ALGORITHMIC DEBUGGING
FOR COMPLEX LAZY FUNCTIONAL PROGRAMS

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Maarten Faddegon
June 2017

Abstract

An algorithmic debugger finds defects in programs by systematic search. It relies on the programmer to direct the search by answering a series of yes/no questions about the correctness of specific function applications and their results.

Existing algorithmic debuggers for a lazy functional language work well for small simple programs but cannot be used to locate defects in complex programs for two reasons: Firstly, to collect the information required for algorithmic debugging existing debuggers use different but complex implementations. Therefore, these debuggers are hard to maintain and do not support all the latest language features. As a consequence, programs with unsupported language features cannot be debugged. Also inclusion of a library using unsupported languages features can make algorithmic debugging unusable even when the programmer is not interested in debugging the library. Secondly, algorithmic debugging breaks down when the size or number of questions is too great for the programmer to handle.

This is a pity, because, even though algorithmic debugging is a promising method for locating defects, many real-world programs are too complex for the method to be usuable.

I claim that the techniques in in this thesis make algorithmic debugging useable for a much more complex lazy functional programs.

I present a novel method for collecting the information required for algorithmically debugging a lazy functional program. The method is non-invasive, uses program annotations in suspected modules only and has a simple implementation.

My method supports all of Haskell, including laziness, higher-order functions and exceptions. Future language extensions can be supported without changes, or with minimal changes, to the implementation of the debugger.

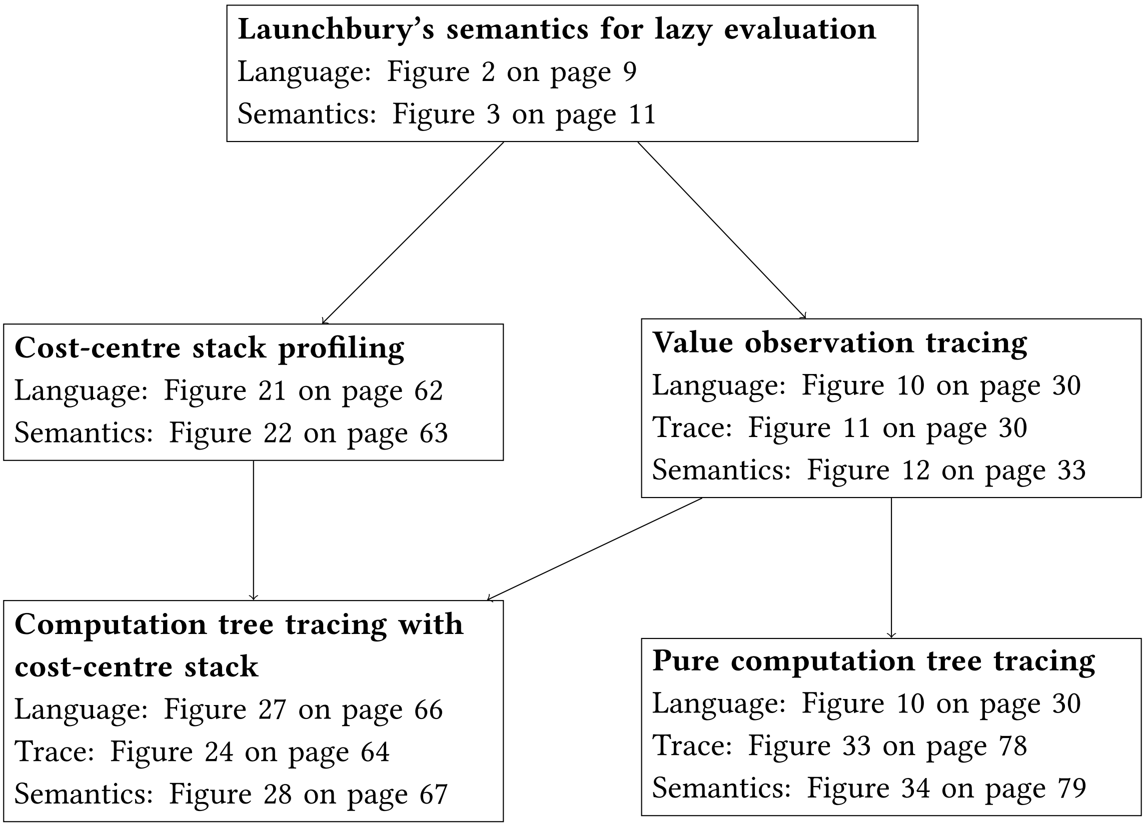
With my method the programmer can focus on untrusted code – lots of trusted libraries are unaffected. This makes traces, and hence the amount of questions that needs to be answered, more manageable.

I give a type-generic definition to support custom types defined by the programmer.

Furthermore, I propose a method that re-uses properties to answer automatically some of the questions arising during algorithmic debugging, and to replace others by simpler questions. Properties may already be present in the code for testing; the programmer can also encode a specification or reference implementation as a property, or add a new property in response to a statement they are asked to judge.

Contents

Languages


Launchbury's semantics for lazy evaluation
Language: Figure 2 on page 9
Semantics: Figure 3 on page 11
Cost-centre stack profiling
Language: Figure 21 on page 62
Semantics: Figure 22 on page 63
Value observation tracing
Language: Figure 10 on page 30
Trace: Figure 11 on page 30
Semantics: Figure 12 on page 33
Computation tree tracing with cost-centre stack
Language: Figure 27 on page 66
Trace: Figure 24 on page 64
Semantics: Figure 28 on page 67
Pure computation tree tracing
Language: Figure 10 on page 30
Trace: Figure 33 on page 78
Semantics: Figure 34 on page 79

1 Introduction

Defective programs cause all kinds of problems and lead to huge economic costs for society (Tassey 2002). Programmers are estimated to spend more than half of their time locating defects in misbehaving programs (Beizer 1990). The time needed to write and maintain software can be reduced when the program is defined in a high-level programming language that abstracts from the details of how the machine runs the program. The programmer uses a computer program, called a compiler, to translate the high-level code into machine language that can be executed.

In this thesis I focus on the family of high-level programming languages that are lazy and functional. A *lazy* language uses an evaluation model where an expression is evaluated to a value when the value is needed. The primary operation in a *functional* program is the application of functions to arguments (Hughes 1989). In many functional languages the programmer can define a *higher-order* function, a function that can be applied to another function or that returns a function as result. I discuss functional programming and lazy evaluation in more detail in Section 2.1.

Structuring code well makes development easier and provides modules that can be re-used in the future, lazy evaluation and higher-order functions can contribute significantly to modularity (Hughes 1989). A large scale study of open-source projects showed that lazy functional programs misbehave less often than programs written in other languages (Ray et al. 2014). For these reasons, lazy functional languages are used to develop programs in complex domains where correctness is important (Wadler 1998a; Augustsson 1999).

Lazy functional languages, however, do not eliminate all programming mistakes. A *debugger* is a tool that, given a program with observable faulty behaviour, supports the programmer finding a defect in the code that caused the program to misbehave. Support for debugging is essential for wider adoption of lazy functional languages

(Wadler 1998b; Zielonka and the GHC Team 2005). The algorithmic debugging method is particularly suitable for pure computations (cf. Zeller (2009)) and thus lazy functional languages. Existing algorithmic debuggers for lazy functional languages work well for small and simple programs but cannot be used to locate defects in complex programs This is a pity, because a promising approach to debugging is therefore not applicable to many real-world programs. So I reach the motivating question for this thesis:

> *How can algorithmic debugging for lazy functional languages be made useable not only for small and simple programs but also for complex programs?*

Algorithmic Debugging Let us consider the program of Listing 1. This program is written in the lazy functional language Haskell (Peyton Jones et al. 2003; Marlow et al. 2010) which I use for examples throughout the thesis, for the reader unfamiliar with Haskell I describe the language in Section 2.1.3. The program sorts a list `xs` by starting with the empty list `[]` (which is trivially ordered) and inserting one element from `xs` to create a new ordered list, this is repeated until the final result is an ordered list containing all elements of `xs`. However, the second definition of insert is wrong: when `x <= y` is true the result should be `x:y:ys` instead of `x:ys` (element `y` from the ordered list should not be discarded).

Listing 1 A defective program for sorting lists.

```
sort xs = foldr insert [] xs

insert x [] = [x]
insert x (y:ys) = if x <= y then x : ys else y : insert x ys
```

Evaluating the function `sort` applied to “cab" I get the unexpected result “ac", I observe a symptom of the defect in the code. When I wrote the program of Listing 1 my intention was that `sort` applied to a list `xs` evaluates to an ordered list that contains every element from `xs`. The function application and result

```
sort "cab" = "ac"
```

does not satisfy my intention because the element *b* from the argument is not an element of the result. I will refer to a function applied an argument and its result as a *computation statement.* When a function has no side effects, such as reading or writing globally accessible memory or showing data on the screen, evaluating a function application is completely described by its computation statement. For a higher-order function argument or result can be a functional value.

An algorithmic debugger (Shapiro 1983) records and presents computation statements of, otherwise hidden, subcomputations to an oracle who is asked to judge if the computation statements are *right* or *wrong* according to the intentioned behaviour/specification of the function. Commonly the oracle is a human, namely the programmer who tries to find and correct a mistake. For my example program the interaction with the algorithmic debugger could be as follows, with the answers of the oracle written in *italics*:

```
sort "cab" = "ab" ? wrong
insert 'c' "a" = "ac" ? right
insert 'a' "b" = "a" ? wrong
defect located in insert observed by insert 'a' "b" = "a"
```

How did the algorithmic debugger generate this sequence of questions and come to the conclusion? For these the algorithmic debugger uses a computation tree. The nodes of the tree are computation statements. A statement $f\ x = v$ *depends on* another statement $g\ y = w$ if there exists a part w' of w such that w' is needed to determine v and x can be determined without w'. If a parent node is wrong but all its child nodes are right the node is *defective*. The part of the program associated with the node (here the definition of the function `insert`) must be defective.

Figure 1 shows the computation tree of my example. Statements I judged as wrong are annotated with ✘ and the statement I judged as right is annotated with ✔. When recording the trace from which the tree is constructed I marked the functions `sort` and `insert` as *suspected*. On the other hand, the functions `foldr` and `(<=)` are *trusted* and hence applications of the latter functions are not recorded in the computation tree.

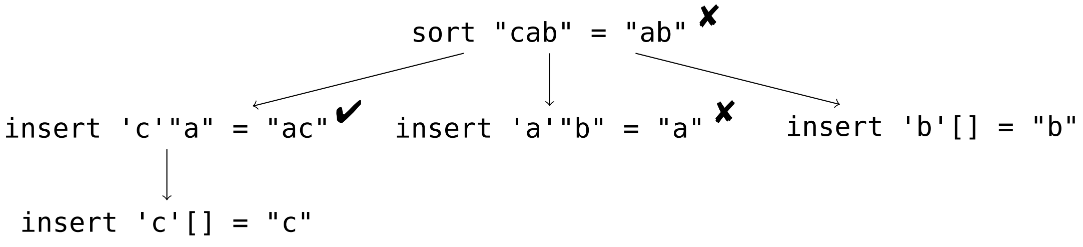


Figure 1: Computation tree of evaluating `sort "cab"`.

Applicability Algorithmic debugging is particularly suitable for finding defects in pure, that is, side-effect free, computations and hence for debugging lazy functional programs (Shapiro 1983; Zeller 2009; Nilsson 1998) . Several algorithmic debuggers, such as Freja (Nilsson and Sparud 1997; Nilsson 1998), Hat (Wallace et al. 2001) and Buddha (Pope 2006), have been built for the language Haskell.

These algorithmic debuggers for Haskell are a great help for finding defects in small simple programs, however all existing algorithmic debuggers share two challenges that limit the usability applied to complex programs:

1. **Limited language support:** Existing algorithmic debuggers have in common that the implementation is very complex — to record computation statements and their dependencies either a specialized run-time system or a transformation of all modules is needed — and hence these debuggers require constant maintenance to keep pace with the evolution of the language. In practice, therefore, many programs cannot be debugged with these tools because language extensions that are not supported by the debugger are used, directly or in a library that the user does not even want to debug.

2. **The human bottleneck:** Algorithmic debugging with the human programmer as oracle does not scale well to large and complex programs. The size and number of computation statements that need to be judged before a defect is located can overwhelm the programmer. Judging computation statements with an automated oracle, using a working program or formal specification as reference, instead of the human programmer has been suggested by many (Shapiro 1983; Drabent and Nadjm-Tehrani 1989; Nilsson and Fritzson 1992). In practice, however, such a reference is often not available and hence these methods are not used widely.

Because of these limitations there is, to my knowledge, currently no algorithmic debugger available that is useable for real-world lazy functional programs. In this thesis I show how to address these shortcomings.

Useable algorithmic debugging In this thesis I present the techniques required to build an algorithmic debugger that is useable for finding defects in complex and large lazy functional programs. I implemented my methods in a new algorithmic debugger for Haskell called Hoed. Hoed is just a library and many new language extension supported by a Haskell compiler can be supported without changes to Hoed.

Property-based testing works by defining a function that evaluates to a true/false/undefined value and describes a desired property of the implementation. A tool, such as QuickCheck (Claessen and Hughes 2000a), generates many values and tests if the property holds for all these values. If there is a value for which the property does not hold the tool detected a defect, otherwise the programmer is given confidence, depending on how representative the tested values are, in the soundness of their implementation. For example, we detect that our program is defective because the property

```
prop_insert_complete x y ys =
  x ‘elem‘ (y:ys) ==> x ‘elem‘ (insert y ys)
```

evaluates to `False` for the values `x='c'`, `y='c'` and `ys="ab"`.

By using test-properties to automatically judge computation statements algorithmic debugging scales to large programs. Properties may already be present in the code for testing; the programmer can also encode a specification or reference implementation as a property, or add a new property in response to a statement they are asked to judge. Properties that are added during a debugging session may be used again for further testing in the future.

Main contributions Together, the following contributions make algorithmimic debugging a method that is useable for finding defects in a much wider range of lazy functional programs than with existing algorithmic debuggers:

- **A semantics for value observation tracing**, a technique to record the values of otherwise hidden values of intermediate computations. Gill (2000) previously presented the technique in a paper and implemented it in the Haskell Object Observation Debugger (HOOD) but the technique has not been formalized before. HOOD can be used to obtain computation statements, for algorithmic debugging the statements need to be connected and form a computation tree (Chapter 3).
- **A type-generic definition of how a value should be observed.** While HOOD comes with definitions for the base types and many other commonly used types, for user defined types the programmer has to write their own definition. This is not only a laborious task, also strictness of the program is easily affected. With my type-generic definition the compiler can for any value derive how to observe it (Chapter 4).
- **A method to construct a computation tree from a value observation trace**

and trace stacks used for profiling, based on my observation that the information required for connecting individual intermediate computations to a computation tree is closely related to the information available in the trace stacks. I give a semantics that combines maintaining a trace stack with value observation tracing such that a trace stack is associated with each computation statement. The resulting computation trees are sound for algorithmic debugging but the trees contain surplus edges because trace stacks approximate the information needed to connect the computation statement (Chapter 5).

- **A method that constructs a computation tree purely from a value observation trace** based on my discovery that value observation traces contain more information than previously thought. A value observation trace is a sequence of events, written in the order in which the program is evaluated. Each atomic value of an observed expression has a request and response event which I together call a span. I discovered that the spans, which occur nested and in sequence, correspond to the dependencies between computation statements. With this method I derive computation trees without surplus dependencies, and I remove the first method's dependency on the profiling run-time environment (Chapter 6).

- **A method to test soundness of a computation tree for algorithmic debugging**. Previous work in which new methods for computation tree construction are presented often come without soundness proof or systematic testing. Chitil and Luo (2007) revisited an existing computation tree construction method and proofed soundness, but the proof does not directly transfer to my approach because they use a slightly different programming language with a semantics defined by graph rewriting . Given a randomly generated program of which a random part is declared defective my test-method validates that algorithmic debugging with the program's computation tree indeed locates the defect in the part declared defective (Chapter 7).

- **A method to judge computation statements using test properties** to make algorithmic debugging scale to large and complex programs. Previous work already suggested using a reference program or a complete specification, however, in practice these are seldom available. Furthermore, I present several methods to remove dependencies from properties on unevaluated expressions (Chapter 8).

- **An evaluation of my implementation of above techniques with use-cases based on defects in open-source projects** (Chapter 9). I implemented my techniques in the algorithmic debugger Hoed and made it available from the Haskell package archive Hackage: `cabal install Hoed`.

The focus of this thesis is on algorithmic debugging of lazy functional languages. However, many of my insights and techniques are of wider applicability. My discovery of spans in an observation trace also helps to build algorithmic debuggers for strict languages that work well in the presence of higher-order functions. Using properties to judge computation statements is applicable to algorithmic debugging programs written in many different programming languages. My type-generic definition of which part of a value is atomic is now also part of the original HOOD library.

Relation to my earlier publications I presented the basic idea for lightweight computation tree construction from a value observation trace with at poster at the International Conference on Functional Programming (ICFP) in Götenborg, Sweden in 2014. My type-generic definition of atomic values for tracing was presented at Trend in Functional Programming 2014 in Soesterberg, The Netherlands (Faddegon and Chitil 2014), an extended article was published in the journal on Computer Languages, Systems & Structures (Faddegon and Chitil 2017). The method for using profiling information to construct a computation tree was then presented at the conference on Programming Language Design and Implementation (PLDI) in Portland, USA (Faddegon and Chitil 2015). My insight that HOOD traces contain much more information than previously thought, and the method to construct a computation tree from these was presented at the conference on Programming Language Design and Implementation in Santa Barbara, USA (Faddegon and Chitil 2016). The Artifact Evaluation Commitees of PLDI 2015 and 2016 evaluated earlier version of my test-method and case-studies with Hoed-cc and Hoed-pure and declared that these met or exceeded expectations. Using properties as oracle for algorithmic debugging was presented at the Workshop on Implementation of Functional Languages in Leuven, Belgium (Faddegon and Runciman 2016). This thesis includes parts from these publications, but in most parts the material has been substantially revised and extended.

2 Algorithmic debugging for lazy functional programs

Here I give a brief introduction to functional programming, Launchbury's semantics for lazy evaluation and earlier work on algorithmic debugging.

2.1 Functional programming

The lambda calculus, defined by Church (1936), is a formal system that is at the core of *functional programming*. In this chapter I give just enough background about functional programming and the lambda calculus to make this thesis self-contained, more thorough introductions are given by e.g. Barendregt and Barendsen (1984) and Bird and Wadler (1988).

At the core of the lambda calculus are functions such as $\lambda x.x$ and $\lambda y.3$, the former returns the value to which the function is applied and the latter always returns the value 3. An expression is evaluated by applying functions. For example

$$(\lambda y.3)((\lambda x.x)4)$$

could be evaluated by first applying $(\lambda x.x)$ to 4 such that the expression reduces to

$$(\lambda y.3)4$$

and application of $(\lambda y.3)$ to 4 gives 4.

This is just one way of evaluating the expression, alternatively we could for example first evaluate the application of $(\lambda y.3)$ to $((\lambda x.x)4)$. There are many operational

expression	e	$::=$	v	
		$\mid$	$e\ x$	application
		$\mid$	$\texttt{let}\ \{x_k = e_k\}_{k=1}^{n}\ \texttt{in}\ e$	recursive binding
		$\mid$	$\texttt{case}\ e\ \texttt{of}\ \{c_k\ x_1 \ldots x_{m_k} \rightarrow e_k\}_{k=1}^{n}$	case
		$\mid$	x	variable
		$\mid$	$x_1 \oplus x_2$	application of a primitive
value	v	$::=$	$\lambda x.e$	abstraction
		$\mid$	$c\ x_1 \ldots x_n$	saturated application of data constructor

Figure 2: Syntax of the core language.

and semantical choices when specifying a set of rules that define how an expression is rewritten. I use Launchbury's semantics for lazy evaluation as a base in this thesis.

2.1.1 Core language

Before discussing semantics I first give the syntax of an expression in the core language that I use and extend in the rest of this thesis in Figure 2. The basis is Launchbury's core language together with his data constructors and primitive operations.

When a variable x is *bound* to an expression e, then any occurance of x may be replaced by e or the value to which e evaluates. Given a recursive binding $\texttt{let}\ \{x_k = e_k\}_{k=1}^{n}\ \texttt{in}\ e$, the variable x_1 is bound to e_1 in any e_k and in e, x_2 is bound to e_2 in any e_k and in e, ... and x_n is bound to e_n in any e_k and in e. For example, in the following expression the variable x_2 is bound to the expression x_1:

$$\texttt{let}\ \{x_1\ \texttt{=}\ \texttt{3,}\ x_2\ \texttt{=}\ \underset{\substack{\uparrow \\ \text{expression } x_2 \text{ is bound to}}}{x_1}\ \texttt{,}\ x_3\ \texttt{=}\ x_3\}\ \texttt{in}\ \overset{\substack{\text{free} \\ \downarrow}}{\texttt{y}}\ \ \underset{\substack{\uparrow \\ \text{bound}}}{x_2}$$

A variable is *free* in an expression when the variable is not bound in the expression. For example y is a free variable in above expression. A program is an expression without free variables.

The language contains two sorts of values: a saturated appliction of a data constructor, and a functional value. A data constructor has an arity n and an application of the data constructor is saturated when the data constructor is applied to n variables. Integer values are just data constructors with arity 0. An exception is also just the constructor `Exception`. I include exceptions in my language, because in practice defective programs often raise exceptions. An abstraction $\lambda x.e$ is an expression e that depends on variable x. In other words the abstraction defines a function from x to e. An abstraction $\lambda x.e$ is applied to a variable y by substituting any occurrence of x in e for y. For example $(\lambda x.fx)\ y$ becomes fy.

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 (Launchbury 1993).

2.1.2 Semantics for lazy evaluation of the core language

A program is evaluated by systematically rewriting the expression as defined by a set of rules, the semantics of the language. When defining the rewrite rules many choices are to be made. Is an argument evaluated before applying a function to the argument, or when the function body needs the argument value? In the former case the semantics is *strict* in the latter it is *non-strict.* For a non-strict semantics: is the result of evaluation *shared* between uses of the argument or is the argument re-evaluated for every use? Non-strictness and preventing repeated evaluation together define a *lazy* language.

A heap is a partial function from variables to expressions. I 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 3 defines the computation statement $\Gamma_1 : e \Downarrow \Gamma_2 : v$, which means that the expression e in the context of the heap Γ_1 reduces to the value v together with the modified heap Γ_2. A computation starts with an empty heap. Some rules are of the form

$$\frac{\Gamma_1 : e_2 \Downarrow \Gamma_2 : v_2}{\Gamma_1 : e_1 \Downarrow \Gamma_2 : v_1} \quad \text{or} \quad \frac{\Gamma_1 : e_2 \Downarrow \Gamma_2 : v_2 \qquad \Gamma_2 : e_3 \Downarrow \Gamma_3 : v_3}{\Gamma_1 : e_1 \Downarrow \Gamma_3 : v_1}$$

which means that respectively $\Gamma_1 : e_1 \Downarrow \Gamma_2 : v_1$ follows from *subevaluation* $\Gamma_1 : e_2 \Downarrow \Gamma_2 : v_2$, and that $\Gamma_1 : e_1 \Downarrow \Gamma_3 : v_1$ follows from the subevaluations $\Gamma_1 : e_2 \Downarrow \Gamma_2 : v_2$ and $\Gamma_2 : e_3 \Downarrow \Gamma_3 : v_3$.

The seven rules Lam, Con, Var, Let, App, Case and Prim are almost identical to rules with the same names in Launchbury's semantics (Launchbury 1993). Launchbury's semantics assume expression are in a normal form where:

- a bound variable is distinct from all other bound variables such that scope becomes irrelevant; and
- the argument to which a λ-expression is applied is a variable.

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

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

$$\frac{\Gamma_1 : e \Downarrow \Gamma_2 : v}{\Gamma_1[x \mapsto e] : x \Downarrow \Gamma_2[x \mapsto v] : \hat{v}}\ \text{Var}$$

$$\frac{\Gamma_1[x_i \mapsto e_i]_{i=1}^{n} : e \Downarrow \Gamma_2 : v}{\Gamma_1 : \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e \Downarrow \Gamma_2 : v}\ \text{Let}$$

$$\frac{\Gamma_1 : e \Downarrow \Gamma_2 : v \qquad \texttt{notAbs}\ v}{\Gamma_1 : e\ x \Downarrow \Gamma_2 : \texttt{Exception}}\ \text{EApp}$$

$$\frac{\Gamma_1 : e_1 \Downarrow \Gamma_2 : \lambda x.e_2 \qquad \Gamma_2 : e_2[y/x] \Downarrow \Gamma_3 : v}{\Gamma_1 : e_1\ y \Downarrow \Gamma_3 : v}\ \text{App}$$

$$\frac{\Gamma_1 : e \Downarrow \Gamma_2 : v \qquad \texttt{notCon}\ v\ \{c_i\}_{i=1}^{n}}{\Gamma_1 : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \Downarrow \Gamma_2 : \texttt{Exception}}\ \text{ECase}$$

$$\frac{\Gamma_1 : e \Downarrow \Gamma_2 : c_k\ x_1 \ldots x_{m_k} \qquad \Gamma_2 : e_k[x_i/y_i]_{i=1}^{m_k} \Downarrow \Gamma_3 : v}{\Gamma_1 : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \Downarrow \Gamma_3 : v}\ \text{Case}$$

$$\frac{\Gamma_1 : e_1 \Downarrow \Gamma_2 : v_1 \qquad \Gamma_2 : e_2 \Downarrow \Gamma_3 : v_2}{\Gamma_1 : e_1 \oplus e_2 \Downarrow \Gamma_3 : v_1 \underline{\oplus} v_2}\ \text{Prim}$$

Figure 3: Semantics for lazy evaluation of core language.

$$\frac{\dfrac{\Gamma_1 : f \Downarrow \Gamma_1 : \lambda x'.x' \qquad \dfrac{\Gamma_1 : y \Downarrow \Gamma_1 : 3}{\Gamma_1 : z \Downarrow \Gamma_2 : 3}}{\Gamma_1 : f\ z \Downarrow \Gamma_2 : 3}}{\langle\ \rangle : \texttt{let}\{f = \lambda x.x, y = 3, z = y\}\ \texttt{in}\ f\ z \Downarrow \Gamma_2 : 3}$$

where
$\Gamma_1 = \langle f \mapsto \lambda x.x, y \mapsto 3, z \mapsto y \rangle$
$\Gamma_2 = \langle f \mapsto \lambda x.x, y \mapsto 3, z \mapsto 3 \rangle$

Figure 4: Example evaluation with the rules from the core semantics.

Launchbury shows that any expression can be transformed into this normal form by introducing let-expressions. The bindings in a let-expression are mutually recursive.

In the Var rule the result value is duplicated. To preserve the invariant that all bound variables are distinct, I rename all bound variables with fresh variables in the copy. This renaming is written as $\widehat{v}$.

The App rule makes the language non-strict. The value of the variable y to which the abstraction $\lambda x.e$ is applied is not demanded, instead y is substituted for all occurrences of x in e (for this I use the notation $e[y/x]$). The Var rule prevents repeated evaluation: when variable x is demanded and x maps to expression e, and e evaluates to value v then the heap is updated such that x now maps to v.

I added the rules ECase and EApp to 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. In the Prim rule $\underline{\oplus}$ is the total semantic function associated with the syntactic operator $\oplus$.

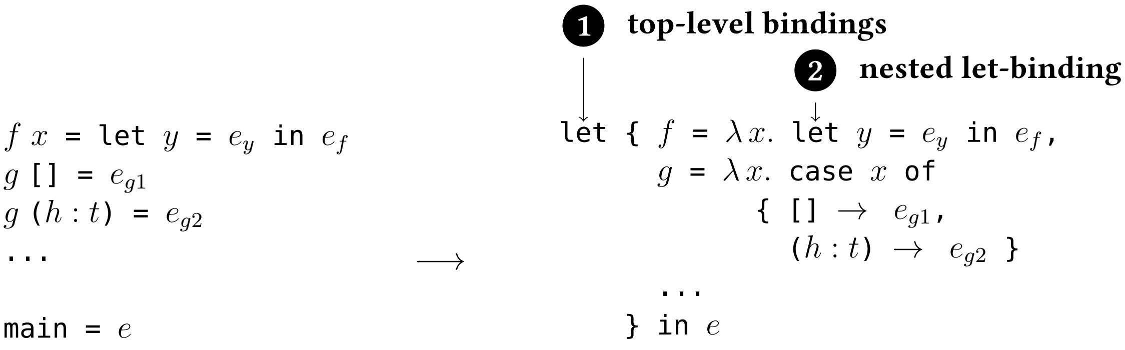
Consider example program $\texttt{let}\{f = \lambda x.x, y = 3, z = y\}\ \texttt{in}\ f\ \ z$. The tree in Figure 4 shows evaluation to value 3 as defined by the core semantics.

2.1.3 The language Haskell

Haskell (Peyton Jones et al. 2003; Marlow et al. 2010; Hudak et al. 2007) has a far more complex syntax than our core language. The goal of these constructs is to make Haskell programs easier to read and write. A Haskell program can be translated to our core language.

Haskell has a powerful type system that helps prevent the programmer making some mistakes. The type system is mostly out of scope for this work, although I do not consider defective programs that could have been prevented by the type systems (e.g. trying to apply a function that expects a number to a functional value) and in Chapter 4 I use type information to genericly define how a value should be recorded in the trace.

Let-bindings can be nested, a *top-level* let-binding is not nested in any other let-binding. Consider for example the Haskell program and its equivalent in the core



```
f x = let y = e_y in e_f
g [] = e_g1
g (h : t) = e_g2
...

main = e
```

$\longrightarrow$

```
let { f = λx. let y = e_y in e_f,
      g = λx. case x of
             { [] → e_g1,
               (h : t) → e_g2 }
      ...
    } in e
```

Figure 5: A Haskell program (left) translated to our core language (right).

language in Figure 5. The functions f and g in the Haskell program are top-level `let`-bound functions ❶ in the core language program. I refer to f and g as top-level functions, and to a nested let-bound function such as y as a *local* function ❷.

An abstraction in Haskell can be defined with the argument left of the '=' sign, such as argument x in the definition of f. Furthermore, the definition may be split in different cases based on the value of the function argument, as for function g.

I explain my tracing methods in this thesis using the core language, and implemented the methods for all of Haskell in the tracer and debugger Hoed. Tracing scales to full Haskell because I observe values and, although Haskell is more complex than my core language, Haskell does not have many more kinds of values than my core language.

