Algorithmic Debugging of Real-World Haskell Programs: Deriving Dependencies from the Cost Centre Stack

Maarten Faddegon
University of Kent, UK
mf357@kent.ac.uk

Olaf Chitil
University of Kent, UK
oc@kent.ac.uk

Abstract

Existing algorithmic debuggers for Haskell require a transformation of all modules in a program, even libraries that the user does not want to debug and which may use language features not supported by the debugger. This is a pity, because a promising approach to debugging is therefore not applicable to many real-world programs. We use the cost centre stack from the Glasgow Haskell Compiler profiling environment together with runtime value observations as provided by the Haskell Object Observation Debugger (HOOD) to collect enough information for algorithmic debugging. Program annotations are in suspected modules only. With this technique algorithmic debugging is applicable to a much larger set of Haskell programs. This demonstrates that for functional languages in general a simple stack trace extension is useful to support tasks such as profiling and debugging.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords tracing, algorithmic debugging, lazy evaluation, Haskell

1. Introduction

Support for debugging is essential for wider adoption of non-strict functional languages [24, 27]. The algorithmic debugging method is particularly suitable for pure computations (cf. [26]) and thus non-strict functional languages. During program execution intermediate computations are recorded and organised in a computation tree. Post-mortem the algorithmic debugger asks the user to judge whether these intermediate computations agree with their intentions and the debugger eventually locates a defect in the program [21].

The construction of a computation tree is challenging for non-strict functional languages: existing algorithmic debuggers [13, 15, 18, 25] never gained popularity, probably because they have to specially compile or transform all modules of a program, even libraries that the user does not want to debug and which may use language features not supported by the debugger. Thus they cannot be applied to most real-world programs.

Intermediate computations can be recorded with isolated annotations in the code using HOOD [7]. For algorithmic debugging, however, these computations need to be connected and form a computation tree.

In contrast to debugging, time and space profiling of lazy functional programs is not only well-studied [9, 11, 19], but the Glasgow Haskell Compiler[1] (GHC) also provides reliable profiling for real-world Haskell programs. Trace stacks are key for it attributing time and space costs to parts of a program. Today most Haskell libraries are automatically installed with a variant compiled with the profiling flag.

Our key observation is that the information required for connecting individual intermediate computations to a computation tree is closely related to the information available in trace stacks. Essentially the information from trace stacks is an approximation of the former and it is sufficient for constructing a computation tree.

Our work is of broader interest than just Haskell and GHC. Adding a trace stack to an evaluator of a functional language is a small extension that is useful for numerous purposes. Already when Marlow revisited stacks [9], he was aiming to use them for time and space profiling, coverage analysis and traditional debugging. Now we suggest another use of trace stacks. A trace stack is particularly useful for lazy functional language implementations, where the need for debugging tools is most urgent. However, we agree with Marlow that also in eagerly evaluated higher-order languages the call stack does not give accurate information, hence implementing a trace stack combined with our method would be useful for such languages as well.

Based on this observation we present a novel approach to algorithmic debugging of Haskell programs. We need program annotations only in suspected code, where we are looking for defects, not in any trusted modules, which we assume to be correct. We just compile all modules of the faulty program for profiling and the executable uses the standard runtime system for profiling. In this paper we make the following contributions:

- *We present a method for constructing a computation graph from HOOD's observations plus profiling information, which we use for algorithmic debugging* (Section 2).
- *We define a semantics that describes how to collect the information needed to construct a computation graph.* Our semantics is based on Launchbury's semantics for lazy evaluation and it is close to GHC's profiling semantics. Like HOOD our semantics

PLDI'15 , June 13–17, 2015, Portland, OR, USA
Copyright © 2015 ACM 978-1-4503-3468-6/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2737924.2737985
This is the extended version of the paper presented at PLDI.

[1] `http://haskell.org/ghc`

collects the information in a trace that is a sequence of atomic events. (Section 3).

- *We present transformation algorithms to construct a computation tree for algorithmic debugging from the trace.* From the trace we reconstruct computations and connect them in a computation graph (Section 4). In general the graph will contain cycles. We simplify the graph and transform it into a computation tree. (Section 5).
- *We verify soundness for algorithmic debugging on computation trees produced by our semantics and the transformation algorithms.* We test with the property-based testing tool QuickCheck [5] that the program slice identified as defective by our method is indeed defective. We describe how to test fully automatically although algorithmic debugging ordinarily requires interaction with a human user. (Section 6).
- *We implemented a prototype debugger that uses our method and applied it to real-world Haskell programs.* The debugger is a library for the Glasgow Haskell Compiler. The library has to be imported by the program that shall be debugged. The library provides functions for annotating suspected code. During program execution a trace is recorded and post-mortem the actual algorithmic debugging process starts. Our debugger is available from the Haskell package archive Hackage: `cabal install Hoed`. We introduced defects in several Haskell programs including the game Raincat and used our debugger to find the defects (Section 7).

2. Background, Observation and Idea

We use the example program of Figure 1 throughout this section. The expected result of the program is the ordered list `[3,4,5]`, but when executed the program prints `[3,5,4]` instead. The program uses many standard library functions such as `++` and `foldr` that are trusted, that is, assumed to be correct. The defect is in the definition of the `insert` function: the `>` operator should be replaced with `<` and `xs` should be swapped with `ys`.

2.1 Algorithmic Debugging

An algorithmic debugger records information during a program execution that shows unintended behaviour. Post-mortem the debugger presents the user with questions about intermediate computation statements. The user has to judge whether these statements agree with their intentions. After some questions and answers the debugger locates a defect in a slice (i.e. part) of the program [21]. For our example program the interaction could look as follows, with the answers of the user written in *italics*:

```
isort [4,3,5] = [3,5,4]?  no
insert 5 [] = [5]?  yes
insert 3 [5] = [3,5]?  yes
insert 4 [3,5] = [3,5,4]?  no
Defect located in the definition of "insert"!
```

```
main :: IO ()
main = print (isort [4,3,5])

isort :: [Int] -> [Int]
isort = foldr insert []

insert :: Int -> [Int] -> [Int]
insert n ms = let (xs,ys) = span (>n) ms
              in ys ++ (n : xs)
```

Figure 1: A defective program for sorting integers.

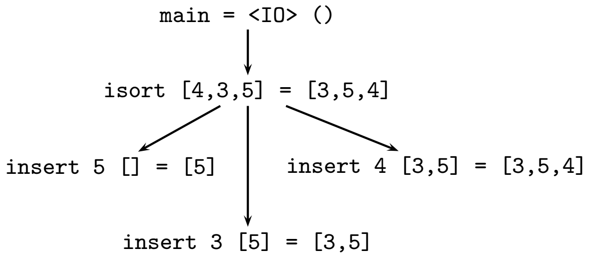


Figure 2: A computation tree for the sorting program.[2]

How did the algorithmic debugger generate this sequence of questions and come to the conclusion? During execution it records information to construct a computation tree, as shown in Figure 2. Computation statements are the nodes of the tree. A computation statement is a child of another computation statement, if computing the latter entails computing the former. We say that the parent computation depends on its child computations. If a computation statement disagrees with the user's intentions, but all child computation statements it depends on do agree with their intentions, then there must be a defect in the program slice associated with the parent computation statement. That is the case, because the computation tree is constructed such that a parent computation is fully determined by its child computations plus the program slice associated with the parent.

2.2 Observing Intermediate Data with HOOD

Gill [7] developed a method for observing intermediate data structures of a Haskell program execution using compact local annotations. Although side-effects are used to collect information, the method guarantees not to change the behaviour of the lazy program. The Haskell Object Observation Debugger (HOOD) is a library that implements the method. For example, if we annotate our program as follows

```
insert = observe "insert" insert'
insert' n ms = let (xs, ys) = span ...
```

then after termination HOOD produces the output:

```
-- insert
  { \ 4 [3,5] -> [3,5,4]
  , \ 3 [5] -> [3,5]
  , \ 5 [] -> [5] }
```

So we observe the value of the function `insert` as far as it was demanded during the program execution. Values such as integers and lists are given in familiar syntax, but functional values have an extensional representation as finite maps, from observed arguments to observed results.[3] Note that only the observed expression is annotated. Other parts of the program that actually produce the value, in particular the definitions of helper functions such as `span` and `++`, are left unchanged.

