Comprehending Finite Maps for Algorithmic Debugging of Higher-Order Functional Programs

Olaf Chitil
University of Kent, UK

Thomas Davie
University of Kent, UK

ABSTRACT

Algorithmic debuggers for higher-order functional languages have to display functional values. Originally functional values had been represented as partial applications of function and constructor symbols, but a recent approach represents functional values as finite maps. The two representations require the computation tree that is central to algorithmic debugging to be structured rather differently. In this paper we present a unifying framework that formally defines algorithmic debugging for both representations in an implementation-independent way. On this basis we prove the soundness of algorithmic debugging with finite maps. Our framework shows how a single implementation can support both forms of algorithmic debugging. The proof exposed that algorithmic debugging with finite maps does not handle arbitrary functional programs, but in current practice the problematic ones are excluded by Haskell's type system. Both framework and proof suggest variations of algorithmic debugging with finite maps and thus are tools for further improvement of this form of debugging.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: processors; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

Keywords

Algorithmic debugging, tracing, functional programming

1. INTRODUCTION

Algorithmic debugging is a semi-automated method of locating faults in declarative programs. Consider the following faulty Haskell program[1]. The function `allOdd` shall determine whether all numbers in a given tree are odd. The worker function `allOddC` uses a continuation to traverse the tree from left to right.

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a

allOdd :: Tree Int -> Bool
allOdd t = allOddC id t True

allOddC :: (Bool->Bool) -> Tree Int -> Bool -> Bool
-- spec: allOddC c t b = b && c (allOdd t)
allOddC c (Leaf n) b = b && c (odd n)
allOddC c (Branch l r) b = allOddC (allOddC c r) l b

odd :: Int -> Bool
odd x = x `mod` 3 == 1

id :: a -> a
id x = x

main = allOdd (Branch (Leaf 7) (Leaf 5))
```

Evaluation of `main` yields the unexpected answer `False`. So which fault causes this erroneous behaviour? A standard algorithmic debugger asks us, the user, a series of questions about the computation, namely whether given equations agree with our intentions or not. Our answers are highlighted in *italics*.

```
1. main = False ?  no
2. allOdd (Branch (Leaf 7) (Leaf 5)) = False ?  no
3. allOddC id (Branch (Leaf 7) (Leaf 5)) True =
     False ?  no
4. allOddC (allOddC id (Leaf 5)) (Leaf 7) True =
     False ?  no
5. odd 7 = True ?  yes
6. allOddC id (Leaf 5) True = False ?  no
7. odd 5 = False ?  no
Fault in definition:  odd x = x `mod` 3 == 1
```

Soon the debugger identifies a *faulty definition* that needs to be modified. Inspecting the definition we find that `3` needs to be replaced by `2`.

Standard algorithmic debugging works, but question 4 indicates a problem: it contains already three occurrences of function symbols (`id` and twice `allOddC`). To answer such a question, we have to consider the intended meaning of all function symbols that appear in the question concurrently.

Standard algorithmic debugging represents functional values as function symbols and their partial applications. The number of function symbol occurrences in a single functional value is unbounded. For many higher-order functional programs, especially those using continuations, combinator libraries or monads, the questions of the standard algorithmic debugger become incomprehensible and thus unanswerable.

[1]We disregard that `main` should have the type `IO ()`.

Hence Pope [11, 12] and later independently Davie and Chitil [5] proposed representing a functional value as a finite map from arguments to results.

With finite maps an algorithmic debugging session looks as follows:

