Lightweight Computation Tree Tracing for Lazy Functional Languages

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

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

Abstract

A computation tree of a program execution describes computations of functions and their dependencies. A computation tree describes how a program works and is at the heart of algorithmic debugging. To generate a computation tree, existing algorithmic debuggers either use a complex implementation or yield a less informative approximation. We present a method for lazy functional languages that requires only a simple tracing library to generate a detailed computation tree. With our algorithmic debugger a programmer can debug any Haskell program by only importing our library and annotating suspected functions.

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

Keywords algorithmic debugging, lazy evaluation, tracing

1. Introduction

Consider the defective Haskell implementation of a parity function in Figure 1. The program includes a property `prop_notBothOdd` for testing. Using the property-based testing tool QuickCheck [9] we get:

```
> quickCheck prop_notBothOdd
*** Failed! Falsifiable (after 1 test): 2
```

So for argument value 2 the property does not hold.

Figure 2 shows a computation tree for evaluating the expression `quickCheck prop_notBothOdd`. The special root node ★ connects two subtrees. All other nodes of a computation tree are computation statements. A computation statement is usually a function (identifier) applied to argument values together with a result value. A computation statement describes a subcomputation of the entire computation of the program. In the tree a node is the parent of a child node, if and only if the computation of the child contributes to the computation of the parent. More precisely, the tree structure must have the following property of algorithmic debugging: We call a computation statement right, if it agrees with the intentions/expectations/specification of the programmer for the program. Otherwise we call a computation statement wrong. If a parent node is wrong but all its child nodes are right, then the definition of the function appearing in the parent node must be defective. In our example `modTwo 4 = 2` is wrong and because that node has no children, the definition of `modTwo` must be defective, as indeed it is.

The function of a child node is not necessarily called by the function of its parent node, although that is often the case. Firstly, not all functions that contribute to an entire computation have to appear in a computation tree. Usually a computation tree contains only nodes for functions that the programmer suspects of being defective; hence our example tree has no nodes for, e.g., `(+)` or `prop_notBothOdd`. Secondly, for higher-order functions at least two different def-

```
isOdd n = isEven (plusOne n)
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = div n 2

prop_notBothOdd :: Int -> Bool
prop_notBothOdd x = isOdd x /= isOdd (x+1)
```

Figure 1. A defective program with a test property.

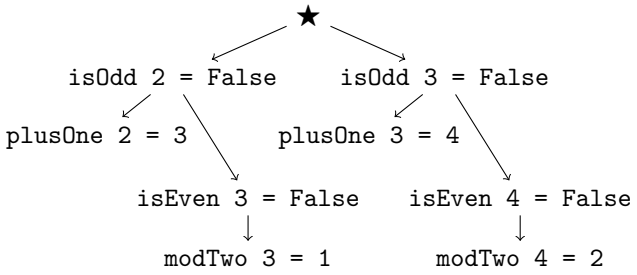

Figure 2. Computation tree for `prop_notBothOdd 2`.

[To appear in the proceeding of PLDI 2016.]

initions for the parent-child relation of a computation tree exist (cf. Section 6.1).

The ultimate goal of our research is to provide better support for debugging lazy functional programs, which is very needed [29, 32]. A computation tree is a key means for understanding how a program works or why it does not work. A computation tree is at the heart of the algorithmic debugging method [26, 31].

In this paper we present a lightweight method for obtaining a computation tree for a computation of a lazy functional program. The method is lightweight in that it does not require any complex implementation infrastructure such as a modified compiler or runtime system. Note that the runtime stack cannot be used to determine computation statement is the child of which other computation statement: the runtime stack of a lazily evaluated language relates to demand and not the nesting of function calls; even for an eagerly evaluated language the runtime stack differs for higher-order functions from our desired parent-child relation. Furthermore, our method does not require any changes to trusted libraries and without additional work it supports a large set of language features. Thus we can use our implementation for debugging any Haskell program.

We start with Andy Gill's lightweight value observation technique for debugging Haskell programs [15]. Its implementation Hood is just a library. The programmer imports the library in their program and annotates all expressions of interest with Hood's `observe` function. Hood guarantees that the input-output behaviour of the executed program remains unchanged, but after termination, Hood shows for every annotated expression all the values the expression had during the computation. Hood reconstructs these values from a value observation trace, a simple sequence of events, that Hood creates as a side-effect during the computation.

We can use Hood to obtain computation statements by annotating functions. In this paper we show that the value observation trace contains more information than previously thought: from the value observation trace we can also reconstruct the parent-child relation between computation statements for a computation tree.

Consider annotating functions in our defective program as shown in Figure 3.[1] Figure 4 shows the simplified value observation trace created by Hood when evaluating the expression `quickCheck prop_notBothOdd`. The trace is a sequence of events, written in the order in which the program is evaluated. There are two main types of events: request and corresponding response events. When evaluation of an expression starts, a request event is recorded (the value of the expression is requested). When evaluation of an expression ends, a response event records the value of the expression. We call the pair of a corresponding request and response event a request-response span.

[1] The first argument of `observe` can be an arbitrary String, but we use the function name to have it in the trace for constructing the computation tree.

```
isOdd = observe "isOdd" isOdd'
isOdd' n = isEven (plusOne n)

isEven = observe "isEven" isEven'
isEven' n = modTwo n == 0

plusOne = observe "plusOne" plusOne'
plusOne' n = n + 1

modTwo  = observe "modTwo" modTwo'
modTwo' n = div n 2

prop_notBothOdd :: Int -> Bool
prop_notBothOdd x = isOdd x /= isOdd (x+1)
```

Figure 3. Defective program with observation annotations.

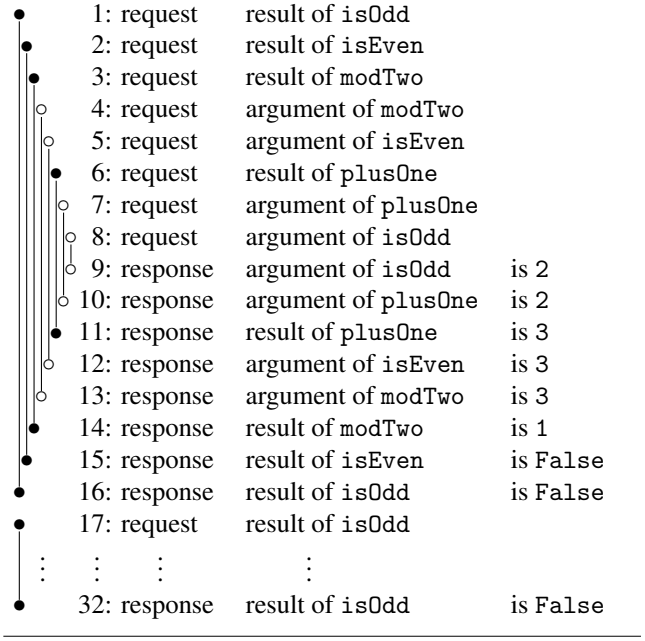

Figure 4. Simplified trace for `prop_notBothOdd 2`.

The correspondence between span nesting and the parent-child relation is not trivial: The application of an observed function usually has a request-response span for both its result and its argument. In Figure 4 on the left the request-response spans for function results are marked with •-brackets, whereas the request-response spans for function arguments are marked with ∘-brackets. Each event has a unique index i. The result span $\langle 6, 11\rangle$ for `plusOne` is nested within the result spans of all the other functions. However, it is also nested within the argument spans $\langle 5, 12\rangle$ and $\langle 4, 13\rangle$ for `isEven` and `modTwo`. Within these argument spans the computations for the two functions `isEven` and `modTwo` are suspended; the events inside these spans are actually for the computation of the result of `isOdd` and hence the computation statement for `plusOne` has to be a child of the computation statement of `isOdd`. Overall we obtain from the value observation trace the computation tree of Figure 2.

Our key observation is that the events of a value observation trace are organised in nested request-response spans and whether a computation statement is the parent of another computation statement follows from the nesting of the various spans forming the computation statements. We make the following contributions:

- We give a semantics that defines the value observation trace and how it is created by Hood (Section 2).
- We explain the correspondence between the nesting of request-response spans and the parent-child relation of computation statements. Afterwards we give an algorithm for generating a computation tree from Hood's value observation trace (Section 3).
- We implement our method in the new algorithmic debugger Hoed-pure (Section 4.1). Our debugger is available from the Haskell package archive Hackage:

  ```
  cabal install Hoed
  ```

  We describe our experience of finding defects in several real-world Haskell programs (Section 7).

