Proving the Correctness of Algorithmic Debugging for Functional Programs

Yong Luo and Olaf Chitil

Computing Laboratory, University of Kent, Canterbury, Kent, UK
Email: {Y.Luo, O.Chitil}@kent.ac.uk

Abstract

This paper presents a formal model of tracing for functional programs based on a small-step operational semantics. The model records the computation of a functional program in a graph which can be utilised for various purposes such as algorithmic debugging. The main contribution of this paper is to prove the correctness of algorithmic debugging for functional programs based on the model. Although algorithmic debugging for functional programs is implemented in several tracers such as Hat, the correctness has not been formally proved before. The difficulty of the proof is to find a suitable induction principle and a sufficiently general induction hypothesis.

1 INTRODUCTION

Usually, a computation is treated as a black box that performs input and output actions. However, we have to look into the black box when we want to see how the different parts of the program cause the computation to perform the input/output actions. The most common need for doing this is debugging: When there is a disparity between the actual and the intended semantics of a program, we need to locate the part of the program that causes the disparity. Traditional debugging techniques are not well suited for declarative programming languages such as Haskell, because it is difficult to understand how programs execute (or their procedural meaning). Algorithmic debugging (also called declarative debugging) was invented by Shapiro [8] for logic programming languages. Later the method was transferred to other programming languages, including functional programming languages. A question of an algorithmic debugger must fully describe a subcomputation; hence algorithmic debugging works best for purely declarative languages, which do not use side-effects but make all data and control flow explicit. As Haskell is a purely functional programming language that even separates input/output operations from the rest of the language, it is particularly suitable for algorithmic debugging. There exists three algorithmic debuggers for Haskell: Freja [4], Hat [11] and Buddha/Plargleflarp [7].

In contrast to this advance of algorithmic debugging in practise and the relative simplicity of the underlying idea, there are few theoretical foundations and no proofs that these debuggers do actually work correctly. We need a full understanding of algorithmic debugging for functional languages to determine its limits and to develop more powerful extensions and variations. That is the problem we

address in this paper. We shall give a direct and simple definition of *trace* that will enable us to formally relate a view to the semantics of a program. The *evaluation dependency tree (EDT)* will be generated from a computation graph. We can correctly locate program faults, and the correctness will be formally proved. This is a non-trivial proof since the simple induction principle, the size of graph, does not work.

In the next section we give a brief overview of algorithmic debugging. Related work is also discussed. In Section 3, some basic definitions and the *augmented redex trail* (ART) are formally presented. In Section 4, we show how to generate an EDT from an ART. In Section 5, we prove the properties of an EDT, in particular, the correctness of algorithmic debugging. Future work will be discussed in the last section.

2 ALGORITHMIC DEBUGGING

Algorithmic debugging can be thought of searching a fault in a program. When a program execution has produced a wrong result an algorithmic debugger will ask the programmer a number of questions about the computation. Each question asks whether a given subcomputation is correct, that is, whether it agrees with the intentions of the programmer. After a number of questions and answers the algorithmic debugger gives the location of a fault in the program.

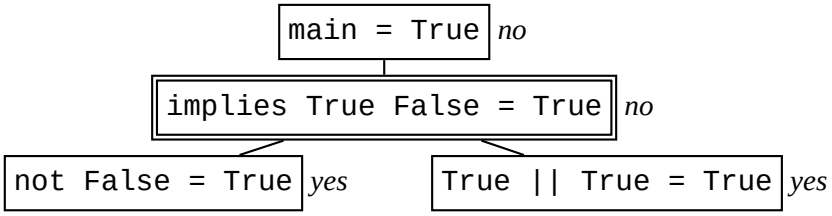
For example, for an execution of the Haskell program

```
main = implies True False
implies x y = not y || x
```

a session with an algorithmic debugger might be as follows, with answers given by the programmer in italics:

```
1) main = True ?                  no
2) implies True False = True ?    no
3) not False = True ?             yes
4) True || True = True ?          yes
Fault located. The definition of 'implies' is faulty.
```

The principle of algorithmic debugging is relatively simple. From the computation that produces the wrong result a *computation tree* is built; each node of the computation tree is labelled with a question about a subcomputation:



An algorithmic debugger traverses the computation tree asking the questions of the nodes until it locates a so-called *faulty node*, that is, a node whose computation is erroneous according to the programmer's intentions, but the computations of all its children are correct. The algorithmic debugger reports the definition of the function reduced in the faulty node as the fault location.

Naish [3] 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. Soundness depends on the exact definition of the computation tree. Programming languages with different semantics, for example logic languages vs. functional languages, require different definitions of the computation tree. Even for a single programming language several definitions are possible. For lazy functional programming languages Nilsson and Sparud [6, 5, 9] 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. The example computation tree given above is an EDT. The algorithmic debuggers Freja, Hat and Buddha/Plargleflarp are based on the EDT. The construction of an EDT during the computation of a program is non-trivial, because the structure of the EDT is very different from the structure of the computation as determined by the evaluation order.

For a lazy functional logic language Caballero et al. [1] give a formal definition of an EDT and sketch a soundness proof of algorithmic debugging. However, this approach relies on the EDT being defined through a high-level non-deterministic big-step semantics[1]. Thus this definition of the EDT is far removed from any real implementation of an algorithmic debugger.