```
1. main = False ?  no
2. allOdd (Branch (Leaf 7) (Leaf 5)) = False ?  no
3. id False = False ?  yes
4. allOddC {False ↦ False}
      (Branch (Leaf 7) (Leaf 5)) True = False ?  no
5. allOddC {False ↦ False} (Leaf 5) True = False ?
      no
6. odd 5 = False ?  no
Fault in definition:  odd x = x ‘mod‘ 3 == 1
```

A finite map includes only arguments to which the function was applied during the computation. When answering a question, the user assumes that the function maps any other argument to the undefined value $\perp$. Every question contains only one function symbol. With a different function representation the meaning of questions changes and hence questions have to be asked in a different order.

Most questions are far easier to understand with finite maps than with partial applications, as plenty of examples in [11, 12, 5] demonstrate. Furthermore, no algorithmic debugger supported λ-abstractions meaningfully before the introduction of finite maps. We discuss another advantage in Section 9.

Pope gives a detailed technical description of his implementation of finite maps in the algorithmic debugger Buddha [12]. This description is specific to his implementation and thus does not support proper comprehension of the principles, proof of correctness and exploration of variations and extensions. For example, if the function `id` was also used in other parts of our program, would the argument of `allOddC` in question 4 look like `{False ↦ False, True ↦ True, 42 ↦ 42, 'c' ↦ 'c', ...}`? Surely we want to have less argument-result pairs, but which ones do we have to include? To answer such questions we formally define a comprehensible model of algorithmic debugging with a finite map representation of functional values.

Our model relates algorithmic debugging to a simple graph reduction semantics. Although we use Haskell's syntax, all definitions and theorems are independent of whether the language semantics is strict or non-strict. Even though algorithmic debugging with functions as partial applications and algorithmic debugging with functions as finite maps use rather differently structured computation trees, we describe them in a single framework. On the practical side this integration shows how a single implementation can support both forms of algorithmic debugging. On the theoretical side it clarifies the differences between both variants of algorithmic debugging. We prove that algorithmic debugging with finite maps is sound; on the way we observe that finite maps are well-defined only if certain programs are excluded, as they are by Haskell's type system. Model and soundness proof allow simple experimentation with variations and extensions of algorithmic debugging; we outline a number of useful ones.

2. OVERVIEW

Algorithmic debugging is based on the representation of the computation, which yielded the erroneous result, as a *computation tree*. Figure 1 shows two computation trees for our tree traversal program, the standard *evaluation dependency tree (EDT)* [10], where functions are represented as partial applications, and the *function dependency tree (FDT)*, where functions are represented as finite maps. We use strings such as `rr` and `rrrra` as nodes, for reasons discussed later. Centrally, each node of a computation tree is labelled with a subcomputation. In the EDT and FDT the subcomputation is a big-step reduction of a function symbol with argument values to its result value: $f\,M_1 \ldots M_k = M_0$. A subcomputation on its own can be judged to be either correct ($\surd$), that is, agreeing with the user's intentions, or incorrect ($\times$). Each node is associated with a slice of the program; here we associate each node with the program equation that was used to reduce the redex $f\,M_1 \ldots M_k$ of the label.

In algorithmic debugging the user's `yes`/`no` answers direct a path through the tree to a node associated with a faulty slice [8]. If in a computation tree all the child nodes of an incorrect node are correct, then this node is said to be *faulty*. A tree is a *computation tree*, if the slice associated with a faulty node is faulty, that is, the program slice disagrees with the user's intentions. Reformulated: if all the subcomputations of the children of a node are correct and the slice associated with the node is not faulty, then the subcomputation of the node itself must be correct. So a computation tree must be compositional. If the root node of a computation tree is incorrect, then algorithmic debugging will locate a faulty node in the tree (Propositions 1 and 3 in [8]).

The main difference between the EDT and the FDT is their structure. In the EDT a node $g\,N_1 \ldots N_l = N_0$ is a child of a node $f\,M_1 \ldots M_k = M_0$, if $g\,N_1 \ldots N_l$ was called from function f, more precisely, if the *application* $g\,N_1 \ldots N_l$ appears in the right hand side of the definition of f. In contrast, in the FDT a node $g\,N_1 \ldots N_l = N_0$ is a child of a node $f\,M_1 \ldots M_k = M_0$, if the *function symbol* g appears in the right hand side of the definition of f. Intuitively the function symbol is relevant, because the node $f\,M_1 \ldots M_k = M_0$ considers g to be equivalent to its finite map representation, which is justified by children such as $g\,N_1 \ldots N_l = N_0$.

To prove that the FDT has the fault location property, that is, the program equation associated with a faulty node is faulty, we first have to clarify what we mean by a reduction or a program equation being correct. We say that a *reduction* $f\,M_1 \ldots M_k = M_0$ *is correct* if and only if $f\,M_1 \ldots M_k \sqsupseteq M_0$ for some binary relation $\sqsupseteq$ that we call the intended semantics. The intended semantics may exist in the mind of the user or be derived from some form of specification:

Definition 1. *An* intended semantics *is a binary relation* $\sqsupseteq$ *on terms*[2] M*,* N *and* O *with the following consistency properties:*

1. *Reflexivity:* $M \sqsupseteq M$
2. *Transitivity:* $M \sqsupseteq N \wedge N \sqsupseteq O \implies M \sqsupseteq O$
3. *Closure:* $M \sqsupseteq N \implies M\,O \sqsupseteq N\,O \wedge O\,M \sqsupseteq O\,N$
4. *Least element:* $M \sqsupseteq \{\}$
5. *Application:* $\{N_1 \mapsto M_1, \ldots, N_k \mapsto M_k\}\,N_i \sqsupseteq M_i$

[2]These *computation terms* will be formally defined in Definition 5.

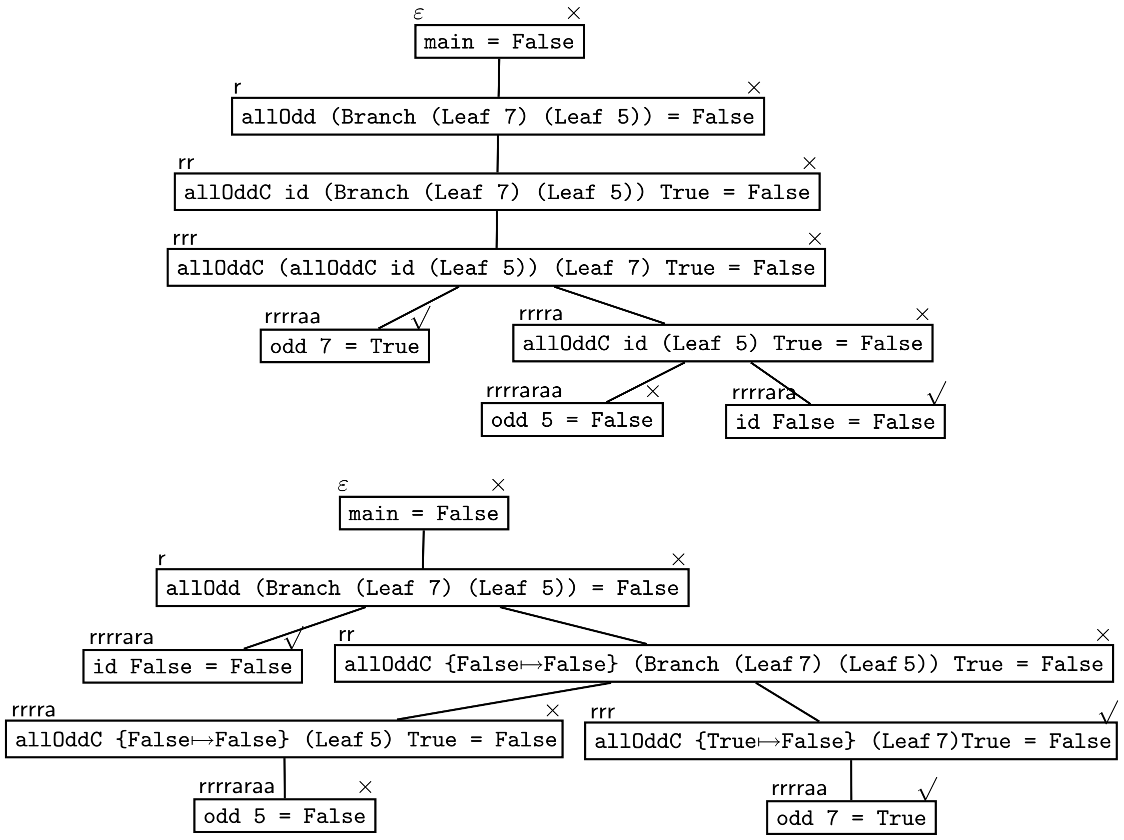
ε
main = False
r
allOdd (Branch (Leaf 7) (Leaf 5)) = False
rr
allOddC id (Branch (Leaf 7) (Leaf 5)) True = False
rrr
allOddC (allOddC id (Leaf 5)) (Leaf 7) True = False
rrrraa
odd 7 = True
rrrra
allOddC id (Leaf 5) True = False
rrrraraa
odd 5 = False
rrrrara
id False = False
ε
main = False
r
allOdd (Branch (Leaf 7) (Leaf 5)) = False
rrrrara
id False = False
rr
allOddC {False↦False} (Branch (Leaf 7) (Leaf 5)) True = False
rrrra
allOddC {False↦False} (Leaf 5) True = False
rrr
allOddC {True↦False} (Leaf 7) True = False
rrrraraa
odd 5 = False
rrrraa
odd 7 = True

Figure 1: EDT and FDT for the full computation of the tree traversal program

6. *Abstraction:* $ON_1 \sqsupseteq M_1 \wedge ... \wedge ON_k \sqsupseteq M_k \Rightarrow$
$O \sqsupseteq \{N_1 \mapsto M_1, ..., N_k \mapsto M_k\}$

The last two properties state that a finite map is a function as described by its entries. $\sqsupseteq$ is a pre-congruence with $\{\}$ as least element. So $M \sqsupseteq N$ can be read as "N approximates the value of M". The definition leaves much freedom. For example, for a set library both `insert 2 [1]` $\sqsupseteq$ `[2,1]` and `insert 2 [1]` $\sqsupseteq$ `[1,2]` may hold. A runtime error is represented as a special data constructor `Error` and hence `head []` $\sqsupseteq$ `Error` may hold. In this paper we just assume that an intended semantics exists.