[2] The monadic IO value of `main` requires special treatment that goes beyond the scope of this paper. Hence we omit monadic IO elsewhere in the paper.

[3] Parts of values, for example list elements, that are not demanded in the computation are just shown as underscore _. Questions by the algorithmic debugger may also contain _ [3], but because of limited space we do not discuss their meaning further in this paper.

2.3 Idea A: Observing Computation Statements

Observing functions à la HOOD allows us to obtain computation statements for the computation tree of algorithmic debugging without making any changes to trusted modules of the program. An observation contains the label written after `observe` and the observed value. We assume that the label is the name of the observed function, as in our example. Then a single argument-result pair of a finite map gives rise to a computation statement. Annotating also `main` and `sort` in our example program, we obtain all computation statements of the tree in Figure 2.

We still need the dependencies between computation statements to construct the tree.

2.4 GHC's Cost Centre Stacks for Profiling

Profiling is the process of attributing time or space costs to parts of a program, for a particular program execution. The GHC profiler expects the user to label program slices with so-called cost centres [19]. For example, we can label the definition of the `isort` function with the cost centre `"isort"` using the `scc` (set cost centre) construct:

```
isort = scc "isort" isort'
isort' = foldr insert []
```

By choosing which expressions we label with cost centres we choose the granularity of assigning profiling costs. We can for example choose not to annotate `foldr` and thus subsume its computation costs into the cost centre `"isort"`.

Morgan and Jarvis [11] noticed that users of profilers often want to change the granularity. For example, they might start with a few cost centres, but on noticing that one cost centre has particularly high costs, they might introduce more cost centres to break down the costs of the conspicuous cost centre. However, re-labelling and re-executing the program takes considerable time. Morgan's and Jarvis' solution is to maintain a stack of cost centres during a computation and attribute a time or space cost to a stack of cost centres. The profiling trace contains cost centre stacks that describe lexical containment in the call-graph. For example, we may label `main`, `isort` and `insert` with cost centres. The costs of the function `isort` without the costs of `insert` is attributed to the stack $\langle \texttt{main}, \texttt{isort} \rangle$, whereas the costs of the function `insert` as called by `isort` is attributed to the stack $\langle \texttt{main}, \texttt{isort}, \texttt{insert} \rangle$. Many different analyses can be performed on the augmented profiling trace.

The cost-centre stack represents a program context that constructs or calls the currently evaluated expression. In contrast, in a lazy language the run-time stack represents a program context that demands the value of the currently evaluated expression. Hence these two stacks differ substantially.

2.5 Idea B: Dependencies from Observed Stacks

We have to combine the two sorts of annotations, one for observing a function à la HOOD and the other one for setting a cost centre, into a single annotation that initiates observing a function and that also records a snapshot of the cost-centre stack with this observation. The stacks enable us to determine dependencies between computation statements and thus construct a computation tree.

2.6 The Computation Tree

The stacks provide only an approximation of the run-time dependencies: First, a stack only contains labels, not complete computations. Hence if for example our program used `isort` twice, we would not know which computations of `insert` belonged to which computation of `isort`. Second, as proposed by Morgan and Jarvis, GHC does not record complete stacks but uses compressed stacks that approximate the real ones, to minimise the overhead of profiling. Thus some precision is lost.

$$\begin{array}{llcll}
\text{expression} & e & ::= & v & \\
 & & | & e\ x & \text{application}\\
 & & | & \texttt{let}\{x_i = e_i\}e & \text{binding}\\
 & & | & x & \text{variable}\\
\text{value} & v & ::= & c & \\
 & & | & \lambda x.e & \text{abstraction}\\
\text{constant} & c & ::= & 0|1|2|\ldots &
\end{array}$$

Figure 3: Syntax of the core language with constants.

Because we use approximations of the dependencies, we obtain a computation graph with cycles, not a tree. We transform that graph, again using safe approximations, to obtain a tree. Finally we can perform standard algorithmic debugging with that computation tree.

Chitil and Davie [3] make the point that there are several structurally different computation trees for the same computation. They study two choices for making a computation statement $f\ v_f = v'_f$ the parent of a computation statement $g\ v_g = v'_g$: either function identifier g appears in the definition of function f, or the application of g to v_g appears in the definition of function f. In a higher-order language the function identifier and the application may appear in different program parts. The first choice requires functional values to be represented extensionally as finite maps, whereas the second choice requires them to be represented intensionally, usually as partial applications of function identifiers and data constructors. Most algorithmic debuggers presented in the literature [2, 13, 15, 25] choose the second structure, but here we choose the first: observing values can represent functional values only as finite maps, not obtain an intensional representation.

3. Tracing Semantics

We define a semantics to unambiguously describe what information we record in a trace while evaluating an expression. We start with Launchbury's semantics for lazy evaluation [8] and add cost centre stacks following Marlow [9]. Subsequently we extend the semantics with tracing based on the HOOD implementation from Gill [7]. We extend the semantics such that a snapshot of the stack of labels is included in each HOOD-like observation.

3.1 Launchbury's Semantics for Lazy Evaluation

Launchbury's semantics assumes normalised λ-expressions as shown in Figure 3. The argument of every application is a variable. All bound variables are distinct such that scope becomes irrelevant. The bindings in a let-expression are mutually recursive. We include numeric constants amongst values here to construct simple examples later.

A heap is a partial function from variables to expressions. We write $\Gamma[x \mapsto e]$ for the heap that is equal to the heap Γ but additionally maps the variable x to the expression e. Figure 4 defines the computation statement $\Gamma : e \Downarrow \Delta : v$, which means that the expression e in the context of the heap Γ reduces to the value v together with the modified heap Δ. In the Var rule the result value is duplicated. To preserve the invariant that all bound variables are distinct, we rename all bound variables with fresh variables in the copy. This renaming is written as $\widehat{v}$.

3.2 Adding a Stack

We reformulate Marlow's framework of stacks as implemented in GHC for our computation statements. Marlow's semantics is quite different from Morgan and Jarvis' semantics [11]; this paper and

$$\frac{\Gamma, \mathcal{S} \lhd l : e \Downarrow \Delta, \mathcal{S}' : v}{\Gamma, \mathcal{S} : \texttt{push}\ l\ e \Downarrow \Delta, \mathcal{S}' : v}\ \text{Push} \qquad \frac{\Gamma, \mathcal{S} : e \Downarrow \Delta, \mathcal{S}' : \lambda y.e' \qquad \Delta, \mathcal{S}' : e'[x/y] \Downarrow \Theta, \mathcal{S}'' : v}{\Gamma, \mathcal{S} : e\ x \Downarrow \Theta, \mathcal{S}'' : v}\ \text{App}$$

$$\Gamma, \mathcal{S} : v \Downarrow \Gamma, \mathcal{S} : v \quad \text{Val} \qquad \frac{\Gamma[x_i \mapsto (\mathcal{S}, e_i)], \mathcal{S} : e \Downarrow \Delta, \mathcal{S}' : v}{\Gamma, \mathcal{S} : \texttt{let}\{x_i = e_i\}\ e \Downarrow \Delta, \mathcal{S}' : v}\ \text{Let}$$

$$\frac{\Gamma, \mathcal{S}_{\text{h}} : e \Downarrow \Delta, \mathcal{S}' : c}{\Gamma[x \mapsto (\mathcal{S}_{\text{h}}, e)], \mathcal{S} : x \Downarrow \Delta[x \mapsto (\mathcal{S}', c)], \mathcal{S}' : c}\ \text{VarC}$$

$$\frac{\Gamma, \mathcal{S} : e \Downarrow \Delta, \mathcal{S}_\lambda : \lambda y.e'}{\Gamma[x \mapsto (\mathcal{S}, e)], \mathcal{S}_{\text{app}} : x \Downarrow \Delta[x \mapsto (\mathcal{S}_\lambda, \lambda y.e')], \mathcal{S}_{\text{app}} \bowtie \mathcal{S}_\lambda : \widehat{\lambda y.e'}}\ \text{VarL}$$

Figure 6: New and updated rules adding a stack to Launchbury's semantics.

$$\Gamma : v \Downarrow \Gamma : v \quad \text{Val}$$

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.e' \qquad \Delta : e'[x/y] \Downarrow \Theta : v}{\Gamma : e\ x \Downarrow \Theta : v}\ \text{App}$$

$$\frac{\Gamma[x_i \mapsto e_i] : e \Downarrow \Delta : v}{\Gamma : \texttt{let}\{x_i = e_i\} e \Downarrow \Delta : v}\ \text{Let}$$

$$\frac{\Gamma : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : \widehat{v}}\ \text{Var}$$

Figure 4: Launchbury's semantics.

expression	e	$::=$	$\dots$	
		$\mid$	$\texttt{push}\ l\ e$	push label onto stack

Figure 5: Extending the syntax to label expressions.

the GHC implementation follow the former. First, in Figure 5 we extend the syntax by an operation for labelling any subexpression within a program. The nature of labels is irrelevant; in practice we use strings.

