Replacing Unevaluated Parts in the Traces of Functional Programs

Yong Luo and Olaf Chitil

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

Abstract In functional programming languages such as Haskell, it happens often that some parts of a program are not evaluated because their values are not demanded. In practice, those unevaluated parts are often replaced by a placeholder (e.g. _) in order to keep the trace size smaller. For algorithmic debugging, this also makes the questions shorter and clearer. In this paper, we present a formal model of tracing in which unevaluated parts are replaced by the symbol _. Some properties such as the correctness of algorithmic debugging will also be proved.

1 Introduction

Tracing for functional programs based on graph rewriting is a process to record the information about computation. The trace can be viewed in various ways. The most common need for tracing is debugging. 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). 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 developed in logic and functional programming languages [10,8,9].

Several tracing systems for lazy functional languages are available, all for Haskell [8,4,13,9,12]. 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 [3]. Hence the Haskell tracer Hat integrates several methods [12]. 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 [10,8,9], following redex trails [11] and observing functions [4].

A direct and simple mode of tracing for functional programs is presented in [2,5]. The augmented redex trail (ART) is formally defined and its properties are proved. The ART is independent of any particular evaluation order and low-level operational details are ignored. In [5], the evaluation dependency tree (EDT) for algorithmic debugging is formally generated from the ART. Some important properties such as the correctness of algorithmic debugging are also proved.

In functional programming languages such as Haskell, it happens often that some parts of a program are not evaluated because their values are not demanded. For example, in the evaluation $fst\ (a, b) = a$, the term b may not be evaluated. In practice, those unevaluated parts are often replaced by a placeholder (e.g. _) in order to keep the trace size smaller. For algorithmic debugging, this also makes the questions shorter and clearer. In this paper, we use the formal model of tracing and the definition of the ART and EDT in [2,5]. The unevaluated parts of the ART are replaced by the _s if they satisfy certain conditions. These conditions are formally presented and the reasons for the conditions are explained. The most important property, the correctness of algorithmic debugging, will also be proved.

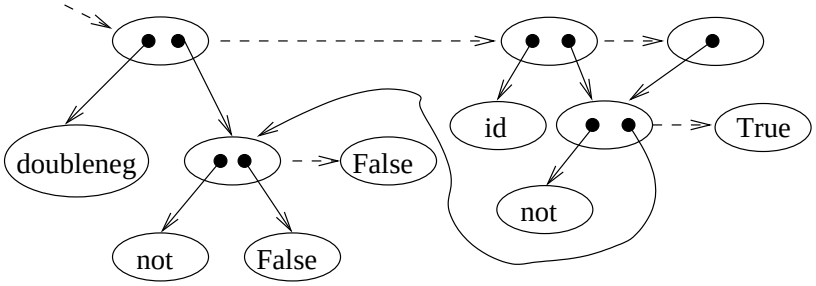
2 Overview of ART and EDT

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. The existing graph

will never be modified. A detailed example can be found in [2]. 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. For example, the double negation function is mistakingly defined as

$$doubleneg\ x = id\ (not\ x)$$

(the right hand side should be $not\ (not\ x)$). The ART for a starting term $doubleneg\ (not\ True)$ is the following.

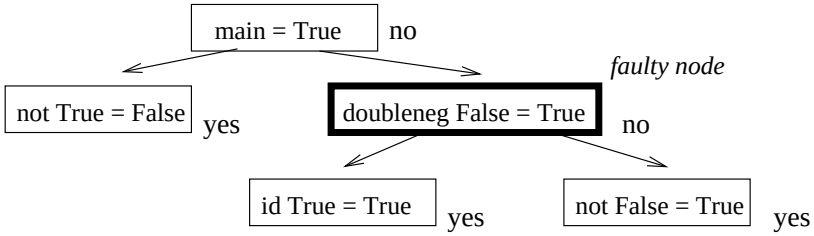


An evaluation dependency tree (EDT), as described in [6], is for users to determine if a node is erroneous. Algorithmic debugging can be thought of as searching an EDT for a fault in a program. The user answers whether the equations in an EDT are correct. If a node in an EDT is erroneous but has no erroneous children, then this node is called *a faulty node.*

In algorithmic debugging scheme, one need to answer several questions according to the EDT and intended semantics in order to find a faulty node. For example, the questions and answers for a starting term $doubleneg\ (not\ True)$ are as follows.

