Proving the Correctness of Algorithmic Debugging for Functional Programs

Olaf Chitil and Yong Luo

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

Abstract

This paper formally presents a 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 more 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. Tracing is the process of obtaining additional information about the internal workings of a computation.

Traditional debugging techniques are not well suited for declarative programming languages such as Haskell, because it is diffi cult to determine the evaluation order. In fact, functional programmers want to ignore low-level operational details, in particular the evaluation order, but take advantage of properties such as explicit data flow and absence of side effects. Algorithmic debugging (also called declarative debugging) is proposed and adapted in logic and functional programming languages [14, 9, 13].

Several tracing systems for lazy functional languages are available, all for Haskell [15, 9, 5, 17, 18, 13]. All systems take a two-phase approach to tracing:

1. During the computation information about the computation is recorded in a data structure, the trace.

2. After termination of the computation the trace is used to view the computation. Usually an interactive tool displays fragments of the computation on demand. The programmer uses their knowledge of the intended behaviour of the program to locate faults.

Each tracing method gives a different view of a computation; in practice, the views are complementary and can productively be used together [4]. Hence the Haskell tracer Hat integrates several methods [17]. During a computation a single unified trace is generated, the *augmented redex trail (ART)*. Separate tools provide different views of the ART, for example algorithmic debugging [14, 9, 13], following redex trails [16] and observing functions [5].

Although several tracing systems and methods have been implemented for functional programs, there is a lack of theoretical foundation for tracing. In this paper, our aim is to 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)* for algorithmic debugging will also be generated from a computation graph. We can correctly locate program faults, and the correctness will be formally proved. It has not been proved for logic or functional programs. This is a non-trivial proof and the difficulty is to find a suitable induction principle and a more general induction hypothesis.

In the next section we give an brief overview of the ART and EDT. Related work are also discussed. In Section 3, some basic definitions and the ART are formally presented. In Section 4, we show how to generate an EDT for algorithmic debugging from an ART. In Section 5, we prove the properties of an EDT, in particular, the correctness of algorithmic debugging. Some future work will be discussed in the last section.

2 OVERVIEW OF ART AND EDT

Term graph rewriting [12] provides an operational semantics for functional programs that is abstract and closely related to standard term rewriting semantics. In contrast to terms/expressions, graphs allow the sharing of common subexpressions as it happens in real implementations of functional languages, such as the G-machine [6] or the more efficient STG-machine [11]. Term graph rewriting can correctly model the asymptotic time and space complexity of real implementations [1]. For us sharing is the key for a space efficient trace structure and closeness to the implementation also promises easy creation of a trace.

The augmented redex trail (ART) is a compact but detailed representation of the computation; in particular, it directly relates each redex with its reduct. The ART does not overwrite a redex with its reduct, but adds the reduct into the graph. Although there may be some sharing nodes but the existing graph will never be modified. A detailed example can be found in [3]. In this paper the ART has no information about the order of computation because this information is irrelevant. We formulate and prove properties without reference to any reduction strategy. This observation agrees with our idea that functional programmers abstract from time.

We concentrate on the ART because it was already distilled as a unified trace from several other traces. This focus on the ART does not preclude revisions of its definition in the light of new insights. We are aware of several shortcomings (lack of information) that we intend to remove. Although the ART is only used for Haskell, it is suitable for both strict and non-strict pure functional languages.

An evaluation dependency tree (EDT), as described in [10, 7], is for users to determine if a node is erroneous. And algorithmic debugging can be thought of as searching an EDT for a fault in a program. The user answers whether the equations in an EDT are correct. If a node in an EDT is erroneous but has no erroneous children, then this node is called *a faulty node [7]*. We shall concentrate on faulty nodes in the paper.

The idea of EDT for algorithmic debugging has been implemented for several logic or functional languages. Some theoretical foundations for algorithmic debugging have been studied [2]. However, one of the key properties of EDT, the correctness of algorithmic debugging, has not been formally proved.

Related Work

In [16], the idea of *redex trail* is developed and the computation builds its own trial as reduction proceeds. In [17], *Hat*, a tracer for Haskell 98, is introduced. The trace in Hat is recorded in a file rather than in memory. Hat integrates several viewing methods such as Functional Observations, Reduction Trails and Algorithmic debugging.