Figure 6 shows the semantic rules. We add a stack $\mathcal{S}$ of labels to the computation statements. This stack does not influence the result value. The result stack is just a basic trace of the computation. The statement $\Gamma, \mathcal{S} : e \Downarrow \Delta, \mathcal{S}' : v$ means that the expression e in the context of the heap Γ and the stack $\mathcal{S}$ reduces to the value v in the context of the modified heap Δ and modified stack $\mathcal{S}'$. The Push rule pushes the label onto the stack using the function $\lhd$. The heap now stores a stack with every expression. When the Let rule stores an expression in the heap, it includes the current stack. When the VarC and VarL rules evaluate an expression from the heap, they temporarily restore the stack. The VarL rule has a stack $\mathcal{S}_\lambda$ for the λ-abstraction and a stack $\mathcal{S}_{\text{app}}$ for the application of this λ-abstraction. In a higher-order language these stacks can differ substantially. The function $\bowtie$ merges the two stacks for the result.

Figure 7 gives Marlow's definitions of the two functions $\lhd$ and $\bowtie$. Our dependency generation technique is independent of the precise definition of these stack operations. We write the stack as a sequence of labels that grows to the right. The definition of $\lhd$ ensures that every label occurs at most once in a stack. Limiting the size of stacks limits the size of (profiling and debugging) traces. The definition of $\bowtie$ ensures that information of both stacks is used and that the semantics with stacks has useful properties for program optimisation.

$$\begin{aligned} &\langle l_0, \dots, l_n \rangle \lhd l = \langle l_0, \dots, l_j, l \rangle \\ &\quad \text{where } l \notin \{l_0, \dots l_j\} \text{ and } (j = n \text{ or } l_{j+1} = l) \end{aligned}$$

$$\begin{aligned} &\langle l_0, \dots, l_j, l_a, \dots, l_b \rangle \bowtie \langle l_0, \dots, l_j, l_c, \dots, l_d \rangle \\ &\quad = \langle l_0, \dots, l_j \rangle \lhd l_c \lhd \dots \lhd l_d \lhd l_a \lhd \dots \lhd l_b \\ &\quad \text{where } l_a \neq l_c \end{aligned}$$

Figure 7: Definitions of $\lhd$ and $\bowtie$ as used in GHC.

trace	$\mathcal{T}$	$::=$	$t_0, \dots, t_n$	sequence growing right
event	t	$::=$	$\text{T}\ l\ \mathcal{S}$	root
		$\mid$	$\text{T}_{\text{c}}\ p\ c$	constant value
		$\mid$	$\text{T}_\lambda\ p$	abstraction
		$\mid$	$\text{T}_{\text{a}}\ p$	application
identifier	i	$::=$	$0 \mid 1 \mid 2 \mid \dots$	
parent	p	$::=$	$\text{P}\ i$	parent is i
		$\mid$	$\text{P}_{\text{a}}\ i$	argument of application i
		$\mid$	$\text{P}_{\text{r}}\ i$	result of application i

Figure 8: Syntax of the events in our trace.

3.3 Adding a Value Observation Trace

Finally we reformulate HOOD's implementation of value observations for our computation statements. Evaluation is demand driven and hence evaluation of an observed expression does not happen in a consecutive sequence of steps but is interleaved with other reduction steps. Hence during program execution we collect a list of observation events. Post-mortem we combine these to representations of values.

Figure 8 gives the syntax of events. A trace $\mathcal{T} = t_0, \dots, t_n$ is a list of events, where each individual event is identified by its position in the list. Indexing starts at 0. Except for a root event T l S, every event contains an identifier i. The identifier states the parent event in the tree. Thus events form a forest of trees. We store label and stack separately in the root event because pushing is not lossless.

Consider the example trace in Figure 9. It consists of five events. They form the single tree shown in Figure 10, which represents the observed functional value $\{\texttt{\9->9}\}$ with label "*id*" and stack $\langle\rangle$.

$$\frac{\Gamma, \mathcal{S} \triangleleft l, \mathcal{T} \lessdot \mathrm{T}\ l\ \mathcal{S} : \texttt{obs}\ (\mathrm{P}\ |\mathcal{T}|)\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{push}\ l\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}\ \text{Push}$$

$$\frac{\Gamma, \mathcal{S}, \mathcal{T} : e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : c}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{obs}\ p\ e \Downarrow \Delta : \mathcal{S}', \mathcal{T}' \lessdot \mathrm{T}_{\mathrm{c}}\ p\ c : c}\ \text{ObsC}$$

$$\frac{\Gamma, \mathcal{S}, \mathcal{T} : e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : \lambda x.e'}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{obs}\ p\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' \lessdot \mathrm{T}_{\lambda}\ p : \lambda x.\texttt{obs}_{\lambda}\ (\mathrm{P}\ |\mathcal{T}'|)\ x\ e'}\ \text{ObsL}$$

$$\frac{\Gamma, \mathcal{S}, \mathcal{T} \lessdot \mathrm{T}_{\mathrm{a}}\ p : \texttt{let}\{x' = \texttt{obs}\ (\mathrm{P}_{\mathrm{a}}\ |\mathcal{T}|)\ x\}((\lambda x.\texttt{obs}\ (\mathrm{P}_{\mathrm{r}}\ |\mathcal{T}|)\ e)\ x') \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{obs}_{\lambda}\ p\ x\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}\ \text{Obs}_{\lambda}$$

Figure 12: Modified and new rules added to the profiling semantics for tracing.

0 :	T “*id*” ⟨⟩	label with empty stack
1 :	T_{λ} (P 0)	abstraction
2 :	T_{a} (P 1)	application of the abstraction
3 :	T_{c} (P_{a} 2) 9	argument of this application
4 :	T_{c} (P_{r} 2) 9	result of this application

Figure 9: Trace produced by evaluating the expression $\texttt{let}\{\texttt{id} = \texttt{push}$ “*id*” $(\lambda x.x), \texttt{y} = 9\}\ (\texttt{id}\ \texttt{y})$.

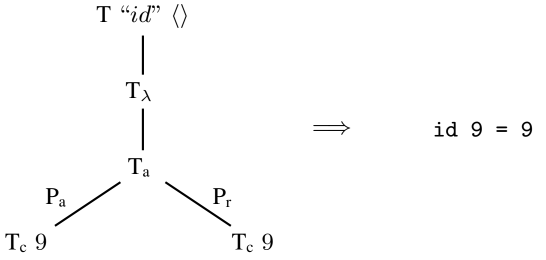


Figure 10: Tree of events from the trace in Figure 9. An application event has two events as children, the argument with parent P_{a} and the result with parent P_{r} .

$$\begin{array}{lcll} \text{expression} & e & ::= & \ldots \\ & & | & \texttt{obs}\ p\ e \quad \text{observe expression} \\ & & | & \texttt{obs}_{\lambda}\ p\ x\ e \quad \text{observe application} \end{array}$$

Figure 11: Extended syntax to observe expressions.

To observe the values of an expression we use two pseudo-functions `obs` and $\texttt{obs}_{\lambda}$. Figure 11 adds them to the syntax. They are never used in a program but appear only during program execution.

