:: Byte := Char`).

3.10.5 Writing byte sequences to file. There is a function for writing a string (unboxed `Char` arrays in Clean) to a file. However, lists are easier to work with, we defined a function to write a list of `Chars` to a file 11.

The `!` in the type specifies that the arguments are strict, it can improve program efficiency where laziness is not needed. `♯!` is a strict let notation, assigning the output of `fwritec b f` to `f`. This `f` is not the same variable as the `f` in the line before. It introduces a new scope and shadows the previous variable, encouraged in Clean with unique types, and it makes the explicit passing of the unique file.

```
writeBytes :: ![Byte] !*File → *File
writeBytes []     f = f
writeBytes [b:bs] f
  ♯! f = fwritec b f
  = writeBytes bs f
```

Listing 11. Writing a list of bytes into a file

3.10.6 Integer and byte conversion. We need to manually define a function to convert a non-negative integer to a list of bytes in *little-endian* order for later use (Listing 12). It takes an argument that specifies how many bytes the number should be represented, e.g., if the argument is 2, then the output will represent a 16-bit word, the rest of the number is truncated. The function uses simple recursion and basic operators from `StdEnv`. The first parameter is the number of bytes, the second one is the integer to be converted.

```
uintToBytesLE :: !Int !Int → [Byte]
uintToBytesLE i n
    | i ≤ 0 = []
    = [ toChar (n bitand 255)
      : uintToBytesLE (i - 1) (n >> 8) ]
```

Listing 12. Converting an integer to a list of bytes

3.10.7 Interface for writing a WAV file. We implemented writing to a Wave file in (L)PCM format due to its simplicity. The type of the function for writing a Wave file is given in a `dcl` file (definition module). It takes some parameters that specify the structure of the file, and a list of bytes as the binary data in the data chunk.

```
:: PcmWavParams =
    { numChannels    :: !Int
      // Number of channels
    , numBlocks      :: !Int
      // Number of samples (for each channel)
    , samplingRate   :: !Int
      // Sampling rate in Hz (samples per second)
    , bytesPerSample :: !Int
      // Number of bytes in each sample
    }
writePcmWav :: !PcmWavParams ![Byte] !*File → *File
```

Listing 13. Interface of writing a Wave file in PCM format

All data that the Wave file needs can be calculated from these parameters. `numBlocks` represents the total number of blocks in the data chunk, where each block contains `numChannels` samples. `bytesPerSample` is how many bytes each sample contains.

3.10.8 Implementation of writing a WAV file. The main function is composed of three smaller functions. The first one writes the RIFF header into the file, as in the Listing 14.

```
writeHeader :: !Int !*File → *File
writeHeader l f
  ♯! f = fwrites "RIFF" f
  ♯! f = writeUint 4 l f
  ♯! f = fwrites "WAVE" f
  = f
```

Listing 14. Writing RIFF header into a file

The first argument is the length of the whole file minus the first eight bytes. It was calculated in the main function with 4 (the bytes `WAVE`) + 24 (size of the format chuk) + 8 (header of the data chunk) + `bytesPerSample` × `numChannels` × `numBlocks` (size of the binary data) + (1 or 0 depending on whether the size of the binary data is odd or even). `writeUint` is a utility function that combines `writeBytes` and `uintToBytesLE`. The second function writes the format chunk, which contains writing in the following data [7], using the same method as the previous function (Table **??**).

The last function takes the length and the list of the binary data, and writes it into the file using `writeBytes` after writing in the chunk header. It also takes care of adding a padding byte if the size of the data in bytes is odd. The main function composes the smaller functions and evaluates the size of the binary data so that it does not need to be calculated more than once in the sub-functions (Listing 15).

```
writePcmWav :: !PcmWavParams ![Byte] !*File → *File
writePcmWav p d f
  ♯! l = p.bytesPerSample * p.numChannels * p.numBlocks
  ♯! f = writeHeader (l + if (isEven l) 36 37) f
  ♯! f = writeFormat p f
  ♯! f = writeData l d f
  = f
```

Listing 15. The main function for writing Wave files

After running a file through the function, a Wave file is written, which can be played on a music player software. The whole process can be seen in Figure 7.

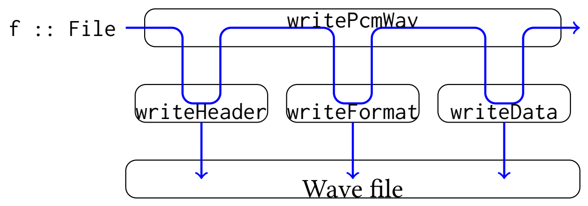


Figure 7. The process of writing a Wave file in Clean

4 Results

In the initial test runs of the application, we used a hard-coded notation of Beethoven's Für Elise as input. The first 16 measures of Für Elise was chosen as an initial test input as the notation involved only a single instrument, and the melodic and harmonic lines contained only monophonic lines. The initial test render of Für Elise with only digital synthesis signals from the signal generation modules took a total amount of time ranging between 900 - 1000 seconds to complete. Further iteration on the application, in which the wavetable implementation was changed from lists to arrays, resulted in a subsequent rendering time of 4-6 seconds.

Following later implementation of the MIDI input and the Envelope modules, we were then able to do test renders using a variety of MIDI files. The first one we utilized was *simple.mid*, a MIDI file that the team created specifically to test the synth generation capacity of the program. *Simple.mid* consisted of a series of A4 (440hz) notes at varying time intervals, from 1/16 of a beat to a double beat. The design of *simple.mid* was done to test the synth generation at different lengths of note values. Afterward, a sustained *CMaj* chord is played to test the ability of the program to layer notes polyphonically.

The next MIDI file that we used to test the program with is *FurElise – Short.mid*. The MIDI file contains the first 16 measures of Für Elise. For the same reason as the hardcoded version of Für Elise, this MIDI file tests the program's capability to render two monophonic lines of melodic and harmonic content in parallel. The *FurElise.mid* file is a MIDI file that extends this by containing the first 32 measures of Für Elise. While the melodic and harmonic characteristics do not change much from the first 16 measures, the additional length of the track is a good test of complexity efficiency.

5 Related Work

Euterpea [3] is a Haskell library for algorithmic MIDI generation and low-level sound synthesis. Although it is written in a purely functional language similar to our work, it makes use of functional reactive programming, which is a different approach relying on interactivity, whereas we focus on music synthesis using abstraction available in standard functional programming.

Eric Zuubier's [17] work has similarities in the generation of music in Clean while using higher abstraction levels. Contrarily, our paper utilizes MIDI and WAV files for a more generalized digital synthesis, whereas Zuubier focuses on a specialized digital synthesis for *just intonation* (the tuning of musical intervals as whole number ratios of frequencies).

Jerzy Karczmarczuk's work [4] is also written in Clean, and both share the ability to handle multiple instruments. Our approach places emphasis on a mathematical model in contrast to the physics and circuit like implementation were characteristic of Karczmarczuk's approach. Finally, while we were able to generate music, this was not done by Karczmarczuk's work [4].

Maximillian [8] is one of many C++ frameworks [14] that is similar to our project in that it implements various waveform generators and envelopes. While we use similar techniques, the C++ implementation within Maximillian is optimized for the use of the procedural paradigm and live buffers while ours focuses on letting users create their waveforms via the Fourier series. Additionally, our implementations for envelope are far more intuitive to the actual sound design process.

6 Conclusion

The digital synthesizer application successfully demonstrated another major application of functional programming. There were some challenges in the process. These included: the creation of the framework for writing to WAV, the conversion of data to bit format, and the integration of the variety of specifications and conventions within the MIDI and WAV file formats.

The team was successful in implementing full-featured frameworks for importing MIDI files, writing to WAV files, and creating synths via additive synthesis subtractive synthesis, and envelopes.

7 Further Work

The application can be easily extended with further functionality to become more competitive with current offerings within the digital synthesis ecosystem. Support can be added for more import file types such as MusicXML and export file types such as .mp3, .flac, and .ogg. Additional functionality can be added with filters based on frequency (e.g., passes, shelves, and EQ), effects based on amplitude (e.g., compression, gate, and distortion), and effects based on time (e.g., delay, reverb, and chorus).

Lastly, adding support for live MIDI input, sample banks, VST3 support, and a graphical user interface will further bring the application in line with other digital synthesizers.

