Computer Science at Kent

Replacing Unevaluated Parts in the Traces of Functional Programs

Yong Luo and Olaf Chitil

Technical Report No. 7 - 07
August 2007

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

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 non-strict 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. In the process of algorithmic debugging, one needs to answer several questions in order to locate a program fault. Replacing unevaluated parts makes these questions shorter and semantically clearer. In this paper, we present a formal model of tracing in which unevaluated parts are replaced by the symbol _. The most important property, the correctness of algorithmic debugging, is 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) has been developed for logic and functional programming languages [11,8,10].

Several tracing systems for lazy functional languages are available, all for Haskell [8,4,15,10,14]. Each tracing method gives a different view of a computation; in practice, the views are complementary and can productively be used together [3]. A direct and simple model 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.

Problems and Motivation

In non-strict 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 $fst\ (a, large\_term) = a$, the term

large_term may be a very large term and unevaluated. It has little meaning to keep a large unevaluated term in a trace. In practice, those unevaluated parts are often replaced by a placeholder (e.g. _) in order to keep the trace size smaller. In the process of algorithmic debugging, one will have a smaller and clearer question:

$$fst\ (a, \_) = a \quad \text{yes or no?}$$

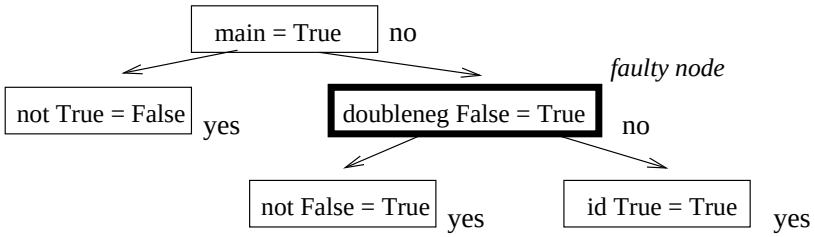
However, there still are two problems:

1. Intuitively, it is quite clear that unevaluated parts can be replaced by _s, and this idea has been implemented in Hat and Freja. However, it is not easy to formulate this intuition, in particular, when dealing with high-order functions, sharing and partial applications together. The examples in Section 4 will demonstrate the subtleties. In the paper, we shall formally present the conditions to decide whether a node in a trace should be replaced by _ or not. The reasons for these conditions will be explained as well.
2. There is lack of theoretical foundation. For example, the meaning of the questions for algorithmic debugging may be different after unevaluated parts are replaced by _s. How do we know the debugging scheme is still correct? 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 the conditions. The most important property, the correctness of algorithmic debugging, is proved. This is a non-trivial proof because the traces include some interesting features such as high-order functions, sharing and partial application. This paper is the first one to deal with all these features together in terms of replacing unevaluated parts in a trace.

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]. 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.

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.* 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 questions and answers are as follows. Then we locate a faulty node which is erroneous but has no erroneous children.



Related Work

In [13], the idea of *redex trail* is developed and the computation builds its own trail as reduction proceeds. In [14], *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 [7,12,9], 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 an 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 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 and generate an EDT. To make the paper self-contented, we copy many definitions and properties from [5], but some details and proofs are omitted. Readers who are familiar with the work may skip this section.

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

- ***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) $cp_1...p_n$ 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 $f\ p_1 ... p_n = R$ 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 1. $id\ x = x$, $not\ True = False$, $map\ f\ (x : xs) = f\ x\ :\ map\ f\ xs$ 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. 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 l and r mean the left-hand and right-hand side of an application respectively. The letter t means a small step of computation.

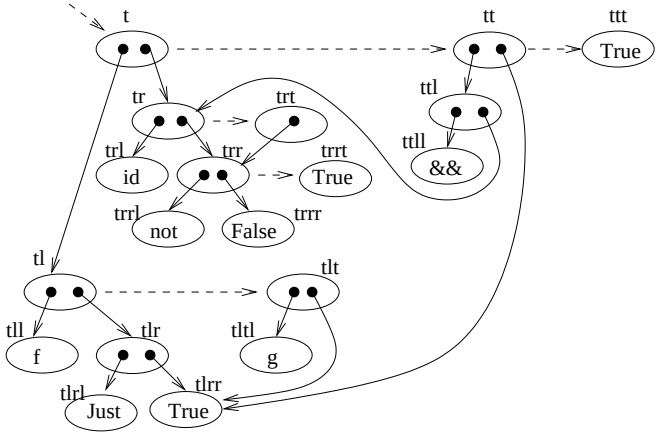
Definition 2. ***(Node, Node expression and Computation graph)***

- *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. We have a Haskell program, $f\ (Just\ x) = g\ x$ and $g\ x\ y = y\ \&\&\ x$. The following is a computation graph for the starting term $f\ (Just\ True)\ (id\ (not\ False))$.