Figure 12 shows the non-trivial changes that turn the profiling semantics into a tracing semantics. For the other rules we just add a trace $\mathcal{T}$ on both left- and right-hand side of every reduction.

$t_0, \ldots, t_n \lessdot t = t_0, \ldots, t_n, t$ appends an event to the trace. $|\mathcal{T}|$ determines the length of trace $\mathcal{T}$ and thus the index of the event that is appended next.

We modified the Push rule such that it records a root event which includes the current stack. Furthermore it wraps the observed expression with the pseudo-function `obs`. The index of the parent event is passed to `obs` to enable connecting events later.

An observed expression evaluates to either a constant or an abstraction. For constants the ObsC rule records the constant value itself as a trace event. For abstractions the ObsL rule adds a $\mathrm{T}_{\lambda}\ p$ event to the trace and continues observing every application of the abstraction using the pseudo-function $\texttt{obs}_{\lambda}$.

We reduce $\texttt{obs}_{\lambda}$ every time an observed abstraction is applied using the Obs_{λ} rule: an application event is recorded in the trace and we continue observing argument and result with the pseudo-function `obs`. Both argument and result of the function have the application event as parent. If a functional value is applied several times, then several $\mathrm{T}_{\mathrm{a}}\ p$ events will share the same $\mathrm{T}_{\lambda}\ p'$ event, representing several applications within one finite map.

4. Processing the Trace

We defined how a trace with a forest of event trees results from evaluating a program. Now we are ready to translate each tree into one or several computation statements and use the snapshots of the stack to derive dependencies between the statements and form a computation graph.

4.1 From Trace to Computation Statements

We want to translate a forest of events. At the root of a tree we find an event with the label given to an observed expression. We assume this label is equal to the function or variable name used in the program slice. Next we look at the child node of the root.

A constant node is translated into a simple computation statement with the label of the root node on the left-hand-side and the representation from the constant event on the right-hand-side. For example, if the label is “x” and the value 2, then the resulting computation statement is `x = 2`.

For observed functions the structure is more complicated. Consider for example the tree in Figure 10. The root of the tree is again an event with a label. The child of the root event is an abstraction event. An abstraction event has one or more application events as children. Each application event yields at least one computation statement. An application event can have one argument event and one result event. In our example those are both constant events.

When the argument to a function is not used to compute the functions result, there is also no argument event in the trace. In that case the argument is represented by an underscore (e.g. `f _ = 3` for a function always returning 3).

The child of an application event can be another abstraction event. Figure 13 illustrates the two possible cases:

On the left: the observed function takes another function as argument. In the computation statement the outer argument is represented by a finite map from nested arguments to nested results.

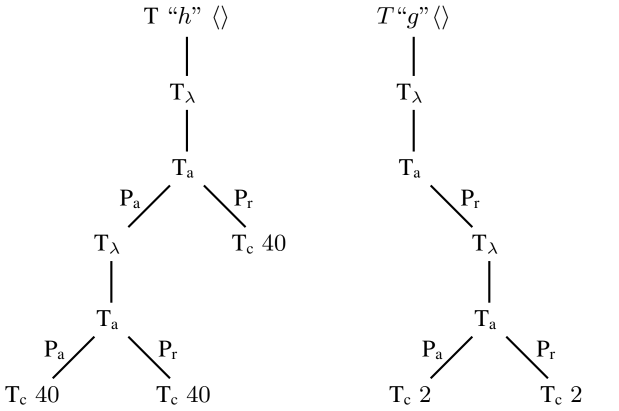


Figure 13: A forest of events translating to the statements "`h {\40->40} = 40`" and "`g _ 2 = 2`".

On the right: the observed function produces another function as result. Another way of looking at this case is saying that the observed function takes multiple arguments. So the computation statement has an application of the function name to several arguments on the left-hand-side of `=`.

4.2 Constructing a Computation Graph

For each computation statement c we also have a snapshot of the stack $\mathcal{S}$ and a label l from which we derive dependencies $c_1 \rightarrow c_2$ between computation statements and construct a computation graph. We need to consider the two ways in which we manipulate the stack of labels in our semantics: the two functions $\triangleleft$ and $\bowtie$.

4.2.1 Stack Push Function $\triangleleft$

In our semantics we push a label onto the stack after recording the stack in the trace and before evaluating the labelled expression. Therefore the computation graph has a dependency $c_1 \rightarrow c_2$ when $\mathcal{S}_1 \triangleleft l_1 = \mathcal{S}_2$ for statement c_1 with label l_1 and stack $\mathcal{S}_1$, and c_2 with l_2 and $\mathcal{S}_2$. Consider for example the labelled expressions in the following program:

$$\texttt{let}\ \{\texttt{y} = 3\}\ (\texttt{push}\ \text{“}A\text{”}\ (\lambda x.((\texttt{push}\ \text{“}B\text{”}\ (\lambda y.y))\ x))\ \texttt{y}$$

Evaluation gives us the two computation statements `A 3 = 3` and `B 3 = 3`. Here label "A" and empty stack $\langle\rangle$ are associated with the former statement, and label "B" and stack $\langle \text{“}A\text{”} \rangle$ with the latter statement. Because $\langle\rangle \triangleleft \text{“}A\text{”} = \langle \text{“}A\text{”} \rangle$, there is a dependency `A 3 = 3` $\rightarrow$ `B 3 = 3` in the computation graph.

Recursively applied functions truncate the stack, resulting in additional edges. Consider for example a program with a function r that is recursively applied three times, and a function m that applies r. Assume m produces statement c_1, r applied in m produces statement c_2, the first recursive application of r gives us c_3 and the second recursive application c_4. Statement c_1 has label m and stack $\langle\rangle$, c_2 has r and $\langle m \rangle$, c_3 has r and $\langle m, r \rangle$, and c_4 has r and the truncated stack $\langle m, r \rangle$. This give us the dependencies $c_1 \rightarrow c_2$, $c_2 \rightarrow c_3$, $c_2 \rightarrow c_4$ and $c_3 \rightleftarrows c_4$.

4.2.2 Stack Merge Function $\bowtie$

The computation graph has dependencies $c_1 \rightarrow c_2 \rightarrow c_3$ when $\mathcal{S}_3 = \mathcal{S}_1 \triangleleft l_1 \bowtie \mathcal{S}_2 \triangleleft l_2$ for statement c_1 with label l_1 and stack $\mathcal{S}_1$, c_2 with l_2 and $\mathcal{S}_2$, and c_3 with l_3 and $\mathcal{S}_3$. The idea is to include dependencies on the constant part of a function definition. Consider for example the following program where the constant part of function f is underlined:

$$\begin{array}{l} \texttt{let}\ \{\texttt{k} = 3, \\ \qquad \texttt{f} = \underline{\texttt{push}\ \text{“}f\text{”}\ (\texttt{let}\ \{g = \lambda x.x\}} \\ \qquad\qquad\qquad\qquad \lambda y.(\texttt{push}\ \text{“}f\_in\text{”}\ \texttt{g})\ y)\} \\ \texttt{push}\ \text{“}main\text{”}\ (\texttt{f k}) \end{array}$$

Evaluating this program gives us the three computation statements below. Note how the VarL rule from Figure 6 affects the stack of the f_in statement.

$$\begin{array}{lll} \texttt{main = 3} & \text{“}main\text{”} & \langle\rangle \\ \texttt{f 3 = 3} & \text{“}f\text{”} & \langle\rangle \\ \texttt{f\_in 3 = 3} & \text{“}f\_in\text{”} & \langle \text{“}f\text{”}, \text{“}main\text{”} \rangle \end{array}$$

Because $\langle \text{“}f\text{”}, \text{“}main\text{”} \rangle = \langle \text{“}main\text{”} \rangle \bowtie \langle \text{“}f\text{”} \rangle$ we have dependencies `main = 3` $\rightarrow$ `f 3 = 3` $\rightarrow$ `f_in 3 = 3` in our computation graph.

4.2.3 Constants

Constants are computed when needed and shared among their users. Nilsson [13] already noted that it is difficult to define when a computation statement depends on a constant's subtree. He presents two solutions: tracking references to free variables explicitly, or sorting computation trees by evaluation order. In the free-browse mode of our prototype debugger the user is responsible for exploring constants first.

