Algorithmic Debugging and Trusted Functions

Yong Luo and Olaf Chitil

Technical Report No. 10 - 07
August 2007

Algorithmic Debugging and Trusted Functions

Yong Luo and Olaf Chitil

Computing Laboratory, University of Kent

Abstract As the name states, trusted functions do not have bugs. It is up to user to specify which function is trusted. Commonly used functions in standard library are normally trusted. In the process of algorithmic debugging, we search for faulty nodes to locate bugs. Since a trusted function cannot be a faulty node, there is no point to keep trusted functions in Evaluation Dependency Trees (EDT) for algorithmic debugging. In this report, we create smaller tree structures by removing trusted functions. There are two different ways to achieve this: generating a smaller tree structure directly from the original trace; or creating a smaller trace first and then from which generating a smaller tree structure.

1 Definition

By "a trusted function", we mean that the definition of the function in a program is correct. We trust that there is nothing wrong about the code of the function. For example, we may trust that the function *map* in the standard Haskell library is correctly defined.

Since we have high-order functions such as *map*, the concept of "trusted function" could be a little confusing. For a first-order function f, if f is trusted then any equation of the form

$$f\ v_1......v_n = u$$

must be correct in terms of intentional semantics. However, if f is a high-order function, even if it is trusted, the equation may not be correct. For example,

$$map\ \ g\ \ [a] = [b]$$

may not be correct because the definition of g may be wrong.

2 EDTs from original ARTs

2.1 Original ART

In this draft, we shall use the full ART as defined in our TFP paper [1]. The ART has enough information for general purposes. We shall only remove trusted functions from EDT.

2.2 New EDT

The definition of *children* is the same as before.

Now, let us define a concept called "untrusted children". The EDT has untrusted functions only.

- If n is a child of m, and $head(n)$ is not trusted, then n is an untrusted child of m.
- If n is a child of m, and $head(n)$ is trusted, then all untrusted children of n are also untrusted children of m.

2.3 Properties

There are two properties:

1. If m is a node in an old EDT, and $head(m)$ is trusted function, then m cannot be a faulty node. (Need the proof of correctness of the old EDT.)
2. A faulty node in the new EDT is also a faulty node in the old one. (Need property 1 to prove this.) Since we have proved the correctness of the old EDT, this proves the correctness of the new one.

Note that a faulty node in the old EDT is not necessarily a faulty node in the new one. The following example shows the point. The root node $main = 0$ is a faulty node in the old EDT, but it is not in the new EDT.

```
main :: Int
main = f 5  -- should be: main = f 5 + 1

f :: Int -> Int
f x = g x - h x  -- f is trusted.

g :: Int -> Int
g x = x  -- should be: g x = x + x

h :: Int -> Int
h x = x  -- should be: h x = x + x
```

3 Smaller ARTs and EDTs

We build a smaller ART in order to get a smaller EDT by omitting the trusted functions in the ART.

3.1 Smaller ARTs

Definition 1. ***(Partial and Full application)*** *An application $f\ a_1 \ldots a_n$ is a full application if the arity of f is n, otherwise it is a partial application.*

Remark 1. We shall only compute nodes which are full applications. With this restriction, we don't need to ask questions like $f\ c = g$ in an EDT. It seems a lot easier to deal with trusted functions with this restriction. Moreover, the concept of full application is also useful for replacing unevaluated parts in an ART since we only want to replace unevaluated nodes which are full applications.

An example.