A *program equation* $L = R$ *is faulty* if there exists a substitution σ such that the instance is not within the intended semantics: $L\sigma \not\sqsupseteq R\sigma$.

Algorithmic debugging considers only those substitutions of program equations that occurred within a computation; therefore it locates faults in both strict and non-strict functional programs.

To keep this paper self-contained we have to recapitulate definitions and propositions of the augmented redex trail [3] and algorithmic debugging with functions as partial applications [6].

3. THE AUGMENTED REDEX TRAIL

To relate algorithmic debugging to the computation of a program, we need a formal description of the computation. We use the augmented redex trail (ART) [3], a data structure that describes the computation of a functional program in detail, including all reductions, intermediate terms and sharing. This ART is a model of the trace used by the Haskell tracer Hat [14]. The ART is a graph whose structure was inspired by standard graph reduction implementations of functional languages. Basically an ART describes a state of a graph reduction machine, except that when a graph reduction step happens, the redex is not overwritten by the reduct, but the reduct is added to the ART and redex and reduct are connected via a reduction edge. Because nothing is overwritten, the whole history of a computation is preserved. The graph structure ensures space efficient sharing.

3.1 Term Graphs

Figure 2 shows the ART $\mathcal{P}$ for the computation of our example program after six reduction steps. A dotted arrow indicates a reduction. Each node's label, which may refer to further nodes, is depicted inside an oval. *Nodes* themselves are (possibly empty) strings of the letters f, a and r, that is, $n, m, o \in \{\mathsf{f}, \mathsf{a}, \mathsf{r}\}^*$. Thus nodes alone partially describe the graph structure: f means going to the function component of an application, a means going to the argument component of an application, and r means following a reduction edge to the reduct. An *atom* a is a *function symbol* f or a *data constructor* C.

Definition 2 (Term graph).

$$\begin{array}{llll} \textit{label} \ L & := & a & \textit{atom} \\ & | & n\,m & \textit{application} \\ & | & n & \textit{indirection} \end{array}$$

A term graph *is a partial function from nodes to labels,* $\mathcal{G}$:

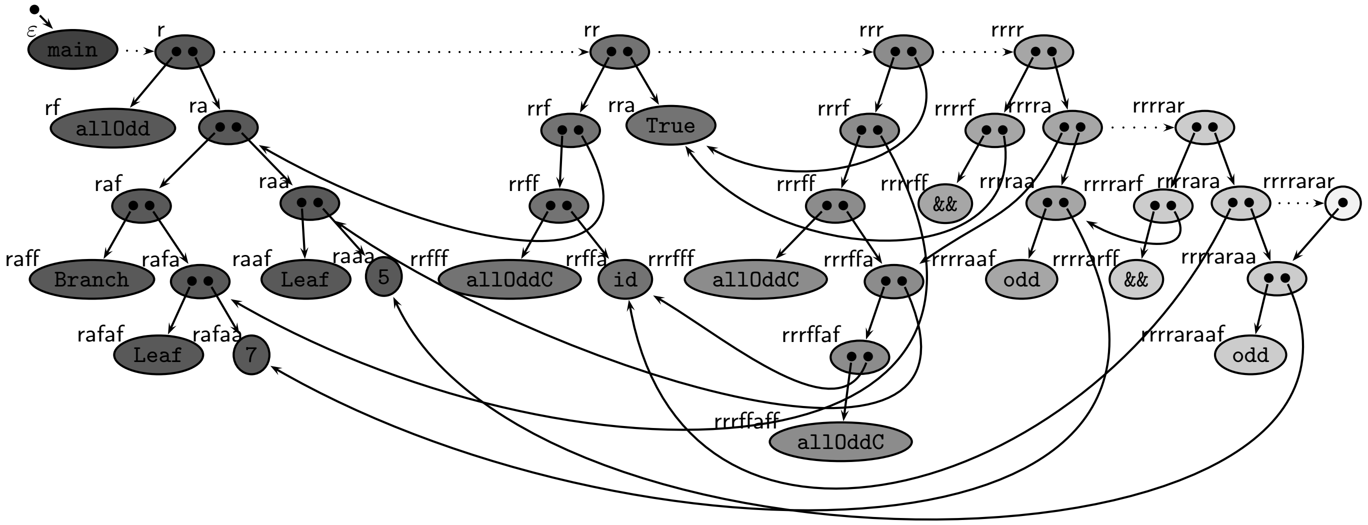

Figure 2: The ART $\mathcal{P}$ of a partial computation of the tree traversal program

$n \mapsto L$. *The domain* $\text{dom}(\mathcal{G})$ *of term graph* $\mathcal{G}$ *is the set of nodes for which the function is defined. We sometimes regard a term graph* $\mathcal{G}$ *as a set of tuples* $\{(n, \mathcal{G}(n)) \mid n \in \text{dom}(\mathcal{G})\}$.

Reduction edges are given implicitly: If and only if node $n\mathsf{r}$ exists in the graph, then there is a reduction edge from node n to node $n\mathsf{r}$. So to record a reduction we have to add at least one node. Therefore we need a special indirection node to record the reduction of a projection such as `id x = x`. In Figure 2 only node rrrrarar is an indirection node.

We will often need to follow a reduction or indirection edge. Hence we define a relationship on nodes:

Definition 3 (Predecessor-successor relation). *Let* $\mathcal{G}$ *be a term graph and* $n \in \text{dom}(\mathcal{G})$. *Then*

$$\begin{aligned} n \succ_{\mathcal{G}} m \iff & m = n\mathsf{r} \in \text{dom}(G) \vee \\ & (n\mathsf{r} \notin \text{dom}(G) \wedge \mathcal{G}(n) = m) \end{aligned}$$

Clearly a node n has at most one unique[3] successor m; but a node m may have several predecessors n.

We will often need to follow a chain of reduction and indirection edges to its end:

Definition 4 (Last node). *Let* $\mathcal{G}$ *be a term graph and* $n \in \text{dom}(\mathcal{G})$. *Then*

$$\lceil n \rceil_{\mathcal{G}} = m \iff n \succ^*_{\mathcal{G}} m \wedge \nexists o.\, m \succ_{\mathcal{G}} o$$

If there is no infinite sequence of successors, then the last node is defined and it is always unique. For example, in the term graph $\mathcal{P}$ of Figure 2, $\lceil \varepsilon \rceil_{\mathcal{P}} =$ rrrr and $\lceil$rrrrara$\rceil$ = rrrraraa.

3.2 Programs

We still have to define how we construct an ART for a particular program. Our programs are applicative term rewriting systems such as the program in the introduction.

[3]In an ART indirection nodes are never reduced, i.e., $\mathcal{G}(n) = m \implies n\mathsf{r} \notin \text{dom}(\mathcal{G})$; so for ARTs we could simplify the definition.

Definition 5 (Various terms).

$$\begin{array}{llcll} \textit{term} & M, N & := & a & \textit{atom} \\ & & \mid & n & \textit{node} \\ & & \mid & x & \textit{variable} \\ & & \mid & M\,N & \textit{application} \end{array}$$

Terms may contain both nodes and variables. A label term *is a term that does not contain variables. A* program term *is a term that does not contain nodes. A* computation term *is a term that contains neither variables nor nodes.*

Each atom a is associated with a natural number, its *arity*. A *pattern* P is a program term without function symbols. $f P_1 \ldots P_n = R$ is a *program equation*, provided that f is a function symbol of arity n and $P_1 \ldots P_n$ are patterns and R is a program term such that the variables of R are a subset of the variables of $f P_1 \ldots P_n$. A *program* is a set of program equations. We assume that the meaning of each predefined function such as `(&&)` and `mod` is given by a possibly infinite set of program equations.

3.3 Augmented Redex Trails

Augmented redex trails (ARTs) are defined inductively. The graph representation of an initial term M, $\text{graph}(\varepsilon, M)$, is an ART. If $\mathcal{G}$ is an ART and $\mathcal{G}$ reduces in one step with program P to $\mathcal{G}'$, that is, $\mathcal{G} \to_P \mathcal{G}'$, then $\mathcal{G}'$ is an ART:

Definition 6 (Augmented redex trail). *Let* P *be a program and* M *a computation term. A term graph* $\mathcal{G}$ *with* $\text{graph}(\varepsilon, M) \to^*_P \mathcal{G}$ *is an* augmented redex trail (ART) *for initial term* M *and program* P.

Figure 3 defines all functions used in our definition of ARTs. Detailed explanations are given in [3].

Figure 2 shows one of many ARTs for our example program. The reduction relation is non-deterministic and hence ARTs can describe computations of strict and non-strict languages and aborted computations. In later examples $\mathcal{F}$ denotes the ART of the full computation of our tree traversal program.

ARTs are finite and acyclic (see Proposition 7.2 in [3]). Hence the last node of a chain $\lceil n \rceil_{\mathcal{G}}$ is defined for any node n, and so is $\text{match}_{\mathcal{G}}(n, M)$ (cf. Section 7 in [3]).

The term graph for a given label term:

$$\begin{aligned}
\mathrm{graph}(n,a) &= \{(n,a)\}\\
\mathrm{graph}(n,m) &= \{(n,m)\}\\
\mathrm{graph}(n,M\,N) &= \begin{cases}
\{(n,M\,N)\} & \text{, if } M,\ N \text{ are nodes}\\
\{(n,M\,n\mathsf{a})\}\cup\mathrm{graph}(n\mathsf{a},N) & \text{, if only } M \text{ is a node}\\
\{(n,n\mathsf{f}\,N)\}\cup\mathrm{graph}(n\mathsf{f},M) & \text{, if only } N \text{ is a node}\\
\{(n,n\mathsf{f}\,n\mathsf{a})\}\cup\mathrm{graph}(n\mathsf{f},M)\cup\mathrm{graph}(n\mathsf{a},N) & \text{, otherwise}
\end{cases}
\end{aligned}$$

Matching is defined inductively over the structure of the matched label term:

$$\begin{aligned}
&\mathrm{match}_{\mathcal{G}}(o,a) = (\mathcal{G}(o)=a)\\
&\mathrm{match}_{\mathcal{G}}(o,M\,N) =\\
&\quad \exists m,n.(\mathcal{G}(o)=m\,n)\ \wedge\\
&\qquad \mathrm{match}_{\mathcal{G}}(\textit{if } M \textit{ is a node then } m \textit{ else } \lceil m\rceil_{\mathcal{G}}, M)\ \wedge\\