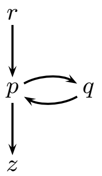


Figure 14: Reducible

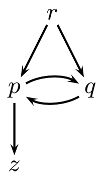


Figure 15: Irreducible

5. From Computation Graph to Tree

So from the trace we obtain a computation graph that may have cycles. In this section we discuss how we remove the cycles to obtain a directed acyclic graph (DAG). A directed acyclic graph (DAG) is just a more efficient representation of a tree where equal subtrees may be shared. Finding defective nodes in a computation tree is an established technique [21].

A computation graph consists of nodes (the computation statements) and edges (the dependencies). If there is a dependency $c \rightarrow d$, then d is a successor of c in the computation graph. A *cycle* is formed by a set of statements for which there exists a path from any statement to any other statement in the set. Node d *dominates* node p, if every paths from the root node to p contains d. Node d is the *dominator* of a set, if d dominates all other nodes in the set. A cycle is *reducible*, if the set contains a dominator, *irreducible* otherwise (e.g. the cycle in Figure 14 has dominator p whereas the cycle in Figure 15 has no dominator). If node d dominates node p, then $p \rightarrow d$ is a *back edge*.

5.1 Reducible Cycles

The actual computation dependencies form a tree and by definition a tree does not have back edges. Therefore we can safely remove the back edges from our computation graph. This will break any reducible cycle. Consider for example the reducible cycle of p and q in the graph of Figure 14. Without back edge $q \rightarrow p$ this is an acyclic graph.

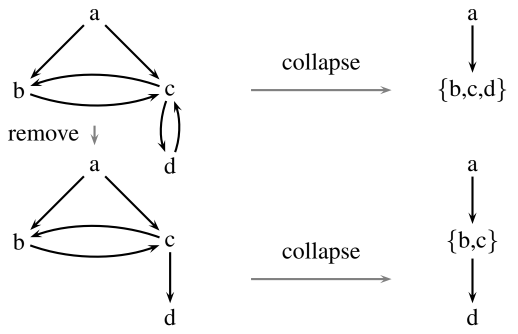


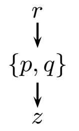
Figure 16: Order of `collapse` and `remove` for reducible cycle $\{c, d\}$ nested in irreducible cycle $\{b, c, d\}$.

5.2 Irreducible Cycles

After removing back-edges some irreducible cycles may remain. We *collapse* an irreducible cycle by replacing the statements in the cycle with a single node in which these statements are combined. For example, we replace the statements p and q in the irreducible cycle of Figure 15 with node $\{p, q\}$. If any of the combined statements is judged wrong, then we consider the node wrong. If the node is defective, then the actual defect is in the union of the program slices associated with the wrong statements in the node.

Dependencies from a statement outside an irreducible cycle to a statement in the irreducible cycle are represented by a dependency onto the collapsed node. Vice versa dependencies from inside the cycle on a statement outside the cycle are represented by a dependency from the collapsed node. Dependencies between two statements in the same irreducible cycle are not represented. Any other dependencies in the computation graph are left unchanged.

Consider the graph of Figure 15 again. The dependencies $r \to p$ and $r \to q$ into the cycle are represented by the dependency $r \to \{p, q\}$. The dependency from p inside the cycle on z outside the cycle is represented by $\{p, q\} \to z$. Thus collapsing gives us the following graph:



5.3 Accuracy and Order

We first remove back edges of all reducible sets, collapse any remaining (irreducible) cycles and use the resulting DAG to find defective nodes. Instead of both removing back edges and then collapsing irreducible cycles we could just collapse. However, there is a good reason not to do so: the collapsed node covers a larger slice of the program than the individual nodes. If the collapsed node is defective, then we can give the programmer less precise direction to where they need to correct the program.

A reducible cycle can be nested inside an irreducible cycle. In that case the order matters: removing back edges can reduce the number of statements in the irreducible cycle that need to be collapsed. Thus a smaller set of cost centres is covered by the collapsed node. Figure 16 shows an example.

6. Soundness

We assume that any defective program slice is labelled. Algorithmic debugging is known to be sound for a computation tree, that is, if the root node of the computation tree disagrees with the programmer's intentions, then the algorithmic debugging process will find a defective node in the tree [12, 21]. However, does our method, which takes dependencies from an approximated stack, allow us to construct a computation tree? In other words:

> Is the program slice associated with an identified defective node really defective?

A proof would be a major undertaking, because in the preceding sections we formally defined only central parts of our method and even the relatively small prototype implementation is quite complex. Hence instead of a proof we test, following Marlow [9], who verified properties of the stack operations $\lhd$ and $\bowtie$ with the property-based testing tool QuickCheck [5]. Already during development of our idea thorough testing proved useful: it uncovered several mistakes in earlier versions.

6.1 How to Test an Algorithmic Debugger

QuickCheck checks for a number of randomly-generated inputs that a property expressed in a logic holds. We can easily generate a random program expression. We can also evaluate such an expression with a Haskell implementation of our semantics to obtain a trace.[4] However, to test with a QuickCheck property that our method indeed constructs a computation tree for algorithmic debugging, we need to clarify three tasks:

1. We need to declare some randomly chosen slices to be defective, so that we know where the defects are and can compare with what our algorithmic debugger finds.
2. During evaluation a defective slice somehow has to cause an infection such that unintended values are computed.
3. After the computation tree has been constructed from the recorded trace, we need to judge automatically whether a given computation statement is intended or not. Normally a human user does this interactively.

We implement these three related tasks with the type:

$$b \ ::= \ ☺ \mid ☹$$

6.2 Defective Slices

We extend labelling such that every program slice is declared working or defective. For example in

$$\texttt{push}\ \text{“}\textit{outer}\text{”}\ ☺\ (\lambda x.\texttt{push}\ \text{“}\textit{inner}\text{”}\ ☹\ x)$$

the slice labelled "outer" is a working slice, but that labelled "inner" is a defective slice. By default any unlabelled slice is considered working.

6.3 Abstracting Values

Algorithmic debugging does not care about specific values, that is, whether a value is a number 42 or a function $\{\backslash 2 \to 4, \backslash 4 \to 8\}$, but only whether a value has been infected and whether in consequence a computation statement is intended or not. We replace in our syntax the numeric constants by our Booleans ☺ and ☹.

In a randomly-generated expression ☹ never appears, but all constants are ☺.

[4] Evaluation may not terminate. We abort evaluation after a given number of steps. An expression may be ill-formed. We abort evaluation when a constant is applied to some argument. QuickCheck allows us to ignore these cases, considering them neither as counter examples nor as successful tests.

$$\frac{\Gamma, \mathcal{S} \lhd l, \mathcal{T} \lessdot \mathrm{T}\ l\ \mathcal{S} : \texttt{obs}\ (\mathrm{P}\ |\mathcal{T}|)\ b\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{push}\ l\ b\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}\ \text{Push}$$

$$\frac{\Gamma, \mathcal{S}, \mathcal{T} : e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : b' \qquad b'' = b \wedge b'}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{obs}\ p\ b\ e \Downarrow \Delta : \mathcal{S}', \mathcal{T}' \lessdot \mathrm{T}_{\mathrm{c}}\ p\ b'' : b''}\ \text{ObsC}$$

$$\frac{\Gamma, \mathcal{S}, \mathcal{T} : e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : \lambda x.e'}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{obs}\ p\ b\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' \lessdot \mathrm{T}_{\lambda}\ p : \lambda x.\texttt{obs}_{\lambda}\ (\mathrm{P}\ |\mathcal{T}'|)\ b\ x\ e'}\ \text{ObsL}$$

$$\frac{\Gamma, \mathcal{S}, \mathcal{T} \lessdot \mathrm{T}_{\mathrm{a}}\ p : \texttt{let}\{x' = \texttt{obs}\ (\mathrm{P}_{\mathrm{a}}\ |\mathcal{T}|)\ \smiley\ x\}((\lambda x.\texttt{obs}\ (\mathrm{P}_{\mathrm{r}}\ |\mathcal{T}|)\ b\ e)\ x') \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}{\Gamma, \mathcal{S}, \mathcal{T} : \texttt{obs}_{\lambda}\ p\ b\ x\ e \Downarrow \Delta, \mathcal{S}', \mathcal{T}' : v}\ \text{Obs}_{\lambda}$$