3 FORMALISING THE AUGMENTED REDEX TRAIL (ART)

An augmented redex trail (ART) is a graph that represents a computation of a functional program. A graph enables sharing of subexpressions which is the key both to a space efficient trace structure and closeness to the implementations of functional languages. The one essential difference to standard graph rewriting of functional language implementations is that ART rewriting does not overwrite a redex with its reduct, but adds the reduct to the graph, keeping the reduct and thus the computation history.

In this section we give some basic definitions which will be used throughout the paper, and we describe how to build an ART.

Definition 1. ***(Atoms, Terms. Patterns, Rewriting rule and Program)***

[1] Non-determinism is essential for this approach, irrespective of whether the programming language has logical features or not.

- ***Atoms*** *consist of function symbols and constructors.*
- ***Terms:*** *(1) an atom is a term; (2) a variable is a term; (3) MN is a term if M and N are terms.*
- ***Patterns:*** *(1) a variable is a pattern; (2) $cp_1 \ldots p_n$ is a pattern if c is a constructor and p_1,..., p_n are patterns, and the arity of c is n.*
- *A* ***rewriting rule*** *is of the form $f\ p_1 \ldots p_n = R$ where f is a function symbol and p_1,..., p_n ($n \geq 0$) are patterns and R is a term.*
- *A* ***program*** *is a finite set of rewriting rules.*

Example 2. $id\ x = x$, $not\ True = False$, $map\ f\ (x : xs) = f\ x\ :\ map\ f\ xs$ and $ones = 1 : ones$ are rewriting rules.

Note that we only allow disjoint patterns if there is more than one rewriting rule for a function. We also require that the number of arguments of a function in the left-hand side must be the same. For example, if there is a rewriting rule $f\ c_1 = g$, then $f\ c_2\ c_3 = c_4$ is not allowed. The purpose of disjointness is to prevent us from giving different values to the same argument when we define a function. Disjointness is one of the ways to guarantee the property of Church-Rosser. In many programming languages such as Haskell the requirement of disjointness is not needed, because the patterns for a function have orders. If a closed term matches the first pattern, the algorithm will not try to match the rest patterns. We also require that all the patterns are linear because conversion test is difficult sometimes. Many functional programming languages such as Haskell only allow linear patterns.

Now, we define computation graphs and choose a particular naming scheme to name the nodes in a computation graph. The letters f and a mean the function component and the argument component of an application respectively. The letter r means a small step of reduction.

Definition 3. ***(Node, Node expression and Computation graph)***

- *A* ***node*** *is a sequence of letters* r, f *and* a, i.e. $\{\mathsf{r}, \mathsf{f}, \mathsf{a}\}^*$.
- *A* ***node expression*** *is either an atom, or a node, or an application of two nodes, which is of the form $m \circ n$.*
- *A* ***computation graph*** *is a set of pairs which are of the form (n, e), where n is a node and e is a node expression.*

Example 4. We have a Haskell program:

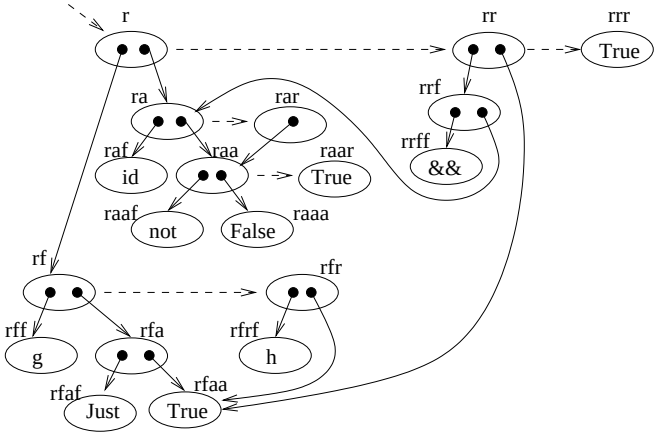
$$g\ (Just\ x) = h\ x$$

$$h\ x\ y = y\ \&\&\ x$$

The following is a computation graph for the starting term $g\ (Just\ True)\ (id\ (not\ False))$.

$\{(\mathsf{r}, \mathsf{rf} \circ \mathsf{ra}), (\mathsf{rf}, \mathsf{rff} \circ \mathsf{rfa}), (\mathsf{rff}, g), (\mathsf{rfa}, \mathsf{rfaf} \circ \mathsf{rfaa}), (\mathsf{rfaf}, \mathit{Just}), (\mathsf{rfaa}, \mathit{True}),$
$(\mathsf{ra}, \mathsf{raf} \circ \mathsf{raa}), (\mathsf{raf}, \mathit{id}), (\mathsf{raa}, \mathsf{raaf} \circ \mathsf{raaa}), (\mathsf{raaf}, \mathit{not}), (\mathsf{raaa}, \mathit{False}),$
$(\mathsf{rar}, \mathsf{raa}), (\mathsf{raar}, \mathit{True}), (\mathsf{rfr}, \mathsf{rfrf} \circ \mathsf{rfaa}), (\mathsf{rfrf}, h), (\mathsf{rr}, \mathsf{rrf} \circ \mathsf{rfaa}), (\mathsf{rrf}, \mathsf{rrff} \circ \mathsf{ra}),$
$(\mathsf{rrff}, \&\&), (\mathsf{rrr}, \mathit{True})\}$

It can be depicted as follows:



The dashed edges represent the computation steps. If a node $m\mathsf{r}$ is in a graph then there is a computation from the node m to $m\mathsf{r}$. So, the pairs of the form $(m, m\mathsf{r})$ are omitted in the formal representation of the graph. For example, $(\mathsf{r}, \mathsf{rr})$ and $(\mathsf{rf}, \mathsf{rfr})$ are not included in the above graph.