2. Creating a Value Observation Trace

Gill invented the technique for obtaining a value observation trace from a computation and implemented it in the Haskell library Hood [15]. Faddegon and Chitil [12] gave a first formal definition of Gill's technique by extending Launchbury's lazy semantics [17] with observation tracing. That definition is insufficient here, because it omits the request events that Hood provides. Request events are unnecessary for reconstructing computation statements from the trace but they are essential for our new method of reconstructing from the trace the parent-child relation for the computation tree.

Furthermore, here we cover a larger language than Faddegon and Chitil. We include exceptions in our language, because in practice defective programs often raise exceptions and when constructing the computation tree later we assume that in the trace every request is followed by a corresponding response, which might be an exception.

2.1 Language

We define tracing for the language given in Figure 5. The basis is Launchbury's core language together with his data constructors and primitive operations. Our language includes integer values; they are constructors of arity zero. An exception is also just the constructor `Exception`. To make heap allocation explicit, Launchbury requires the arguments of applications, data constructors and primitive operations to be variables. A language without this argument restriction can easily be translated into the core by inserting let-bindings [17]. Thus the language covers all examples in this paper.

We extend Launchbury's language with expressions for observing values. The programmer uses `observe` to annotate expressions that they want to observe. The `obs` expressions and $\texttt{obs}_\lambda$ values should not appear in a program. Rather they are introduced by evaluation of an `observe` expressions. A single applied data constructor or a single λ-abstraction may be observed several times; the latter case leads to values such as $\texttt{obs}_\lambda\ p_1\ (\texttt{obs}_\lambda\ p_2\ (\ldots(\lambda x.e)\ldots))$ during evaluation.

Our semantics scales to the many different expressions in Haskell, because we observe only values and Haskell has only few different sorts of values.

2.2 Value Observation Trace

A trace is a sequence of events as defined in Figure 6. The events are written in the order in which the program is evaluated. Each event has a unique event number i, which is its index in the trace. There are two main types of events: request and corresponding response events. When evaluation of an expression starts, a request event is recorded (the value of the expression is requested). When evaluation of an expression ends, a response event records the value of the expression. Our semantics will ensure for a trace that every request event has a later corresponding response event, which may be an exception.

Every event except for $i : \text{Root}\ f$ has a field p, which identifies its parent event and its particular role as child of that parent event. Note that this parent/child terminology of events is taken from Hood [15]. As we will see, these parents and children express the relation between expressions and their subexpressions; they are unrelated to the parent/child structure of nodes of the computation tree.

An $i : \text{Root}\ f$ event records the function identifier supplied by the expression $\texttt{observe}\ f\ e$. The event $i : \text{Enter}\ p$ expresses the request for the value of an expression. There are two possible response events: $j : \text{Con}\ p\ c\ a$ and $j : \text{Lam}\ p$. The former expresses that the value is a saturated application of a constructor c of arity a, the latter expresses that the value is a function, a λ-expression. A constructor event $j : \text{Con}\ p\ c\ a$ may be the parent of up to a children, each with a parent $\text{P}_\text{c}\ j\ m$ where $1 \leq m \leq a$.

Functional values are recorded extensionally, as a finite map from arguments to results. Hence an $i : \text{Lam}\ p$ event may have an arbitrary number of $j : \text{MapsTo}\ p$ events as children. Each $j : \text{MapsTo}\ p$ event describes a pair of an argument and a result. Note the difference in structure: an application expression $e\,x$ consists of a function e and an argument x; the whole expression evaluates to some result. In contrast, an $j : \text{MapsTo}\ p$ event may have an argument child with parent $\text{P}_\text{a}\ j$ and a result child with parent $\text{P}_\text{r}\ j$; its parent is the function that was applied.

Overall, most events can have children, but, because lazy evaluation may not evaluate some function or data constructor arguments, some events do not necessarily have these children.

2.3 Semantics

Figure 7 defines the semantics of our language. A heap Γ is a finite map from variables to expressions. The relation

expression	e	$::= v$	
		$\mid e\ x$	application
		$\mid \texttt{let}\ \{x_k = e_k\}_{k=1}^{n}\ \texttt{in}\ e$	recursive binding; variables x_k are bound in any e_k and e
		$\mid \texttt{case}\ e\ \texttt{of}\ \{c_k\ x_1 \ldots x_{m_k} \to e_k\}_{k=1}^{n}$	case; bound variables $x_1, \ldots, x_{m_k}$ may appear in e_k
		$\mid x$	variable
		$\mid x_1 \oplus x_2$	application of a primitive operator such as + or ==
		$\mid \texttt{observe}\ f\ e$	label expression with function identifier f and observe it
		$\mid \texttt{obs}\ p\ e$	observe expression
value	v	$::= c\ x_1 \ldots x_n$	saturated application of data constructor c of arity n
		$\mid v_\lambda$	functional value
functional value	v_λ	$::= \lambda x.e$	λ-abstraction; variable x is bound in e
		$\mid \texttt{obs}_\lambda\ p\ v_\lambda$	observed functional value

Figure 5. Syntax of the core language.

trace	$\mathcal{T}$	$::= t_0, \ldots, t_n$	sequence growing right
event number	i	$\in \{0, \ldots, n\}$	refers to an event in the trace
trace event	t	$::= i{:}\text{Root}\ f$	root with function identifier f
		$\mid i{:}\text{Enter}\ p$	enter evaluating expression — request
		$\mid i{:}\text{Con}\ p\ c\ a$	value is saturated application of data constructor c with arity $a \in \{0, 1, \ldots\}$ — response
		$\mid i{:}\text{Lam}\ p$	value is an abstraction — response
		$\mid i{:}\text{MapsTo}\ p$	pair of argument and result of a function application
parent	p	$::= \text{P}\ i$	parent is event $i{:}\text{Root}\ f$ or $i{:}\text{Lam}\ p'$
		$\mid \text{P}_\text{c}\ i\ m$	argument m; parent is constructor event $i{:}\text{Con}\ p'\ c\ a$ with $m \le a$
		$\mid \text{P}_\text{a}\ i$	argument; parent is an event $i{:}\text{MapsTo}\ p'$
		$\mid \text{P}_\text{r}\ i$	result; parent is an event $i{:}\text{MapsTo}\ p'$

Figure 6. Syntax of the trace and its events.

$$\Gamma, \mathcal{T} : \lambda x.e \Downarrow \Gamma, \mathcal{T} : \lambda x.e \quad \text{Lam}$$

$$\Gamma, \mathcal{T} : c\ x_1 \ldots x_n \Downarrow \Gamma, \mathcal{T} : c\ x_1 \ldots x_n \quad \text{Con}$$

$$\frac{\Gamma, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : \lambda x.e' \qquad \Gamma', \mathcal{T}' : e'[y/x] \Downarrow \Gamma'', \mathcal{T}'' : v}{\Gamma, \mathcal{T} : e\ y \Downarrow \Gamma'', \mathcal{T}'' : v}\ \text{App}$$

$$\frac{\Gamma, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : v}{\Gamma[x \mapsto e], \mathcal{T} : x \Downarrow \Gamma'[x \mapsto v], \mathcal{T}' : \hat{v}}\ \text{Var}$$

$$\frac{\Gamma, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : c_k\ x_1 \ldots x_{m_k} \qquad \Gamma', \mathcal{T}' : e_k[x_i/y_i]_{i=1}^{m_k} \Downarrow \Gamma'', \mathcal{T}'' : v}{\Gamma, \mathcal{T} : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \Downarrow \Gamma'', \mathcal{T}'' : v}\ \text{Case}$$

$$\frac{\Gamma[x_i \mapsto e_i]_{i=1}^{n}, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : v}{\Gamma, \mathcal{T} : \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e \Downarrow \Gamma', \mathcal{T}' : v}\ \text{Let}$$

$$\frac{\Gamma, \mathcal{T} : e_1 \Downarrow \Gamma', \mathcal{T}' : v_1 \qquad \Gamma', \mathcal{T}' : e_2 \Downarrow \Gamma'', \mathcal{T}'' : v_2}{\Gamma, \mathcal{T} : e_1 \oplus e_2 \Downarrow \Gamma'', \mathcal{T}'' : v_1 \underline{\oplus} v_2}\ \text{Prim}$$

$$\frac{\Gamma, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : v \qquad \texttt{notAbs}\ v}{\Gamma, \mathcal{T} : e\ x \Downarrow \Gamma', \mathcal{T}' : \texttt{Exception}}\ \text{EApp}$$

$$\frac{\Gamma, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : v \qquad \texttt{notCon}\ v\ \{c_i\}_{i=1}^{n}}{\Gamma, \mathcal{T} : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \Downarrow \Gamma', \mathcal{T}' : \texttt{Exception}}\ \text{ECase}$$