$main = True$	no
$not\ True = False$	yes
$doubleneg\ False = True$	no
$id\ True = True$	yes
$not\ False = True$	yes

And then we locate a faulty node.



Related Work

In [11], the idea of *redex trail* is developed and the computation builds its own trial as reduction proceeds. In [12], *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 [6], 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 [7], 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 [1], Caballero et al formalise both the declarative and the operational semantics of programs in a simple language which combines the expressiveness 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. However, the starting point in [1] is an operational semantics (*i.e.* a goal solving calculus) that is high-level and far from a real efficient implementation. For example, there is no sharing of replicated terms. In contrast we use the ART as base, which is a model of trace used in the Hat system. In [2], important properties of the ART have also been proved. In [5], the EDT is directly generated from the ART, and some important properties such as the correctness of algorithmic debugging are formally proved.

3 Formalising an ART and EDT

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 and Patterns)***

- **Atoms** consist of function symbols and constructors.
- **Terms:**
  - an atom is a term;
  - a variable is a term;
  - MN is a term if M and N are terms.
- **Patterns:**
  - 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*

$$f\ p_1...p_n = N$$

where f is a function symbol and p_1,..., p_n ($n \geq 0$) are patterns and N is a term.

Example 1. $id\ x = x$, $not\ True = False$ and $ones = 1 : ones$ 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 define 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 first pattern, the algorithm will not try to match other patterns. In this paper, we only consider disjoint 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.

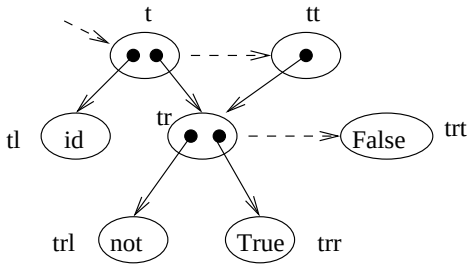
Definition 3. ***(Node, Node expression and Computation graph)***

- *A* ***node*** *is a sequence of letters t, l and r,* i.e. $\{t, l, r\}^*$.
- *A* ***node expression*** *is either 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 2. 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 is 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.

3.1 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 reduction starting at node m, $last(G, m)$.

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

The purpose of this function is to find out the most evaluated point for m. For example, if G is the graph in Example 2, then we have $last(G, t) = trt$ 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.

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

For example, if G is the graph in Example 2, 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 2, then we have $args(G, t) = \langle tr \rangle$ and $args(G, tr) = \langle trr \rangle$.

Now, we define 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.

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

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

- $$match_2(G, \langle m_1, ..., m_n \rangle, \langle p_1, ..., p_n \rangle) = match_1(G, m_1, p_1) \cup ... \cup match_1(G, m_n, p_n)$$

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

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

And 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, $\langle m_1, ..., m_r \rangle$ does not match $\langle p_1, ..., p_s \rangle$ if $r \neq s$.

- We say that G at node m matches a rewriting rule $fp_1...p_n = N$ 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 2, 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 2, no matter which node is reduced first, t or tr, the final graph will be the same.

3.2 Building an ART

Graph for substituted expressions. When a term is substituted by a sequence of shared nodes, it becomes a 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.

$$graph(n, e) = \{(n, e)\} \qquad \text{where } e \text{ is an atom or a node}$$

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

Note that all the variables in a term will be substituted by some nodes during the computations.

- 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 $fp_1...p_n = N$ 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 3. 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

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

Note that the order of computation is irrelevant because the result of pattern matching at the node t is always $[tr/x]$, 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.

3.3 Generating an EDT

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

Definition 4. ***(Parent edges)***

$$parent(nl) = parent(n)$$
$$parent(nr) = parent(n)$$
$$parent(nt) = n$$

Note that $parent(t) = \varepsilon$ where ε is the empty sequence.

Definition 5. *(children* ***and*** *tree) Let G be an ART, and mt a node in G.*

children and tree are defined as follows.

- *children*

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

- *tree*

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

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

Example 4. If G is the graph in Example 2, $tree(G, \varepsilon) = \{(\varepsilon, t), (\varepsilon, tr)\}$.