Figure 17: Modified trace semantics for testing.

$$\begin{array}{lcll}
\text{expression } e & ::= & \ldots & \\
 & | & \texttt{push}\ l\ b\ e & \text{push label onto stack} \\
 & | & \texttt{obs}\ p\ b\ e & \text{observe expression} \\
 & | & \texttt{obs}_{\lambda}\ p\ b\ x\ e & \text{observe application} \\
 & | & b & \text{Boolean constant}
\end{array}$$

Figure 18: Modified syntax for testing.

$$\begin{aligned}
\llparenthesis f\ v_1, \ldots, v_n = v \rrparenthesis &= \llparenthesis v_1 \rrparenthesis \wedge \ldots \wedge \llparenthesis v_n \rrparenthesis \Rightarrow \llparenthesis v \rrparenthesis \\
\llparenthesis \smiley \rrparenthesis &= \smiley \\
\llparenthesis \frownie \rrparenthesis &= \frownie \\
\llparenthesis \_ \rrparenthesis &= \smiley \\
\llparenthesis \{m_1, \ldots, m_n\} \rrparenthesis &= \llparenthesis m_1 \rrparenthesis \wedge \ldots \wedge \llparenthesis m_n \rrparenthesis \\
\llparenthesis \backslash v_1 \ldots v_n \rightarrow v \rrparenthesis &= \llparenthesis v_1 \rrparenthesis \wedge \ldots \wedge \llparenthesis v_n \rrparenthesis \Rightarrow \llparenthesis v \rrparenthesis
\end{aligned}$$

Figure 19: Judging computation statements.

6.4 Defects Infect during Evaluation

Evaluation of a defective slice changes every ☺ into a ☹. So any constant becomes a ☹. Applying a defective function returns an infected result value,[5] irrespective of the argument.

However, we need to consider carefully how a defective slice causes an infection. Applying a defective function should still force evaluation of the argument, if that argument was demanded in the standard semantics. Hence we implement infection by piggybacking on the observation mechanism, which was designed not to influence computation.

Figure 18 shows our syntax for testing with the Booleans ☺ and ☹ as new constants, markers of labelled expressions and additional parameters of observations. Figure 17 shows the changes in the semantics. Constants are infected in the ObsC rule by taking the conjunction of the evaluation result b' and the correctness of the slice b. Here conjunction regards ☺ as true and ☹ as false. For abstractions we pass the slice's correctness to the observation of the result in the Obs$_\lambda$ rule.

6.5 Judging Computation Statements

From the trace we construct computation statements and their dependencies and subsequently transform the graph into a computation tree. Finally algorithmic debugging requires judging whether a computation statement is as intended or not. Usually the judgement is given by the user. Let us consider the program:

$$\begin{array}{l}
\texttt{let}\ \{\texttt{i} = 4; \texttt{dbl} = \texttt{push}\ \text{“}\textit{dbl}\text{”}\ (\lambda x.x)\} \\
\texttt{push}\ \text{“}\textit{main}\text{”}\ (\texttt{dbl i})
\end{array}$$

and assume that the function `dbl` is intended to double its argument. With our algorithmic debugger the programmer is asked the two questions:

```
main = 4   ? no
dbl 4 = 4  ? no
```

and the body of `dbl` is correctly identified as defective. The corresponding program for testing is:

$$\begin{array}{l}
\texttt{let}\ \{\texttt{i} = \smiley; \texttt{dbl} = (\texttt{push}\ \text{“}\textit{dbl}\text{”}\ \frownie\ (\lambda x.x)\} \\
\texttt{push}\ \text{“}\textit{main}\text{”}\ \smiley\ (\texttt{dbl i})
\end{array}$$

For testing the judgement is based on the infected and normal values appearing in the computation statement. Figure 19 defines the judgement function $\llparenthesis \cdot \rrparenthesis$. Again conjunction and implication regard ☺ as true and ☹ as false. If a function takes infected arguments, then these might violate the pre-condition of the function and hence we conservatively judge the function to meet intentions. The judgements for the computation tree of our example are:

$$\begin{aligned}
\llparenthesis \texttt{main = } \frownie \rrparenthesis &= \frownie \\
\llparenthesis \texttt{dbl } \smiley \texttt{ = } \frownie \rrparenthesis &= \frownie
\end{aligned}$$

which correctly identify the defective slice.

Computation statements for higher-order functions are often hard to judge. A benefit of our approach is that higher-order functions (often imported from libraries) are easy to trust. However, when the user writes their own higher-order functions, it can be necessary to annotate these functions for tracing. Now let us consider a program with a higher-order function:

$$\texttt{let}\ \{\texttt{i} = 2; \texttt{f} = \lambda x.x; \texttt{h} = \lambda f.\lambda x.f\ x\}\ \texttt{h f i}$$

Next we create an equivalent program for testing and mark `h` as defective. We obtain the following three computation statements

[5] In reality defective code does not cause an infection in every program run. However, if it does not cause an infection, then the program behaves as intended and no debugging will take place.

with their automatic judgements:

$$\begin{aligned} (\!|\texttt{main = } \frownie|\!) &= \frownie \\ (\!|\texttt{h } \{\backslash\smiley \rightarrow \smiley\}\ \smiley \texttt{ = } \frownie|\!) &= \frownie \\ (\!|\texttt{f } \smiley \texttt{ = } \smiley|\!) &= \smiley \end{aligned}$$

From these h is correctly identified as defective.

Let us now assume that h is correct and mark f as defective. Then the computation statements and automatic judgements that we obtain are:

$$\begin{aligned} (\!|\texttt{main = } \frownie|\!) &= \frownie \\ (\!|\texttt{h } \{\backslash\smiley \rightarrow \frownie\}\ \smiley \texttt{ = } \smiley|\!) &= \smiley \\ (\!|\texttt{f } \smiley \texttt{ = } \frownie|\!) &= \frownie \end{aligned}$$

which correctly identifies f as defective.

6.6 Quickcheck Properties

We implemented the modified semantics for testing in Haskell. Simplified our test property looks as follows:

```
sound :: Expr -> Property
sound e = valid result ==>
  property (defects e ‘anyElem‘ algDebug tree)
  where
  (result, tree) = mkTree (eval e)

anyElem :: Eq a => [a] -> [[a]] -> Bool
anyElem ys = foldr ((&&) . any (‘elem‘ ys)) True
```

Here algDebug tree is a list of lists of labels as each node that algorithmic debugging identifies as defective may contain several labels because of collapsing irreducible cycles when constructing the tree. We verified that the property holds for our semantics with over 100000 well-formed randomly generated expressions of up to 1000 subexpressions each.

7. Implementation

GHC allows programmatic access to the cost-centre stack of the profiling environment. Hence our implementation consists of a tracing library based on HOOD and an algorithmic debugger. The algorithmic debugger shows a webpage with the computation tree and allows the programmer to judge the computation statements in free order.

The language we present in this paper uses push to both add a label onto the stack and to observe the value of the enclosed expression. We implement the former with the GHC pragma Set Cost Centre (SCC) and the latter with the combinator observe from our tracing library. See for example the annotations of the isort function:

```
isort :: [Int] -> [Int]
isort = observe "isort" ({-# SCC "isort" #-}
        (foldr insert []))
```

Annotating a function is a straightforward and mechanical process. A compiler pass could annotate all top-level functions in a module.

In our semantics we did not discuss data constructors and exceptions, but the actual implementation handles these as well. The annotation functions can be derived for values of different types using the generic framework of Faddegon and Chitil [6].

With our implementation we successfully identified the defects in several examples programs that came with the Haskell Tracer Hat[6]. We also used our debugger for more complex programs. Our own debugger had a defect in the code that renders computation statements. We tracked down the defect with the debugger itself.

The game Raincat[7] is a mix of IO and pure computations. We introduced defects in Raincat and identified these with our debugger. While our debugger is most useful on pure computations, it proved no problem that the program also contained IO.