$$\frac{\Gamma, \mathcal{T} \lessdot (i{:}\text{Root}\ f) : \texttt{obs}\ (\text{P}\ i)\ e \Downarrow \Gamma', \mathcal{T}' : v \qquad i = |\mathcal{T}|}{\Gamma, \mathcal{T} : \texttt{observe}\ f\ e \Downarrow \Gamma', \mathcal{T}' : v}\ \text{Observe}$$

$$\frac{\Gamma, \mathcal{T} \lessdot (i{:}\text{Enter}\ p) : e \Downarrow \Gamma', \mathcal{T}' : c\ x_1 \ldots x_n \qquad i = |\mathcal{T}| \qquad j = |\mathcal{T}'|}{\Gamma, \mathcal{T} : \texttt{obs}\ p\ e \Downarrow \Gamma'[y_1 \mapsto \texttt{obs}\ (\text{P}_\text{c}\ j\ 1)\ x_1, \ldots, y_n \mapsto \texttt{obs}\ (\text{P}_\text{c}\ j\ n)\ x_n], \mathcal{T}' \lessdot (j{:}\text{Con}\ p\ c\ (\text{arity}\ c)\ ) : c\ y_1 \ldots y_n}\ \text{ObsCon}$$

$$\frac{\Gamma, \mathcal{T} \lessdot (i{:}\text{Enter}\ p) : e \Downarrow \Gamma', \mathcal{T}' : v_\lambda \qquad i = |\mathcal{T}| \qquad j = |\mathcal{T}'|}{\Gamma, \mathcal{T} : \texttt{obs}\ p\ e \Downarrow \Gamma', \mathcal{T}' \lessdot (j{:}\text{Lam}\ p) : \texttt{obs}_\lambda\ (\text{P}\ j)\ v_\lambda}\ \text{ObsLam}$$

$$\frac{\Gamma, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : \texttt{obs}_\lambda\ p\ v_\lambda \qquad \Gamma'[y \mapsto \texttt{obs}\ (\text{P}_\text{a}\ j)\ x], \mathcal{T}' \lessdot (j{:}\text{MapsTo}\ p) : \texttt{obs}\ (\text{P}_\text{r}\ j)\ (v_\lambda\ y) \Downarrow \Gamma'', \mathcal{T}'' : v \qquad j = |\mathcal{T}'|}{\Gamma, \mathcal{T} : e\ x \Downarrow \Gamma'', \mathcal{T}'' : v}\ \text{ObsApp}$$

Figure 7. A natural semantics for lazy evaluation with generation of a trace.

$\Gamma, \mathcal{T} : e \Downarrow \Gamma', \mathcal{T}' : v$ states that the expression e with heap Γ and trace $\mathcal{T}$ evaluates to the value v with heap Γ' and trace $\mathcal{T}'$. A computation starts with an empty heap and empty trace.

The seven rules at the top (Lam, Con, Var, Let, App, Case and Prim) are almost identical to rules with the same names in Launchbury's semantics [17]. Like Launchbury we require that all bound variables of an expression are distinct. The $\hat{v}$ in the Var rule indicates that all bound variables in v are renamed to fresh ones. For y_1 to y_n in ObsCon and y in ObsApp we also pick fresh variables. In the Prim rule $\underline{\oplus}$ is the total semantic function associated with the syntactic operator $\oplus$.

The rules ECase and EApp define basic exceptions. An exception is just a constructor `Exception` and thus already handled by most rules. The exception rules just ensure that if a computation with the other rules would get "stuck", then it evaluates to an exception. The expression $\texttt{notCon}\, v\, \{c_i\}_{i=1}^n$ in the ECase rule is true iff v is not a constructor application such that the constructor occurs in the set $\{c_i\}_{i=1}^n$. Similarly $\texttt{notAbs}\, v$ in the EApp rule is true iff v is not an abstraction. The constructor `Exception` may appear in a program but should not appear in a pattern of a case expression to catch an exception, as this would substantially change the equational theory of the language; however, this would still not affect tracing itself.

The only tracing-specific extension in all the previous rules is that they thread the trace as an additional global state through the computation .

Finally consider the rules that actually construct the trace. $t_0, \ldots, t_n \lessdot t = t_0, \ldots, t_n, t$ 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.

The Observe rule records an $i : \text{Root}\ f$ event and wraps the observed expression with the pseudo-function `obs`. The index of the i : Root f event is passed to `obs` to enable connecting to the parent event later. Before reducing e in $\texttt{obs}\ p\ e$ the ObsLam and ObsCon rules add the request event $i : \text{Enter}\ p$ to the trace. When e is reduced to a value, this value is also recorded in the trace with the same parent p. Thus the trace records the request-response spans that we discussed in the Introduction.

For an application of a constructor $c\ x_1 \ldots x_n$ the ObsCon rule adds a Con event and continues observing the arguments $x_1, \ldots, x_n$ of the constructor using the pseudo function `obs`.

For a functional value v_λ the ObsLam rule adds an event $i : \text{Lam}\ p$ to the trace. For every application of the resulting observed functional value $\texttt{obs}_\lambda\ (\text{P}\ j)\ v_\lambda$ the ObsApp rule adds an event $k : \text{MapsTo}\ (\text{P}\ j)$ to the trace and continues observing the argument and result using the pseudo function `obs`.

So only when evaluation reaches a constructor application that is recorded in the trace. When that constructor application is destructed by a `case` expression, nothing is recorded in the trace. In contrast, when evaluation reaches a functional value that is recorded in the trace and whenever that functional value is applied to an argument, the pair of argument and result are recorded in the trace. We have this asymmetry, because our syntax uses a saturated constructor application as value, which contains a constructor and its arguments; in contrast, a functional value can be applied to an arbitrary number of arguments in a computation.

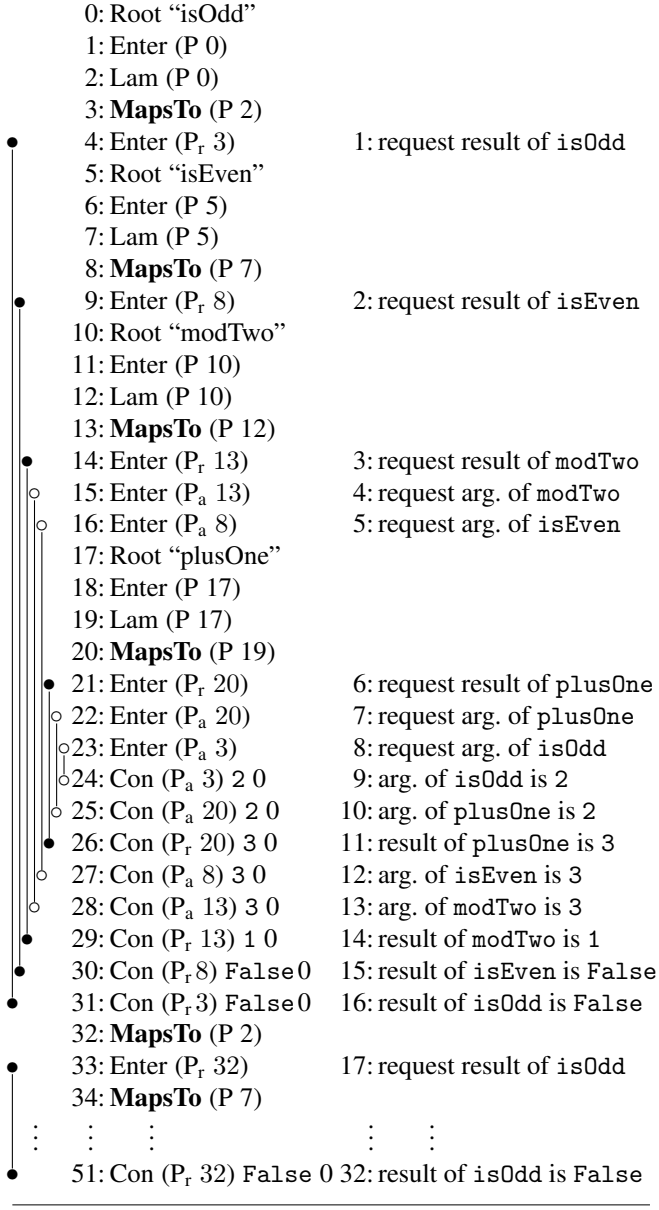

Figure 8. Full trace with corresponding simplified events.

2.4 A Trace

If our introductory example is annotated as in Figure 3, then the semantics gives us the trace shown in Figure 8. On

the right side the simplified trace of Figure 4 is given for comparison.