Definition 6. ***(Most Evaluated Form)*** *Let G be an ART. The most evaluated form of a node m is a term and is defined as follows.*

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

where

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

Example 5. If G is the graph in Example 2, then

$$mef(G,t) = mef(G,tt) = meft(G,tt) = mef(G,tr) = False$$

Definition 7. ***(redex)*** *Let G be an ART, and mt a node in G. redex is defined as follows.*

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

Notice that the case $(m,n) \in G$ is not defined in this definition because $mt \in dom(G)$ implies $(m,n) \notin G$ for any node n.

Definition 8. ***(Evaluation Dependency Tree for algorithm debugging)*** *Let G be an ART. The evaluation dependency tree (EDT) of G for algorithm debugging consists of the following two parts.*

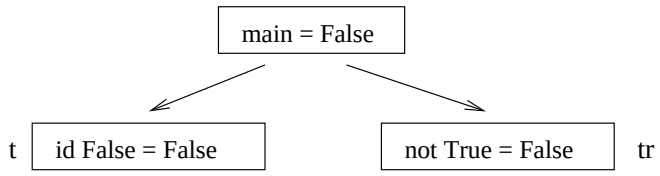
1. *The set $tree(G,\varepsilon)$;*
2. *The set of equations, $\{redex(G,m) = mef(G,m) \mid m \in tree(G,\varepsilon)\}$.*

Note that we write $mef(G,\varepsilon)$ for $mef(G,t)$.

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

$redex(G,m) = mef(G,m)$ 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 $redex(G,m) = mef(G,m)$ depends on the evaluation $redex(G,n) = mef(G,n)$.

Example 6. The EDT for the graph in Example 2 is the following.



4 Replacing the unevaluated parts by _s.

In this section, we present the conditions of replacing the unevaluated parts by _s, and give examples to explain these conditions.

Conditions

If $m \in dom(G)$ satisfies the following conditions it can be replaced by _.

1. $mt \notin dom(G)$; and
2. $head(G, m)$ is a function; and
3. $(i, n \circ j) \notin G$ for any i and j, where $m = nt^*$ and the last letter of n is not t.

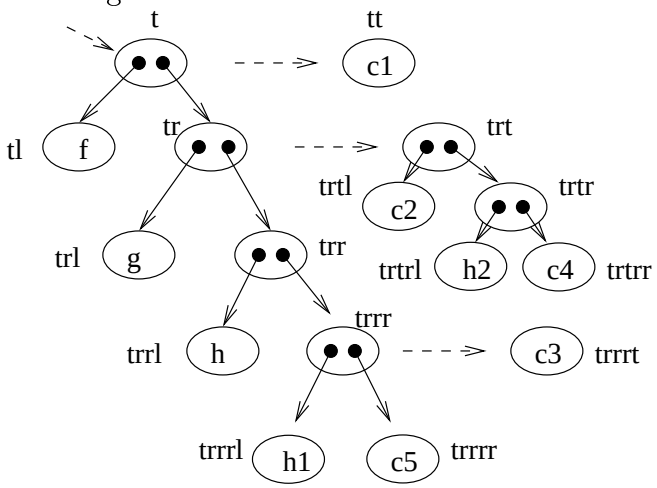
If the above conditions are satisfied for a node $m \in dom(G)$, we shall remove some parts (or pairs) from the original ART.

1. For any pair $(n, e) \in G$, if $n = m\{l, r, t\}^+$ then the pair (n, e) will be removed from the original ART.
2. For the pair $(m, e) \in G$, it will be replaced by $(m, -)$.

We give two examples to illustrate these conditions. More explanation of these conditions will be given later.

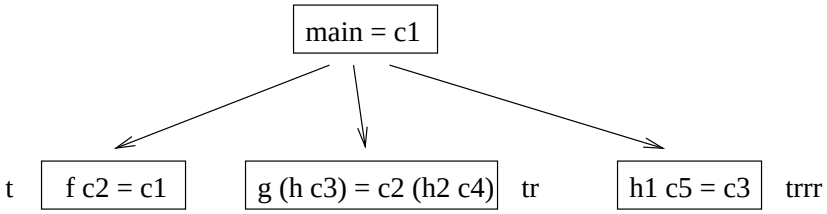
Example 7. We give some ARTs and EDTs. The programs are omitted.