In [7], Naish presents a very abstract and general scheme for algorithmic debugging. The scheme represents a computation as a tree and relies on a way of determining the correctness of a subcomputation represented by a subtree. In Nilsson's thesis [8], a basis for algorithmic debugging of lazy functional programs is developed in the form of EDT which hides operational details. The EDT is constructed efficiently in the context of implementation based on graph reduction. In [2], Callaero et al formalise both the declarative and the operation semantics of programs in a simple language which combines the expressivity of pure Prolog and a significant subset of Haskell, and provide firm theoretical foundations for the algorithmic debugging of wrong answers in lazy functional logic programming.

3 FORMALISING AN ART

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. ***(Terms and Patterns)***

- A variable or constructor is a term.
- MN is a term if M and N are terms.
- A variable is a pattern.

- $cp_1...p_n$ is a pattern if c is a constructor and p_1,..., p_n are patterns, and the arity of c is n.

Definition 2. ***(Rewriting rule)*** *A rewriting rule is of the form*

$$f\ p_1...p_n = t$$

where p_1,..., p_n ($n > 0$) are patterns and t is a term.

Example 3. $id\ x = x$ and $not\ True = False$ are rewriting rules.

We only allow disjoint patterns if there are more than one rewriting rules for a function. We also require that the number of the arguments of a function in the left hand side must be the same. For example, if there is a computation 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 defi ne a function. It 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 fi rst pattern, the algorithm will not try to match other patterns. In this paper, we only consider disjoint patterns.

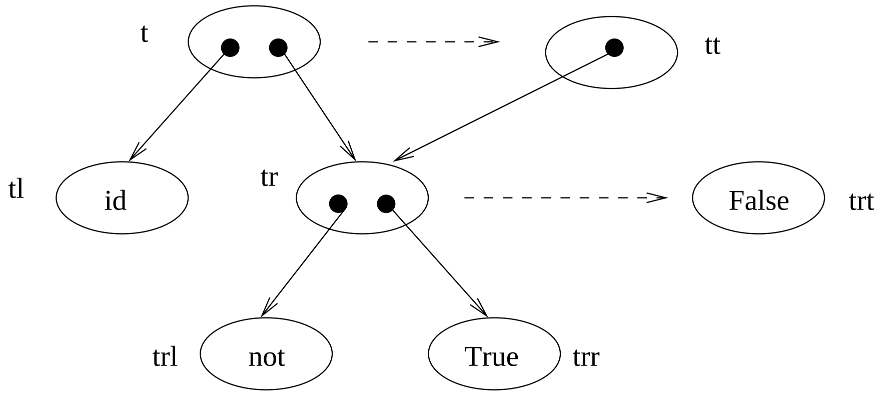
Definition 4. ***(Node, Node expression and Computation graph)***

- *A* ***node*** *is a sequence of letters t, l and r,* i.e. $\{t,l,r\}^*$.
- *A* ***node expression*** *is either a variable, or a constructor, 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 5. The following is a computation graph for the term $id\ (not\ True)$.

$$\{(t,tl \circ tr),(tl,id),(tr,trl \circ trr),(trl,not),(trr,True),\\ (tt,tr),(trt,False)\}$$

And it represents the following graph.



The letters l and r mean the left and right hand side of an application respectively. The letter t means a small step of computation. The computation steps are omitted in a graph because if a node mt in a graph then there is a computation from the node m to mt. For example, (t,tt) and (tr,trt) are not included in the above graph.

Notation: $dom(G)$ 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". First, we give some notational defi nitions.

- The fi nal node in a sequence of reduction starting at node m, $last(G,m)$.

$$last(G,m) = \begin{cases} last(G,mt) & \text{if } mt \in dom(G) \\ m & \text{otherwise} \end{cases}$$

  The purpose of this function is to fi nd out the most evaluated point for m. For example, if G is the graph in Example 5, then we have $last(G,t) = tt$ and $last(G,tr) = trt$.

- The head of the term at node m, $head(G,m)$, where G is a graph and m is a node in G.

$$head(G,m) = \begin{cases} head(G, last(G,i)) & \text{if } (m, i \circ j) \in G \\ f & \text{if } (m, f) \in G \end{cases}$$

  For example, if G is the graph in Example 5, then we have $head(G,t) = id$ and $head(G,tr) = not$.