There is a one-to-one correspondence between function calls (of observed functions) during the computation and MapsTo events in the trace. MapsTo events that have the same parent record applications of the same function. For example, the two events 3 : MapsTo (P 2) and 32 : MapsTo (P 2) have the same parent. They are both recordings of calling the function `isOdd`.

In the remainder of the paper we assume the existence of a trace $\mathcal{T}$ of a computation.

2.5 Request-Response Spans

Only the ObsLam rule and the ObsCon rule add request and response events to the trace. Each rule introduces a pair of a request and a response event. For the trace threaded through the whole computation each of these rules adds a request event to the trace coming in and adds a response event just before passing the trace out. Hence these events always appear as pairs in a trace and they appear in sequence or nested, like parentheses in the language of balanced parentheses. In the following we call such a pair a *request-response span* and write it as $\langle i, j\rangle$ where i and j are the numbers of the request and, respectively, response event.

Because request-response spans are like balanced parantheses, we can easily determine for each request event its corresponding response event through a sequential traversal of the trace from beginning to end.

The ObsLam and ObsCon rule also guarantee the following invariant: the request and the response event of a request-response span have the same parent. Because a sequential traversal of the trace easily determines for each requeust event its corresponding response event, it is not actually necessary for response events to have parents at all. However, we include parents in response events, because Hood does so, it simplifies some algorithms, and it allows additional sanity checks in our implementation.

Request-response spans are the key to constructing a computation tree from a trace. In Figure 8 nearly all request-response spans are marked on the left side with vertical lines terminated by • or ◦. Trivial spans that directly follow a Root event, such as $\langle 1, 2\rangle$ and $\langle 6, 7\rangle$, are not marked, because we do not need trivial spans for constructing a computation tree.

3. From Trace to Computation Tree

We now have a precise definition of the value observation trace and have to obtain from it a computation tree. In the following we assume that we observe only top-level variables bound to λ-abstractions, such as `isOdd`, which is bound to `λn. isEven (plusOne n)` in Figure 1. We discuss the reasons for this restriction in Section 5.1.

We first discuss how we construct the nodes of the computation tree. Afterwards we discuss how we construct the edges, that is, the parent-child relation of the computation tree.

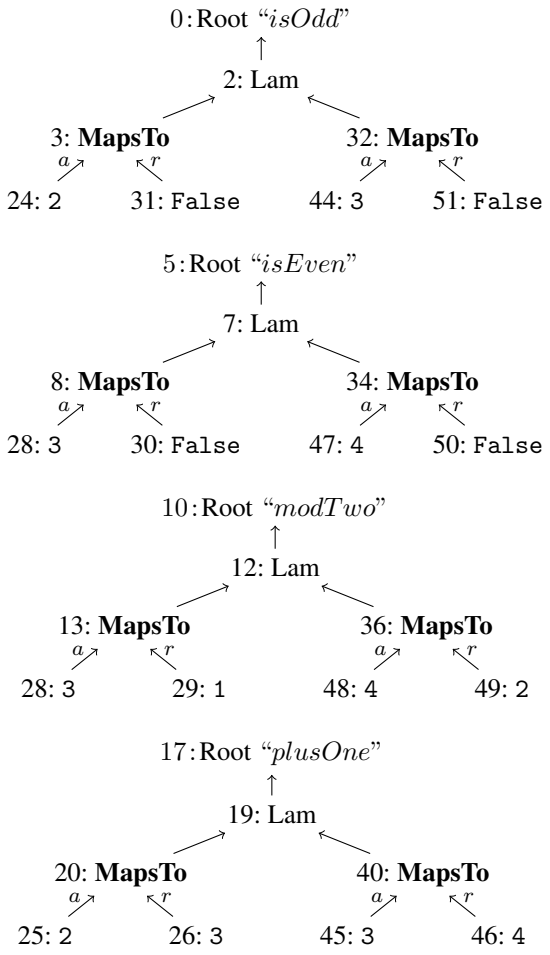

Figure 9. Trace of Figure 8 shown as forest of event trees.

3.1 Event Trees

We construct the nodes of the computation tree in two steps: First we translate the event trace into a forest of event trees. Subsequently we translate the forest into nodes of the computation tree. Note that an *event* tree and a *computation* tree are two very different structures.

The nodes of an event tree are events. Enter events are not needed for constructing the nodes of a computation tree. Every event of the event trace that is not an Enter event becomes a node in an event tree. The edges of an event tree are determined by the parent fields of the events: An event with parent P i, $\mathrm{P_c}\ i\ a$, $\mathrm{P_a}\ i$ or $\mathrm{P_r}\ i$ has the event with number i as parent. Therefore every Root event of the event trace becomes the root of an event tree.

Figure 9 shows the four event trees that we obtain from the trace of Figure 8. Because parent fields determine the tree edges, we do not include them in the tree nodes. The argument event of a MapsTo event is marked with a and the result event is marked with r. Constructor events are abbreviated to show only the constructor name.

In our example each observed function is applied exactly twice in the traced computation. Therefore each Lam node has exactly two MapsTo nodes as children.

From the semantic rules of Figure 7 we obtain the following properties of an event tree:

- An i : Root f node has exactly one child (Observe). Because we observe only variables bound to λ-abstractions, that child is a Lam node.
- An i : Lam p node has MapsTo nodes as children. There are zero or more such children. All children have the same parent P i (ObsLam).
- An i : MapsTo p node has at most one child with parent $\mathrm{P_a}\ i$ and one child with parent $\mathrm{P_r}\ i$ (ObsApp).
- An i : Con $p\ c\ a$ node has at most one child with parent $\mathrm{P_c}\ i\ k$ for every $k \in \{1, \ldots, a\}$ (ObsCon). Recall that a is the arity of the constructor c.

Because of lazy evaluation, some expressions are never evaluated and hence certain children may not exist in the trace.

Because we observe only top-level variables, which are evaluated at most once, each observation yields at most one event tree. Removing an observation from the program leads to removing its corresponding event tree from the forest of event trees. The remaining events of the trace would have different indices but appear in unchanged order.

3.2 Constructing the Nodes of the Computation Tree

The nodes of the computation tree are computation statements. Figure 10 defines the syntax of computation statements. A computation statement is a function identifier plus a singleton map. A singleton map maps an argument value to a result value. A value can be unknown when lazy evaluation did not require its evaluation, the saturated application of a constructor to values, or a functional value. A functional value is represented extensionally as a finite map from arguments to results. Hence we define it as a finite set of singleton maps.

The algorithm of Figure 11 constructs computation statements from an event tree. We write et_p for a subtree of an event tree that has a root node with parent p. As the last equation emphasises, because of lazy evaluation such a subtree can be empty.

For every MapsTo event that is a grandchild of a Root event we construct a computation statement. So there is a one-to-one relation between the nodes in the computation tree and the MapsTo events whose grandfather is a Root. So from the eight MapsTo events of Figure 9 we obtain the eight computation statements

```
  isOdd = 2 ↦ False      isOdd = 3 ↦ False
 isEven = 3 ↦ False     isEven = 4 ↦ False
 modTwo = 3 ↦ 1         modTwo = 4 ↦ 2
plusOne = 2 ↦ 3        plusOne = 3 ↦ 4
```

statement	s ::= f = a	
singleton map	b ::= $w \mapsto w$	
statement value	w ::= _	unknown
	$\mid\ c\ w_1 \ldots w_n$	$n =$ arity c
	$\mid\ \{b_1, \ldots, b_k\}$	functional value

Figure 10. Abstract syntax of computation statements.

$$\mathrm{mkStmts}\ \left(\begin{array}{c} i\text{:Root } f \\ \uparrow \\ j : \mathrm{Lam} \\ \nearrow \quad \cdots \quad \nwarrow \\ et_{(\mathrm{P}\ j)} \quad\quad et_{(\mathrm{P}\ j)} \end{array}\right) =$$

$$\{f = \mathrm{mkSMap}\ et_{(\mathrm{P}\ j)}, \ldots, f = \mathrm{mkSMap}\ et_{(\mathrm{P}\ j)}\}$$

$$\mathrm{mkSMap}\ \left(\begin{array}{c} i : \mathrm{MapsTo} \\ a \nearrow \quad\quad \nwarrow r \\ et_{(\mathrm{P_a}\ i)} \quad\quad et_{(\mathrm{P_r}\ i)} \end{array}\right) =$$

$$\mathrm{mkVal}\ et_{(\mathrm{P_a}\ i)} \mapsto \mathrm{mkVal}\ et_{(\mathrm{P_r}\ i)}$$

$$\mathrm{mkVal}\ \left(\begin{array}{c} i : \mathrm{Con}\ c\ \ a \\ 1 \nearrow \quad\quad \nwarrow a \\ et_{(\mathrm{P_c}\ i\ 1)} \quad \cdots \quad et_{(\mathrm{P_c}\ i\ a)} \end{array}\right) =$$

$$c\ (\mathrm{mkVal}\ et_{(\mathrm{P_c}\ i\ 1)}, \ldots, \mathrm{mkVal}\ et_{(\mathrm{P_c}\ i\ a)}$$

$$\mathrm{mkVal}\ \left(\begin{array}{c} i : \mathrm{Lam} \\ \nearrow \quad\quad \nwarrow \\ et_{(\mathrm{P}\ i)} \quad \cdots \quad et_{(\mathrm{P}\ i)} \end{array}\right) =$$

$$\{\mathrm{mkSMap}\ et_{(\mathrm{P}\ i)}, \ldots, \mathrm{mkSMap}\ et_{(\mathrm{P}\ i)}\}$$

$$\mathrm{mkVal}\ (\textit{empty event tree}) =$$

$$\_$$

Figure 11. From event tree to computation statements.