Acknowledgments

This work was supported by the European Union, co-financed by the European Social Fund, grant. no **EFOP-3.6.3-VEKOP-16-2017-00002.**

References

[1] Achten, P., Plasmeijer, R.: The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1), 81-110, 1995

[2] Clean Language Report, https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf

[3] Euterpea, http://www.euterpea.com/

[4] Functional Framework for Sound Synthesis, https://www.researchgate.net/publication/220802768_Functional_~Framework_for_Sound_Synthesis

[5] Guide to the MIDI Software Specification, http://somascape.org/midi/tech/spec.html

[6] Hudak, P., Quick, D.: Haskell School of Music – From Signals to Symphonies, *Cambridge University Press*, 2018

[7] Kabal, P.: Wave file specifications, http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html

[8] Maximilian: C++ Audio and Music DSP Library, https://github.com/micknoise/Maximilian

[9] Mersenne Twister Algorithm, https://ir.lib.hiroshima-u.ac.jp/files/public/1/15032/20141016122634147579/ACMTraModel_8_3.pdf

[10] Microsoft and IBM Corporation, August 1991, http://www.tactilemedia.com/info/MCI_Control_Info.html

[11] MIDI Files Specification, http://somascape.org/midi/tech/mfile.html

[12] Risset J.-C., Computer music experiments, *Computer Music Journal*, vol. 9, no. 1, pp. 11–18, 1985

[13] Subtractive Synthesis, *Moore, R. F. (1990). Elements of Computer Music*

[14] Szanto, G.: C++ Audio Library Options, *Superpowered.com*, 2018, https://superpowered.com/audio-library-list

[15] Thompson, S.: Haskell: The Craft of Functional Programming, *Addison-Wesley Professional*, 3rd edition, 2011

[16] Variable Length Value, http://www.ccarh.org/courses/253/handout/vlv/ https://soundbridge.io/what-are-waveforms-how-they-work/

[17] Zuurbier, E.: Organ Music in Just Intonation, https://www.ji5.nl/

HoCL: High level specification of dataflow graphs

Jocelyn Sérot[*][†]
[*]Institut Pascal, UMR 6602 UCA/CNRS/SIGMA
[†]Univ. Rennes, INSA Rennes, IETR - UMR CNRS 6164
jocelyn.serot@uca.fr

***Abstract*—We introduce HoCL (Higher order Coordination Language), a novel, functional, Domain Specific Language for specifying dataflow graphs. HoCL can be used to describe mixed-grain, parameterized, hierarchical and recursive graph topologies. Compared to existing specification formalisms and tools, the main originality of HoCL is the ability to describe graphs using a purely functional notation. This style of description offers a very concise way of denoting graph structures and allows graph patterns to be encapsulated as user-definable higher order functions. HoCL also supports Model of Computation (MoC) specific annotations and its compiler backend is able to export graph definitions to external tools including Graphviz, DIF, PREESM and SystemC. Those features make HoCL ready to use with existing dataflow visualization, analysis or simulation tools.**

***Index Terms*—Functional programming, Domain specific language, Dataflow modeling, digital signal processing.**

I. Introduction

Dataflow modeling is used extensively for designing digital signal processing (DSP) systems. With this approach, applications to be implemented are described as graphs of persistent processing entities, named actors, connected by first in, first out (FIFO) channels and performing processing ("firing") when their incoming FIFOs contain enough data tokens. By varying the semantics of these firing rules, many dataflow models of computations (MoCs) can be defined, offering different trade-offs between expressivity and predictability, while keeping the key property of dataflow models : their ability to naturally express the intrinsic parallelism of DSP applications.

As a result, a wide variety of dataflow-based design tools have been developed, such as Ptolemy [1], LabView or Preesm [2], for specification, simulation and synthesis for hardware or software implementation of dataflow-oriented applications.

With these tools, the specification of the application is typically carried out textually, using some form of graph notation or graphically, using a dedicated Graphical User Interface (GUI). In both cases, the specification of large or/and complex graphs quickly become tedious and error-prone.

In this paper, we propose a domain-specific language (DSL), named HoCL aimed at simplifying and streamlining the description of dataflow graphs with large and/or complex topologies. The key feature of this language is the ability to describe graph structures as *functions*, so that several well-known and powerful concepts drawn from functional programming languages – such as polymorphic typing and higher order functions – can be applied both to ease and secure the task of describing these graphs.

The design of the HoCL language was also guided by the following concerns.

First, the ability to describe *hierarchical graphs*, *i.e.* graphs built from nodes which are either *atomic* actors (the action of which is performed as a single, indivisible operation) or decomposed as a *subgraph*. Hierarchical specifications are at core of top-down design methodologies, which are known to be highly applicable to DSP systems.

Second, ability to describe *parameterized graphs*, *i.e.* graphs for which the behavior of nodes (atomic or subgraph) can be declared as dependent on a set of dedicated values, distinct from the data flows. Graph parameterization is at the core of reconfigurable dataflow MoCs such as PSDF [4] or πSDF [5], which offer interesting trade-offs between expressivity, predictability and efficiency for implementing DSP applications.

Third, MoC-agnosticism, *i.e.* the idea that the language should be general enough to describe dataflow graphs with no assumption on the underlying dataflow semantics. The main goal of HoCL is to act as a *coordination language*[1], aimed at describing the *topology* of the graphs independently of the *behavior* of the atomic actors appearing in this graph. This said, and when required, MoC-specific informations (for example, production and consumption rates for SDF graphs) can be attached to descriptions by means of *annotations*.

Fourth, *mixed-style* descriptions. The HoCL language *allows* graph to be described using a functional notation, but it does not *force* the programmer to do so. Classical descriptions, in which graphs are described by explicitly listing nodes and edges, are still available. Both styles can be freely mixed.

The rest of this paper is organized in seven sections. Section II is a general, informal presentation of the language, by means of small examples. Section III gives some insights on MoC-specific annotations. Section IV is a more formal presentation of the language, including details on the syntax and semantics. Section V is a short account on the language implementation, focusing on the available backends for graph visualisation and code generation. Section VI describes the implementation, with HoCL, of a complete DSP application. Section VII is short review of related work and section VIII concludes the paper.

II. Language overview

As an introductory example, consider the dataflow graph (DFG) depicted in Fig. 1, where

- `i` (resp. `o`) is an *input* (resp. *output*) node,

[1]Hence its name

- nodes labeled `f`, `k` and `h` correspond to dataflow actors,
- clusters labeled `g` denotes dataflow subgraphs.

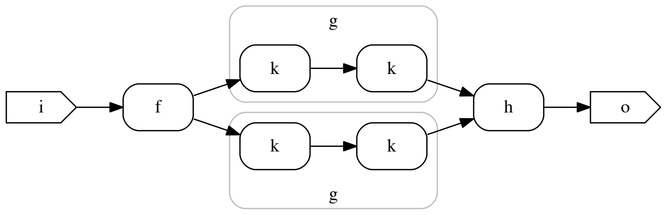


Fig. 1

A **structural** description of this DFG in HoCL is given in Listing 1.

```
1  node f in (i: t) out (o1: t, o2:t);
2  node k in (i: t) out (o: t);
3  node h in (i1: t, i2:t) out (o:t);
4
5  node g in (i: t) out (o: t)
6  struct
7    wire x: t
8    box n1: k(i)(x)
9    box n2: k(x)(o)
10 end;
11
12 graph top in (i: t) out (o: t)
13 struct
14   wire x1,x2: t
15   wire y1,y2: t
16   box n1: f(i)(x1,x2)
17   box n2: g(x1)(y1)
18   box n3: g(x2)(y2)
19   box n4: h(y1,y2)(o)
20 end;
```

Listing 1: A structural description of Fig.1 in HoCL

Lines 1–3 and 5–10 define *node models*, from which the graphs described in the specification are built. Each occurrence of a node model in a graph creates an *instance* of the model. A node declaration comprises an *interface* and a *description*. The interface gives the name of the node and the name and type of each input and output. In this introductory example, for simplification, all types have been identified to an abstract type named `t`. Nodes with an empty description (such as `f`, `h` or `k`) describe *atomic actors*. Such nodes are viewed as black boxes at the specification level[2]. A node description can also be given in the form of a *subgraph*, as for the `g` node in the example. In this case, the corresponding subgraph is expanded whenever the node is instantiated.

Lines 12–20 define the toplevel graph. A graph declaration is also made up from an interface and a description but, at the difference of node declarations, its description is always a subgraph and the corresponding graph is automatically (implicitly) instantiated[3].

[2]Backend-specific annotations – such as the name of the sequential function implementing the actor behavior for simulation for example – can also be attached to atomic actors.

[3]A valid specification in HoCL is therefore made up of at least one `graph` declaration.