Notation: $\mathit{dom}(G)$ denotes the set of nodes in a computation graph G.

Pattern matching in a graph

The pattern matching algorithm for a graph has two different results, either a set of substitutions or "doesn't match".

- The final node in a sequence of reductions starting at node m, $\mathit{last}(G, m)$:

$$\mathit{last}(G, m) \quad = \quad \begin{cases} \mathit{last}(G, m\mathsf{r}) & \text{if } m\mathsf{r} \in \mathit{dom}(G) \\ \mathit{last}(G, n) & \text{if } (m, n) \in G \text{ and } n \text{ is a node} \\ m & \text{otherwise} \end{cases}$$

  The purpose of this function is to find out the most evaluated point for m. For example, if G is the graph in Example 4, then we have $\mathit{last}(G, \mathsf{r}) = \mathsf{rrr}$ and $\mathit{last}(G, \mathsf{ra}) = \mathsf{raar}$.

- The head of the term at node m, $\mathit{head}(G, m)$, where G is a graph and m is a node in G:

$$\mathit{head}(G, m) \quad = \quad \begin{cases} \mathit{head}(G, \mathit{last}(G, i)) & \text{if } (m, i \circ j) \in G \\ a & \text{if } (m, a) \in G \text{ and } a \text{ is an atom} \\ \text{undefined} & \text{otherwise} \end{cases}$$

For example, if G is the graph in Example 4, then we have $head(G,\mathsf{r}) = h$ and $head(G,\mathsf{rf}) = g$.

- The arguments of the function at node m, $args(G,m)$:

$$args(G,m) \quad = \quad \begin{cases} \langle args(G, last(G,i)), j\rangle & \text{if } (m, i\circ j) \in G \\ \langle\rangle & \text{otherwise} \end{cases}$$

Note that the arguments of a function are a sequence of nodes. For example, if G is the graph in Example 4, then we have $args(G,\mathsf{r}) = \langle \mathsf{rfaa}, \mathsf{ra}\rangle$ and $args(G,\mathsf{ra}) = \langle \mathsf{raa}\rangle$.

Now, we define two functions $match_1$ and $match_2$ which are mutually recursive. The arguments of $match_1$ are a node and a pattern. The arguments of $match_2$ are a sequence of nodes and a sequence of patterns.

- $match_1(G,m,x) = [m/x]$ where x is a variable.

$$\begin{array}{l} match_1(G,m,cq_1...q_k) \\ = \begin{cases} match_2(G, args(G,m'), \langle q_1,...,q_k\rangle) & \text{if } head(G,m') = c \\ \text{does not match} & \text{otherwise} \end{cases} \end{array}$$

where $m' = last(G,m)$.

-

$$\begin{array}{l} match_2(G, \langle m_1,...,m_n\rangle, \langle p_1,...,p_n\rangle) \\ = match_1(G,m_1,p_1) \cup ... \cup match_1(G,m_n,p_n) \end{array}$$

where $\cup$ is the union operator. Notice that if $n = 0$ then

$$match_2(G, \langle\rangle, \langle\rangle) = [\,]$$

If any m_i does not match p_i, $\langle m_1,...,m_n\rangle$ does not match $\langle p_1,...,p_n\rangle$. If the length of two sequences are not the same, they do not match. For example, $\langle m_1,...,m_s\rangle$ does not match $\langle p_1,...,p_{s'}\rangle$ if $s \neq s'$.

- We say that G at node m matches the left-hand side of a rewriting rule $f\,p_1...p_n = R$ with $[m_1/x_1,...,m_k/x_k]$ if $head(G,m) = f$ and

$$match_2(G, args(G,m), \langle p_1,...,p_n\rangle) = [m_1/x_1,...,m_k/x_k]$$

In the substitution form $[m/x]$, m is not a term but a node. In Example 4, the graph at node r matches $h\,x\,y$ with $[\mathsf{rfaa}/x, \mathsf{ra}/y]$. The definition of pattern matching and its result substitution sequence will become important for making computation order irrelevant when we generate graphs. In Example 4, no matter which node is reduced first, ra or raa, the final graph will be the same.