In practice a debugger may introduce some syntactic sugar for nodes of a computation tree. For example, a function argument can be moved to the left side of the equals sign. Also repeated singleton maps in a functional value can be omitted.

3.3 Argument and Result Spans

Request-response spans are the key to constructing the edges of the computation tree, that is, determining the parent-child relation between computation statements. In the sequential value observation trace every request event, that is,

an i : Enter p event, is sooner or later followed by a corresponding response event, that is an i : Con p c a or i : Lam p event. To determine request-response events and their nesting structure we again focus on the sequential structure of the value observation trace. The forst of event trees that gave us the computation statements is now of less importance.

In this paper we ignore the trivial $\langle i, (i+1) \rangle$ spans with i : Enter p and $i+1$: Lam p that follow each $(i-1)$: Root event.

The forest of event trees tells us for every response event to which computation statement it belongs. Similarly we say for its corresponding request event and even the whole request-response span that they belong to the same computation statement. So every computation statement has one or more request-response spans.

Because every computation statement is of the form f `=` $w_a \mapsto w_r$ and any request event belonging to it either belongs to the argument value w_a or the result value w_r, we can divide the spans of a computation statement into *argument spans* and *result spans*. A value can have more than one span; for example the value `(3,4)` has three spans: for the constuctor `(,)` and for each of the integers 4 and 4. Because of lazy evaluation the argument of a function may never be evaluated; hence we conclude that a computation statement can have one or more result spans and zero or more argument spans.

So how do these argument and result spans determine the parent-child relation between computation statements? As outlined in the introduction, a computation statement is a child, if it contributes to the computation of its parent. Here "contribution" is defined by the fault-localisation property of algorithmic debugging: If a parent computation statement f `=` $w_a \mapsto w_r$ is wrong but all its child computation statements are correct, then the definition of function f must be defective.

A result span of a computation statement encloses events that record computation activity of that very computation statement. So when a result span of a computation statement s_1 is directly nested in the result span of a computation statement s_2, then s_1 is a child of s_2.

3.4 Positive and Negative Spans

Because Haskell is lazily evaluated, function arguments are not evaluated before a function call but only when needed during the evaluation of the called function. Hence, an argument span encloses events that record computation activity that did not contribute to the computation statement of the span. Instead, that computation activity has to be attributed to the function that passed the argument in its definition. In the following we call a span of a computation statement *positive*, if the events nested in the span contribute to the computation statement and *negative*, if they do not.

Because our language is higher-order, not every argument span is negative and not every result span positive.

```
let { i = observe "i" (λz.z),
      f = observe "f"
            (λg.let {x = 42, y = i x} in g y),
      h = λu.u
    } in f h
```

Figure 12. A higher-order program.

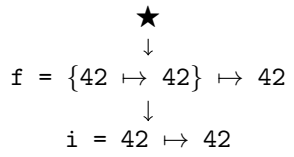

Figure 13. Computation tree for the higher-order program.

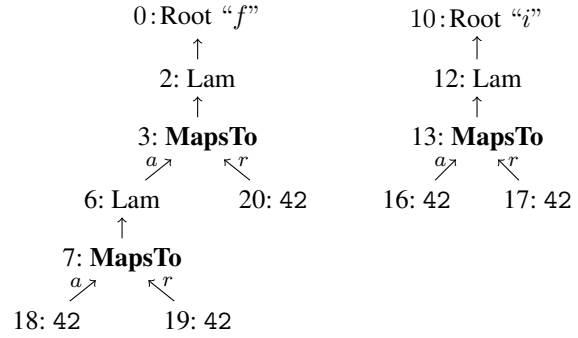

Figure 14. Event trees of the higher-order program.

Consider the higher-order program in Figure 12. Because function `f` uses and calls function `i`, we expect the computation tree to look as shown in Figure 13. Function `h` is passed as argument to function `f`, but inside the body of `f` function `h` is applied to an argument and the subcomputation for this argument has to be a child computation statement for the computation statement of `f`. Figure 14 shows the event trees of the value observation trace. All spans of the computation statement `i = 42` $\mapsto$ `42` are nested in the span $\langle 9, 18 \rangle$ of the argument of the argument of `f`. So for the computation statement `i = 42` $\mapsto$ `42` to be considered a child of the computation statement `f =` $\{$`42` $\mapsto$ `42`$\}$ $\mapsto$ `42`, this span $\langle 9, 18 \rangle$ has to be positive. Seeing that $\langle 9, 18 \rangle$ is the span of an argument of an argument, the method for determining whether any span is positive or negative becomes clear: Follow the path of event parents from the span upwards to the MapsTo event of the computation statement. If the path has an odd number of $\mathrm{P_a}$ i parents, then the span contributes negatively. If the path has an even number of $\mathrm{P_a}$ i parents, then the span contributes positively. A MapsTo is the one and only contravariant event: It flips the contribution of any span concerning its argument from positive to negative and vice versa.

Function `isPos` defines for the number i of a request or response event whether its span contributes positively:

```
mkCompTree [] n tree = tree
mkCompTree (e:trc) n tree
 | isStartOfPosSpan e =
    if isChildOf m n tree
     then mkCompTree trc m tree
     else mkCompTree trc m (mkChild m n tree)
 | isEndOfPosSpan e || isStartOfNegSpan e =
    mkCompTree trc (parentOf n tree) tree
 | isEndOfNegativeSpan e =
    mkCompTree trc m tree
 | otherwise =
    mkCompTree trc n tree
 where m = statementOf e
```

Figure 15. Algorithm for constructing the computation tree.

$$\text{isPos } i = \begin{cases} \text{True} & \text{, if } t_i = i\text{:Root } f \\ \text{not (isPos } k) & \text{, if } p_i = \text{P}_\text{a}\ k \\ \text{isPos } k & \text{, if } p_i = \text{P}_\text{r}\ k \text{ or P } k \text{ or P}_\text{c}\ k\ m \end{cases}$$

where p_i is the parent field of event t_i

3.5 Constructing the Edges of the Computation Tree

From the event trees we constructed the computation statements, the nodes of the computation tree. To determine which node is child of which other node, we sequentially traverse the value observation trace, considering the request-response spans.

Figure 15 shows the final algorithm for constructing a computation tree. The function `mkCompTree` takes a value observation trace (a list of events), a current node and a current tree to produce the final computation tree. We initially call `mkCompTree` with the whole value observation trace, the root node ★, and a tree without edges that has just the root node and all previously constructed computation statements as nodes.

Throughout the algorithm, the current node `n` keeps track of the composition of nested positive and negative spans. The current node indicates to which computation the next events contribute. The algorithm traverses the sequence of events from the beginning to the end, performing special operations at the start and end of most spans. In particular, at the start of a positive span the algorithm checks whether the computation statement `m` of that span is already a child of the current computation statement `n` within the current tree. If it is not yet, then an edge is added to the tree to make it a child. The algorithm continues with `m` as current computation statement.

In every step of the algorithm the current node `n` has a parent all the way up to ★. At the end of traversing the trace we have the computation tree. In the resulting tree every statement has exactly one parent: either the root node ★ or another statement.

```
and = observe "and" and'
and' b True = b
-- Missing "and' b False" -> Exception!

foldl = observe "foldl" foldl'
foldl' f z []     = z
foldl' f z (h:t) = let z' = f z h in foldl f z' t
```

Figure 16. Example program with observations.