2.2 Algorithmic debugging

Algorithmic debugging (Shapiro 1983) is a method for finding defects in programs by systematic search using a two-phase process:

- In the first phase, sufficient information is collected during execution of the defective program to construct a *computation tree*. Each node in the computation tree records a function application and its result — a *computation statement*. I refer to the function f applied in a computation statement S as the *subject function*, and to S itself as an f-statement.

- In the second phase, an *oracle* is asked to judge computation statements. A computation statement is *right* if the evaluated value agrees with the (informal) specification of the programmer for the program, otherwise the statement is *wrong*. The systematic examination of the computation tree in the second phase continues until the oracle has provided enough right/wrong judgements and a *faulty* *node* has been found in the computation tree: the oracle has judged as wrong the statement attached to this node, but has judged as right all the statements it depends on. The definition of the subject function of a faulty node contains a defect.

The computation tree structure must have the following property of algorithmic debugging: if a statement S_0 is wrong than S_0 *depends* on another statement S_1 that is also wrong or the definition of the subject function appearing in S_0 must be defective. If a program misbehaves the root node is wrong. The tree is guaranteed to have a faulty node, if the root node is wrong.

Consider the defective Haskell program in Listing 2. The program contains a parity function ❶ which returns an unexpected result for some argument values due to a defect in the definition of another function ❷: `modTwo n` is incorrectly defined as `n ‘div‘ 2` instead of `n ‘mod‘ 2`. The program includes a property for testing ❸. Using the property-based testing tool QuickCheck (Claessen and Hughes 2000a) we get:

```
> quickCheck prop_notBothOdd
*** Failed! Falsifiable: 2
```

So for argument value 2 the property does not hold.

Listing 2 A defective program with a test property.

```
isOdd n = isEven (plusOne n)     ←—— ❶
isEven n = modTwo n == 0
plusOne n = n + 1
modTwo n = n ‘div‘ 2    ←—— ❷

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

❶ **Parity test**

Defect: modulo operator replaced by divide operator

Evaluates to `True` for all `x` with sound implementation

In this thesis I assume the functions in the program are divided in a set of functions *trusted* by the programmer to be correct and the functions which the programmer *suspects* to contain a defect. For example `isOdd`, `isEven`, `plusOne` and `modTwo` are suspected and `quickCheck`, `div`, `(==)` and `prop_notBothOdd` are trusted. Only suspected functions are recorded in the computation tree and the oracle is not asked to judge statements with trusted functions.

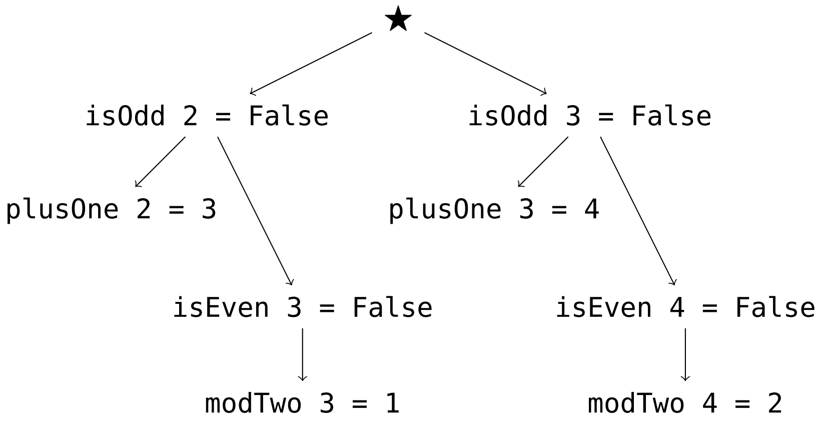


Figure 6: Computation tree for `prop_notBothOdd 2`.

Evaluating a program may entail evaluation of several independent applications of suspected functions, and hence result in multiple subtrees that I connect with a special root node ★. When a program misbehaves at least one of the children of ★ is wrong. Figure 6 shows a computation tree for evaluating the expression `prop_notBothOdd 2` (2 is the value property-based testing found as counter-example). All other nodes of a computation tree are computation statements.

The computation statement `modTwo 4 = 2` is wrong (the expected result is 0) and because that node has no children in the tree, the definition of `modTwo` must be defective, as indeed it is. The statements `isOdd 3 = False` and `isEven 4 = False` are wrong but not necessarily faulty because they each depend on another statement that is also wrong.

2.2.1 Kinds of computation trees

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 my example tree has no nodes for function the progammer *trusts* to be correct, e.g., `(+)` or `prop_notBothOdd`.

Secondly, for higher-order functions different definitions for the parent-child relation of a computation tree exist. Previous research already found two definitions and here I give more variations. To explain the different definitions I use the faulty program from Listing 3 that unexpectedly prints “oops" ❶. In this program the argument `f` of the function `app` is another function ❷.

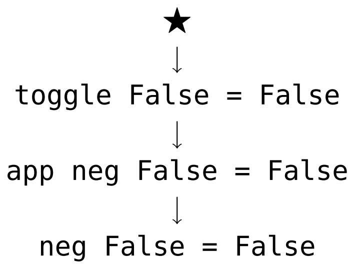
The representation in a computation statement of an argument with a functional value can be *intensional* or *extensional.* The intensional representation of a functional value is a function symbol (e.g. “`neg`"), or a partial application of a function symbol (e.g. “`and False`"). The extensional representation of a functional value is a finite map of arguments and results (e.g. “`{\True -> False; \False -> False}`").

Listing 3 Defective example program with a higher-order function.

```
main :: IO ()
main = print (if (toggle False == True)    ⟵ ❶ Prints “oops!"
              then "ok!" else "oops!")

toggle :: Bool -> Bool
toggle b = app neg b

app :: (Bool->Bool) -> Bool -> Bool
app f b = f b    ⟵ ❷ Higher-order function

neg :: Bool -> Bool
neg b = case b of True -> False; False -> False    ⟵ ❸
```

Defect: if `b` is `False` then `neg` should return `True`

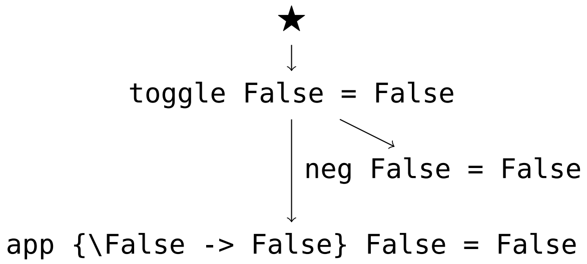
Evaluation dependence tree The first computation tree structure that was proposed for lazy functional languages is the evaluation dependence tree (EDT) (Nilsson and Sparud 1997). The EDT represents functional values intensionally: as function identifiers or partial applications. When in the definition of a function g a higher-order function h is applied to a function f then the computation statement of h depends on the computation statement of f. The EDT of the example program is:



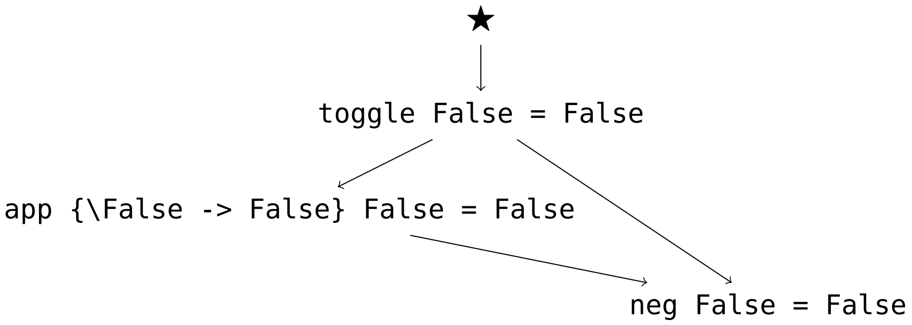
Most algorithmic debuggers for lazy languages (Nilsson and Fritzson 1992; Nilsson 1998; Wallace et al. 2001; Caballero, López-Fraguas and Rodríguez-Artalejo 2001; Braßel and Siegel 2008) construct EDTs.

Function dependence tree 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 (Pope 2005, 2006).

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 $g = \ldots$ 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 and Davie (2008) formally define the corresponding FDT, compare the two tree structures and prove that both have the essential property for algorithmic debugging. The FDT of the example program is:



Acyclic directed computation graph A directed acyclic computation graph over-approximates a computation tree at least contains all arcs from that tree and contains exactly the same nodes as the computation tree it approximates. The following graph is an over-approximation of above FDT tree:



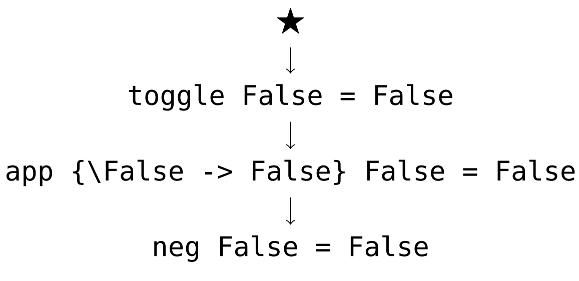
Any acyclic graph that is an over-approximation of either the EDT or the FDT is also sound for algorithmic debugging.

Property 2.1. *Let t be a computation graph with at least one faulty node. Let d be an acyclic directed graph with exactly the same nodes as t and a superset of the edges of t. There exists a faulty node in t that is also faulty in d.*

Let v be a faulty node in t. **Case v has a successor $\mathbf{w_0}$ that is wrong:** let $[w_0 \ldots w_n | w_{(i-1} \rightarrow w_i \wedge \texttt{isWrong}\ \ w_i]$. Because d is acyclic $w_n \neq v$ and $w_n \notin [w_0 \ldots w_{n-1}$. There exists no successor of w that is wrong or it would have been included in $[w_0 \ldots w_n]$. Hence node w_n is faulty in d. Tree t also has node w_n, and w_n has as most the successors w_n has in d, none of which are wrong, hence w_n is also faulty in t. **Case v has no successors in d, or all successors are right:** by definition v is also faulty in d.

In many cases the surplus edges require more judgements from the oracle when using a computation graph compared to using the tree that the graph approximates.

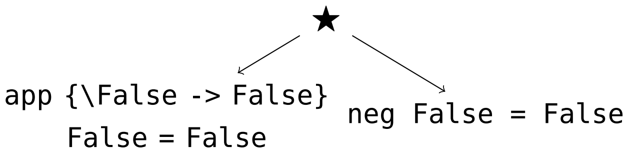
Mixed Representation Mixing the structure of an FDT with the representation of an EDT is not sound in general. Consider for example the following tree:

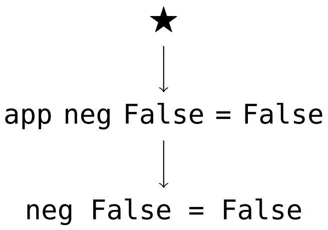


An algorithmic debugger using this tree could incorrectly conclude that the fault is in the definition of the function `toggle` rather than in the definition of `neg`:

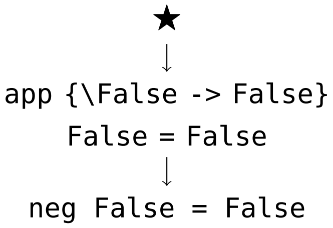
```
toggle False = False ? wrong
app {\False -> False} False = False ? right
Fault detected in function toggle!
```

However, sound mixing of EDT and FDT is possible. Let's consider the `toggle` function in our example program as a trusted function. Tracing evaluation gives the following FTD (left) and EDT (right):





Compare with the following computation tree which mixes EDT with FDT:



The actual fault is in the `neg` function definition. The `app`-statement is right. The `neg`-statement is wrong and because the statement has no children in the computation tree, the statement is identified as faulty. Vice versa, if I assume `app` to be faulty and `neg` 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.

However, this mixed tree does not have the property that, in the computation tree of an observable misbehaving program, if the root of a subtree is judged right that a faulty node can be found in a different part of the tree. This property is used by algorithmic debuggers in the second phase to minimize the number of computation statements the oracle is asked to judge — when a statement is judged right the debugger does not consider the subtree of which the statement is a root of for further inspection. The root of the mixed tree also does not have the property that one of its children is wrong, even though the program misbehaves.

2.2.2 Unevaluated expressions in a computation statement

A term in a computation statement of a lazily evaluated program can be a value or an unevaluated expression. Consider the following computation statement from the program of Listing 4 where I use the symbol _ to represent an unevaluated expression:

```
paint (Square Red _) = Paint Red _
```

When we judge this statement, we need to consider possible *completions* of it, in which the two unevaluated expressions are replaced by values of appropriate types. But which of the following interpretations is correct?

(a) The statement is right if *every* completion makes it right.

(b) The statement is right if *some* completion makes it right.

(c) The statement is right if for every completion of the argument some completion of the result makes it right.

Interpretation (a) requires too much. Statements are about functions, so for any completion of the argument(s) there can be at most one right way to complete the result. Requiring *every* argument completion to be right would include silly alternative results such as:

```
paint (Square Red 2) = Paint Red 345
```

Interpretation (b) requires too little. That the expression is not evaluated (and thus not used to compute the result) may exactly be the reason that the result is not as specified. This interpretation would, for example, allow

```
paint (Rect _ 2 _) = Paint _ 4
```

to be judged right, as it is right when completed (from left-to-right) by `Red`, 2 and `Red`. But the area is not 4 regardless of the height of the rectangle!

Interpretation (c) is correct. We should interpret the `paint` statement as:

$\forall x.\exists y.$ `paint (Square Red` x`) = Paint Square` y

That is, for all values for the unevaluated expressions in the argument, if there are any values for the unevaluated expressions in a computation statements result that would make the computation statement right, then the statement should be judged right (Naish 1992).

2.2.3 Algorithmic debugging strategies

The number of statements an oracle must judge before a faulty node is found varies depending on the shape of the computation tree and the strategy adopted by the algorithmic debugger. A strategy determines an order in which statements in the tree are considered (Shapiro 1983). Algorithmic debugging strategies are not my focus, although I shortly discuss strategies for algorithmic debugging with multiple oracles in Chapter 8.

Two often-used options are a *top-down* strategy, or a *divide-and-query* strategy. The shape of a tree is characterized by the following three properties:

- Number of nodes N in the tree.
- Maximum depth D, the largest number of steps from the root node to another node in the tree.
- Average branch factor B, the average number of children at non-leaf nodes.

A top-down strategy traverses the computation tree depth-first, asking the oracle to judge computation statements in pre-order, starting at the root of the tree (Av-Ron 1984). When a statement is judged wrong we next consider one of the child statements. When a statement is judged as right we return to the parent statement and select another child that has no judgement. If there is no such child the defect is located.

Relatively many computation statements need to be considered using the top-down strategy. The oracle has to judge $B \times D$ statements to find the defect in a tree with depth D and branch factor B (Silva 2007).

A divide-and-query strategy repeatedly selects a statement that splits the computation tree into two parts that are roughly equal in size (Shapiro 1983). That is, we select an unjudged statement at the root of a subtree with roughly half of the unjudged statements. When the statement is right, all statements in the subtree are right. When the statement is wrong we search for the defect in the subtree.

Comparatively few computation statements need to be considered using the divide-and-query strategy. Using this strategy the oracle is asked $B \times \log N$ questions to find a faulty node(Silva 2007). The advantage is smaller when the parts are less equal (consider for example applying the strategy on a tree with all nodes directly under the root). Cabrera (2016) discusses in detail with which tree characteristics the divide-and-query strategy is a good choice.

Despite the apparent advantage of needing fewer judgements, there are other considerations. A top-down strategy provides the user a clearer sense of context, and also of control since judging a statement wrong means “step into this computation” and judging a statement right means “step over this computation”. Using divide-and-query strategy, a human oracle has less context, and less sense of control over the debugging process.

Many other algorithmic debugging strategies have been devised. For example, the ratio of right/wrong judgements of previous statements can be used to predict which of the possible next statements is most likely wrong, based on its subject function (Davie and Chitil 2006b). For an overview of alternative strategies, see (Silva 2007).

Listing 4 Defective program for computing amount of paint needed to fill a shape.

```
data Clr = Red | Blue
data Shape = Square Clr Int | Rect Clr Int Int       | Determines
data Paint = Paint Clr Int                           | colour and
                                                     | amount of paint
paint shape = Paint (colour shape)                  ←| needed to fill a
                    ((width shape) * (height shape)) | shape

colour (Square c x) = c   ←— Desired colour of shape
colour (Rect c w h) = c

width (Square c x) = x
width (Rect c w h) = w

                             | Defect: w and h in Rect-pattern match
height (Square c x) = x      | are swapped
height (Rect c h w)  = h    ←|
```

2.2.4 Oracle

To determine which node is faulty, and thus which part of the code contains a defect, an algorithmic debugger some method to judge a computation statement as right or wrong. Possible methods for judging a computation statement are:

Consulting a human oracle Using the programmer needing to find and correct mistakes as oracle works in the absence of a reference implementation or formal specification and is easy to implement. Papers on algorithmic debugging tend to focus on the construction and soundness of computation trees and many actual implementations of algorithmic debuggers rely on a human oracle. A drawback of using a human oracle is that if the size or the number of questions is too great for the programmer to handle, then the technique breaks down.

Re-using a previous judgement Knowledge provided by the human programmer in the form of right/wrong judgements is lost after a debugging session ends. To retain this knowledge statements and their judgement can be stored in a database (Tamarit et al. 2016). The stored information can both be used to answer questions in a future debugging session and for unit testing.

Tamarit et al. (2016) assume fully evaluated computation statements — they implemented their idea in an algorithmic debugger for a strict language. When a term in a computation statement can be an unevaluated expression some statements are a *more general* version of another statement. Consider for example the following statements:

$\mathbf{S_0}$ `sort [7,8] = 8 : _`

$\mathbf{S_1}$ `sort [7,8] = [8,7]`

$\mathbf{S_2}$ `sort [2,1,3] = 1 : _`

$\mathbf{S_3}$ `sort [2,1,3] = [1,2,3]`

$\mathbf{S_4}$ `sort [2,1,3] = [1,3,2]`

Statement S_0 is a more general version of S_1, it follows therefore that if S_0 is wrong then S_1 is also wrong. Statement S_2 is a more general version of S_3, it follows therefore that if S_3 is right then S_2 is also right.

Brehm (2001) defines a more-general relationship for computation statements and a rule based on this relationship to use a previous judgements. However, Brehm's rule also seem to allow the unsound conclusion that because statement S_2 is a more general version of S_4 it follows that S_4 is right when S_2 is right. Brehm implemented his method in the algorithmic debugger Hat for Haskell but it was removed in a later version.

Comparing with a reference implementation When a program is slow for certain inputs a programmer might want to make changes to the code to make the program run faster. When a defect is introduced during the optimization attempt we have the old correct version of the code and a new defective version of the code.

The idea is to use the old version as reference implementation to debug the new version as follows: given a computation statement from evaluating the new version of the

Listing 5 Defective program to select the smallest from the two values in a tuple (top), and a reference implementation (bottom).

```
tmin x = fst (tsort x)
tsort (x,y) = if x > y then (snd (x,y), x) else (x,y)
fst (x,y) = y    ⟵ Defect causing observed faulty behaviour
snd (x,y) = x    ⟵ Defect that may cause faulty behaviour in
                   other cases
tmin_ref x = fst (tsort_ref x)
tsort_ref (x,y) = if x > y then (snd (x,y), x) else (x,y)
fst_ref (x,y) = x
snd_ref (x,y) = y
```

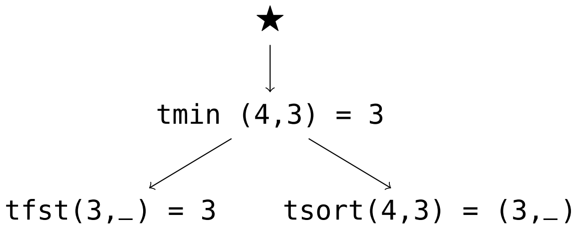


Figure 7: Computation tree for `tmin (4,3)`.

program we repeat the computation with the subject function's reference implementation. If the recorded result differs from the result of the reference implementation, then the computation statement is wrong and otherwise the statement is right.

Consider for example the computation statement

```
tmin (4,3) = 3
```

from Listing 5. The reference implementation gives

```
tmin_ref (4,3) = 4
```

and hence the `tmin`-statement is wrong.

Although using a reference implementation as oracle is often mentioned (Shapiro 1983; Drabent and Nadjm-Tehrani 1989; Nilsson and Fritzson 1992), to my knowledge, there is no work that specifies exactly how such an implementation is soundly used in a lazy context. For example, consider the following statement:

```
tsort (4,3) = (3,_)
```

We could force the computation and obtain:

```
tsort (4,3) = (3,3)
```

Which gives us a result different from the reference computation `tsort_ref (4,3) = (3,4)`. However, this leads to the incorrect conclusion that `tsort` is faulty because the defect in `snd` was not part of the computation `tmin (4,3)` for which we observed faulty behaviour, hence there is no `snd`-statement in the computation tree (Figure 7). Furthermore, a forced computation may not even terminate.

Using a formal specification A formal specification is another automated method suggested in several works for judging computation statements (Shapiro 1983; Drabent and Nadjm-Tehrani 1989; Nilsson and Fritzson 1992). A formal specification differs from a reference implementation in that it checks some desired property of the function. Such a property could be “The result of applying `sort` to a list is an ordered list". While encoding properties as Haskell functions is common practice for testing purposes (Claessen and Hughes 2000b), however to my knowledge there is no existing work that describes how these properties – or any other encoding – can be used as oracle for algorithmic debugging of a lazy functional program.

3 Value observation tracing

A lazily evaluated functional program can be modularized into a generator part that constructs a large number of possible answers and a selector part that selects the right answer. Consider the Haskell program

```
main = sel (gen 3)
```

where the function `gen` computes intermediate values that are used as inputs for the function `sel`. We could first run `gen` and store all intermediate values in memory, and then apply `sel` to the values. However, `gen` might construct so many intermediate values that not all values can fit in memory at the same time.

Because the code of a program modularised into a generator and selector part is simpler, the programmer is less likely to make a mistake, however mistakes can still be made. When a program misbehaves we cannot tell from the output whether the selector or the generator part is defective. To find out which part is defective we need to know the otherwise hidden intermediate values produced by the generator part and demanded by the selector part of my program.

To observe intermediate values while preserving the semantics for lazy evaluation Gill (2000) developed a library called HOOD. With HOOD the programmer can mark certain expressions for *observation*. The values to which an observed expression evaluates are recorded and can be inspected by the programmer after termination of the program.

3.1 Finding a defect with value observation tracing

Consider the defective program in Listing 6. The generator function `mkTreeRat` constructs a sorted infinite binary tree of rational numbers (the Stern-Brocot tree) and the selector function `toFloat` uses the tree for finding the rational number that represents a floating point number. For completeness my `TreeRat` type also has a `Leaf` constructor, even though the tree I define is infinite and has no leafs. Floating point number 0.75 can for example be represented by the rational number $\frac{3}{4}$ (for which I use the notation `3 :% 4` in the program). However, when I run the program it prints the unexpected result `1 :% 2` instead. How do I find out if the defect is in the definition of the generator function or in the definition of the selector function?

Listing 6 A defective program for finding a rational representation of a floating point number using an sorted infinite tree of rational numbers.

```
data TreeRat = Node Rational TreeRat TreeRat | Leaf
data Rational = Integer :% Integer
 deriving Show

mkTreeRat :: Integer -> Integer -> Integer -> Integer -> TreeRat
mkTreeRat a b c d =
 Node (x :% y) (mkTreeRat a b x y) (mkTreeRat x y c d)
 where x = a+c
       y = b+d

toFloat :: TreeRat -> Float -> Rational
toFloat (Node x left right) y
 | delta <= 0 = x                ⟵ Defect: operator (<=) should be (==)
 | delta > 0  = toFloat left  y
 | otherwise  = toFloat right y
 where delta = (fromRational x) - y

main = print (toFloat (mkTreeRat 0 1 1 0) 0.75)
```

When I could inspect the intermediate values that are used as inputs for the selector function I could see if these are right (the defect is in the selector) or wrong (the defect is in the generator).

In a naive attempt to inspect the intermediate values I derive `Show` for the `TreeRat` type and try to reveal the intermediate values with

```
main = print (mkTreeRat 0 1 1 0)
```

```
-- Combinators used to make observations
runO :: IO a -> IO ()
observe :: Observable a => String -> a -> a

-- Every observed value has to be of a type that is an
-- instance of this class
class Observable a where
  obs :: a -> Parent -> a

-- Helper functions to implement an instance of Observable
send :: String -> ObserverM a -> Parent -> a
(<<) :: (Observable a) => ObserverM (a->b) -> a -> ObserverM b

-- Abstract types used above
type Parent
type ObserverM
```

Figure 8: Essential parts of the HOOD API.

however, `print` tries to `show` all values in the infinite tree constructed by `mkTreeRat` and hence this program never terminates.

Value observation tracing preserves the semantics for lazy evaluation and reveals only the intermediate values that are actually demanded by the selector function. HOOD implements value observation tracing in Haskell (Gill 2000). HOOD is unobtrusive: it is just a library and requires no changes to the run-time system. Figure 8 shows the essential parts of the library's API.

The two main combinators of the library are `observe` and `runO`. The function `observe` takes a label as parameter and then behaves like the identity function; as side effect a value is observed and associated with the label. The `runO` function evaluates its argument expression and afterwards uses the trace of events to print the observed values. The rest of the API is used to define how values are observed, I discuss this in more detail in Section 3.4. In the example program I add the following underlined annotations to observe the `toFloat` function:

```
main = runO (print (toFloat’ (mkNode 0 1 1 0) 0.75))
 where toFloat’ = observe "toFloat" toFloat
```

After running my program as usual `runO` prints Figure 9 with the representation of the observed arguments and result values. The symbol _ is used to represent a part of the

```
toFloat (Node (1 :% 1) (Node (1 :% 2) _ _) _) 0.75 = 1 :% 2
```

Figure 9: Recorded arguments and result values of `toFloat`.

tree that is not used in this context of the evaluated function.

From these recorded values I can use the specification of the Stern-Brocot tree (Graham, Knuth and Patashnik 1989; Gibbons, Lester and Bird 2006) to conclude that the argument of `toFloat`, the part of the tree evaluated in this context, is correct while the result is not correct. Hence, the defect is not in the generator function `mkTreeRat` and must therefore be in the selector function `toFloat`.

3.2 Under the hood of value observation tracing

To explain what is exactly recorded in the trace by an `observe` annotation I define a small language and with a semantics I specify how expressions from my language are lazily evaluated with generation of a trace. For the purpose of explaining my method I use a language that has not as many different expressions as Haskell, however value observation based tracing scales to full Haskell because Haskell has only few different sorts of values.

3.2.1 Language

In Figure 10 I extend Launchbury's language with expressions for observing values. To annotate an expression e with an identifier f the programmer uses `observe` f e. In principle the programmer is free to annotate any expression with any identifier, however for the construction of computation statements for algorithmic debugging I require a more systematic approach labelling e in function definition $f = e$ with identifier f (I discuss construction of computation statements in Section 3.3).

The `obs` expressions and $\mathtt{obs}_\lambda$ values should not appear in a program. Rather they are introduced by evaluation of an `observe` expressions. The field p (for parent) is used to keep track of which parts of the trace belong together and is discussed in more detail in the next section. A single applied data constructor or a single λ-abstraction may be observed several times; the latter case leads to values such as $\mathtt{obs}_\lambda\ p_1\ (\mathtt{obs}_\lambda\ p_2\ (\ldots(\lambda x.e)\ldots))$ during evaluation. My semantics scales to the many different expressions in Haskell, because I observe only values and Haskell has only few different sorts of values.

expression e	
$::= v$	value
$\mid e\ x$	application
$\mid$ `let` $\{x_k = e_k\}_{k=1}^{n}$ `in` e	recursive binding
$\mid$ `case` e `of` $\{c_k\ x_1 \dots x_{m_k} \to e_k\}_{k=1}^{n}$	case
$\mid x$	variable
$\mid x_1 \oplus x_2$	application of a primitive
$\mid$ **`observe` f e**	**labelled expression**
$\mid$ **`obs` p e**	**observed expression**
value v	
$::= \boldsymbol{v_\lambda}$	**functional value**
$\mid c\ x_1 \dots x_n$	saturated application of data constructor
functional value $\boldsymbol{v_\lambda}$	
$::= \lambda x.e$	abstraction
$\mid$ **`obs`$_\lambda$ p v$_\lambda$**	**observed functional value**

Figure 10: Syntax of the core language extended (differences in **bold**) to label and observe expressions.

trace	$\mathcal{T}$	$::= t_0, \dots, t_n$	sequence growing right
event number	i	$\in \{0, \dots, n\}$	refers to an event in the trace
trace event	t	$::= i{:}\mathrm{Root}\ f$	root with label f
		$\mid\ i{:}\mathrm{Con}\ p\ c\ a$	value is application of constructor c
		$\mid\ i{:}\mathrm{Lam}\ p$	value is an abstraction
		$\mid\ i{:}\mathrm{MapsTo}\ p$	function application
parent	p	$::= \mathrm{P}\ i$	parent is event $i{:}\mathrm{Root}\ f$ or $i{:}\mathrm{Lam}\ p'$
		$\mid\ \mathrm{P_c}\ i\ m$	argument m; parent is constructor event $i{:}\mathrm{Con}\ p'\ d\ a$ with $m \leq a$
		$\mid\ \mathrm{P_a}\ i$	argument
		$\mid\ \mathrm{P_r}\ i$	result

Figure 11: Syntax of the trace and its events.

3.2.2 A trace of events

A trace is a sequence of events, as defined in Figure 11. A value is observed by adding events to the trace. Evaluating for example `let {x = 3 :% 4} in fromRational (observe "x" x)` gives the trace

1: Root "x"
2: Con (P 1) ":%" 2
3: Con ($\mathrm{P_c}$ 2 2) "4" 0
4: Con ($\mathrm{P_c}$ 2 1) "3" 0

Each event has a unique event number i that corresponds one-to-one to the index of the event in the trace. Several `observe` annotations may add events that occur interleaved in the trace; and as we see in the example above even the events describing a single value may occur in a different order than we might expect. To identify which events belong to which observation, every event, except the root events, contains a parent field p. This field both identifies which event is their parent and what the role of the child event with respect to the parent event is. Following Gills terminology I call the latter its *port*. For example, the parent field $\mathrm{P_c}\ 2\ 1$ of event 4 above tells us that event 4 is the first argument of the constructor recorded by event 2.

An i : Root f event records the label with which the programmer labeled an expression. The first event in the example trace above records the label "x".

An event i : Con $p\ d\ a$ records that the value is a saturated application of a constructor c where d is the representation of that constructor. A constructor may be the parent of up to a children, each child event with a parent field $\mathrm{P_c}\ i\ m$ such that $1 \leq m \leq a$. Because lazy evaluation may not evaluate some data constructor arguments, the event i : Con $p\ d\ a$ does not necessarily have all m children.

I can also observe functional values: either directly, as argument to a constructor or as argument or result of another functional value. Functional values are recorded extensionally, that is, as a finite map from argument to result value. An i : Lam p event has one or more i' : MapsTo p' events as children. Each i' : MapsTo p' event corresponds to an application of the observed function. A function application always has exactly one result value (which in turn may have more children) and may have one optional argument value. Consider for example the program