$$f\ x = c_1$$

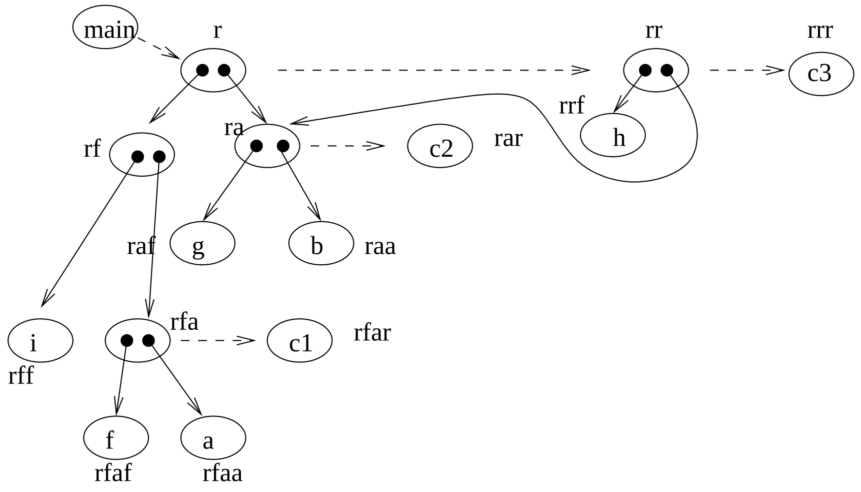
$$g\ y = c_2$$

$$h\ x = c_3$$

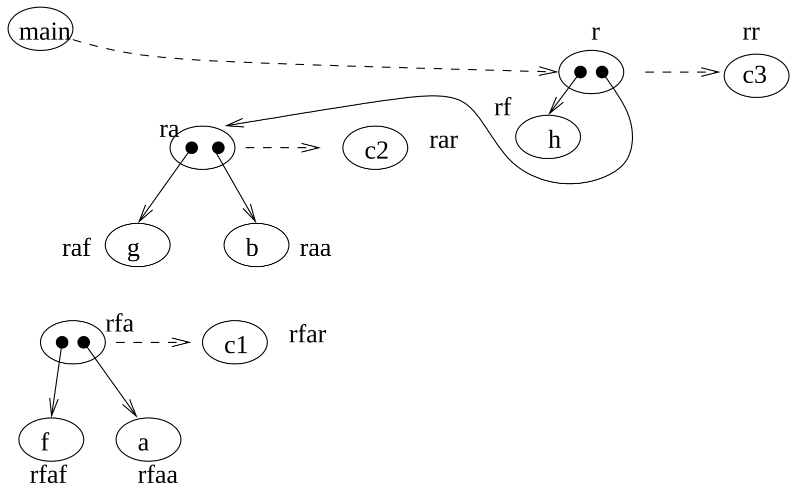
$$i\ c_1\ x = h\ x$$

$$main = i\ (f\ a)\ (g\ b)$$

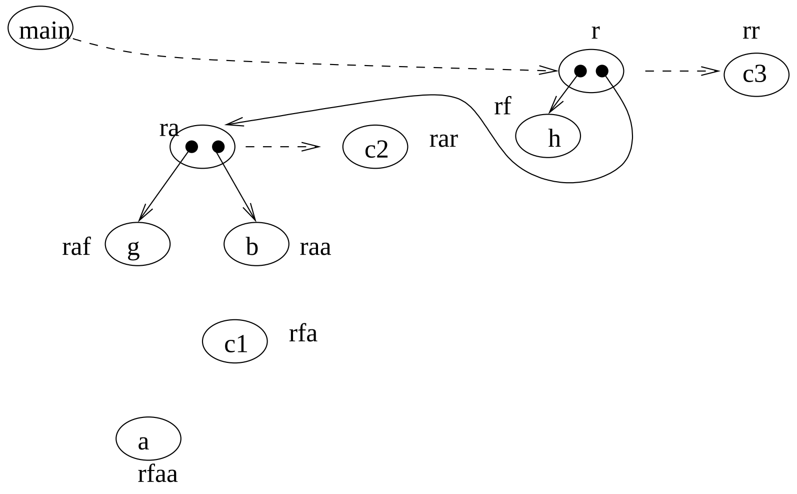
The original graph is the following. (We assume the nodes t etc are full applications)



If i is trusted then the graph will be the following. The old node is replaced by a new node)



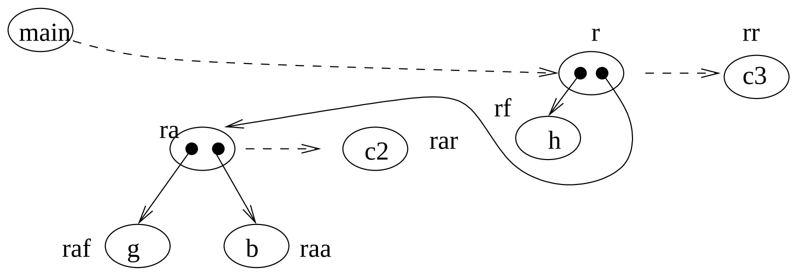
If i and f are trusted then the graph will be



Remark 2. The new graphs look disconnected but we can still figure out the parent edges by the names of nodes.

The smaller ARTs are sufficient for algorithmic debugging. One problem is that the smaller ARTs are not suitable for other purposes because a lot of information was lost.

The disconnected nodes without computation can also be removed.



3.2 Smaller EDTs

The definitions of *children* and EDT are the same as those in TFP paper [1].

Trusted constants

In practice, a trusted constant is often too large to display. In the definition of "most evaluated form", we may just keep the name of the constant. For example,

$$e = a\_huge\_term$$

$$f\ x\ c = c'$$

$$main = f\ e\ c$$

where e is trusted.

4 Another way for smaller ARTs and EDTs

The node labels (eg rr and rrr) do not change, but the definitions of *mef* and *children* are different.