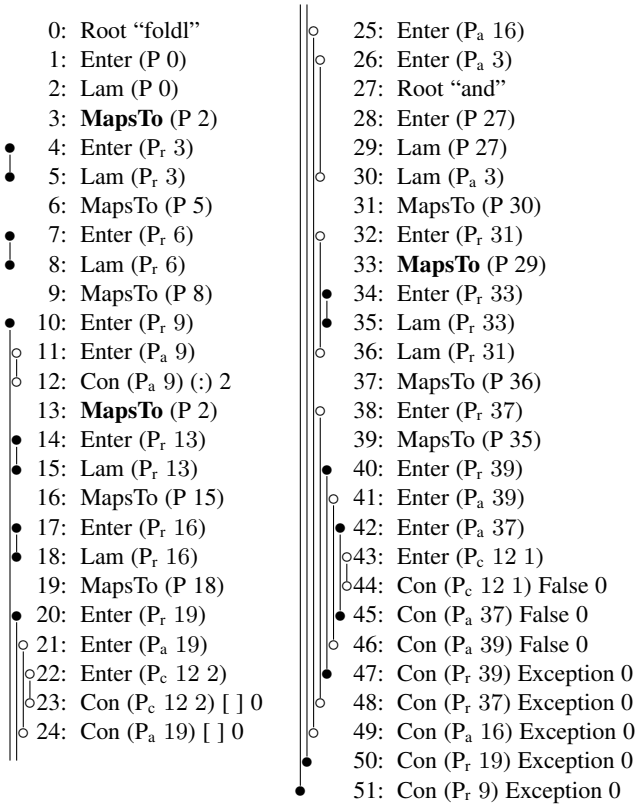

Figure 17. Trace of computation with higher order function.

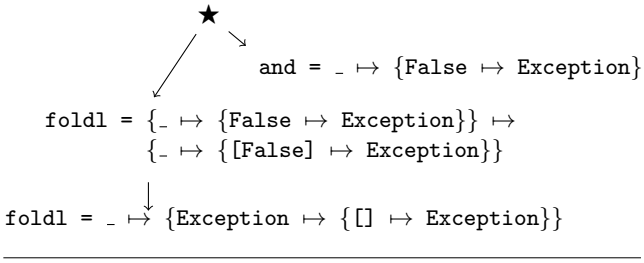

Figure 18. Computation tree for trace of Figure 17.

3.6 Example Construction of a Computation Tree

The program in Figure 16 defines recursively a higher-order function `foldl` over lists and contains an incomplete definition of the function `and`. Evaluating the expression `foldl and True [False]` results in an exception. The trace of the computation is given in Figure 17. We use • to mark

a positive span and ∘ to mark a negative span. These are not always the same as result, respectively argument spans.

The computation tree has three computation statements, corresponding to the three highlighted MapsTo events in the trace, two for the function `foldl` and one for the function `and`. The construction of the computation tree starts with the root ★. The traversal of the event sequence first reaches the span $\langle 4, 5\rangle$. Consequently the node for `foldl` that corresponds to the event $3 :$ MapsTo becomes a child of the current node ★. Later the spans $\langle 7, 8\rangle$ and $\langle 10, 51\rangle$ just confirm this parent-child edge. When reaching the span $\langle 14, 15\rangle$ the current node is the computation statement that corresponds to the event $3 :$ MapsTo and hence the computation statement for the event $13 :$ MapsTo becomes its child. Several subsequent spans change the current node but only at span $\langle 34, 45\rangle$ the computation for `and` that corresponds to the event $33 :$ MapsTo is added as new child to ★, which is the current node at the time. Again later spans change the current node, but do not change the tree any more. Figure 18 shows the final tree. The computation statement for `and`, which given a second argument `False` raises an exception, indicates a defect.

4. Our Algorithmic Debugger Hoed-pure

An algorithmic debugger [19, 26] uses a computation tree to find a defect in a program. An oracle judges computation statements of a computation tree, that is, the oracle decides whether a computations statement is right or wrong. Usually the programmer is the oracle.

To see how algorithmic debugging works, let us consider the computation tree of Figure 2. An interaction with the algorithmic debugger might look as follows, with the answers of the oracle/programmer written in *italics*:

```
isOdd 2 = False? right
isOdd 3 = False? wrong
plusOne 3 = 4? right
isEven 4 = False? wrong
modTwo 4 = 2?  wrong
Defect is in the definition of "modTwo"!
```

If a parent computation statement is wrong but all its children are right, then the definition of the function appearing in the parent is defective. Not all of the computation statements need to be judged. The default strategy starts asking questions at the root of the tree and the number of questions asked is proportional to the length of the path from the root to the defective node and the branch factor, the average number of children per node [27].

4.1 Implementation

We implemented our method for Haskell in the tracer and algorithmic debugger Hoed-pure. For simplicity Hoed-pure includes a reimplementation of Hood. Hoed-pure is also just a library. After execution of the main program has terminated, Hoed-pure constructs the computation tree from the trace and then serves an interactive webpage to any browser. The webpage provides both free exploration of the computation tree and guided algorithmic debugging. Hoed-pure has the same run-time overhead as Hood. It defines a type class `Observable`. A class instance implements tracing for a type. The type of any argument and the result of an observed function has to be an instance of `Observable`. Instances are derived with type-generic programming techniques [11].

The manipulation of the trace in our natural semantics is implemented like in Hood by using side-effects that write the trace. An optimising compiler might transform the program such that the order of trace events is changed. Gill [15] already argues that a compiler is unlikely to change the order of the side-effects and we have not observed any such problem in practice.

Our semantics describes how to handle exceptions in principle, but in Haskell exceptions are not simple constructors. Hence our implementation follow Hood in that every instance of class `Observable` catches any exception. If an exception occurs, then a response event for it is recorded in the trace and afterwards the same exception is re-raised.

4.2 Non-terminating Programs

Some defective programs do not terminate. To obtain a trace, the programmer lets the program run for a while and interrupts it. The interrupt will be recorded as an exception in the trace and Hoed-pure will produce a computation tree. However, such an asynchronous exception is not modelled in the semantics presented in this paper. The computation tree can still help the programmer understand why a program is misbehaving but algorithmic debugging is not guaranteed to find the defect.

5. Soundness of Algorithmic Debugging

Our method and our implementation Hoed-pure construct a value observation trace and from that a tree for any program with `observe` annotations. However, these annotations need to meet some conditions for the tree to be a computation tree suitable for algorithmic debugging.

5.1 Restrictions on Observation Annotations

Figure 19 shows the form of an annotated program that guarantees the generation of a computation tree suitable for algorithmic debugging. Our example program of Figure 3 is of this form (modulo syntactic sugar). Currently the programmer has to annotate the functions of interest. In the future a simple tool or compiler pass could annotate all top-level functions of a module.

Firstly, only a complete `let`-bound expression is annotated with the `observe` function and the label given as first argument to `observe` has to be the name of the let-bound variable. This ensures that a computation statement corresponds to the original, unannotated program.

Secondly, only expressions bound by the top-level `let` are annotated. All local bindings, that is, of `lets` nested

```
let { f = observe "f" (λx.e_f),
      g = observe "g" (λx.e_g),
      ...
      h = e_h,    } unobserved functions and
      x = e_x,    } data structures
      ...
    } in e
```

Figure 19. General annotated program.

within the top-level `let`, are excluded, because the bound expressions might contain free variables. Values of free variables are currently not included in a computation statement and hence such a computation statement is an incomplete description of a subcomputation. The question whether such a computation statement is right or wrong cannot be answered without knowing the values of free variables. For example, consider evaluating `myMain` for the program

```
f x = let g = observe "g" g'
          g' y = x+y
      in g 42
myMain = (f 3) + (f 5)
```

The observed two computation statements `g 42 = 45` and `g 42 = 47` even break equational reasoning.

The restriction to top-level definitions limits the precision of our algorithmic debugger. If a local function is defective, only the surrounding top-level function will be identified as defective. In our example we can only observe function `f` and any possible defect in the definition of `g` can only be identified as a defect in the definition of `f`. In the future we intend to lift this restriction by also recording the values of free variables in the value observation trace and adding the information to computation statements in the computation tree.

Finally, the sharing of computations because of lazy evaluation can prevent construction of a computation tree. Consider the example

```
ones = observe "ones" (1 : ones)
f = observe "f" (\x -> (head ones) + x)
myMain = (f 2) + (f 4)
```

Section 3 describes only how to construct computation statements for function applications. However, the method can easily be extended to also construct a computation statement such as `ones = 1 : _`. The problem is in determining the parent-child relationships.

