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

Olaf Chitil and Thomas Davie

University of Kent, UK

Abstract. Bernie Pope introduced the idea of representing functional values as finite maps instead of partial applications into algorithmic debugging of higher-order functional languages. He implemented it in his Haskell debugger Buddha. Here we give an implementation-independent formal definition of algorithmic debugging for both representation in a single framework, even though the computation trees for the two representations have rather different structures. On this basis we prove the soundness of algorithmic debugging with finite maps. Our model 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 model and proof suggests variations of algorithmic debugging with finite maps and thus are tools for further improvement of this form of debugging.

1 Introduction

Algorithmic debugging is a semi-automated method of locating faults in declarative programs. Consider the following 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
-- specification: 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
```

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

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

```
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 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. Hence Pope [8, 9] and later independently Davie and Chitil [3] 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 $\bot$. Every question contains

only one function symbol. With a different function representation the meaning of question 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 [8, 9, 3] demonstrate. Furthermore, no algorithmic debugger supported λ-abstractions meaningfully before the introduction of finite maps. We discuss another advantage in Section 7.

Pope gives a detailed technical description of his implementation of finite maps in the algorithmic debugger Buddha [9]. 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` $\mapsto$ `False, True` $\mapsto$ `True, 42` $\mapsto$ `42, 'c'` $\mapsto$ `'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.

To keep this paper self-contained we have to recapitulate definitions and propositions of the augmented redex trail [2] and algorithmic debugging with functions as partial applications [4].

2 The Augmented Redex Trail (ART)

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) [2], 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 [10]. 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

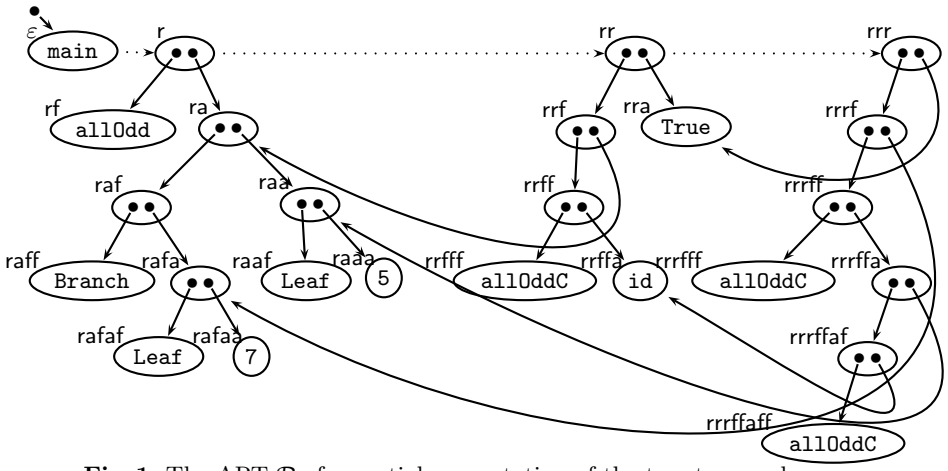

Fig. 1. The ART $\mathcal{P}$ of a partial computation of the tree traversal program

whole history of a computation is preserved. The graph structure ensures space efficient sharing.

2.1 Term Graphs

Figure 1 shows the ART $\mathcal{P}$ for the computation of our example program after three 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.

Definition 1 (Term graph).

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

A term graph *is a partial function from nodes to labels, $\mathcal{G} : n \mapsto L$. The domain* $\mathrm{dom}(\mathcal{G})$ *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 $\{(n, \mathcal{G}(n)) \mid n \in \mathrm{dom}(\mathcal{G})\}$.*

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 $n\mathsf{r}$. The reduction of a projection such as `id x = x` is recorded by adding an indirection node.

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

Definition 2 (Last node). *Let $\mathcal{G}$ be a term graph and $n \in \mathrm{dom}(\mathcal{G})$. Then*

$$n \succ_{\mathcal{G}} m \iff m = n\mathsf{r} \vee \mathcal{G}(n) = m$$
$$\lceil n \rceil_{\mathcal{G}} = m \iff n \succ_{\mathcal{G}}^{*} m \wedge \nexists o.\, m \succ_{\mathcal{G}} o$$

For example, in the term graph $\mathcal{P}$ of Figure 1, $\lceil\varepsilon\rceil_{\mathcal{P}} = \mathsf{rrr}$.