In the example of Listing 1, both the toplevel graph `top` and the subgraph associated to node `g` are defined **structurally** (as evidenced by the `struct` keyword at lines 6 and 13 resp.). In other words, the corresponding (sub)graphs are described by explicitly listing all node instances (here called *boxes*) composing the graph and all edges (called *wires*) connecting these nodes. Describing graphs in a *structural* manner — be it textually, by means of `wire` and `box` declarations or graphically, using more or less sophisticated GUIs – quickly becomes tedious and error-prone. To overcome this problem, HoCL allows graphs to be specified using a *functional* notation. This notation is actually a small, purely functional, higher-order and polymorphic functional language.

Listing 2 gives another specification of the DFG in Fig. 1 in which both the graph `top` and the subgraph `g` are here described **functionally**.

```
1  node f in (i: t) out (o1: t, o2:t);
2  node k in (i: t) out (o: t);
3  node h in (i1: t, i2:t) out (o:t);
4
5  node g in (i: t) out (o: t)
6  fun
7    val o = k (k i)
8  end;
9
10 graph top in (i: t) out (o: t)
11 fun
12   val (x1,x2) = f i
13   val y1 = g x1
14   val y2 = g x2
15   val o = h y1 y2
16 end;
```

Listing 2: A functional description of Fig.1 in HoCL

The key idea behind functional dataflow graph description is that node models are viewed as *functions* and node instantiation corresponds to function application.

The definition of node `g` (at lines 6-8) for example, says that the corresponding (sub)graph is built by

- instantiating node `k` a first time (`k i`), creating a box `n1` and connecting the input wire `i` to its input,
- instantiating node `k` a second time (`k (...)`), creating a box `n2` and connecting the output of box `n1` to its input,
- connecting the output of box `n2` to the output wire `o` (`val o = ...`).

The definition at line 7 can also be written using the reverse application operator[4] $\triangleright$ as follows :

```
val o = i ▷ k ▷ k
```

in which the form of the RHS expression nicely "mimics" that of the described subgraph.

The definition of the toplevel graph `top` (at lines 11–16) uses the `val` keyword to bind intermediate values, which here corresponds to naming connecting wires :

[4]$x \triangleright f = f\ x$.

- line 12 respectively binds `x1` and `x2` to the first and second output of the `f` node[5],
- lines 13 and 14 respectively bind `y1` and `y2` to the output of the first and second instance of the `g` node, *i.e.* to the output of the corresponding subgraphs,
- line 15 binds these two values to the inputs of the `h` node and its output to the graph output `o`.

Thanks to referential transparency, the definition given at lines 12–15 can be rewritten in a slightly more concise manner, without explicitly binding `y1` and `y2`, as :

```
val (x1,x2) = f i
val o = h (g x1) (g x2)
```

A. Wiring functions

Values bound by `val` declarations are not limited to wires but can also be, as in any functional programming language, functions.

For example, the definition of subgraph `g` in Listing 2 can be rewritten as follows :

```
node g in (i: t) out (o: t)
  val o = twice k x
end;
```

where `twice` is the function defined, classically, as :

```
val twice f x = f (f x)
```

and has type : $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

The definition of the `top` graph in Listing 2 can also be reformulated as follows, using a function :

```
graph top in (i: t) out (o: t)
fun
  val diamond left middle right x =
   let (x1,x2) = left x in
   right (middle x1) (middle x2)
  val o = diamond f g h i
end;
```

The `diamond` function takes four arguments : three functions, `left`, `middle`, and `right` and a value `x` and applies the given functions to `x` to form the diamond-shaped pattern exemplified in Fig. 1. For this, it first applies function `left` to `x`, giving two intermediate values `x1` and `x2`, then applies function `middle`, in parallel, both to `x1` and `x2` and finally applies function `right` to the results. This definition is conveyed using a *local definition* (`let ... in`). The semantics of local definition is that of classical FPLs : the scope of the values defined in the `let` part is limited to the declarations occurring in the `in` part. In essence, the `diamond` function captures ("encapsulates") the depicted graph pattern, just as the `twice` function was capturing the repetition pattern depicted in the subgraph `g`.

Functions like `twice` or `diamond` may be viewed as a means of capturing wiring patterns in dataflow graphs. For this reason, we call them *wiring functions*, to distinguish them from "ordinary" functions operating on scalar values.

HoCL comes with a *standard library* defining several useful wiring functions encapsulating classical graph patterns such as, for example :

- `iter`, for applying a given function n times in sequence; so that the function `twice` can actually be defined as :

```
    val twice = iter 2 f
```

- `pipe`, a variant of `iter` in which a distinct function is applied at each stage (see Sec. VI),
- `map`, to apply the same function to a list of values,
- `mapf` to apply a list of functions to a given value,
- ...

An important feature is that all these functions are defined using regular HoCL declarations, *i.e. within the language* itself[6]. For example, the definition of the `iter` wiring function is just, and as expected :

```
val rec iter n f x =
  if n=0 then x
  else iter (n-1) f (f x)
```

The set of available higher order graph patterns is therefore not fixed but can be freely modified and extended by the application programmer to suit her specific needs. This is in strong contrast with most dataflow-based design tools in which similar abstraction mechanisms rely on a predefined and fixed set of patterns.

B. Recursive graphs

In a dataflow context, a *recursive graphs* is a graph in which the refinement of some specific nodes is the graph itself. A typical example is provided by Lee and Parks in their classical paper on dataflow process networks [6].

This example is an analysis/synthesis filter bank under the SDF (Synchronous Data Flow) model. The corresponding dataflow graph has a regular structure which can be characterized by its "depth". Fig. 2, for example, shows a graph of depth three[7].

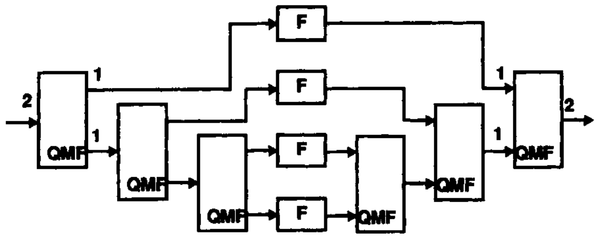


Fig. 2: A filter bank of depth 3 under the SDF model (from [6], Sec III-C, p 792)

For the sake of generality, Lee and Parks propose to view this graph as an instance of a "recursive template", depicted in Fig. 3.

[5]Strictly speaking, to the first and second output of the *box* resulting from the instantiation of node `f`. For simplification, and unless explicitly noted, we will now denote a node instance by the name of the corresponding node model.

[6]In file `lib/hocl/stdlib.hcl` technically.

[7]The meaning of the actor QMF and F and the numbers on the wires are irrelevant here.

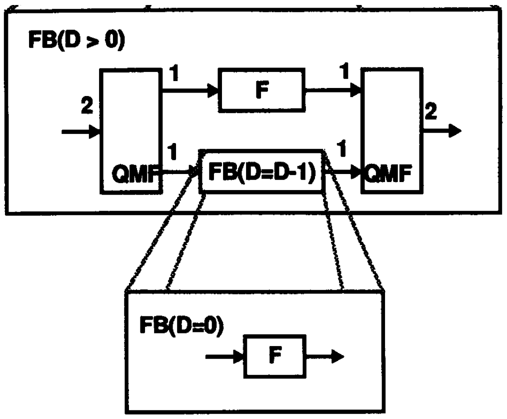


Fig. 3: A recursive template for filter bank of depth D under the SDF model (from [6], Sec III-C, p 793)

The recursive nature of this description is evidenced by occurrence, in the definition of the graph labeled `FB(D)`, of a node labeled `FB(D-1)`. The graph labeled `FB(D=0)` provides the base case for the recursion.

This graph structure can be readily encoded in HoCL as follows :

```
val rec fb d x =
  if d=0 then
    f x
  else
    let (x1,x2) = qmf x in
    qmf (f x1) (fb (d-1) x2)
end;
```

so that the graph of Fig.2 can be simply defined as

```
graph leeparks3 in (i: int) out (o: int)
fun
  val o = fb 3 i
end;
```

C. Cyclic graphs

Recursive definitions can also be used to encode cyclic graph structures, in which the output of a node is fed back to one of its input, as exemplified in Fig. 4. The corresponding graph can be described as follows in HoCL :

```
graph top in (i: int) out (o: int)
fun
  val rec (o,z) = f i (g z)
end;
```

The `rec` keyword is required here because the value `z`, here bound to the second output of node `f`, is also used as an *input* of the same node.