&\qquad \mathrm{match}_{\mathcal{G}}(\textit{if } N \textit{ is a node then } n \textit{ else } \lceil n\rceil_{\mathcal{G}}, N)\\
&\mathrm{match}_{\mathcal{G}}(o,m) = (o=m)
\end{aligned}$$

The reduction relation $\rightarrow_P$ *on term graphs for program* P *is defined as follows. If*

- $\mathcal{G}$ *is a term graph with* $n\in\mathrm{dom}(\mathcal{G})$ *and* $n\mathsf{r}\notin\mathrm{dom}(\mathcal{G})$*,*
- $L=R$ *is a program equation of the program* P*,*
- σ *is a substitution replacing variables by nodes,*
- $\mathrm{match}_{\mathcal{G}}(n,L\sigma)$*,*

then $\mathcal{G}\rightarrow_{P,n}\mathcal{G}\cup\mathrm{graph}_{\mathcal{G}}(n\mathsf{r},R\sigma)$ *with program equation* $L=R$ *and substitution* σ*.*

Figure 3: Definitions for the ART

4. COMPUTATION TREES

It is a central property of the ART that every reduction step performed in its construction can easily be reconstructed from it by traversing a small part of the graph.

4.1 Most Evaluated Forms

Because of reduction edges, a single node of a term graph usually represents many computation terms. An algorithmic debugger mostly shows values and hence we are interested in the most evaluated form represented by a given node. Because an ART may contain unevaluated expressions (incomplete, aborted computation or lazy evaluation) we speak of "most evaluated forms" and not of values. To construct a most evaluated form we always follows reduction and indirection edges. The most evaluated form (mef) of the node ε of the ART $\mathcal{P}$ in Figure 2 is `(&&) True ((&&) (odd7) (odd 5))`, in the ART $\mathcal{F}$ it is `False`. We have to decide whether we want to represent functional values as partial applications ($^{\mathsf{P}}$) or finite maps ($^{\mathsf{M}}$). For example, $\mathrm{mef}^{\mathsf{P}}_{\mathcal{F}}(\mathsf{rrffa}) = \mathtt{id}$ and $\mathrm{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrffa}) = \mathtt{\{False \mapsto False\}}$.

Definition 7 (Most evaluated form $\mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}$).

$$\mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(n) = \mathrm{mefT}^{\mathsf{P}}_{\mathcal{G}}(\mathcal{G}(\lceil n\rceil_{\mathcal{G}}))$$

$$\begin{aligned}
\mathrm{mefT}^{\mathsf{P}}_{\mathcal{G}}(a) &= a\\
\mathrm{mefT}^{\mathsf{P}}_{\mathcal{G}}(m\,n) &= \mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(m)\ \mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(n)
\end{aligned}$$

For example:

$$\begin{aligned}
\mathrm{mef}^{\mathsf{P}}_{\mathcal{P}}(\mathsf{rrffa}) &= \mathrm{mefT}^{\mathsf{P}}_{\mathcal{P}}(\mathcal{P}(\lceil\mathsf{rrffa}\rceil_{\mathcal{P}}))\\
&= \mathrm{mefT}^{\mathsf{P}}_{\mathcal{P}}(\mathcal{P}(\mathsf{rrffa})) = \mathrm{mefT}^{\mathsf{P}}_{\mathcal{P}}(\mathtt{id}) = \mathtt{id}
\end{aligned}$$

For finite maps we have to extend our definition of computation terms by the alternative form $\{N_1\mapsto M_1,\ldots,N_k\mapsto M_k\}$ for $k\geq 0$.

Definition 8 (Most evaluated form $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}$).

$$\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) = \begin{cases}
\mathrm{fMap}_{\mathcal{G}}(n), & \textit{if } M = f\,N_1...N_k \wedge 0\leq k < \mathrm{arity}(f)\\
\{\} & \text{, } \textit{if } M = f\,N_1...N_k \wedge k\geq\mathrm{arity}(f)\\
M & \text{, } \textit{otherwise}
\end{cases}$$
$$\textit{where } M = \mathrm{mea}_{\mathcal{G}}(n)$$

$$\mathrm{mea}_{\mathcal{G}}(n) = \mathrm{meaT}_{\mathcal{G}}(\mathcal{G}(\lceil n\rceil_{\mathcal{G}}))$$

$$\begin{aligned}
\mathrm{meaT}_{\mathcal{G}}(a) &= a\\
\mathrm{meaT}_{\mathcal{G}}(m\,n) &= \mathrm{mea}_{\mathcal{G}}(m)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)
\end{aligned}$$

$$\begin{aligned}
\mathrm{fMap}_{\mathcal{G}}(n) = \{\ &\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)\mapsto\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)\\
&\mid\ \mathcal{G}(m) = n'\,o\wedge n'\succ^{*}_{\mathcal{G}} n\wedge\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)\neq\{\}\ \}
\end{aligned}$$

Let us first consider two examples:

$$\begin{aligned}
&\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(\mathsf{rrffa})\\
=\ &(\mathrm{mea}_{\mathcal{P}}(\mathsf{rrffa}) = \mathtt{id})\\
&\mathrm{fMap}_{\mathcal{P}}(\mathsf{rrffa})\\
=\{\ &\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(o)\mapsto\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(m)\\
&\mid\ \mathcal{P}(m) = n'\,o\wedge n'\succ^{*}_{\mathcal{P}}\mathsf{rrffa}\wedge\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(m)\neq\{\}\ \}\\
=\ &\begin{cases}\{\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(\mathsf{rrrraraa})\mapsto\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(\mathsf{rrrrara})\}, & \text{if } \mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(\mathsf{rrrrara})\neq\{\}\\ \{\} & \text{, otherwise}\end{cases}\\
=\ &(\mathrm{mea}_{\mathcal{P}}(\mathsf{rrrrara}) = \mathtt{odd\ 5}\implies\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(\mathsf{rrrrara}) = \{\})\\
&\{\}
\end{aligned}$$

So the finite map is empty, because the result of the single application of the function, `odd 5`, was not reduced, but in the ART $\mathcal{F}$ of the full computation:

$$\begin{aligned}
&\mathrm{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrffa})\\
=\ &(\mathrm{mea}_{\mathcal{F}}(\mathsf{rrffa}) = \mathtt{id})\\
&\mathrm{fMap}_{\mathcal{F}}(\mathsf{rrffa})\\
=\ &\begin{cases}\{\mathrm{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrrraraa})\mapsto\mathrm{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrrrara})\}, & \text{if } \mathrm{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrrrara})\neq\{\}\\ \{\} & \text{, otherwise}\end{cases}\\
=\{\ &\mathtt{False}\mapsto\mathtt{False}\ \}
\end{aligned}$$

The most evaluated applicative form $\mathrm{mea}_{\mathcal{G}}(n)$ is defined identically to the most evaluated form with partial applications $\mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(n)$, except that arguments of applications are represented with finite maps. The most evaluated applicative form $\mathrm{mea}_{\mathcal{G}}(n)$ always contains an atom in the left-most position (it is an n-ary application of an atom or just an atom). The most evaluated form with finite maps $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ does not contain any function symbol, only data constructors, applications and finite maps.

The definition of the most evaluated form with finite maps $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ distinguishes three cases. A partial application of a function symbol is represented as a finite map. The application of a function symbol of arity n to n or more arguments identifies an unevaluated term and is simply represented as $\{\}$. Hence our soundness proof will also demonstrate that information about unevaluated terms is unnecessary for algorithmic debugging[4]. An empty map $\{\}$ represents both an unevaluated term and a functional value that was never applied to as many arguments as demanded by its arity. Distinguishing the two cases just would complicate the formalisation. Finally a most evaluated form can be a data constructor or an application of a data constructor. This representation is left unchanged. So not all functional values are represented as finite maps. Partial applications of data constructors are still simply represented as partial applications of data constructors.

A function map $\mathrm{fMap}_{\mathcal{G}}(n)$ is defined recursively, locating arguments and the result by locating all applications of the function at node n. To keep finite maps small, a finite map comprises applications of a specific node, not of a function symbol. For the code