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

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

  Note that the arguments of a function are a sequence of nodes. For example, if G is the graph in Example 5, then we have $args(G,t) = \langle tr\rangle$ and $args(G,tr) = \langle trr\rangle$.

Now, we defi ne two functions $match_1$ and $match_2$ mutually. 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.

- $match_1(G,m,x) = [m/x]$

- For $match_1(G,m,cq_1...q_k)$, let $m' = last(G,m)$ and $arguments = args(G,m')$, if $head(G,m') = c$ then

$$match_1(G,m,cq_1...q_k) = match_2(G, arguments, \langle q_1,...,q_k\rangle)$$

  otherwise m does not match $cq_1...q_k$.

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

  However, if any m_i does not match p_i, $\langle m_1, ..., m_n\rangle$ does not match $\langle p_1, ..., p_n\rangle$.

- We say that G at node m matches $f\,p_1...p_n = N$ with $[m_1/x_1, ..., m_k/x_k]$ if $head(G,m) = f$ and

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

In the substitution form $[m/x]$, m is not a term but a node. In Example 5, the graph at node t matches $id\ x = x$ with $[tr/x]$. The definition of pattern matching and its result substitution sequence will become important for making computation order irrelevant when we generate graphs. In Example 5, no matter which node is reduced first, t or tr, the final graph will be the same.

Building an ART

Graph for substituted expressions. When a term is substituted by a sequence of shared nodes, it becomes substituted expression. The function *graph* defined in the following has two arguments: a node and a substituted expressions. The result of *graph* is a computation graph.

$$\begin{aligned} graph(n,x) &= \{(n,x)\} && \text{where } x \text{ is a variable} \\ graph(n,c) &= \{(n,c)\} && \text{where } c \text{ is a constructor} \\ graph(n,k) &= \{(n,k)\} && \text{where } k \text{ is a node} \\ graph(n,MN) &= \begin{cases} \{(n, M \circ N)\} & \text{if } M \text{ and } N \text{ are nodes} \\ \{(n, M \circ nr)\} \cup graph(nr, N) & \text{if only } M \text{ is a node} \\ \{(n, nl \circ N)\} \cup graph(nl, M) & \text{if only } N \text{ is a node} \\ \{(n, nl \circ nr)\} \cup graph(nl, M) & \text{otherwise} \\ \quad \cup graph(nr, N) & \end{cases} \end{aligned}$$

- For a start term M, the start ART is $graph(t, M)$. Note that the start term has no nodes inside.

- ***(ART rule)*** If an ART G at m matches $f\,p_1...p_n = N$ with $[m_1/x_1, ..., m_k/x_k]$, then we can build a new ART

  $$G \cup graph(mt, N[m_1/x_1, ..., m_k/x_k])$$

An ART is generated from a start ART and by applying the *ART rule* repeatedly.

Example 6. If the start term is $id\ (not\ True)$, then the start graph is

$$\{(t, tl \circ tr), (tl, id), (tr, trl \circ trr), (trl, not), (trr, True)\}$$

The new parts built from t and tr are

$$\begin{aligned}graph(tt, x[tr/x]) = graph(tt, tr) = \{(tt, tr)\}\\ graph(trt, False) = \{(trt, False)\}\end{aligned}$$

Note that the order of computation is irrelevant.

One may also notice that there is no parent's edges in an ART. They need not be given explicitly because the way that the nodes are labelled give us the parents of all nodes implicitly, and a function of parent will be defi ned in the next section.

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

Lemma 7. *Let G be an ART.*

If $m \in dom(G)$, then there is at least one letter t in m. And if $mt \in dom(G)$ then $m = \varepsilon$ or $m \in dom(G)$.

4 GENERATING AN EVALUATION DEPENDENCY TREE

In this section we generate the *Evaluation Dependency Tree* (EDT) for algorithmic debugging from a give ART.

Definition 8. ***(Evaluation dependency tree for algorithmic debugging)*** *Let G be an ART. The evaluation dependency tree of G for algorithmic debugging is $tree(\varepsilon)$, where ε is the empty sequence, and $tree$ and some notations are defined as follows.*

- *parent*

$$\begin{array}{rcl} parent(nl) & = & parent(n) \\ parent(nr) & = & parent(n) \\ parent(nt) & = & n \end{array}$$

- *children*

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

- *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 5, then the EDT for algorithmic debugging is $tree(\varepsilon) = \{(\varepsilon, t), (\varepsilon, tr)\}$.