$\{(t, tl \circ tr), (tl, tll \circ tlr), (tll, f), (tlr, tlrl \circ tlrr), (tlrl, Just), (tlrr, True),$
$(tr, trl \circ trr), (trl, id), (trr, trrl \circ trrr), (trrl, not), (trrr, False),$
$(trt, trr), (trrt, True), (tlt, tltl \circ tlrr), (tltl, g), (tt, ttl \circ tlrr), (ttl, ttll \circ tr),$
$(ttll, \&\&), (ttt, True)\}$

It can be depicted as follows. The dashed edges represent the computation steps. The pairs of the form (m, mt) are omitted in the formal representation of the graph. For example, (t, tt) and (tl, tlt) are not included in the graph.



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". We shall denote the set of nodes in a computation graph G by $dom(G)$.

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

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

  For example, if G is the graph in Example 2, then we have $last(G, t) = ttt$ and $last(G, tr) = trrt$.
- 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 $head(G, t) = g$ and $head(G, tl) = f$.
- The arguments of the function at node m, $args(G, m)$:

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

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

Now, we define two functions $match_1$ and $match_2$ which are mutually recursive.

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

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

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

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

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

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

  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.

- We say that G at node m matches the left-hand side of a rewriting rule $fp_1...p_n = R$ 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 2, the graph at node t matches $g\ x\ y$ with $[tlrr/x, tr/y]$.

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. The function *graph* defined in the following has two arguments: a node and a label term. The result of *graph* is a computation graph.

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

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

3.1 Building an ART

- 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 the left-hand side of a rewriting rule $fp_1...p_n = R$ with $[m_1/x_1, ..., m_k/x_k]$, then we can build a new ART

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

- 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 3. In Example 2, the new parts built from the nodes t and tr are

$$\begin{aligned} & graph(tt, (y\ \&\&\ x)[tlrr/x, tr/y]) \\ = & \ graph(tt, (tr\ \&\&\ tlrr)) \\ = & \ \{(tt, ttl \circ tlrr), (ttl, ttll \circ tr), (ttll, \&\&)\} \end{aligned}$$

$$graph(trt, x[trr/x]) = \{(trt, trr)\}$$

Note that the order of computation is irrelevant because the result of pattern matching at the node tr is always $[trr/x]$, no matter which node is computed first.

3.2 Generating an EDT

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 [13].

Definition 3. ***(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 4. *(children* ***and*** *tree) Let G be an ART, and mt a node in G (*i.e. *$mt \in dom(G)$).*

- *$children(G, m) = \{n \mid parent(n) = m$ and $nt \in dom(G)\}$. The condition $nt \in dom(G)$ is to make sure that only evaluated nodes become children.*
- *$tree(G, m) = \{(m, n_1), ..., (m, n_k)\} \cup tree(n_1) \cup ... \cup tree(n_k)$ where $\{n_1, ..., n_k\} = children(G, m)$.*

Example 4. If G is the graph in Example 2 then

$$tree(G, \varepsilon) = \{(\varepsilon, t), (\varepsilon, tr), (\varepsilon, trr), (\varepsilon, tl), (t, tt)\}$$

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 5. ***(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

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

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

Definition 6. ***(redex)*** *Let G be an ART, and mt a node in G (*i.e. *$mt \in dom(G)$). redex is defined as follows.*

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

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

$$mef(G, t) = mef(G, tt) = meft(G, ttt) = True$$

$$redex(G, t) = mef(G, tl)\ mef(G, tr) = g\ True\ True$$

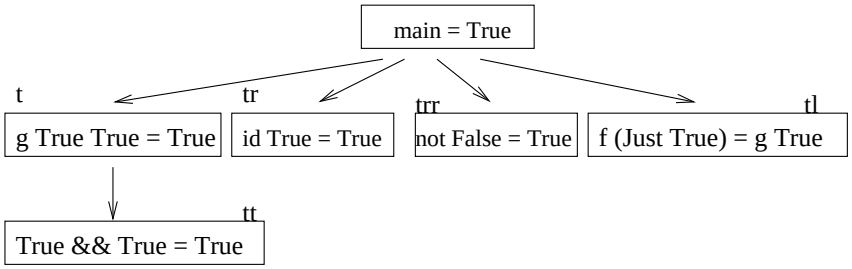
Now, we define the evaluation dependency tree of a graph.