1. The original ART:

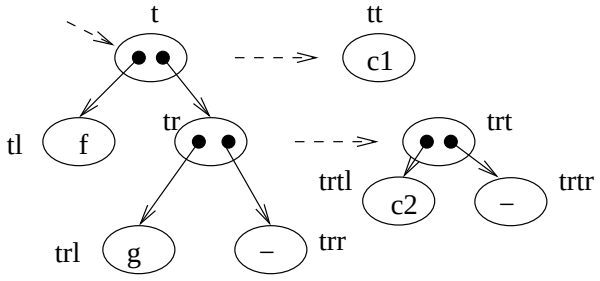


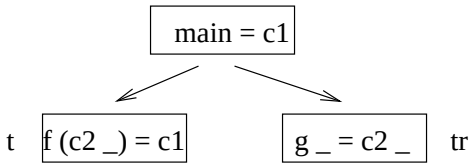
Notice that ARTs have no information about evaluation order and are independent from any particular evaluation strategy. The computation at $trrr$ may happen according to the definition of ART.

And the original EDT:

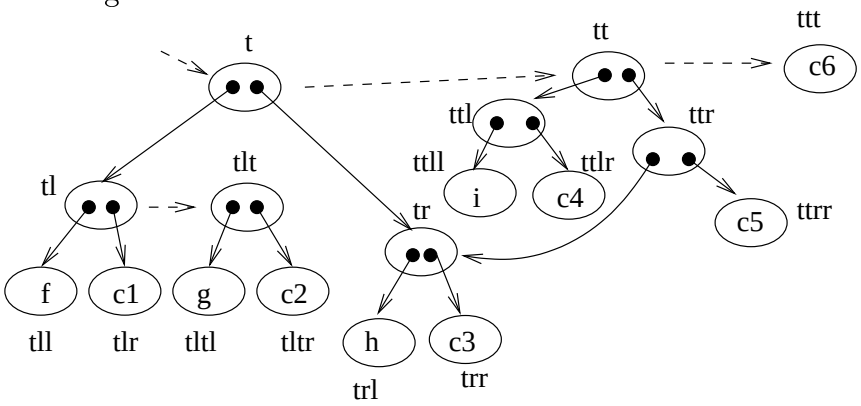


Since the nodes trr and $trtr$ satisfy the three conditions, the pair $(trr, trrl \circ trrr)$ and $(trtr, trtrl \circ trtrr)$ are replaced by $(trr, -)$ and $(trtr, -)$ respectively, and other pairs such as $(trrl, h)$ are removed from the original ART. Then we have a new ART and EDT as follows.

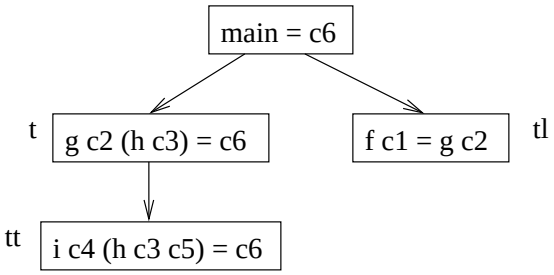




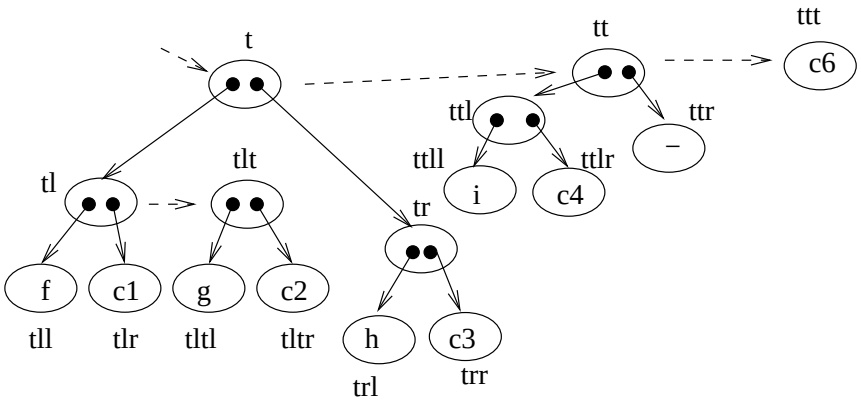
2. The original ART:



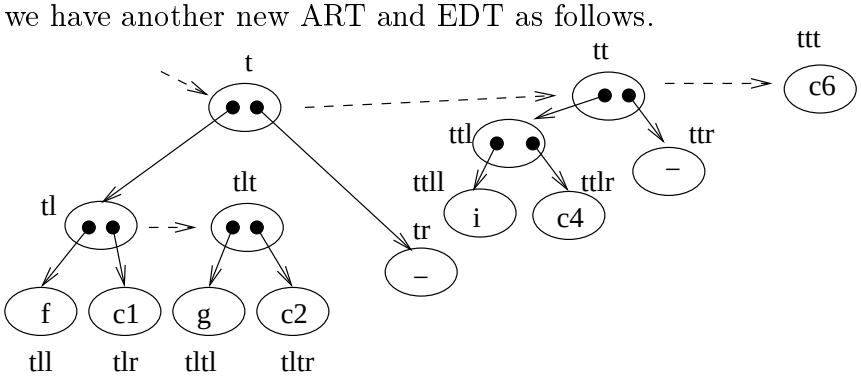
The original EDT:

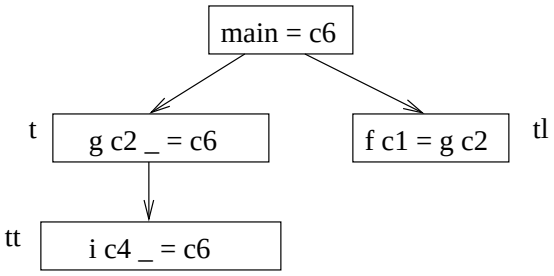


Since the node ttr satisfies the three conditions, the pair $(ttr, tr \circ ttrr)$ is replaced by $(ttr, -)$, and $(ttrr, c2)$ is removed from the original ART. Then we have a new ART as follows.



Now, the node tr satisfies the three conditions although it did not before because $(ttr, tr \circ ttrr)$ was in the original ART. Then we have another new ART and EDT as follows.





Now, we explain why these three conditions must be satisfied for a node m before we replace it by _.

1. $mt \notin G$. This means that there is no computation at m. We do not intend to remove any evaluated parts. However, m may be removed because other node satisfies all the three conditions.

For example, the node $trrr$ in Example 7(1) is removed although $trrrt \in G$.

2. $head(G, m)$ is a function. This means that the value at m is not demanded. If $head(G, m)$ is not a function, *i.e.* it is a constructor, then the value (or the weak head normal form) at m may be demanded. For example, the node trt in Example 7(1) should not be replaced because the head c_2 may be used for pattern-matching.
3. $(i, n \circ j) \notin G$ for any i and j, where $m = nt^*$ and the last letter of n is not t. This means that m cannot be the left-hand-side of any application. If the left-hand-side of an application is replaced by _, important information about computation may be lost. For example, the nodes ttl and tlt in Example 7(2) should not be replaced.

There is another thing which we may take into our consideration. After some replacements, it is possible that $(mt...t, -) \in G$ for some $m \in dom(G)$ where the last letter of m is not t. In this case, we can replace m by _ and remove all the intermediate reduction steps. Notice that this kind of replacement will not change any questions (or equations) for algorithmic debugging but will only remove some trivial questions of the form $M = -$ which are always true. We don't consider this kind of replacement in the paper.

An ART is a computation graph. After some replacement of unevaluated parts, some pares of the form $(m, -)$ is in the ART. We expand the definition 3 by allowing _ to be a node expression. We also expand the definition of *meft*. If $(m, -) \in G$, then $meft(G, m) = -$.

Notation: We say G is an ART_ if it is get from an original ART G_0 by replacing some unevaluated parts.

Lemma 1. *Let G be an ART_.*

- If $(m, -) \in G$ then $mt \notin dom(G)$.
- If $(m, -) \notin G$ then $head(G, m) \neq -$.

Proof. By the three conditions.

Lemma 2. *Let G_0 be an original ART, G its ART_ and m a node in G.*

- $m \in dom(G_0)$.
- *If* $mt \in dom(G_0)$ *then* $mt \in dom(G)$.
- *If* $mt \in dom(G_0)$ *and* $(m, e) \in G_0$ *then* $(m, e) \in G$.
- *If* $mt \in dom(G_0)$ *then* $head(G_0, m) = head(G, m)$.

Proof. By the three conditions.