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

$$mef(m) = \begin{cases} mef(mt) & \text{if } mt \in dom(G) \\ meft(e) & \text{otherwise and } (m, e) \in G \end{cases}$$

where

$$\begin{array}{lcll} \mathit{meft}(x) & = & x & \textit{where x is a variable} \\ \mathit{meft}(c) & = & c & \textit{where c is a contructor} \\ \mathit{meft}(n) & = & \mathit{mef}(n) & \textit{where n is a node} \\ \mathit{meft}(i \circ j) & = & \mathit{mef}(i)\ \mathit{mef}(j) & \end{array}$$

Example 11. If G is the graph in Example 5, then

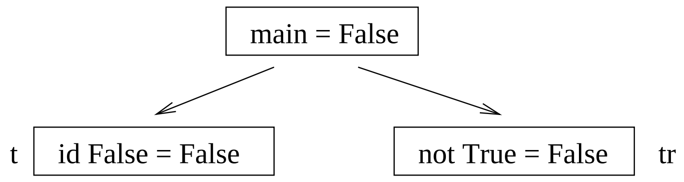
$$\mathit{mef}(\varepsilon) = \mathit{mef}(t) = \mathit{mef}(tt) = \mathit{meft}(tr) = \mathit{mef}(tr) = \mathit{False}$$

Definition 12. ***(Equations for an evaluation dependency tree)*** *Let G be an ART, and m a node in its EDT . There is a corresponding equation* $\mathit{redex}(m) = \mathit{mef}(m)$*, where redex is defined as follows.*

- $\mathit{redex}(\varepsilon) = \mathit{main}$
- $\mathit{redex}(m) = \mathit{mef}(i)\ \mathit{mef}(j)$ *if* $(m, i \circ j) \in G$

If m is a node of an EDT , then $m \equiv \varepsilon$ or $(m, i \circ j)$ is in the graph for some i and j. So, we only consider these two cases for the defi nition of *redex*. $\mathit{redex}(m) = \mathit{mef}(m)$ 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 $\mathit{redex}(m) = \mathit{mef}(m)$ depends on the evaluation $\mathit{redex}(n) = \mathit{mef}(n)$.

Example 13. The EDT for the graph in Example 5 is the following.



5 PROPERTIES OF AN EDT

In this section, we briefly present the properties and the correctness of algorithmic debugging. Because of the limitation of space, some proof details are omitted.

The following theorems suggest that the EDT of an ART covers all the computation in the graph. If there is any reduction that is unexpected in the ART, then this reduction will affect the evaluation equation in the EDT, and one must be able to fi nd the bugs in the program. Although two evaluations may depend on the same evaluation in an ART, every evaluation for algorithmic debugging only needs to be examined once.

Lemma 14. *Let G be an ART, and T its EDT for algorithmic debugging. If there is a sequence of nodes* $m_1, m_2, \ldots, m_k$ *such that*

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

then $m \in \mathit{dom}(T)$.

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

Lemma 15. *Let G be an ART. If $mt \in dom(G)$, then $m \equiv \varepsilon$ or there is a sequence of index numbers $m_1, m_2, ..., m_k$ such that*

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

Proof. By induction on the size of m, and by Lemma 7.

Since $mt \in dom(G)$, by Lemma 7, we only need to consider the following two cases.

- If $m = \varepsilon$, the statement is obviously true.
- If $m \in dom(G)$, by Lemma 7, there is at least one letter t in m. We consider the following two sub-cases.
    - $m = tn$, where there is no t in n, in other words, t is the first and the last t in m. Since $mt \in dom(G)$ and $parent(tn) = \varepsilon$, we have $tn \in children(\varepsilon)$.
    - $m \equiv m_1tn$, where there is no t in n, in other words, t is the last t in m. Since $mt \in dom(G)$ 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

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

Notation: $dom(T)$ denotes the set of nodes in an evaluation dependency tree T.

Theorem 16. *Let G be an ART, and T its EDT for algorithmic debugging. If $mt \in dom(G)$, then $m \in dom(T)$. In other word, T covers all the computations in G.*

Proof. By Lemma 15 and 14.