2.2 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. An *atom* a is a *function symbol* f or a *data constructor* C. Each atom a is associated with a natural number, its *arity*.

Definition 3 (Term, label term, program term, computation term).

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

Terms 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.*

A *pattern* P is a program term without function symbols. $fP_1 \ldots P_n = R$ is a *rewrite rule*, 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 $fP_1 \ldots P_n$. A *program* is a set of rewrite rules. We assume that the meaning of each predefined function such as `(&&)` and `mod` is given by a possibly infinite set of rewrite rules.

2.3 Augmented Redex Trails

Augmented redex trails (ARTs) are defined inductively. The graph representation of an initial term M, $\mathrm{graph}(\varepsilon, M)$, 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 4 (Augmented redex trail). *Let P be a program and M a computation term. A term graph $\mathcal{G}$ with* $\mathrm{graph}(\varepsilon, M) \to_P^* \mathcal{G}$ *is an* augmented redex trail (ART) for initial term M and program P.

Figure 2 defines all functions used in our definition of ARTs. Detailed explanations are given in [2].

Figure 1 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.

3 Constructing Terms and Equations

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.

The 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:

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

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 rewrite rule 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 rewrite rule $L = R$ and substitution σ.

Fig. 2. Definitions for the ART

3.1 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. The most evaluated form of the node ε in Figure 1 is `allOddC ... (Leaf 7) True`, 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 $\mathrm{mef}^{\mathsf{M}}_{\mathcal{F}}(\mathsf{rrffa}) = \{\mathtt{False} \mapsto \mathtt{False}\}$.

Definition 5 (Most evaluated form with partial applications).

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

The most evaluated form always follows reduction and indirection edges. It is well-defined, because ARTs are acyclic (see Proposition 7.2 in [2]).

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

Definition 6 (Most evaluated form with finite maps).

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

$$\mathrm{mea}_{\mathcal{G}}(n) = \mathrm{mea}\Gamma_{\mathcal{G}}(\mathcal{G}(\lceil n \rceil_{\mathcal{G}}))$$
$$\mathrm{mea}\Gamma_{\mathcal{G}}(a) = a$$
$$\mathrm{mea}\Gamma_{\mathcal{G}}(m\,n) = \mathrm{mea}_{\mathcal{G}}(m)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$$

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

The most evaluated applicative form $\mathrm{mea}_{\mathcal{G}}(n)$ always contains an atom in the left-most position (it is an application of an atom or just an atom).

The definition of the most evaluated form $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ distinguishes three cases. A partial application of a function symbol is represented as a finite map. A full or over-application of a function symbol 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[2]. An empty map $\{\}$ represents both an unevaluated term and a functional value that was never applied to sufficient arguments. Distinguishing the two cases just complicates 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

[2] 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.

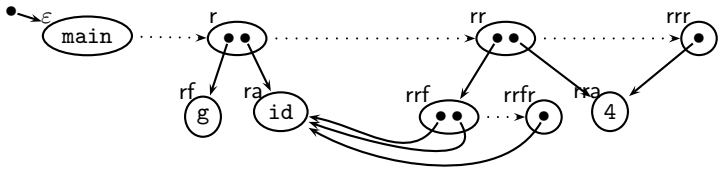

Fig. 3. The ART $\mathcal{I}$ of the full computation of a program requiring rank-2 types

```
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. 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}.

3.2 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 7 (Redexes and equations). *Let n be a redex node in $\mathcal{G}$.*

$$\mathrm{equation}^{\mathsf{P}}_{\mathcal{G}}(n) = \mathrm{redex}^{\mathsf{P}}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{P}}_{\mathcal{G}}(n)$$
$$\mathrm{equation}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$$

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

3.3 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 3 shows the ART $\mathcal{I}$ of the full computation. We have:

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

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 [5]. 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.

Alternatively 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 of functional type). The additional indirections would enable us to distinguish different instances of the same function. 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 [2,7,4]` 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 {} [] = []
```

4 The Computation Tree

Algorithmic debugging is based on the representation of the computation, which yielded the erroneous result, as a *computation tree*. Each node of a computation tree must be labelled with a subcomputation that on its own can be judged to be either correct ($\surd$), that is agreeing with the user's intentions, or incorrect ($\times$). The user's `yes`/`no` answers direct a path through the tree to a node that is associated with the faulty definition [6].