```
main = map increase [1,2] ++ map increase [3,4]
```

the ART contains two nodes for `increase` and hence we will obtain the equations

```
map {1 ↦ 2, 2 ↦ 3} [1,2] = [2,3]
map {3 ↦ 4, 4 ↦ 5} [3,4] = [4,5]
```

and *not* the longer equations

```
map {1 ↦ 2, 2 ↦ 3, 3 ↦ 4, 4 ↦ 5} [1,2] = [2,3]
map {1 ↦ 2, 2 ↦ 3, 3 ↦ 4, 4 ↦ 5} [3,4] = [4,5]
```

In the definition the condition $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \neq \{\}$ avoids superfluous collection of partial applications. Otherwise we would have

$$\mathrm{mef}^{\mathsf{M}}_{\mathcal{P}}(\mathsf{rrffa}) = \{\ \{\} \mapsto \{\}\ \}$$

The definition yields finite maps of the form `{1 ↦ {2 ↦ 3, 3 ↦ 4}}`, but in practice we may prefer to display them as `{1 2 ↦ 3, 1 3 ↦ 4}`.

4.2 Well-Definedness of Finite Maps

For partial applications the most evaluated form is well-defined, because ARTs are acyclic. However, the definitions of $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}$ and $\mathrm{fMap}_{\mathcal{G}}$ are mutually recursive and may not be well-founded. The following program exposes the problem:

```
main = g id
id x = x
g h = (h h) 4
```

Figure 4 shows the ART $\mathcal{I}$ of the full computation. We have:

$$\begin{aligned}\mathrm{mef}^{\mathsf{M}}_{\mathcal{I}}(\mathsf{ra}) &= \mathrm{fMap}_{\mathcal{I}}(\mathsf{ra})\\ &= \{\underline{\mathrm{mef}^{\mathsf{M}}_{\mathcal{I}}(\mathsf{ra})} \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{I}}(\mathsf{rrf}), \mathrm{mef}^{\mathsf{M}}_{\mathcal{I}}(\mathsf{rra}) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{I}}(\mathsf{rr})\}\\ &= \{\underline{\mathrm{mef}^{\mathsf{M}}_{\mathcal{I}}(\mathsf{ra})} \mapsto \{4 \mapsto 4\}, 4 \mapsto 4\}\end{aligned}$$

[4]We could drop this case and thus include unevaluated terms as we do in the definition of $\mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}$. Or we could define $\mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(m\,n) = \{\}$, if $\mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(m)\ \mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(n) = f\,N_1 \ldots N_k \wedge k \geq \mathrm{arity}(f)$ and thus exclude unevaluated terms there as well.

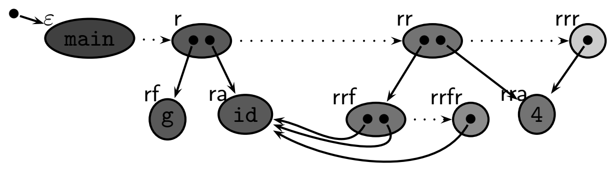

Figure 4: The ART $\mathcal{I}$ of the full computation of a program requiring rank-2 types

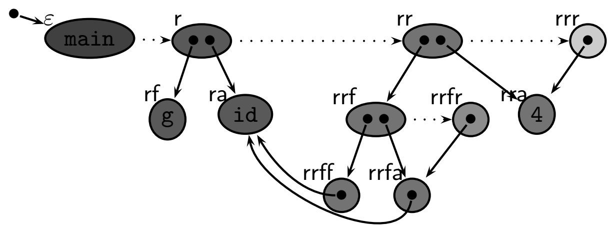

Figure 5: The ART $\mathcal{J}$ of the full computation of a program requiring rank-2 types with additional indirections

So $\mathrm{mef}_{\mathcal{I}}(\mathsf{ra})$ is clearly not well defined. However, the above program is not accepted by the Haskell 98 type system nor any other type system based on the Hindley-Milner type system [7]. Such type systems disallow applying a parameter to itself as occurs here in function `g`. The Hindley-Milner type system and its extensions for Haskell 98 or ML have the property that every polymorphic function is instantiated monomorphically at every occurrence where it is used in the program. Hence we can also type an ART by assigning a monomorphic type to each node. In the definition of $\mathrm{fMap}_{\mathcal{G}}$ for a node n the nodes o and m, to which $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}$ is applied recursively, have types that are components (argument and result type respectively) of the functional type of n. So the arguments of the recursive applications are strictly smaller and thus $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}$ is well-defined.

4.3 Extension: More Indirection Nodes

We could define finite maps for any program if we modified our definition of ARTs. Instead of using indirection nodes only for projections we would insert an indirection for all variables (or just all variables of functional type) by modifying our construction of the graph of a reduct as follows:

$$\begin{aligned}\mathrm{graph}(n,a) &= \{(n,a)\}\\ \mathrm{graph}(n,m) &= \{(n,m)\}\\ \mathrm{graph}(n,M\,N) &= \{(n, n\mathsf{f}\ n\mathsf{a})\} \cup \mathrm{graph}(n\mathsf{f},M) \cup \mathrm{graph}(n\mathsf{a},N)\end{aligned}$$

Figure 5 shows the ART $\mathcal{J}$ for the full computation of our example program with these additional indirection nodes. The additional indirections enable us to distinguish different instances of the same function:

$$
\begin{aligned}
\mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{ra}) &= \mathrm{fMap}_{\mathcal{J}}(\mathsf{ra})\\
&= \{\underline{\mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rrfa})} \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rrf})\\
&\quad ,\mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rra}) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rr})\ \}\\
&= \{\{\mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rra}) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rr})\} \mapsto\\
&\qquad \{\mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rra}) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rr})\}\\
&\quad ,\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rra}) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{J}}(\mathsf{rr})\ \}\\
&= \{\{4 \mapsto 4\} \mapsto \{4 \mapsto 4\},\ 4 \mapsto 4\ \}
\end{aligned}
$$

Such additional indirections would also make some finite maps smaller and thus simplify questions. For example, for the standard recursive definition of the function `map` the evaluation of `map odd [2,7]` yields the equations