```
let f = observe "f" (λ x . 7); y = 8 in f y
```

which evaluates to the following trace:

1: Root “`f`”
2: Lam (P 1)
3: MapsTo (P 2)
4: Con ($\mathrm{P_r}$ 3) “`7`” 0

3.2.3 Semantics for lazy evaluation with generation of a trace

Figure 12 extends the semantics for lazy evaluation of the core language from Section 2.1.2 with a trace. The rules of the extended semantics define how trace $\mathcal{T}_1$ and an expression e in the context of heap Γ_1 are evaluated to value v, heap Γ_2 and trace $\mathcal{T}_2$. The initial heap and trace are empty. The resulting trace $\mathcal{T}_2$ is the sequence of events that correspond to the values to which the observed expressions in e evaluated in the context of evaluating e to v.

The nine rules at the top (Lam, Con, Let, Var, EApp, App, ECase, Case and Prim) are similar to the rules with the same names in the semantics of Section 2.1.2. I added a trace as an additional global state that is passed and possibly changed by a subevaluation but not by the rules themselves. As before I require that all bound variables of an expression are distinct. For y_1 to y_n in ObsCon and y in ObsApp I pick fresh variables, just like the $\hat{v}$ in the Var rule.

Finally consider the last four rules that four rules define how the trace is constructed. $t_0, \ldots, t_n \lessdot t = t_0, \ldots, t_n, t$ 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 $i : \mathrm{Root}\ f$ event and wraps the observed expression with the pseudo-function `obs`. The index of the $i : \mathrm{Root}\ f$ event is passed to `obs` to enable connecting to the parent event later.

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

So only when evaluation reaches a constructor application that constructor application 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. I

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

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

$$\frac{\Gamma_1, \boldsymbol{\mathcal{T}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : v}{\Gamma_1[x \mapsto e], \boldsymbol{\mathcal{T}_1} : x \;\Downarrow\; \Gamma_2[x \mapsto v], \boldsymbol{\mathcal{T}_2} : \hat{v}} \text{ Var}$$

$$\frac{\Gamma_1[x_i \mapsto e_i]_{i=1}^{n}, \boldsymbol{\mathcal{T}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : v}{\Gamma_1, \boldsymbol{\mathcal{T}_1} : \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : v} \text{ Let}$$

$$\frac{\Gamma, \boldsymbol{\mathcal{T}_1} : e \;\Downarrow\; \Gamma', \boldsymbol{\mathcal{T}_2} : v \qquad \texttt{notAbs}\ v}{\Gamma, \boldsymbol{\mathcal{T}_1} : e\ x \;\Downarrow\; \Gamma', \boldsymbol{\mathcal{T}_2} : \texttt{Exception}} \text{ EApp}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{T}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : \lambda x.e' \qquad \Gamma_2, \boldsymbol{\mathcal{T}_2} : e'[y/x] \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{T}_3} : v}{\Gamma_1, \boldsymbol{\mathcal{T}_1} : e\ y \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{T}_3} : v} \text{ App}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{T}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : v \qquad \texttt{notCon}\ v\ \{c_i\}_{i=1}^{n}}{\Gamma_1, \boldsymbol{\mathcal{T}_1} : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : \texttt{Exception}} \text{ ECase}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{T}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : c_k\ x_1 \ldots x_{m_k} \qquad \Gamma_2, \boldsymbol{\mathcal{T}_2} : e_k[x_i/y_i]_{i=1}^{m_k} \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{T}_3} : v}{\Gamma_1, \boldsymbol{\mathcal{T}_1} : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{T}_3} : v} \text{ Case}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{T}_1} : e_1 \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{T}_2} : v_1 \qquad \Gamma_2, \boldsymbol{\mathcal{T}_2} : e_2 \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{T}_3} : v_2}{\Gamma_1, \boldsymbol{\mathcal{T}_1} : e_1 \oplus e_2 \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{T}_3} : v_1 \underline{\oplus} v_2} \text{ Prim}$$

$$\frac{\boldsymbol{\Gamma_1, \mathcal{T}_1 \lessdot (i{:}\mathbf{Root}\ f) : \texttt{obs}\ (\mathbf{P}\ i)\ e \;\Downarrow\; \Gamma_2, \mathcal{T}_2 : v \qquad i = |\mathcal{T}_1|}}{\boldsymbol{\Gamma_1, \mathcal{T}_1 : \texttt{observe}\ f\ e \;\Downarrow\; \Gamma_2, \mathcal{T}_2 : v}} \textbf{ Observe}$$

$$\frac{\boldsymbol{\Gamma_1, \mathcal{T}_1 : e \;\Downarrow\; \Gamma_2, \mathcal{T}_2 : c\ x_1 \ldots x_n \qquad i = |\mathcal{T}_2|}}{\begin{array}{c}\boldsymbol{\Gamma_1, \mathcal{T}_1 : \texttt{obs}\ p\ e \;\Downarrow\; \Gamma_2[y_1 \mapsto \texttt{obs}\ (\mathbf{P_c}\ i\ 1)\ x_1, \ldots, y_n \mapsto \texttt{obs}\ (\mathbf{P_c}\ i\ n)\ x_n],} \\ \boldsymbol{\mathcal{T}_2 \lessdot (i{:}\mathbf{Con}\ p\ c\ (\mathbf{arity}\ c)) : c\ y_1 \ldots y_n}\end{array}} \textbf{ ObsCon}$$

$$\frac{\boldsymbol{\Gamma_1, \mathcal{T}_1 : e \;\Downarrow\; \Gamma_2, \mathcal{T}_2 : v_\lambda \qquad i = |\mathcal{T}_2|}}{\boldsymbol{\Gamma_1, \mathcal{T}_1 : \texttt{obs}\ p\ e \;\Downarrow\; \Gamma_2, \mathcal{T}_2 \lessdot (i{:}\mathbf{Lam}\ p) : \texttt{obs}_\lambda\ (\mathbf{P}\ i)\ v_\lambda}} \textbf{ ObsLam}$$

$$\frac{\boldsymbol{\Gamma_1, \mathcal{T}_1 : e \;\Downarrow\; \Gamma_2, \mathcal{T}_2 : \texttt{obs}_\lambda\ p\ v_\lambda} \qquad \begin{array}{l}\boldsymbol{\Gamma_2[y \mapsto \texttt{obs}\ (\mathbf{P_a}\ i)\ x],} \\ \boldsymbol{\mathcal{T}_2 \lessdot (i{:}\mathbf{MapsTo}\ p) :} \\ \boldsymbol{\texttt{obs}\ (\mathbf{P_r}\ i)\ (v_\lambda\ y) \;\Downarrow\; \Gamma_3, \mathcal{T}_3 : v}\end{array} \qquad \boldsymbol{i = |\mathcal{T}_2|}}{\boldsymbol{\Gamma_1, \mathcal{T}_1 : e\ x \;\Downarrow\; \Gamma_3, \mathcal{T}_3 : v}} \textbf{ ObsApp}$$

Figure 12: Rules added to Launchbury's semantics of Figure 3 for lazy evaluation with generation of a trace. New rules and difference to existing rules in **bold**.

1:Root "toFloat"
2:Lam (P 1)
3:MapsTo (P 2)
4:Lam P_r 3
5:MapsTo (P 4)
6:Con (P_r 4) ":%" 2
7:Con (P_a 3) "Node" 3
8:Con (P_c 7 1) ":%" 2
9:Con (P_c 8 1) "1" 0
10:Con (P_c 8 2) "1" 0
11:Con (P_r 5) "0.75" 0
12:Con (P_c 7 2) "Node" 3
13:Con (P_c 12 1) ":%" 2
14:Con (P_c 13 1) "1" 0
15:Con (P_c 13 2) "2" 0
16:Con (P_c 6 1) "1" 0
17:Con (P_c 6 2) "2" 0

Figure 13: Trace obtained by evaluating introductory example.

have this asymmetry, because my 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.

3.2.4 Example trace

During the evaluation of the annotated program from Section 3.1 the `observe` annotation records, as defined in the semantics of Section 3.2, the event trace listed in Figure 13.

Note how for example event 7 records an arity of 3 but only has two children (event 7 and 12) from which I conclude that the expression that describes the right-tree was not demanded in this context, and hence a _ is printed.

3.3 From trace to 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. I assume that the label is the name of the observed function, as in the example. Then a single argument-result pair of a finite map gives rise to a computation statement. I construct the nodes of the computation tree in two steps: First I translate the event trace into a forest of event trees. Subsequently I translate the forest into nodes of the computation tree.

Note that an *event* tree and a *computation* tree are two very different structures. An algorithmic debugger still needs the dependencies between computation statements to

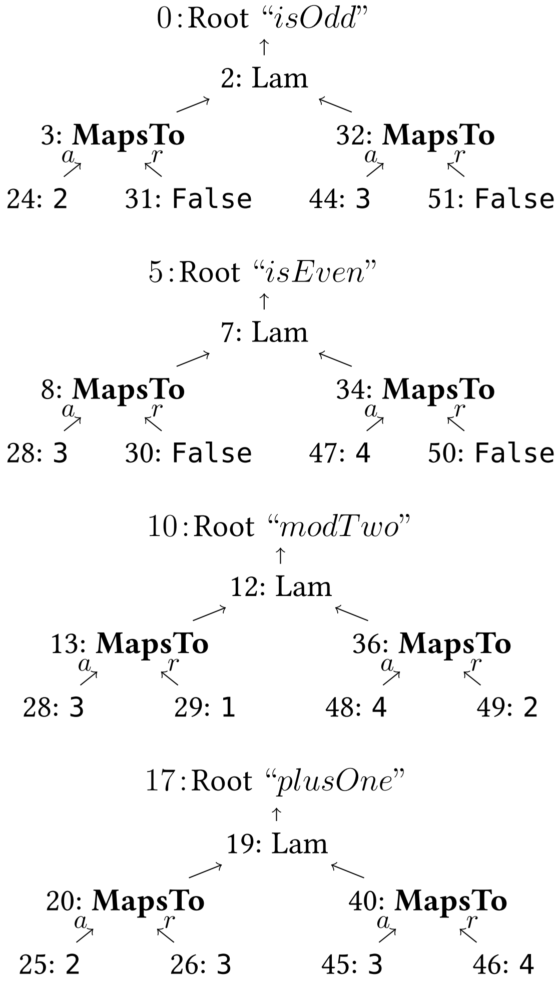


Figure 14: Trace of Figure 35 shown as forest of event trees.

determine which question to ask and eventually to conclude which part of the code is faulty. The construction of a computation tree is discussed in Chapter 5 and 6.

3.3.1 Event trees

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 14 shows the four event trees that I obtain from the trace of Figure 35. Because parent fields determine the tree edges, I 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 the 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 34 I obtain the following properties of an event tree:

- An i : Root f node has exactly one child (by the Observe rule). Because I 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 (by the ObsLam rule).

- 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 (by the ObsApp rule).

- An i : Con p c a node has at most one child with parent $\mathrm{P_c}$ i k for every $k \in \{1, \ldots, a\}$ (by the ObsCon rule). Recall that a is the arity of constructor c.

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

Because I 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.

3.3.2 Constructing the nodes of the computation tree

The nodes of the computation tree are computation statements. Figure 15 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 I define it as a finite set of singleton maps.

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

Figure 15: Abstract syntax of computation statements.

$$
\text{mkStmts} \left(\begin{array}{ccc}
 & i : \text{Root } f & \\
 & \uparrow & \\
 & j : \text{Lam} & \\
 \nearrow & & \nwarrow \\
 et_{(\text{P } j)} & \cdots & et_{(\text{P } j)}
\end{array} \right) = f = \text{mkSMap } \ et_{(\text{P } j)}, \ldots, f = \text{mkSMap } \ et_{(\text{P } j)}\}
$$

$$
\text{mkSMap} \left(\begin{array}{ccc}
 & i : \text{MapsTo} & \\
 \overset{a}{\nearrow} & & \overset{r}{\nwarrow} \\
 et_{(\text{P}_\text{a}\ i)} & & et_{(\text{P}_\text{r}\ i)}
\end{array} \right) = \text{mkVal } \ et_{(\text{P}_\text{a}\ i)} \mapsto \text{mkVal } \ et_{(\text{P}_\text{r}\ i)}
$$

$$
\text{mkVal} \left(\begin{array}{ccc}
 & i : \text{Con } c\ a & \\
 \overset{1}{\nearrow} & & \overset{a}{\nwarrow} \\
 et_{(\text{P}_\text{c}\ i\ 1)} & \cdots & et_{(\text{P}_\text{c}\ i\ a)}
\end{array} \right) = c\ (\text{mkVal } \ et_{(\text{P}_\text{c}\ i\ 1)}, \ldots, \text{mkVal } \ et_{(\text{P}_\text{c}\ i\ a)}
$$

$$
\text{mkVal} \left(\begin{array}{ccc}
 & i : \text{Lam} & \\
 \nearrow & & \nwarrow \\
 et_{(\text{P } i)} & \cdots & et_{(\text{P } i)}
\end{array} \right) = \{\text{mkSMap } \ et_{(\text{P } i)}, \ldots, \text{mkSMap } \ et_{(\text{P } i)}\}
$$

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

Figure 16: From event tree to computation statements.

The algorithm of Figure 16 constructs computation statements from an event tree. I 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 I 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 14 I obtain the eight computation statements

$$\begin{array}{rl}
\texttt{isOdd} = \texttt{2} \mapsto \texttt{False} \qquad & \texttt{isOdd} = \texttt{3} \mapsto \texttt{False} \\
\texttt{isEven} = \texttt{3} \mapsto \texttt{False} \qquad & \texttt{isEven} = \texttt{4} \mapsto \texttt{False} \\
\texttt{modTwo} = \texttt{3} \mapsto \texttt{1} \qquad & \texttt{modTwo} = \texttt{4} \mapsto \texttt{2} \\
\texttt{plusOne} = \texttt{2} \mapsto \texttt{3} \qquad & \texttt{plusOne} = \texttt{3} \mapsto \texttt{4}
\end{array}$$

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.

3.4 Implementing value observation tracing

In this section I explain how my semantics for value observation tracing can be implemented in Haskell. The presented implementation is a variation on Gill's tracing library HOOD[1].

The data constructors of Listing 7 implement the syntax of the trace from Figure 11. Labels and constructor representations are implemented with a `String`. The syntax of Haskell is more complex than the syntax of the language in Figure 10, however, value observation tracing scales to full Haskell because values in Haskell are the same as in my language.

The trace that is passed in the rules of the semantics from Figure 12 is implemented using an `IORef` which is written to as a side-effect. Listing 8 shows the implementation of $\mathcal{T} \lessdot t$ in Haskell. Notice how `addEvent` returns $|\mathcal{T}|$, that is, the index of t in the trace.

The function `observe` is not parametrically polymorphic in the Haskell implementation because it is impossible to define in a single definition how to define a value of any type. Instead I introduce a class `Observable` and `observe` can be applied to any expression of which the type is `Observable` (Listing 9).

[1] The sourcecode of HOOD can be downloaded from `http://hackage.haskell.org/package/hood`, note that the latest version of HOOD is extended with the type-generic techniques I present in the next chapter.

Listing 7 Syntax of the trace implemented as Haskell data constructors.

```
type Trace = [Event]

data Event = Root   String
           | Con    Parent String Int
           | Lam    Parent
           | MapsTo Parent

data Parent = P  Int
            | Pc Int Int
            | Pa Int
            | Pr Int
```

Listing 8 Implementing a trace and the primitive $\mathcal{T} \lessdot t$ in Haskell.

```
addEvent :: Event -> IO Int
addEvent t = do
  trace <- readIORef traceRef
  writeIORef traceRef (t:trace)
  return (length trace)

traceRef :: IORef Trace
traceRef = unsafePerformIO (newIORef [])
```

Listing 9 Implementing the Observe rule.

```
observe :: (Observable a) => String -> a -> a
observe label f = unsafePerformIO $ do
  idx <- addEvent (Root label)
  return (obs f (P idx))
```

Every data constructor application needs to be observed in a slightly different way. The `Observable` class allows us to specify the ObsCon rule per type by providing an implementation of the `obs` method (Listing 10). A Haskell compiler, or interpreter, chooses between the the various ObsCon implementations based on the type of the expression `obs` is applied to.

Listing 10 The `obs` method.

```
class Observable a where
  obs  :: a -> Parent -> a
```

The ObsCon rule in Figure 12 defines how the application of a constructor c applied to $x_1, \ldots, x_n$ records the arity of c, and passes $\mathrm{P_c}\ i\ j$ with the index j of x_j for further observation of the value of x_j. Both can be obtained with the state monad `ObserverM` from Listing 11.

Listing 11 A state monad for counting arguments.

```
newtype ObserverM a = ObserverM  runMO :: Int -> Int -> (a,Int)

instance Functor ObserverM where
  fmap  = liftM

instance Applicative ObserverM where
  pure  = return
  (<*>) = ap

instance Monad ObserverM where
  return a = ObserverM (\c i -> (a,i))
  fn >>= k = ObserverM (\c i -> case runMO fn c i of
                                 (r,i2) -> runMO (k r) c i2)
```

Listing 12 provides the Haskell implementation `obsCon` of the ObsCon rule for a constructor c given:

- `c`, the representation of constructor c
- `fn`, the data constructor c wrapped in the state monad, observing the arguments to which c is applied

Listing 12 Helper function to implement instances of the ObsCon rule

```
obsCon :: String -> ObserverM a -> Parent -> a
obsCon c fn p = unsafePerformIO $ do
  trace <- readIORef traceRef
  let (v,a) = runMO fn (length trace) 0
  idx <- addEvent (Con p c a)
  return v
```

- `p`, the parent of the event that is added to the trace

Now I have all ingredients for writing an instance of the `obs` method. The simple case is to observe a data constructor with arity 0. For example a constant value such as the integer 123 has a representation `c = "123"` and has no arguments to observe, hence `fn = return 123`. Thus, the ObsCon rule for values of types `Int` and `Float` can be implemented as:

```
instance Observable Int   where obs x = obsCon (show x) (return x)

instance Observable Float where obs x = obsCon (show x) (return x)
```

Other implementations of the ObsCon rule for constants, that is data constructors with arity 0, follow the same scheme.

How do I observe data constructors with an arity greater than 0? Let us consider for example the `TreeRat` type (from the example of Listing 6) with the `Node` and `Leaf` data constructors. Evaluating an observed expression e of type `TreeRat` by the ObsCon rule is either evaluated as

$$\frac{\Gamma_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{T}_2 : \texttt{Node}\ x_1\ x_2\ x_3 \qquad i = |\mathcal{T}_2|}{\begin{aligned}\Gamma_1, \mathcal{T}_1 : \texttt{obs}\ p\ e \Downarrow{} & \Gamma_2[y_1 \mapsto \texttt{obs}\ (\mathrm{P_c}\ i\ 1)\ x_1, y_2 \mapsto \texttt{obs}\ (\mathrm{P_c}\ i\ 2)\ x_2, \\ & y_3 \mapsto \texttt{obs}\ (\mathrm{P_c}\ i\ 3)\ x_3], \mathcal{T}_2 \lessdot (i : \mathrm{Con}\ p\ \text{“}\texttt{Node}\text{”}\ 3) : \texttt{Node}\ y_1\ y_2\ y_3\end{aligned}}$$

or as

$$\frac{\Gamma_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{T}_2 : \texttt{Leaf} \qquad i = |\mathcal{T}_2|}{\Gamma_1, \mathcal{T}_1 : \texttt{obs}\ p\ e \Downarrow \Gamma_2, \mathcal{T}_2 \lessdot (i : \mathrm{Con}\ p\ \text{“}\texttt{Leaf}\text{”}\ 0) : \texttt{Leaf}}$$

To implement the behaviour of the ObsCon rule from Figure 12 in Haskell for these two types the programmer writes an instance of the `obs` method. The goal of each instance of class `Observable` is twofold:

1. The obs records in the trace a message with a string representation of the data constructor of the value, such as “Node" or “Leaf". I cannot use show for value with an internal structure, such as a Node-value, because it would change the semantics by demanding the value of the variables Node is applied to.

2. The obs puts further obss on the constructor arguments, the components of the value value. Each argument is observed with a different port. Finally constructor and observed arguments have to be recombined to a value.

To record an event in the trace the HOOD library provides the function send. The send function takes the message to record, the value “wrapped” in the type ObserverM and the parent for the event. The latter is given to it from the obs function itself.

The type ObserverM is a state monad that counts the constructor arguments of the observed value and thus gives each the correct port number. The data constructor's arguments may be evaluated in arbitrary order; an argument may not even be evaluated at all. Hence port numbers are assigned to each of the arguments' obs methods. For example when observing the value Node x_1 x_2 x_3 the obs of argument x_1 is assigned port number 1, x_2 gets port number 2 and x_3 port number 3.

Listing 13 provides two helper functions that make the ObserverM monad easier to use. To assign increasing port numbers to constructor arguments the function thunk is used. Using the ObserverM monad can become rather involved for constructors with many arguments, therefore the helper function (<<) can be used when defining an instance of the obs method.

Listing 13 Helper functions for counting data constructor arity

```
thunk :: (Observable a) => a -> ObserverM a
thunk a = ObserverM $ \ i m -> (obs a (Pc i m), m+1)

(<<) :: (Observable a) => ObserverM (a -> b) -> a -> ObserverM b
fn << a = do  fn’ <- fn ; a’ <- thunk a ; return (fn’ a’)
```

To place applications of the obs method on the constructor arguments of a value, the value is decomposed by pattern matching, e.g. into the constructor Node and the arguments x1, x2 and x3. From the observed arguments and the original constructor a transformed value can easily be reassambled.

Overall, the definition for observing the tree of the example is:

```
instance Observable TreeRat where
```

```
  obs (Node x1 x2 x3) = obsCon "Node" (return Node << x1 << x2 << x3)
  obs Leaf            = obsCon "Leaf" (return Leaf)
```

Similar, for values of type `Rat` the ObsCon rule can be implemented as:

```
 instance Observable Rat where
   obs ((:%) x1 x2) = obsCon "(:%)" (return (:%) << x1 << x2)
```

Functional values are recorded as a finite map from argument value to result value, as defined in the ObsLam and ObsApp rules. In Listing 14 I implement these rules by providing an instance of the `obs` method for the function type.

Listing 14 Implementation of the ObsLam and ObsApp rules

```
instance (Observable a,Observable b) => Observable (a -> b) where
  obs fn p = unsafePerformIO $ do
    idx <- addEvent (Lam p)
    return (obsLam fn (P idx))
    where
    obsLam fn p arg = obsApp (do arg <- thunk arg
                                 thunk (fn arg)) p

obsApp :: ObserverM a -> Parent -> a
obsApp e p = unsafePerformIO $ do
  trace <- readIORef traceRef
  let (v,_) = runMO e (length trace) 0
  idx <- addEvent (MapsTo p)
  return v
```

Finally, a value observation tracing library needs a place in which it can render the trace or start an algorithmic debugger. For this the `runO` annotation is used. Listing 15 gives an example implementation that prints the list of events and their indices. Instead `render` could also print a list of computation statements (with the method from Section 3.3) or it could use the trace to construct a computation tree (with the method of Chapter 5 or 6) and start an algorithmic debugging session.

Listing 15 Implemenation of the `runO` annotation with a simple `render` function

```
runO prgm = do
  prgm
  trace <- readIORef traceRef
  putStr (render trace)

render :: Trace -> String
render trc = concat $ map showEvent (zip [0..] (reverse trc))
  where
  showEvent (i,e) = show i ++ ": " ++ show e ++ "\n"
```

4 Type generic value observation tracing

Although writing an instance of class `Observable` is quite schematic, the requirement to do so has limited the use of HOOD in practice substantially:

1. Writing all required instances is a substantial burden to the programmer. Imagine the work required for observing an abstract syntax tree of a compiler.

2. During program development programmers tend to change the definitions of data types. Having to adapt the instances of `Observable` too is another burden.

3. The programmer may accidentally write a wrong instance of class `Observable`. Bugs in the debugging tool can make debugging a very hard task.

A wrong instance can easily be written by making a small mistake in the repetitive schema, or by taking a seemingly innocuous shortcut: A programmer can change the lazy semantics of `observe` with the definition of their `obs` instance. For example using `show` on the arguments of the constructor can result in a non-terminating program:

```
instance Observable TreeRat where
 obs (Node x1 x2 x3)
  = send (show x1  ++ ", left: " ++ show x2
                   ++ ", right: " ++ show x3)
         (return (Node x1 x2 x3))
 obs Leaf
  = send "Leaf" (return Leaf)
```

I extended HOOD so that the programmer can derive how a value is observed from its type using data generic programming techniques. This chapter mostly describes my engineering effort extending HOOD rather than that the chapter is a scientific contribution. The ability to derive how to observe a value from the value's type makes HOOD easier to use and less prone to misuse.

Data generic programming techniques are a well researched area resulting in a multitude of libraries and language extensions. A fairly complete overview and comparison is given by Hinze, Jeuring and Löh (2007). Because of its expressivity and availability I chose to use the Generic Deriving Mechanism (GDM) (Magalhães et al. 2010). GDM is an extension of Haskell for generic programming. The Glasgow Haskell compiler[1] (GHC) and the Utrecht Haskell compiler[2] (UHC) implement GDM.

4.1 The user's perspective

A user who wants to observe some intermediate data structure does not need to know how GDM works. They only need to know that all types of such intermediate data structures have to be instances of the classes `Observable` and `Generic`. The instances of both classes can be derived.

So they either derive the instances in the type declaration

```
data TreeRat = Node Rational TreeRat TreeRat | Leaf Rational
  deriving (Generic, Observable)
```

or, if the type is defined in a separate library that shall not be modified, then they derive them with standalone declarations:

```
instance Generic TreeRat
instance Observable TreeRat
```

Advanced users still can choose to define their own `Observable` instances: There is a trade-off between the risk to make a mistake and change the semantics, and being able to observe values of a certain type in a special way.

4.2 The implementer's perspective: product-sum types

Type generic programming defines behaviour based on the structure of a value's type. GDM is based on the fact that every type is isomorphic to a product-sum type. Figure 17 gives a simplified definition of product-sum types. There is one base type `Integer`.

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

[2] `http://www.cs.uu.nl/wiki/UHC`

$$\begin{array}{rll} T ::= & \mathrm{Integer} & \text{base type} \\ \mid & T_1 \mathrel{:+:} T_2 & \text{sum type, to encode choice} \\ \mid & T_1 \mathrel{:*:} T_2 & \text{product type, to encode structured data} \\ \mid & \mathrm{M}\; c & \text{meta type} \end{array}$$

Figure 17: Syntax of the product-sum types.

The name of a type constructors is encoded with the meta type M. Choice is encoded with the sum type. For example the Haskell type for Boolean values

```
data Bool = True | False
```

is represented in the simplified product-sum type syntax as

$$(\mathrm{M}\;\texttt{True}) \mathrel{:+:} (\mathrm{M}\;\texttt{False})$$

The product type is used to represent structured data. For example the Haskell type for Rational values

```
data Rational = Integer :% Integer
```

is encoded as

$$(\mathrm{M}\;\texttt{:\%}) \mathrel{:*:} \texttt{Integer} \mathrel{:*:} \texttt{Integer}$$

and the Haskell type `TreeRat` from the introductory example

```
data TreeRat = Node Rational TreeRat TreeRat | Leaf
```

is encoded as

$$((\mathrm{M}\;\texttt{Node}) \mathrel{:*:} \texttt{Rational} \mathrel{:*:} \texttt{TreeRat} \mathrel{:*:} \texttt{TreeRat}) \mathrel{:+:} (\mathrm{M}\;\texttt{Leaf})$$

4.2.1 Type generic observation

A type generic function is implemented with GDM by converting the observed value of a type `a` to a product-sum representation `Rep a` with the method `to`, manipulating this representation and converting back from the changed representation with the method `from`. To convert a value into a product-sum representation, its type should be of the `Generic` class. Slightly simplified, this class looks as follows:

```
class Generic a where
  type Rep a
  from  :: a -> Rep a
  to    :: Rep a -> a
```

For the Generic Deriving framework I provide a default implementation of obs:

```
class Observable a
 where
 obs         :: a -> Parent -> a
 default obs :: (Generic a, GObservable (Rep a)) =>
                        a -> Parent -> a
 obs x c     = to (gdmobs (from x) c)
```

With GDM I define how obs can be derived from a type representation. This representation is defined for instances of the Generic class.

For each value that I want to observe with my generic obs I use GDM's from-function to construct a product-sum representation. The returned type representation (with observed constructor arguments) is decoded to the original type with GDM's to-function.

Now I need to define the method gdmobs that operates on GDM's product-sum representation. I decompose the behaviour of gdmobs into three parts: render a shallow representation of the value, as a side-effect record this representation, and observe components of the value.

```
gdmobs x = send (shallowShow x) (observeChildren x)
```

It the next sections I discuss type generic definitions of shallowShow, which produces the message to record, and observeChildren, which wraps the value. I can use the polymorphic function send as it is.

I introduce a class GObservable. For each of GDM's representation-types I define an instance of GObservable, as I show in the subsequent subsections.

```
class GObservable f
 where
 gdmobs :: f a -> Parent -> f a
 gdmObserveArgs :: f a -> ObserverM (f a)
 gdmShallowShow :: f a -> String
```

4.2.2 Sum type

Choice between data constructors of the same type is encoded in GDM[3] with the following sum type representation:

[3] I simplified the actual representation of GDM, the full representation is presented by Magalhães et al. in (Magalhães et al. 2010). Instance of GObservable with the full representation can be obtained via https://hackage.haskell.org/package/Hoed.