An example.

$$f\ x = c_1$$

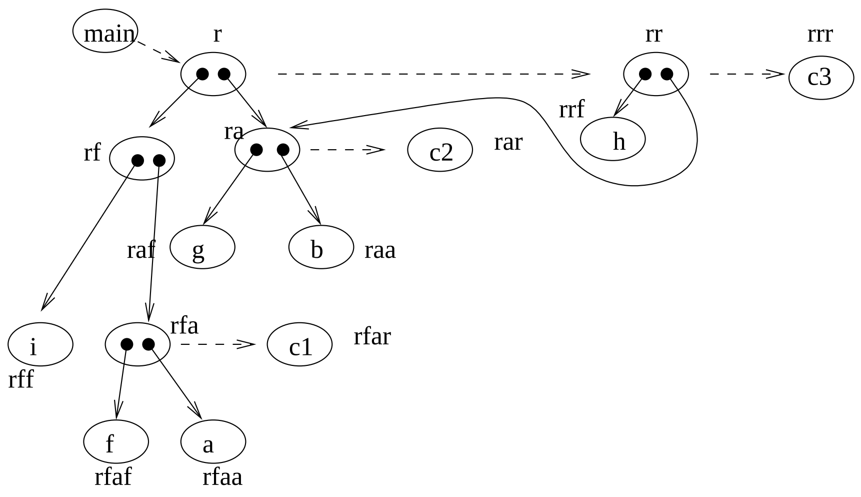
$$g\ y = c_2$$

$$h\ x = c_3$$

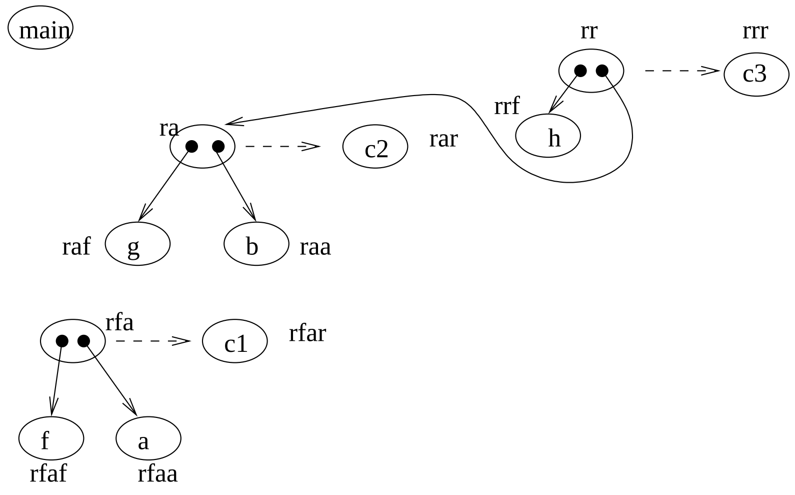
$$i\ c_1\ x = h\ x$$

$$main = i\ (f\ a)\ (g\ b)$$

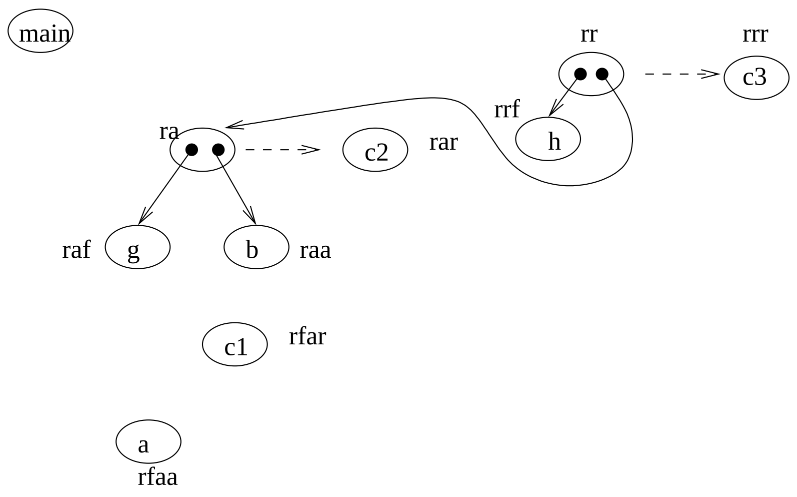
The original graph.



If i is trusted then the graph will be



If i and f are trusted then the graph will be



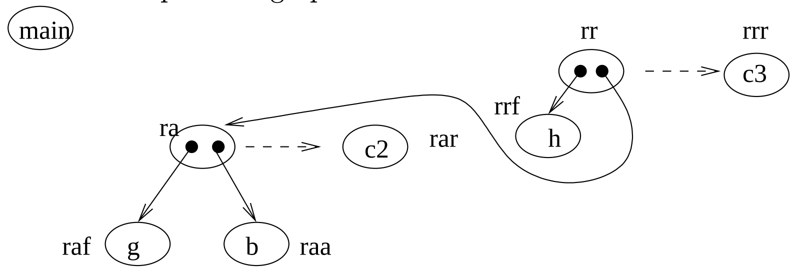
The new definition of "most evaluated form", *mef*.

$$mef(m) = meft(m\mathsf{r}...\mathsf{r})$$

The new definition of "children".

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

The disconnected nodes without computation can also be removed. *main* is not keep in the graph.



References

1. 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.