Definition 7. ***(Evaluation Dependency Tree)*** *Let G be an ART. The evaluation dependency tree (EDT) of G consists of the following two parts.*

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

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

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 three conditions it can be replaced by _.

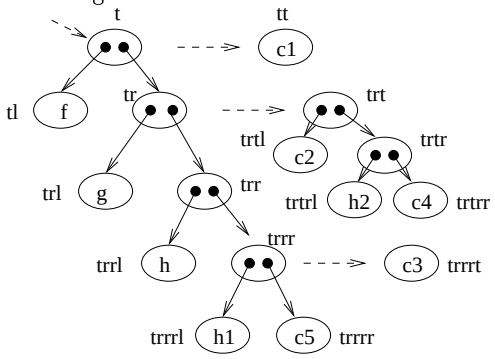
1. $mt \notin dom(G)$; and
2. $head(G, m)$ is a function; and
3. $(i, n \circ j) \notin G$ for any nodes i, n and j, where $last(G, n) = m$.

We give two examples to illustrate these conditions. More explanation of these conditions will be given after the examples. 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. The pair (m, e) will be replaced by $(m, \_)$.

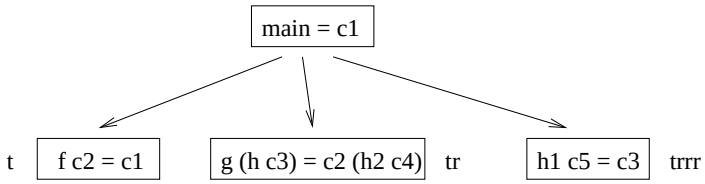
Example 7. We give some ARTs and EDTs. The programs are omitted.

1. The original ART:

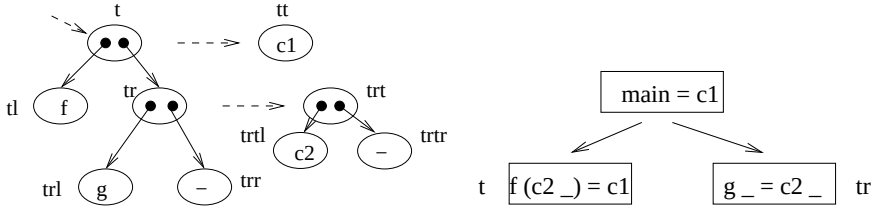


Notice that ARTs are non-deterministic about evaluation order and independent of any particular evaluation strategy. The computation at $trrr$ may happen according to the definition of ART although it may not happen in any lazy evaluation strategy.

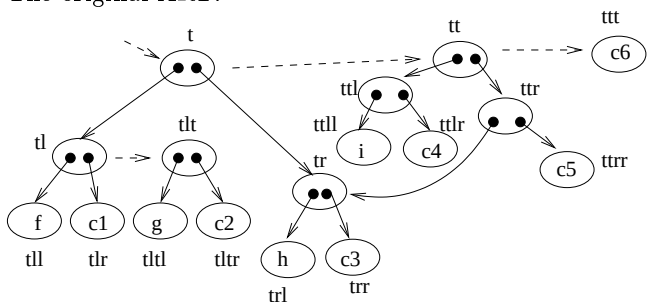
The original EDT is:



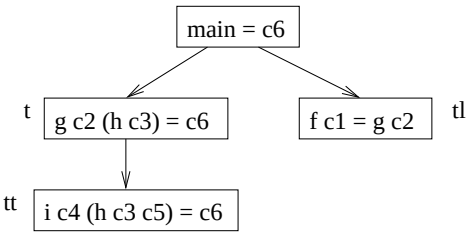
Since the nodes trr and $trtr$ satisfy the three conditions, the pair $(trr, trrl \circ trrr)$ and $(trtr, trtrl \circ trtrr)$ are replaced by $(trr, \_)$ and $(trtr, \_)$ respectively, and other pairs such as $(trrl, h)$ are removed from the original ART. Then we have a new ART and a new EDT as follows.