Graph for label terms. During the computations all the variables in a term will be substituted by some nodes. When the variables are substituted by a sequence

of shared nodes, it becomes a label term. For example, $(y \,\&\&\, x)[\mathsf{rfaa}/x, \mathsf{ra}/y] \equiv \mathsf{ra} \,\&\&\, \mathsf{rfaa}$ is a label term. The function *graph* defined in the following has two arguments: a node and a label term. The result of *graph* is a computation graph.

$$
\begin{array}{rcl}
graph(n,e) & = & \{(n,e)\} \quad \text{where } e \text{ is an atom or a node} \\
graph(n,MN) & = & \begin{cases}
\{(n, M \circ N)\} & \text{if } M \text{ and } N \text{ are nodes} \\
\{(n, M \circ n\mathsf{a})\} \cup graph(n\mathsf{a}, N) & \text{if only } M \text{ is a node} \\
\{(n, n\mathsf{f} \circ N)\} \cup graph(n\mathsf{f}, M) & \text{if only } N \text{ is a node} \\
\{(n, n\mathsf{f} \circ n\mathsf{a})\} \cup graph(n\mathsf{f}, M) & \text{otherwise} \\
\quad \cup graph(n\mathsf{a}, N) &
\end{cases}
\end{array}
$$

Building an ART

- For a start term M, the start ART is $graph(\mathsf{r}, M)$. Note that the start term has no nodes inside.
- ***(ART rule)*** If an ART G at m matches the left-hand side of a rewriting rule $f\,p_1 \ldots p_n = R$ with $[m_1/x_1, \ldots, m_k/x_k]$, then we can build a new ART

$$G \cup graph(m\mathsf{r}, R[m_1/x_1, \ldots, m_k/x_k])$$

- An ART is generated from a start ART and by applying the *ART rule* repeatedly. Note that the order in which nodes are chosen has no influence in the final graph.

Example 5. If the start term is $g\ (Just\ True)\ (id\ (not\ False))$ as in Example 4, then the start graph is

$$\{(\mathsf{r}, \mathsf{rf} \circ \mathsf{ra}), (\mathsf{rf}, \mathsf{rff} \circ \mathsf{rfa}), (\mathsf{rff}, g), (\mathsf{rfa}, \mathsf{rfaf} \circ \mathsf{rfaa}), (\mathsf{rfaf}, Just), (\mathsf{rfaa}, True),$$
$$(\mathsf{ra}, \mathsf{raf} \circ \mathsf{raa}), (\mathsf{raf}, id), (\mathsf{raa}, \mathsf{raaf} \circ \mathsf{raaa}), (\mathsf{raaf}, not), (\mathsf{raaa}, False)\}$$

The new parts built from r and ra are

$$
\begin{array}{rl}
 & graph(\mathsf{rr}, (y \,\&\&\, x)[\mathsf{rfaa}/x, \mathsf{ra}/y]) \\
= & graph(\mathsf{rr}, (\mathsf{ra} \,\&\&\, \mathsf{rfaa})) \\
= & \{(\mathsf{rr}, \mathsf{rrf} \circ \mathsf{rfaa}), (\mathsf{rrf}, \mathsf{rrff} \circ \mathsf{ra}), (\mathsf{rrff}, \&\&)\}
\end{array}
$$

$$graph(\mathsf{rar}, x[\mathsf{raa}/x]) = \{(\mathsf{rar}, \mathsf{raa})\}$$

Note that the order of computation is irrelevant because the result of pattern matching at the node ra is always $[\mathsf{raa}/x]$, no matter which node is computed first. The definition of pattern matching simplifies the representation of ART. Otherwise we would have several structurally different graphs representing the same reduction step. Multiple representations just cause confusion and would later lead us to give a complex definition of an equivalence class of graphs.

The following simple properties of an ART will be used later.

Lemma 6. *Let G be an ART.*

- *If* $m \in dom(G)$ *then there is at least one letter* r *in* m.
- *If* $m\mathsf{r} \in dom(G)$ *then* $m \in dom(G)$ *or* $m = \varepsilon$ *where* ε *is the empty sequence.*
- *If* $m\mathsf{r} \in dom(G)$ *then* $(m,n) \notin G$ *for any node* n.

Proof. The first and second are trivial. The third is proved by contradiction. If $(m,n) \in G$ then $head(G,m)$ is undefined. There cannot be a computation at m, *i.e.* $m\mathsf{r} \notin G$.

4 GENERATING AN EVALUATION DEPENDENCY TREE

In this section we generate the *Evaluation Dependency Tree* (EDT) from a given ART.

The real Hat ART also includes so-called *parent edges*. Each node has a parent edge that points to the top of the redex that caused its creation. Parent edges are key ingredient for the redex trail view of locating program faults [10]. One may notice that there are no parent edges in the ART here. They need not be given explicitly because the way that the nodes are labelled gives us the parents of all nodes implicitly.

Definition 7. ***(Parent edges)***

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

Note that $parent(\mathsf{r}) = \varepsilon$ where ε is the empty sequence.

Definition 8. ***(****children* ***and*** *tree****)*** *Let* G *be an ART, and* $m\mathsf{r}$ *a node in* G *(*i.e. $m\mathsf{r} \in dom(G)$*).*

children and tree are defined as follows.

