Type-Inference Based Short Cut Deforestation (nearly) without Inlining

Olaf Chitil

Lehrstuhl für Informatik II, RWTH Aachen, Germany
chitil@informatik.rwth-aachen.de

Abstract. Deforestation optimises a functional program by transforming it into another one that does not create certain intermediate data structures. Our type-inference based deforestation algorithm performs extensive inlining, but only limited inlining across module boundaries is practically feasible. Therefore we here present a type-inference based algorithm that splits a function definition into a worker definition and a wrapper definition. For deforestation we only need to inline the small wrappers which transfer the required information. We show that we even can deforest definitions of functions that consume their own result with the worker/wrapper scheme, in contrast to the original algorithm with inlining.

1 Type-Inference-Based Short Cut Deforestation

In lazy functional programs two functions are often glued together by an intermediate data structure that is produced by one function and consumed by the other. For example, the function `any`, which tests whether any element of a list `xs` satisfies a given predicate `p`, may be defined as follows in Haskell [13]:

```
any :: (a -> Bool) -> [a] -> Bool

any p xs = or (map p xs)
```

The function `map` applies `p` to all elements of `xs` yielding a list of boolean values. The function `or` combines these boolean values with the logical or operation.

Although lazy evaluation makes this modular programming style practicable [8], it does not come for free. Each cell of the intermediate boolean list has to be allocated, filled, taken apart and finally garbage collected. The following monolithic definition of `any` is more efficient.

```
any p []     = False
any p (x:xs) = p x || any p xs
```

It is the aim of *deforestation* algorithms to automatically transform a functional program into another one that does not create intermediate data structures. We say that a producer (`map p xs`) and a consumer (`or`) of a data structure are *fused*.

1.1 Short Cut Deforestation

The fundamental idea of short cut deforestation [5, 6] is to restrict deforestation to intermediate lists that are consumed by the function `foldr`. This higher-order function uniformly replaces the constructors `(:)` in a list by a given function `c` and the empty list constructor `[]` by a constant `n`:[1]

$$\texttt{foldr c n } [x_1, \ldots, x_k] = x_1 \text{ `}\texttt{c}\text{` } (x_2 \text{ `}\texttt{c}\text{` } (x_3 \text{ `}\texttt{c}\text{` } (\ldots (x_k \text{ `}\texttt{c}\text{` } \texttt{n}) \ldots)))$$

The idea of short cut deforestation is to replaces the list constructors already at compile time. However, the obvious rule

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ e \quad \rightsquigarrow \quad e\ [e_{\texttt{(:)}}/\texttt{(:)},\ e_{\texttt{[]}}/\texttt{[]}]$$

is *wrong*. Consider for example $e =$ `(map p [1,2])`. Here the constructors in `[1,2]` are not to be replaced but those in the definition of `map`, which is not even part of e.

Therefore we need the producer e in a form that makes exactly those list constructors that build the intermediate list explicit such that they can easily be replaced. The solution is to demand that the producer is in the form `(\c n ->` e'`) (:) []`, where the λ-abstracted variables `c` and `n` mark the constructors `(:)` and `[]` of the intermediate list. Then fusion is performed by the rule:

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ \texttt{((\textbackslash c n ->}\ e'\texttt{) (:) [])} \quad \rightsquigarrow \quad \texttt{(\textbackslash c n ->}\ e'\texttt{)}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

The rule removes the intermediate list constructors. A subsequent β-reduction puts the consumer components $e_{\texttt{(:)}}$ and $e_{\texttt{[]}}$ into the places that were before occupied by the list constructors.