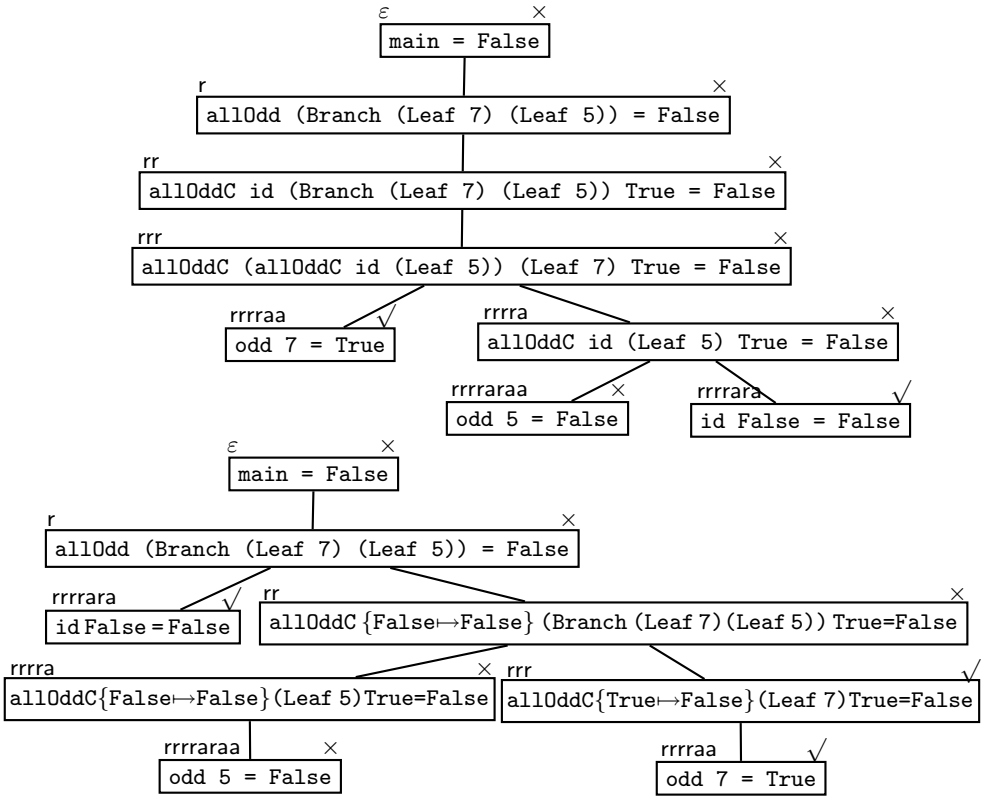

Fig. 4. EDT and FDT for the full computation of the tree traversal program

Figure 4 shows two computation trees for our tree traversal program, the standard *evaluation dependency tree (EDT)* [7], where functions are represented as partial applications, and the *function dependency tree (FDT)*, where functions are represented as finite maps. Each node is labelled with the big-step reduction of a function symbol with argument values to its result value. Each node is associated with the definition of this function symbol.

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 for every faulty node its associated slice 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 correct (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 [6]).

Both EDT and FDT have a node for each redex node in the ART (excluding reductions of trusted predefined functions such as `(&&)` and `mod`). In the pre-

ceding section we defined the equations of the nodes, which differ only in how functional values are represented. The main difference between EDT and FDT is their structure. In the EDT a node $g\,M'_1 \ldots M'_{k'} = N'$ is a child of a node $f\,M_1 \ldots M_k = N$, if $g\,M'_1 \ldots M'_k$ was called from function f, more precisely, if the *application* $g\,M'_1 \ldots M'_{k'}$ appears in the right hand side of the definition of f. In contrast, in the FDT a node $g\,M'_1 \ldots M'_{k'} = N'$ is a child of a node $f\,M_1 \ldots M_k = N$, if the *function symbol* g appears in the right hand side of the function f. Intuitively the function symbol is relevant, because the node $f\,M_1 \ldots M_k = N$ considers g to be equivalent to its finite map representation, which is justified by children such as $g\,M'_1 \ldots M'_{k'} = N'$.

