Computer Science at Kent

Algorithmic Debugging with Cyclic Traces of Lazy Functional Programs

Yong Luo and Olaf Chitil

Technical Report No. 9 - 07
August 2007

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

Algorithmic Debugging with Cyclic Traces of Lazy Functional Programs

Yong Luo and Olaf Chitil

Computing Laboratory, University of Kent

Abstract We have proved the correctness of algorithmic debugging for functional programs if the traces are acyclic [3]. For cyclic traces, however, does algorithmic debugging still work? There does not exist a common understanding of how to debug cyclic traces in functional programming communities for a long time. In this paper we give two small examples to demonstrate that it is extremely difficult to find a generic algorithmic debugging scheme for cyclic traces. We conjecture that it is impossible to have a generic scheme for cyclic traces because the examples are very small and the choices of reasonable debugging trees are very limited. We also present acyclic traces in which constants are shared unless shared constants result in a cycle. The normal algorithmic debugging scheme works fine for acyclic traces and the proof is very similar to our previous paper [3].

1 Introduction

Tracing for functional programs based on graph rewriting is a process that records information about computations. 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 [8,6,7].

In this paper a trace is an *augmented redex trail (ART)* which is a compact but detailed representation of computations; 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 our previous paper [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 computation strategy. This observation agrees with our idea that functional programmers abstract from time.

Algorithmic debugging can be thought of as searching an debugging tree for a fault in a program. One need to answer several questions according to the

intended semantics in algorithmic debugging scheme [4]. An evaluation dependency tree (EDT) is for algorithmic debugging. If the evaluation of a node in an EDT is not intended then the node is erroneous. All the branches of a node are the children of the node. If a node in an EDT is erroneous but has no erroneous children, then this node is called *a faulty node.* The evaluated function at a faulty node should be a faulty in a program. 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 and EDT for a starting term $main = doubleneg\ (not\ True)$ are in Figure 1 and 2.

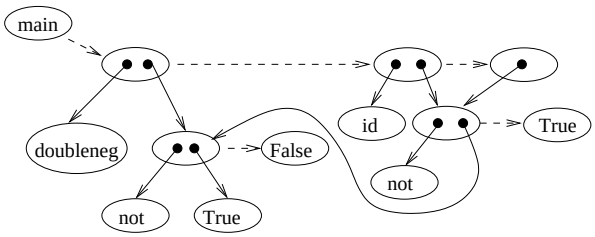

Figure 1. The ART for the Introduction Section

where the dashed lines represent one-step computations.

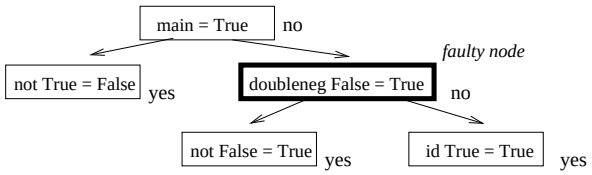

Figure 2. The EDT for the Introduction Section

We have formally presented the ART and EDT and proved important properties, in particular, the correctness of algorithmic debugging [3]. The ART is acyclic. It has sharing (i.e. the arguments of a function can be shared) but constants are not shared.

2 Problem

If we want to share constants there may be cycles in an ART. Sharing constants itself does not make much trouble for algorithmic debugging if there is no cycle in the ART. However, when there are cycles in the ART algorithmic debugging becomes extremely difficult.

First counter example

The following program has one mistake, i.e. the definition of a is faulty.

```
main :: Int
main = h a

h :: (Int, Int) -> Int
h (x, y) = x + y

a :: (Int, Int)
a = f (g a) 1      -- should be: a = f (g a) 2

f :: Int -> Int -> (Int, Int)
f x 1 = (x, 3)
f x 2 = (x, 5)

g :: (Int, Int) -> Int
g (x, y) = snd a + 4
```

The intended semantics:

```
a = f (g a) 2 = (9, 5)

g (x, y) = snd a + 4 = 9

main = h a = 14
```

The cyclic ART for the first counter example is in Figure 3. One simple choice of EDT is in Figure 4.

Now, there is a problem. We know that the definition of a is faulty, but from the EDT in Figure 4 the faulty definition is the function g.