In all cases we were able to find the defect. In these experiments we typically added between 3 and 15 annotations. These produced computation graphs with 10 to several hundred computation statements. The number of questions that needed to be answered varied enormously. In almost all cases the defect could be found in less than 10 answers. However, in some cases it was hard to tell which questions to answer first: in the worst case hundreds of questions were asked before the defect was found. In future work we expect to address this by providing specialised annotations and search strategies to help the user with determining which question to answer first. In our current work we use algorithmic debugging as described by Shapiro; later variants were conceived with which we would like to experiment [22].

8. Related Work

Building computation trees to algorithmically debug programs written in a lazily evaluated functional language has received considerable attention.

Nilsson's Freja constructs computation trees using an instrumented runtime environment [13–15]. With Freja all modules are compiled specifically for debugging. The environment supports most of Haskell 98, but type classes and IO are missing.

Hat, its predecessor Redex Trails, and Buddha are based on program transformation [4, 17, 18, 23]. Every function is given an extra argument to pass information needed to connect various parts of the trace. Laziness, classes and higher-order functions make restricting such a transformation to a subset of the code hard.

To determine dependencies between computation statements Freja, Hat and Buddha transfer references to a location in the trace from one function application to another using either a transformation of the code or a specialized run-time implementation. Consider constructing a computation tree for the sorting program we used as example in the introduction. To add a dependency from the isort statement onto the insert statements, the trusted library function foldr either needs to be transformed (Hat and Buddha) or compiled specifically for the debugging environment (Freja). Neither recompiling nor transforming of trusted libraries is necessary with our approach, because we use the cost centre stack from GHC's standard profiling environment.

To minimise overhead of profiling, GHC does not record complete stacks but compresses stacks by truncating on recursion. Compressed stacks provide less information forcing us to approximate dependencies. Allwood et al. [1] suggest an alternative scheme of stack compression that maintains linear space overhead while providing greater precision by telling us where labels are dropped.

Gill [7] gives no semantics for HOOD. Pareja et al. [16] describe a variation of the Mark I machine from Sestoft [20]. They allow a tracer access to the state before and after each reduction step without changing the heap or expression, trivially ensuring that a traced program will produce the same result as the original one. They implement a simplified tracer based on HOOD.

An alternative to tracing is an interactive breakpoint-style debugger [10]. An interactive debugger is attractive, because its implementation is relative straightforward. However, exposure to evaluation order can be confusing. Furthermore, the debugger changes the behaviour of the program when the user requests to see the value of an otherwise unevaluated expression.

[6] http://projects.haskell.org/hat/

[7] http://bysusanlin.com/raincat/

9. Conclusions and Future Work

All previous work on algorithmic debuggers for Haskell required either a specialised run-time system or a transformation of all modules including libraries. Resulting tools are therefore of limited use for real-world Haskell programs. This paper describes a method to construct a computation tree for algorithmic debugging that needs only local annotations and GHC's profiling run-time system.

Thorough testing of our approach with randomly generated expression gives us confidence in the soundness of using our computation trees for algorithmic debugging. A formal proof and further case studies of real-world programs would reinforce this confidence. The semantics presented in this paper is a good starting point for a proof.

Recently we constructed a small higher-order program for which our debugger constructs an unexpected computation tree. We have to investigate this problem.

Using our debugger on real-world programs already demonstrated its value. However there is also room for improvement. To find defects in programs that use monads, it will be useful, if a sense of order is expressed in the computation tree. Cases where either a wide but shallow or a narrow but deep computation tree is constructed will benefit from using a starting point different from the tree root. Our prototype's free-browse mode allows to start elsewhere, but large trees can be overwhelming. In future work we plan to investigate searching for suspicious nodes.

Acknowledgments

We thank the anonymous reviewers for their thorough reviews and insightful comments.

References

[1] T. O. Allwood, S. Peyton Jones, and S. Eisenbach. Finding the needle: stack traces for GHC. In *Proceedings of the symposium on Haskell*, pages 129–140. ACM, 2009.

[2] O. Chitil. Source-based trace exploration. In *Proceedings of Implementation and Application of Functional Languages*, LNCS 3474. Springer, 2005.

[3] O. Chitil and T. Davie. Comprehending finite maps for algorithmic debugging of higher-order functional programs. In *Proceedings of the Principles and practice of declarative programming*. ACM, 2008.

[4] O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In *Implementation of Functional Languages*, LNCS 2670, pages 165–181. Springer, 2003.

[5] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the International Conference on Functional Programming*, volume 46, pages 268–279. ACM, 2000.

[6] M. Faddegon and O. Chitil. Type Generic Observing. In *Proceedings of Trends in Functional Programming*, LNCS 8843. Springer, 2014.

[7] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electronic Notes in Theoretical Computer Science*, 41, 2000. ACM SIGPLAN Workshop on Haskell.

[8] J. Launchbury. A natural semantics for lazy evaluation. In *Principles of programming languages*, pages 144–154. ACM, 1993.

[9] S. Marlow. Solving an old problem: How do we get a stack trace in a lazy functional language? Haskell Implementors Workshop, `http://community.haskell.org/~simonmar/Stack-traces.pdf`, 2012.

[10] S. Marlow, J. Iborra, B. Pope, and A. Gill. A lightweight interactive debugger for Haskell. In *Proceedings of the Haskell workshop*, pages 13–24. ACM, 2007.

[11] R. G. Morgan and S. A. Jarvis. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programming*, 8(3):201–237, 1998.

[12] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 3, 1997.

[13] H. Nilsson. *Declarative debugging for lazy functional languages*. PhD thesis, Linköpings universitet, 1998.

[14] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. In *Programming Language Implementation and Logic Programming*, pp. 385–399. Springer LNCS 631, 1992.

[15] H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2): 121–150, 1997.

[16] C. Pareja, R. Pena, F. Rubio, and C. Segura. Adding traces to a lazy monadic evaluator. In *Computer Aided Systems Theory—EUROCAST 2001*, pages 627–641. Springer LNCS 2178, 2001.

[17] B. Pope. Declarative Debugging with Buddha. In *Advanced Functional Programming*, pages 273–308. Springer LNCS 3622, 2005.

[18] B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

[19] P. M. Sansom and S. L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2):334–385, 1997.

[20] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[21] E. Y. Shapiro. *Algorithmic program debugging*. MIT press, 1983.

[22] J. Silva. *A comparative Study of Algorithmic Debugging Strategies*. In *Logic-Based Program Synthesis and Transformation*, pp 143-159. Springer LNCS 4407, 2007.

[23] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *Programming Languages: Implementations, Logics and Programs*, pages 291–308. Springer LNCS 1292, 1997.

[24] P. Wadler. Why No One Uses Functional Languages. *ACM SIGPLAN Notices*, 33(8):23–27, 1998. ISSN 0362-1340. doi: 10.1145/286385.286387. Functional programming column.

[25] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new Hat. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, 2001.

[26] A. Zeller. *Why Programs Fail, 2nd Edition*. Morgan Kaufmann, 2009.

[27] T. Zielonka and the GHC Team. `http://www.haskell.org/ghc/survey2005-summary`, 2005.

A. Appendix: Deriving Computation Trees for Programs with Higher-Order Functions

In the conclusion we wrote that we constructed a small higher-order program for which our debugger constructs an unexpected computation tree. The implementation of our debugger is more complex than the test set-up, after studying use-cases of programs producing very large traces we applied several optimizations. One of these optimizations introduced a defect in Hoed. In the version of Hoed we published on Hackage this problem is resolved. The semantics we present in this paper was always correct.

The program that showed the defect in Hoed is interesting in itself. Without defect a correct computation tree is produced that is different from the two variations of computation trees which are known to be sound for algorithmic debugging. In this appendix we explain why this specific tree is nonetheless sound for algorithmic debugging; and we explain how we did further testing to reinforce our trust that in general the computation trees derived with our method are sound for algorithmic debugging.