Lemma 17. *Let G be an ART, and T its EDT for algorithmic debugging. If $(m,n) \in T$, then $n \in children(m)$ and $parent(n) \equiv m$.*

Proof. By the definition of $tree$.

Theorem 18. *Let G be an ART, and T its EDT for algorithmic debugging. If $(m,n) \in T$ and $m \not\equiv k$, then $(k,n) \notin T$.*

Proof. By Lemma 17.

Basic rule:

$$\frac{M \rightarrow_P N}{M \simeq_P N}$$

General semantical equality rules:

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

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

Figure 1. Semantical equality rules

CORRECTNESS OF ALGORITHMIC DEBUGGING

Definition 19. *If the following statement is true we say the algorithmic debugging is correct.*

- *If the equation of a faulty node is $fa_1...a_n = R$, then the definition of the function f in the program is faulty.*

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.*

In order to prove the correctness, we need some defi nitions fi rst.

Definition 20. *Suppose the equation $fp_1...p_n = N$ is in a program. If there exists a substitution σ such that $(fp_1...p_n)\sigma \equiv fa_1...a_n$ and $N\sigma \equiv R$, then we say $fa_1...a_n \rightarrow_P R$.*

Notations: $M \simeq_P N$ means M is equal to N with respect to the semantics of the program. $M \simeq_I N$ means 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.

Semantical equality rules are given in Figure 1.

If $fa_1...a_n \rightarrow_P R$ but $fa_1...a_n \not\simeq_I R$, then the defi nition of the function f in the program has a bug, because from $fa_1...a_n$ to R is a single step computation and there is no computation in $a_1,...,a_n$. If the equation of a node m is $fa_1...a_n = M$, then $redex(m) = fa_1...a_n$ and $mef(m) = M$ by the defi nition of equations for EDT.

So, for a faulty node m, we shall find a term R and prove $redex(m) \rightarrow_P R \simeq_I mef(m)$. In order to define R, we need other definitions.

Definition 21. *Let G be an ART, m a node in G. $reduct(m)$is defined as follows.*

$$reduct(m) = \begin{cases} mef(e) & \text{if } (m,e) \in G \text{ and } e \text{ is a node or} \\ & \quad \text{constructor or variable} \\ reduct(ml)\ reduct(mr) & \text{if } (m, ml \circ mr) \in G \\ reduct(ml)\ mef(j) & \text{if } (m, ml \circ j) \in G \text{ and } j \neq mr \\ mef(i)\ reduct(mr) & \text{if } (m, i \circ mr) \in G \text{ and } i \neq ml \\ mef(i)\ mef(j) & \text{if } (m, i \circ j) \in G \text{ and } i \neq ml \text{ and } j \neq mr \end{cases}$$

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

We define $children'$ and the reduction principle $depth$ in order to prove the more general result.

Definition 22. ***(Branch and** $children'$**)** We say that n is a branch of m, denoted as $branch(n,m)$, if one of the following holds.*

- $branch(m,m)$;
- $branch(nl,m)$ *if* $branch(n,m)$;
- $branch(nr,m)$ *if* $branch(n,m)$.
- $children'(m) = \{n \mid nt \in dom(G)$ and $branch(n,m)\}$

Lemma 23. *Let G be an ART.*

- *If $n \in children'(ml)$ or $n \in children'(mr)$ then $n \in children'(m)$.*
- *If $mt \in dom(G)$ then $children(m) = children'(mt)$.*

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

Definition 24. ***(depth)** Let m be a node in an ART G.*

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

Lemma 25. *Let G be an ART and m a node in G, i.e. $m \in dom(G)$. If $redex(n) \simeq_I mef(n)$ for all $n \in children'(m)$, then $reduct(m) \simeq_I mef(m)$.*

Proof. By induction on $depth(m)$. When $depth(m) = 0$, by definition we have $reduct(m) \simeq_I mef(m)$. For the step cases, we proceed as follows.