```
data (a :+: b) = L1 a | R1 b
```

When there are more than two constructors, the sum type can be nested.

Our instance of `GObservable` is straightforward:

```
instance (GObservable a, GObservable b) => GObservable (a :+: b)
 where
 gdmobs (L1 x) = send (gdmShallowShow x) (gdmObserveArgs $ L1 x)
 gdmobs (R1 x) = send (gdmShallowShow x) (gdmObserveArgs $ R1 x)
 gdmShallowShow (L1 x) = gdmShallowShow x
 gdmShallowShow (R1 x) = gdmShallowShow x
 gdmObserveArgs (L1 x) = do x' <- gdmObserveArgs x; return (L1 x')
 gdmObserveArgs (R1 x) = do x' <- gdmObserveArgs x; return (R1 x')
```

4.2.3 Product type

Structured data is encoded in GDM using the following product representation:

```
data (f :*: g) = f :*: g

instance (GObservable a, GObservable b) => GObservable (a :*: b)
 where
 gdmobs (a :*: b) cxt =
   (gdmobs a cxt) :*: (gdmobs b cxt)
 gdmObserveArgs (a :*: b) = do
   a'  <- gdmObserveArgs a
   b'  <- gdmObserveArgs b
   return (a' :*: b')
```

Records are a special case of the tuple type where a set of field labels index the tuple (Harper 2013). Currently HOOD does not trace field labels, but with my approach it would be trivial to extend it to do so.

Let us consider how a value of the `TreeRat` type would be encoded. A value with constructor `Node` has three arguments, this is encoded with the product-representation. Our `TreeRat` type can either be `Node` or `Leaf`. The choice between these data constructors is encoded with `L1` for `Node`-values and `R1` for `Leaf`-values. For example assume I want to encode a simple tree with two leafs and one node. The values `x`, `y` and `z` are stored in the tree. I do not elaborate on how these are encoded but just label their representations as `q`, `r` and `s`:

```
from (Node x (Node y Leaf Leaf) Leaf) ⇝
L1 (M1 (q :*:  R1 (M1 r) :*:  R1 (M1 s)))
```

4.2.4 Constructor names

Constructor names can be attached as labels to a type. In GDM this meta-information is encoded with the combination of type `M1` and method `conName`. The type is used in the representation while the method holds the actual constructor label:

```
data M1 c a = M1 a
class Constructor c where conName :: c -> String
```

Note that the `M1` data constructor is used for many different types. The types are distinguished by the `c` type variable. Types for this variable and corresponding `conName` instances need to be generated. In GHC this is done when I derive `Generics` for a type. I would for example for my `TreeRat` generate the types `NodeConstr` and `LeafConstr` such that for some value `m` of the right type evaluation yields the following results:

```
conName (m ::  M1 NodeConstr a) ⇝ "Node"
conName (m ::  M1 LeafConstr a) ⇝ "Leaf"
```

With these I define the following instance for GDM's representation of meta types:

```
instance (GObservable a, Constructor c) => GObservable (M1 c a)
 where
 gdmobs m1 = send (gdmShallowShow m1) (gdmObserveArgs m1)
 gdmObserveArgs (M1 x) = do x' <- gdmObserveArgs x; return (M1 x')
 gdmShallowShow = conName
```

I query the meta-information to find the constructor names and record these.

4.2.5 Base types

Base types such as `Integer` "have no internal structure as far as the type system is concerned" (Pierce 2002). Thus there are no further arguments to be observed and I can use `show` to produce the representation without forcing further evaluation. Base types are not defined by the user and therefore I can supply an instance such as

```
instance Observable Integer where obs = observeBase
observeBase x = send (show x) (return x)
```

for every base type in the value observation tracing library.

4.2.6 Special types, including function types

The values of some types are observed differently from data constructors. Our natural semantics already shows that function values are observed rather differently. Also for an abstract type such as `IO a` I cannot observe a value, but I can record that there is a value. For all these types I just continue using the definitions of `Observable` included in the original HOOD (Gill 2000).

4.3 Mixing observed and not-observed values

In the first part of this chapter I ignored type constructors. However, type constructors such as the list type constructor and `Maybe` are an essential part of writing polymorphic library functions.

Consider the tree library of Listing 16 that contains the type constructor `Tree`. A data type is obtained by *applying* the type constructor to another data type. For example `Tree Integer` describes a tree with all its values of type `Integer`.

Some function definitions are independent of the type of the values in the `Tree` and I can define such a function polymorphically. Instead of an actual type I use a type variable `a`. Our example library contains a polymorphic function `depth` with type `Tree a -> Int -> [a]` to return all nodes at a certain depth in a complete tree.

Assume I use my library to find the list of possible outcomes of flipping a coin n times. When I flip a coin once it can be heads or tails, I represent that with the following data type:

```
data Coin = Heads | Tails
```

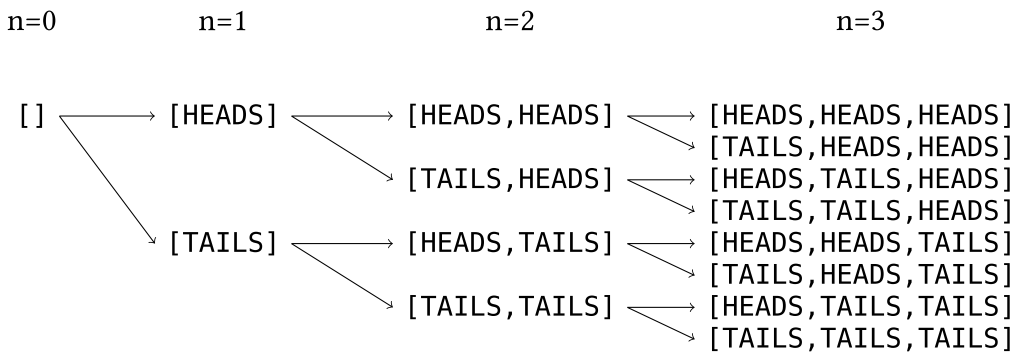
To represent the state after flipping a coin n times I use the type `[Coin]`. When the first time I flip a coin gives us heads and the second time gives us tails I use

```
[Tails,Heads]
```

thus, the list is a sequence of coin flips from most recent outcome to first outcome. The next state after flipping the coin a third time is created by added the new outcome to the front of the list. Every state s_i has two possible next states: `Heads :` s_i and `Tails :` s_i. The states of up to three coin-flips can be represented in a tree as follows:

Listing 16 A library with a `Tree` type constructor, a defective definition of polymorphic breadth-first ordering of the nodes in a tree and a polymorphic definition to get the nodes at a given depth from a complete tree.

```
data Tree a = Node a (Tree a) (Tree a) | Leaf a

depth :: Tree a -> Int -> [a]
depth tree n = take ((n+1)*2) (drop (2^n-1) (breadthFirst tree))

breadthFirst :: Tree a -> [a]
breadthFirst tree = fold [tree]
  where
  fold [] = []
  fold queue = map nodeVal queue
   ++ concatMap (fold . subTrees) queue

nodeVal :: Tree a -> a
nodeVal (Node x t1 t2) = x
nodeVal (Leaf x) = x

subTrees :: Tree a -> [Tree a]
subTrees (Node x t1 t2) = [t1,t2]
subTrees (Leaf x) = []
```



The complete tree with all possible states can be defined with

```
mkTree c = Node c (mkTree (Head : c)) (mkTree (Tail : c))
```

One property of this tree is that given all states at a certain depth, `Heads` occurs as often as `Tails`, expressed in the following QuickCheck property:

```
prop_depthSound n = length heads == length tails
 where
 (heads,tails) = partition (==Heads) outcomes
 outcomes = concat (depth (mkTree []) n)
```

However `prop_depthSound 3` unexpectedly evaluates to `False`! To find out why this property fails I want to inspect the argument and result value of the applications of `breadthFirst`. This will reveal the structure of the tree visited (helping us to decide if the definition of `breadthFirst` is sound and it will reveal which values stored in the nodes are evaluated in the context of this function application helping us to decide if the definition of `depth` is sound.

To annotate the program the programmer transforms the program as follows:

1. Derive `Observable` for my `Tree` type in my library.

   ```
   data Tree a = Node a (Tree a) (Tree a) | Leaf a
     deriving (Generic,Observable)
   ```

2. Label the `breadthFirst` function definition for tracing.

   ```
   breadthFirst = observe "breadthFirst" (\tree -> fold [tree])
   ```

3. Now the programmer must also change the type declaration of `breadthFirst` because the derived instance of the `obs` method expects `a` in `Tree a` to be `Observable`. Thus the type declaration becomes

```
breadthFirst
  (Node _
    (Node _
      (Node _
        (Node _
          (Node (Head : Head : Head : Head : [])
            (Node (Head : Head : Head : Head : Head : [])
              (Node (Head : Head : Head : Head : Head :
                Head : [])
                (Node (Head : Head : Head : Head : Head :
                  Head : Head : []) _ _)
                (Node (Tail : Head : Head : Head : Head :
                  Head : Head : []) _ _))
               (Node (Tail : Head : Head : Head : Head : Head :
                 []) _ _))
            (Node (Tail : Head : Head : Head : Head : []) _ _))
          (Node (Tail : Head : Head : Head : []) _ _))
        _)
      _)
    _)
  = _ : _ : _ : _ : _ : _ : _ : (Head : Head : Head : Head :
    []) : (Tail : Head : Head : Head : []) : (Head : Head :
    Head : Head : Head : []) : (Tail : Head : Head : Head :
    Head : []) : (Head : Head : Head : Head : Head : Head :
    []) : (Tail : Head : Head : Head : Head : Head : []) :
    (Head : Head : Head : Head : Head : Head : Head : []) :
    (Tail : Head : Head : Head : Head : Head : Head : []) : _
```

Figure 18: Observing a defective breadth first implementation.

```
breadthFirst
  (Node _
    (Node _
      (Node _
        (Node _
          (Node <?>
            (Node <?>
              (Node <?> _ _)
              (Node <?> _ _))
            (Node <?> _ _))
          (Node <?> _ _))
        _)
      _)
    _)
 = _ : _ : _ : _ : _ : _ : _
   : <?> : <?> : <?> : <?>
   : <?> : <?> : _
```

```
breadthFirst
  (Node _
    (Node _
      (Node _
        (Node <?> _ _)
        (Node <?> _ _))
      (Node _
        (Node <?> _ _)
        (Node <?> _ _)))
    (Node _
      (Node _
        (Node <?> _ _)
        (Node <?> _ _))
      (Node _
        (Node <?> _ _)
        (Node <?> _ _))))
  = _ : _ : _ : _ : _ : _ : _
    : <?> : <?> : <?> : <?>
    : <?> : <?> : <?> : <?> : _
```

Figure 19: Two observations of a tree without observing the elements in the tree. On the left-hand side the result of observing a defective implementation; on the right-hand side the result of observing a sound implementation.

```
breadthFirst :: Observable a => Tree a -> [a]
```

and hence the programmer also needs to change the type declaration of the `depth` function

```
depth :: Observable a => Tree a -> Int -> [a]
```

4. But now the changed type declaration is exposed to modules outside my tree library! Thus the programmer also has to make changes in other modules. They need to derive `Observable` for any concrete type `a` of values with type `Tree a` to which `depth` is ever applied. In the example program that is

```
data Coin = Head | Tail
  deriving (Generic, Observable)
```

Step three and four are unfortunate, although it is a task that can easily be automated, I rather not force the programmer to make changes outside a library module.

Furthermore, the traced coin-flip states do not provide much information to the programmer to understand their code, in fact they clutter the output printed after evaluating the program (Figure 18).

The symbol _ is already widely used for expressions unevaluated in this context. Now I introduce another symbol <?> for evaluated but not observed values. Compare the value representation on the left in Figure 19 with the value representation of Figure 18. From the latter it is easier to infer that the defect must be in the definition of `breadthFirst`: in the last line of the definition instead of exploring the subtrees of the queued nodes and concatenating the result

```
concatMap (fold . subTrees) queue -- defective
```

the function should concatenate the subtrees of the queued nodes and then explore such as in the definition

```
fold (concatMap subTrees queue) -- correct
```

To record an event with the symbol <?> in our trace for a value of a type for which the user did not provide a specific instance, I provide an overlappable instance[4] in my tracing library:

```
instance {-# OVERLAPPABLE #-} Observable a where
  obs x = obsCon "<?>" (return x)
```

4.4 Summary

In this chapter I show how to overcome the restriction of hand-written `Observable` instances for datatypes of values that I want to observe. Furthermore I present a method for observing up to a certain data type or type variable, which makes HOOD easier to use in libraries and testing frameworks. I implemented my idea with generic programming techniques.

I introduced an unobserved value <?> that can help to make a cluttered computation statement simpler and easier to understand for the human programmer. The mechanism also allows the programmer to annotate polymorphic functions in a library, without modifying consumers of the library.

[4] https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/glasgow_exts.html#overlapping-instances

5 Computation tree construction with trace stacks

With the method of Chapter 3 I obtain the nodes of a computation tree, now we need to derive the edges. In this chapter and the next chapter I discuss two alternative methods for deriving the edges of a computation tree. The programmer has to answer less questions and gets a more precise defect location when the algorithmic debugger uses a computation tree derived with the method of Chapter 6 then when the algorithmic debugger uses the method from this chapter (see Section 6.4 for a more detailed comparison). However, this chapter is not completely obsoleted: this chapter's method works with any runtime environment that supports the cost centre stack extension and can therefore, for example, without further work be applied to parallel and concurrent computations.

In contrast to debugging, time and space profiling of lazy functional programs is not only well-studied (Sansom and Peyton Jones 1997; Morgan and Jarvis 1998; Marlow 2012), but the Glasgow Haskell Compiler[1] (GHC) also provides reliable profiling for real-world Haskell programs. For attributing time and space costs to parts of a program these parts are labelled with identifiers. The profiling environment maintains a trace stack of these identifiers. Section 5.1.2 explains trace stacks in detail. Today most Haskell libraries are automatically installed with a variant compiled with the profiling flag.

Key observation in this chapter 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.

My work is of broader interest than just Haskell and GHC. Adding a trace stack to

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

an evaluator of a functional language is a small extension that is useful for numerous purposes. Already when Marlow (2012) revisited trace stacks, he was aiming to use them for time and space profiling, coverage analysis and traditional debugging. Now I 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, I 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 my method would be useful for such languages as well.

Based on this observation I present a novel approach to algorithmic debugging of Haskell programs. My method needs program annotations only in suspected code, where the programmer is looking for defects, not in any trusted modules, which the programmer assumes to be correct. The programmer just compiles all modules of their faulty program for profiling and the executable uses the standard runtime system for profiling.

5.1 Background, observation and idea

I use the example program in Listing 17 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 ❶. 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 ❷: the `>` operator should be replaced with `<` and `xs` should be swapped with `ys`.

Listing 17 A defective example program for sorting integers.

```
main :: IO ()
main = print (isort [4,3,5])    ⟵ ❶ Sort and print result

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

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

5.1.1 Locating the defect with my algorithmic debugger

In Section 2.2 I discussed how an algorithmic debugger locates the defect in a program by asking an oracle (e.g. the programmer who tries to locate a defect in their code) questions about intermediate computations. For the example program the interaction could look as follows, with the answers of the oracle 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"!
```

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 20. 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. I 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.

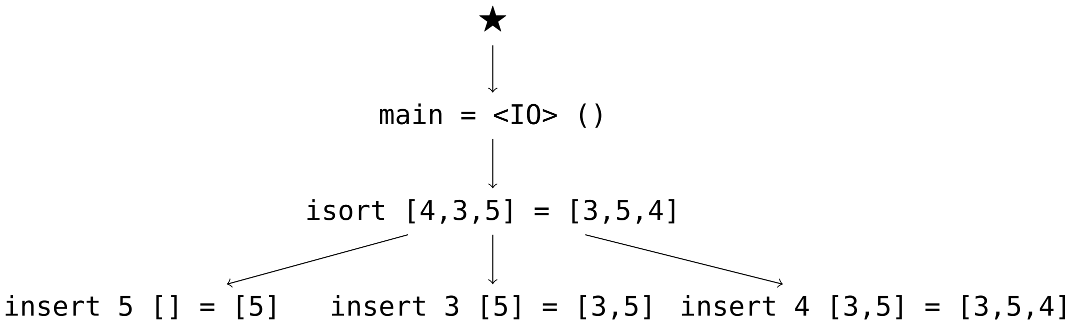


Figure 20: A computation tree for the sorting program.[2]

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

5.1.2 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 (Sansom and Peyton Jones 1997). For example, I 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 the programmer labels with cost centres they choose the granularity of assigning profiling costs. In our example program I can for example choose not to annotate `foldr` and thus subsume its computation costs into the cost centre `"isort"`.

Morgan and Jarvis (1998) 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, I 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 ⟨`main`, `isort`⟩, whereas the costs of the function `insert` as called by `isort` is attributed to the stack ⟨`main`, `isort`, `insert`⟩. 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. Consider for example evaluating the expression `prop_absPos 3` which detects a defect in the following program:

```
isPos x = x > 0
abs y = -y
prop_absPos z = isPos (abs z)
```

The function appliation `abs z` is evaluated when `y` in the body of `isPos` is demanded, hence the run-time stack during the evaluation of the body of the `abs` function is ⟨`prop_absPos`, `isPos`, `abs`⟩ and the cost centre stack is ⟨`prop_absPos`, `abs`⟩.

5.1.3 Idea: dependencies from observed stacks

I 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.

5.1.4 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 my program used `isort` twice, I 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.

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

Chitil and Davie (2008) 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 (Nilsson and Sparud 1997; Nilsson 1998; Wallace et al. 2001; Chitil 2005) choose the second structure, but here I choose the first: observing values can represent functional values only as finite maps, not obtain an intensional representation.

5.2 Tracing semantics

I define a semantics to unambiguously describe what information I record in a trace while evaluating an expression. I start with adding adding cost centre stacks following Marlow (2012) to the semantics of the core language from Section 2.1.2. Subsequently I extend the semantics with value observation tracing such that a snapshot of the cost centre stack is included in each observed value.

5.2.1 Adding a stack

I reformulate Marlow's framework of stacks as implemented in GHC for my computation statements. Marlow's semantics is quite different from the semantics of Morgan and Jarvis (1998); this chapter and the GHC implementation follow the former. First, in Figure 21 I extend the syntax by an operation for labelling any subexpression within a program. The nature of labels is irrelevant; in practice I use strings.

Figure 22 shows the semantic rules. I 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 $\Gamma_1, \mathcal{S}_1 : e \Downarrow \Gamma_2, \mathcal{S}_2 : v$ means that the expression e in the context of the heap Γ_1 and the stack $\mathcal{S}_1$ reduces to the value v in the context of the modified heap Γ_2 and modified stack $\mathcal{S}_2$. The Push rule pushes the label onto the stack using the function $\triangleleft$. The heap now stores a stack with every expression. When the Let rule stores an expression in the heap, it includes the current stack. The Var rule from the core semantics of Figure 3 is replaced by the VarC rule for constructor-values and the VarL rule for λ-abstractions. 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

expression e	
$::= v$	value
$\mid e\ x$	application
$\mid$ `let` $\{x_k = e_k\}_{k=1}^{n}$ `in` e	recursive binding
$\mid$ `case` e `of` $\{c_k\ x_1 \ldots x_{m_k} \to e_k\}_{k=1}^{n}$	case
$\mid x$	variable
$\mid x_1 \oplus x_2$	application of a primitive
$\mid$ **`push`** $\boldsymbol{f}\ \boldsymbol{e}$	**push label f onto stack**

Figure 21: Syntax of core language extended to label expressions (difference to Figure 2 in **bold**).

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

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

$$\frac{\Gamma_1, \boldsymbol{\mathcal{S}_{\mathbf{h}}} : e \Downarrow \Gamma_2, \boldsymbol{\mathcal{S}_2} : c\, x_1 \ldots x_n}{\Gamma_1[x \mapsto \boldsymbol{(\mathcal{S}_{\mathbf{h}}, e)}], \boldsymbol{\mathcal{S}_1} : x \Downarrow \Gamma_2[x \mapsto \boldsymbol{(\mathcal{S}_2, c\, x_1 \ldots x_n)}], \boldsymbol{\mathcal{S}_2} : c\, x_1 \ldots x_n}\ \textbf{VarC}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{S}_{\mathbf{h}}} : e \Downarrow \Gamma_2, \boldsymbol{\mathcal{S}_{\lambda}} : \lambda y.e'}{\Gamma_1[x \mapsto \boldsymbol{(\mathcal{S}_{\mathbf{h}}, e)}], \boldsymbol{\mathcal{S}}_{\text{app}} : x \Downarrow \Gamma_2[x \mapsto \boldsymbol{(\mathcal{S}_{\lambda}, \lambda y.e')}], \boldsymbol{\mathcal{S}}_{\text{app}} \bowtie \boldsymbol{\mathcal{S}_{\lambda}} : \widehat{\lambda y.e'}}\ \textbf{VarL}$$

$$\frac{\Gamma_1[x_i \mapsto \boldsymbol{(\mathcal{S}_1, e_i)}], \boldsymbol{\mathcal{S}_1} : e \Downarrow \Gamma_2, \boldsymbol{\mathcal{S}_2} : v}{\Gamma_1, \boldsymbol{\mathcal{S}_1} : \texttt{let}\{x_i = e_i\}\ e \Downarrow \Gamma_2, \boldsymbol{\mathcal{S}_2} : v}\ \text{Let}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{S}_2} : v \qquad \texttt{notAbs}\ v}{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e\ x \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{S}_2} : \texttt{Exception}}\ \text{EApp}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e_1 \Downarrow \Gamma_2, \boldsymbol{\mathcal{S}_2} : \lambda y.e_2 \qquad \Gamma_2, \boldsymbol{\mathcal{S}_2} : e_2[x/y] \Downarrow \Gamma_3, \boldsymbol{\mathcal{S}_3} : v}{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e_1\ x \Downarrow \Gamma_3, \boldsymbol{\mathcal{S}_3} : v}\ \text{App}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{S}_2} : v \qquad \texttt{notCon}\ v\ \{c_i\}_{i=1}^{n}}{\Gamma_1, \boldsymbol{\mathcal{S}_1} : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{S}_2} : \texttt{Exception}}\ \text{ECase}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{S}_2} : c_k\ x_1 \ldots x_{m_k} \qquad \Gamma_2, \boldsymbol{\mathcal{S}_2} : e_k[x_i/y_i]_{i=1}^{m_k} \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{S}_3} : v}{\Gamma_1, \boldsymbol{\mathcal{S}_1} : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{S}_3} : v}\ \text{Case}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e_1 \;\Downarrow\; \Gamma_2, \boldsymbol{\mathcal{S}_2} : v_1 \qquad \Gamma_2, \boldsymbol{\mathcal{S}_2} : e_2 \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{S}_3} : v_2}{\Gamma_1, \boldsymbol{\mathcal{S}_1} : e_1 \oplus e_2 \;\Downarrow\; \Gamma_3, \boldsymbol{\mathcal{S}_3} : v_1 \underline{\oplus}\, v_2}\ \text{Prim}$$

$$\frac{\boldsymbol{\Gamma_1, \mathcal{S}_1 \lhd f : e \Downarrow \Gamma_2, \mathcal{S}_2 : v}}{\boldsymbol{\Gamma_1, \mathcal{S}_1 :}\ \texttt{\textbf{push}}\ \boldsymbol{f\ e \Downarrow \Gamma_2, \mathcal{S}_2 : v}}\ \textbf{Push}$$

Figure 22: New and updated rules adding a stack to the semantics of the core language. Differences with Figure 3 in **bold**, new rules VarC and VarL with respect to Var rule which the two new rules replace.

$\langle f_0, \ldots, f_n \rangle \lhd f = \langle f_0, \ldots, f_j, f \rangle$
  where $f \notin \{f_0, \ldots f_j\}$ and ($j = n$ or $f_{j+1} = f$)

$\langle f_0, \ldots, f_j, f_a, \ldots, f_b \rangle \bowtie \langle f_0, \ldots, f_j, f_c, \ldots, f_d \rangle$
  $= \langle f_0, \ldots, f_j \rangle \lhd f_c \lhd \ldots \lhd f_d \lhd f_a \lhd \ldots \lhd f_b$
  where $f_a \neq f_c$

Figure 23: Definitions of $\lhd$ and $\bowtie$ as used in GHC.

event	t	$::=$	**i:Root f $\mathcal{S}$**	**root with label f and cost centre stack $\mathcal{S}$**
		$\mid$	i:Con p c a	value is application of constructor c
		$\mid$	i:Lam p	value is an abstraction
		$\mid$	i:MapsTo p	function application

Figure 24: Extended syntax of events in the observation trace from Figure 11 to record cost centre stacks (updated event in **bold**).

this λ-abstraction. In a higher-order language these stacks can differ substantially. The function $\bowtie$ merges the two stacks for the result.

Figure 23 gives Marlow's definitions of the two functions $\lhd$ and $\bowtie$. My dependency generation technique is independent of the precise definition of these stack operations. I 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.

5.2.2 Adding a value observation trace

Finally I reformulate value observation to record the information needed to construct a computation graph. Starting point is the trace described in Section 3.2.2 which already contains the information we need to construct computation statements. Here, I additionally record a snapshot of the cost centre stack when observation of a labelled expression starts in a root event. Figure 24 gives the updated syntax of events. I store label and stack separately in the root event because pushing is not lossless.

Consider the example trace in Figure 25. It consists of five events. They form the single tree shown in Figure 26, which represents the observed functional value {\9->9} with label "*id*" and stack $\langle\rangle$.

To observe the values of an expression I use two pseudo-functions `obs` and `obs`$_\lambda$.

0:Root "*id*" $\langle\rangle$	label with empty stack
1:Lam (P 0)	abstraction
2:MapsTo (P 1)	application of the abstraction
3:$\mathrm{T_c}$ ($\mathrm{P_a}$ 2) 9	argument of this application
4:$\mathrm{T_c}$ ($\mathrm{P_r}$ 2) 9	result of this application

Figure 25: Trace produced by evaluating the expression $\texttt{let}\{\texttt{id} = \texttt{push}$ "*id*" $(\lambda x.x), \texttt{y} = 9\}$ $(\texttt{id}\ \texttt{y})$.

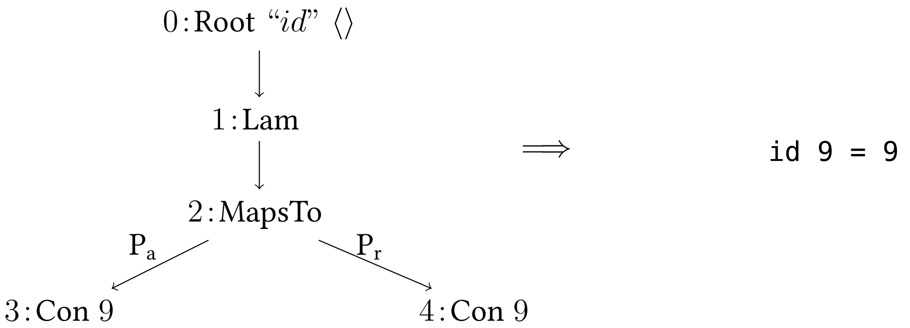


Figure 26: Tree of events from the trace in Figure 25. An application event has two events as children, the argument with parent $\mathrm{P_a}$ and the result with parent $\mathrm{P_r}$.

Figure 27 adds them to the syntax. They are never used in a program but appear only during program execution.

Figure 28 shows the semantics that records the trace I construct computation trees with. The semantics combines the cost centre stack semantics (Figure 22) and the value observation tracing semantics (Figure 12). The key change of the new semantics, that will later allow us to construct a computation tree, is in the Push rule which combines the behaviour of the Push rule of the cost centre stack semantics and the Observe rule of the value observation tracing semantics such that the rule records a root event which includes the current stack. Furthermore it wraps the observed expression with the pseudo-function `obs`.

For the ObsCon, ObsLam and ObsApp rule are almost identical to the rules with the same name from the value observation tracing semantics (Figure 12) but with a stack $\mathcal{S}$ on both left- and right-hand side of every reduction. The remaining rules (Lam, Con, VarC, VarL, Let, EApp, App, ECase, Case and Prim) are almost identical to the rules with the same name from the cost centre stack semantics (Figure 22) but with a trace $\mathcal{T}$ on both left- and right-hand side of every reduction.

expression e	
$::= v$	value
$\mid e\ x$	application
$\mid \texttt{let}\ \{x_k = e_k\}_{k=1}^{n}\ \texttt{in}\ e$	recursive binding
$\mid \texttt{case}\ e\ \texttt{of}\ \{c_k\ x_1 \ldots x_{m_k} \to e_k\}_{k=1}^{n}$	case
$\mid x$	variable
$\mid \texttt{push}\ f\ e$	push label **and observe expression**
$\mid \boldsymbol{\texttt{obs}\ p\ e}$	**observed expression**
value v	
$::= \boldsymbol{v_\lambda}$	**functional value**
$\mid c\ x_1 \ldots x_n$	saturated application of data constructor
functional value $\boldsymbol{v_\lambda}$	
$::= \lambda x.e$	abstraction
$\mid \texttt{obs}_\lambda\ \mathrm{p}\ \mathrm{v}_\lambda$	**observed functional value**

Figure 27: Syntax of the language extended to label and observe expressions. Differences with Figure 21 in **bold**.

5.3 Processing the trace

I defined how a trace results from evaluating a program. In Section 3.3 I explained how to construct computation statements from a trace. Now I am ready to use the snapshots of the stack to derive dependencies between the statements and form a computation graph.

5.3.1 Constructing a computation graph

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

5.3.1.1 Stack push function $\lhd$

In my semantics I 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 $c_1 \to c_2$ when $\mathcal{S}_1 \lhd f_1 = \mathcal{S}_2$ for statement c_1 with label f_1 and stack $\mathcal{S}_1$, and c_2 with f_2 and $\mathcal{S}_2$. Consider for example the labelled expressions in the following

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

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

$$\frac{\Gamma_1,\mathcal{S}_\text{h},\mathcal{T}_1 : e \Downarrow \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : c\ x_1 \ldots x_n}{\Gamma_1[x \mapsto (\mathcal{S}_\text{h}, e)],\mathcal{S}_1,\mathcal{T}_1 : x \Downarrow \Gamma_2[x \mapsto (\mathcal{S}_2, c\ x_1 \ldots x_n)],\mathcal{S}_2,\mathcal{T}_2 : c\ x_1 \ldots x_n}\ \text{VarC}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e \Downarrow \Gamma_2,\mathcal{S}_\lambda,\mathcal{T}_2 : \lambda y.e'}{\Gamma_1[x \mapsto (\mathcal{S}_1, e)],\mathcal{S}_\text{app},\mathcal{T}_1 : x \Downarrow \Gamma_2[x \mapsto (\mathcal{S}_\lambda, \lambda y.e')],\mathcal{S}_\text{app} \bowtie \mathcal{S}_\lambda,\mathcal{T}_2 : \widehat{\lambda y.e'}}\ \text{VarL}$$

$$\frac{\Gamma_1[x_i \mapsto e_i]_{i=1}^{n},\mathcal{S}_1,\mathcal{T}_1 : e \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : v}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : \mathtt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \mathtt{in}\ e \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : v}\ \text{Let}$$

$$\frac{\Gamma,\mathcal{S}_1,\mathcal{T}_1 : e \ \Downarrow\ \Gamma_2,\mathcal{T}_2 : v \qquad \mathtt{notAbs}\ v}{\Gamma,\mathcal{T}_1 : e\ x \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : \mathtt{Exception}}\ \text{EApp}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : \lambda x.e' \qquad \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : e'[y/x] \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e\ y \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v}\ \text{App}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : v \qquad \mathtt{notCon}\ v\ \{c_i\}_{i=1}^{n}}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : \mathtt{case}\ e\ \mathtt{of}\ \{c_i\ y_1 \ldots y_{m_i} \rightarrow e_i\}_{i=1}^{n} \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : \mathtt{Exception}}\ \text{ECase}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : c_k\ x_1 \ldots x_{m_k} \qquad \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : e_k[x_i/y_i]_{i=1}^{m_k} \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : \mathtt{case}\ e\ \mathtt{of}\ \{c_i\ y_1 \ldots y_{m_i} \rightarrow e_i\}_{i=1}^{n} \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v}\ \text{Case}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e_1 \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : v_1 \qquad \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : e_2 \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v_2}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e_1 \oplus e_2 \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v_1 \underline{\oplus} v_2}\ \text{Prim}$$

$$\frac{\Gamma_1,\mathcal{S}_1 \triangleleft f,\mathcal{T} \lessdot (i\!:\!\text{Root}\ f\ \mathcal{S}_1) : \mathtt{obs}\ (\text{P}\ i)\ e \Downarrow \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : v \qquad i = |\mathcal{T}|}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : \mathtt{push}\ f\ e \Downarrow \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : v}\ \text{Push}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : c\ x_1 \ldots x_n \qquad i = |\mathcal{T}_2|}{\begin{array}{c}\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : \mathtt{obs}\ p\ e \ \Downarrow\ \Gamma_2[y_1 \mapsto \mathtt{obs}\ (\text{P}_\text{c}\ i\ 1)\ x_1, \ldots, y_n \mapsto \mathtt{obs}\ (\text{P}_\text{c}\ i\ n)\ x_n],\\ \mathcal{S}_2,\mathcal{T}_2 \lessdot (i\!:\!\text{Con}\ p\ c\ (\text{arity}\ c)) : c\ y_1 \ldots y_n\end{array}}\ \text{ObsCon}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e \Downarrow \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : v_\lambda \qquad i = |\mathcal{T}_2|}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : \mathtt{obs}\ p\ e \Downarrow \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 \lessdot (i\!:\!\text{Lam}\ p) : \mathtt{obs}_\lambda\ (\text{P}\ i)\ v_\lambda}\ \text{ObsLam}$$

$$\frac{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e \ \Downarrow\ \Gamma_2,\mathcal{S}_2,\mathcal{T}_2 : \mathtt{obs}_\lambda\ p\ v_\lambda \qquad \begin{array}{l}\Gamma_2[y \mapsto \mathtt{obs}\ (\text{P}_\text{a}\ i)\ x],\\ \mathcal{S}_2,\mathcal{T}_2 \lessdot (i\!:\!\text{MapsTo}\ p) :\\ \mathtt{obs}\ (\text{P}_\text{r}\ i)\ (v_\lambda\ y) \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v\end{array} \qquad i = |\mathcal{T}_2|}{\Gamma_1,\mathcal{S}_1,\mathcal{T}_1 : e\ x \ \Downarrow\ \Gamma_3,\mathcal{S}_3,\mathcal{T}_3 : v}\ \text{ObsApp}$$

Figure 28: Cost centre stack and value observation tracing semantics.

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 $\langle$“A”$\rangle$ with the latter statement. Because $\langle\rangle \triangleleft \text{“}A\text{”} = \langle\text{“}A\text{”}\rangle$, there is a dependency `A 3 = 3` → `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 $c_1 \to c_2$, $c_2 \to c_3$, $c_2 \to c_4$ and $c_3 \rightleftarrows c_4$.

5.3.1.2 Stack merge function $\bowtie$

The computation graph has dependencies $c_1 \to c_2 \to c_3$ when $\mathcal{S}_3 = \mathcal{S}_1 \triangleleft f_1 \bowtie \mathcal{S}_2 \triangleleft f_2$ for statement c_1 with label f_1 and stack $\mathcal{S}_1$, c_2 with f_2 and $\mathcal{S}_2$, and c_3 with f_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:

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

Evaluating this program gives us the three computation statements below. Note how the VarL rule from Figure 22 affects the stack of the f_in statement.

`main = 3`	“$main$”	$\langle\rangle$
`f 3 = 3`	“f”	$\langle\rangle$
`f_in 3 = 3`	“f_in”	$\langle$“f”, “$main$”$\rangle$

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

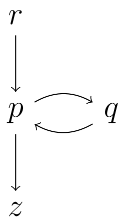


Figure 29: Reducible cycle.

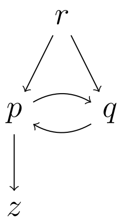


Figure 30: Irreducible cycle.

5.3.2 From computation graph to tree

So from the trace I obtain a computation graph that may have cycles. For algorithmic debugging I need to 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 faulty nodes in a computation tree is an established technique (Shapiro 1983).

A computation graph consists of nodes (the computation statements) and edges (the dependencies). If there is a dependency $c \to d$, 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 29 has dominator p whereas the cycle in Figure 30 has no dominator). If node d dominates node p, then $p \to d$ is a *back edge.*

5.3.3 Removing a reducible cycle

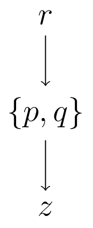
The actual computation dependencies form a tree and by definition a tree does not have back edges. Therefore I can safely remove the back edges from my computation graph. This will break any reducible cycle. Consider for example the reducible cycle of p and q in the graph of Figure 29. Without back edge $q \to p$ this is an acyclic graph.

5.3.4 Removing an irreducible cycle

After removing back-edges some irreducible cycles may remain. I *collapse* an irreducible cycle by replacing the statements in the cycle with a single node in which these statements are combined. For example, I replace the statements p and q in the irreducible cycle of Figure 30 with node $\{p, q\}$. If any of the combined statements is judged wrong, then I 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 30 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 the following graph:



5.3.5 Accuracy and order

I first remove back edges of all reducible sets, collapse any remaining (irreducible) cycles and use the resulting DAG to find faulty nodes. Instead of both removing back edges and then collapsing irreducible cycles I 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 I 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 31 shows an example.

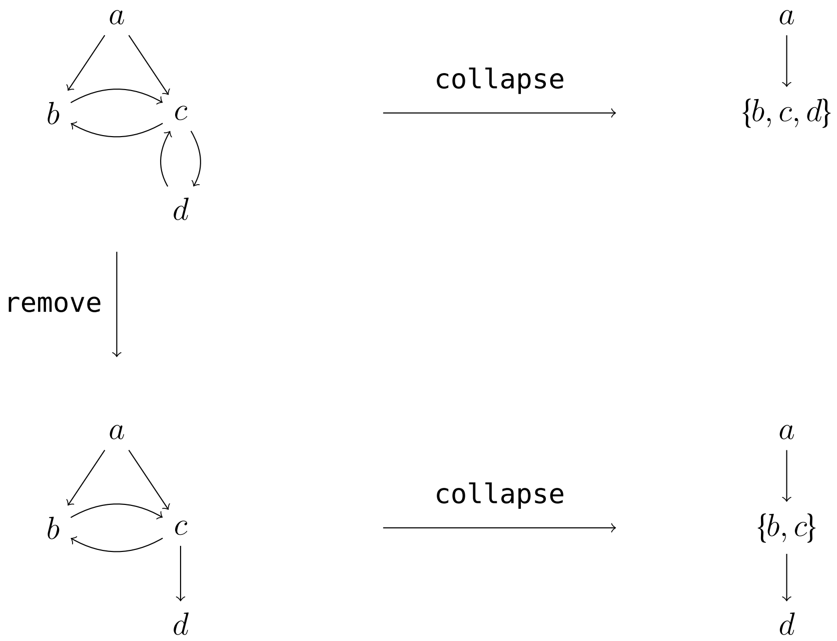


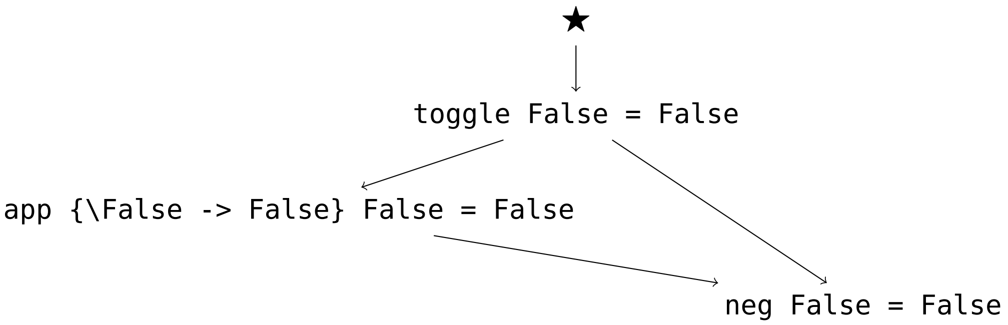
Figure 31: Order of `collapse` and `remove` for reducible cycle $\{c, d\}$ nested in irreducible cycle $\{b, c, d\}$.

5.4 Higher-order functions

In Section 2.2.1 I discussed different kinds of computation trees. In general my method gives an over-approximiation of the FDT and in some cases it produces a sound computation tree which mixes EDT with FDT. Consider the example program from Listing 3.

Case 1: `toggle`, `app` and `neg` are annotated

Evaluating the example program with `toggle`, `app` and `neg` annotated produces a trace from which I derive the following computation tree:



This is an over-approximation of the FDT and thus sound for algorithmic debugging. Where does the edge `toggle False = False` $\rightarrow$ `neg False = False` come from? Let us take a closer look at the recorded cost centre stacks. We have:
Statement $c_f =$ `toggle False = False` with stack $\mathcal{S}_f = \langle\rangle$
Statement $c_a =$ `app {\False -> False} False = False` with stack $\mathcal{S}_a = \langle\text{“}toggle\text{”}\rangle$
Statement $c_n =$ `neg False = False` with stack $\mathcal{S}_n = \langle\text{“}toggle\text{”}, \text{“}app\text{”}\rangle$
Two of the arcs are derived from potential stack push operations:

- $\mathcal{S}_f \triangleleft \text{“}toggle\text{”} = \mathcal{S}_a$ therefore $c_f \rightarrow c_a$
- $\mathcal{S}_a \triangleleft \text{“}app\text{”} = \mathcal{S}_n$ therefore $c_a \rightarrow c_n$

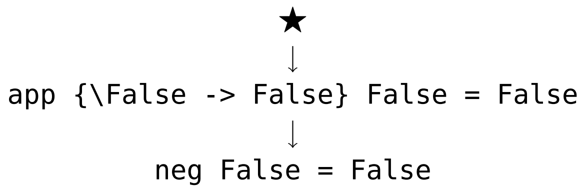
The arc $c_f \rightarrow c_n$ that makes it an over-approximation of an FDT is derived from a potential call operation:

- $(\mathcal{S}_a \triangleleft \text{“}app\text{”}) \bowtie (\mathcal{S}_f \triangleleft \text{“}toggle\text{”}) = \mathcal{S}_n$ therefore $c_a \rightarrow c_f$ and $c_f \rightarrow c_n$

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

Case 2: `app` and `neg` are annotated

When I do not annotate `toggle`, evaluation gives us a trace from which I 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 `neg` function definition. The `app`-statement is right. The `neg`-statement is wrong and because the statement has no children in the computation tree, the statement is identified as faulty. Vice versa, if I assume `app` to be faulty and `neg` 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.

In the next section I show how I verified that the computation trees constructed with my method are sound in general.

5.5 My algorithmic debugger Hoed-cc

GHC allows programmatic access to the cost centre stack of the profiling environment. Hence my 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 I present in this chapter uses `push` to both add a label onto the stack and to observe the value of the enclosed expression. I implement the former with the GHC pragma Set Cost Centre (SCC) ❶ and the latter with the combinator `observe` from my tracing library ❷. See for example the annotations of the `isort` function in Listing 18.

Annotating a function is a straightforward and mechanical process. A compiler pass could annotate all top-level functions in a module. The annotation functions can be derived for values of different types using the generic framework of Chapter ??.

Listing 18 Implementing the `push`-combinator.

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

❷ **Observe value and stack**

❶ **Push label**

5.6 Summary

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. In this chapter I described a method to construct a computation tree for algorithmic debugging that needs only local annotations and GHC's profiling run-time system.

I implemented my method in the debugger Hoed-stack. Using my debugger on real-world programs already demonstrated its value, case studies with my debugger are discussed in Chapter 9. Thorough testing of my approach with randomly generated expression gives confidence in the soundness of using my computation trees for algorithmic debugging.

However there is also room for improvement: surplus dependencies can lead the algorithmic debugger to asking unnecessary questions or to a sound but innacurate conclusion. Furthermore, not all run-time environments support the cost centre stack extension. In the next chapter we discuss an alternative method for computatation tree construction that works with any run-time environment and produces trees without surplus dependencies.

6 Pure computation tree construction

In this chapter I show that the value observation trace contains more information than previously thought: I use the value-observation trace to derive the parent-child relation between computation statements for a computation tree *without* using the cost-centre stack from GHC's profiling environment.

To derive dependencies between computation statements I discuss in this chapter a new type of observation trace event: a *request event* is recorded when evaluation of an expression starts (the value of the expression is requested). I refer to an event that records a value as a *response event.* For the construction of a computation statement we only need response events (Section 3.3). Therefore I left out request events from the semantics and example traces in Chapter 3-5.

A request event has a corresponding response event in a value observation trace in the same way that a left parenthesis has a corresponding right parenthesis in a word from the language of balanced parenthesis. I call the pair of a corresponding request and response event a *request-response span.* Spans can appear nested and in sequence in the trace.

There is a non-trivial correspondence between span nesting in a trace and the parent-child relation between computation statements constructed from that trace. The application of an observed function usually has a request-response span for both its result and its argument. Consider for example evaluating the expression `quickCheck prop_notBothOdd` from Listing 19 which gives us the simplified value observation trace from Figure 32. Left of the trace I marked request-response spans for function results with •-brackets, and request-response spans for function arguments with ○-brackets. The result span of the events with index 6 and 11 for `plusOne` is nested within the result spans of all the other functions. However, it is also nested within the

Listing 19 Defective program with observation annotations.

```
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               ←— Defect: primitive div instead of mod

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

argument span of the events with index 5 and 12, and the span of events with index 4 and 13 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 I obtain from the value observation trace the computation tree of Figure 6 on page 15.

My main 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.

6.1 Creating a value observation trace

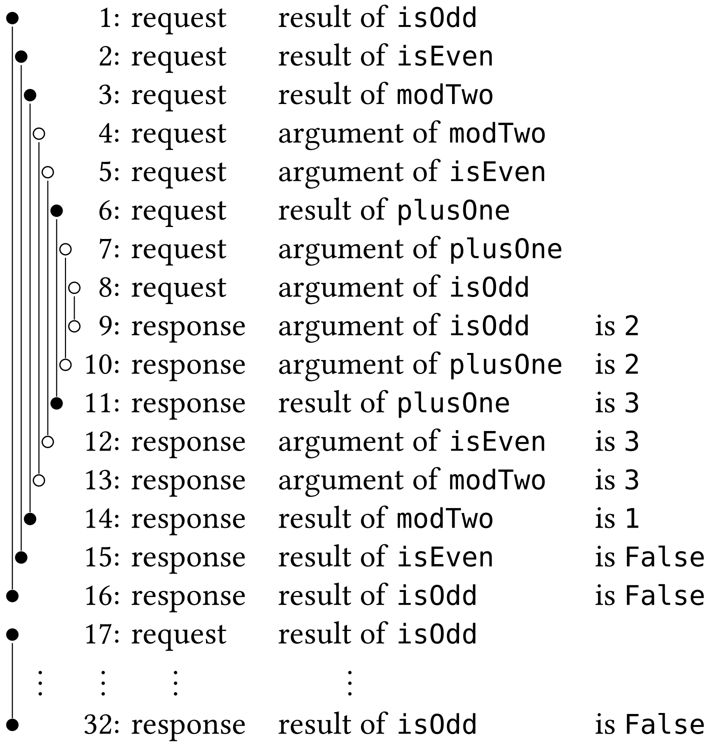
The tracing semantics from Chapter 3 is insufficient here, because it omits the request events that HOOD provides. Request events are unnecessary for reconstructing computation statements from the trace but they are essential for my new method of reconstructing from the trace the parent-child relation for the computation tree. I use the language defined in Figure 10 in this Chapter.

When experimenting with value observation tracing Gill already noted that he can create an event just before evaluation of a thunk starts, and he added this to his library HOOD without having a purpose in mind (Gill and Faddegon 2016). Because request events are not necessary for constructing computation statements without dependencies he did not mention request events in his paper (Gill 2000).

6.1.1 Value observation trace

A trace is a sequence of events as defined in Figure 33. 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. My 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 : 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 (Gill 2000). As we will see, these parents and children



1: request	result of `isOdd`	
2: request	result of `isEven`	
3: request	result of `modTwo`	
4: request	argument of `modTwo`	
5: request	argument of `isEven`	
6: request	result of `plusOne`	
7: request	argument of `plusOne`	
8: request	argument of `isOdd`	
9: response	argument of `isOdd`	is `2`
10: response	argument of `plusOne`	is `2`
11: response	result of `plusOne`	is `3`
12: response	argument of `isEven`	is `3`
13: response	argument of `modTwo`	is `3`
14: response	result of `modTwo`	is `1`
15: response	result of `isEven`	is `False`
16: response	result of `isOdd`	is `False`
17: request	result of `isOdd`	
⋮ ⋮	⋮	
32: response	result of `isOdd`	is `False`

Figure 32: Simplified trace for `prop_notBothOdd 2`.

trace event	$t ::=$	$i\!:\!\text{Root}\ l$	root with label l	
	$\mid$	$\boldsymbol{i}\!:\!\textbf{Enter}\ \boldsymbol{p}$	enter evaluating expression	(request)
	$\mid$	$i\!:\!\text{Con}\ p\ c\ a$	value is application of constructor c	(response)
	$\mid$	$i\!:\!\text{Lam}\ p$	value is an abstraction	(response)
	$\mid$	$i\!:\!\text{MapsTo}\ p$	function application	

Figure 33: Extended syntax of events in observation trace from Figure 11 with a request event (new event in **bold**).

express the relation between expressions and their subexpressions; they are unrelated to the parent/child structure of nodes of the computation tree.

An i : Root f event records the function identifier f supplied by the expression `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 : Con $p\ c\ a$ and j : 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 : 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 : 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.

6.1.2 Semantics

Figure 34 gives the two rules I change in the tracing semantics of Section 3.2.3 to include request events in the trace: before reducing e in `obs` $p\ e$ the ObsLam and ObsCon rules add the request event i : 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 aforementioned request-response spans.

I mentioned in Chapter 2 that the constructor should not appear in a pattern of a case expression to catch an exception. It is worth noting here that the reason for not

$$\frac{\Gamma_1, \boldsymbol{\mathcal{T}_1 \lessdot (i\!:\!\textbf{Enter}\ p)} : e \Downarrow \Gamma_2, \mathcal{T}_2 : c\ x_1 \ldots x_n \qquad \boldsymbol{i = |\mathcal{T}_1|} \qquad j = |\mathcal{T}_2|}{\begin{array}{r}\Gamma_1, \mathcal{T}_1 : \mathtt{obs}\ p\ e \Downarrow \Gamma_2[y_1 \mapsto \mathtt{obs}\ (\mathrm{P_c}\ j\ 1)\ x_1, \ldots, y_n \mapsto \mathtt{obs}\ (\mathrm{P_c}\ j\ n)\ x_n],\\ \mathcal{T}_2 \lessdot (j\!:\!\mathrm{Con}\ p\ c\ (\mathrm{arity}\ c)\ ) : c\ y_1 \ldots y_n\end{array}}\ \mathrm{ObsCon}$$

$$\frac{\Gamma_1, \boldsymbol{\mathcal{T}_1 \lessdot (i\!:\!\textbf{Enter}\ p)} : e \Downarrow \Gamma_2, \mathcal{T}_2 : v_\lambda \qquad \boldsymbol{i = |\mathcal{T}_1|} \qquad j = |\mathcal{T}_2|}{\Gamma_1, \mathcal{T}_1 : \mathtt{obs}\ p\ e \Downarrow \Gamma_2, \mathcal{T}_2 \lessdot (j\!:\!\mathrm{Lam}\ p) : \mathtt{obs}_\lambda\ (\mathrm{P}\ j)\ v_\lambda}\ \mathrm{ObsLam}$$

Figure 34: Updated rules of the tracing semantics of Figure 12 on page 33 to include request events (difference in **bold).**

allowing to catch an exception with a case construct is that this would substantially change the equational theory of the language, but if we would allow such a construct that this would not affect tracing itself.

6.1.3 A trace

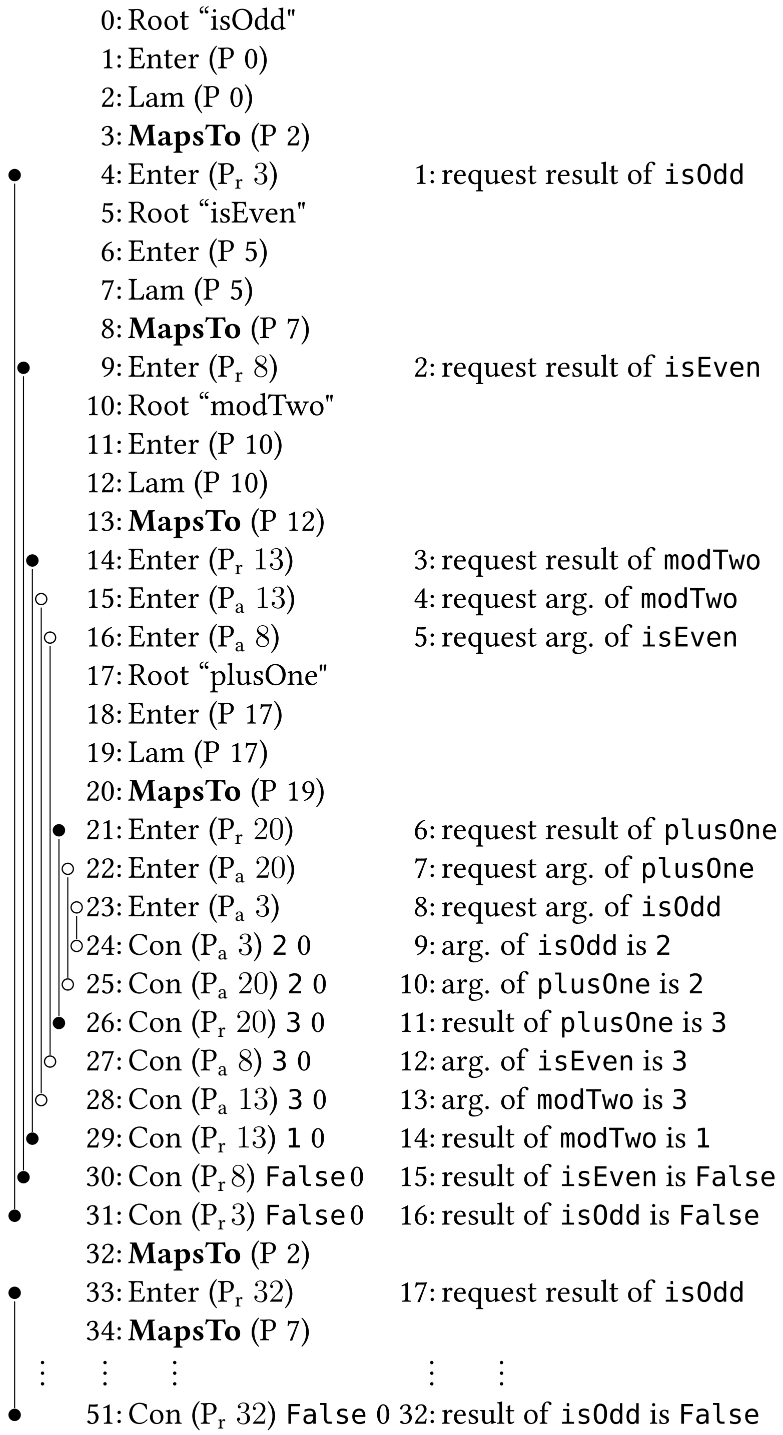
When my introductory example is annotated as in Listing 19, then the semantics gives us the trace shown in Figure 35. On the right side the simplified trace of Figure 32 is given for comparison.

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 $3\!:\!\mathrm{MapsTo}\ (\mathrm{P}\ 2)$ and $32\!:\!\mathrm{MapsTo}\ (\mathrm{P}\ 2)$ have the same parent. They are both recordings of calling the function `isOdd`.

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

6.1.4 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 I 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.



0: Root “isOdd"	
1: Enter (P 0)	
2: Lam (P 0)	
3: **MapsTo** (P 2)	
4: Enter ($\mathrm{P_r}$ 3)	1: request result of `isOdd`
5: Root “isEven"	
6: Enter (P 5)	
7: Lam (P 5)	
8: **MapsTo** (P 7)	
9: Enter ($\mathrm{P_r}$ 8)	2: request result of `isEven`
10: Root “modTwo"	
11: Enter (P 10)	
12: Lam (P 10)	
13: **MapsTo** (P 12)	
14: Enter ($\mathrm{P_r}$ 13)	3: request result of `modTwo`
15: Enter ($\mathrm{P_a}$ 13)	4: request arg. of `modTwo`
16: Enter ($\mathrm{P_a}$ 8)	5: request arg. of `isEven`
17: Root “plusOne"	
18: Enter (P 17)	
19: Lam (P 17)	
20: **MapsTo** (P 19)	
21: Enter ($\mathrm{P_r}$ 20)	6: request result of `plusOne`
22: Enter ($\mathrm{P_a}$ 20)	7: request arg. of `plusOne`
23: Enter ($\mathrm{P_a}$ 3)	8: request arg. of `isOdd`
24: Con ($\mathrm{P_a}$ 3) `2` 0	9: arg. of `isOdd` is `2`
25: Con ($\mathrm{P_a}$ 20) `2` 0	10: arg. of `plusOne` is `2`
26: Con ($\mathrm{P_r}$ 20) `3` 0	11: result of `plusOne` is `3`
27: Con ($\mathrm{P_a}$ 8) `3` 0	12: arg. of `isEven` is `3`
28: Con ($\mathrm{P_a}$ 13) `3` 0	13: arg. of `modTwo` is `3`
29: Con ($\mathrm{P_r}$ 13) `1` 0	14: result of `modTwo` is `1`
30: Con ($\mathrm{P_r}$8) `False`0	15: result of `isEven` is `False`
31: Con ($\mathrm{P_r}$3) `False`0	16: result of `isOdd` is `False`
32: **MapsTo** (P 2)	
33: Enter ($\mathrm{P_r}$ 32)	17: request result of `isOdd`
34: **MapsTo** (P 7)	
⋮ ⋮ ⋮	⋮ ⋮
51: Con ($\mathrm{P_r}$ 32) `False` 0	32: result of `isOdd` is `False`

Figure 35: Full trace (left) with corresponding simplified events (right).

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, I include parents in response events, because HOOD does so, it simplifies some algorithms, and it allows additional sanity checks in my implementation.

Request-response spans are the key to constructing a computation tree from a trace. In Figure 35 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 I do not need trivial spans for constructing a computation tree.

6.1.5 Properties of the trace

Before we continue with constructing a computation tree from the value observation trace, I first give two properties of the trace that help to understand the intuition behind the tree construction algorithm. I used the same approach as in Section 7.4 and verified the properties by testing them against a large set of randomly generated expressions.

A word is in the language of balanced parentheses if and only if the word has both as many left parenthesis as right parenthesis and any prefix of the word has at least as many left parenthesis as right parenthesis (Kozen 2012). Let's consider a request event as a left parenthesis and a response event as a right parenthesis, then given a trace or a part of the trace `numLeft` gives the number of request events in that (part of the) trace and function `numRight` gives the number of response events.

Property 6.1. *Request and response events are like the language of balanced parentheses. This is defined in the following test-property:*

```
prop_balanced e = wellFormed e ⇒
 numLeft 𝒯 == numRight 𝒯 && all geqLeft (prefixes 𝒯)
 where
 geqLeft prefix = numLeft prefix >= numRight prefix
 {},⟨⟩  :  e   ⇓   Γ,𝒯  :  v
```

Both request and response events record the relation to the event's parent in the trace. Assume we are given the function `sameParent` that evaluates, applied to two events, to True when the parent field of the two events is an exact match and to False otherwise. Note that for example sameParent $(\mathrm{P_a}\ 3)$ $(\mathrm{P_r}\ 3)$ is False.

Property 6.2. *A request event (opening parenthese) and corresponding response event (closing parenthese) found by balancing have the same parent. This is defined in the following test-property:*

```
prop_parentsBalanced e = wellFormed e ==> parentsBalanced trc []
 where
 {},⟨⟩  :  e   ⇓   Γ,𝒯  :  v
 parentsBalanced [] stk = stk == []
 parentsBalanced (t:trc) stk
  | isReq  t  = parentsBalanced trc (t:stk)
  | isResp t  = case stk of
      []        -> False
      (t':stk') -> sameParent t t' && parentsBalanced trc stk'
  | otherwise = parentsBalanced trc stk
```

Thus the request and response event found by balancing form a span.

6.2 From trace to computation tree

I now have a precise definition of the value observation trace and have to obtain from it a computation tree. In the following I assume that I observe only top-level variables bound to λ-abstractions, such as isOdd, which is bound to λn. isEven (plusOne n) in Listing 2. I discussed the reasons for this restriction in Section 7.4.4.

Ignoring request events we can just following the method I discussed in Section 3.3 to construct the nodes of the computation tree. I now discuss how to add the edges, that is, the parent-child relation of the computation tree.

6.2.1 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 I focus on the sequential structure of the value observation trace.

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

The forest of event trees tells us for every response event to which computation statement it belongs. Similarly I 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, I 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 I 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.

6.2.2 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 I 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 my language is higher-order, not every argument span is negative and not every result span positive.