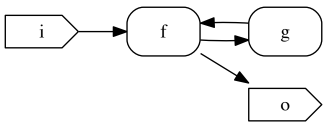


Fig. 4: A graph with a cycle

Mutual recursion is also possible, as exemplified by the following description of the graph depicted in Fig. 5 :

```
node f in (i1: t, i2: t) out (o1: t, o2: t);
node g in (i1: t, i2: t) out (o1: t, o2: t);

graph top in (i1: t, i2: t) out (o1: t, o2: t)
fun
  val rec ((o1,z1),(z2,o2)) = f i1 z2,
                              g z1 i2
end;
```

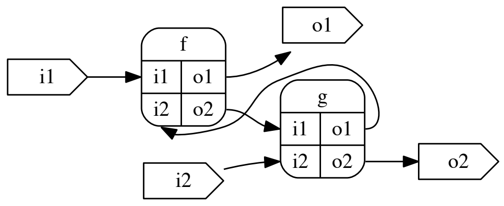


Fig. 5: Graph example 7

D. Parameterized graphs

The term *parameterized dataflow* was introduced in [4] to describe a meta-model which, when applied to a given dataflow model of computation (MoC), extends this model by adding *dynamically reconfigurable* actors. Reconfigurations occur when values are dynamically assigned to parameters of such actors, causing changes in the computation they perform and/or their consumption and production rates. The precise nature of changes triggered by reconfigurations and the instants at which these reconfigurations can occur both depend on the target MoC. HoCL offers a MoC-agnostic interface to this feature using a dedicated type to distinguish parameters from "regular" data flows.

Consider, for example, a node `mult`, taking and producing a flow of integers and parameterized by an integer value corresponding to the factor by which each input is multiplied to produce an output. Such a node could be declared as follows :

```
node mult in (k: int param, i:int) out (o:int)
```

with the corresponding function having type

$$\text{int param} \to \text{int} \to \text{int}$$

As shown by the above signature, parameters are supplied to nodes used curried application. The following program, for instance, instantiates node `mult` with `k=2`, giving the graph depicted in Fig. 6 :

```
graph top in (i: int) out (o: int)
fun
  val o = mult '2' i
end;
```

In Fig. 6, local parameters are drawn as house-shaped nodes and parameter dependencies using dashed lines. In the code, the simple quote around the parameter value 2 is used to distinguish parameter values from ordinary values[8].

[8]From a typing perspective, the `'.'` operator has type `t -> t param`.

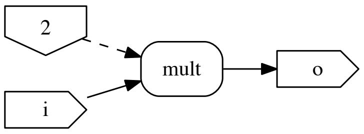


Fig. 6: A graph with a parameterized node

Because parameterized nodes are viewed as curried functions, they can be *partially applied*. We could therefore have written the line 3 of the previous example as :

```
val mult2 = mult '2'
val o = mult2 i
```

or even

```
val o = i ▷ mult '2'
```

In this view, partial application has a direct interpretation in terms of node configuration, a concept which DSP programmers are familiar with.

E. Parameters and hierarchy

When a parameterized node is refined as a subgraph, the value of the parameter(s) can be used to parameterize the nodes of the subgraph, either directly or by means of some dependent computations. This allows parameters to be propagated across graph hierarchies. This is illustrated by the following program, which expands into the graphs depicted in Fig. 7.

```
node sub param (k:int) in (i:int) out (o:int)
fun
  val o = i ▷ mult k ▷ mult (k+1)
end;

graph top in (i: int) out (o: int)
fun
    val o = i ▷ sub 2
end;
```

In graph `sub`, `k` is viewed as an input parameter (drawn as a dashed input port in Fig. 7) and used to parameterize both instances of the `mult` actor, first directly and second by through the *parameter expression* `k+1`. It is important to note that, although it could make sense in this particular example, parameter expression are *not* statically evaluated by the HoCL compiler since their interpretation ultimately depends on the target MoC (which controls, in particular, *when* parameters are evaluated to trigger the reconfiguration of the dependent actors).

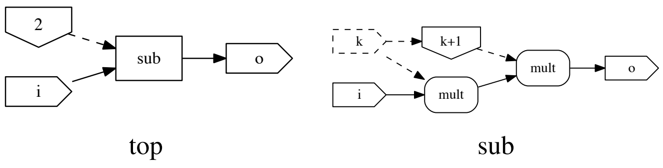


Fig. 7: A hierarchical graph with parameter passing

Parameter dependencies create *dependency trees*. The root of these trees can be either constants, as in the previous example, or specified as *top level input parameters*, as illustrated in the following program, which is an equivalent reformulation of the previous example. Note that, at the difference of node parameters, toplevel parameters must be given a value.

```
graph top
  in (n:int param=2, i:int) out (o:int)
fun
    val o = i ▷ sub n
end;
```

F. Labeled arguments

For nodes having a large number of inputs, passing the arguments to the corresponding function "in the right order" may become error-prone. This is specially true if a large proportion of these inputs have the same type, because the resulting error(s) will not be caught by the type checker in this case[9].

To circumvent this problem, HoCL supports label-based passing of arguments. This is illustrated in Listing. 3, in which the three instantiations of node `f` are valid and equivalent. Port binding is done by position at line 7 and by name (label) at line 8 and 9. The second form allows arguments to be passed in any order, as shown at line 9.

```
1  node f in (x: int, y: bool) out (o:  bool);
2
3  graph top
4    in (i1: int, i2: bool)
5    out (o1: bool, o2:bool, o3:bool)
6  fun
7    val o1 = f i1 i2
8    val o2 = f x:i1 y:i2
9    val o3 = f y:i2 x:i1
10 end;
```

Listing 3: A HoCL program illustrating label-based passing of arguments

Labeled arguments also relaxes the constraints on parameterized node signatures. Consider, for example, this alternate definition of the node `mult` introduced in Sec. II-D, in which the parameter `k` is here declared as the *second* argument :

```
node mult_bis
  in (i: int, k: int param) out (o:int)
```

This definition forbids partial application of node `mult_bis`. In particular, it is no longer possible to write :

```
val o = i ▷ mult_bis '2'
```

because, this requires the parameter `k` to be passed as the first parameter.

The solution is to pass this parameter with a label :

```
val o = i ▷ mult_bis k:'2'
```

[9]This is frequently the case in DSP applications because the sequential functions implementing the node behaviors often (over)use the `int` type to represent data.

III. MoC SPECIFIC ANNOTATIONS

As stated in Sec. I, HoCL is essentially MoC-agnostic. However, the language provides some mechanisms to "inject" some MoC-specific informations into the graph specifications, with the idea that these informations will be exploited by dedicated backends.

In the current state of the project, the language provides support for the synchronous dataflow (SDF) model.

In SDF, the number of tokens produced and consumed by an actor at each activation is fixed (known at compile time), which makes it suitable for modeling multi-rate DSP systems. To each edge e in an SDF graph are attached two integer-valued attributes, $P(e)$, $C(e)$, which specify the number of tokens respectively produced by the node connected at the source end of e and consumed by the node connected at the sink end of e.

In HoCL, and because directly annotating edges is not possible when writing functional graph descriptions (edges are implicit in this case), production and consumption rates are specified by annotating the source and destination node of the corresponding edge.

For example, the graph depicted in Fig. 8, in which the actor `f` respectively consumes and produces 3 and 2 tokens per activation and the actors `i` and `o` respectively produce and consume one token per iteration, will be described as follows :

```
node i in () out (o: t[1]);
node f in (i: t[2]) out (o: t[3]);
node o in (i: t[1]) out ();

graph top in () out ()
fun
  val _ = () ▷ i ▷ f ▷ o
end;
```

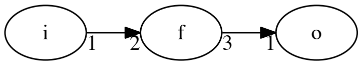


Fig. 8: A SDF graph

When a node accepts parameters, these parameters can be used to SDF-annotate some of the actor ports (the semantics of this annotation being, again, dependent on the chosen backend). For example, the following node declaration, in which the consumption rate on the input port of the actor is fixed by the parameter `n` is valid in HoCL, and can be viewed as limited form of dependent typing :

```
node downsample
  in (n:int param, i:t[n]) out (o:t[1]);
```

IV. LANGUAGE DEFINITION

The abstract syntax of the language[10] is given in Fig. 9.

A program consists of three sections : type declarations, global value declarations and node declarations. The first two can be omitted.

[10]Here deliberately limited to the subset dealing with *functional* graph descriptions.

Type declarations introduce type names, attached to node IOs and wires. At the specification level, these types are opaque and only used for checking the consistency of the graph. The actual semantics of types ultimately depends on the backend, in relation with the node dynamic behavior. For convenience, HoCL pre-defines a few basic types such as `int` and `bool`. As introduced in Sec. II-D, type expressions also include `t param`, where `t` is a basic type, for denoting node parameters.