```
map {2 ↦ False, 7 ↦ True} [2,7] = [False, True]
map {2 ↦ False, 7 ↦ True} [7] = [True]
map {2 ↦ False, 7 ↦ True} [] = []
```

because there is only one shared node for the function `odd`. With additional indirections we would have separate nodes and thus obtain:

```
map {2 ↦ False, 7 ↦ True} [2,7] = [False, True]
map {7 ↦ True} [7] = [True]
map {} [] = []
```

However, for Haskell 98 and ML additional indirection nodes are not necessary but would yield larger traces. We are still exploring further alternatives for tracing programs independent of any type system.

4.4 Equations

From a *redex node* n, that is, a node n with $n\mathsf{r} \in \mathrm{dom}(\mathcal{G})$, we can reconstruct an equation to be displayed as a question in an algorithmic debugging session. An equation is a pair of a redex, that is, an application of a function symbol, and a most evaluated form:

Definition 9 (Redexes and equations). *Let n be a redex node in $\mathcal{G}$.*

$$
\begin{aligned}
\mathrm{equation}^{\mathsf{P}}_{\mathcal{G}}(n) &= \mathrm{redex}^{\mathsf{P}}_{\mathcal{G}}(n) \texttt{=} \mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(n)\\
\mathrm{equation}^{\mathsf{M}}_{\mathcal{G}}(n) &= \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \texttt{=} \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{redex}^{\mathsf{P}}_{\mathcal{G}}(n) &= \mathrm{mefT}^{\mathsf{P}}_{\mathcal{G}}(\mathcal{G}(n))\\
\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) &= \mathrm{meaT}^{\mathsf{M}}_{\mathcal{G}}(\mathcal{G}(n))
\end{aligned}
$$

To construct a redex for a redex node n we do not follow the reduction edge from n, but otherwise the redex is defined like the most evaluated form.

4.5 EDT and FDT

Both EDT and FDT have a tree node for each redex node in the ART. In the preceding subsection we defined the equations of the nodes, which differ only in how functional values are represented. Now we have to define the structures of the two trees. Hence we have to relate instances of right-hand-sides to instances of left-hand-sides. The structure of ART nodes makes it easy to determine for a given node n the redex node parent(n) that caused its creation:

Definition 10 (ART parent node).

$$
\begin{aligned}
\mathrm{parent}(n\mathsf{r}) &= n\\
\mathrm{parent}(n\mathsf{f}) &= \mathrm{parent}(n)\\
\mathrm{parent}(n\mathsf{a}) &= \mathrm{parent}(n)\\
\mathrm{parent}(\varepsilon) &= \textit{undefined}
\end{aligned}
$$

For example, in Figure 2 parent$(\mathsf{rr}) = \mathsf{r}$ and parent$(\mathsf{rrfff}) = \mathsf{r}$.

We can identify the function node of a redex node:

Definition 11 (Function node). *Let n be a redex node of an ART $\mathcal{G}$.*

$$
\mathrm{fun}_{\mathcal{G}}(n) = \begin{cases} n & \textit{, if } \mathcal{G}(n) = a\\ \mathrm{fun}_{\mathcal{G}}(\lceil m \rceil_{\mathcal{G}}) & \textit{, if } \mathcal{G}(n) = m\, o \end{cases}
$$

Definition 12 (EDT, FDT). *The set of* tree nodes*,* treeNodes$_{\mathcal{G}}$*, is the set of redex nodes.*

The evaluation dependency tree (EDT) *for an ART $\mathcal{G}$ consists of the tree nodes* treeNodes$_{\mathcal{G}}$ *labelled with* equation$^{\mathsf{P}}_{\mathcal{G}}$ *and related via* parent *[6].*

The function dependency tree (FDT) *for an ART $\mathcal{G}$ consists of the tree nodes* treeNodes$_{\mathcal{G}}$ *labelled with* equation$^{\mathsf{M}}_{\mathcal{G}}$ *and related via* $\mathrm{parentFDT}_{\mathcal{G}} = \mathrm{parent} \cdot \mathrm{fun}_{\mathcal{G}}$*.*

The root of any non-empty EDT or FDT is ε.

The EDT is basically the proof tree of a natural semantics for a call-by-value computation that may skip some subcomputations. The structure of the EDT is determined by the parent of the application of a redex, the structure of the FDT is determined by the parent of the function symbol of a redex. In a first-order program function symbol and application always appear together in the right hand side of a definition. Hence then the EDT and the FDT are identical (there are also no functional arguments to be displayed as finite maps).

5. SOUNDNESS PROOF FOR THE FDT

Recall from Section 2 that the intended semantics is a binary relation $\sqsupseteq$ on terms meeting six consistency properties. Furthermore:

Definition 13 (Correctness and faultiness).

FDT tree node n correct	$\Leftrightarrow$	$\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$
all FDT children of tree node n correct	$\Leftrightarrow$	$\forall m \in \mathrm{treeNodes}_{\mathcal{G}}\,.\ \mathrm{parentFDT}_{\mathcal{G}}(m) = n \Rightarrow$ *tree node m correct*
FDT tree node n faulty	$\Leftrightarrow$	*FDT tree node n incorrect and all FDT children of tree node n correct*
program equation $L = R$ faulty	$\Leftrightarrow$	$\exists\sigma.\ L\sigma \not\sqsupseteq R\sigma$

We have to prove that, if an FDT tree node is faulty, then its associated program equation is faulty.

To do so for any FDT tree node n, we split the computation $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ into an initial reduction step from redex to reduct, $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$, and a remaining computation from reduct to most evaluated form,

$\text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) = \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. We will prove that, if n is faulty, then $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$. From the latter immediately follows, that the associated program equation is faulty:

PROPOSITION 1. *Let n be a redex node of ART $\mathcal{G}$. If $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$, then its associated program equation is faulty.*

PROOF. Analogous to Proposition 8.9 of [3]. $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$ is an instance of the associated program equation. □

We still need to define how we reconstruct from a redex node n of an ART the reduct of the reduction, that is, the instance of the right hand side of the program equation used for the reduction. All nodes that have n as parent form the right-hand-side of the program equation and therefore belong to the reduct. So the reduct comprises of all nodes that are reachable from the top node of the reduct via the node letters f and a. Shared subterms are represented in their most evaluated form. The reduct body $\text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n)$ is the part of a reduct below the top node n.

DEFINITION 14 (REDUCT OF A REDEX NODE).

$$\text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) = \text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{r})$$

$$\text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = \begin{cases} a & \text{, if } \mathcal{G}(n) = a \\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(m), & \text{if } G(n) = m \\ \text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f})\ \text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a}) & \\ \quad \text{, if } \mathcal{G}(n) = n\mathsf{f}\, n\mathsf{a} & \\ \text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f})\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(o) & \\ \quad \text{, if } \mathcal{G}(n) = n\mathsf{f}\, o \text{ and } o \neq n\mathsf{a} & \\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(m)\ \text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a}) & \\ \quad \text{, if } \mathcal{G}(n) = m\, n\mathsf{a} \text{ and } m \neq n\mathsf{f} & \\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(m)\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(o) & \\ \quad \text{, if } \mathcal{G}(n) = m\, o,\ m \neq n\mathsf{f} \text{ and } o \neq n\mathsf{a} & \end{cases}$$

For example:

$$\begin{aligned} & \text{reduct}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{r}) \\ = {} & \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rr}) \\ = {} & \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrf})\ \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rra}) \\ = {} & \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrff})\ \text{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{ra})\ \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rra}) \\ = {} & \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrfff})\ \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrffa})\ \text{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{ra})\ \text{reductB}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rra}) \\ = {} & \texttt{allOddC id (Branch (Leaf 7) (Leaf 5)) True} \end{aligned}$$

Now to the proof: Proposition 2 makes a statement about the intended semantics of a function symbol: if certain FDT nodes are correct, then the function symbol is correctly approximated by the finite map. Proposition 3 makes a statement about the intended semantics of application: applying a most evaluated form (usually a finite map) to another most evaluated form is correctly approximated by the most evaluated form of the application. Based on these two propositions we prove by a structural induction on the right-hand-side of a program equation that algorithmic debugging with finite maps is sound (Corollary 1). So for any future variation of algorithmic debugging with finite maps we will aim to ensure that Propositions 2 and 3 still hold and then soundness of that variation is guaranteed.

The complete proofs are in the appendix.

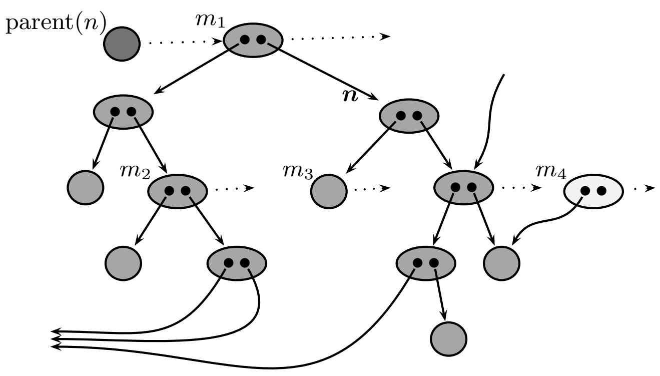

Figure 6: Extract of an ART concentrating on one reduct to outline the proof idea of Proposition 4 on the Correctness of the Reduct

PROPOSITION 2. *If $\mathcal{G}(n) = f$ for some function symbol f with $\text{arity}(f) > 0$ and all FDT children of $\text{parent}(n)$ are correct, then $f \sqsupseteq \text{fMap}_{\mathcal{G}}(n)$.*

PROOF. Because $\text{arity}(f) > 0$, we have $n \notin \text{treeNodes}_{\mathcal{G}}$ and $\text{mef}^{\mathsf{M}}_{\mathcal{G}}(n) = \text{fMap}_{\mathcal{G}}(n)$.

We prove the more general property that if $n \in \text{dom}(\mathcal{G})$ with $n \notin \text{treeNodes}_{\mathcal{G}}$ and $\text{mea}_{\mathcal{G}}(n) = f\, N_1 \ldots N_k$ for some function symbol f and computation terms $N_1 \ldots N_k$ with $\text{arity}(f) > k \geq 0$, and all FDT children of $\text{parentFDT}_{\mathcal{G}}(n)$ are correct, then $\text{mea}_{\mathcal{G}}(n) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

Let $j = \text{arity}(f) - k$. Proof by induction on j. □

PROPOSITION 3. *In the FDT application is in the intended semantics, that is,*

$$\mathcal{G}(n) = p\, o \implies \text{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$$

PROOF. Proof by case analysis on $\text{mea}_{\mathcal{G}}(p)$. □

PROPOSITION 4 (CORRECTNESS OF THE REDUCT). *If n is a tree node and all its FDT children are correct, then $\text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.*

PROOF. $\text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) = \text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n r)$. $\text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n r) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ follows from the more general property: If $n \in \text{dom}(\mathcal{G})$ and all FDT children of $\text{parent}(n)$ are correct, then $\text{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

Induction on $\text{height}_{\mathcal{G}}(n) = \max\{|o| \mid o \in \{\mathsf{f}, \mathsf{a}\}^* \wedge no \in \text{dom}(\mathcal{G})\}$. □

The inductive proof of Proposition 4 is outlined in Figure 6. It shows in light grey all the nodes representing a right-hand-side of a program equation, plus their parent node $\text{parent}(n)$ and one node m_4 that shares a node of this right-hand-side. The base step of the induction covers the leaf nodes, that is atom nodes and application nodes pointing to shared nodes (indicated by edges going to the bottom left). The induction step then covers inner application nodes such as n. The nodes m_1, m_2, m_3 and m_4 are the FDT children of the node $\text{parent}(n)$. Roughly, Proposition 2 is needed for the base step and Proposition 3 for the induction step.

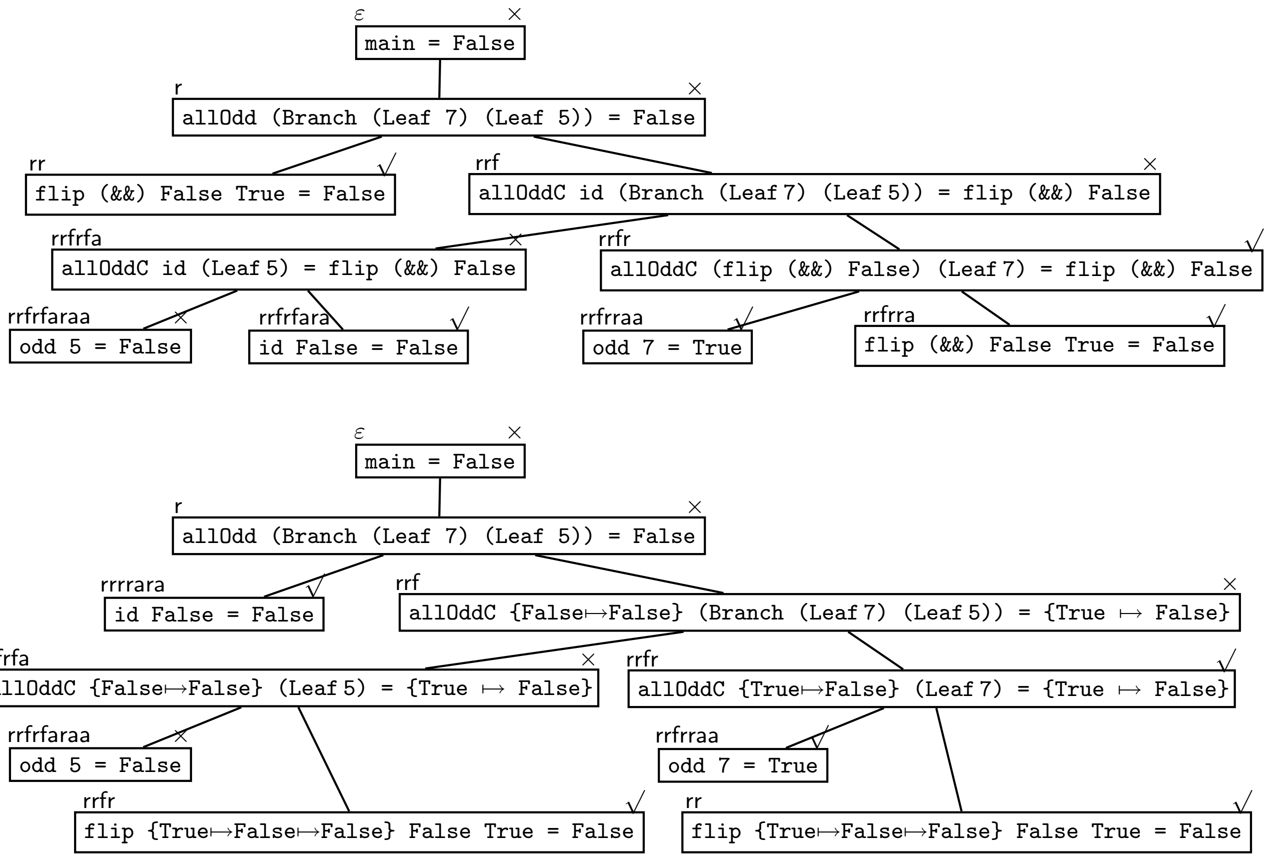
ε main = False
r allOdd (Branch (Leaf 7) (Leaf 5)) = False
rr flip (&&) False True = False
rrf allOddC id (Branch (Leaf 7) (Leaf 5)) = flip (&&) False
rrfrfa allOddC id (Leaf 5) = flip (&&) False
rrfr allOddC (flip (&&) False) (Leaf 7) = flip (&&) False
rrfrfaraa odd 5 = False
rrfrfara id False = False
rrfrraa odd 7 = True
rrfrra flip (&&) False True = False
ε main = False
r allOdd (Branch (Leaf 7) (Leaf 5)) = False
rrrrara id False = False
rrf allOddC {False↦False} (Branch (Leaf 7) (Leaf 5)) = {True ↦ False}
rrfrfa allOddC {False↦False} (Leaf 5) = {True ↦ False}
rrfr allOddC {True↦False} (Leaf 7) = {True ↦ False}
rrfrfaraa odd 5 = False
rrfrraa odd 7 = True
rrfr flip {True↦False↦False} False True = False
rr flip {True↦False↦False} False True = False

Figure 7: EDT and FDT for the full computation of the example variant

Corollary 1 (FDT is a computation tree). *If tree node n is faulty, then* $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$.

Proof. Let n be a tree node. According to Proposition 4 we have $\text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. Assume $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$. By transitivity $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ in contradiction to our hypothesis that the tree node is incorrect. Hence $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \text{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$. $\square$

6. TRUSTING

In Definition 12 of the EDT and the FDT every redex node of the ART becomes a tree node. However, in our example EDT and FDT of Figure 1 we excluded reductions of predefined functions such as `(&&)` and `mod`. We can do so, because we *trust* the definitions of these functions to be correct. If we trust definitions to be correct, then tree nodes associated with these definitions must be correct. Hence the algorithmic debugger does not have to ask about these tree nodes. Even better, we can remove such trusted tree nodes from the tree. A trusted tree node may still have children (common for higher-order functions in the EDT; also we may trust a testing scaffold calling our suspected code); so if we remove a trusted tree node, its parent node gains the children of the removed node.

As noted already by Shapiro [13], algorithmic debugging is still sound when trusting is applied. Trusting applies to any computation tree, independent of its particular definition and conceptually is applied after constructing the complete computation tree. To save time and space, it is desirable in practise to avoid tracing and constructing computation tree nodes for trusted code.

7. AN EXAMPLE VARIANT

What happens if we make a small change to our example program, if we remove the Boolean argument `b` from the equations of `allOddC`?