Consider the higher-order program in Listing 20. Because function `f` uses and calls function `i` ❶, I expect the computation tree to look as shown in Figure 36. Function

Listing 20 A higher-order program.

```
let { i=observe "i" (λz.z),
      f=observe "f" (λg.let {x=42,y=i x}  ⟵ ❶ Applies i
                         in g y), ⟵ ❷ Applies argument
      h=λu.u
    } in f h   ⟵ ❸ Function h as argument
```

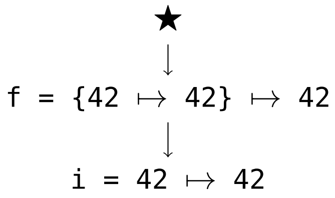


Figure 36: Computation tree for the higher-order program.

h is passed as argument to function f ❸, but inside the body of the f function h is applied to an argument ❷ and the subcomputation for this argument has to be a child computation statement for the computation statement of f.

Figure 37 shows the event trees of the value observation trace. All spans of the computation statement i = 42 ↦ 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 ↦ 42 to be considered a child of the computation statement f = {42 ↦ 42} ↦ 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:

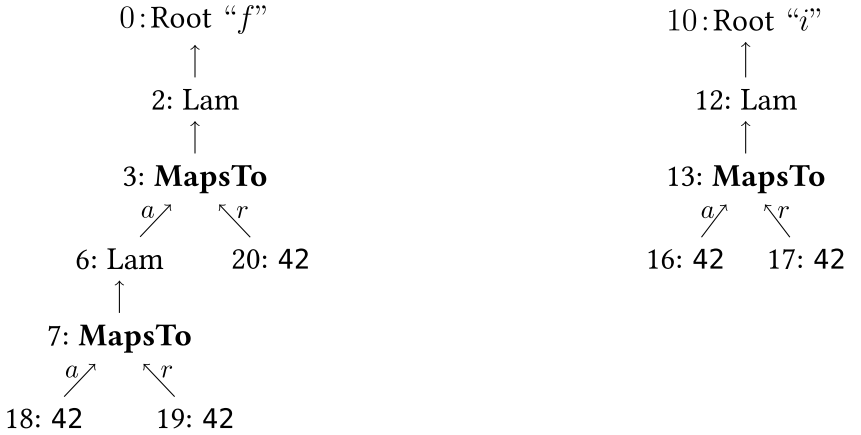


Figure 37: Event trees of the higher-order program.

$$\text{isPos } i = \begin{cases} \text{True} & \text{, if } t_i = i\!:\!\text{Root } f \\ \text{not (isPos } k) & \text{, if } p_i = \text{P}_\text{a}\ k \\ \text{isPos } k & \text{, if } p_i = \text{P}_\text{r}\ k \text{ or P } k \text{ or } \text{P}_\text{c}\ k\ m \end{cases}$$
$$\text{where } p_i \text{ is the parent field of event } t_i$$

6.2.3 Constructing the edges of the computation tree

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

Listing 21 shows the final algorithm for constructing a computation tree. The function `mkCompTree` takes a list of computation statements, a value observation trace (a list of events), a current node and a current tree to produce the final computation tree. I initially call `mkCompTree` with the whole value observation trace, the root node ★, and a tree without edges that has the root node and all previously constructed computation statements as nodes. For this first application `(e:es) = trc`. The list of computation statements can be constructed from the list of events with the method from Section 3.3.2.

Given an event and a value observation trace the following functions

```
isStartOfPosSpan :: Event -> [Event] -> Bool
isEndOfPosSpan   :: Event -> [Event] -> Bool
```

```
isStartOfNegSpan :: Event -> [Event] -> Bool
isEndOfNegSpan   :: Event -> [Event] -> Bool
```

determine whether the event starts or ends a span and whether the span is positive or negative. The start and end of a span correspond with the request and response events of Figure 33. Section 6.2.2 describes how to determine whether a span is positive or negative. The function

```
mkChild :: CompStmt -> CompStmt -> CompTree -> CompTree
```

applied to the computation statements `m` and `n`, and a computation tree `t`, evaluates to a computation tree with all edges in `t`, and a new edge from `n` to `m`. Finally

```
statementOf :: Event -> [CompStmt] -> CompStmt
```

evaluates, given an event `e` and the list of computation statement, to the computation statement associated with `e`. That is, the computation statement is constructed from an event tree containing `e` (see Section 3.3.2). When a response event `e'` is associated with a statement `n` then the request event that forms a span with `e'` is also associated with `n`.

Listing 21 Algorithm for constructing the computation tree.

```
mkCompTree :: [CompStmt] -> [Event] -> [Event]
           -> CompStmt -> CompTree -> CompTree
mkCompTree ns []     trc n tree = tree
mkCompTree ns (e:es) trc n tree
 | isStartOfPosSpan e trc =
    if isChildOf m n tree
     then mkCompTree ns es trc m tree                   | Discover
     else mkCompTree ns es trc m (mkChild m n tree) ⟵ ❶ | dependency
 | isEndOfPosSpan e trc || isStartOfNegSpan e trc =
    mkCompTree es trc (parentOf n tree) tree ⟵ ❷ | Computation termi-
 | isEndOfNegSpan e trc =                         | nated or suspended
    mkCompTree es trc m tree ⟵ ❸ | Computation
 | otherwise =                    | continues
    mkCompTree es trc n tree
 where m = statementOf e ns
```

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 ❶. The algorithm continues with `m` as current computation statement.

The end of a positive span terminates the current computation and the start of a negative span suspends the current computation. Therefore the current node after an event that is the end of a positive span or the start of a negative span is the parent of the current node before that event ❷.

The end of a negative span indicates that the computation of the statement `m` associated with the span, that was suspended at the beginning of the negative span, continues. Hence, the current statement is `m` again ❸.

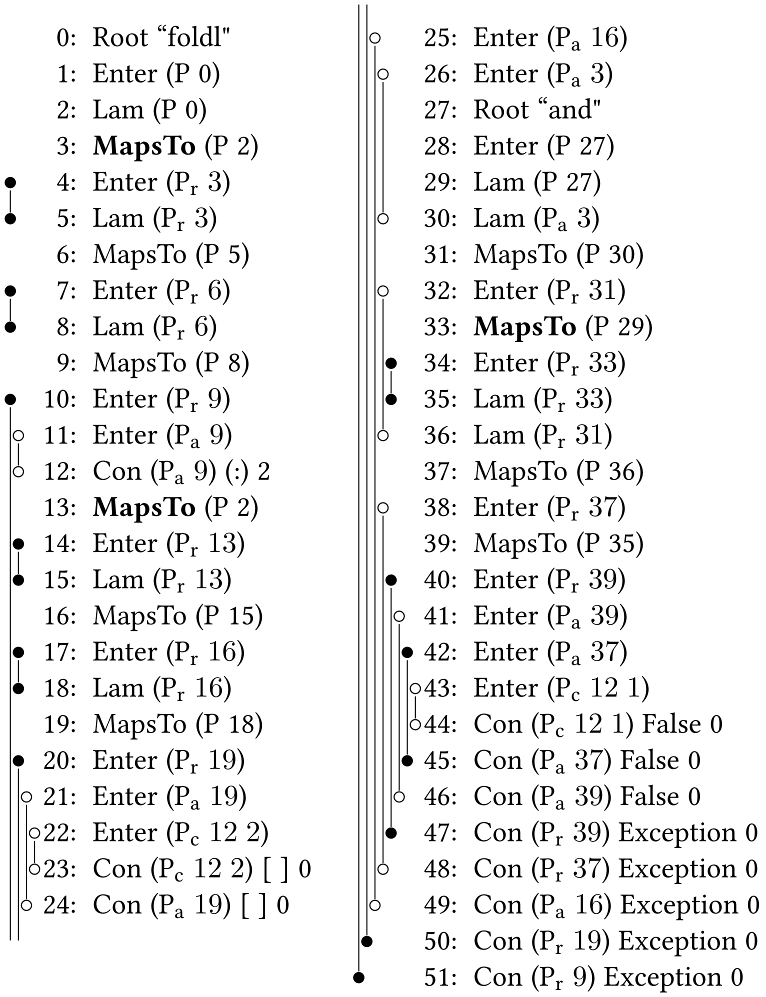
In every step of the algorithm the current node `n` has a parent all the way up to the root node ★. At the end of traversing the trace I have the computation tree. In the resulting tree every statement has exactly one parent: either the root node ★ or another statement.

When a statement of a labelled expression is wrong either the expression is wrong or the computation depends on another computation that is also wrong. For sound algorithmic debugging therefore a computation statement should depend on another computation statement if the latter computation contributes to the former. Events nested in a positive span contribute to the span's computation statement and events nested in a negative span do not (Section 6.2.2). Hence a computation statement in a computation tree constructed with the algorithm of Listing 21 depends on any computation statement that contributed to the former. From this it follows that a computation tree constructed with the algorithm of Listing 21 is sound for algorithmic debugging.

6.2.4 Example construction of a computation tree

The program in Listing 22 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 38. I 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



0: Root “foldl"
1: Enter (P 0)
2: Lam (P 0)
3: **MapsTo** (P 2)
4: Enter (P_r 3)
5: Lam (P_r 3)
6: MapsTo (P 5)
7: Enter (P_r 6)
8: Lam (P_r 6)
9: MapsTo (P 8)
10: Enter (P_r 9)
11: Enter (P_a 9)
12: Con (P_a 9) (:) 2
13: **MapsTo** (P 2)
14: Enter (P_r 13)
15: Lam (P_r 13)
16: MapsTo (P 15)
17: Enter (P_r 16)
18: Lam (P_r 16)
19: MapsTo (P 18)
20: Enter (P_r 19)
21: Enter (P_a 19)
22: Enter (P_c 12 2)
23: Con (P_c 12 2) [ ] 0
24: Con (P_a 19) [ ] 0
25: Enter (P_a 16)
26: Enter (P_a 3)
27: Root “and"
28: Enter (P 27)
29: Lam (P 27)
30: Lam (P_a 3)
31: MapsTo (P 30)
32: Enter (P_r 31)
33: **MapsTo** (P 29)
34: Enter (P_r 33)
35: Lam (P_r 33)
36: Lam (P_r 31)
37: MapsTo (P 36)
38: Enter (P_r 37)
39: MapsTo (P 35)
40: Enter (P_r 39)
41: Enter (P_a 39)
42: Enter (P_a 37)
43: Enter (P_c 12 1)
44: Con (P_c 12 1) False 0
45: Con (P_a 37) False 0
46: Con (P_a 39) False 0
47: Con (P_r 39) Exception 0
48: Con (P_r 37) Exception 0
49: Con (P_a 16) Exception 0
50: Con (P_r 19) Exception 0
51: Con (P_r 9) Exception 0

Figure 38: Trace of computation with higher order function.

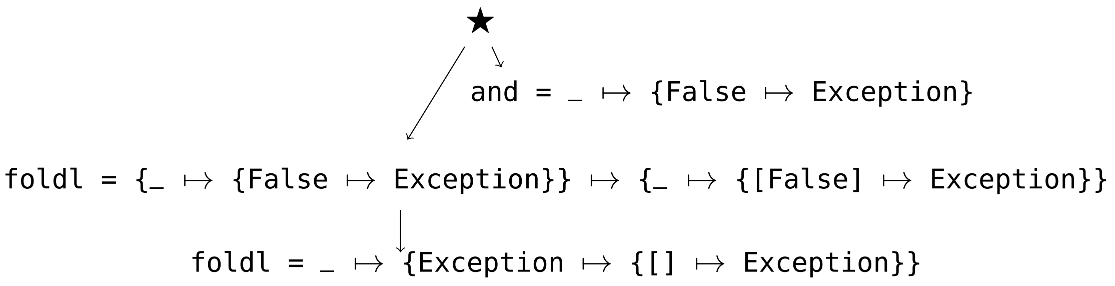

★
and = _ ↦ {False ↦ Exception}
foldl = {_ ↦ {False ↦ Exception}} ↦ {_ ↦ {[False] ↦ Exception}}
foldl = _ ↦ {Exception ↦ {[] ↦ Exception}}

Figure 39: Computation tree for trace of Figure 38.

Listing 22 Example program with observations.

```
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
```

traversal of the event sequence first reaches the span $\langle 4, 5\rangle$. Consequently the node for `foldl` that corresponds to the event $3 : \text{MapsTo}$ 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 $3 : \text{MapsTo}$ and hence the computation statement for the event $13 : \text{MapsTo}$ becomes its child. Several subsequent spans change the current node but only at span $\langle 34, 45\rangle$ the computation statement for `and` that corresponds to the event $33 : \text{MapsTo}$ 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 39 shows the final tree. The computation statement for `and`, which given a second argument `False` raises an exception, indicates a defect.

6.2.5 Examples of particular language features

Data structures The program in Listing 23 defines a data type `T` with data constructors `C` and `D` that each take one argument. The `notD` function works on this new data type. We want to check the result of evaluating `notD (D True)`. However, our semantics only tells us that the value of that expression is `D x` for some heap variable `x`. Hence we add the function `getC` to our program. We trust that its definition is correct. To keep the trace small, we do not observe `getC`. The computation of `getC (notD (D True)` returns 1, although we expected 0.

Figure 40 shows the trace of our computation: on the left side as sequence of events with spans marked and on the right side as forest of two event trees. Constructing the two computation statements from the two event trees is straightforward. However, in this trace there are two positive spans for the computation of `notD`: the span $\langle 4, 7\rangle$ for the constructor `D` and the span $\langle 8, 19\rangle$ for the argument of this constructor. Similarly there are two negative spans for the computation of `notD`: the span $\langle 5, 6 \rangle$ for the constructor `D` and the span $\langle 15, 16 \rangle$ for the argument of this constructor. So because of data structures a computation statement may have several negative and several positive spans.

The method for building the computation tree is nonetheless as discussed before. Here the positive span $\langle 13, 18 \rangle$ of `not` is directly within one of the positive spans of `notD`, namely $\langle 8, 19 \rangle$, and hence the computation statement of the former is a direct child of the computation statement of the latter. Figure 41 displays the computation tree. The tree clearly shows that there is a defect in the definition of the function `not` for the argument `True`.

Listing 23 Annotated program with data structures.

```
data T = C Int | D Bool

not = observe "not" not'
not' True  = True      ⟵ Defect: True in definition should be False
not' False = True

notD = observe "notD" notD'
notD' (C i) = C i
notD' (D b) = D (not b)

getC (C i) = i
getC (D True)  = 1
getC (D False) = 0
```

Multi-Argument Functions The program in Listing 24 defines the logical connectives `and` and `or`. Each function takes two arguments. Our core language and Haskell support multi-argument functions through currying and hence we can view this program as a simple example of defining higher-order functions. This program also demonstrates that lazy evaluation does not evaluate all arguments, some MapsTo events in our trace lack arguments, and hence the unknown value _ appears in some computation statements.

Figure 42 shows the trace for evaluating `or False True` to the unexpected result `False`. On the left side is the sequence of events with marked spans and on the right side is the forest of three event trees.

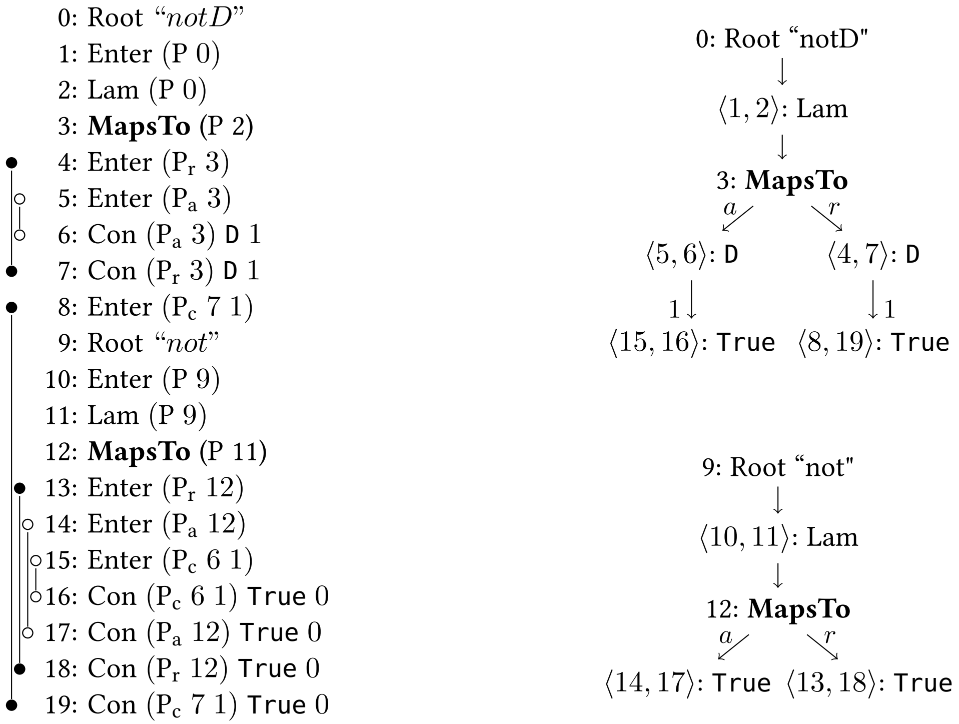


Figure 40: Trace and event trees for evaluation of `getC (notD (D False))`.

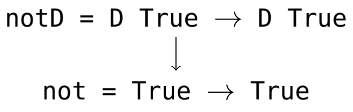


Figure 41: Computation tree for trace of Figure 40.

There are three event trees for the three observation annotations, but the trace yields four computation statements. The function `not` is applied twice in the computation, yielding the two MapsTo events at indices 11 and 23. Recall that each computation statement corresponds to a MapsTo event that has a Root event as grandfather.

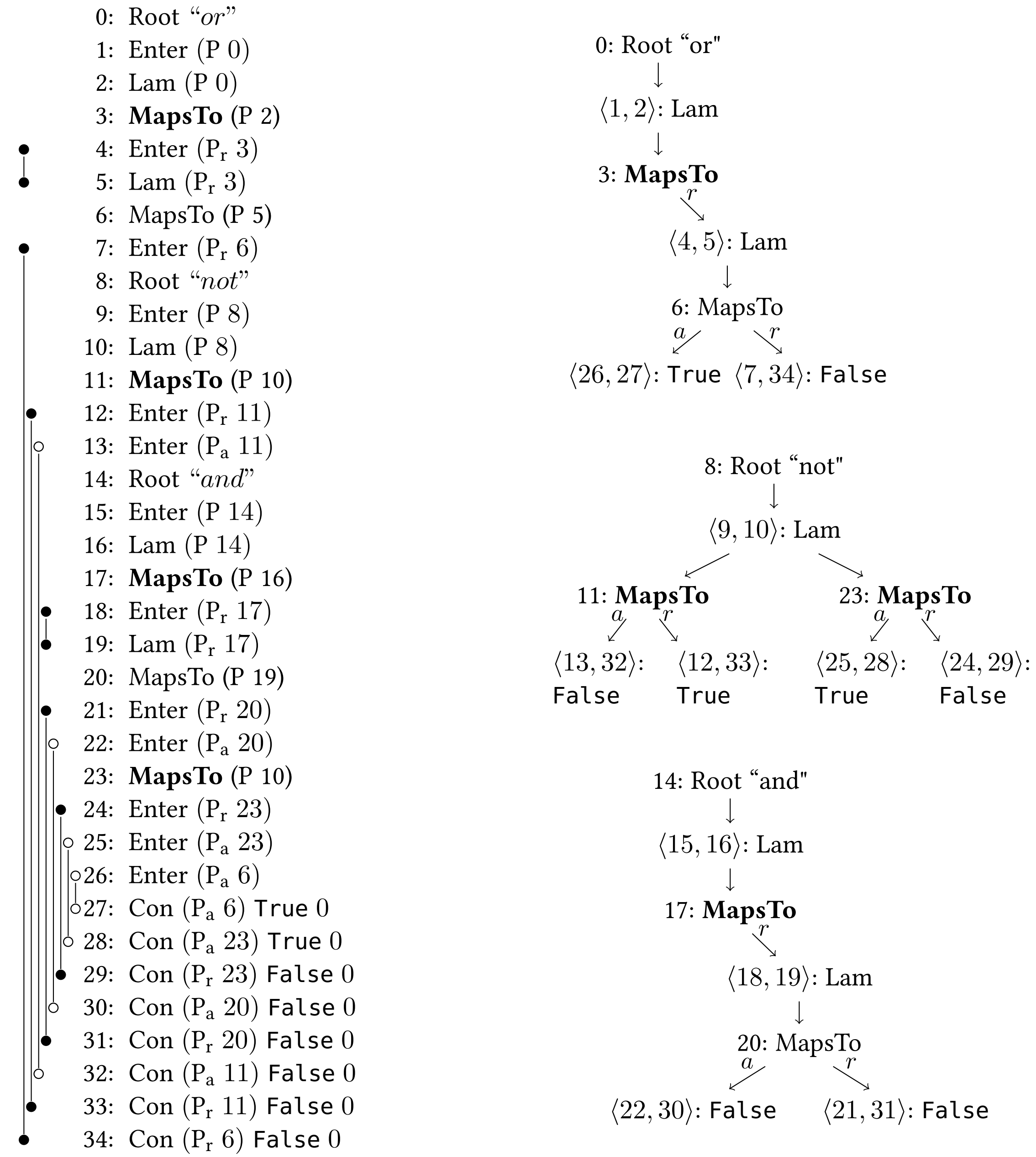
Consider the event tree of function `or`. The MapsTo event at index 3 has as result a Lam event, which is applied at index 6 to the second function argument, which is computed in the span $\langle 26, 27\rangle$. So this tree structure clearly shows currying taking place. For the construction of the computation tree we can view Lam and MapsTo events in the result of a function similar to a data structure. A computation statement just has several positive and negative spans. As before, the events for an argument of a MapsTo event indicate that computation of the corresponding computations statement is supspended, so for example $\langle 26, 27\rangle$ is a negative span.

To construct the computation tree we first note that the two outermost spans $\langle 4, 5\rangle$ and $\langle 7, 34\rangle$ are both positive and belong to the computation statement of `or`, which hence is the root of the computation tree. One of the computation statements of `not` has the positive span $\langle 12, 33\rangle$ and hence is a child of the root. Furthermore the spans $\langle 18, 19\rangle$ and $\langle 21, 31\rangle$ are positive spans of `and`, they are within the gap of the negative span of `not`, and hence its computation statement is also a child of the root. Finally the second application of `not` has positive span $\langle 24, 29\rangle$, which because of two gaps made by negative spans is directly within a span of `or` and hence also is a child of the computation statement of `or`. Figure 43 displays the computation tree. The tree shows that there is a defect in the definition of the function `not` for the argument `False`.

Listing 24 Annotated program with multi-argument functions.

```
not = observe "not" not'
not' True  = False
not' False = False

and = observe "and" and
and' b True = b
and' b False = False

or = observe "or" or'
or' b d = not (and (not b) (not d))
```

Higher order functions The program in Listing 25 defines the third-order function `appN`, which takes the function `appT` as argument, which receives the function `not` as



0: Root “*or*”
1: Enter (P 0)
2: Lam (P 0)
3: **MapsTo** (P 2)
4: Enter (P_r 3)
5: Lam (P_r 3)
6: MapsTo (P 5)
7: Enter (P_r 6)
8: Root “*not*”
9: Enter (P 8)
10: Lam (P 8)
11: **MapsTo** (P 10)
12: Enter (P_r 11)
13: Enter (P_a 11)
14: Root “*and*”
15: Enter (P 14)
16: Lam (P 14)
17: **MapsTo** (P 16)
18: Enter (P_r 17)
19: Lam (P_r 17)
20: MapsTo (P 19)
21: Enter (P_r 20)
22: Enter (P_a 20)
23: **MapsTo** (P 10)
24: Enter (P_r 23)
25: Enter (P_a 23)
26: Enter (P_a 6)
27: Con (P_a 6) `True` 0
28: Con (P_a 23) `True` 0
29: Con (P_r 23) `False` 0
30: Con (P_a 20) `False` 0
31: Con (P_r 20) `False` 0
32: Con (P_a 11) `False` 0
33: Con (P_r 11) `False` 0
34: Con (P_r 6) `False` 0

Figure 42: Trace for evaluation of `or False True`.

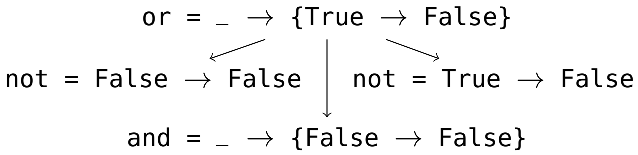


Figure 43: Computation tree for trace of Figure 42.

an argument.

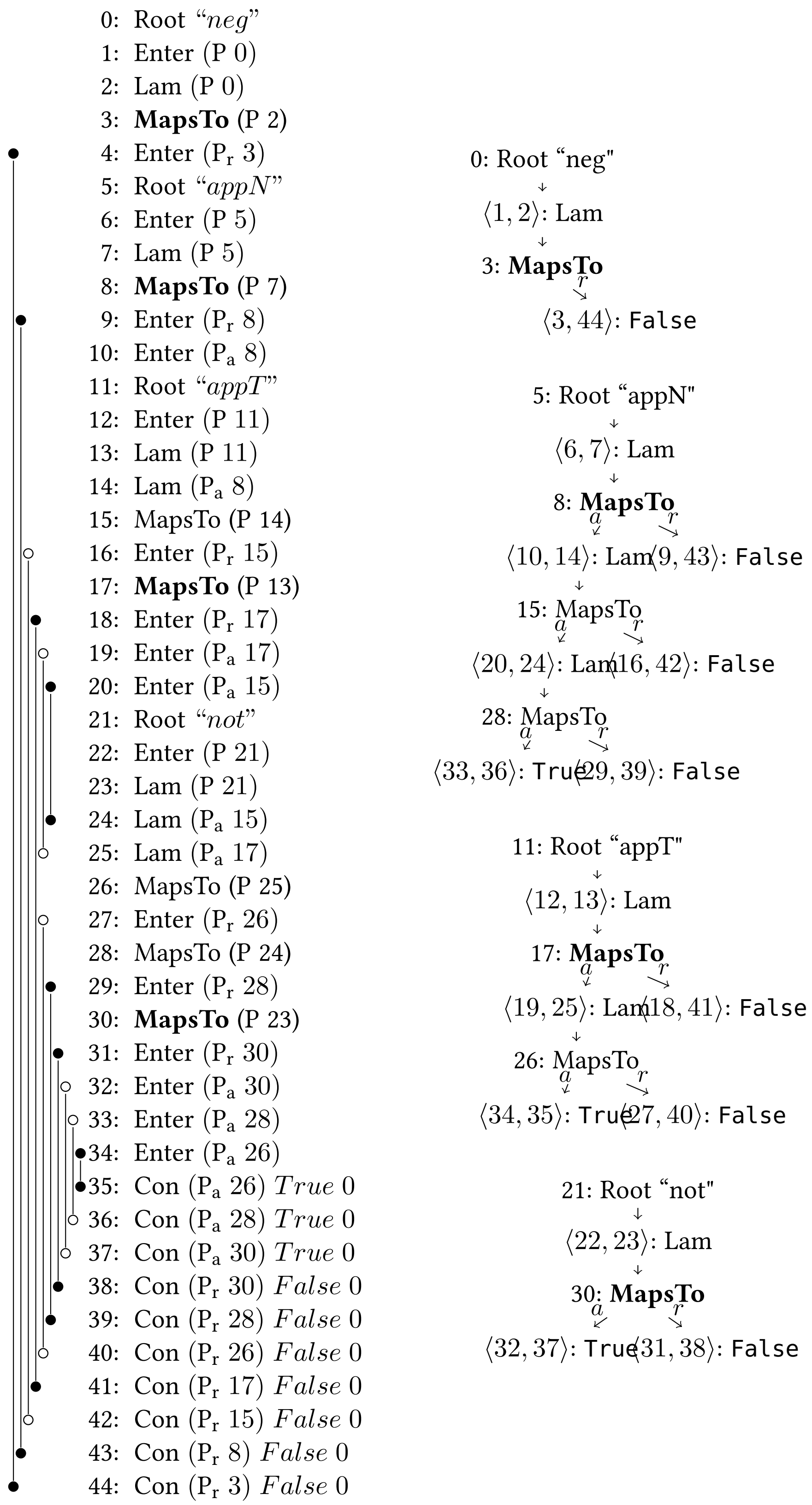
Figure 44 shows the trace for evaluating `neg False` to the unexpected result `False`. Again the figure shows the trace as sequence of events and forest of event trees.

Why, for example, is the span $\langle 16, 42\rangle$ marked as negative, even though it is for computing the result of the MapsTo event at index 15? The reason is that this MapsTo event is an argument of the MapsTo event at index 8. So the computation of this result actually serves to construct the argument for function `appN`. When we have functions as arguments (or as parts of data structures), then to determine whether a span contributes positively to the computation statement, we have to consider in the event tree the path from the span upwards to the MapsTo event of the computation statement (see Section 6.2.2).

From the marked spans left of the trace events the structure of the computation tree given in Figure 45 follows. The root of the computation tree `neg = _ → False` already clarifies that the function `neg` always returns `False`, ignoring its argument. Assuming that the semantics of `appN` is as intended by the programmer, the computation tree identifies the definition of `neg` as defective.

Listing 25 Annotated program with third-order function.

```
not = observe "not" not’
not False = True
not True = False

appT = observe "appT" appT’
appT’ f = f True

appN = observe "appN" appN’
appN’ g = g not

neg = observe "neg" neg’
neg b = appN appT
```



0: Root “*neg*”
1: Enter (P 0)
2: Lam (P 0)
3: **MapsTo** (P 2)
4: Enter ($\mathrm{P_r}$ 3)
5: Root “*appN*”
6: Enter (P 5)
7: Lam (P 5)
8: **MapsTo** (P 7)
9: Enter ($\mathrm{P_r}$ 8)
10: Enter ($\mathrm{P_a}$ 8)
11: Root “*appT*”
12: Enter (P 11)
13: Lam (P 11)
14: Lam ($\mathrm{P_a}$ 8)
15: MapsTo (P 14)
16: Enter ($\mathrm{P_r}$ 15)
17: **MapsTo** (P 13)
18: Enter ($\mathrm{P_r}$ 17)
19: Enter ($\mathrm{P_a}$ 17)
20: Enter ($\mathrm{P_a}$ 15)
21: Root “*not*”
22: Enter (P 21)
23: Lam (P 21)
24: Lam ($\mathrm{P_a}$ 15)
25: Lam ($\mathrm{P_a}$ 17)
26: MapsTo (P 25)
27: Enter ($\mathrm{P_r}$ 26)
28: MapsTo (P 24)
29: Enter ($\mathrm{P_r}$ 28)
30: **MapsTo** (P 23)
31: Enter ($\mathrm{P_r}$ 30)
32: Enter ($\mathrm{P_a}$ 30)
33: Enter ($\mathrm{P_a}$ 28)
34: Enter ($\mathrm{P_a}$ 26)
35: Con ($\mathrm{P_a}$ 26) $True$ 0
36: Con ($\mathrm{P_a}$ 28) $True$ 0
37: Con ($\mathrm{P_a}$ 30) $True$ 0
38: Con ($\mathrm{P_r}$ 30) $False$ 0
39: Con ($\mathrm{P_r}$ 28) $False$ 0
40: Con ($\mathrm{P_r}$ 26) $False$ 0
41: Con ($\mathrm{P_r}$ 17) $False$ 0
42: Con ($\mathrm{P_r}$ 15) $False$ 0
43: Con ($\mathrm{P_r}$ 8) $False$ 0
44: Con ($\mathrm{P_r}$ 3) $False$ 0

Figure 44: Trace for evaluation of `neg False`.