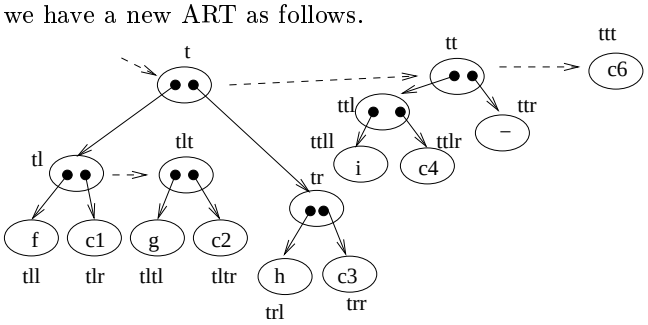
2. The original ART:



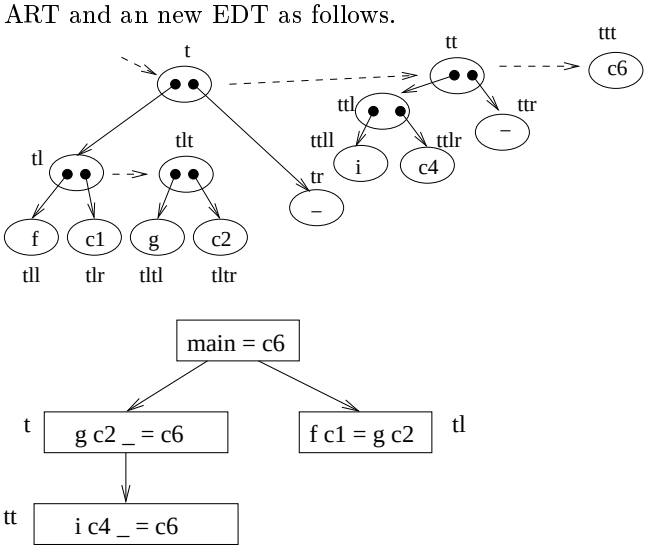
The original EDT:



Since the node ttr satisfies the three conditions, the pair $(ttr, tr \circ ttrr)$ is replaced by $(ttr, \_)$, and $(ttrr, c5)$ 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 an new 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 head of m must not be a constructor. 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 pattern-matching. Since we only consider the computation graphs (*i.e.* traces) without the information of programs, it is hard to decide whether a constructor is demanded by pattern-matching. Therefore, it is in general unsafe to replace a node whose head is a constructor. For example, the node *trt* in Example 7(1) should not be replaced because the head $c2$ may be used for pattern-matching.
3. $(i, n \circ j) \notin G$ for any nodes i, n and j, where $last(G, n) = m$. This means that m cannot be the left-hand side of any application. Otherwise, if the left-hand side of an application is replaced by _, important information about computation may be lost. This condition is particularly important when there are high-order functions and partial applications. For example, the nodes *ttl* and *tlt* in Example 7(2) should not be replaced although they satisfy the other two conditions.

We may take into consideration that after some replacements it is possible that $(last(G, m), \_) \in G$ for some $m \in dom(G)$. In this case we can replace m by _ and remove all the intermediate reduction steps. This kind of replacement will not change any questions for algorithmic debugging but will only remove some trivial questions of the form $M =$ _ which are always true. We do not consider this kind of replacement in the paper.

Notation: We say G is a ***Condensed ART*** if it is obtained from an original ART G_0 by replacing some unevaluated parts. There are some pairs of the form $(m, \_)$ in a condensed ART. From now on we shall regard _ as an ***atom***.

Lemma 1. *Let G_0 be an original ART, G a condensed ART of G_0 and m a node in G (*i.e. $m \in dom(G)$*).*

1. $m \in dom(G_0)$.
2. *If $head(G_0, m) = c$ and c is a constructor then $(m, \_) \notin G$. This means that any node whose head is a constructor cannot be replaced by _.*
3. *If $(m, i \circ j) \in G$ then $(last(G, i), \_) \notin G$. This means that any node that is the left-hand side of an application cannot be replaced by _.*
4. *If $(m, \_) \in G$ then $mt \notin dom(G)$ and $mef(G, m) =$ _.*
5. *If $mt \in dom(G_0)$ then $mt \in dom(G)$ and $(m, \_) \notin G$.*
6. *If $(m, n) \in G_0$ and n is a node then $(m, n) \in G$.*
7. $last(G_0, m) = last(G, m)$.
8. *If $(m, \_) \notin G$ and $(m, i \circ j) \in G_0$ then $(m, i \circ j) \in G$.*
9. *If $(m, \_) \notin G$, $(m, a) \in G_0$ and a is an atom then $(m, a) \in G$.*
10. *If $(m, \_) \notin G$ and $head(G_0, m) = a$ then $head(G, m) = a$.*
11. *If $head(G_0, m) = c$ and c is a constructor then $head(G, m) = c$.*
12. *If $(m, \_) \notin G$ then $args(G_0, m) = args(G, m)$.*