So 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 the redex node that caused its creation:

Definition 8 (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 1 $\mathrm{parent}(\mathsf{rr}) = \mathsf{r}$ and $\mathrm{parent}(\mathsf{rrfff}) = \mathsf{r}$.

We can identify the function node of a redex node:

Definition 9 (Function node). *Let n be a redex node of an ART $\mathcal{G}$.*

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

Definition 10 (EDT, FDT). *The set of* tree nodes, $\mathrm{treeNodes}_{\mathcal{G}}$, *is the set of redex nodes n such that* $\mathrm{fun}_{\mathcal{G}}(n)$ *is not a predefined function symbol.*

The evaluation dependency tree (EDT) *for an ART $\mathcal{G}$ consists of the tree nodes* $\mathrm{treeNodes}_{\mathcal{G}}$ *labelled with* $\mathrm{equation}^{\mathsf{P}}_{\mathcal{G}}$ *and related via* parent *[4].*

The function dependency tree (FDT) *for an ART $\mathcal{G}$ consists of the tree nodes* $\mathrm{treeNodes}_{\mathcal{G}}$ *labelled with* $\mathrm{equation}^{\mathsf{M}}_{\mathcal{G}}$ *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

To prove that the FDT has the fault location property, that is, the function definition associated with a faulty node is faulty, we first have to clarify what we mean by a reduction or a function definition being correct. We say that a reduction $f\,M_1 \ldots M_k = N$ is correct if and only if $f\,M_1 \ldots M_k \sqsupseteq N$ 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 11. *An* intended semantics *is a binary relation on computation terms $\sqsupseteq$ 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:* $\{N_1 \mapsto M_1, \ldots, N_k \mapsto M_k\}\,N_i \sqsupseteq M_i$
6. *Abstraction:* $ON_1 \sqsupseteq M_1 \wedge \ldots \wedge ON_k \sqsupseteq M_k \Rightarrow O \sqsupseteq \{N_1 \mapsto M_1, \ldots, N_k \mapsto M_k\}$

The last two properties state that a finite map is a function as described by its entries. $\sqsupseteq$ is a partial order 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 the following we just assume that an intended semantics exists.

We already defined $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n)$ for a redex node n. We still need to define how we can also reconstruct from a redex node n the reduct of the reduction, that is, the instance of the right hand side of the program equation used for the reduction.

Definition 12 (Reduct of a redex node).

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

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

E.g., $\mathrm{reduct}^{\mathsf{M}}_{\mathcal{F}}(r) =$ `allOddC {False↦False} (Branch (Leaf 7) (Leaf 5)) True`.

Definition 13 (Correctness and faultiness in the FDT).

$$\begin{aligned} \textit{Tree node } n \textit{ correct} &\iff \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n) \\ \textit{Tree node } n \textit{ faulty} &\iff \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) \\ \textit{Program equation } L = R \textit{ faulty} &\iff \exists\sigma.\ L\sigma \not\sqsupseteq R\sigma \end{aligned}$$

The expected connection between faulty nodes and program equations holds:

Proposition 1. *If a tree node is faulty, then its associated program equation is faulty.*

Proof. Analogous to Proposition 8.9 of [2]. □

Proposition 2 makes a statement about the intended semantics of a function symbol and Proposition 3 about the intended semantics of application. Based on these two propositions we prove 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. For simplicity we assume that every redex node is a tree node.

Proposition 2. *Let* $n \in \mathrm{dom}(\mathcal{G})$ *with* $n \notin \mathrm{treeNodes}_{\mathcal{G}}$ *and* $\mathcal{G}(n) = f$ *for some function symbol* f *with* $\mathrm{arity}(f) > 0$. *If for all* $m \in \mathrm{treeNodes}_{\mathcal{G}}$ *it is the case that* $\mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parent}(n)$ *implies* $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(m) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)$ *, then* $f \sqsupseteq \mathrm{fMap}_{\mathcal{G}}(n)$.