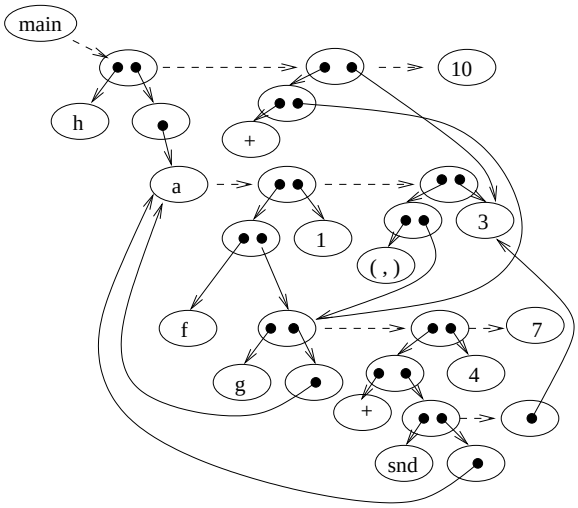

Figure 3. The cyclic ART for the first counter example

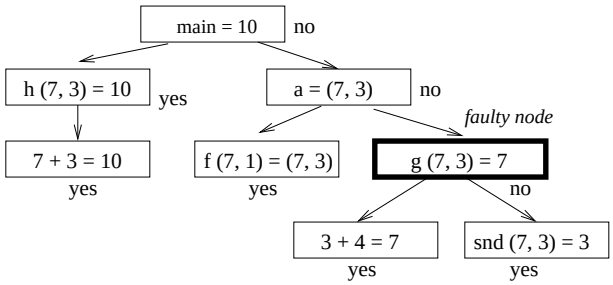

Figure 4. An EDT for the first counter example

Second counter example

The intention of the following program is to demonstrate a black-hole problem, but it has one mistake, i.e. the definition of h is not strict enough.

```
main = f a
f C = C'
a = g (h a)
g C'' = C
h x = C''  -- should be: h C = C''
--where C, C' and C'' are constructors.
```

The cyclic ART and one simple EDT for the second counter example are in Figure 5 and 6.

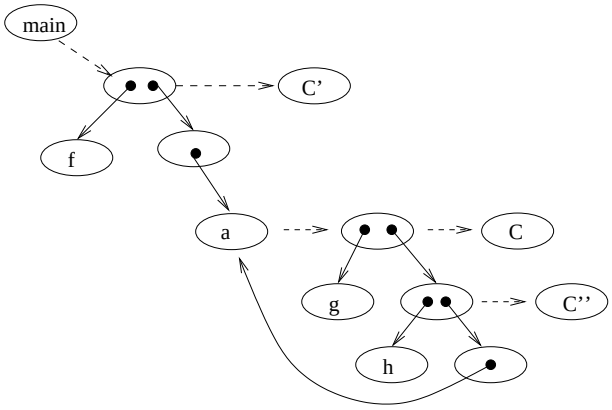

Figure 5. The cyclic ART for the second counter example

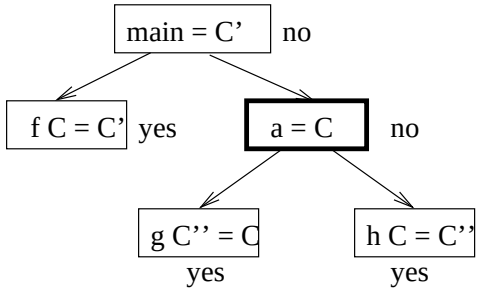

Figure 6. An EDT for the second counter example

The answers to the equations are the following.

main = C'	No, should not have any result at all
f C = C'	Yes
a=C	No, should not have any result at all
g C" = C	Yes
h C = C"	Yes, intended semantics.

There is also a problem here. We know that the definition of h is faulty, but from the EDT in Figure 6 the faulty definition is the function a.

These two examples are very small, and the choices of reasonable debugging trees are very limited. We cannot think of any workable and generic debugging trees for these two examples. So we conjecture that there is not a generic algorithmic debugging scheme for cyclic traces.

3 A proposed solution

Since cycles are killers, an immediate solution is that we only generate acyclic ARTs. On the other hand we want constants to be shared. So we share constants as long as there is no cycle in the ART. We use indirections pointing to shared constants. Indirections help us to have a easier naming scheme to decide computation dependencies, i.e. the parent nodes and their children. The confluence property still hold in the sense that different evaluation orders do not yield different ARTs. We give one more example in the paper. The formal details and proofs can be established as those in our previous paper [3] because the essence is the same, i.e. ARTs are acyclic. We omit the formal presentation here.