- *children*

  $$children(m) = \{n \mid parent(n) = m \text{ and } n\mathsf{r} \in dom(G)\}$$

  The condition $n\mathsf{r} \in dom(G)$ *is to make sure that only evaluated nodes become children.*

- *tree*

  $$tree(m) = \{(m,n_1),...,(m,n_k)\} \cup tree(n_1) \cup ... \cup tree(n_k)$$

  where $\{n_1,...,n_k\} = children(m)$

Example 9. If G is the graph in Example 4 then

$$tree(\varepsilon) = \{(\varepsilon,\mathsf{r}),(\varepsilon,\mathsf{ra}),(\varepsilon,\mathsf{raa}),(\varepsilon,\mathsf{rf}),(\mathsf{r},\mathsf{rr})\}$$

Notation: In the above definitions such as *children*, the ART G should be one of the arguments but it is omitted. For example, we write $children(m)$ for $children(G,m)$. We shall use this notation later when no confusion may occur.

Usually, a single node of a computation graph represents many different terms. We are particularly interested in two kinds of terms of nodes, the most evaluated form and the redex.

Definition 10. ***(Most Evaluated Form)*** *Let G be an ART. The most evaluated form of a node m is a term and is defined as follows.*

$$mef(m) = \begin{cases} mef(m\mathsf{r}) & \text{if } m\mathsf{r} \in dom(G) \\ meft(m) & \text{otherwise} \end{cases}$$

where

$$meft(m) = \begin{cases} a & (m,a) \in G \text{ and } a \text{ is an atom} \\ mef(n) & (m,n) \in G \text{ and } n \text{ is a node} \\ mef(i)\ mef(j) & (m, i \circ j) \in G \end{cases}$$

One may also use the definition of $last(G,m)$ to define the most evaluated form.

Example 11. If G is the graph in Example 4, then

$$mef(\mathsf{r}) = mef(\mathsf{rr}) = meft(\mathsf{rrr}) = True$$

$$mef(\mathsf{ra}) = mef(\mathsf{rar}) = meft(\mathsf{rar}) = mef(\mathsf{raa}) = True$$

Definition 12. ***(redex)*** *Let G be an ART, and $m\mathsf{r}$ a node in G (*i.e. $m\mathsf{r} \in dom(G)$*). redex is defined as follows.*

- $redex(\varepsilon) = main$
- $redex(m) = \begin{cases} mef(i)\ mef(j) & \text{if } (m, i \circ j) \in G \\ a & \text{if } (m,a) \in G \text{ and } a \text{ is an atom} \end{cases}$

Note that the case $(m,n) \in G$ is not defined in this definition because $(m,n) \notin G$ for any node n by Lemma 6.

Example 13. If G is the graph in Example 4, then

$$redex(\mathsf{r}) = mef(\mathsf{rf})\ mef(\mathsf{ra}) = h\ True\ True$$

Now, we define the evaluation dependency tree of a graph.

Definition 14. ***(Evaluation Dependency Tree)*** *Let G be an ART. The evaluation dependency tree (EDT) of G consists of the following two parts.*

1. *The set* $tree(\varepsilon)$*;*

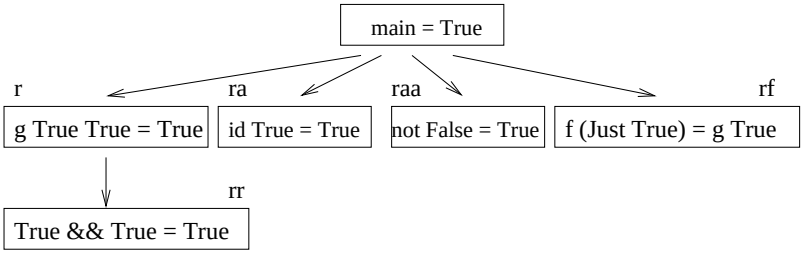
2. *The set of equations; for any node in* $tree(\varepsilon)$ *there is a corresponding equation* $redex(m) = mef(m)$.

Note that we write $mef(\varepsilon)$ for $mef(\mathsf{r})$.

Notation: For an EDT T, $dom(T)$ denotes the set of all the nodes in $tree(\varepsilon)$. We also say $(m,n) \in T$ if $(m,n) \in tree(\varepsilon)$.

$redex(m) = mef(m)$ represents an evaluation at node m from the left-hand side to the right-hand side. A pair (m,n) in an EDT represents that the evaluation $redex(m) = mef(m)$ depends on the evaluation $redex(n) = mef(n)$.

Example 15. The EDT for the graph in Example 4 is the following.



5 PROPERTIES OF AN EDT

In this section, we present the properties of the EDT and prove the correctness of algorithmic debugging.

The following theorems suggest that the EDT of an ART covers all the computation in the ART. Although two evaluations may rely on the same evaluation in an ART, every evaluation for algorithmic debugging only needs to be examined once.

Lemma 16. *Let G be an ART, and T its EDT. If there is a sequence of nodes* $m_1, m_2, ..., m_k$ *such that*

$$m \in children(m_1), m_1 \in children(m_2), ...,$$
$$m_{k-1} \in children(m_k), m_k \in children(\varepsilon)$$

then $m \in dom(T)$.

Proof. By the definition of $tree(\varepsilon)$.