We observe that generally $e_{\texttt{(:)}}$ and $e_{\texttt{[]}}$ have different types from `(:)` and `[]`. Hence for this transformation to be type correct, the function `\c n ->` e' must be polymorphic. This can be expressed in Haskell with the help of a special function `build` with a second-order type:

```
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

$$\texttt{foldr}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ \texttt{(build}\ e_{\texttt{p}}\texttt{)} \quad \rightsquigarrow \quad e_{\texttt{p}}\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

In this paper $e_{\texttt{p}}$ will always have the form `\c n ->` e', but this is not necessary for the correctness of the transformation. Strikingly, the polymorphic type of $e_{\texttt{p}}$ already guarantees the correctness [5, 6]. Intuitively, $e_{\texttt{p}}$ can only build its result of type `b` from its two term arguments, because only these have the right types.

[1] Note that `[`$x_1, \ldots, x_k$`]` is only syntactic sugar for x_1`:(`x_2`:(`$\ldots$`(`x_k`:[]))`$\ldots$`)`.

1.2 Derivation of Producers through Type Inference

Whereas using `foldr` for defining list consumers is generally considered as good, modular programming style, programmers can hardly be demanded to use `build`. The idea of the first works on short cut deforestation is that all list-manipulating functions in the standard libraries are defined in terms of `foldr` and `build`. However, thus deforestation is confined to combinations of these standard list functions.

On the other hand we see that, if we can transform a producer e of type `[`τ`]` into the form `build (\c n ->` e'`)`, then the type system guarantees that we have abstracted exactly those list constructors that build the intermediate list. Based on this observation we presented in [1] a type-inference based algorithm which abstracts the intermediate list type and its constructors from a producer to obtain a `build` form.

For the producer `map p [1,2]` for example, this list abstraction algorithm observes, that the intermediate list is constructed by the function `map`. Therefore it inlines the body of `map` to be able to proceed. Afterwards the algorithm decides that the list constructors in the body of `map` have to be abstracted whereas the list constructors in `[1,2]` remain unchanged. With this answer the algorithm terminates successfully. In general, the algorithm recursively inlines all functions that are needed to be able to abstract the result list from the producer, only bounded by an arbitrary code size limit. We recapitulate the algorithm in more detail in Section 3.

1.3 The Problem of Inlining

It is neat that the algorithm determines exactly the functions that need to be inlined, but nonetheless inlining causes problems in practise. Extensive inlining across module boundaries would defeat the idea of separate compilation. Furthermore, inlining, although trivial in principal, is in practise "a black art, full of delicate compromises that work together to give good performance without unnecessary code bloat" [15]. It is best implemented as a separate optimisation pass. Consequently, we would like to use our list abstraction algorithm without it having to perform inlining itself.

To separate deforestation from inlining we split each definition of a list-producing function into a possibly large definition of a worker and a small definition of a wrapper. The latter is inlined everywhere, also across module boundaries, and transfers enough information to permit short cut deforestation. The worker may be inlined by a separate inlining transformation but need not be inlined to enable deforestation.

A worker/wrapper scheme has first been used for propagating strictness information [14]. More importantly, Gill suggested a worker/wrapper scheme for the original short cut deforestation method [5]. In Section 4 we will present our worker/wrapper scheme and also explain why it is more expressive than Gill's. Subsequently we show in Section 5 how the list-producing function definitions

$$\begin{array}{rl}
\text{Type constructors} & C ::= \texttt{[]} \mid \texttt{Int} \mid \ldots \\
\text{Type variables} & \alpha, \beta, \gamma, \delta \\
\text{Types} & \tau ::= C\,\overline{\tau} \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\tau \\
\text{Term variables} & x, c \\
\text{Terms} & e ::= x \mid \lambda x : \tau.e \mid e_1\,e_2 \mid \texttt{case}\ e\ \texttt{of}\ \{c_i\,\overline{x}_i \rightarrow e_i\}_{i=1}^{k} \mid \\
& \qquad \texttt{let}\ \{x_i : \tau_i = e_i\}_{i=1}^{k}\ \texttt{in}\ e \mid \lambda\alpha.e \mid e\,\tau
\end{array}$$

Fig. 1. Terms and types of the language

$$\frac{}{\Gamma + x : \tau \vdash x : \tau}\ \textsc{var}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \rightarrow \tau_2}\ \textsc{term abs} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\,e_2 : \tau}\ \textsc{term app}$$

$$\frac{\forall i = 1..k \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash e_i : \tau_i \quad \Gamma + \{x_j : \tau_j\}_{j=1}^{k} \vdash e : \tau}{\Gamma \vdash \texttt{let}\ \{x_i : \tau_i = e_i\}_{i=1}^{k}\ \texttt{in}\ e : \tau}\ \textsc{let}$$

$$\frac{\Gamma \vdash e : C\,\overline{\rho} \quad \Gamma(c_i) = \forall\overline{\alpha}.\overline{\rho}_i \rightarrow C\,\overline{\alpha} \quad \Gamma + \{\overline{x}_i : \overline{\rho}_i[\overline{\rho}/\overline{\alpha}]\} \vdash e_i : \tau \quad \forall i = 1..k}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{c_i\,\overline{x}_i \mapsto e_i\}_{i=1}^{k} : \tau}\ \textsc{case}$$

$$\frac{\Gamma \vdash e : \tau \quad \alpha \notin \text{freeTyVar}(\Gamma)}{\Gamma \vdash \lambda\alpha.e : \forall\alpha.\tau}\ \textsc{type abs} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e\,\rho : \tau[\rho/\alpha]}\ \textsc{type app}$$

Fig. 2. Type system

of a program are split into wrappers and workers by an algorithm that is based on our list abstraction algorithm without using inlining.

In Section 6 we study definitions of functions that consume their own result. These cannot be deforested by our original algorithm with inlining but can be deforested with the worker/wrapper scheme. To split these function definitions into the required worker and wrapper definitions we need to extend our worker/wrapper split algorithm. As basis we use Mycroft's extension of the Hindley-Milner type inference algorithm by polymorphic recursion [12].

2 The Second-Order Typed Language

We use a small functional language with second-order types, which is similar to the intermediate language Core used inside the Glasgow Haskell compiler [4]. The syntax is defined in Figure 1 and the type system in Figure 2. The language is essentially the second-order typed λ-calculus augmented with `let` for arbitrary mutual recursion and `case` for decomposition of algebraic data structures. We view a typing environment Γ as both a mapping from variables to types and a set of tuples $x : \tau$. The operator $+$ combines two typing environments under the assumption that their domains are disjunct. We abbreviate $\Gamma + \{x:\tau\}$ by $\Gamma + x:\tau$. Data constructors c are just special term variables. The language does not have explicit definitions of algebraic data types like `data` $C\,\overline{\alpha}$ `=` $c_1\,\overline{\tau}_1$ `|` $\ldots$ `|` $c_k\,\overline{\tau}_k$. Such a definition is implicitly expressed by having the data constructors in the typing environment: $\Gamma(c_i) = \tau_{1,i} \to \ldots \to \tau_{n_i,i} \to C\,\overline{\alpha} = \overline{\tau}_i \to C\,\overline{\alpha}$. Hence for the polymorphic type list, which we write `[`α`]` instead of `[]` α, we have $\Gamma($`(:)`$) = \forall\alpha.\alpha \to$ `[`α`]` $\to$ `[`α`]` and $\Gamma($`[]`$) = \forall\alpha.$`[`α`]`. The functions `foldr` and `build` are defined as follows