Example 3 The program is the following.

```
main = f a a b
f (C x) (C (C y)) z = C'
a = b
b = C a
--where C and C' are constructors.
```

The acyclic ART and EDT for Example 3 are in Figure 7 and 8.

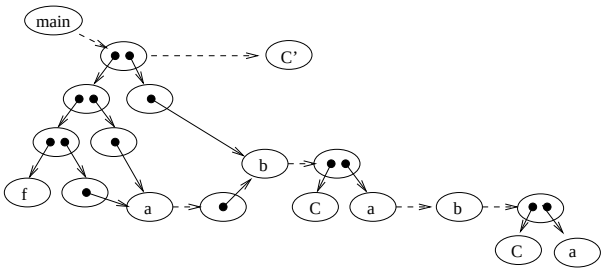

Figure 7. The acyclic ART for Example 3

The constants a and b in the example are shared but not always shared. If sharing a constant results in a cycle then we will start a new node for the constant. Otherwise it will be shared.

Note that the question "b = C (C a)" that comes from the same node in the ART (see Figure 7) is one of the children of "main = C"' and the child of "a = C (C a)" (see Figure 8). So, one question that comes from the same place could appear more than once in an EDT because of constant sharing. In general, such repeated questions in an EDT cannot be removed, otherwise we may end up locating a wrong faulty node. But repeated questions only need to be answered once in practice. We can also use a graph to represent the EDT (see Figure 9).

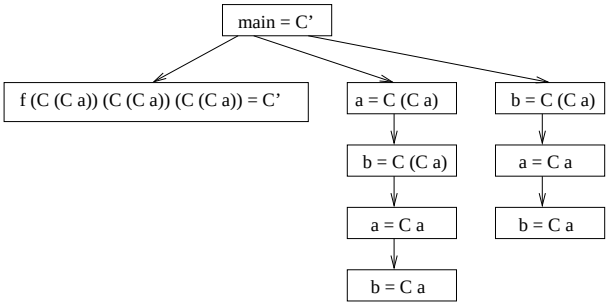

Figure 8. The EDT for Example 3

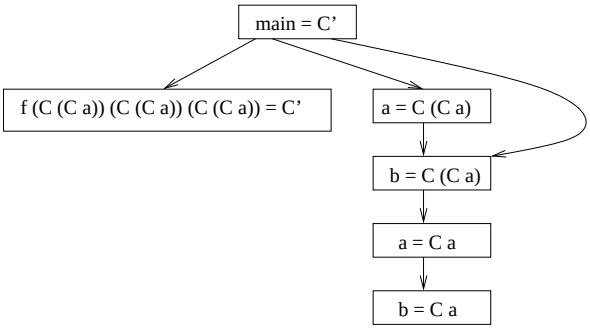

Figure 9. A graph representation of the EDT for Example 3

Now, we give acyclic ARTs and new EDTs for the two counter examples (see Figure 10 - 14). The acyclic ARTs are not as efficient as the cyclic ones because there are more computation in the acyclic ARTs. But the new EDTs derived from the acyclic ARTs can correctly locate the faulty definitions in lazy functional programs.