Lemma 17. *Let G be an ART. If* $m\mathsf{r} \in dom(G)$*, then* $m \equiv \varepsilon$ *or there is a sequence of nodes* $m_1, m_2, ..., m_k$ *such that*

$$m \in children(m_1), m_1 \in children(m_2), ...,$$
$$m_{k-1} \in children(m_k), m_k \in children(\varepsilon)$$

Proof. By induction on the size of m, and by Lemma 6.

Since $m\mathsf{r} \in dom(G)$, by Lemma 6, we only need to consider the following two cases.

- If $m = \varepsilon$, the statement is obviously true.
- If $m \in dom(G)$, by Lemma 6, there is at least one letter r in m. We consider the following two sub-cases.
  - $m = \mathsf{r}n$, where there is no r in n. Since $m\mathsf{r} \in dom(G)$ and $parent(\mathsf{r}n) = \varepsilon$, we have $\mathsf{r}n \in children(\varepsilon)$.
  - $m \equiv m_1\mathsf{r}n$, where there is no r in n. Since $m\mathsf{r} \in dom(G)$ and $parent(m) = m_1$, we have $m \in children(m_1)$. Now, because m_1 is a sub-sequence of m, by induction hypothesis, there is a sequence of index numbers $m_2, ..., m_k$ such that

    $$m_1 \in children(m_2), ..., m_{k-1} \in children(m_k), m_k \in children(\varepsilon)$$

    So, there is a sequence of index numbers $m_1, m_2, ..., m_k$ such that

    $$m \in children(m_1), m_1 \in children(m_2), ..., m_k \in children(\varepsilon)$$

Theorem 18. *Let G be an ART, and T its EDT.*

If $m\mathsf{r} \in dom(G)$, then $m \in dom(T)$. In other word, T covers all the computations in G.

Proof. By Lemma 17 and 16.

Lemma 19. *Let G be an ART, and T its EDT.*

If $(m,n) \in T$, then $n \in children(m)$ and $parent(n) \equiv m$.

Proof. By the definition of *tree*.

Theorem 20. *Let G be an ART, and T its EDT.*

If $(m,n) \in T$ and $m \not\equiv k$, then $(k,n) \notin T$.

Proof. By Lemma 19.

The above theorem suggests that every evaluation for algorithmic debugging only needs to be examined once although two evaluations may rely on the same evaluation. For example, g is defined as $g\ x = (not\ x, not\ x, not\ x)$. When we compute $g\ (not\ True)$, the equation $not\ True = False$ only appears once in the EDT.

In the algorithmic debugging scheme, one needs to answer several questions according to the EDT and intended semantics in order to locate a faulty node.

Notations: $M \simeq_I N$ means that M is equal to N with respect to the semantics of the programmer's intention. If the evaluation $M = N$ of a node in an EDT is in the programmer's intended semantics, then $M \simeq_I N$. Otherwise, $M \not\simeq_I N$ *i.e.* the node is erroneous.

General semantical equality rules:

$$\frac{}{M \simeq_I M} \qquad \frac{M \simeq_I N}{N \simeq_I M} \qquad \frac{M \simeq_I N \quad M' \simeq_I N'}{MM' \simeq_I NN'} \qquad \frac{M \simeq_I N \quad N \simeq_I R}{M \simeq_I R}$$

Figure 1. Semantical equality rules

Semantical equality rules are given in Figure 1, which will be used in Lemma 27 later.

As mentioned in Section 2, if a node in an EDT is erroneous but has no erroneous children, then this node is called *a faulty node.* The following figure shows what a faulty node looks like, where $n_1, n_2, ..., n_k$ are the children of m.

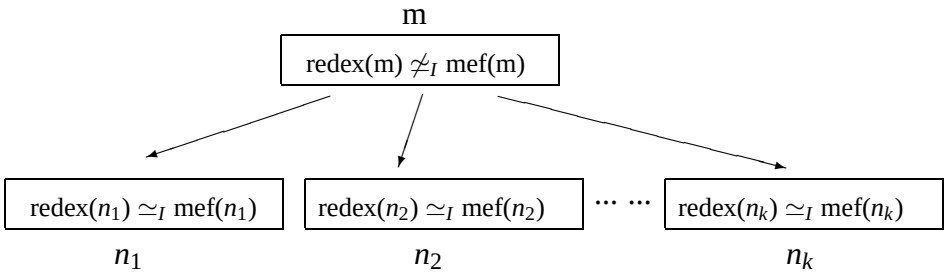


Figure 2. m is a faulty node

Definition 21. *Suppose the equation $fp_1...p_n = R$ is in a program P. If there exists a substitution σ such that $(fp_1...p_n)\sigma \equiv fb_1...b_n$ and $R\sigma \equiv N$, then we say that $fb_1...b_n \rightarrow_P N$.*

If $fb_1...b_n \rightarrow_P N$ but $fb_1...b_n \not\simeq_I N$, then we say that the definition of the function f in the program is faulty.

$fb_1...b_n \rightarrow_P N$ means that it is a single step computation from $fb_1...b_n$ to N according to one of the rewriting rules in the program P, and there is no computation in $b_1, ..., b_n$.