Proof. By the three conditions and the definitions of *mef*, *last*, *head* and *args*.

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, $M \not\simeq_I N$, *i.e.* the node is erroneous.

Semantical equality rules are given in Figure 1. They will be used in Lemma 5 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 $\exists \overline{x} \forall \overline{y}.(M' \not\simeq_I N')$.

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'. $M \sqsupseteq N$ denotes $\forall \overline{x} \exists \overline{y}.(M' \simeq_I N')$. $M \not\sqsupseteq N$ denotes $\exists \overline{x} \forall \overline{y}.(M' \not\simeq_I N')$. If there is no _ in M and N, $M \sqsupseteq N$ means $M \simeq_I N$ and $M \not\sqsupseteq N$ means $M \not\simeq_I N$.

Lemma 2. *We have the following lemmas.*

1. $\_ \sqsupseteq \_$, $M \sqsupseteq M$, *and* $M \sqsupseteq \_$ *for any* M.
2. *If* M' *is obtained from* M *by replacing some parts by _s, then* $M \sqsupseteq M'$.

Proof. For the first, $\forall x \exists y.x \simeq_I y$ is true. For other lemmas, similar arguments suffice.

Lemma 3. *We also have the following lemmas by the semantical equality rules in Figure 1.*

1. *If* $M \sqsupseteq N$ *and* $N \sqsupseteq R$ *then* $M \sqsupseteq R$.
2. *If* $M \sqsupseteq N$ *and* $M' \sqsupseteq N'$ *then* $MM' \sqsupseteq NN'$.
3. *If* $M_1 \sqsupseteq N_1$,..., $M_k \sqsupseteq N_k$ *then* $R[M_1/x_1, ..., M_k/x_k] \sqsupseteq R[N_1/x_1, ..., N_k/x_k]$.

Theorem 1. *If* $M \not\sqsupseteq N$ *and* $R \sqsupseteq N$, *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*. Figure 2 shows what a faulty node looks like, where $n_1, n_2, ..., n_k$ are the children of m.

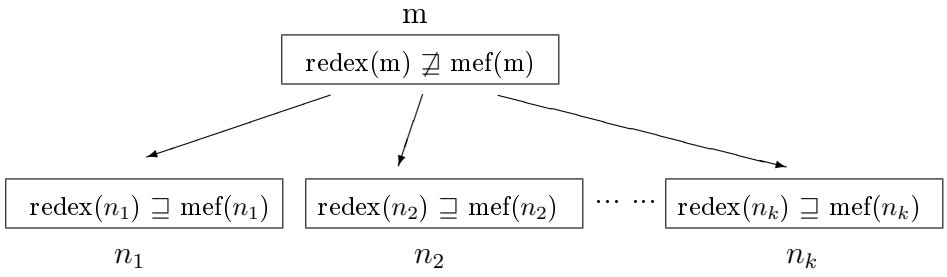


Figure 2. m is a faulty node

Definition 8. *Suppose the equation* $fp_1...p_n = R$ *is in a program. 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* $fb_1...b_n \rightarrow_P N$. *Note that there is no* _ *in* σ, $fb_1...b_n$ *and* 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 $b_1, ..., b_n$.

Definition 9. ***(Correctness of Algorithmic Debugging)*** *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.*

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

Definition 10. *(branch* ***and*** *branch')* *We say that* n *is a branch node of* m, *denoted as* $branch(n, m)$, *if one of the following holds.*

- $branch(m, m)$;

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

Let G be a condensed ART.