Theorem 1. *Let* G_0 *be an ART,* G *its ART_ and* m *a node in* G. *If* $mt \in dom(G_0)$ *and* G_0 *at node* m *matches a rewriting rule* $fp_1...p_n = N$ *with* $[m_1/x_1, ..., m_k/x_k]$, *then* G *at node* m *matches the rewriting rule* $fp_1...p_n = N$ *with* $[m_1/x_1, ..., m_k/x_k]$.

5 Correctness of Algorithmic Debugging

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

Notations: $M \simeq_I N$ 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, $M \not\simeq_I N$.

Semantical equality rules are given in Figure 1, which will be used in Lemma 6 later.

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

When there are _s in an equation in an EDT, for example, $g\ -=\ c2\ -$ as in Example 7(1), if this is the programmer's intention, then it means that $\forall x \exists y.(g\ x\ \simeq_I\ c2\ y)$. If it is not the programmer's intention, then it means that $\exists x \forall y.(g\ x\ \not\simeq_I\ c2\ y)$. In general, for any equation $M = N$ in an EDT, we replace _s by fresh variables then the equation becomes $M' = N'$, and suppose $\{x_1, ..., x_n\}$ is the

set of variables in M' and $\{y_1, ..., y_m\}$ is in N'. If $M = N$ is the programmer's intention, it means that $\forall\overline{x}\exists\overline{y}.(M' \simeq_I N')$. If $M = N$ is not the programmer's intention, it means that $\exists\overline{x}\forall\overline{y}.(M' \not\simeq_I N')$. If there is no _ in M and N, $M \overrightarrow{\simeq} N$ means $M \simeq_I N$ and $M \overrightarrow{\not\simeq} N$ means $M \not\simeq_I N$.

Notations: Let M and N be terms with _s. Replace _s by fresh variables then the equation becomes $M' = N'$, and suppose $\{x_1, ..., x_n\}$ is the set of variables in M' and $\{y_1, ..., y_m\}$ is in N'. $M \overrightarrow{\simeq} N$ denotes $\forall\overline{x}\exists\overline{y}.(M' \simeq_I N')$ or $M \equiv N$. $M \overrightarrow{\not\simeq} N$ denotes $\exists\overline{x}\forall\overline{y}.(M' \not\simeq_I N')$. If there is no _ in M and N, $M \overrightarrow{\simeq} N$ means $M \simeq_I N$ and $M \overrightarrow{\not\simeq} N$ means $M \not\simeq_I N$.

Lemma 3. *We have the following lemmas.*

- $- \overrightarrow{\simeq} -$.
- $M \overrightarrow{\simeq} -$.
- *If M' is got from M by replacing some parts by _s, then $M \overrightarrow{\simeq} M'$.*

Proof. $\forall x \exists y.x \simeq_I y$ is true.

Lemma 4. *We also have the following lemmas by the semantical equality rules in Figure 1.*

- *If $M \overrightarrow{\simeq} N$ and $N \overrightarrow{\simeq} R$ then $M \overrightarrow{\simeq} R$.*
- *If $M \overrightarrow{\simeq} N$ and $M' \overrightarrow{\simeq} N'$ then $MM' \overrightarrow{\simeq} NN'$.*
- *If $M_1 \overrightarrow{\simeq} N_1, ..., M_k \overrightarrow{\simeq} N_k$ then $R[M_1/x_1, ..., M_k/x_k] \overrightarrow{\simeq} R[N_1/x_1, ..., N_k/x_k]$.*

Theorem 2. *If $M \overrightarrow{\not\simeq} N$ and $R \overrightarrow{\simeq} N$, and there is no _ in M and R, then $M \not\simeq_I R$.*

Proof. If $M \simeq_I R$, then we will have a contradiction.

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 figure 2 shows what a faulty node looks like, where $n_1, n_2, ..., n_k$ are the children of m.