CORRECTNESS OF ALGORITHMIC DEBUGGING

Definition 22. *If the following statement is true, then we say that algorithmic debugging is correct.*

- *If the equation of a faulty node is $fb_1...b_n = M$, then the definition of the function f in the program is faulty.*

For a faulty node m, we have $redex(m) \not\simeq_I mef(m)$. We shall find a term N and prove $redex(m) \rightarrow_P N \simeq_I mef(m)$. In order to define N, we need other definitions.

Definition 23. *Let G be an ART and m a node in G. $reduct(m)$ is defined as follows.*

$$reduct(m) = \begin{cases} a & \text{if } (m,a) \in G \text{ and } a \text{ is an atom} \\ mef(n) & \text{if } (m,n) \in G \text{ and } n \text{ is a node} \\ reduct(m\mathsf{f})\ reduct(m\mathsf{a}) & \text{if } (m, m\mathsf{f} \circ m\mathsf{a}) \in G \\ reduct(m\mathsf{f})\ mef(j) & \text{if } (m, m\mathsf{f} \circ j) \in G \text{ and } j \neq m\mathsf{a} \\ mef(i)\ reduct(m\mathsf{a}) & \text{if } (m, i \circ m\mathsf{a}) \in G \text{ and } i \neq m\mathsf{f} \\ mef(i)\ mef(j) & \text{if } (m, i \circ j) \in G \text{ and } i \neq m\mathsf{f} \text{ and } j \neq m\mathsf{a} \end{cases}$$

reduct represents the result of a single-step computation. And we shall prove $redex(m) \rightarrow_P reduct(m\mathsf{r}) \simeq_I mef(m)$ for a faulty node m. Note that $mef(m) = mef(m\mathsf{r})$ and so we want to prove $reduct(m\mathsf{r}) \simeq_I mef(m\mathsf{r})$. In order to prove this, we prove a more general result $reduct(m) \simeq_I mef(m)$ for all $m \in dom(G)$ (see Lemma 27 for the conditions).

We define *branch* and the reduction principle $depth$ in order to prove this general result.

Definition 24. ***(**branch **and** branch′**)*** *We say that n is a branch node of m, denoted as $branch(n,m)$, if one of the following holds.*

- $branch(m,m)$*;*
- $branch(n\mathsf{f},m)$ *if* $branch(n,m)$*;*
- $branch(n\mathsf{a},m)$ *if* $branch(n,m)$.

Let G be an ART.

$$branch'(m) = \{n \mid n\mathsf{r} \in dom(G) \text{ and } branch(n,m)\}$$

Note that $branch'(m)$ is the set of all evaluated branch nodes of m.

Lemma 25. *Let G be an ART.*

- *If $n \in branch'(m\mathsf{f})$ or $n \in branch'(m\mathsf{a})$ then $n \in branch'(m)$.*
- *If $m\mathsf{r} \in dom(G)$ then $children(m) = branch'(m\mathsf{r})$.*

Proof. By the definitions of *children* and $branch'$.

Definition 26. ***(depth)*** *Let m be a node in an ART G.*

$$depth(m) = \begin{cases} 1 + max\{depth(m\mathsf{f}), depth(m\mathsf{a})\} & \text{if } (m, m\mathsf{f} \circ m\mathsf{a}) \in G \\ 1 + depth(m\mathsf{f}) & \text{if } (m, m\mathsf{f} \circ j) \in G \text{ and } j \neq m\mathsf{a} \\ 1 + depth(m\mathsf{a}) & \text{if } (m, i \circ m\mathsf{a}) \in G \text{ and } i \neq m\mathsf{f} \\ 1 & \text{if } (m, i \circ j) \in G \text{ and } i \neq m\mathsf{f} \text{ and } j \neq m\mathsf{a} \\ 0 & \text{otherwise} \end{cases}$$

Lemma 27. *Let G be an ART and m a node in G. If $redex(n) \simeq_I mef(n)$ for all $n \in branch'(m)$, then $reduct(m) \simeq_I mef(m)$.*

Proof. By induction on $depth(m)$.

When $depth(m) = 0$, we have $(m, e) \in G$ where e is a node or an atom.

- If e is a node, then $m\mathsf{r} \in G$ by Lemma 6. Then by the definitions of *reduct* and *mef*, we have $reduct(m) = mef(e)$ and $mef(m) = meft(m) = mef(e)$.
- If e is an atom, we have $reduct(m) = e$. Now, we consider the following two cases. If $m \in branch'(m)$, then we have $m\mathsf{r} \in dom(G)$ and $mef(m) \simeq_I redex(m) = e$. If $m \notin branch'(m)$, then we have $m\mathsf{r} \notin dom(G)$ and $mef(m) = meft(m) = e$.

For the step cases, we proceed as follows.