Proof. We prove the more general property that if $n \in \mathrm{dom}(\mathcal{G})$ with $n \notin \mathrm{treeNodes}_{\mathcal{G}}$ and $\mathrm{mea}_{\mathcal{G}}(n) = f\, N_1 \ldots N_k$ for some function symbol f and computation terms $N_1 \ldots N_k$ with $\mathrm{arity}(f) > k \geq 0$, then

$$(\forall m \in \mathrm{treeNodes}_{\mathcal{G}}\,.\, \mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parentFDT}_{\mathcal{G}}(n) \Rightarrow \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(m) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)) \implies \mathrm{mea}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$$

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

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

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

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

Proposition 4 (Correctness of the Reduct). *If* n *is a tree node and all its children are correct, then* $\mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$.

Proof. $\mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{r})$. $\mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n\mathsf{r}) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ follows from the more general property

$$\forall n \in \mathrm{dom}(\mathcal{G}).\ (\forall m \in \mathrm{treeNodes}_{\mathcal{G}}\,.\ \mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parent}(n) \Rightarrow \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(m) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)) \implies \mathrm{reductB}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$$

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

Corollary 1 (FDT is a computation tree).
If a tree node is incorrect and all its children are correct, then the tree node is faulty, that is,

$$\begin{aligned}&\forall n \in \mathrm{treeNodes}_{\mathcal{G}}\,.\ (\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)) \wedge \\ &(\forall m \in \mathrm{treeNodes}_{\mathcal{G}}\,.\ \mathrm{parentFDT}_{\mathcal{G}}(m) = n \implies \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(m) \sqsupseteq \mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(m)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \implies \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)\end{aligned}$$

Proof. According to Proposition 4 we have $\mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$. Assume $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$. By transitivity $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$ in contradiction to our hypothesis. Hence $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) \not\sqsupseteq \mathrm{reduct}^{\mathsf{M}}_{\mathcal{G}}(n)$. □

6 Related Work

Naish [6] 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 [7] 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.* [1] 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) [2] 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 [4]. 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 [8, 9] 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.

7 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. Thus we proved the soundness of algorithmic debugging with the FDT, that is, that every located fault is indeed a fault.

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.

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 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.

References

1. Rafael Caballero, Francisco J. López-Fraguas, and Mario Rodríguez-Artalejo. Theoretical foundations for the declarative debugging of lazy functional logic programs. In Herbert Kuchen and Kazunori Ueda, editors, *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings*, LNCS 2024, pages 170–184. Springer, 2001.
2. Olaf Chitil and Yong Luo. Structure and properties of traces for functional programs. In Ian Mackie, editor, *Proceedings of the 3rd International Workshop on Term Graph Rewriting, Termgraph 2006*, ENTCS 176(1), pages 39–63, 2007.
3. Thomas Davie and Olaf Chitil. Display of functional values for debugging. In *Draft Proceedings of IFL 2006*, pages 326–337, Budapest, Hungary, September 2006. Eötvös Loránd University. Technical Report No 2006-SO1.
4. Yong Luo and Olaf Chitil. Proving the correctness of algorithmic debugging for functional programs. In *Trends in Functional Programming*, volume 7, pages 19–34. Intellect Books, 2007.
5. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.
6. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.
7. Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering: An International Journal*, 4(2):121–150, April 1997.
8. Bernie Pope. Declarative debugging with Buddha. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming, 5th International School, AFP 2004*, LNCS 3622, pages 273–308. Springer Verlag, September 2005.
9. Bernie Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.
10. Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin 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.

A 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* $\mathrm{arity}(f)$ *arguments cannot be a tree node, that is*

$$\mathcal{G}(n) = p\,o \wedge \mathrm{mea}_{\mathcal{G}}(p) = a\,N_1 \ldots N_k \wedge (a = C \vee \mathrm{arity}(a) \neq k+1) \\ \implies n \notin \mathrm{treeNodes}_{\mathcal{G}}$$

Proof. Assume $n \in \mathrm{treeNodes}_{\mathcal{G}}$. Then $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n) = \mathrm{mea}\mathbb{T}_{\mathcal{G}}(p\,o) = \mathrm{mea}_{\mathcal{G}}(p)\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = a\,N_1 \ldots N_k\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. Following Lemma 8.9 of [2] $\mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(n)$ 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 $\mathrm{arity}(f)$ patterns. Because of this contradiction our original assumption that $n \in \mathrm{treeNodes}_{\mathcal{G}}$ must be wrong. □