Node and graph declarations have been introduced in Sec. II. Their syntax is similar. For graph declarations, values can be attached to parameter inputs (this is not reflected here but will be enforced by the type checker).

Value declarations (introduced by the `val` keyword) can appear either at the program or node level. In the first case, the scope of the defined symbol is the whole program. In the second case, this scope is restricted to the node being defined. Their semantics is that of `let` declarations in ML-like languages, except that left-hand side patterns are here limited to identifiers, tuples and unit value. They can be recursive and mutually recursive.

The expression-level language is classical, except that builtin values are limited to integer and boolean constants and unit.

$$\begin{array}{rcl}
\langle\text{program}\rangle & ::= & \langle\text{type\_decl}\rangle^* \ \langle\text{val\_decl}\rangle^* \ \langle\text{node\_decl}\rangle^+ \\
\langle\text{type\_decl}\rangle & ::= & \textbf{type}\ \textit{ident} \\
\langle\text{node\_decl}\rangle & ::= & \textbf{node}\ \textit{ident}\ \textbf{(}\ \langle\text{io\_decl}\rangle^*_{,}\ \textbf{)} \\
 & & \textbf{(}\ \langle\text{io\_decl}\rangle^*_{,}\ \textbf{)}\ [\langle\text{node\_impl}\rangle] \\
 & | & \textbf{graph}\ \textit{ident}\ \textbf{(}\ \langle\text{io\_decl}\rangle^*_{,}\ \textbf{)} \\
 & & \textbf{(}\ \langle\text{io\_decl}\rangle^*_{,}\ \textbf{)}\ \langle\text{node\_impl}\rangle \\
\langle\text{io\_decl}\rangle & ::= & \textit{ident}\ \textbf{:}\ \langle\text{type\_expr}\rangle\ [\textbf{=}\ \langle\text{const\_expr}\rangle] \\
\langle\text{node\_impl}\rangle & ::= & \langle\text{val\_decl}\rangle^* \\
\langle\text{val\_decl}\rangle & ::= & \textbf{val}\ [\textbf{rec}]\ \langle\text{binding}\rangle^+_{\textbf{and}} \\
\langle\text{binding}\rangle & ::= & \langle\text{pattern}\rangle\ \textbf{=}\ \langle\text{expr}\rangle \\
\langle\text{pattern}\rangle & ::= & \textit{ident} \\
 & | & \textbf{(}\ \langle\text{pattern}\rangle^+_{,}\ \textbf{)} \\
 & | & \textbf{(\,)} \\
\langle\text{expr}\rangle & ::= & \langle\text{const\_expr}\rangle \\
 & | & \textit{ident} \\
 & | & \langle\text{expr}\rangle\ \langle\text{expr}\rangle \\
 & | & \textbf{(}\ \langle\text{expr}\rangle^+_{,}\ \textbf{)} \\
 & | & \textbf{fun}\ \langle\text{funpat}\rangle \rightarrow \langle\text{expr}\rangle \\
 & | & \textbf{let}\ [\textbf{rec}]\ \langle\text{binding}\rangle^+_{\textbf{and}}\ \textbf{in}\ \langle\text{expr}\rangle \\
 & | & \textbf{(\,)} \\
\langle\text{funpat}\rangle & ::= & \textit{ident} \\
\langle\text{const\_expr}\rangle & ::= & \textit{int} \\
 & | & \textbf{true} \\
 & | & \textbf{false} \\
\langle\text{type\_expr}\rangle & ::= & \langle\text{base\_type}\rangle \\
 & | & \langle\text{base\_type}\rangle\ \textbf{param} \\
\langle\text{base\_type}\rangle & ::= & \textit{ident}
\end{array}$$

Fig. 9: Abstract syntax

Typing rules are classical and not reproduced here. They are given in [8]. The only distinctive feature is the interpretation of node declarations as function signatures. At the typing level, a node declared as

```
node f
  in (i1:τ1, ..., im:τm)
  out (o1:υ1 ..., on:υn)
```

is viewed as a function having type

$$i_1 : \tau_1 \to \ldots \to\ i_m : \tau_m \to \upsilon_1 \times \ldots \times \upsilon_n$$

A. Semantics

The semantics gives the interpretation of HoCL programs, described with the abstract syntax given above, as a set of *dataflow graphs*, where each graph is defined as a set of *boxes* connected by *wires*. The formulation given here assumes that the program has been successfully type checked. This semantics is built upon the semantic domain described in Fig. 10.

Values in the category Loc correspond to graph *locations*, where a location comprises a box index and a selector. Selectors are used to distinguish inputs (resp. outputs when the box has several of them.

Nodes are described by

- a category, indicating whether the node is a toplevel graph or an ordinary node[11],
- a list of inputs, each with an attached value[12],
- a list of outputs,
- an implementation, which is either empty (in case of opaque actors) or given as a graph.

Boxes are described by

- a category,
- a input environment, mapping selector values (1,2,...) to wire identifiers,
- a output environment, mapping selector values to sets of wire identifiers[13],
- an optional value.

Box categories separate boxes

- resulting from the instantiation of a node,
- materializing graph inputs and outputs,
- materializing graph input parameters,
- materializing graph local parameters.

The optional box value is only meaningful for local parameters bound to constants or for toplevel input parameters (giving in this case the constant value).

Wires are pairs of graph locations : one for the source box and the other for the destination box.

Closures correspond to functional values.

[11]This avoids having two distincts but almost identical semantic values for nodes and toplevel graphs.

[12]These values are used to handle partial application.

[13]A box output can be broadcasted to several other boxes.

Primitives correspond to builtin functions operating on integer or boolean values (+, =, ...).

The environments E, B and W respectively bind

- identifiers to semantic values,
- box indices to box description,
- wire indices to wire description.

Fig. 11 gives the most salient inference rules describing the semantics. The complete version is available at [8]. In these rules, all *environments* are viewed as partial maps from keys to values. If E is an environment, the domain of E is denoted by $\mathrm{dom}(E)$. The empty environment is written $\varnothing$. $[x \mapsto y]$ denotes the singleton environment mapping x to y. $E(x)$ denotes the result of applying the underlying map to x (for ex. if E is $[x \mapsto y]$ then $E(x) = y$) and $E \oplus E'$ the environment obtained by adding the mappings of E' to those of E, assuming that E and E' are disjoints.

Rule Program gives the semantics of programs. Global values are first evaluated to give a value environment (boxes and wires resulting from this evaluation are here ignored). Nodes declarations are evaluated in this environment. The result is an environment associating a node description to each defined node. The initial environment E_0 contains, the value of the builtin primitives (+, =, ...).

Rules NodeDecl1 and NodeDecl2 gives the semantics of node declaration. The former concerns nodes with no attached definition. These are are mapped to opaque actors. The Unit value initially attached to inputs here means "yet unconnected"). The latter concerns nodes with an attached definition. This definition is evaluated in an environment augmented with its input and output declarations, and the resulting graph (a pair of boxes and wires) is attached to the node description.

Rule Binding gives the semantics of bindings occurring in value declarations. The $\overleftarrow{\oplus}$ operator used in this rule merges box descriptors. If a box appears in both argument environments, the resulting environment contains a single occurrence of this box in which the respective input and output environments have been merged. For example

$$\begin{array}{rl} & [1 \mapsto \mathsf{Box}\langle \mathsf{actor}, [1 \mapsto 0], [1 \mapsto \{2\}]\rangle] \\ \overleftarrow{\oplus} & [1 \mapsto \mathsf{Box}\langle \mathsf{actor}, [1 \mapsto 4], [1 \mapsto \{3\}]\rangle] \\ = & [1 \mapsto \mathsf{Box}\langle \mathsf{actor}, [1 \mapsto 4], [1 \mapsto \{2,3\}]\rangle] \end{array}$$

Rules EAppN1 and EAppn2 gives the semantics of application when the LHS refers to a node. The former concerns the *partial* application of nodes. The value resulting from the evaluation of the arguments (which must be a graph location) is simply "pushed" on the list of supplied inputs. The latter concerns the complete application of nodes. It creates a new box and a set of wires connecting the parameters and arguments to the inputs of the inserted box (parameters first, then arguments).