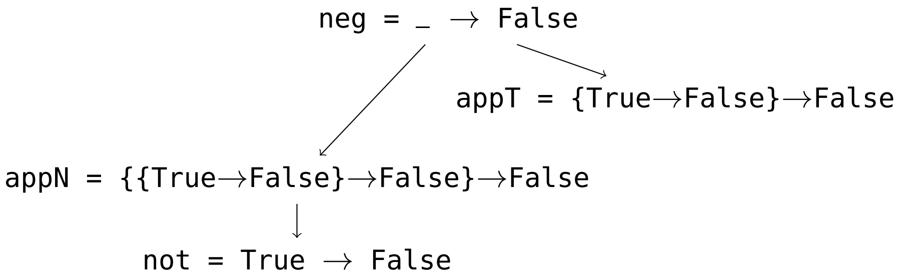


Figure 45: Computation tree for trace of Figure 44.

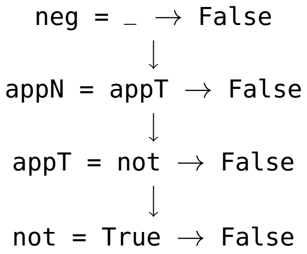


Figure 46: Evaluation dependency tree (EDT) for `neg False`.

In Section 2.2.1 I discussed the two tree structures, evaluation dependency tree (EDT) and function dependency tree (FDT), that are both computation trees for algorithmic debugging. With the method discussed in this chapter we construct FDTs. Consider the EDT in Figure 46 for the evaluation of `neg False` with the program of Listing 25, in contrast to the FDT shown in Figure 45.

6.2.6 Properties of the computation tree

Property 6.3. *All computation statements are reachable from the root. This is defined in the following test-property:*

```
prop_connected e =
 wellFormed e ⇒ all (reachableRoot []) (statements tree)
 where
 reachableRoot seen ★ = True
 reachableRoot seen c | c ‘elem‘ seen = False
 reachableRoot seen c = case predecessors t c of
  [c’] -> reachableRoot seen c’
  _    -> False
 t = mkCompTree 𝒯
 {},⟨⟩  :  e   ⇓   Γ,𝒯  :  v
```

A computation graph from the previous chapter approximates a computation tree and has surplus edges: edges that are not necessary for sound algorithmic debugging and that can make it harder to locate the defect (Section 2.2.1). The algorithm of Listing 21 processes the trace and at the start of a positive span adds an edge from the current node to the node associated with the start of the positive span ❶. Several positive spans can be associated with the same node; multiple successors can be added but only one predecessor.

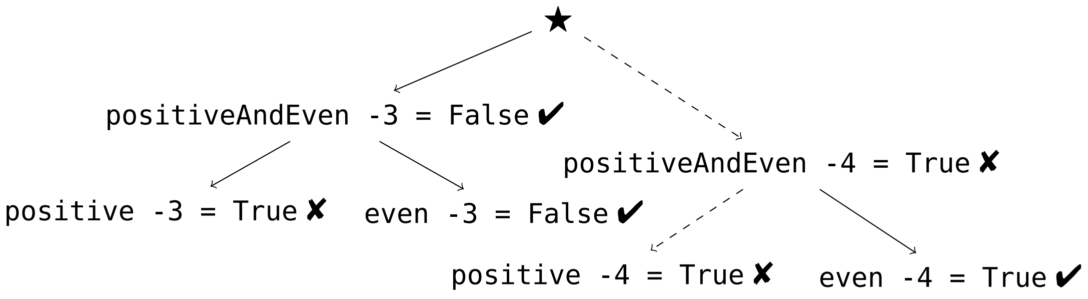
Property 6.4. *A statement in the computation tree has no more than one parent statement. This is defined in the following test-property:*

```
prop_minimal :: Expr -> Property
prop_minimal e =
 wellFormed e ⇒ all (\c -> length (predecessors t c) <= 1)
                    (statements t)
 where
 t = mkCompTree 𝒯
 {},⟨⟩  :  e   ⇓   Γ,𝒯  :  v
```

Thus the computation tree is indeed a tree and not a graph and a computation statement has a unique predecessor which I will call the parent:

```
    parent t c | c /= ★ = case predecessors t c of [p] -> p
```

In an observable misbehaving program there is a chain of wrong statements that connects the root node ★ with a faulty node. Consider for example path of dashed arrows (⇢) in the computation tree below in which statements are annotated with right/wrong (✔/✘) judgements:



There can be an additional faulty statement that is wrong but did not cause the program to misbehave, for example in above computation tree, because -3 is not even the (wrong) outcome of `positive -3` is irrelevant for the behaviour of the program. I do not model that by chance an infection is stopped in my test environment.

Property 6.5. *For every statement that is wrong, the parent statement is also wrong or the parent is the special root node ★. This is defined in the following test-property:*

```
prop_reachable e = wellFormed e ⇒
 all parentWrong (filter (\c -> c /= ★ && isWrong c)
                        (statements t))
 where
 parentWrong c = let p = parent t c in p == ★ || isWrong p
 t = mkCompTree 𝒯
 {}, ⟨⟩  :  e   ⇓   Γ, 𝒯  :  v
```

When we invert this property we get: a wrong statement is reachable from the special root node ★ via a chain of wrong statements. This is true in my test environment where an infected value always causes further infection but in a real program infection may be stopped. However, a program that produced the wrong result must have at least one chain were the infection was not stopped.

6.3 My algorithmic debugger Hoed-pure

I implemented my 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 (Section **??**).

The manipulation of the trace in my 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 (2000) already

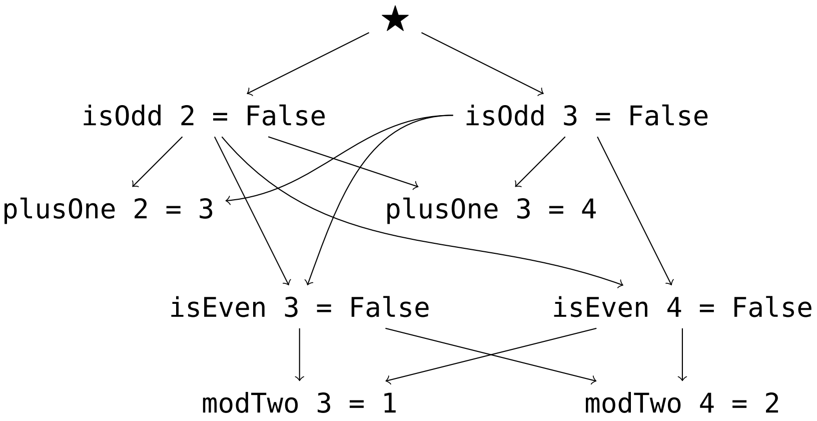


Figure 47: Hoed-cc's computation tree (graph).

argues that a compiler is unlikely to change the order of the side-effects and I have not observed any such problem in practice.

My semantics describes how to handle exceptions in principle, but in Haskell exceptions are not simple constructors. Hence my 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.

6.3.1 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 chapter. The computation tree can still help the programmer understand why a program is misbehaving but algorithmic debugging is not guaranteed to find the defect.

6.4 Comparison with Hoed-cc

Hoed-cc, discussed in the previous chapter, is a library that combines value observation tracing 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 run-time

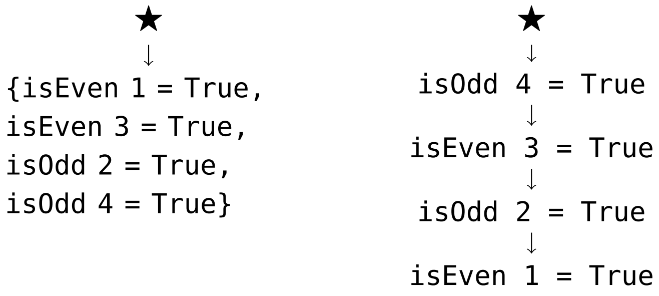


Figure 48: Computation trees of Hoed-cc and Hoed-pure

system with cost centre stack support. For example, the interpreter GHCi does not support cost centre stacks. Hoed-pure works with any Haskell run-time system and the pure computation tree construction method can easily be implemented for other functional languages.

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 47 shows Hoed-cc's computation tree (actually an acyclic directed graph) for the example I used in the beginning of this chapter. Compare it with Figure 6. 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 tree of Figure 6, constructed purily from the observation trace, may require up to 5 questions and the tree from Figure 47, constructed with cost centre stacks, may require up to 8 questions.

Algorithmic debugging with a Hoed-pure computation tree locates the defect more precisely because a node in a Hoed-cc computation tree may contain a set of computation statements. Consider evaluating `isOdd 4` for the program in Listing 26. Figure 48 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.

Listing 26 Program with mutual recursion.

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

6.5 Summary

I have presented a new lightweight method for generating a computation tree. The starting point is my 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. My tracing semantics is specific to lazy evaluation, but my 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.

I implemented my 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. I showed that the algorithmic debugger asked substantially fewer questions using Hoed-pure’s computation tree.

7 Sound algorithmic debugging

Different kind of computation statements and dependencies exist. Some of these are sound for algorithmic debugging. Firstly I discuss constants and free variables and why I only observe top-level functions with my method. Secondly I explain how to test an algorithmic debugger and how I tested the methods for computation tree construction described in the previous chapters.

7.1 Constants in a computation tree

Consider construction of a computation tree for the computation

```
prop_conj_commutative True False
```

from Listing 27. How should the constant `ctable` be represented in the computation tree? There are at least three possible trees that can be used for sound algorithmic debugging:

Figure 49-a: The computation tree has a statement for each use of the constant. Such a statement shows the part of the value demanded for that use of the constant.

Figure 49-b: The computation tree has one statement s per constant, showing all parts of the constant that are demanded at some point during evaluation of the program. When a computation depends on the value of the constant, then the computation statement depends on s. Thus, the computation tree is actually a directed acyclic graph, copying s for each statement dependent on s gives a tree.

Figure 49-c: The computation tree has a separate subtree directly nested under the special root node ★ for the constant statement and its dependencies. The constant statement shows all parts of the statement that are demanded at some point during evaluation of the program. For sound algorithmic debugging the statements of constants need to be considered first, a constant that was demanded before another constant needs to be considered before a constant that was demanded later (Nilsson and Sparud 1997). When different parts of constants are demanded at different times during the evaluation, it is hard to determine which constant is to be considered first.

Listing 27 Conjunction and disjunction defined by a lookup table.

```
(h:t) !! 0 = h
(h:t) !! i = t !! (i-1)

toInt False False = 0
toInt False True  = 1
toInt True  False = 2
toInt True  True  = 3
```

← **Interpret two boolean values as bit 1 (with value $2^1 = 2$) and bit 0 (with value $2^0 = 1$) of a two bit integer value**

```
disj b1 b0 = dtable !! (toInt b1 b0)
dtable = [False, True, True, True]
conj b1 b0 = ctable !! (toInt b1 b0)
ctable = [False, False, True, True]
```

← **Defect: value at position 2 should be `False` (because `True`∧`False` = `False`)**

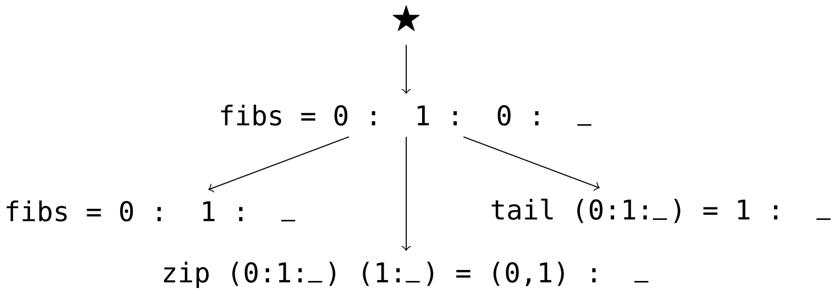
```
prop_disj_commutative x y = disj x y == disj y x
prop_conj_commutative x y = conj x y == conj y x
```

A constant definition can contain function applications and constants. Consider for example the following recursive definition of a list with all numbers in the Fibonacci sequence:

↓ **Defect: ∗ should be +**

```
fibs = 0 : 1 : [ a * b | (a, b) <- zip fibs (tail fibs)]
```

When the constant or applied function is also traced, then the `fibs`-computation statement depends on the statement of the function application or constant from the definition. Thus, the computation tree of `fibs !! 2` is:



In Section 7.3 I discuss programs with constants and my tracing method.

7.2 Free variables in a computation statement

As discussed in Section 2.1.1, a variable can occur bound or free in an expression. Consider for example free variable `xs` in local function `myFilter` of Listing 28 on page 106.

In Section 2.2 I defined a computation statement as a function applied to argument values and the result value. Consider the following computation statement for the function `myFilter` from Listing 28:

```
myFilter [3,2,4] = [3,2,4]
```

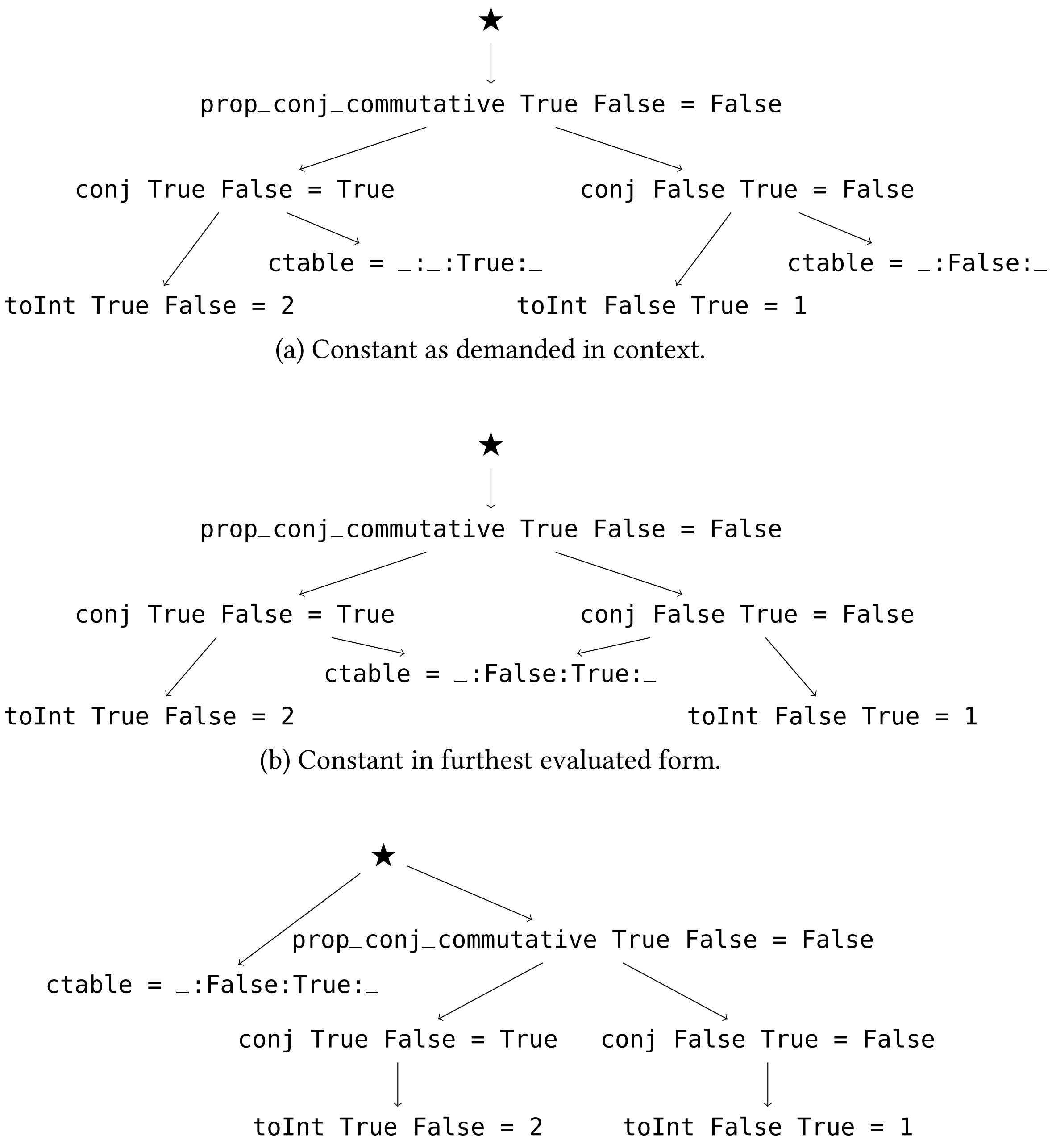
Is this statement right or wrong?

There is not enough information in above computation statement for the oracle to judge the correctness of the statement: if `xs` was for example `[1,3,2,4]` then the statement is wrong, but the statement is right when `xs` is for example `[5,3,2,4]`. To soundly reason about the computation statement it should include name and value of any free variable used in the definition associated with the statement's function (Nilsson 1998). That is, above computation statement should really be:

```
myFilter [3,2,4] = [3,2,4] where xs = [5,3,2,4]
```

The function `minimum` is also free in the definition of `myFilter`, however, there is no need to add `minimum` to the `myFilter`-statement because the oracle judges the `myFilter`-statement assuming `minimum` behaves as expected.

In the rest of this thesis I avoid having to keep track of free variables by only tracing top-level functions and constants. Turning local functions into top-level functions is a well known transformation (Peyton Jones and Lester 1992, Chapter 6).



(a) Constant as demanded in context.

(b) Constant in furthest evaluated form.

(c) Constant as seperate subtree.

Figure 49: Three different trees of computating `prop_conj_commutative True False`. Library function `(!!)` omitted.

Listing 28 Defective program to remove all occurances of the minimum from a list.

Defect: to return smallest of `h` and `m` the expression `h<m` should be `h>m`.

```
minimum = λ xs . case xs of {
    [x] → x,
    (h:t) → let m = minimum t in if h < m then m else h }

rmMinimum = λ xs .
  let myFilter = λ ys . case ys of {
              [] → [],
              (h:t) → if h == minimum xs then minimum t
                                              else h : minimum t }
  in myFilter xs
```

Variable `xs` is free in definition of local function `myFilter` and bound in definition body `rmMinimum`.

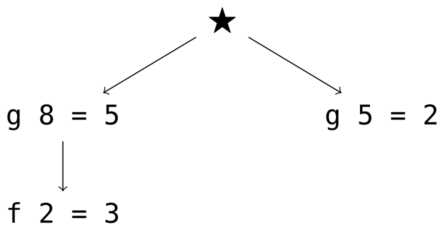
7.3 Constants and their dependencies

Sharing makes keeping track of all computations that depend on the value of a constant difficult: the techniques from Chapter 5 and Chapter 6 only record the first dependency. This problem also arises when tracing functions, consider for example the program of Listing 29 with the constants `main` and `k`.

Listing 29 Program with re-used constant.

```
f = observe "f" λ x . x+1
g = observe "g" λ y . y-k
k = f 2
main = g (g 8)
```

Function `f` is only applied the first time `k` is demanded, then `k` on the heap refers no longer to `f 3` but to 4. Using my method gives the following computation tree:



This tree is not sound for algorithmic debugging: assume function `f` is faulty, then as a result statement `g 5 = 2` may be wrong while `g` is right. But `g 5 = 2` does not have any dependencies and an algorithmic debugger could incorrectly conclude that function `g` is to blame!

No sharing The missing-dependency problem can be avoided by preventing constants to be shared. Sharing can for example be prevented with a program transformation that introduces an unused argument. The program of Listing 29 becomes:

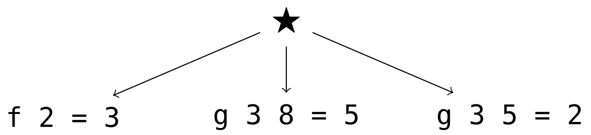
```
f = observe "f" λ x . x+1
g = observe "g" λ y . y - (k undefined)
k z = f 2
main = g (g 8)
```

A drawback of this approach is that preventing sharing impacts performance, and that the approach requires transformation of any constant that is suspected or that (indirectly) depends on the application of a suspected function.

Sharing through arguments To handle constants Nilsson (1998) suggests a program transformation that eliminates constants that are free in a function definition by adding arguments to the function. For example, the transformed program of Listing 29 would be:

```
f = observe "f" λ x . x+1
g = observe "g" λ c y . y-c
k = f 2
main = g k (g k 8)
```

Computing `main` then gives us the computation tree:



The transformation can be automated, and the statement `g 3 8 = 5` presented to the user as “`g 8 = 5` under the assumption `k = 3`".

A drawback of this approach that it requires transformation of any constant that is suspected or that (indirectly) depends on the application of a suspected function.

Tracking use Another option is to track the use of constants through annotations. This approach requires any constant that (indirectly) applies a suspected function to be annotate with `observe`. To track the use of a constant we introduce a new expression:

$$\texttt{use}\ l\ e$$

The program of Listing 29 with `use`-annotation is:

```
f = observe "f" λ x . x+1
g = observe "g" λ y . y-(use "k" k)
k = observe "k" (f 2)
main = g (g 8)
```

Evaluating a `use` $l\ e$ expression adds an event i : `Use l` to the trace.

$$\frac{\Gamma_1, \mathcal{T}_1 \lessdot (i : \texttt{Use}\ \ l) : e \Downarrow \Gamma_2, \mathcal{T}_2 : v \qquad \boldsymbol{i = |\mathcal{T}_1|}}{\Gamma_1, \mathcal{T}_1 : \texttt{use}\ \ l\ e \Downarrow \Gamma_2, \mathcal{T}_2 : v}\ \text{Use}$$

When processing the trace with the method of Chapter 6 computation statement `m` of a `Use`-event is determined by label l (not by the usual reversal of parent-pointers). Then, if the current statement `n` does not yet depend on `m` a dependency is added. Unlike an event that indicates the start of a positive span statement `n` stays the current node.

```
| isUse e if isChildOf m n tree
            then mkCompTree trc n tree
            else mkCompTree trc n (mkChild m n tree)
```

In most cases in practice constants do not depend on suspected functions and then nothing needs to be done. Future work may explore which method is best to be sure that the computation tree is correct.

7.4 Soundness

I 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 faulty node in the tree (Shapiro 1983; Naish 1997b).

However, does my two methods, described in the previous chapters, allow me to construct a computation tree? In other words, can we trust, given a computation tree constructed with either method, and judgements of the computation statements in the tree, that the following property holds: *if a node is faulty, then the node is associated with a program slice that is really defective.*

A proof would be a major undertaking, because in the preceding sections I formally defined only central parts of my method and even the relatively small prototype implementation is quite complex. Hence instead of a proof I test, following Marlow (2012), who verified properties of the stack operations $\lhd$ and $\bowtie$ with the property-based testing tool QuickCheck (Claessen and Hughes 2000a). Already during development of my idea thorough testing proved useful: it uncovered several mistakes in earlier versions.

7.4.1 How to test an algorithmic debugger

Tools for property-based testing search for test cases where expected properties fail. The library QuickCheck allows the programmer to define a property as an executable function and checks for a number of randomly-generated inputs that the property expressed holds.

I can easily generate a random program expression. I can also evaluate such an expression with a Haskell implementation of my semantics to obtain a trace.[1] However, to test with a QuickCheck property that my method indeed constructs a computation tree for algorithmic debugging, I need to clarify three tasks:

1. I need to declare some randomly chosen slices to be defective, so that I know where the defects are and can compare with what my 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, I need to judge automatically whether a given computation statement is intended or not. Normally a human user does this interactively.

I implement these three related tasks with a Boolean type. To emphasise the purpose, I write correctness values as ✔ and ✘ instead of true and false, respectively

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

7.4.2 Defective slices and infection during evaluation

To mark a program slice as defective, and to infect other parts of the program during evaluation I 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 I 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.

In a generated program the `infect`-expression occurs as the immediate nested expression of a defective slice, and only there. For example in

`push` “*outer*” $(\lambda x.$`push` “*inner*” $($`infect` $x)\ )$

the slice labelled “outer” is a working slice, but that labelled “inner” is a defective slice.

To keep track of which values are infected during evaluation I annotate every data value, that is, saturated application of a data constructor, with a flag b stating its correctness. In a generated program the flag of all constructors are annotated with ✔, during evaluation `infect` changes ✔ into ✘. For example the expression

`let` $f =$ `push` “*f*” $($`infect` $(\lambda x.x)); y = f\ 3^{\text{✔}}$
`in push` “*just*” $(\lambda x.$`Just`$^{\text{✔}}\ x)\ y$

evaluates to

`Just`$^{\text{✔}}\ 3^{\text{✘}}$

A functional value is not annotated with a flag, instead further infection is caused by applying `infect` to the body of the abstraction. For example the definition

`infect` $(\lambda x.x)$

of the function f in above program evaluates to

$\lambda x.$`infect` x

Applying a defective function returns an infected result value irrespective of the argument. 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.

My mechanism with annotated data constructors and `infect`-expressions is designed to not change order of evaluation in any way. Applying a defective function still forces evaluation of the argument, if that argument was demanded in the standard semantics.

Figures 50 and 51 show how I modified the syntax of the language and trace events. Figure 52 lists the most important rules of the altered semantics. I add rules for evaluating the pseudo-function `infect`. 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 I assume the presence of a type system that prevents such a defect. All semantic rules handling data constructors need to be altered, but the most interesting one is Case: Only if the inspected data constructor is wrong, then the computation continues with an infected value. 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.

7.4.3 Judging computation statements

From the trace I 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:

let {`i` $= 4;$ `dbl` $=$ `push` “*dbl*” $(\lambda x.x)\}$
`push` “*main*” (`dbl` `i`)

and assume that the function `dbl` is intended to double its argument. With my 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:

let {`i` $= 4$✔; `dbl` $=$ (`push` “*dbl*” ✘ $(\lambda x.x)\}$
`push` “*main*” ✔ (`dbl` `i`)

expression e	
$::= v$	value
$\mid e\ x$	application
$\mid$ `let` $\{x_k = e_k\}_{k=1}^{n}$ `in` e	recursive binding
$\mid$ `case` e `of` $\{c_k\ x_1 \ldots x_{m_k} \to e_k\}_{k=1}^{n}$	case
$\mid x$	variable
$\mid$ `push` $f\ e$	push label and observe expression
$\mid$ `obs` $p\ e$	observed expression
$\mid$ **`infect`** $\boldsymbol{e}$	**infected expression**
value v	
$::= v_\lambda$	functional value
$\mid c^{\boldsymbol{b}}\ x_1 \ldots x_n$	saturated application of data constructor
correctness $\boldsymbol{b}$	
::= ✔	**right (true)**
| ✘	**wrong (false)**

Figure 50: Altered program syntax for testing.

event	t	$::=$	i : Root $f\ \mathcal{S}$	root with label f and cost centre stack $\mathcal{S}$
		|	i : Con $\boldsymbol{b}\ p\ c\ a$	value is application of constructor c
		|	i : Lam p	value is an abstraction
		|	i : MapsTo p	function application

Figure 51: Extended trace event for data constructor application with correctness field.

$$\frac{\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{T}_2 : \lambda x.e_2}{\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : \texttt{infect}\ e \Downarrow \Gamma_2, \mathcal{T}_2 : \lambda x.\texttt{infect}\ e_2}\ \text{InfLam}$$

$$\frac{\Gamma_1, \mathcal{S}_1 \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{S}_2, \mathcal{T}_2 : c^b\ x_1 \ldots x_n}{\begin{array}{c}\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : \texttt{infect}\ e \Downarrow \Gamma_2[y_1 = \texttt{infect}\ x_1, \ldots, y_n = \texttt{infect}\ x_n], \\ \mathcal{S}_2, \mathcal{T}_2 : c^{\text{✘}}\ y_1 \ldots y_n\end{array}}\ \text{InfCon}$$

$$\frac{\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{S}_2, \mathcal{T}_2 : c_k^{\text{✘}}\ x_1 \ldots x_{m_k} \qquad \begin{array}{c}\Gamma_2, \mathcal{S}_2, \mathcal{T}_2 : \texttt{infect}\ (e_k[x_i/y_i]_{i=1}^{m_k}) \\ \Downarrow \Gamma_3, \mathcal{S}_3, \mathcal{T}_3 : v\end{array}}{\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \Downarrow \Gamma_3, \mathcal{S}_3, \mathcal{T}_3 : v}\ \text{InfCase}$$

$$\frac{\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{S}_2, \mathcal{T}_2 : c_k^{\text{✔}}\ x_1 \ldots x_{m_k} \qquad \begin{array}{c}\Gamma_2, \mathcal{S}_2, \mathcal{T}_2 : e_k[x_i/y_i]_{i=1}^{m_k} \\ \Downarrow \Gamma_3, \mathcal{S}_3, \mathcal{T}_3 : v\end{array}}{\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^{n} \Downarrow \Gamma_3, \mathcal{S}_3, \mathcal{T}_3 : v}\ \text{NoInfCase}$$

$$\frac{\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{S}_2, \mathcal{T}_2 : c^b\ x_1 \ldots x_n \qquad i = |\mathcal{T}_2|}{\begin{array}{c}\Gamma_1, \mathcal{S}_1, \mathcal{T}_1 : \texttt{obs}\ p\ e \Downarrow \Gamma_2[y_1 \mapsto \texttt{obs}\ (\mathrm{P_c}\ i\ 1)\ x_1, \ldots, y_n \mapsto \texttt{obs}\ (\mathrm{P_c}\ i\ n)\ x_n], \\ \mathcal{S}_2, \mathcal{T}_2 \lessdot (i : \mathrm{Con}\ b\ p\ c\ (\mathrm{arity}\ c^b)) : c^b\ y_1 \ldots y_n\end{array}}\ \text{ObsCon}$$

Figure 52: New and modified rules for testing the tracing semantics.