A.1 Example

Consider the following faulty program that unexpectedly prints ”oops”:

```
not :: Bool -> Bool
not b = case b of True -> True; False -> False

app :: (Bool->Bool) -> Bool -> Bool
app f b = f b

flip :: Bool -> Bool
flip b = app not b

main :: IO ()
main = print (if (flip False == True)
              then "ok!" else "oops!")
```

In this program the argument `f` of the function `app` is another function. The representation in a computation statement of an argument with a functional value can be intensional or extensional. The intensional representation of a functional value is a function symbol (e.g. “`not`”, or a partial application of a function symbol (e.g. “`and False`”). The extensional representation of a functional value is a finite map of arguments and results (e.g. “{`\True -> False;` `\False -> False`}”).

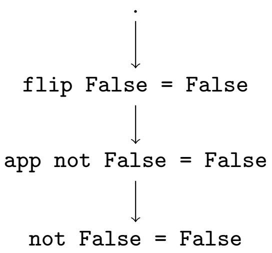
Previous work showed soundness for algorithmic debugging with either representation. However, depending on representation a computation tree should be structured differently. Assume that in the definition of a function g a higher order function h is applied to a function f.

- The Evaluation Dependency Tree (EDT) uses an intensional representation. The computation statement of h depends on the computation statement of f.
- The Function Dependency Tree (FDT) uses an extensional representation. The computation statement of g depends on the computation statement of f.

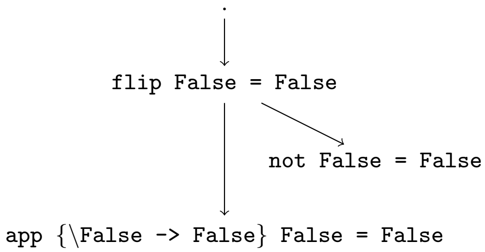
In both cases the computation statement of g depends on the computation statement of h.

An EDT and FDT have a different structure because a computation statement of a higher order function is judged different depending on the chosen representation. Try for example to judge the statements of our example program in the following two trees.

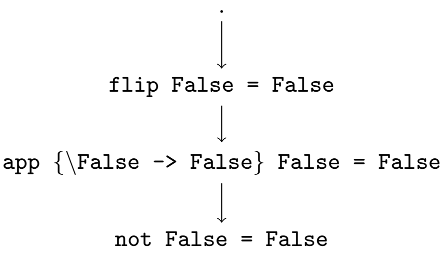
EDT of our example program:



FDT of our example program:



Mixing the structure of an FDT with the representation of an EDT is not sound in general. Consider for example the following tree:



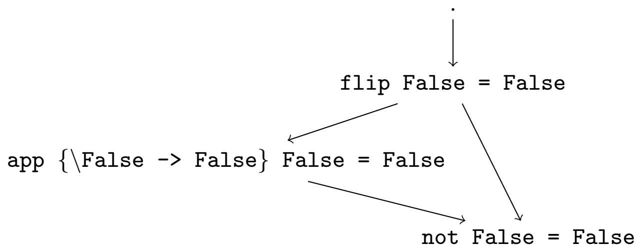
An algorithmic debugger using this tree could incorrectly conclude that the fault is in the definition of the function `flip` rather than in the definition of `not`:
`flip False = False` ? *no*
`app {\False -> False} False = False` ? *yes*
`Fault detected in function flip!`

It is trivial to show that any graph that is an over-approximation of either the EDT or the FDT is sound for algorithmic debugging. A graph that over-approximates a tree at least contains all arcs from that tree. A computation graph that we find with our method contains exactly the same nodes as the computation tree it approximates.

We represent a functional value as a finite map, therefore we know that our computation graphs are sound when they are an over-approximation of an FDT. A computation graph derived with our method is in most but not all cases an over-approximation of the FDT. In both cases they are sound. Let us consider the computation graphs for two variations of our example program. In the first case we annotate `app`, `not` and `flip` for tracing and find an over-approximation of the FDT. In the second case we only annotate `app` and `not` for tracing and find a tree that is not an over-approximation of an FDT but that is still sound.

Case 1: `flip`, `app` and `not` are annotated

Evaluating the example program with `flip`, `app` and `not` annotated produces a trace from which we derive the following computation tree:



This is an over-approximation of the FDT and thus sound for algorithmic debugging. Where does the edge `flip False = False` → `not False = False` come from? Let us take a closer look at the recorded cost centre stacks. We have:
$c_f =$ `flip False = False` with stack $\mathcal{S}_f = \langle\rangle$
$c_a =$ `app {\False -> False} False = False` with stack $\mathcal{S}_a = \langle \text{“}\mathit{flip}\text{”} \rangle$
$c_n =$ `not False = False` with stack $\mathcal{S}_n = \langle \text{“}\mathit{flip}\text{”}, \text{“}\mathit{app}\text{”} \rangle$
Two of the arcs are derived from potential stack push operations:

- $\mathcal{S}_f \lhd \text{“}\mathit{flip}\text{”} = \mathcal{S}_a$ therefore $c_f \to c_a$
- $\mathcal{S}_a \lhd \text{“}\mathit{app}\text{”} = \mathcal{S}_n$ therefore $c_a \to c_n$

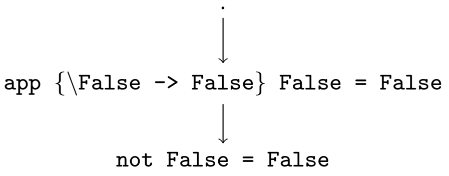
The arc $c_f \to c_n$ that makes it an over-approximation of an FDT is derived from a potential call operation:

- $(\mathcal{S}_a \lhd \text{“}\mathit{app}\text{”}) \bowtie (\mathcal{S}_f \lhd \text{“}\mathit{flip}\text{”}) = \mathcal{S}_n$ therefore $c_a \to c_f$ and $c_f \to c_n$

The edge $c_a \to c_f$ is not part of the computation tree because we remove back edges from the initial estimated computation graph to eliminate cycles (see Section 5.1).

Case 2: `app` and `not` are annotated

When we do not annotate `flip`, evaluation gives us a trace from which we derive the following tree:



This computation graph seems worrying because it does not over-approximate the FDT of this program. However, soundness of this specific tree is easily demonstrated.

The actual fault is in the `not` function definition. The `app`-statement is right. The `not`-statement is wrong and because the statement has no children in the computation tree, the statement is identified as faulty. Vice versa, if we assume `app` to be faulty and `not` to be correct then the `app` statement is wrong and its child is right. In all possible cases algorithmic debugging finds the actual faulty slice, therefore this specific tree is sound.

A.2 Testing More

The computation trees produced by our method for the example program are correct, but is this true in the general case? Testing indicates this, but after studying the trees from the example program we developed a concern about our testing method.

Nilsson already described how free variables and constants are a challenge for algorithmic debugging. Known solutions are to either sort by age or to explicitly track free variables. Our testing set-up uses age sorting. We implemented this such that first algorithmic debugging finds a list of possible faulty statements from which we then select the first evaluated. We only verify if this statement is actually faulty.

In theory this is a valid method, but determining which computation statement is evaluated before which other computation statement can be difficult. Algorithmic debugging with an unsound computation tree can find both an actual faulty slice and a slice that is incorrectly marked as faulty. We do not have any indication that this is the case in our implementation, but what if age sorting leads to a bias towards only verifying the actual faulty slice?

We added a modified variation of our testing set-up that tests if all slices that algorithmic debugging identifies are faulty are actually faulty. This is only true for expressions without variables that are free outside an annotated program slice. For example a program containing the following expression cannot be used for testing:

```
f = observe "f" (\x -> let k = 3
                           g = observe "g" (\y -> k)
                           h = observe "h" (\y -> y)
                       in (g k) + (h k))
```

That does not mean we cannot allow free variables at all. The following expression can be used to test our method:

```
f = observe "f" (\x -> let k = 3
                           g = observe "g" (\y -> y)
                           h = observe "h" (\y -> y)
                       in (g k) + (h k))
```

We believe the testing described in this paper, and verified by the PLDI Artifact Evaluation Committee, shows that our method is sound. Now we additionally tested with success that for more than 100000 well-formed expressions *all* slices identified as defective are indeed defective for expressions without variables that are free outside their respective program slices.