Variable	Set ranged over	Definition	Meaning
v	Val	$\mathsf{Loc} + \mathsf{Node} + \mathsf{Tuple} + \mathsf{Clos}$	Value
		$\mathsf{Unit} + \mathsf{Int} + \mathsf{Bool} + \mathsf{Prim}$	
ℓ	Loc	$\langle \mathsf{bid}, \mathsf{sel} \rangle$	Graph location
n	Node	$\langle \mathsf{NCat}, \{\mathsf{id} \mapsto \mathsf{Val}\}, \{\mathsf{id}\}, \mathsf{NImpl} \rangle$	Node description
κ	NCat	$\mathsf{node} + \mathsf{graph}$	Node category
vs	Tuple	Val^+	Tuple
cl	Clos	$\langle \mathit{pattern}, \mathit{expr}, \mathsf{Env} \rangle$	Closure
E	Env	$\{\mathsf{id} \mapsto \mathsf{Val}\}$	Value environment
η	NImpl	$\mathsf{actor} + \mathsf{Graph}$	Node implementation
g	Graph	$\langle \mathsf{Boxes}, \mathsf{Wires} \rangle$	Graph description
B	Boxes	$\{\mathsf{bid} \mapsto \mathsf{Box}\}$	Box environment
W	Wires	$\{\mathsf{wid} \mapsto \mathsf{Wire}\}$	Wire environment
L	Locs	Loc^*	Location set
b	Box	$\langle \mathsf{BCat}, \{\mathsf{sel} \mapsto \mathsf{wid}\}, \{\mathsf{sel} \mapsto \mathsf{wid}^*\}, \mathsf{Val} \rangle$	Box
c	BCat	$\mathsf{actor} + \mathsf{graph} + \mathsf{src} + \mathsf{snk} + \mathsf{rec}$	Box category
		$\mathsf{inParam} + \mathsf{localParam}$	
w	Wire	$\langle \langle \mathsf{bid}, \mathsf{sel} \rangle, \langle \mathsf{bid}, \mathsf{sel} \rangle \rangle$	Wire (src loc, dst loc)
l, l'	bid	$\{0, 1, 2, \ldots\}$	Box id
k, k'	wid	$\{0, 1, 2, \ldots\}$	Wire id
s, s'	sel	$\{0, 1, 2, \ldots\}$	Slot selector
	Int	$\{\ldots, -2, -1, 0, 1, \ldots\}$	Integer value
β	Bool	$\{\mathsf{true}, \mathsf{false}\}$	Boolean value
π	Prim	$\{\mathsf{Value} \mapsto \mathsf{Value}\}$	Primitive function

Fig. 10: Semantic domain

V. Implementation

A prototype compiler, implementing the semantics described in the previous section has been written in OCaml. The source code is available at [7]. The distribution includes a command-line compiler, `hoclc`, turning HoCL source files into various dataflow graph representations, and a toplevel interpreter, supporting interactive building of dataflow graphs.

The command-line compiler comes with four distinct backends.

A `dot` backend produces graphical representations of the generated graphs in `.dot` format. All the graph representations used in this paper have been produced by this backend from the corresponding programs.

A `Dif` backend produces representations in the *Dataflow Interchange Format* (DIF). DIF [9] provides a standard, textual, notation for dataflow graphs aimed at fostering tool cooperation. By using DIF as an intermediate format, graphs specified in HoCL can be passed to a variety of tools for analysis, optimisation and implementation.

A `Preesm` backend directly generates code for PREESM [2], an open source prototyping tool for implementing dataflow-based signal processing applications on heterogeneous multi/many-core embedded systems.

A `SystemC` backend generates executable SystemC code for the simulation of simple DDF (Dynamic DataFlow) and SDF (Synchronous DataFlow) graphs (for which the behavior of the actors is described in C or C++).

VI. A complete example

In order to demonstrate the gain in abstraction and programmer's productivity offered by the HoCL language, we consider a small DSP application consisting in applying in parallel a sequence of three filters on a single data stream and selecting the "best" output according to a given criterion. Apart from the fact that it's typical of the kind of processing performed in the DSP domain, this application was chosen because we already had a working implementation, obtained with the Preesm [2] tool.

The dataflow graph, initially specified "by hand" using the Preesm GUI is depicted in Fig. 12, where :

- gray boxes denote actors,
- orange boxes denote dedicated broadcasting nodes,
- blue triangle-shaped boxes denote parameter sources,
- black arrows denote data wires and
- dashed, blue arrows denote parameter wires.

Input data, generated by the `src` node, is passed, through the `bcast` node to three parallel chains of nodes. In the first chain (bottom), data goes first through filter `f1`, then `f2` and finally `f3`. In the second (middle), the order is `f3`, then `f1` and finally `f2`. In the third (top), it is `f2`, `f3`, `f1`. The respective output data are finally given as input to the `select` node. Each filter node `f` takes a parameter input named `p`. For simplicity, the value of this parameter has here been considered as constant for all filters. The `select` node also takes a parameter, named `thr`.

Listing 4 gives a possible description of the graph depicted in Fig. 12 in HoCL.

```
1  type f16;
2
3  node src in () out (o:f16);
4  node snk in (o: f16) out ();
5  node f1 in (p: int param, i: f16) out (o:f16);
6  node f2 in (p: int param, i: f16) out (o:f16);
7  node f3 in (p: int param, i: f16) out (o:f16);
8  node select
9    in (thr: int param, i1:f16, i2:f16, i3:f16)
10   out (o: f16);
11
12 graph top
13   in (p: int param=2, thr: int param=128)
14   out ()
15 fun
16   val fs = [ f1 p; f2 p; f3 p ]
17   val chain s x = x ▷ pipe (shuffle s fs)
18   val sel c1 c2 c3 x =
19     select thr (c1 x) (c2 x) (c3 x)
20   val o = () ▷ src ▷ sel (chain [0;1;2])
21                          (chain [1;2;0])
22                          (chain [2;0;1])
23 end;
```

Listing 4: A description of the graph depicted in Fig. 12 in HoCL

Lines 3–10 declare the involved atomic actors. We have assumed here that all processed data has type `f16` (a shorthand for the `fix16` type used in the original implementation). Both the `p` parameter of the `f1`, `f2` and `f3` actors and the `thr` parameter of the `select` actors are here declared as `int`.

The graph itself is described in the `top` declaration, lines 12–23. The global parameters `p` and `thr`, with a default value (here arbitrarily set to 2 and 128), are declared as input parameters of this graph.

The value `fs`, defined at line 16, is a list made of the three filters, with their supplied parameter.

The wiring function `chain`, defined at line 17, is used to build the horizontal chains of filters depicted in Fig. 12. It takes a list of integers `s` and a input wire `x` and connects `x` to the sequence of nodes obtained by permuting the elements of the `fs` list. Permutation is done by the `shuffle` function and chaining by the `pipe` function. These functions can be defined informally by :

$$\texttt{shuffle}\ [\texttt{k}_1,\ \ldots,\ \texttt{k}_n]\ [\texttt{x}_1,\ \ldots,\ \texttt{x}_n] = [\texttt{x}_{k1},\ \ldots,\ \texttt{x}_{kn}]$$

$$\texttt{pipe}\ [\texttt{f}_1,\ \ldots,\ \texttt{f}_n]\ \texttt{x} = \texttt{f}_n\ (\ldots\ \texttt{f}_2\ (\texttt{f}_1\ \texttt{x})\ \ldots)$$

The code of these functions, which are defined in the HoCL standard library, is reproduced is Appendix A.

The wiring function `sel`, defined at lines 16–18, encodes the main graph pattern : it applies its arguments `c1`, `c2` and `c3` in parallel to its argument `x` and routes the three results to the `select` actor.

The top level graph is built, at lines 20–22 by applying the `sel` function to the three chains of filters, themselves obtained by applying the `chain` function to the corresponding lists of permutation indices.

The program in Listing 4 is only 23 lines. All wiring errors are caught immediately by the type checker, allowing immediate correction. As a result, obtaining the correct dataflow graph took less than 10 minutes. By contrast, describing the initial version of the graph using the Preesm GUI took more than 45 minutes. This times includes the definition of the node interfaces (4), the placement of the nodes (14) on the canvas and, above all, the manual, cumbersome, drawing of the connexions between the nodes. This represent a four time increase in productivity. Moreover, and most importantly, whereas it's straightforward, with the HoCL formulation, to modify the graph (adding or modifying the number of chains, changing the permutation choices, *etc.*) to test new application configurations, this task is much more tedious and error-prone with the purely GUI-based representation.

VII. Related work

In [6], Lee and Parks relates functional languages to dataflow process networks in two ways. First, for interpreting the behavior of actors operating on streams and second for describing graphs resulting from the replication of a given actor on parallel streams, using the `map` higher order function. The second idea is similar, in principle, to that used in HoCL but, in [6], no attempt is made to generalize the correspondence between functional expressions and graph structures beyond the particular pattern captured by the `map` HOF.