$$
\begin{array}{l}
\texttt{foldr} : \forall\alpha.\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
\quad = \lambda\alpha.\,\lambda\beta.\,\lambda\texttt{c}{:}\alpha \to \beta \to \beta.\,\lambda\texttt{n}{:}\beta.\,\lambda\texttt{xs}{:}[\alpha].\,\texttt{case xs of}\ \{ \\
\qquad \texttt{[]} \quad \to \texttt{n} \\
\qquad \texttt{y:ys} \to \texttt{c y (foldr}\ \alpha\ \beta\ \texttt{c n ys)}\}
\end{array}
$$

$$
\begin{array}{l}
\texttt{build} : \forall\alpha.(\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta) \to [\alpha] \\
\quad = \lambda\alpha.\,\lambda\texttt{g}{:}\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta.\,\texttt{g}\ [\alpha]\ \texttt{((:)}\ \alpha\texttt{)}\ \texttt{([]}\ \alpha\texttt{)}
\end{array}
$$

and the fusion rule takes the form:

$$\texttt{foldr}\ \tau_1\ \tau_2\ e_{\texttt{(:)}}\ e_{\texttt{[]}}\ \texttt{(build}\ \tau_1\ e_{\texttt{p}}\texttt{)} \quad \leadsto \quad e_{\texttt{p}}\ \tau_2\ e_{\texttt{(:)}}\ e_{\texttt{[]}}$$

3 List Abstraction through Type Inference

Our list abstraction algorithm is described in detail in [1]. To understand its mode of operation we study an example. We have to start with the typing of the producer from which we want to abstract the produced list:[2]

$$
\begin{array}{l}
\{\texttt{mapInt}:(\texttt{Int}{\to}\texttt{Int}){\to}\ [\texttt{Int}] \to [\texttt{Int}],\ \texttt{inc}:\texttt{Int}{\to}\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int}, \\
\ \texttt{(:)}:\forall\alpha.\alpha \to [\alpha] \to [\alpha],\ \texttt{[]}:\forall\alpha.[\alpha]\} \\
\vdash \texttt{mapInt inc ((:) Int 1 ((:) Int 2 ([] Int))) : [Int]}
\end{array}
$$

The algorithm replaces every list constructor application `(:) Int`, respectively `[] Int`, by a different variable c_i, respectively n_i. Furthermore, the types in the expression and in the typing environment have to be modified. To use the existing ones as far as possible, we just replace every list type `[Int]` by a new type variable. Furthermore, we add c_i : `Int` $\to \gamma_i \to \gamma_i$, respectively $n_i : \gamma_i$, to the typing environment, where γ_i is a new type variable for every variable c_i, respectively n_i.

$$
\begin{array}{l}
\{\texttt{mapInt}:(\texttt{Int}{\to}\texttt{Int}){\to}\ \gamma_1 \to \gamma_2,\ \texttt{inc}:\texttt{Int}{\to}\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int}, \\
\ \texttt{n}_1:\gamma_3,\ \texttt{c}_1:\texttt{Int}{\to}\ \gamma_4 \to \gamma_4,\ \texttt{c}_2:\texttt{Int}{\to}\ \gamma_5 \to \gamma_5\} \\
\vdash \texttt{mapInt inc (c}_1\ \texttt{1 (c}_2\ \texttt{2 n}_1\texttt{)) :}\ \gamma
\end{array}
$$

[2] We only consider a monomorphic version of `map`. Inlineable polymorphic functions require an additional instantiation step which we skip here (see [1], Section 4.2).

This invalid typing with type variables is the input to a modified version of the Hindley-Milner type inference algorithm [2, 11]. On the one hand the algorithm was extended to cope with explicit type abstraction and application. On the other hand the type generalisation step (type closure) at `let` bindings was dropped. The type inference algorithm replaces some of the type variables so that the typing is again derivable from the type inference rules, that is, the expression is well-typed in the type environment. Note that type inference cannot fail, because the typing we start with is derivable. We just try to find a more general typing.

$$\begin{array}{l}\{\texttt{mapInt}:(\texttt{Int}\to\texttt{Int})\to \gamma_1 \to \gamma,\ \texttt{inc}:\texttt{Int}\to\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int},\\ \ \texttt{n}_1:\gamma_1,\ \texttt{c}_1:\texttt{Int}\to \gamma_1 \to \gamma_1,\ \texttt{c}_2:\texttt{Int}\to \gamma_1 \to \gamma_1\}\\ \vdash \texttt{mapInt inc (c}_1\texttt{ 1 (c}_2\texttt{ 2 n}_1\texttt{)) : }\gamma\end{array}$$

The type of the expression is a type variable which can be abstracted, but this type variable also appears in the type of the function `mapInt`. So the definition of `mapInt` has to be inlined, all lists types and list constructors be replaced by new variables and type inference be continued.

$$\begin{array}{l}\{\texttt{inc}:\texttt{Int}\to\texttt{Int},\ \texttt{1}:\texttt{Int},\ \texttt{2}:\texttt{Int},\ \texttt{n}_1:\texttt{[Int]},\ \texttt{n}_2:\gamma,\\ \ \texttt{c}_1:\texttt{Int}\to\texttt{[Int]}\to\texttt{[Int]},\ \texttt{c}_2:\texttt{Int}\to\texttt{[Int]}\to\texttt{[Int]},\ \texttt{c}_3:\texttt{Int}\to \gamma \to \gamma\}\\ \vdash \texttt{let mapInt : (Int}\to\texttt{Int)}\to\texttt{[Int]}\to \gamma\\ \qquad\qquad\quad \texttt{=}\ \lambda\texttt{f:Int}\to\texttt{Int.foldr Int}\ \gamma\ \texttt{(}\lambda\texttt{v:Int.}\lambda\texttt{w:}\gamma\texttt{.c}_3\texttt{ (f v) w) n}_2\\ \quad \texttt{in mapInt inc (c}_1\texttt{ 1 (c}_2\texttt{ 2 n}_1\texttt{)) : }\gamma\end{array}$$

Now the type of the expression is still a type variable that, however, does not occur in the typing environment except in the types of the c_i and n_i. Hence the algorithm terminates successfully. The typing environment tells us that $\texttt{c}_3$ and $\texttt{n}_2$ construct the result of the producer whereas $\texttt{c}_1$, $\texttt{c}_2$, and $\texttt{n}_1$ have to construct lists that are internal to the producer. So the type and the constructors of the produced list can be abstracted as follows:

$$\begin{array}{l}\lambda\gamma.\ \lambda\texttt{c:Int}\to \gamma \to \gamma.\ \lambda\texttt{n:}\gamma.\\ \quad \texttt{let mapInt : (Int}\to\texttt{Int)}\to\texttt{[Int]}\to \gamma\\ \qquad\qquad\quad \texttt{=}\ \lambda\texttt{f:Int}\to\texttt{Int. foldr Int}\ \gamma\ \texttt{(}\lambda\texttt{v:Int.}\lambda\texttt{w:}\gamma\texttt{.c (f v) w) n}\\ \quad \texttt{in mapInt inc ((:) Int 1 ((:) Int 2 ([] Int)))}\end{array}$$

This list abstracted producer is suitable as argument for `build`. In reality, our short cut deforestation algorithm never explicitly constructs this `build` form. The deforestation algorithm searches for occurrences of `foldr`, abstracts the result list from the producer and then directly applies the fusion rule.

4 The Worker/Wrapper Scheme

To be able to abstract the result list from a producer without using inlining, all list constructors that produce the result list already have to be present in the producer. Therefore we split every definition of a function that produces a list into a definition of a worker and a definition of a wrapper. The definition of the worker is obtained from the original definition by abstracting the result list type and its list constructors. The definition of the wrapper, which calls the worker, contains all the list constructors that contribute to the result list. For example, we split the definition of `map`