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

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

Lemma 4. *Let G be a condensed ART. If $mt \in dom(G)$ then $children(G, m) = branch'(G, mt)$.*

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

Definition 11. *Let G be a condensed ART and m a node in G. $reduct(G, m)$ is defined as follows.*

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

Definition 12. ***(depth)*** *Let m be a node in a condensed ART G.*

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

Lemma 5. *Let G be a condensed ART and m a node in G. If $redex(G, n) \sqsupseteq mef(G, n)$ for all $n \in branch'(G, m)$, then $reduct(G, m) \sqsupseteq mef(G, m)$.*

Proof. By induction on $depth(G, m)$. The structure of the proof is very similar to the one in [5], but the lemmas used in this proof are about the relation $\sqsupseteq$.

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

- If $e \equiv \_$, we have $mt \notin dom(G)$ by Lemma 1(5). Then $reduct(G, m) = \_$ and $mef(G, m) = \_$. Therefore, by Lemma 2(1), we have $reduct(G, m) \sqsupseteq mef(G, m)$.
- If e is an atom and $e \not\equiv \_$, we have $reduct(G, m) = e$. Now, we consider the following two cases.
  - If $m \in branch'(m)$, then we have $mt \in dom(G)$ and $redex(G, m) = e$ and $redex(G, m) \sqsupseteq mef(G, m)$. Note that $e \not\equiv \_$ in this case. Therefore, we have $reduct(G, m) \sqsupseteq mef(G, m)$.

· If $m \notin branch'(m)$, then we have $mt \notin dom(G)$ and $mef(G,m) = meft(G,m) = e$. Therefore, by Lemma 2(2), we have $reduct(G,m) \sqsupseteq mef(G,m)$.

- 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 $mef(G,m) = meft(G,m) = mef(G,e)$. Therefore, by Lemma 2(2), we have $reduct(G,m) \sqsupseteq mef(G,m)$.

For the step cases, we proceed as follows.

- If $m \in branch'(G,m)$, then we have $mt \in dom(G)$ and $redex(G,m) \sqsupseteq mef(G,m)$.
  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 $n \in branch'(G,ml)$, by Lemma 4, we have $n \in branch'(G,m)$ and hence $redex(G,n) \sqsupseteq mef(G,n)$. By the definition of *depth*, we also have $depth(G,ml) < depth(G,m)$. Now, by induction hypothesis, we have $reduct(G,ml) \sqsupseteq mef(G,ml)$. Therefore, by Lemma 3(2), we have $reduct(G,m) \sqsupseteq redex(G,m)$. And by Lemma 3(1), we have $reduct(G,m) \sqsupseteq mef(G,m)$.
- If $m \notin branch'(G,m)$, 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

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

  Similar arguments as above suffice.

Corollary 1. *Let G be a condensed ART and mt a node in G. If $redex(G,n) \sqsupseteq mef(G,n)$ for all $n \in children(G,m)$, then $reduct(G,mt) \sqsupseteq mef(G,m)$.*

Proof. By Lemma 4 and 5.

Lemma 6. *Let G_0 be an ART, G a condensed ART of G_0 and m a node in G. If $mt \in dom(G_0)$ and G_0 at node m matches the left-hand of a rewriting rule $fp_1...p_n = R$ with $[m_1/x_1, ..., m_k/x_k]$, then we have:*

1. *G at node m matches $fp_1...p_n$ with $[m_1/x_1, ..., m_k/x_k]$,*
2. *$redex(G,m) \equiv (fp_1...p_n)[mef(G,m_1)/x_1, ..., mef(G,m_k)/x_k]$,*
3. *$R[mef(G,m_1)/x_1, ..., mef(G,m_k)/x_k] \sqsupseteq reduct(G,mt)$.*

Proof. For the first two, by induction on the definition of pattern matching and Lemma 1. Similar results have been proved in [2,5].

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

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

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

Theorem 2. *Let m be a faulty node in the EDT for a condensed ART G and the equation of m is $fb_1...b_n = M$, then the definition of the function f in the program is faulty.*