The work of Sane *et al.* [10] is more closely related to ours. They propose an extension to the DIF [9] notation supporting the use of so-called *topological patterns* for explicit and scalable representation of regular structures in DFGs. The definition of these patterns explicitly relies on a indexing mechanism for nodes and edges. HoCL is more general in the sense that *any* dependency pattern can be described, and not only those based on explicit indexing. Moreover, in the work described in [10], patterns are built-in and the set of available patterns is therefore fixed. By contrast, patterns are first class values in HoCL, and can therefore be defined directly by the programmer, *within the language*[14].

The HoCL language was inspired, in part, by the network description language used in the CAPH language for dataflow-based high-level synthesis [3]. Some design decisions were also motivated by conclusions of a retrospective assessment of the CAPH project reported in [11]. The idea of *mixed-style* description, for example, in which functional descriptions can co-exist with structural ones, can be viewed as a way to limit the "disruptiveness" of a purely functional notation, by presenting them as a *possible alternative* to the classical structural notation (with the idea that programmers will eventually "switch" to the former when they realize the benefits). The early adoption of DIF as a target backend can also be viewed

[14]Mention is made, in [10], of "user-defined patterns", by means of external "procedural Java or C code" but no detail is given on how the corresponding definitions are injected into the DIF language and semantics and we have not able to find examples of this mechanism in the literature.

as an answer to the "invasiveness" problem mentioned at the end of [11].

VIII. Conclusion

The design and development of the HoCL language started very recently and this paper should be viewed more as a draft specification than as a definite language reference.

In particular, the way MoC-specific features can be "injected" into the language without compromising its generality is an important issue which remain to be fully investigated. It is still uncertain, for example, whether relying on annotations, such as presented in Sec. III, is always feasible or whether some specific MoCs may require deeper changes to the syntax or semantics of the language itself.

Work is undergoing for reformulating in HoCL complex DSP applications, initially developed with tools using lower-order specification formalisms, such as Ptolemy, DIF or Preesm, in order to further assess the gain in expressivity and in the effort required by the specification of the input dataflow graph.

References

[1] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[2] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. F. Nezan, and S. Aridhi, "PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming," in *EDERC*, Italy, Sep. 2014, p. 36.

[3] J. Sérot and F. Berry, "High-level dataflow programming for reconfigurable computing," in *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, 2014, pp. 72–77.

[4] B. Bhattacharya and S. Bhattacharyya, "Parameterized dataflow modeling for dsp systems," *Signal Processing, IEEE Transactions on*, vol. 49, pp. 2408 – 2421, 11 2001.

[5] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi, "PiMM: Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration," in *13th International Conference on Embedded Computer Systems: Architecture, Modeling and Simulation (SAMOS XIII)*, Samos, Greece, Jul. 2013, pp. 41 – 48.

[6] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[7] J. Sérot, The HoCL compiler. Available online at `github.com/jserot/hocl`.

[8] J. Sérot, Formal definition of the HoCL language. Available online at `github.com/jserot/hocl/blob/master/doc`.

[9] C.-J. Hsu, F. Keceli, M.-Y. Ko, S. Shahparnia, and S. S. Bhattacharyya, "Dif: An interchange format for dataflow-based design tools," in *Computer Systems: Architectures, Modeling, and Simulation*, A. D. Pimentel and S. Vassiliadis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 423–432.

[10] N. Sane, H. Kee, G. Seetharaman, and S. Bhattacharyya, "Scalable representation of dataflow graph structures using topological patterns," in *IEEE Workshop On Signal Processing Systems*, 11 2010, pp. 13 – 18.

[11] J. Sérot and F. Berry, "The CAPH Language, Ten Years After," in *Embedded Computer Systems: Architectures, Modeling and Simulation (19th International Conference, SAMOS 2019)*, Aug. 2019, pp. 336–347.

Appendix A

`shuffle` : $\mathsf{int\ list} \rightarrow \alpha\ \mathsf{list} \rightarrow \alpha\ \mathsf{list}$

```
val rec shuffle ks xs = match ks with
    [] -> []
  | k::ks -> nth k xs :: shuffle ks xs
```

where `nth` is the function returning the k^{th} element of a list.

`pipe` : $(\alpha \rightarrow \alpha)\ \mathsf{list} \rightarrow \alpha \rightarrow \alpha$

```
val rec pipe fs x = match fs with
    [] -> x
  | f::fs -> pipe fs (f x)
```

$$\frac{\begin{array}{c}\mathrm{E}_0, \varnothing \vdash \mathit{valdecls} \Rightarrow \mathrm{E}, \mathrm{B}, \mathrm{W} \\ \mathrm{E}, \varnothing \vdash \mathit{nodedecls} \Rightarrow \mathrm{E}'\end{array}}{\vdash \mathbf{program}\ \mathit{typedecls}\ \mathit{valdecls}\ \mathit{nodedecls} \Rightarrow \mathrm{E}'} \quad (\textsc{Program})$$

$$\frac{\mathtt{n} = \mathsf{Node}\langle \mathsf{node}, [\mathrm{id}_1 \mapsto \mathsf{Unit}, \dots, \mathrm{id}_m \mapsto \mathsf{Unit}], [\mathrm{id}'_1, \dots, \mathrm{id}'_n], \mathsf{actor}\rangle}{\mathrm{E}, \mathrm{B} \vdash \mathbf{node}\ \mathrm{id}\ (\mathrm{id}_1 : \mathrm{t}_1, \dots, \mathrm{id}_m : \mathrm{t}_m)\ (\mathrm{id'}_1 : \mathrm{t'}_1, \dots, \mathrm{id'}_n : \mathrm{t'}_n)\ \Rightarrow \mathrm{E} \oplus [\mathrm{id} \mapsto n], \mathrm{B}} \quad (\textsc{NodeDecl1})$$

$$\frac{\begin{array}{c}\mathrm{B} \vdash_i (\mathrm{id}_1 : \mathrm{t}_1, \dots, \mathrm{id}_m : \mathrm{t}_m) \Rightarrow \mathrm{E}_i, \mathrm{B}_i \\ \mathrm{B}_i \vdash_o (\mathrm{id'}_1 : \mathrm{t'}_1, \dots, \mathrm{id'}_n : \mathrm{t'}_n) \Rightarrow \mathrm{E}_o, \mathrm{B}_o \\ \mathrm{E} \oplus \mathrm{E}_i \oplus \mathrm{E}_o,\ \mathrm{B} \oplus \mathrm{B}_i \oplus \mathrm{B}_o \vdash \mathit{valdecls} \Rightarrow \mathrm{B}',\ \mathrm{W}' \\ \mathtt{n} = \mathsf{Node}\langle \mathsf{node}, [\mathrm{id}_1 \mapsto \mathsf{Unit}, \dots, \mathrm{id}_m \mapsto \mathsf{Unit}], [\mathrm{id}'_1, \dots, \mathrm{id}'_n], \mathsf{Graph}\langle \mathrm{B}', \mathrm{W}'\rangle\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash \mathbf{node}\ \mathrm{id}\ (\mathrm{id}_1 : \mathrm{t}_1, \dots, \mathrm{id}_m : \mathrm{t}_m)\ (\mathrm{id'}_1 : \mathrm{t'}_1, \dots, \mathrm{id'}_n : \mathrm{t'}_n)\ \mathit{valdecls} \Rightarrow \mathrm{E} \oplus [\mathrm{id} \mapsto \mathtt{n}], \mathrm{B} \oplus \mathrm{B}_i \oplus \mathrm{B}_o} \quad (\textsc{NodeDecl2})$$

$$\frac{\begin{array}{c}\mathrm{E}, \mathrm{B} \vdash \mathit{expr} \Rightarrow \mathrm{v}, \mathrm{B}', \mathrm{W}' \\ \mathrm{E}, \mathrm{B} \overleftarrow{\oplus} \mathrm{B}' \vdash_p \mathit{pat},\ \mathrm{v} \Rightarrow \mathrm{E}', \mathrm{B}'', \mathrm{W}''\end{array}}{\mathrm{E}, \mathrm{B} \vdash \mathit{pat} = \mathit{expr} \Rightarrow \mathrm{E}',\ \mathrm{B}' \overleftarrow{\oplus} \mathrm{B}'',\ \mathrm{W}' \oplus \mathrm{W}''} \quad (\textsc{Binding})$$