```
map : ∀α.∀β. (α → β)→[α]→[β]
    = λα. λβ. λf:α → β.
        foldr α [β] (λv:α. λw:[β]. (:) β (f v) w) ([] β)
```

into definitions of a worker `mapW` and a wrapper `map`:

```
mapW : ∀α.∀β.∀γ. (β → γ → γ)→ γ →  (α → β)→[α]→ γ
     = λα. λβ. λγ. λc:β → γ → γ. λn:γ. λf:α → β.
         foldr α γ (λv:α. λw:γ. c (f v) w) n

map : ∀α.∀β. (α → β)→[α]→[β]
    = λα. λβ. mapW α β [β] ((:) β) ([] β)
```

For deforestation we only need to inline the wrapper. Consider for example deforestation of the body of the definition of `any`:

```
   or (map τ Bool p xs)
⇝ {inlining of or and map}
   foldr Bool Bool (||) False
     (mapW τ Bool [Bool] ((:) Bool) ([] Bool) p xs)
⇝ {list abstraction from the producer}
   foldr Bool Bool (||) False
     (build Bool (λγ. λc:β→γ→γ. λn:γ. mapW τ Bool γ c n p xs))
⇝ {fusion and subsequent β-reduction}
   mapW τ Bool Bool (||) False p xs
```

It is left to the standard inliner, if `mapW` is inlined. Across module boundaries or if its definition is large, a worker may not be inlined. This is, however, irrelevant for deforestation.

Note that in the definition of the worker we insert the new λ-abstraction between the type abstractions and the term abstractions. We cannot insert the new term abstractions in front of the original type abstractions, because the list type `[`β`]`, from which we abstract, contains the type variable β which is bound in the type of the function. To insert the new abstractions before the original term abstractions has two minor advantages. First, we thus do not require that all term arguments are λ-abstracted at the top of the original definition body. Second, the wrapper can be inlined and β-reduced even at call sites where it is only partially applied, because its definition partially applies the worker.

4.1 Functions that Produce Several Lists

A worker even can abstract from several lists. For example, the definition of the function `unzip`, which produces two lists, can be split into the following worker and wrapper definitions:

$$\begin{array}{l}
\mathtt{unzipW} : \forall\alpha.\forall\beta.\forall\gamma.\forall\delta.(\alpha\to\gamma\to\gamma)\to\gamma\to(\beta\to\delta\to\delta)\to\delta\to\mathtt{[(}\alpha\mathtt{,}\beta\mathtt{)]}\to(\gamma\mathtt{,}\delta)\\
\qquad = \lambda\alpha.\,\lambda\beta.\,\lambda\gamma.\,\lambda\delta.\,\lambda\mathtt{c}_1 : \alpha\to\gamma\to\gamma.\,\lambda\mathtt{n}_1 : \gamma.\,\lambda\mathtt{c}_2 : \beta\to\delta\to\delta.\,\lambda\mathtt{n}_2 : \delta.\\
\qquad\quad \mathtt{foldr}\ (\alpha\mathtt{,}\beta)\ (\gamma\mathtt{,}\delta)\\
\qquad\qquad (\lambda\mathtt{y}:(\alpha\mathtt{,}\beta).\lambda\mathtt{u}:(\gamma\mathtt{,}\delta).\mathtt{case\ y\ of}\,\{\mathtt{(v,w)}\to\mathtt{case\ u\ of}\,\{\\
\qquad\qquad\quad \mathtt{(vs,ws)}\to\mathtt{(,)}\ \gamma\ \delta\ (\mathtt{c}_1\ \mathtt{v\ vs})\ (\mathtt{c}_2\ \mathtt{w\ ws})\ \}\})\\
\qquad\qquad (\mathtt{(,)}\ \gamma\ \delta\ \mathtt{n}_1\ \mathtt{n}_2)
\end{array}$$

$$\begin{array}{l}
\mathtt{unzip} : \forall\alpha.\forall\beta.\mathtt{[(}\alpha\mathtt{,}\beta\mathtt{)]}\to(\mathtt{[}\alpha\mathtt{]},\mathtt{[}\beta\mathtt{]})\\
\qquad = \lambda\alpha.\lambda\beta.\ \mathtt{unzipW}\ \alpha\ \beta\ \mathtt{[}\alpha\mathtt{]}\ \mathtt{[}\beta\mathtt{]}\ (\mathtt{(:)}\,\alpha)\ (\mathtt{[]}\,\alpha)\ (\mathtt{(:)}\,\beta)\ (\mathtt{[]}\,\beta)
\end{array}$$

The subsequent transformations demonstrate how the wrapper enables deforestation without requiring inlining of the larger worker:

$$\begin{array}{l}
\quad \mathtt{foldr}\ \tau_1\ \tau_3\ e_{\mathtt{(:)}}\ e_{\mathtt{[]}}\ \ (\mathtt{fst}\ \mathtt{[}\tau_1\mathtt{]}\ \mathtt{[}\tau_2\mathtt{]}\ (\mathtt{unzip}\ \tau_1\ \tau_2\ \mathtt{zs}))\\
\leadsto \{\text{inlining of the wrapper } \mathtt{unzip}\}\\
\quad \mathtt{foldr}\ \tau_1\ \tau_3\ e_{\mathtt{(:)}}\ e_{\mathtt{[]}}\ (\mathtt{fst}\ \mathtt{[}\tau_1\mathtt{]}\ \mathtt{[}\tau_2\mathtt{]}\\
\qquad (\mathtt{unzipW}\ \tau_1\ \tau_2\ \mathtt{[}\tau_1\mathtt{]}\ \mathtt{[}\tau_2\mathtt{]}\ (\mathtt{(:)}\,\tau_1)\ (\mathtt{[]}\,\tau_1)\ (\mathtt{(:)}\,\tau_2)\ (\mathtt{[]}\,\tau_2)\ \mathtt{zs}))\\
\leadsto \{\text{list abstraction from the producer}\}\\
\quad \mathtt{foldr}\ \tau_1\ \tau_3\ e_{\mathtt{(:)}}\ e_{\mathtt{[]}}\ (\mathtt{build}\ \tau_1\ (\lambda\gamma.\,\lambda\mathtt{c}:\tau_1\to\gamma\to\gamma.\,\lambda\mathtt{n}:\gamma.\,\mathtt{fst}\ \gamma\ \mathtt{[}\tau_2\mathtt{]}\\
\qquad (\mathtt{unzipW}\ \tau_1\ \tau_2\ \gamma\ \mathtt{[}\tau_2\mathtt{]}\ \mathtt{c}\ \mathtt{n}\ (\mathtt{(:)}\,\tau_2)\ (\mathtt{[]}\,\tau_2)\ \mathtt{zs}))\\
\leadsto \{\text{fusion and subsequent } \beta\text{-reduction}\}\\
\quad \mathtt{fst}\ \tau_3\ \mathtt{[}\tau_2\mathtt{]}\ (\mathtt{unzipW}\ \tau_1\ \tau_2\ \tau_3\ \mathtt{[}\tau_2\mathtt{]}\ e_{\mathtt{(:)}}\ e_{\mathtt{[]}}\ (\mathtt{(:)}\,\tau_2)\ (\mathtt{[]}\,\tau_2)\ \mathtt{zs})
\end{array}$$

4.2 List Concatenation

The list append function `(++)` is notorious for being difficult to fuse with, because the expression `(++)` τ `xs ys` does not produce the whole result list itself. Only `xs` is copied but not `ys`. However, we can easily define a worker for `(++)` by abstracting not just the result list but simultaneously the type of the second argument:

$$\begin{array}{l}
\mathtt{appW} : \forall\alpha.\forall\gamma.(\alpha\to\gamma\to\gamma)\to\gamma\to\mathtt{[}\alpha\mathtt{]}\to\gamma\to\gamma\\
\qquad = \lambda\alpha.\,\lambda\gamma.\,\lambda\mathtt{c}:\alpha\to\gamma\to\gamma.\,\lambda\mathtt{n}:\gamma.\,\lambda\mathtt{xs}:\mathtt{[}\alpha\mathtt{]}.\,\lambda\mathtt{ys}:\gamma.\,\mathtt{foldr}\ \alpha\ \gamma\ \mathtt{c\ ys\ xs}
\end{array}$$

$$\begin{array}{l}
\mathtt{(++)} : \forall\alpha.\mathtt{[}\alpha\mathtt{]}\to\mathtt{[}\alpha\mathtt{]}\to\mathtt{[}\alpha\mathtt{]}\\
\qquad = \lambda\alpha.\,\mathtt{appW}\ \alpha\ \mathtt{[}\alpha\mathtt{]}\ (\mathtt{(:)}\ \alpha)\ (\mathtt{[]}\ \alpha)
\end{array}$$

The type of `appW` implies, that we can only abstract the result list constructors of an application of `(++)`, if we can abstract the result list constructors of its second argument. We believe that this will seldom restrict deforestation in practise. For example the definition

$\texttt{concat} : \forall\alpha.\texttt{[[}\alpha\texttt{]]} \to \texttt{[}\alpha\texttt{]}$
$\quad\quad = \lambda\alpha.\,\texttt{foldr [}\alpha\texttt{] [}\alpha\texttt{] ((++) }\alpha\texttt{) ([] }\alpha\texttt{)}$

can be split into a worker and a wrapper definition thanks to the wrapper `appW`:

$\texttt{concatW} : \forall\alpha.\forall\gamma.(\alpha\to\gamma\to\gamma) \to \gamma \to \texttt{[[}\alpha\texttt{]]} \to \gamma$
$\quad\quad = \lambda\alpha.\,\lambda\gamma.\,\lambda\texttt{c}{:}\alpha\to\gamma\to\gamma.\,\lambda\texttt{n}{:}\gamma.\,\texttt{foldr [}\alpha\texttt{] }\gamma\texttt{ (appW }\alpha\ \gamma\texttt{ c n) n}$

$\texttt{concat} : \forall\alpha.\texttt{[[}\alpha\texttt{]]} \to \texttt{[}\alpha\texttt{]}$
$\quad\quad = \lambda\alpha.\,\texttt{concatW }\alpha\texttt{ [}\alpha\texttt{] ((:) }\alpha\texttt{) ([] }\alpha\texttt{)}$

4.3 Gill's Worker/Wrapper Scheme

Gill does not consider any automatic list abstraction but assumes that some list producing functions (those in the standard libraries) are defined in terms of `build`. He developed a worker/wrapper scheme ([5], Section 7.4) to inline `build` as far as possible without inlining of large expressions. Note that for the `foldr`/`build` fusion rule it is only necessary that the producer is in `build` form, the argument of `build` is of no interest but is just rearranged by the transformation.

So, for example, Gill starts with the definition of `map` in `build` form