- If $m \in children'(m)$, then we have $mt \in dom(G)$ 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 $(m, ml \circ j) \in G$ and $j \neq mr$, then by the definitions we have

$$\begin{array}{rcl} redex(m) & = & mef(ml)\ mef(j) \\ reduct(m) & = & reduct(ml)\ mef(j) \end{array}$$

  Since for any $n \in children'(ml)$, by Lemma 23, we have $n \in children'(m)$ and hence $redex(n) \simeq_I mef(n)$. By the definition of $depth$, we also have $depth(ml) < depth(m)$. Now, by induction hypothesis, we have $reduct(ml) \simeq_I mef(ml)$. And hence we have $redex(m) \simeq_I reduct(m)$.

- If $m \notin children'(m)$, then $mt \notin dom(G)$.
  Let us also consider only one case. The other cases are similar. Suppose $(m, ml \circ j) \in G$ and $j \neq mr$, then by the definitions we have

$$\begin{array}{rcl} mef(m) & = & mef(ml)\ mef(j) \\ reduct(m) & = & reduct(ml)\ mef(j) \end{array}$$

  The same arguments as above suffice.

Corollary 26. *Let G be an ART and mt a node in G i.e. $mt \in dom(G)$. If $redex(n) \simeq_I mef(n)$ for all $n \in children(m)$, then $reduct(mt) \simeq_I mef(m)$.*

Proof. By Lemma 23 and 25.

Lemma 27. *Let G be an ART and mt a node in G i.e. $mt \in dom(G)$. Then, $redex(m) \rightarrow_P reduct(mt)$.*

Proof. We need to prove the existence of a substitution σ.

In fact, if G at node m matches the rewriting rule $f\, p_1 ... p_n = N$ with $[m_1/x_1, ..., m_k/x_k]$, then $\sigma = [mef(m_1)/x_1, ..., mef(m_k)/x_k]$.

Now, we need to prove that $redex(m) = (f\, p_1 ... p_n)\sigma$ and $reduct(mt) = N\sigma$. 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*.

Theorem 28. *Let G be an ART, T its EDT and m a faulty node in T. If G at m matches a rewriting rule $f\, p_1 ... p_n = N$ in the program, then this rewriting rule is faulty.*

Proof. By Lemma 27 and Corollary 26.

6 FUTURE WORK

In this paper, we have formally presented the ART and EDT, and proved the correctness of algorithmic debugging. However, there is still more work that needs to be done.

Currently we are studying four extensions of the ART model, and the resulting EDT for algorithmic debugging.

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. Consider different reduction strategies and add error messages to an ART when there is a pattern matching failure.

3. Add local rewriting rules to the program.

4. Add rewriting rules for constants to the program, for example,
   $ones = 1 : ones$.
   The ART in the paper has no cycles, but some tracer such as Hat has cycles for constant rewriting. We intend to represent constant rewriting not by cycles but by creating a new node of the constant for every reduction.

How these four 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 grant EP/C516605/1.

References

[1] Adam Bakewell. Using term-graph rewriting models to analyse relative space efficiency. In *TERMGRAPH 2002 International Workshop on Term Graph Rewriting*, volume 72 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[2] 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.

[3] Olaf Chitil and Yong Luo. Towards a theory of tracing for functional programs based on graph rewriting. In Ian Mackie, editor, *Draft Proceedings of the 3rd International Workshop on Term Graph Rewriting, Termgraph 2006*, page 10, April 2006.

[4] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, LNCS 2011, pages 176–193. Springer, 2001.

[5] Andy Gill. Debugging Haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. 2000 ACM SIGPLAN Haskell Workshop.

[6] Thomas Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69. ACM Press, 1984.

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

[8] Henrik Nilsson. A declarative approach to debugging for lazy functional languages. Licentiate Thesis No. 450, Department of Computer and Information Science, Linköping University, S-581 83, Linköping, Sweden, September 1994.

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

[10] 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.

[11] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[12] Detlef Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, chapter 1, pages 3–61. World Scientific, 1999. Volume 2: Applications, Languages and Tools.

[13] 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.

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

[15] Jan Sparud and Colin Runciman. Complete and partial redex trails of functional computations. In C. Clack, K. Hammond, and T. Davie, editors, *Selected papers from 9th Intl. Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS Vol. 1467, September 1997.

[16] 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.

[17] 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).

[18] Malcolm Wallace, Olaf Chitil, and Colin Runciman. Hat: transforming lazy functional programs for multiple-view tracing. In preparation, 2004.