Definition 9. ***(Correctness of Algorithmic Debugging)*** *If the following statement is true, then we say that the algorithmic debugging is correct.*

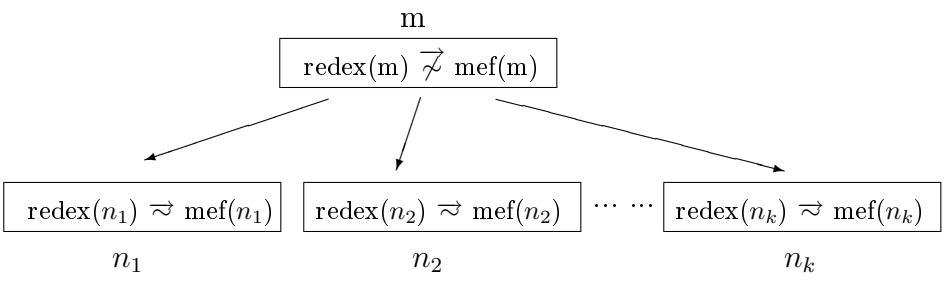


Figure 2. m is a faulty node

- 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.

Definition 10. *Suppose the equation $fp_1...p_n = N$ is in a program. If there exists a substitution σ such that $(fp_1...p_n)\sigma \equiv fb_1...b_n$ and $N\sigma \equiv R$, then we say that $fb_1...b_n \rightarrow_P R$.*

If $fb_1...b_n \rightarrow_P R$ but $fb_1...b_n \not\rightarrow_I R$, then the definition of the function f in the program is faulty, because from $fb_1...b_n$ to R is a single step computation and there is no computation in $b_1, ..., b_n$.

In order to prove the correctness, we need some definitions first.

Definition 11. *(branch **and** children') We say that n is a branch node of m, denoted as $branch(n, m)$, if one of the following holds.*

- *$branch(m, m)$;*
- *$branch(nl, m)$ if $branch(n, m)$;*
- *$branch(nr, m)$ if $branch(n, m)$.*

- Let G be an ART_.
  $children'(G, m) = \{n \mid nt \in dom(G) \text{ and } branch(n, m)\}$

Lemma 5. *Let G be an ART_.*

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

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

Definition 12. *Let G be an ART_ and m a node in G. $reduct(G, m)$ is defined as follows.*

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

Definition 13. ***(depth)*** *Let m be a node in an $ART\_$ G.*

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

Lemma 6. *Let G be an $ART\_$ and m a node in G.*
If $redex(G,n) \overline{\approx} mef(G,n)$ for all $n \in children'(G,m)$, then $reduct(G,m) \overline{\approx} mef(G,m)$.

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

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

- If e is $\_$, we have $mt \notin dom(G)$ by Lemma 1. Then $reduct(G,m) = -$ and $mef(G,m) = meft(G,m) = -$.
- If e is a node, then $mt \notin G$. Then by the definitions of $reduct$ and mef, we have $reduct(G,m) = mef(G,e)$ and $mef(G,m) = meft(G,m) = mef(G,e)$.
- If e is an atom, we have $reduct(G,m) = e$. Now, we consider the following two cases. If $m \in children'(m)$, then we have $mt \in dom(G)$ and $redex(G,m) = e$ and $redex(G,m) \overline{\approx} mef(G,m)$. If $m \notin children'(m)$, then we have $mt \notin dom(G)$ and $mef(G,m) = meft(G,m) = e$.

For the step cases, we proceed as follows.

- If $m \in children'(G,m)$, then we have $mt \in dom(G)$ and $redex(G,m) \overline{\approx} mef(G,m)$.
  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

$$redex(G,m) = mef(G,ml)\ mef(G,j)$$
$$reduct(G,m) = reduct(G,ml)\ mef(G,j)$$

Since for any $n \in children'(G,ml)$, by Lemma 5, we have $n \in children'(G,m)$ and hence $redex(G,n) \overline{\approx} mef(G,n)$. By the definition of $depth$, we also have $depth(G,ml) < depth(G,m)$. Now, by induction hypothesis, we have $reduct(G,ml) \overline{\approx} mef(G,ml)$. And hence we have $reduct(G,m) \overline{\approx} mef(G,m)$ by Lemma 3.

- If $m \notin children'(G,m)$, 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

$$mef(G,m) = mef(G,ml)\ mef(G,j)$$
$$reduct(G,m) = reduct(G,ml)\ mef(G,j)$$

  The same arguments as above suffice.

Corollary 1. *Let G be an ART$_$ and mt a node in G. If $redex(n) \overline{\approx} mef(n)$ for all $n \in children(m)$, then $reduct(mt) \overline{\approx} mef(m)$.*

Proof. By Lemma 5 and 6.