$$\frac{\begin{array}{c}\mathrm{E}, \mathrm{B} \vdash \mathit{exp}_1 \Rightarrow \mathsf{Node}\langle \kappa, [\mathrm{id}_1 \mapsto \ell_1, \dots, \mathrm{id}_{k-1} \mapsto \ell_{k-1}, \mathrm{id}_k \mapsto \mathsf{Unit}, \dots, \mathrm{id}_m \mapsto \mathsf{Unit}], [\mathrm{id}'_1, \dots, \mathrm{id}'_n], \eta\rangle, \mathrm{B}_f, \mathrm{W}_f \\ k < m - 1 \\ \mathrm{E}, \mathrm{B} \vdash \mathit{exp}_2 \Rightarrow \ell, \mathrm{B}_a, \mathrm{W}_a \\ \mathsf{Node}\langle \kappa, [\mathrm{id}_1 \mapsto \ell_1, \dots, \mathrm{id}_{k-1} \mapsto \ell_{k-1}, \mathrm{id}_k \mapsto \ell, \dots, \mathrm{id}_m \mapsto \mathsf{Unit}], [\mathrm{id}'_1, \dots, \mathrm{id}'_n], \eta\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash \mathit{exp}_1\ \mathit{exp}_2 \Rightarrow \mathtt{n}, \mathrm{B}_f \overleftarrow{\oplus} \mathrm{B}_a, \mathrm{W}_f \oplus \mathrm{W}_a} \quad (\textsc{EAppN1})$$

$$\frac{\begin{array}{c}\mathrm{E}, \mathrm{B} \vdash \mathit{exp}_1 \Rightarrow \mathsf{Node}\langle \kappa, [\mathrm{id}_1 \mapsto \ell_1, \dots, \mathrm{id}_{m-1} \mapsto \ell_{m-1}, \mathrm{id}_m \mapsto \mathsf{Unit}], [\mathrm{id}'_1, \dots, \mathrm{id}'_n], \eta\rangle, \mathrm{B}_f, \mathrm{W}_f \\ \mathrm{E}, \mathrm{B} \vdash \mathit{exp}_2 \Rightarrow \ell_m, \mathrm{B}_a, \mathrm{W}_a \\ \mathrm{l} \notin \mathit{Dom}(\mathrm{B}) \\ \forall j.\ 1 \leq j \leq m, \quad \mathrm{k}_j \notin \mathit{Dom}(\mathrm{W}), \quad \mathrm{w}_j = \langle \ell_j, \mathsf{Loc}\langle \mathrm{l}, j\rangle\rangle \\ \mathrm{b} = \mathsf{Box}\langle \mathsf{cat}(\kappa), [1 \mapsto k_1, \dots, m \mapsto k_m], [1 \mapsto \varnothing, \dots, n \mapsto \varnothing]\rangle \\ \mathrm{B}' = [\mathrm{l} \mapsto b] \\ \mathrm{W}' = [k_1 \mapsto w_1, \dots, k_m \mapsto w_m] \\ \mathrm{v}' = \langle \mathsf{Loc}\langle \mathrm{l}, 1\rangle, \dots, \mathsf{Loc}\langle \mathrm{l}, n\rangle\rangle\end{array}}{\mathrm{E}, \mathrm{B} \vdash \mathit{exp}_1\ \mathit{exp}_2 \Rightarrow \mathrm{v}', \mathrm{B}_f \overleftarrow{\oplus} \mathrm{B}_a \overleftarrow{\oplus} \mathrm{B}', \mathrm{W}_f \oplus \mathrm{W}_a \oplus \mathrm{W}'} \quad (\textsc{EAppN2})$$

Fig. 11: Selected semantics inference rules

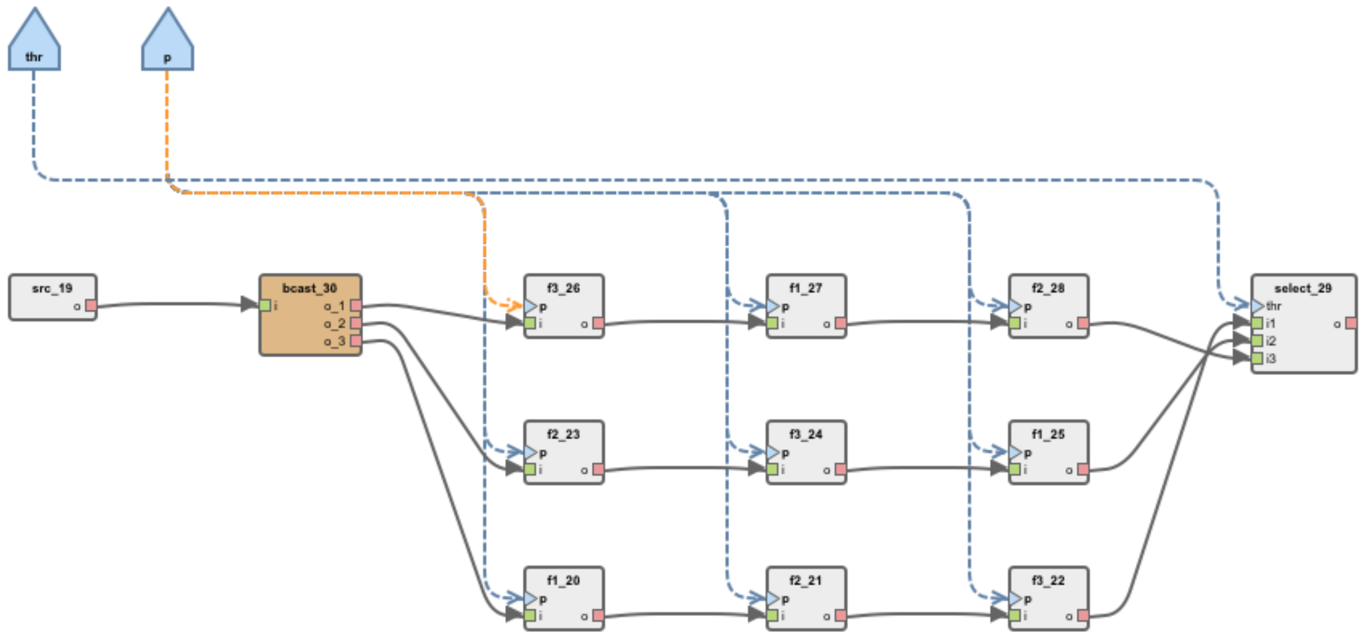


Fig. 12: The DFG of the `multifilt` application, as specified using the Preesm CAD tool

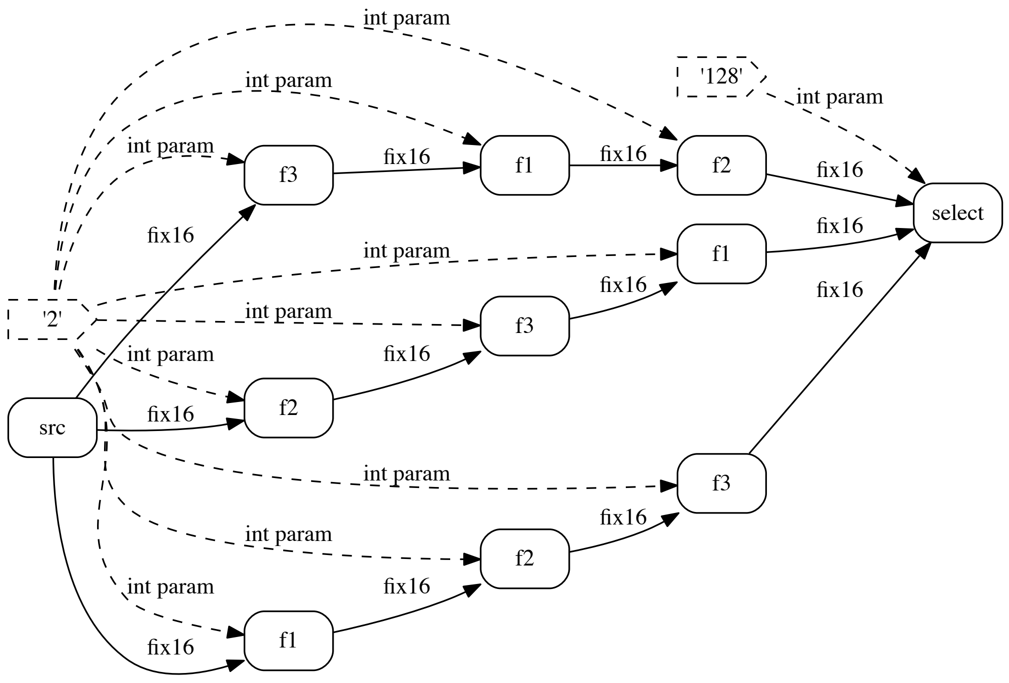


Fig. 13: The DFG resulting from the compilation of the program in Listing 4