Proof of Proposition 2
We prove the more general property that if $n \in \mathrm{dom}(\mathcal{G})$ with $n \notin \mathrm{treeNodes}_{\mathcal{G}}$ and $\mathrm{mea}_{\mathcal{G}}(n) = f\,N_1 \ldots N_k$ for some function symbol f and computation terms $N_1 \ldots N_k$ with $\mathrm{arity}(f) > k \geq 0$, then

$$(\forall m \in \mathrm{treeNodes}_{\mathcal{G}}\ .\ \mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parentFDT}_{\mathcal{G}}(n) \Rightarrow \mathrm{redex}^{\mathsf{M}}_{\mathcal{G}}(m) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)) \\ \implies \mathrm{mea}_{\mathcal{G}}(n) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(n)$$

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

case $j = 1$:
Let $m \in \mathrm{dom}(\mathcal{G})$ with $G(m) = n'\,o$ and $n' \succ^*_{\mathcal{G}} n$ and $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) \neq \{\}$.
Assume $m \notin \mathrm{treeNodes}_{\mathcal{G}}$. Then we have $\mathrm{mea}_{\mathcal{G}}(m) = f\,N_1 \ldots N_k\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. Because $\mathrm{arity}(f) = k+1$ we get $\mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m) = \{\}$ 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 $\mathrm{redex}_{\mathcal{G}}(m) = \mathrm{mea}\mathbb{T}_{\mathcal{G}}(G(m)) = \mathrm{mea}\mathbb{T}_{\mathcal{G}}(n'\,o) = \mathrm{mea}_{\mathcal{G}}(n')\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o) = f\,N_1 \ldots N_k\ \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(o)$. Furthermore $\mathrm{parentFDT}_{\mathcal{G}}(m) = \mathrm{parentFDT}_{\mathcal{G}}(n)$. Therefore the hypothesis gives us $\mathrm{redex}_{\mathcal{G}}(m) \sqsupseteq \mathrm{mef}^{\mathsf{M}}_{\mathcal{G}}(m)$ and so 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 $m \in \text{treeNodes}_\mathcal{G}$ with $\text{parentFDT}_\mathcal{G}(m) = \text{parentFDT}_\mathcal{G}(n)$ we know that $\text{parentFDT}_\mathcal{G}(m) = \text{parentFDT}_\mathcal{G}(p)$. So for all $m \in \text{treeNodes}_\mathcal{G}$ we have $\text{parentFDT}_\mathcal{G}(m) = \text{parentFDT}_\mathcal{G}(p)$ implies $\text{redex}_\mathcal{G}(m) \sqsupseteq \text{mef}^\mathsf{M}_\mathcal{G}(m)$. Hence $\text{mea}_\mathcal{G}(p) \sqsupseteq \text{mef}^\mathsf{M}_\mathcal{G}(p)$.
Together with $\text{mea}_\mathcal{G}(p) = \text{mea}_\mathcal{G}(n)\ \text{mef}^\mathsf{M}_\mathcal{G}(o)$ we obtain that $\text{mea}_\mathcal{G}(n)\ \text{mef}^\mathsf{M}_\mathcal{G}(o) \sqsupseteq \text{mef}^\mathsf{M}_\mathcal{G}(p)$.
Because we have this last relationship for any p we obtain with the abstraction property of the intended semantics $\text{mea}_\mathcal{G}(n) \sqsupseteq \{\text{mef}^\mathsf{M}_\mathcal{G}(o) \mapsto \text{mef}^\mathsf{M}_\mathcal{G}(p) \mid \mathcal{G}(p) = n'\,o \wedge n' \succ^*_\mathcal{G} n \wedge \text{mef}^\mathsf{M}_\mathcal{G}(p) \neq \{\}\}$. The condition $\text{mef}^\mathsf{M}_\mathcal{G}(p) \neq \{\}$ is not necessary, but the relationship still holds with additional conditions. We conclude that $\text{mea}_\mathcal{G}(n) \sqsupseteq \text{fMap}_\mathcal{G}(n) = \text{mef}^\mathsf{M}_\mathcal{G}(n)$. □