```
allOddC c (Leaf n) = flip (&&) (c (odd n))
allOddC c (Branch l r) = allOddC (allOddC c r) l
```

where the function `flip` swaps arguments:

```
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Figure 7 shows the EDT and FDT for this variant. The new EDT includes a node labelled

```
allOddC (flip (&&) False) (Leaf 7) = flip (&&) False
```

whereas the corresponding node of the FDT is labelled

```
allOddC {True ↦ False} (Leaf 7) = {True ↦ False}
```

The former question requires us to interpret the function symbols `allOddC`, `flip` and `(&&)`. The latter expresses that the finite map `{True ↦ False}` (indeed correctly) approximates the functional value of `allOddC {True ↦ False}`

`(Leaf 7)`. Thus this example demonstrates the occurrence of a functional value as result value.

The structure of the FDT is the same as the one for our original program, whereas the structure of the EDT differs substantially compared to the EDT of the original program. In fact, the structure of the new EDT is similar to the structure of the FDT. Our simple and commonly used program refactoring does not change the occurrences of function symbols in an equation. Thus it leaves the structure of an FDT unchanged, whereas it may substantially change the structure of an EDT.

8. RELATED WORK

Naish [8] gives an abstract description of algorithmic debugging, independent of any particular programming language. He proves that algorithmic debugging is *complete* in the sense that if the program computation produces a wrong result, then algorithmic debugging will locate a fault. No such general proof exists for the *soundness* of algorithmic debugging, that is, the property that the indicated fault location is indeed faulty, because soundness depends on the exact definition of the computation tree.

For lazy functional programming languages Nilsson and Sparud [10] introduced the evaluation dependency tree (EDT) as computation tree. The EDT has the property that the tree structure reflects the static function call structure of the program and all arguments and results are in their most evaluated form. For many years the EDT was considered to be the only useful computation tree for functional programs.

Caballero *et al.* [2] give a formal definition of the EDT for a lazy functional logic language and outline a soundness proof of algorithmic debugging. They introduced the formalisation of the intended semantics that we extend. Their approach relies on the EDT being defined through a high-level non-deterministic big-step semantics. This big-step semantics is unsuitable for defining the FDT, because it would be hard to relate the occurrence of a function symbol in an argument with an application of the function symbol. In contrast, the augmented redex trail (ART) [3] records such information directly in its graph structure and as an explicit data structure it is also easier to manipulate. A formal definition of the EDT based on the ART together with a soundness proof are already given in [6]. The soundness proof for the FDT is similarly structured but longer, because it has to handle the finite map representation and its properties (cf. Propositions 2 and 3). Without additional work it also covers the representation of unevaluated subexpressions by a special symbol (here $\{\}$).

Pope [11, 12] introduced the idea of representing functional values as finite maps into algorithmic debugging of higher-order functional languages and implemented it in the Haskell debugger Buddha. He demonstrates its usefulness and describes the implementation but gives no formal model. He does not use the name FDT but considers it as a variant of the EDT. Pope uses the term *intensional style* for the partial application representation and *extensional style* for the finite map representation of functional values.

The ART is a model of the trace used by the Haskell tracer Hat, which includes an EDT-based algorithmic debugger. Hat records the trace in a file. Hat has been used for debugging the nhc98 compiler [4] and a chess end-game solver that executes 6 million reductions, producing traces of several hundred megabytes, about 40-50 bytes per reduction [14]. The definition of equation$^{\mathsf{P}}_{G}$ demonstrates that the algorithmic debugger needs to read only a tiny fragment of a huge trace file to construct an equation. Even longer computations can be debugged if the majority of the program is trusted and thus its reductions are not recorded in the trace.

Both Nilsson [9] and Pope [12] developed piecemeal tracing schemes to reduce the trace size of algorithmic debuggers for lazy functional languages. Initially only a top piece of the computation tree is recorded; when the fault localisation process reaches the fringe of this piece, the computation is re-executed and another piece is added to the recorded tree. These schemes require complex implementations and still produce relatively large traces.

Braßel *et al.* [1] developed an alternative approach to EDT-based algorithmic debugging of lazy functional languages that reduces the trace size by at least four orders of magnitude. The trace is only a sequence of integers that tells a call-by-value evaluator after how many steps to skip a redex so that it produces the same final result as a lazy computation. To construct an equation, the lazy call-by-value evaluator re-executes a part of the computation. Overall the time for trace generation and equation construction is similar to that of Hat. Our paper should provide a good foundation for determining how the lazy call-by-value evaluator of Braßel *et al.* would need to be modified to support FDT-based algorithmic debugging.

9. SUMMARY AND FUTURE WORK

We formally defined the function dependency tree (FDT), a computation tree for algorithmic debugging of higher-order functional programs that represents functional values as finite maps. We defined the FDT in terms of the augmented redex trail (ART) a trace that describes the graph reduction computation of a functional program in detail, but independent of any evaluation order. Thus we proved the soundness of algorithmic debugging with the FDT, that is, that every located fault is indeed a fault. All definitions and theorems apply to both strict and non-strict functional languages.

Every occurrence of a function symbol in the right hand side of an equation creates at every reduction of this equation a new function node in the ART. The finite map of such a function node contains only the arguments (and results) to which this node was applied, not all arguments (and results) of the function symbol. This smaller set is sufficient for soundness. A function that is passed as a parameter does not create a new node in the ART and hence self-application of a function passed as parameter creates an ART for which the "finite" map of the function is ill-defined; because of cyclic dependencies it would be infinite. This is not a problem for Haskell 98 or Standard ML, because the Hindley-Milner type system excludes such self-application. Alternatively we could modify the definition of the ART to include more indirection nodes: thus we could obtain finite maps for any program and even smaller, more specific finite maps for many programs. We are still looking for alternative solutions that do not require such additional indirection nodes but still yield relatively simple definitions of finite maps.

The ART is a model of the trace used by the Haskell tracer Hat. Thus this paper shows how little effort is needed to extend Hat such that it supports algorithmic debugging with both partial applications and finite maps. A prototype exists, but in practise construction of the finite maps is time consuming and hence we are working on more efficient algorithms.

There is a clear symmetry between the definitions of the standard evaluation dependency tree (EDT) and the FDT. The close relationship suggests that sound mixtures of the two computation trees exist and further variations, for example with equations that do not respect the arity of the original function definitions, are worth exploring.

The FDT also enables algorithmic debugging of top-level definitions independent of local definitions made in where- or let-clauses. The idea is that algorithmic debugging could ask only questions about functions defined at the top-level. When a faulty function is identified, the fault is either in the definition of that function itself or its local function definitions. This kind of low granularity algorithmic debugging requires less questions and it is still possible to locate the fault more precisely by later asking questions about locally defined functions. Such low granularity algorithmic debugging is unsound for the EDT, because the call site for a function passed out of a local scope can be anywhere in the program. In contrast, locally defined function symbols can only occur within the surrounding definition. To prove the soundness of this algorithmic debugging scheme, we will have to extend our ART model to programs with local function definitions.

Acknowledgement

This work has been partially supported by the United Kingdom under EPSRC grant EP/C516605/1.

10. REFERENCES

[1] B. Braßel, M. Hanus, S. Fischer, F. Huch, and G. Vidal. Lazy call-by-value evaluation. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 265–276, 2007.

[2] R. Caballero, F. J. López-Fraguas, and M. Rodríguez-Artalejo. Theoretical foundations for the declarative debugging of lazy functional logic programs. In H. Kuchen and K. Ueda, editors, *Functional and Logic Programming, FLOPS 2001*, LNCS 2024, pages 170–184. Springer, 2001.

[3] O. Chitil and Y. Luo. Structure and properties of traces for functional programs. In I. Mackie, editor, *Proceedings of the 3rd International Workshop on Term Graph Rewriting, Termgraph 2006*, ENTCS 176(1), pages 39–63, 2007.

[4] O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL 2002)*, LNCS 2670, pages 165–181, 2003.

[5] T. Davie and O. Chitil. Display of functional values for debugging. In *Draft Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL 2006*, pages 326–337. Eötvös Loránd University, September 2006. Technical Report No 2006-SO1.

[6] Y. Luo and O. Chitil. Proving the correctness of algorithmic debugging for functional programs. In *Trends in Functional Programming*, volume 7, pages 19–34. Intellect Books, 2007.

[7] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, Dec. 1978.

[8] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[9] H. Nilsson. Tracing piece by piece: affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 36–47. ACM Press, 1999.

[10] H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering: An International Journal*, 4(2):121–150, Apr. 1997.

[11] B. Pope. Declarative debugging with Buddha. In *Advanced Functional Programming, 5th International School, AFP 2004*, LNCS 3622, pages 273–308. Springer Verlag, September 2005.

[12] B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

[13] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[14] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, UU-CS-2001-23. Universiteit Utrecht, 2001.

Appendix: Complete Proofs

The following lemma will be used several times.