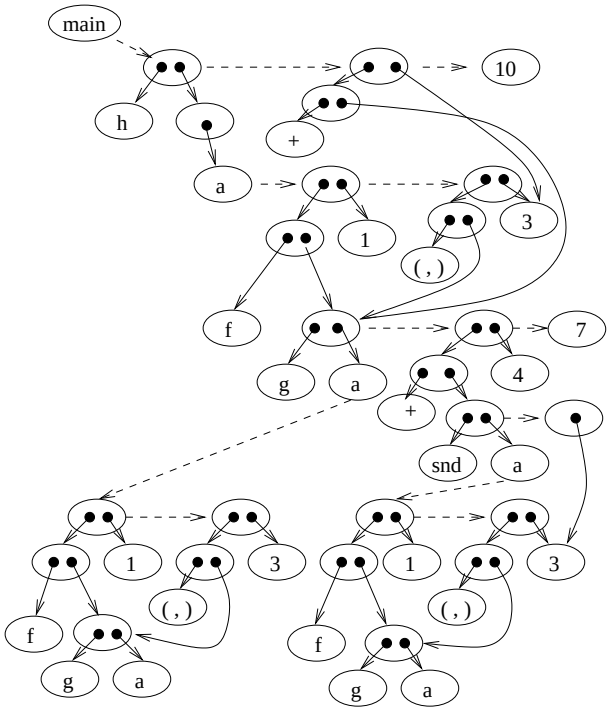

Figure 10. The acyclic ART for the first counter example

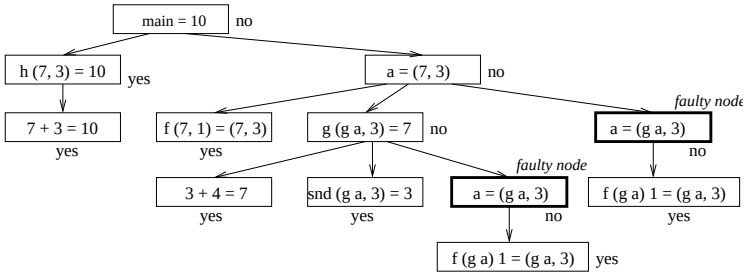

Figure 11. New EDT for the first counter example

If we replace the unevaluated parts by _s, the questions may become clearer.

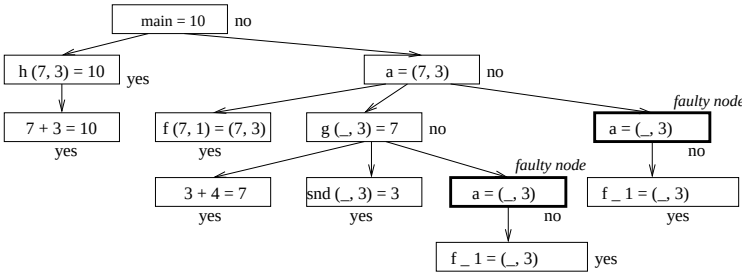

Figure 12. New EDT for the first counter example

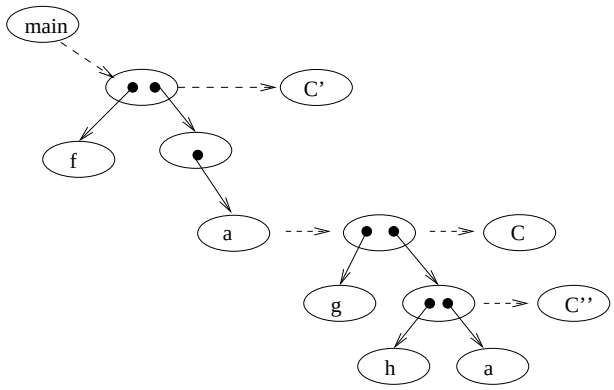

Figure 13. The acyclic ART for the second counter example

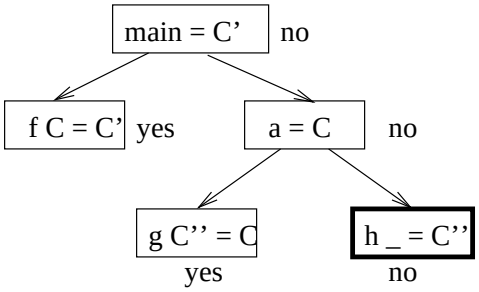

Figure 14. New EDT for the second counter example

Related Work

In some systems such as Freja and Hat, cycles are treated as black boxes. Every cycle (or black box) may have several function definitions. The debuggers can

tell whether there is a bug inside a black box, but cannot tell which function in that box is faulty.

In Nilsson's thesis [5], he demonstrated how to debug cyclic Freja programs. However, the current debugging tool cannot correctly debug the counter examples in this paper. We had extensive discussion about the issue. I was told that Freja could locate the bug if the mutually recursive functions were locally defined. But I have not fully understood this claim.

The idea of *redex trail* is developed and the computation builds its own trial as reduction proceeds [9]. The trace in Hat is recorded in a file rather than in memory [10]. Hat integrates several viewing methods such as Functional Observations, Reduction Trails and Algorithmic debugging.

Naish presents a very abstract and general scheme for algorithmic debugging [4]. 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 [5], 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. 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 [1]. However, the starting point of the approach 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. Important properties of the ART have also been proved [2].

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. 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.
4. Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.
5. 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.
6. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.

7. B. Pope and Lee Naish. Practical aspects of declarative debugging in Haskell-98. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240, 2003.
8. E. Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.
9. Jan Sparud and 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.
10. 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).