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

case $\text{mea}_\mathcal{G}(p) = f\,N_1 \ldots N_k$ and $\text{arity}(f) > k \geq 0$:
So $\text{mef}^\mathsf{M}_\mathcal{G}(p) = \text{fMap}_\mathcal{G}(p) = \{\text{mef}^\mathsf{M}_\mathcal{G}(o') \mapsto \text{mef}^\mathsf{M}_\mathcal{G}(m) \mid G(m) = p'\,o' \wedge p' \succ^*_\mathcal{G} p \wedge \text{mef}^\mathsf{M}_\mathcal{G}(m) \neq \{\}\}$.
case $\text{mef}^\mathsf{M}_\mathcal{G}(n) = \{\}$:
Trivially $\text{mef}^\mathsf{M}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(o) \sqsupseteq \{\} = \text{mef}^\mathsf{M}_\mathcal{G}(n)$.
case $\text{mef}^\mathsf{M}_\mathcal{G}(n) \neq \{\}$:
Then $\text{mef}^\mathsf{M}_\mathcal{G}(o) \mapsto \text{mef}^\mathsf{M}_\mathcal{G}(n) \in \text{fMap}_\mathcal{G}(p)$. With the application property we get $\text{mef}^\mathsf{M}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(o) = \text{fMap}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(o) \sqsupseteq \text{mef}^\mathsf{M}_\mathcal{G}(n)$.
case $\text{mea}_\mathcal{G}(p) = f\,N_1 \ldots N_k$ and $\text{arity}(f) \leq k$:
According to Lemma 1 $n \notin \text{treeNodes}_\mathcal{G}$. So $\text{mea}_\mathcal{G}(n) = \text{mea}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(n) = f\,N_1 \ldots N_k\ \text{mef}^\mathsf{M}_\mathcal{G}(n)$ and $\text{arity}(f) \leq k$. Therefore $\text{mef}^\mathsf{M}_\mathcal{G}(n) = \{\}$. Trivially $\text{mef}^\mathsf{M}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(o) \sqsupseteq \{\} = \text{mef}^\mathsf{M}_\mathcal{G}(n)$.
case $\text{mea}_\mathcal{G}(p) \neq f\,N_1 \ldots N_k$:
Then $\text{mea}_\mathcal{G}(p) = C\,N_1 \ldots N_k$ for some constructor C and terms $N_1 \ldots N_k$. According to Lemma 1 $n \notin \text{treeNodes}_\mathcal{G}$. So $\text{mea}_\mathcal{G}(n) = \text{mea}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(n) = C\,N_1 \ldots N_k\ \text{mef}^\mathsf{M}_\mathcal{G}(n)$. Hence $\text{mef}^\mathsf{M}_\mathcal{G}(n) = \text{mea}_\mathcal{G}(n)$. Therefore we know that $\text{mef}^\mathsf{M}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(o) = \text{mea}_\mathcal{G}(p)\ \text{mef}^\mathsf{M}_\mathcal{G}(o) = \text{mea}_\mathcal{G}(n) = \text{mef}^\mathsf{M}_\mathcal{G}(n)$. □

Proof of Proposition 4 (Correctness of the Reduct)
$\text{reduct}^\mathsf{M}_\mathcal{G}(n) = \text{reductB}^\mathsf{M}_\mathcal{G}(n\mathsf{r})$. $\text{reductB}^\mathsf{M}_\mathcal{G}(n\mathsf{r}) \sqsupseteq \text{mef}^\mathsf{M}_\mathcal{G}(n)$ follows from the more general property

$$\forall n \in \text{dom}(\mathcal{G}).$$
$$(\forall m \in \text{treeNodes}_\mathcal{G}\,.\ \text{parentFDT}_\mathcal{G}(m) = \text{parent}(n) \Rightarrow \text{redex}^\mathsf{M}_\mathcal{G}(m) \sqsupseteq \text{mef}^\mathsf{M}_\mathcal{G}(m)) \implies \text{reductB}^\mathsf{M}_\mathcal{G}(n) \sqsupseteq \text{mef}^\mathsf{M}_\mathcal{G}(n)$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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