Proof. Since m is a faulty node, we know $redex(G,m) \not\sqsupseteq mef(G,m)$ where $redex(G,m) \equiv fb_1...b_n$ and $mef(G,m) \equiv M$. 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 $\exists\overline{x}\forall\overline{y}.(M' \not\simeq_I N')$ because $redex(G,m) \not\sqsupseteq mef(G,m)$. Suppose there are terms $\overline{e}$ such that $\forall\overline{y}.(M'[\overline{e}/\overline{x}] \not\simeq_I N')$. We write $redex(G,m)[\overline{e}/\_]$ for $M'[\overline{e}/\overline{x}]$. Notice that there is no _ in $\overline{e}$ and $redex(G,m)[\overline{e}/\_]$, and we have

$$redex(G,m)[\overline{e}/\_] \not\sqsupseteq mef(G,m) \tag{1}$$

Now, because there is a computation at the node m, we suppose that the original graph at m matches $fp_1...p_n$ with $[m_1/x_1,...,m_k/x_k]$. Then by Lemma 6, G at m matches $fp_1...p_n$ with $[m_1/x_1,...,m_k/x_k]$. By Lemma 6, we also have

$$redex(G,m) \equiv (fp_1...p_n)[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k]$$

and

$$R[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k] \sqsupseteq reduct(G,mt) \tag{2}$$

So, we have

$$\begin{aligned} & redex(G,m)[\overline{e}/\_] \\ \equiv\ & ((fp_1...p_n)[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k])[\overline{e}/\_] \end{aligned}$$

Notice that there is no _ in $fp_1...p_n$. Therefore, if we replace the _s in $mef(G,m_i)$ by some relevant terms from $\overline{e}$ and get M_i, then we have

$$redex(G,m)[\overline{e}/\_] \equiv (fp_1...p_n)[M_1/x_1,...,M_k/x_k]$$

and $M_1 \sqsupseteq mef(G,m_1)$,..., $M_k \sqsupseteq mef(G,m_k)$.

By the definition of $\rightarrow_P$, we have

$$redex(G,m)[\overline{e}/\_] \rightarrow_P R[M_1/x_1,...,M_k/x_k] \tag{3}$$

By Lemma 3(3), we have

$$R[M_1/x_1,...,M_k/x_k] \sqsupseteq R[mef(G,m_1)/x_1,...,mef(G,m_k)/x_k] \tag{4}$$

Now we have $R[M_1/x_1, ..., M_k/x_k] \sqsupseteq reduct(G, mt)$ by the relations (2) and (4) and Lemma 3(1).

Since m is a faulty node, we have $reduct(G, mt) \sqsupseteq mef(G, m)$ by Corollary 1. Now, by Lemma 3(1), we have

$$R[M_1/x_1, ..., M_k/x_k] \sqsupseteq mef(G, m)$$

Since $redex(G, m)[\overline{e}/\_] \not\sqsupseteq mef(G, m)$ (see the relation (1)), and there is no _ in $redex(G, m)[\overline{e}/\_]$ and $R[M_1/x_1, ..., M_k/x_k]$, by Lemma 1, we have

$$redex(G, m)[\overline{e}/\_] \not\simeq_I R[M_1/x_1, ..., M_k/x_k] \tag{5}$$

Since we have proved the relations (3) and (5), the computation from $redex(G, m)[\overline{e}/\_]$ to $R[M_1/x_1, ..., M_k/x_k]$ is a single-step computation according to the rewriting rule $fp_1...p_n = R$, but $redex(G, m)[\overline{e}/\_]$ is not equal to $R[M_1/x_1, ..., M_k/x_k]$ according to the intended semantics. Therefore, the definition of the function f in the program is faulty.

6 Conclusion and Future Work

In this paper, we formally present three conditions for replacing unevaluated parts in a trace of a functional program. The traces have some interesting features such as high-order functions, sharing and partial applications. Compared with some simpler functional programs in which, for instance, high-order functions are disallowed, it is much harder to give formal conditions to capture our intuition. We also give examples to explain the informal meaning of these conditions. The properties of condensed ARTs are proved. The most important property, the correctness of algorithmic debugging, is also proved. This means that a node can be safely replaced by _ if it satisfies the three conditions.

There is still more work that needs to be done. Currently we are studying two extensions of the ART model.

1. Add local rewriting rules to the program.
2. Remove trusted functions from the ART.

How these two 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. O. Chitil and Y. Luo. Towards a theory of tracing for functional programs based on graph rewriting. In *Proceedings of the third international workshop on Term Graph Rewriting, Termgraph*, volume 7, 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. Y. Luo and O. Chitil. Proving the correctness of algorithmic debugging for functional programs. In *Proceedings of 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. Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.
10. 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.
11. E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.
12. 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.
13. 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.
14. 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).
15. Malcolm Wallace, Olaf Chitil, and Colin Runciman. Hat: transforming lazy functional programs for multiple-view tracing. In preparation, 2004.