- If $m \in branch'(m)$, then we have $m\mathsf{r} \in dom(G)$ and $redex(m) \simeq_I mef(m)$. And we need to prove $redex(m) \simeq_I reduct(m)$.
  Let us consider only one case here. The other cases are similar. Suppose $(m, m\mathsf{f} \circ j) \in G$ and $j \neq m\mathsf{a}$, then by the definitions we have

$$\begin{aligned} redex(m) &= mef(m\mathsf{f})\ mef(j) \\ reduct(m) &= reduct(m\mathsf{f})\ mef(j) \end{aligned}$$

  Since for any $n \in branch'(m\mathsf{f})$, by Lemma 25, we have $n \in branch'(m)$ and hence $redex(n) \simeq_I mef(n)$. By the definition of *depth*, we also have $depth(m\mathsf{f}) < depth(m)$. Now, by induction hypothesis, we have $reduct(m\mathsf{f}) \simeq_I mef(m\mathsf{f})$. And hence we have $redex(m) \simeq_I reduct(m)$ by the semantical equality rules in Figure 1.
- If $m \notin branch'(m)$, then $m\mathsf{r} \notin dom(G)$.
  Let us also consider only one case. The other cases are similar. Suppose $(m, m\mathsf{f} \circ j) \in G$ and $j \neq m\mathsf{a}$, then by the definitions we have

$$\begin{aligned} mef(m) &= mef(m\mathsf{f})\ mef(j) \\ reduct(m) &= reduct(m\mathsf{f})\ mef(j) \end{aligned}$$

  The same arguments as above suffice.

Corollary 28. *Let G be an ART and $m\mathsf{r}$ a node in G (*i.e. *$m\mathsf{r} \in dom(G)$). If $redex(n) \simeq_I mef(n)$ for all $n \in children(m)$, then $reduct(m\mathsf{r}) \simeq_I mef(m)$.*

Proof. By Lemma 25 and 27.

The condition, $redex(n) \simeq_I mef(n)$ for all $n \in children(m)$, basically means that m does not have any erroneous child nodes as in Figure 2.

Lemma 29. *Let G be an ART and $m\mathsf{r}$ a node in G (*i.e. $m\mathsf{r} \in dom(G)$*). Then $redex(m) \rightarrow_P reduct(m\mathsf{r})$.*

Proof. Since there is a computation at the node m, we suppose G at node m matches the left-hand side of the rewriting rule $fp_1...p_n = R$ with $[m_1/x_1, ..., m_k/x_k]$. We need to prove that there exists a substitution σ such that $redex(m) = (fp_1...p_n)\sigma$ and $reduct(mt) = R\sigma$. In fact $\sigma = [mef(m_1)/x_1, ..., mef(m_k)/x_k]$.

Now, we need to prove that $redex(m) = (fp_1...p_n)\sigma$ and $reduct(mt) = R\sigma$. For the first, we proceed by the definition of *redex* and pattern matching. For the second, we proceed by the definition of *reduct* and $graph$.

A similar result as in the above lemma is proved in [2].

Now, we come to the most important theorem, the correctness of algorithmic debugging.

Theorem 30. ***(Correctness of Algorithmic Debugging)*** *Let G be an ART, T its EDT and m a faulty node in T. If the equation for the faulty node m is $fb_1...b_n = M$, then the definition of f in the program is faulty.*

Proof. By Lemma 29 and Corollary 28, we have $redex(m) \rightarrow_P reduct(m\mathsf{r})$ and $reduct(m\mathsf{r}) \simeq_I mef(m)$. Since $fb_1...b_n \equiv redex(m) \not\simeq_I mef(m) \equiv M$, we have $fb_1...b_n \rightarrow_P reduct(m\mathsf{r})$ and $fb_1...b_n \not\simeq_I reduct(m\mathsf{r})$. The computation from $fb_1...b_n$ to $reduct(m\mathsf{r})$ is a single step computation, but $fb_1...b_n$ is not semantically equal to $reduct(m\mathsf{r})$. So the definition of f in the program must be faulty.

6 CONCLUSION AND FUTURE WORK

In this paper, we have formally presented the ART and EDT. The ART is an efficient and practical trace, and it is a model of a real implementation (i.e. Hat). The EDT is directly generated from the ART. We proved the most important property of Hat, the correctness of algorithmic debugging. What the theorem proves is the consistency between the answers given be the user and the detection of the faulty node made by the debugging algorithm. Many other related properties of the ART and EDT are also proved.

However, there is still more work that needs to be done. Currently we are studying three extensions of the ART model, and the resulting EDT.

1. Replace the unevaluated parts in an ART by underscore symbols (*i.e.* _). An unevaluated part in an ART intuitively means the value of this part is irrelevant to any reduction in the graph.

2. Add error messages to an ART when there is a pattern matching failure.

3. Add local rewriting rules (or definitions) to the program.

How these three extensions will affect the EDT and algorithmic debugging needs further study.

ACKNOWLEDGEMENTS

The work reported in this paper was supported by the Engineering and Physical Sciences Research Council of the United Kingdom under the grant EP/C516605/1.

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. To appear in ENTCS 2006.

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

[4] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

[5] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[6] 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.

[7] B. Pope and Lee Naish. Practical aspects of declarative debugging in Haskell-98. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240, 2003.

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

[9] Jan Sparud and Hendrik Nilsson. The architecture of a debugger for lazy functional languages. In Mireille Ducassé, editor, *Proceedings of AADEBUG'95*, Saint-Malo, France, May, 1995.

[10] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS Vol. 1292, September 1997.

[11] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, UU-CS-2001-23. Universiteit Utrecht, 2001. Final proceedings to appear in ENTCS 59(2).