```
map f xs = build (\c n -> foldr (c . f) n xs)
```

and splits it up as follows:

```
mapW :: (a -> b) -> [a] -> (b -> c -> c) -> c -> c
mapW f xs c n = foldr (c . f) n xs

map f xs = build (mapW f xs)
```

The similarity to our worker/wrapper scheme becomes obvious. when we inline `build` in these definitions. We do not use `build`, because we do not need it and its use limits the expressive power of Gill's worker/wrapper scheme. The function `build` can only wrap a producer that returns a single list. Hence, for example, the function `unzip` cannot be expressed in terms of `build` and therefore its definition cannot be split into a worker and a wrapper. Also `(++)` cannot be defined in terms of `build`. Gill defines a further second-order typed function `augment` to solve the latter problem. Additionally, because of `build` a wrapper cannot be inlined when it is only partially applied. Note that in Section 4.2 we inlined the partially applied function `(++)` in the definition of `concat` to derive its worker definition. Finally, a `build` in a producer hinders type-inference based fusion. For example, from the producer `build (mapW f xs)` no list constructors can be abstracted, because they are hidden by `build`. We have to inline `build` to proceed with list abstraction.

Altogether we see that list abstraction provides the means for a much more flexible worker/wrapper scheme.

4.4 Effects on Performance

As Gill already noticed, there is a substantial performance difference between calling a function as originally defined (`map` τ' τ) and calling a worker with list constructors as arguments (`mapW` τ' τ `[`τ`]` `((:)`τ`)` `([]`τ`))`. Constructing a list with list constructors that are passed as arguments is more expensive than constructing the list directly. After deforestation all calls to workers that were not needed still have list constructors as arguments. So, as Gill suggested, we must have for each worker a version which is specialised to the list constructors and replace the call to each unused worker by a call to its specialised version. We could use the original, unsplit definition of the function, but by specialising the worker definition we can profit from any optimisations, especially deforestation, that were performed inside the worker definition. Note that we only derive one specialised definition for every worker.

The worker/wrapper scheme increases code size through the introduction of wrapper and specialised worker definitions. However, this increase is bounded in contrast to the code increase that is caused by our original list abstraction algorithm with inlining. An implementation will show if the code size increase is acceptable. Note that the definitions of workers that are not needed for deforestation can be removed by standard dead code elimination after worker specialisation has been performed.

5 The Worker/Wrapper Split Algorithm

For the worker/wrapper scheme each list-producing function definition has to be split into a worker and a wrapper definition. A worker definition is easily derived from a non-recursive function definition by application of the list abstraction algorithm. Consider the definition of `map` as given in Section 4. Only the preceding type abstractions have to be removed to form the input for the list abstraction algorithm:

$$\begin{array}{l}\{\texttt{foldr}:\forall\alpha.\forall\beta.(\alpha\to\beta\to\beta)\to\beta\to[\alpha]\to\beta,\\ \;\texttt{(:)}:\forall\alpha.\alpha\to[\alpha]\to[\alpha],\ \texttt{[]}:\forall\alpha.[\alpha]\}\\ \vdash \lambda\texttt{f}:\alpha\to\beta.\,\texttt{foldr}\ \alpha\ [\beta]\ (\lambda\texttt{v}:\alpha.\lambda\texttt{w}:[\beta].\,\texttt{(:)}\ \beta\ (\texttt{f v})\ \texttt{w})\ (\texttt{[]}\ \beta)\\ :\ (\alpha\to\beta)\to[\alpha]\to[\beta]\end{array}$$

The algorithm returns:

$$\lambda\gamma.\,\lambda\texttt{c}:\beta\to\gamma\to\gamma.\,\lambda\texttt{n}:\gamma.\,\lambda\texttt{f}:\alpha\to\beta.\,\texttt{foldr}\ \alpha\ \gamma\ (\lambda\texttt{v}:\alpha.\lambda\texttt{w}:[\beta].\texttt{c}\ (\texttt{f v})\ \texttt{w})\ \texttt{n}$$

So the result list can be abstracted. The readdition of the abstraction of α and β to obtain the worker definition and the construction of the wrapper definition is straightforward. In the case that no list can be abstracted, no worker/wrapper split takes place.

Because all list types in the the type of the processed function are replaced by type variables, also the workers of `(++)`, `concat` and `unzip` are derived by this algorithm.

5.1 Derivation of Workers of Recursively Defined Functions

In all previous examples recursion was hidden by `foldr`. For recursive definitions we have to slightly modify the list abstraction algorithm. Consider the recursively defined function `enumFrom` which returns an infinite list of integers, starting with a given integer `x`:

```
enumFrom : Int → [Int]
         = λx:Int.(:) Int x (enumFrom (+ x 1))
```

The input typing for the type inference algorithm must contain a type assignment in the typing environment for the recursive call. The typing environment assigns the same type to this identifier as is assigned to the whole definition body. This corresponds to the processing of recursive `lets` in the Hindley-Milner type inference algorithm.

```
{enumFrom:Int → γ₁, +:Int → Int, 1:Int, c:Int → γ₂ → γ₂}
⊢ λx:Int.c x (enumFrom (+ x 1))
: Int → γ₁
```

Type inference yields:

```
{enumFrom:Int → γ, +:Int → Int, 1:Int, c:Int → γ → γ}
⊢ λx:Int.c x (enumFrom (+ x 1))
: Int → γ
```

The construction of the worker and wrapper definitions is again straightforward:

```
enumFromW : ∀γ.(Int→γ→γ) → γ → Int → γ
          = λγ. λc:Int→γ→γ. λn:γ.
               λx:Int.c x (enumFromW γ c n (+ x 1))

enumFrom : Int → [Int]
         = enumFromW [Int] ((:) Int) ([] Int)
```

Note that to abstract the list the recursive call in the definition of the worker must be to the worker itself, not to the wrapper.

If a recursively defined producer f is polymorphic, that is, $f : \forall\overline{\alpha}.\tau$, then we do not only have to remove the abstraction of the type variables $\overline{\alpha}$ from the definition body, but also have to replace all recursive calls $f\,\overline{\alpha}$ by a new identifier f' before type inference.

5.2 Traversal Order

The worker/wrapper split algorithm splits each `let` defined block of mutually recursive definitions separately. In the example of `concat` in Section 4.2 the split was only possible after the wrapper of `(++)` had been inlined. Hence the split algorithm must traverse the program in top-down order and inline wrappers in the remaining program directly after they were derived.

Additionally, definitions can be nested, that is, the right-hand-side of a `let` binding can contain another `let` binding. Here the inner definitions have to be split first. Their wrappers can then be inlined in the body of the outer definition and thus enable the abstraction of more lists from the outer definition.

6 Functions that Consume their Own Result

There are definitions of list functions that consume their own result. The most simple example is the definition of the function that reverses a list in quadratic time:

```
reverse : ∀α.[α] → [α]
        = λα. λxs:[α]. case xs of {
            []   → [] α
            y:ys → (++) α (reverse α ys) ((:) α y ([] α)) }
```

This definition can be split into the following worker and wrapper definitions:

```
reverseW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
         = λα. λγ. λc:α → γ → γ. λn:γ. λxs:[α]. case xs of {
             []   → n
             y:ys → appW α γ c n
                      (reverseW α [α] ((:) α) ([] α) ys) (c y n) }

reverse : ∀α.[α] → [α]
        = λα. reverseW α [α] ((:) α) ([] α)
```

In this definition of `reverseW` the worker `appW` can be inlined:

```
reverseW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
         = λα. λγ. λc:α → γ → γ. λn:γ. λxs:[α]. case xs of {
             []   → n
             y:ys → foldr α γ c (c y n)
                      (reverseW α [α] ((:) α) ([] α) ys) }
```

Then short cut fusion and subsequent β-reduction yields:

```
reverseW : ∀α.∀γ.(α → γ → γ) → γ → [α] → γ
         = λα. λγ. λc:α → γ → γ. λn:γ. λxs:[α]. case xs of {
             []   → n
             y:ys → reverseW α γ c (c y n) ys }
```

The deforested version performs list reversal in linear time. The worker argument that abstracts the list constructor `[]` is used as an accumulator.

The list abstraction algorithm with inlining cannot achieve this transformation of the quadratic version into the linear version. To abstract the intermediate list, that algorithm would inline the definition of `reverse`. Then the intermediate list would be eliminated successfully, but the inlined definition of `reverse` would contain a new starting point for deforestation which would lead to new inlining of `reverse` ... The quadratic version creates at run time an intermediate list between each recursive call. To remove all these intermediate lists through a finite amount of transformation the worker/wrapper scheme is required.

6.1 Worker Derivation with Polymorphic Recursion

Unfortunately, the worker `reverseW` cannot be derived by the algorithm described in Section 5. Compare the recursive definition of `reverseW` (before deforestation) with the recursive definition of `enumFromW`. The former is polymorphically recursive, that is, a recursive call uses type arguments different from the abstracted type variables. Obviously, functions that consume their own result need such polymorphically recursive workers.

Typability in the Hindley-Milner type system with polymorphic recursion is semi-decidable [7, 9], that is, there are algorithms which do infer the most general type of an expression within the Hindley-Milner type system with polymorphic recursion if it is typable. However, if the expression is not typable these algorithms may diverge. Fortunately, the input of the worker/wrapper split algorithm is typable, we only try to find a more general type than we have.

To derive a possibly polymorphically recursive worker definition, we build on Mycroft's extension of the Hindley-Milner type inference algorithm [12]. We start with the most general worker type possible, which is obtained from the original type by replacing every list type by a new type variable and abstracting the list type and its list constructors.

$$
\begin{array}{l}
\{\texttt{reverseW}:\forall\alpha.\forall\delta_1.\forall\delta_2.(\alpha\to\delta_1\to\delta_1)\to\delta_1\to(\alpha\to\delta_2\to\delta_2)\to\delta_2\to(\delta_1\to\delta_2),\\
\ \texttt{appW}:\forall\alpha.\forall\delta.(\alpha\to\delta\to\delta)\to\delta\to\texttt{[}\alpha\texttt{]}\to\delta\to\delta,\ \texttt{n}_1:\gamma_1,\ \texttt{n}_2:\gamma_2,\ \texttt{n}_3:\gamma_3,\ \texttt{n}_4:\gamma_4,\\
\ \texttt{n}_5:\gamma_5,\ \texttt{c}_1:\alpha\to\gamma_6\to\gamma_6,\ \texttt{c}_2:\alpha\to\gamma_7\to\gamma_7,\ \texttt{c}_3:\alpha\to\gamma_8\to\gamma_8,\ \texttt{c}_4:\alpha\to\gamma_9\to\gamma_9\}\\
\vdash \lambda\texttt{xs}:\gamma_{10}\texttt{. case xs of }\{\\
\qquad \texttt{[]}\quad\ \ \to \texttt{n}_1\\
\qquad \texttt{y:ys}\to \texttt{appW}\ \alpha\ \gamma_{11}\ \texttt{c}_1\ \texttt{n}_2\\
\qquad\qquad\qquad (\texttt{reverseW}\ \alpha\ \gamma_{12}\ \gamma_{13}\ \texttt{c}_2\ \texttt{n}_3\ \texttt{c}_3\ \texttt{n}_4\ \texttt{ys})\ (\texttt{c}_4\ \texttt{y}\ \texttt{n}_5)\ \}\\
:\ \gamma_{14}\to\gamma_{15}
\end{array}
$$

We perform type inference to obtain a first approximation of the type of the worker:

$$
\begin{array}{l}
\{\texttt{reverseW}:\forall\alpha.\forall\delta_1.\forall\delta_2.(\alpha\to\delta_1\to\delta_1)\to\delta_1\to(\alpha\to\delta_2\to\delta_2)\to\delta_2\to(\delta_1\to\delta_2),\\
\ \texttt{appW}:\forall\alpha.\forall\delta.(\alpha\to\delta\to\delta)\to\delta\to\texttt{[}\alpha\texttt{]}\to\delta\to\delta,\ \texttt{n}_1:\gamma,\ \texttt{n}_2:\gamma,\ \texttt{n}_3:\texttt{[}\alpha\texttt{]},\ \texttt{n}_4:\texttt{[}\alpha\texttt{]},\\
\ \texttt{n}_5:\gamma,\ \texttt{c}_1:\alpha\to\gamma\to\gamma,\ \texttt{c}_2:\alpha\to\texttt{[}\alpha\texttt{]}\to\texttt{[}\alpha\texttt{]},\ \texttt{c}_3:\alpha\to\texttt{[}\alpha\texttt{]}\to\texttt{[}\alpha\texttt{]},\ \texttt{c}_4:\alpha\to\gamma\to\gamma\}\\
\vdash \lambda\texttt{xs:[}\alpha\texttt{]. case xs of }\{\\
\qquad \texttt{[]}\quad\ \ \to \texttt{n}_1\\
\qquad \texttt{y:ys}\to \texttt{appW}\ \alpha\ \gamma\ \texttt{c}_1\ \texttt{n}_2\\
\qquad\qquad\qquad (\texttt{reverseW}\ \alpha\ \texttt{[}\alpha\texttt{]}\ \texttt{[}\alpha\texttt{]}\ \texttt{c}_2\ \texttt{n}_3\ \texttt{c}_3\ \texttt{n}_4\ \texttt{ys})\ (\texttt{c}_4\ \texttt{y}\ \texttt{n}_5)\ \}\\
:\ \texttt{[}\alpha\texttt{]}\to\gamma
\end{array}
$$

Subsequently we infer anew the type of the definition body, this time under the assumption that reverseW has the type $\forall\alpha.\forall\gamma.(\alpha{\rightarrow}\gamma{\rightarrow}\gamma) \rightarrow \gamma \rightarrow [\alpha] \rightarrow \gamma$, the result of the first type inference pass. This process iterates until the inferred type is stable, that is input and output type are identical. For our example the second iteration already shows that the result of the first iteration is correct. In general, worker derivation stops latest after $n + 1$ iterations, where n is the number of list types in the type of the original function.

6.2 Further Workers with Polymorphic Recursion

Similar to the example reverse are definitions of functions which traverse a tree to collect all node entries in a list. A straightforward quadratic time definition which uses (++) can be split into a polymorphically recursive worker and a wrapper and then be deforested to obtain a linear time definition which uses an accumulating argument.

A different, fascinating example is the definition of the function inits, which determines the list of initial segments of a list with the shortest first.

```
inits: ∀α.[α] → [[α]]
     = λα. λxs:[α].
         case xs of {
           []   → (:) [α] ([] α) ([] [α])
           y:ys → (:) [α] ([] α) (map [α] [α] ((:) α y)
                                     (inits α ys)) }
```

It is split into the following polymorphically recursive worker and wrapper definitions:

```
initsW: ∀α.∀γ.∀δ.(α→γ→γ) → γ → (γ→δ→δ) → δ → [α] → δ
      = λα. λγ. λδ. λc₁:α→γ→γ. λn₁:γ. λc₂:γ→δ→δ. λn₂:δ. λxs:[α].
          case xs of {
            []   → c₂ n₁ n₂
            y:ys → c₂ n₁ (mapW γ γ δ c₂ n₂ (c₁ y)
                     (initsW α γ [γ] c₁ n₁ ((:) γ) ([] γ) ys)) }

inits: ∀α.[α] → [[α]]
     = λα. initsW α [α] [[α]] ((:) α) ([] α) ((:) [α]) ([] [α])
```

Note the abstraction of both (nested) result lists, which cannot be expressed with build. Fusion can be performed in the definition body of initsW:

```
initsW: ∀α.∀γ.∀δ.(α→γ→γ) → γ → (γ→δ→δ) → δ → [α] → δ
      = λα. λγ. λδ. λc₁:α→γ→γ. λn₁:γ. λc₂:γ→δ→δ. λn₂:δ. λxs:[α].
          case xs of {
            []   → c₂ n₁ n₂
            y:ys → c₂ n₁ (initsW α γ δ c₁ n₁
                            (λv:γ. λw:δ. c₂ (c₁ y v) w) n₂ ys)}
```

The n-queens function as defined in Section 5.1 of [5] is another example in the same spirit.

6.3 Inaccessible Recursive Arguments

Unfortunately, a function may consume its own result but not be defined recursively. For example, the function `reverse` should actually be defined in terms of `foldr`, to enable short cut deforestation with `reverse` as consumer.

```
reverse: ∀α.[α] → [α]
        = λα.foldr α [α]
             (λy:α.λr:[α].(++) α r ((:) α y ([] α))) ([] α)
```

The result list cannot be abstracted, because the recursion argument `r` is not a function with a list type and its constructors as arguments. Here type inference with polymorphic recursion cannot help.

To enable list abstraction we can rewrite the definition as follows (cf. Section 7 of [10]):

```
reverse: ∀α.[α] → [α]
        = λα.foldr α [α]
             (λy:α.λr:(α→[α]→[α])→[α]→[α].
               (++) α (r((:)α)([]α)) ((:)α y ([]α)))
             (λc:α → [α] → [α].λn:[α].n)
             ((:) α)
             ([] α)
```

It is, however, unclear when and how such a lifting of the result type of a function that encapsulates recursion can be done in general.

6.4 Deforestation Changes Complexity

Deforestation of the definition of `reverse` changes its complexity from quadratic to linear time. In case of the definition of `inits`, the change of complexity is more subtle. Both the original definition and the deforested definition take quadratic time to produce their complete result. However, to produce only the outer list of the result, with computation of the list elements still suspended, the original definition still takes quadratic time whereas the deforested version only needs linear time.

A polymorphically recursive worker will nearly always enable deforestation that changes the asymptotic time complexity of a function definition. This power is, however, a double-edged sword. A small syntactic change of a program (cf. previous subsection) may cause deforestation to be no longer applicable, and thus change the asymptotic complexity of the program. It can hence be argued that such far-reaching modifications should be left to the programmer.

7 Summary and Future Work

In this paper we presented an expressive worker/wrapper scheme to perform short cut deforestation (nearly) without inlining. An algorithm which is based on our list abstraction algorithm [1] splits all definitions of list-producing functions of a program into worker and wrapper definitions. The wrapper definitions are small enough to be inlined unconditionally everywhere, also across module boundaries. They transfer the information needed for list abstraction in the split algorithm and the actual deforestation algorithm.

The actual deforestation algorithm searches for occurrences of `foldr`, abstracts the result list from the producer and then directly applies the short cut fusion rule. Further optimisations may be obtained by a subsequent standard inlining pass.

The deforestation algorithm is separate from the worker/wrapper split algorithm. The algorithms may be integrated, but the worker/wrapper split is only performed once whereas it may be useful to repeat deforestation several times, because deforestation and other optimisations may lead to new deforestation opportunities.

Finally, we studied functions that consume their own result. Their definitions can be split and deforested if the split algorithm is extended on the basis of Mycroft's extension of Hindley-Milner type inference to polymorphic recursion. Nonetheless they still raise interesting questions.

We focused on how to derive a producer for short cut deforestation without requiring large-scale inlining. Dually the consumer must be a `foldr` and hence sufficient inlining must be performed in the consumer to expose the `foldr`. If the arguments of the `foldr` are large expressions, the standard inliner will refuse to inline the `foldr` expression. So it seems reasonable to also split consumers into `foldr` wrappers and separate workers for the arguments of `foldr`. This transformation, however, does not require any (possibly type-based) analysis but can be performed directly on the syntactic structure.

The worker/wrapper split algorithm is not as efficient as it could be. The list abstraction algorithm traverses a whole definition body once. Even if we ignore polymorphic recursion, if n `let` bindings are nested, then the body of the inner definition is traversed n times. However, as stated in Section 2, the list abstraction algorithm uses a modified version of the Hindley-Milner type inference algorithm. The abstraction of list types corresponds to the generalisation step of the Hindley-Milner algorithm. The list abstraction algorithm just additionally abstracts list constructors and inserts both type and term abstractions into the program. The Hindley-Milner algorithm recursively traverses a program only once. So we plan to integrate explicit type and term abstraction at `let` bindings into this type inference algorithm to obtain a single pass split algorithm. To deal with polymorphic recursion as well, the type inference algorithm of Emms and Leiß, which integrates semiunification into the Hindley-Milner algorithm, may provide a good basis [3].

We have a working prototype of the list abstraction algorithm with inlining. On this basis we are implementing a simple worker/wrapper split algorithm. The final goal is an implementation in the Glasgow Haskell compiler to apply type-inference based short cut deforestation to real-world programs.

Acknowledgements

I thank Simon Peyton Jones for several comments that inspired this paper. Especially, he drew my attention to producers that consume their own result.

References

1. Olaf Chitil. Type inference builds a short cut to deforestation. *ACM SIGPLAN Notices*, 34(9):249–260, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).
2. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, January 1982.
3. Martin Emms and Hans Leiß. Extending the type checker of Standard ML by polymorphic recursion. *Theoretical Computer Science*, 212(1–2):157–181, February 1999.
4. The Glasgow Haskell compiler. http://www.haskell.org/ghc/.
5. Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Glasgow University, 1996.
6. Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A Short Cut to Deforestation. In *FPCA'93, Conference on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM Press, 1993.
7. F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
8. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
9. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.
10. John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95*, pages 314–323. ACM Press, 1995.
11. Oukseh Lee and Kangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.
12. A. Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, LNCS 167, pages 217–228, Toulouse, France, April 1984. Springer.
13. Simon L. Peyton Jones, John Hughes, et al. Haskell 98: A non-strict, purely functional language. http://www.haskell.org, February 1999.
14. Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, LNCS 523, pages 636–666. Springer Verlag, June 1991.
15. Simon L. Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell compiler inliner. IDL '99, http://www.binnetcorp.com/wshops/IDL99.html, 1999.