The spans of `ones = 1 : _` are nested in the spans of the result of `f 2`. When `f 3` is evaluated, no events for `ones = 1 : _` are recorded any more. Hence in the computation tree the node `f 2 = 3` has the child `ones = 1 : _` but the node `f 4 = 5` has no child. On the other hand, for the program

```
onesA = observe "onesA" (\x -> (1 : onesA x))
f = observe "f" (\x -> (head (onesA x)) + x)
myMain = (f 2) + (f 4)
```

each application of `onesA` adds new spans and hence each node `f 2 = 3` and `f 4 = 5` has a separate child node `onesA _ = 1 : _`.

A constant is a variable that is let-bound to an expression. The value of a constant may be required for several, otherwise independent subcomputations of a program. The constant is evaluated only once and then its value is stored in the heap to be provided for all other subcomputations.

Besides the problem of not recording shared computations in the value observation trace, it is unclear what the computation tree for some computations involving constants should look like. In particular, a constant can be used to define a cyclic data structure, which naturally would give rise to a cyclic computation tree, a contradiction in terms. In the past, several alternative proposals for including constants have been made [20] and in the future we will see which of these we can combine with our lightweight tracing approach of constructing the computation tree.

So currently we have to be careful with constants in a program. Most constants in Haskell programs do not cause any problem, because either they do not use any other observed expressions, for example overloaded variables such as `(+)`, or they are evaluated only once, such as `main`, which is the initial expression for evaluating a Haskell program.

Finally, only λ-abstractions are annotated, because data structures in normal form are of little interest.

In summary, we only observe top level λ-abstractions and no observed expression may directly or indirectly use a constant that directly or indirectly uses an observed expression. Our examples obey these restrictions and so do our case studies in Section 7.

5.2 Testing Soundness

To verify the complete implementation of tracing, tree construction and algorithmic debugging we used the fully automated test method that we developed earlier [12]. That is, we randomly generated 100,000 valid programs with observed functions and injected a defect in some of the observed functions. We checked that our computation trees have the property that if a node is wrong but all its children are right, then the definition of the function appearing in the parent node actually contains a defect [18, 26]. For each program we construct a computation tree and use the algorithmic debugging method to produce a set of names of defective functions that we compare with the set of functions in which we injected a defect:

```
noFreeVars e ⇒ (algoDebug t ⊆ defects e)
where t = mkCompTree 𝒯
      {}, ⟨⟩ : e ⇓ Γ, 𝒯 : v
```

Algorithmic debugging does not guarantee to find all defects and some program parts may not even be evaluated, but the set of names found with algorithmic debugging should be a subset of the set of functions in which we injected a defect.

In our earlier work we replace values by judgements [12]. This method does not directly transfer to a language with data constructors and case expressions. Instead, we annotate every data value, that is, saturated application of a data constructor, with a Boolean flag stating whether it is right.

We introduce a pseudo-function `infect`, which traverses its argument and makes all of its parts wrong. So for testing, any occurrence of `infect` is a defect in the program. Infection of a data structure infects all components of that data structure. Infection of a function yields a function that always returns an infected value. Infection never turns a constructor application into a λ-abstraction or vice versa, because we assume the presence of a type system that prevents such a defect. Evaluating a case expression continues with an infected value, if the inspected data constructor is wrong. Hence a data structure may contain infected components, but as long as a computation does not demand any of these infected components, the computation is not infected.

6. Related Work

Our work builds on Andy Gill's value observation technique [15]. We reimplement his library Hood and use its value observation trace. In Section 2 of this paper we provide a formal definition for this trace.

6.1 Computation Trees for Functional Languages

The first computation tree structure that was proposed for lazy functional languages is the evaluation dependence tree (EDT) [22]. Most algorithmic debuggers for lazy languages [1, 3, 20, 21, 30] construct EDTs. The EDT represents functional values as function identifiers or partial applications.

Because a λ-abstraction plus the binding of its free variables is often big, inclusion of λ-abstractions in the EDT is problematic. The algorithmic debugger Buddha is the first to implement the idea of representing functional values extensionally, that is, as finite maps from arguments to results, instead of intensionally [24, 25].

An extensional representation also requires a different tree structure, the function dependence tree (FDT). In an FDT a computation statement $f = \ldots$ is the parent of a computation statement $g = \ldots$ if and only if the function identifier g appears in the definition of the function f. In an EDT that parent-child relation holds, if and only if the application of function g appears in the definition of function f. So for first-order functions the two tree structures coincide. Chitil et al. [5] formally define the corresponding FDT, compare the two tree structures and prove that both have the essential property for algorithmic debugging.

Hoed-pure uses the extensional representation of functional values. It produces an FDT structure. However, the proofs of Chitil, Davie and Yong [5, 6] do not directly transfer, because they use a slightly different programming language with a semantics defined by graph rewriting, not a natural semantics.

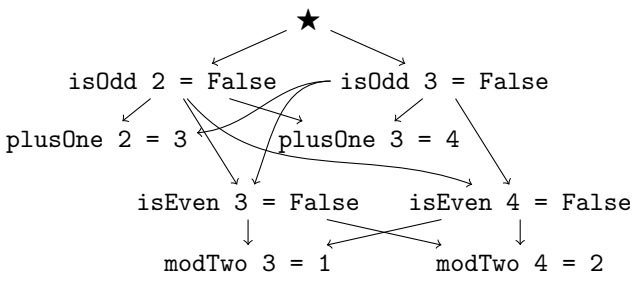

Figure 20. Hoed-cc's computation tree (graph).

6.2 Computation Tree Tracing for Haskell

Freja [20–22] is the first algorithmic debugger for a substantial subset of Haskell. Freja is a complete compiler and uses an instrumented runtime system to construct the computation tree. The system handles CAFs and provides many features for making algorithmic debugging easy to use. The compiler front-end ensures that all information about the source code that is required for algorithmic debugging is passed to the back-end. Adding a language feature would require extending many of the compiler passes and the runtime system.

Hat [8, 28, 30] is a set of tools for tracing Haskell 98 programs. The tracing tool transforms a Haskell program into another Haskell program that, when executed, writes a detailed trace into a file in addition to performing the same computation as the original program. The trace includes a computation tree plus additional information. Hat provides many viewing tools for exploring a trace, one of which is an algorithmic debugger. Chitil et al. compare an old version of Hat with Freja and Hood [7]. Like Freja, Hat supports trusting a module. Computations of a trusted module are not traced and hence do not appear in the computation tree. However, trusted modules still have to be transformed by the tracing tool and hence can use only supported language features. Adding a language feature to Hat would require extending the source-to-source transformation tool.

Buddha [24, 25] is another algorithmic debugger for Haskell. Like Hat, Buddha is also based on program transformation. The trace is a computation tree. The transformation is different from Hat and the resulting program uses a primitive for observing an expression of any type without forcing its evaluation. That primitive was implemented in the Glasgow Haskell compiler. Buddha is the first algorithmic debugger that can provide an extensional representation of functional values. Adding a language feature would require extending the source-to-source transformation and possibly the primitive.

Hoed-cc [12] is the first algorithmic debugger that works for real-world Haskell programs. Continuous evolution of the Haskell language and the complex implementation of Freja, Hat and Buddha means that these debuggers only support subsets of the language features used in real-world

```
isEven x = if x == 1 then True else isOdd (x-1)
isOdd x = if x == 1 then True else isEven (x-1)
```

Figure 21. Program with mutual recursion.

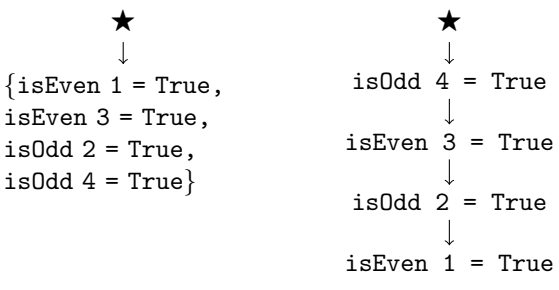

Figure 22. Computation trees of Hoed-cc and Hoed-pure

Haskell programs. In contrast Hoed-cc is a library that combines the Haskell object observation debugger (Hood) [15] with the cost centre stack provided by the profiling option of the Glasgow Haskell compiler. Implementing Hoed-cc's tracing method for other Haskell implementations or other languages would require extending the compiler and runtime system with cost centre stack support. For example, the interpreter GHCi does not support cost-centre stacks.

Because cost centre stacks only contain function names, not arguments of specific calls, and are also compressed, Hoed-cc generates many surplus child-parent dependencies. Figure 20 shows Hoed-cc's computation tree for our example (actually an acyclic directed graph). Compare it with Figure 2. Surplus dependencies in a Hoed-cc computation tree increase the number of statements an algorithmic debugger asks the oracle to judge. Algorithmic debugging with the computation trees from the Introduction may require up to 5 questions with Hoed-pure and 8 questions with Hoed-cc.