A proof of the correctness of algorithmic debugging

If m is a faulty node in the EDT for G which is an ART$_$, we know $redex(G,m) \overrightarrow{\gamma} mef(G,m)$. Let us replace $_$s in $redex(G,m)$ and $mef(G,m)$ by fresh variables, and then $redex(G,m)$ becomes M and $mef(G,m)$ becomes N. Suppose $\overline{x}$ and $\overline{y}$ are variables in M and N respectively. Then we have $\exists \overline{x} \forall \overline{y}.(M \not\simeq_I N)$. Suppose there are terms $\overline{a}$ such that $\forall \overline{y}.(M[\overline{a}/\overline{x}] \not\simeq_I N)$. We write $redex(G,m)[\overline{a}/-]$ for $M[\overline{a}/\overline{x}]$. Notice that there is no $\_$ in $redex(G,m)[\overline{a}/-]$ and

$$redex(G,m)[\overline{a}/-] \overrightarrow{\gamma} mef(G,m)$$

Now, suppose G at m matches a rewriting rule $fp_1...p_n = R$ with $[m_1/x_1, ..., m_k/x_k]$. Then $redex(G,m)$ matches $fp_1...p_n$ with $[mef(G,m_1)/x_1, ..., mef(G,m_k)/x_k]$.

Since $redex(G,m)$ matches $fp_1...p_n$, we know that $redex(G,m)[\overline{a}/-]$ matches $fp_1...p_n$. Suppose $redex(G,m)[\overline{a}/-]$ matches $fp_1...p_n$ with $[M_1/x_1,...,M_k/x_k]$. Then we have $M_1 \overline{\approx} mef(G,m_1)$,..., $M_k \overline{\approx} mef(G,m_k)$, and

$$redex(G,m)[\overline{a}/-] \rightarrow_P R[M_1/x_1,...,M_k/x_k]$$

$$R[M_1/x_1,...,M_k/x_k] \overline{\approx} R[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k]$$

If no node under mt is replaced by _, formally speaking, $(mt\{l,r\}*,-) \notin G$, then we have

$$reduct(G,mt) = R[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k]$$

This result is proved in [2,5]. If some parts under mt are replaced by _s, then we can replace some parts of $R[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k]$ by _s and get $reduct(G,mt)$, and we have

$$R[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k] \overline{\approx} reduct(G,mt)$$

Now we have $R[M_1/x_1,...,M_k/x_k] \overline{\approx} reduct(G,mt)$ by Lemma 4.

Since m is a faulty node, we have $reduct(G,mt) \overline{\approx} mef(G,m)$ by Corollary 1. Now, by Lemma 4, we have

$$R[M_1/x_1,...,M_k/x_k] \overline{\approx} mef(G,m)$$

Since $redex(G,m)[\overline{a}/-] \overrightarrow{\not\sim} mef(G,m)$, and there is no _ in $redex(G,m)[\overline{a}/-]$ and $R[M_1/x_1,...,M_k/x_k]$, we have

$$redex(G,m)[\overline{a}/-] \not\approx_I R[M_1/x_1,...,M_k/x_k]$$

Since we have proved

$$redex(G,m)[\overline{a}/-] \rightarrow_P R[M_1/x_1,...,M_k/x_k]$$

$$redex(G,m)[\overline{a}/-] \not\approx_I R[M_1/x_1,...,M_k/x_k]$$

we know the rewriting rule $fp_1...p_n = R$ in the program is faulty.

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. Towards a theory of tracing for functional programs based on graph rewriting. In *http://www.cs.kent.ac.uk/people/staff/oc/traceTheory.html, submitted to the 3rd International Workshop on Term Graph Rewriting, Termgraph 2006.*
3. 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.
4. Andy Gill. Debugging Haskell by observing intermediate data structures. *Electronic Notes in Theoretical Computer Science*, 41(1), 2001. 2000 ACM SIGPLAN Haskell Workshop.
5. Yong Luo and Olaf Chitil. Proving the correctness of algorithmic debugging for functional programs. In *http://www.cs.kent.ac.uk/people/staff/yl41/TFPpaper.pdf, submitted to the seventh symposium on trends in functional programming, TFP 2006.*
6. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.
7. 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.
8. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.
9. 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.
10. E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.
11. 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.
12. 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).
13. Malcolm Wallace, Olaf Chitil, and Colin Runciman. Hat: transforming lazy functional programs for multiple-view tracing. In preparation, 2004.