LEMMA 1. *An application that is not the application of a function symbol f to* $\text{arity}(f)$ *arguments cannot be a tree node, that is,*

$$\begin{aligned} & \mathcal{G}(n) = p\,o \wedge \\ & \text{mea}_{\mathcal{G}}(p) = a\,N_1 \ldots N_k \wedge \\ & (a = C \vee \text{arity}(a) \neq k+1) \\ \Longrightarrow\ & n \notin \text{treeNodes}_{\mathcal{G}} \end{aligned}$$

PROOF. Assume $n \in \text{treeNodes}_{\mathcal{G}}$. Then $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n) = \text{mea}\overline{\text{T}}_{\mathcal{G}}(p\,o) = \text{mea}_{\mathcal{G}}(p)\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = a\,N_1 \ldots N_k\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. Following Lemma 8.9 of [3] $\text{redex}^{\mathsf{M}}_{\mathcal{G}}(n)$ is an instance of the left-hand-side of a rule. The left-hand-side of a rule is an application of a function symbol f to exactly $\text{arity}(f)$ patterns. Because of this contradiction our original assumption that $n \in \text{treeNodes}_{\mathcal{G}}$ must be wrong. □

PROOF OF PROPOSITION 2. We prove the more general property that, if $n \in \text{dom}(\mathcal{G})$ with $n \notin \text{treeNodes}_{\mathcal{G}}$ and $\text{mea}_{\mathcal{G}}(n) = f\,N_1 \ldots N_k$ for some function symbol f and computation terms $N_1 \ldots N_k$ with $\text{arity}(f) > k \geq 0$, and all FDT children of $\text{parentFDT}_{\mathcal{G}}(n)$ are correct, then $\text{mea}_{\mathcal{G}}(n) \sqsupseteq \text{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

Let $j = \text{arity}(f) - k$. Induction on j.

case $j = 1$:

Let $m \in \text{dom}(\mathcal{G})$ with $G(m) = n'\,o$ and $n' \succ^*_{\mathcal{G}} n$ and $\text{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \neq \{\}$.

Assume $m \notin \text{treeNodes}_{\mathcal{G}}$. Then we have $\text{mea}_{\mathcal{G}}(m) = f\,N_1 \ldots N_k\ \text{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. Because $\text{arity}(f) = k+1$ we get $\text{mef}^{\mathsf{M}}_{\mathcal{G}}(m) = \{\}$ in contradiction to our hypothesis that

$\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \neq \{\}$. Hence our assumption is wrong and $m \in \mathrm{treeNodes}_{\mathcal{G}}$.

We have $\mathrm{redex}_{\mathcal{G}}(m) = \mathrm{mea\Gamma}_{\mathcal{G}}(G(m)) = \mathrm{mea\Gamma}_{\mathcal{G}}(n'\,o) = \mathrm{mea}_{\mathcal{G}}(n')\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = f\,N_1 \ldots N_k\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. Furthermore $\mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parentFDT}_{\mathcal{G}}(n)$. Therefore m is a child of $\mathrm{parentFDT}_{\mathcal{G}}(n)$ and so $\mathrm{redex}_{\mathcal{G}}(m) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)$ and hence we know that $\mathrm{mea}_{\mathcal{G}}(n)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)$. Because we have this for any m, the abstraction property of the intended semantics gives us $\mathrm{mea}_{\mathcal{G}}(n) \sqsupseteq \{\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \mid \mathcal{G}(m) = n'\,o \wedge n' \succ^*_{\mathcal{G}} n \wedge \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \neq \{\}\}$. Therefore $\mathrm{mea}_{\mathcal{G}}(n) \sqsupseteq \mathrm{fMap}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $j > 1$:

Let $p \in \mathrm{dom}(\mathcal{G})$ with $\mathcal{G}(p) = n'\,o$ and $n' \succ^*_{\mathcal{G}} n$.

From $\mathrm{arity}(f) - k = j > 1$ follows $\mathrm{arity}(f) > k + 1$. Together with $\mathrm{mea}_{\mathcal{G}}(n') = \mathrm{mea}_{\mathcal{G}}(n) = f\,N_1 \ldots N_k$ Lemma 1 gives us $p \notin \mathrm{treeNodes}_{\mathcal{G}}$, so $p\mathsf{r} \notin \mathrm{dom}(\mathcal{G})$.

So $\mathrm{mea}(p) = \mathrm{mea}_{\mathcal{G}}(n')\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = \mathrm{mea}_{\mathcal{G}}(n)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = f\,N_1 \ldots N_k\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. Because we have $\mathrm{arity}(f) - (k+1) = j - 1$ and so $\mathrm{arity}(f) > k + 1$, we can apply the induction hypothesis. For all $m \in \mathrm{treeNodes}_{\mathcal{G}}$ with $\mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parentFDT}_{\mathcal{G}}(n)$ we know that $\mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parentFDT}_{\mathcal{G}}(p)$. So for all $m \in \mathrm{treeNodes}_{\mathcal{G}}$ we have $\mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parentFDT}_{\mathcal{G}}(p)$ implies $\mathrm{redex}_{\mathcal{G}}(m) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)$. So $\mathrm{mea}_{\mathcal{G}}(p) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)$. Together with $\mathrm{mea}_{\mathcal{G}}(p) = \mathrm{mea}_{\mathcal{G}}(n)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$ we obtain that $\mathrm{mea}_{\mathcal{G}}(n)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)$.

Because we have this last relationship for any p we obtain with the abstraction property of the intended semantics $\mathrm{mea}_{\mathcal{G}}(n) \sqsupseteq \{\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p) \mid \mathcal{G}(p) = n'\,o \wedge n' \succ^*_{\mathcal{G}} n \wedge \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p) \neq \{\}\}$. The condition $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p) \neq \{\}$ is not necessary, but the relationship still holds with additional conditions. We conclude that $\mathrm{mea}_{\mathcal{G}}(n) \sqsupseteq \mathrm{fMap}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. □

Proof of Proposition 3. Case analysis on $\mathrm{mea}_{\mathcal{G}}(p)$:

case $\mathrm{mea}_{\mathcal{G}}(p) = f\,N_1 \ldots N_k$ and $\mathrm{arity}(f) > k \geq 0$:

So $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p) = \mathrm{fMap}_{\mathcal{G}}(p) = \{\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o') \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \mid G(m) = p'\,o' \wedge p' \succ^*_{\mathcal{G}} p \wedge \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \neq \{\}\}$.

case $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) = \{\}$:

Trivially $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \sqsupseteq \{\} = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) \neq \{\}$:

Then $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \mapsto \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) \in \mathrm{fMap}_{\mathcal{G}}(p)$. With the application property we get $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = \mathrm{fMap}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $\mathrm{mea}_{\mathcal{G}}(p) = f\,N_1 \ldots N_k$ and $\mathrm{arity}(f) \leq k$:

According to Lemma 1 $n \notin \mathrm{treeNodes}_{\mathcal{G}}$. So $\mathrm{mea}_{\mathcal{G}}(n) = \mathrm{mea}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) = f\,N_1 \ldots N_k\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ and $\mathrm{arity}(f) \leq k$. Therefore $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) = \{\}$. Clearly $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) \sqsupseteq \{\} = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $\mathrm{mea}_{\mathcal{G}}(p) \neq f\,N_1 \ldots N_k$:

Then $\mathrm{mea}_{\mathcal{G}}(p) = C\,N_1 \ldots N_k$ for some constructor C and terms $N_1 \ldots N_k$. Lemma 1 gives us $n \notin \mathrm{treeNodes}_{\mathcal{G}}$. So $\mathrm{mea}_{\mathcal{G}}(n) = \mathrm{mea}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) = C\,N_1 \ldots N_k\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. Hence $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{mea}_{\mathcal{G}}(n)$. Therefore we know that $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = \mathrm{mea}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = \mathrm{mea}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. □

Proof of Proposition 4 (Correctness of Reduct). $\mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{r})$. $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{r}) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ follows from the more general property: If $n \in \mathrm{dom}(\mathcal{G})$ and all FDT children of $\mathrm{parent}(n)$ are correct, then $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

Induction on $\mathrm{height}_{\mathcal{G}}(n) = \max\{|o| \mid o \in \{\mathsf{f}, \mathsf{a}\}^* \wedge no \in \mathrm{dom}(\mathcal{G})\}$.

case $\mathrm{height}_{\mathcal{G}}(n) = 0$:

case $\mathcal{G}(n) = a$:

case $n \in \mathrm{treeNodes}_{\mathcal{G}}$:

Because $\mathrm{parentFDT}_{\mathcal{G}}(n) = \mathrm{parent}(n)$ we have $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. So $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = a = \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $n \notin \mathrm{treeNodes}_{\mathcal{G}}$:

case $\mathcal{G}(n) = f$:

If $\mathrm{arity}(f) = 0$ then $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \{\} = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ else $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = f = \mathrm{mea}_{\mathcal{G}}(n)$ and with Proposition 2 we get $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{fMap}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $\mathcal{G}(n) = C$:

$\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = C = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $\mathcal{G}(n) = m$:

By definition $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $\mathcal{G}(n) = p\,o$:

$\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. With Proposition 3 we obtain $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $\mathrm{height}_{\mathcal{G}}(n) > 0$:

Then $G(n) = p\,o$ and $p = n\mathsf{f}$ or $o = n\mathsf{a}$.

case $p = n\mathsf{f}$ and $o \neq n\mathsf{a}$:

$\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f})\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. According to the induction hypothesis $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f}) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f})$. So with context closure of the intended semantics $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$ follows and with Proposition 3 we obtain $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $p \neq n\mathsf{f}$ and $o = n\mathsf{a}$:

$\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a})$. According to the induction hypothesis $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a}) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a})$. So $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$ and thus $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

case $p = n\mathsf{f}$ and $o = n\mathsf{a}$:

$\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f})\ \mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a})$. From the induction hypothesis we obtain $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f}) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{f})$ and $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a}) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{a})$. Therefore $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$ and consequently $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. □