A node in a Hoed-cc computation tree may contain a set of computation statements. Consider evaluating `isOdd 4` for the program in Figure 21. Figure 22 shows Hoed-cc's computation tree on the left and Hoed-pure's on the right. An algorithmic debugger that uses Hoed-cc's tree tells us that the defect is in one of the functions `isEven` and `isOdd`. In contrast, with Hoed-pure's tree the debugger can tell us that the defect is in `isEven` when applied to 1.

A Hoed-cc annotation requires the introduction of a lambda expression and certain compiler optimisations must be disabled to keep the lambda expression in place. To observe for example `isOdd` with Hoed-cc the following annotation is used:

```
isOdd = observe "isOdd"
  (\ n -> {-# SCC "isOdd" #-} (isOdd' n))
```

Our earlier semantics [12] formalises the observation of functional values in a different but equivalent way. In that definition $\texttt{obs}_\lambda$ does not form a value but an expression. The definition is closer to the implementation of Hood, whereas the definition given here expresses the similarities and differences between observing constructor applications and λ-abstractions more clearly.

6.3 Computation Tree Tracing for Other Languages

Shapiro constructed computation trees for the logic language Prolog [26]. Algorithmic debugging has since been applied to many other languages; we give a few notable examples.

Fritzon et al. generalized computation tree tracing to languages with side effects [13]. An algorithmic debugger with a framework to record side-effects in the computation tree is for example available for Java [2, 16]. Tail call optimization is forbidden and higher order functions are not supported.

Algorithmic debugging is also applied to strict functional languages such as Erlang [4]. The implementation is complex and uses a specific run-time system to transform all code, including libraries, during evaluation of the program.

7. Case Studies

We compare Hoed-pure with Hoed-cc in three case studies.

7.1 A Video Game

The game Raincat [14] consists of approximately 2500 lines of Haskell code and uses libraries such as OpenGL that are not written in Haskell. Hoed-cc was the first algorithmic debugger that could handle Raincat [12]. Hoed-pure can also be used to debug Raincat. Because Raincat is an interactive game, its trace is different for every run. Hence we cannot compare debugging sessions in detail.

7.2 A Defective Window Manager

XMonad [10] is an X11 window manager written in roughly 1300 lines of Haskell. We introduced a defect in XMonad which incorrectly duplicates the workspace brought into focus. XMonad's property-based tests detect, without user interaction, that something is wrong. We annotated the 9 functions in the code related to the failing property.

For the counter-example found by XMonad's tests Hoed-pure generated a computation tree with 12 nodes (the artificial root node and a node for each computation statement), 11 edges and a branch factor, the average number of children of non-leaf nodes, of 2.2. We found the defect after judging 3 statements.

Hoed-cc also generated 11 computation statements but organised in a tree with 7 nodes (an artificial root node, four nodes with one statement, a statement with 3 applications of `view` and a node with two applications of `member`, an application of `insertUp` and an application of `shiftWin`), 7 edges and branch factor 2.33. We found the defect after judging 9 statements. Because the defect was not in one of the nodes with multiple statements the precision with Hoed-cc is in this case the same as with Hoed-pure.

7.3 A Defective Pretty-Printer

Within the implementation of Hoed-pure we used version 1.0 of the library FPretty [23] to pretty-print computation statements over several lines with appropriate indentation. We noticed that the library sometimes indents more than we expected and the first author, who was unfamiliar with FPretty, investigated with Hoed-pure.

FPretty is a small library with just 12 functions. Because most of them are higher-order functions that take other higher-order functions as arguments, it was non-trivial to understand what each function should do. We annotated all 12 top-level functions in the library. Then we pretty-printed an example, during which 15327 events were collected. These events translated to 65 computation statements which were organised in a computation tree with 65 edges and branch factor 1.8. We found the defect after judging 11 statements. After we found the defect, we proposed a fix which is included in FPretty 1.1.

With Hoed-cc the 65 computation statements are organised in a tree with 7 nodes, 9 edges and branch factor 3.0. After judging 65 statements the defect was found in a node with computation statements of the defective function and three other functions.

8. Conclusion

A computation tree is a key means for understanding how a program works, or why it does not work. A computation tree can be explored freely, or an algorithmic debugger can be used to systematically traverse a computation tree and find the location of a defect.

We have presented a new lightweight method for generating a computation tree. The starting point is our formal definition of the trace generated by the original Hood library. The definition enables us to see the existence of request-response spans in traces and realise how their nesting determines the structure of a computation tree. The order of events in the trace reflects the evaluation order, but the computation tree has a structure independent of evaluation order and reflects the program structure instead. Our tracing semantics is specific to lazy evaluation, but our idea of observing values by simple instrumentation by a library and transforming the resulting trace into a computation tree is independent of evaluation order and applicable to many programming languages. Negative request-response spans are not only required for lazy evaluation but also call-by-value languages can benefit from the method for relating function calls in the presence of higher-order functions.

We implemented our method in the library Hoed-pure. Hoed-pure supports Haskell language extensions and any Haskell compiler. The user only has to import the library and annotate functions of interest; untraced code remains unchanged. Therefore Hoed-pure is well suitable for debugging real-world Haskell programs, which may use libraries written in other programming languages.

In contrast to Hoed-cc, Hoed-pure is portable and produces a precise computation tree. We showed that the algorithmic debugger asked substantially fewer questions using Hoed-pure's computation tree.

We plan to extend our method to also debug constants and in particular cyclic data structures. We want to explore the structure of value observation traces further. We believe that the value observation technique can obtain even more information than required for a computation tree and that using this information can improve debugging beyond standard algorithmic debugging. To obtain more information, we may have to alter the value observation trace. Finally, to improve debugging of real-world programs, we must shift from constructing a computation tree to examining numerous aspects of the human-computer interface of a practical debugger.

Acknowledgments

We thank the anonymous reviewers of PLDI and especially Simon Peyton Jones for their thorough reviews and insightful comments.

References

[1] B. Braßel and H. Siegel. Debugging lazy functional programs by asking the oracle. In *Implementation and Application of Functional Languages, 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, LNCS 5083, pages 183–200, 2008.

[2] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. *Electronic Notes in Theoretical Computer Science*, 177:75–89, 2007.

[3] R. Caballero, F. J. López-Fraguas, and M. Rodríguez-Artalejo. Theoretical foundations for the declarative debugging of lazy functional logic programs. In *Functional and Logic Programming*, LNCS 2024, pages 170–184, 2001.

[4] R. Caballero, E. Martin-Martin, A. Riesco, and S. Tamarit. EDD: A Declarative Debugger for Sequential Erlang Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 581–586. Springer, 2014.

[5] O. Chitil and T. Davie. Comprehending finite maps for algorithmic debugging of higher-order functional programs. In *Proceedings of the International Conference on Principles and Practice of Declarative Programming*, PPDP 2008, pages 205–216, 2008.

[6] O. Chitil and Y. Luo. Structure and properties of traces for functional programs. *Electronic Notes in Theoretical Computer Science*, 176(1):39–63, 2007.

[7] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood — a comparative evaluation of three systems for tracing and debugging lazy functional programs. In *Implementation of Functional Languages*, LNCS 2011. 2001.

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

[9] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of*

the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pages 268–279, 2000.

[10] Don Stewart. XMonad. `http://xmonad.org`, 2015.

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

[12] M. Faddegon and O. Chitil. Algorithmic Debugging of Real-World Haskell Programs: Deriving Dependencies from the Cost Centre Stack. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 33–42, 2015.

[13] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, Dec. 1992.

[14] Game Creation Society, Carnegie Mellon. Raincat. `http://www.gamecreation.org/game/raincat`, 2014.

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

[16] D. Insa and J. Silva. An algorithmic debugger for Java. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–6. IEEE, 2010.

[17] J. Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the symposium on Principles of programming languages*, POPL 1993, pages 144–154, 1993.

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

[19] L. Naish. A three-valued declarative debugging scheme. In *Computer Science Conference, 2000. ACSC 2000. 23rd Australasian*, pages 166–173, 2000.

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

[21] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, PLILP '92, pages 385–399. LNCS 631, 1992.

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

[23] Olaf Chitil. FPretty. `http://hackage.haskell.org/package/FPretty`, 2012.

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

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

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

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

[28] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *Programming Languages: Implementations, Logics, and Programs*, PLILP '97, pages 291–308. LNCS 1292, 1997.

[29] P. Wadler. Why No One Uses Functional Languages. *SIGPLAN Not.*, 33(8):23–27, 1998.

[30] 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.

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

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