For testing the judgement is based on the infected and normal values appearing in the computation statement. Figure 53 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 I conservatively judge the function to meet intentions. A data structure is wrong if any part of it is wrong. Unevaluated parts are right. The judgements for the computation tree of my example are:

$$\begin{array}{ll}\llparenthesis \texttt{main = } 4^{\text{✘}} \rrparenthesis & = \text{✘} \\ \llparenthesis \texttt{dbl } 4^{\text{✔}} \texttt{ = } 4^{\text{✘}} \rrparenthesis & = \text{✘}\end{array}$$

which correctly identifies the defective slice.

Computation statements for higher-order functions are often hard to judge. A benefit of my 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:

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

$$\begin{aligned}
\llparenthesis f \texttt{ = } a \rrparenthesis &= \llparenthesis a \rrparenthesis \\
\llparenthesis w_1 \to w_2 \rrparenthesis &= \llparenthesis w_1 \rrparenthesis \Rightarrow \llparenthesis w_2 \rrparenthesis \\
\llparenthesis c^b\ w_1 \ldots w_n \rrparenthesis &= b \land \llparenthesis w_1 \rrparenthesis \land \ldots \land \llparenthesis w_n \rrparenthesis \\
\llparenthesis \{a_1, \ldots, a_k\} \rrparenthesis &= \llparenthesis a_1 \rrparenthesis \land \ldots \land \llparenthesis a_k \rrparenthesis \\
\llparenthesis \_ \rrparenthesis &= \text{✔}
\end{aligned}$$

Figure 53: Judgement of a computation statement.

Next I create an equivalent program for testing and mark h as defective. I obtain the following three computation statements with their automatic judgements:

$$\begin{aligned}
&\llparenthesis \texttt{main} \ = 2^{\text{✘}} \rrparenthesis && = \text{✘} \\
&\llparenthesis \texttt{h \{\textbackslash}2^{\text{✔}} \to 2^{\text{✔}}\texttt{\}}\ 2^{\text{✔}} \texttt{ = } 2^{\text{✘}} \rrparenthesis && = \text{✘} \\
&\llparenthesis \texttt{f } 2^{\text{✔}} \texttt{ = } 2^{\text{✔}} \rrparenthesis && = \text{✔}
\end{aligned}$$

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 I obtain are:

$$\begin{aligned}
&\llparenthesis \texttt{main = } 2^{\text{✘}} \rrparenthesis && = \text{✘} \\
&\llparenthesis \texttt{h \{\textbackslash}2^{\text{✔}} \to 2^{\text{✘}}\texttt{\}}\ 2^{\text{✔}} \texttt{ = } 2^{\text{✔}} \rrparenthesis && = \text{✔} \\
&\llparenthesis \texttt{f } 2^{\text{✔}} \texttt{ = } 2^{\text{✘}} \rrparenthesis && = \text{✘}
\end{aligned}$$

which correctly identifies f as defective.

7.4.4 Restrictions on observation annotations

I discussed that a program with a constant that is used more than once, or with a free variable is challening for algorithmic debugging. Here I test debugging a program where a labelled expression has no free variable other than a global function. 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 I cannot allow free variables at all. The following expression can be used to test my method:

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

To formulate my test-property I assume there is function `noFreeVars` that applied to a program returns `True` when the program does not contain any variables that are free outside an annotated program slice, and `False` otherwise.

Listing 30 shows the form of an annotated program that guarantees the generation of a computation tree suitable for algorithmic debugging. My example program of Listing 19 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.

Listing 30 General annotated program.

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

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 `let`s nested 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 (also see discussion of free variables in Section 7.2).

The restriction to top-level definitions limits the precision of my algorithmic debugger. If a local function is defective, only the surrounding top-level function will be identified as defective. In my example I 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 I 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 3.3 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.

I discuss approaches for handling constants in Section 7.3, but with my base method I 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, I 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. My examples obey these restrictions and so do my case studies in Chapter 9. To define test properties in this and the next chapter I assume there is a function `obeysRestrictions` that applied to a program e evaluates to `True` when e is of the form of Listing 30 and `False` otherwise.

7.4.5 Testing computation tree construction with trace stacks

I implemented the modified semantics for testing in Haskell.

The semantics may get "stuck" when evaluating a program, for example when trying to apply an already fully satisfied data constructor to a variable:

```
let {f = 3; y = 5} in f y
```

Other programs may never terminate, such as:

```
let {f = λ x . f x; y = 4} in f y
```

The function `wellFormed` applied to a program returns `False` when the programs gets "stuck" or when the program does not terminate after 5000 reduction steps, and `True` otherwise.

The function `labels` applied to a node evaluates to the list of labels the node is associated with (a computation statement is associated with just one label, but after cycle removal a node in the tree may contain multiple computation statements.)

The function `algoDebug` evaluates the random expression `e`, constructs a computation tree and uses the algorithmic debugging method to produce a set of labels of defective program slices. The function `defects` returns a set of all slices in e that contain the pseudo-function `infect`. Algorithmic debugging does not guarantee finding all defects and some program parts may not even be evaluated, but the set of labels found with algorithmic debugging should be a subset of the set of labels of defective slices.

With these functions I define the following QuickCheck property in pseudo-Haskell. The property holds for 100000 randomly generated programs. Programs for which the precondition does not hold are not counted.

Property 7.1. *For all well-formed programs obeying the restrictions: for every node n that algorithmic debugging finds to be faulty, there exists at least one identifier f associated with n such that f is the identifier of a program slice that is actually faulty. This is defined in the following test-property:*

```
prop_stack_sound e =
 wellFormed e && obeysRestrictions e ⇒ all anyDefective (algoDebug t)
 where
 anyDefective n = any (λ f . f ⊆ defects e) (labels n)
 t = mkCompTree 𝒯
 {}, ⟨⟩  :  e   ⇓   Γ, 𝒯  :  v
```

7.4.6 Testing pure computation tree construction

The method of chapter 6 and my 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. The restrictions of Section 7.4.4 also apply here.

To verify the complete implementation of tracing, tree construction and algorithmic debugging I checked that constructed 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 (Shapiro 1983; Naish 1997a).

Property 7.2. *For all well-formed programs obeying the restrictions: A labelled expression that algorithmic debugging finds faulty actually contains a defect. This is defined in the following test-property:*

```
prop_pure_sound e =
 wellFormed e && obeysRestrictions e ⇒ algoDebug t ⊆ defects e
 where
 t = mkCompTree 𝒯
 {}, ⟨⟩  :  e   ⇓   Γ, 𝒯  :  v
```

8 Properties as oracle for algorithmic debugging

An algorithmic debugger finds defects in programs by systematic search. It relies on the programmer to direct the search by answering a series of yes/no questions about the correctness of specific function applications and their results. If the size or number of questions is too great for the programmer to handle, the technique breaks down.

In this chapter I propose a method that re-uses properties, already present in the code for testing, to answer automatically some of the questions arising during algorithmic debugging, and to replace others by simpler questions.

Judging computation statements by an automated oracle has often been suggested (Shapiro 1983; Drabent and Nadjm-Tehrani 1989). Such an oracle might be based on a previous working version of the program, or a formal specification. But what if these things are not available?

Automatic property-based testing, using QuickCheck or a similar tool, is now a widespread practice of developers using functional languages. Many programs already have associated test properties defined, and questions asked by the algorithmic debugger can inspire the programmer to add new test properties. However, not all properties are equally suitable to be used as oracle. Consider for example the following property:

```
prop_odd n = n >= 0 ==> odd (2*n+1) == True
```

The expression `prop_odd 0` evaluates to `False` from which we incorrectly might conclude that the statement `odd 0 = False` is wrong. So we reach the motivating observation and question for this chapter: *There must surely be ways we can use test properties to help judge computation statements, but how?*

Properties can be used to encode a specification or for testing against a reference implementation. In this chapter I present requirements and techniques to draw sound conclusions when using test properties to provide an oracle. For judging a computation

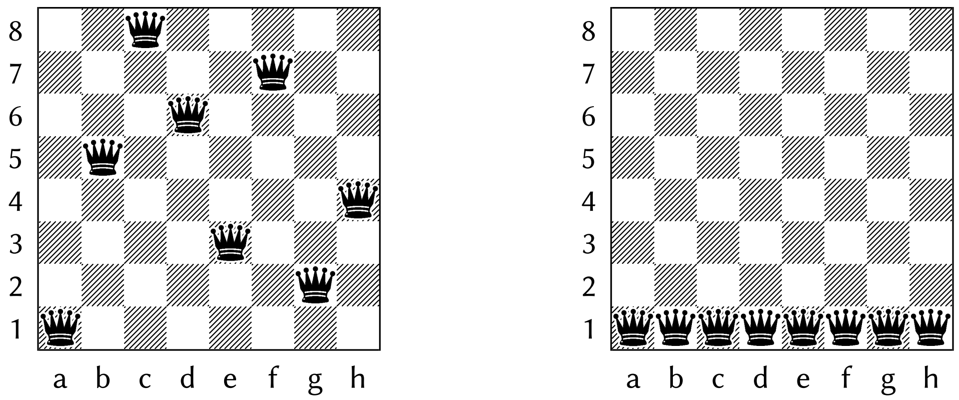


Figure 54: A correct 8-queens solution, and an incorrect one!

statement for function f it is not required that the test property has f at top level, or that the property fully specifies f.

8.1 Defective example program with properties

Consider the defective program in Listing 31. The program is intended to find all solutions to the n-queens problem: place n queens on an $n \times n$ board in such a way that no queen threatens any other (by occupying the same row, column or diagonal). For examples of correct and incorrect solutions when $n = 8$, see Figure 54.

The program represents a board with queens on it by listing row-numbers at which a queen is placed in successive columns

```
type Board = [Int]
```

The correct solution in Figure 54 is represented by `[1,5,8,6,3,7,2,4]`.

The first solution in the list that `queens 8` evaluates to however, is an incorrect solution, with all queens in the bottom row. An algorithmic debugger locates the defect after 15 answers from the human programmer (in *italics*):

```
1: queens 8 = (1 : 1 : 1 : 1 : 1 : 1 : 1 : 1 : []) : _ ? wrong
2: valid 8 8 = (1 : 1 : 1 : 1 : 1 : 1 : 1 : 1 : []) : _ ? wrong
3: valid 7 8 = (1 : 1 : 1 : 1 : 1 : 1 : 1 : []) : _ ? wrong
4: valid 6 8 = (1 : 1 : 1 : 1 : 1 : 1 : []) : _ ? wrong
```

```
 5: valid 5 8 = (1 : 1 : 1 : 1 : 1 : []) : _ ? wrong
 6: valid 4 8 = (1 : 1 : 1 : 1 : []) : _ ? wrong
 7: valid 3 8 = (1 : 1 : 1 : []) : _ ? wrong
 8: valid 2 8 = (1 : 1 : []) : _ ? wrong
 9: valid 1 8 = (1 : []) : _ ? right
10: valid 2 8 = (1 : 1 : []) : _ ? wrong
11: extend 8 ((1 : []) : _) = (1 : 1 : []) : _ ? right
12: valid 2 8 = (1 : 1 : []) : _ ? wrong
13: safe (1 : 1 : []) = True ? wrong
14: no_threat 1 (1 : []) 1 = True ? wrong
15: no_threat _ [] _ = True ? right
Fault located in no_threat applied to 1 (1 : []) 1!
```

A human programmer judges the statements from question 2-8, 10 and 12 as wrong because a valid `Board` has no more than one queen on the same row. In other words, a valid `Board` is a set. With my method, the programmer can give as response to question 2 the property

```
prop_valid_set m n = all (\b -> isSet b) (valid m n)
  where
  isSet xs = all (\ x -> length (filter (==x) xs) == 1) xs
```

The algorithmic debugger automatically uses this property to answer question 3-8, 10 and 12. For question 9 the algorithmic debugger can inform the programmer that `prop_valid_set` holds for `valid 1 8`, however because `prop_valid_set` is not a *full* specification of `valid` the algorithmic debugger has to ask the human programmer to confirm. In Section 9.2 I document an in-depth case study of the defective n-queens solver in which my algorithmic debugger locates the the defect using test properties without consulting a human oracle.

8.2 Dynamic and static dependencies

Consider the defective Haskell implementation of the parity functions `even` and `odd` in Listing 32. The program includes two test properties specifying the intended behaviour of these functions. Using the property-based testing tool QuickCheck (Claessen and Hughes 2000b) we detect that the program is defective:

```
> quickCheck spec_even
```

Listing 31 A defective solver of the n-queens problem.

```
queens :: Int -> [Board]
queens n = valid n n

valid :: Int -> Int -> [Board]
```

Repeatedly add a queen and select valid boards

```
valid 0 _ = [[]]
valid m n = filter safe (extend n (valid (m-1) n))

extend :: Int -> [Board] -> [Board]
extend n bs = consEach [1..n] bs
```

Possible ways of placing a queen on new left column

```
consEach :: [a] -> [[a]] -> [[a]]
consEach []    _ = []
consEach (a:x) y = map (a:) y ++ consEach x y

safe :: Board -> Bool
safe (a:b) = no_threat a b 1
```

Is queen in left column no threat to any other?

```
no_threat :: Int -> Board -> Int -> Bool
no_threat a []    m = True
no_threat a (b:y) m =
  a+m /= b && a-m /= b && no_threat a y (m+1)
```

Defect: detects diagonal but not horizontal threat

```
*** Failed! Falsifiable: 2
> quickCheck spec_odd
*** Failed! Falsifiable: 1
```

So property spec_even does not hold for the value 2 and property spec_odd does not hold for the value 1.

Listing 32 A defective program with parity tests for positive integers, and properties that specify the functions in the program.

```
even 0 = True
even n | n > 0 = odd (n-1)

odd 0 = False                 | Defect: should
odd n | n > 0 = even n      ←| be “even (n-1)"

spec_even n = n >= 0 ==> even n == (n ‘mod‘ 2 == 0)
spec_odd  n = n >= 0 ==> odd n == (n ‘mod‘ 2 == 1)
```

Even though we have for every function in the program a property that fully specifies that function, testing alone does not *locate* the defect in the code. To locate defects discovered by testing, we could perhaps try to sort properties by call-dependencies. But such analysis may not help if there are mutual recursive functions such as even and odd. Furthermore, in a language with higher-order functions, call dependencies may not be known statically (Nilsson 1998, p.105).

To *locate* the defect we could use an algorithmic debugger. The interaction between an algorithmic debugger and an oracle (whose responses are shown in *italics*) could be:

Q1: even 2 = False ? *wrong*
Q2: odd 1 = False ? *wrong*
Q3: even 1 = False ? *right*
Defect located in odd.

In the small example above, the potential usefulness of the test properties is clear:

spec_even 2 evaluates to False (cf. answer Q1),
spec_odd 1 evaluates to False (cf. answer Q2), and
spec_even 1 evaluates to True (cf. answer Q3).

8.3 Testing properties to judge statements

Tools for property-based testing search for test cases where expected properties fail. Test properties are defined as functions with one or more test-value arguments and a boolean result. (Conditional properties defined for QuickCheck (Claessen and Hughes 2000b) instead have the result type `Property` which translates to `True`/`False` or an inconclusive outcome.) The intended interpretation is that the body of the function is a universally quantified assertion: the property should be true *for all* choices of arguments. Libraries for property-based testing check for counter-examples by evaluating properties with many different arguments, generated either at random or in some systematic sequence.

Given an f-statement S, I aim to use tests of already-defined properties involving f to judge S — or to assist in that task. In doing so, I assume that each such property correctly expresses a partial specification of f.

8.3.1 Judging statements wrong

I set out here sufficient further requirements on a property p, and the test cases to which p is applied, to ensure that if a test fails we may safely mark an f-statement S wrong. If S is

$$f\ v_1 \ldots v_n = v_r$$

and p is

$$\forall \bar{x}\ .\ e[f\ \bar{a}] \qquad \text{(Requirement 1)}$$

and

$$\mathrm{dom}(\theta) = \bar{x} \qquad \theta(\bar{a}) = v_1 \ldots v_n \qquad \text{(Requirement 2)}$$

and

$$eval(\theta(e)[v_r]) = \texttt{False} \qquad \text{(Requirement 3)}$$

then S is wrong.

Requirement 1: unique suspicion *All functions other than f applied in p can be trusted as correct.*

Explanation: Otherwise, if a test of p fails, the fault may be incorrect behaviour of a function other than f. A distinction between trusted and suspect functions is commonly made in algorithmic debugging to reduce the size of computation trees — only statements about suspect functions are recorded. Prelude and library functions are trusted. Functions defined as properties, or only as auxiliaries of properties, are trusted. Programmers may also declare trust in specified application modules or functions.

Can this requirement be relaxed? Properties involving multiple suspect functions may still be useful. If testing fails, we know at least one of these suspect functions is faulty. If the failure is simple it may reveal a faulty computation tree simpler than the one currently being considered.

Requirement 2: domain restriction *For any application of f in p each argument is either exactly as in S or it is one of p's argument variables.*

Explanation: By fixing those test-value arguments of p that are also arguments of f, we can ensure that in any test f is only applied to the argument values recorded in S. If a test of p involved an application of f to other arguments, and the test failed, we might be able to conclude that f is faulty, but not that S is wrong. A bug search should not home in on a faultless subtree!

Requirement 3: range restriction *The result of the application of f in p is not demanded to any greater extent than recorded in S.*

Explanation: If p demands the result of f to a greater extent than S records, and a test fails, here too we could not safely mark S wrong. This requirement is hardly amenable to syntactic checking; instead we apply a constraining wrapper to f, raising an exception if its result is demanded to an unsafe extent — see Section 8.4.

Can this requirement be relaxed? What if property testing gives an undefined result in every case because of this restriction, though without it there is a well-defined outcome for one or more properties and test-cases? We consider in Section 8.5 how such unrestricted test results might be used. See also Section 8.6 and Section 8.7 for techniques to avoid undefined test results.

8.3.2 Judging statements right

Most properties are only *partial* specifications of the functions they involve. Also, unless Requirement 2 demands that *all* arguments of a property p are fixed in testing, p remains a universal property that is only checked for a limited set of test values. For both of these reasons, although an f-statement S can be judged wrong if some property p meeting Requirements 1–3 fails when tested, S cannot in general be judged right if testing p succeeds.

Option: declared specification *Programmers may tag a property, or a group of properties, as a full specification of f.*

Explanation: Such tagging may enable statements to be judged right by property testing, or right in all cases tested — see Section 8.3.3. In practice, I tag specifications by a naming convention, introducing properties with names of the form `spec_f`.

What if this option is declined? In the tree for a faulty computation, there is a path of wrong statements from the root to an f-statement for some defective function f. Even without any properties providing a full specification, in the best case, automatic judgement could find that path, assuming only that properties are partial specifications. However, to confirm that f is indeed defective, we must somehow judge that all children of the f-statement are right — using properties known to be full specifications, or relying on a human oracle, or by some other means.

So the benefits are greater if we have full specifications. But even without them, we may hope to judge faulty statements as wrong without troubling the programmer. Though without full specification right statements cannot be verified with certainty, we might assist the user by pointing out associated properties that have been checked.

Option: trusted testing *Programmers may choose to accept as verified a universal property that holds in all test cases evaluated.*

Explanation: The risks attached to this choice depend on the quality of the test-value generator, and the resources used for testing. Often it may speed up the progress of algorithmic debugging towards a fault elsewhere, but sometimes it may turn attention away from the actual location of a fault.

What if this option is declined? The programmer can be advised of the outcome of testing in such cases, but they must act as oracle.

8.3.3 Summary of valid conclusions from property tests

Beyond the requirements of Section 8.3.1, I note two important characteristics that each property p may or may not have, affecting its use to judge an f-statement:

- Is p only a partial specification of f, or a complete one?
- Does p have additional arguments, beyond those used in its application of f, for which test values must be generated?

Depending on these distinctions, and drawing upon the observations of Section 8.3.1 and Section 8.3.2, we have the following rules for drawing conclusions about an f-statement S from the results of testing p.

1. If p is a complete specification of f, without any additional arguments, and the result of testing p is `True`, we may conclude that S is right.

2. If p is a complete specification of a f, but with additional arguments, and for each of a randomly generated set $\bar{t}$ of test-values for these arguments the result of testing p is `True`, we may draw the qualified conclusion that so far as tests using $\bar{t}$ reveal S is right.

3. If p is a complete or partial specification of f, without any additional arguments, and the result of testing p is `False`, we may conclude that S is wrong.

4. If p is a complete or partial specification of a f, but with additional arguments, and for at least one assignment of test-values to these arguments the result of testing p is `False`, we may conclude that S is wrong.

Note also the cases where *no* firm conclusion can be drawn. If p is only a partial specification of f, and testing p gives only `True` results, then regardless of whether p has additional arguments we cannot judge the f-statement S — we can only report the property-test results as advice to a human oracle. There is also the possibility, whatever the characteristics of p, that we obtain only undefined results from testing.

We obtain an undefined result $\perp$ when evaluating p times out, when p evaluates to an exception, when a precondition in p is not met or when evaluating p requires untrusted computation that are not recorded in the computation statement (e.g. an unevaluated value in the argument is demanded).

Figure 55 sets out all cases of the above summary in diagrammatic form.

Often there will be more than one property that meets the requirements to be tested with the aim of judging a statement. Though testing one property may lead only to an undefined or inconclusive outcome, another may provide a conclusive judgement.

8.4 Restricting results of subject functions

To meet Requirement 1 in Section 8.3.1, we must limit the demand made on the result of subject functions when properties are tested.

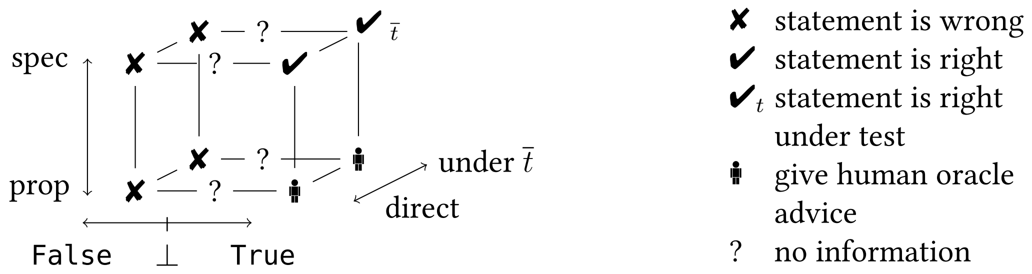


Figure 55: Which conclusion can we draw?

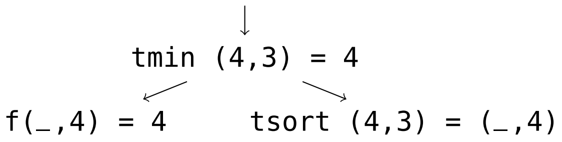
Listing 33 Defective minimum of tuple and a property.

```
tmin t = f (tsort t) ⟵ First in ordered tuple is minimum
tsort (x,y) = if x > y then (s (x,y), x) else (x,y)
f (x,y) = y     ⟵ Defect causing observed faulty behaviour
s (x,y) = x     ⟵ Defect that may cause faulty be-
                   haviour in other cases
```

Constrained applications Consider the program in Listing 33. The property

```
prop_tsort_compl (x,y) =
 x ‘telem‘ (tsort (x,y)) && y ‘telem‘ (tsort (x,y))
 where telem x (y,z) = x == y || x == z
```

seems suitable for judging a `tsort`-statement wrong when the property evaluates to `False`. Evaluation of `tmin (4,3)` gives the following computation tree:



Statement `tsort (4,3) = (_,4)` is right (more on how an oracle should interpret unevaluated expressions in Section 8.7). However, `prop_tsort_compl (4,3)` evaluates to `False` because it forces the whole result (and the computation `s (4,3) = 4` that is not part of the original computation tree), and detects that 3 is not in the result (4,4) of `tsort`. To soundly use a property as oracle we need to constrain the result of a given function by a given partial result.

To do so, I define a type-generic operator `conAp` such that with

```
tsort' = conAp tsort (⊥,4)
```

where $\perp$ is an expression that always evaluates to an exception, the program

```
let (x,y) = tsort' (4,3) in print y
```

prints the value 4, but the program

```
let (x,y) = tsort' (4,3) in print x
```

raises an exception.

To introduce constrained application in a property, we first abstract the subject function to become a property argument — a mechanical modification that can easily be automated. For example, with the property definition

```
prop_tsort_compl' subjFn (x,y) =
 x ‘telem‘ (subjFn (x,y)) && y ‘telem‘ (subjFn (x,y))
 where telem x (y,z) = x == y || x == z
```

the expression

```
prop_tsort_compl' (conAp tsort (⊥,4)) (4,3)
```

would raise an exception signalling an inconclusive result.

Constrained values It remains to implement a mechanism to constrain a value to another value, that is ensure a value is equal-to or a subvalue of another value. For this purpose I define a class of constrainable types

```
class Constrainable t where
  constrain x c ::  t -> t -> t
```

in order that we may simply define `conAp` as follows.

```
conAp :: Constrainable b => (a ->b) -> b -> a -> b
conAp f r x = constrain (f x) r
```

I give a type-generic definition of the `constrain` method. I define the default instance of the `constrain` method as

```
 default constrain x c =
     to (gconstrain (from x) (from c))
```

To enable automatically derived `Constrainable` instances, I provide a generic definition of `gconstrain` for the sum-of-products representation just as I did in Chapter **??** for `observer`.

$$
\begin{aligned}
&\mathtt{gconstrain}\ (\mathrm{L1}\ x)\ (\mathrm{L1}\ c) = \mathrm{L1}\ (\mathtt{gconstrain}\ x\ c)\\
&\mathtt{gconstrain}\ (\mathrm{R1}\ x)\ (\mathrm{R1}\ c) = \mathrm{R1}\ (\mathtt{gconstrain}\ x\ c)\\
&\mathtt{gconstrain}\ (x_1 :\!*\!: x_2)\ (c_1 :\!*\!: c_2) =\\
&\quad (\mathtt{gconstrain}\ x_1\ c_1) :\!*\!: (\mathtt{gconstrain}\ x_2\ c_2)\\
&\mathtt{gconstrain}\ x\ c = \bot
\end{aligned}
$$

Base types such as `Int` and `Char` are atomic, that is, these types cannot be divided into smaller parts and are evaluated as a whole. For these I use equality:

```
bconstrain x c | x == c = x
               | x /= c = ⊥

instance Constrainable Int
  where constrain = bconstrain
instance Constrainable Char
  where constrain = bconstrain
```

For function types I provide the instance:

```
instance Constrainable b => Constrainable (a->b)
  where
  constrain f g x = constrain (f x) (g x)
```

8.5 Testing with an unrestricted subject function

Having a restricted subject function allows us to judge a statement wrong when a test property evaluates to `False`. But what if restriction makes a test result undefined, though the same test with an unrestricted subject function evaluates to `False`?

We can record a second computation tree during the evaluation of a property. This tree has at its root an application and result of the subject function currently under investigation. This new computation tree might be simpler than the current computation tree. So algorithmic debugging with the new tree might be less work than continuing

```
treeComplexity tree = sum (map length  (unjudgedStmts tree))

unevalCount tree = sum (map (length . (filter (==’_’)))
                            (unjudgedStmts tree))
```

Figure 56: Basic method for estimating complexity of a computation tree.

with the current one. If a user opts to switch to a different computation tree, they are still guaranteed to find a defect in the program, but it may not be the defect that caused the originally investigated computation to go wrong.

To decide whether or not to switch the focus of debugging to an alternative computation tree, we need an appropriate way to compare trees. The simple algorithm `treeComplexity` of Figure 56 expresses the complexity of a tree as the combined size of all its unjudged statements (i.e. statements that are neither marked as right or wrong).

Unevaluated expressions can result in an inconclusive judgement from properties as oracle. A refinement is to keep an additional separate count of _ occurrences. The algorithm `unevalCount` in Figure 56 assigns an equal weight to each unevaluated expression, a possible variation assigns a weight based on the combined sizes of type formulae. For example, an unevaluated expression of type `Int` would be less complex than an unevaluated expression of type `(Int,Int)`.

Listing 34 A defective definition of `idMatrix` and a property specifying that it should be equivalent to a reference implementation `idMatrix_ref`.

```
data IntMatrix = M [[Int]]

idMatrix :: Int -> IntMatrix
idMatrix n = M (take n (iterate
 (\(x:xs) -> xs ++ [x]) row0))
 where row0 = 1 : take (n-1) (repeat 0)

spec_idMatrix im n = im n == idMatrix_ref n
idMatrix_ref 1 = M [[1]]
idMatrix_ref n = M (row0 : map (0:) m)
 where row0 = 1 : take (n-1) (repeat 0)
       M m  = idMatrix_ref (n-1)
```

Defect: rotates next row left instead of right

8.6 Parallel equality

Consider the defective function `idMatrix` for constructing a $n \times n$ identity matrix from Listing 34. I represent a row in the matrix as a list of integers and a matrix as a list of rows. Thus the 3×3 identity matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

is in the example program represented as `[[1,0,0], [0,1,0], [0,0,1]]`.

When debugging a program that includes the `idMatrix` function we may need to judge a statement such as:

```
idMatrix 3 = M (_ : [0,0,1] : _)
```

Could the test property from Listing 34 be used to judge the statement? The second row should have the 1 at the second position, so it cannot equal the second row in `idMatrix_ref 3`. From this observation we may expect the property test to give `False`. However, as the first field of the configuration is unevaluated, comparison with `(==)` fails and the property gives an inconclusive result.

Equality is used very frequently in test properties, many of which take the form of equations or conditional equations. Often the equality test is between data structures containing more information than a subject function uses. As a result the test, and hence the property, evaluate to bottom even when there is an observable difference between two values. To make such properties more useful for judging computation statements, we need an equality test that keeps looking for inequalities even when the result of comparing some components is undefined.

So we wish to define an equality function (`|==|`) such that, for example,

```
⊥ : [3,4]|==| ⊥:  [3,6]   and   7 : ⊥|==| 8 : ⊥
```

both evaluate to `False`.

I define a new class `Pareq`. It has a comparison method (`|==|`) that returns `True` or `False` when (in)equality is determined and $\perp$ otherwise.

```
class Pareq t where
  pareq ::  t -> t -> Bool
  default  x|==| y = (from x) |===| (from y)
```

$$\begin{array}{llll}
(\mathrm{L1}\ x) & \texttt{|===|} & (\mathrm{L1}\ y) & = x\ \texttt{|===|}\ y \\
(\mathrm{R1}\ x) & \texttt{|===|} & (\mathrm{R1}\ y) & = x\ \texttt{|===|}\ y \\
(x_1 :\!*\!: x_2) & \texttt{|===|} & (y_1 :\!*\!: y_2) & = (x_1\ \texttt{|===|}\ y_1)\ \texttt{|\&\&|}\ (x_2\ \texttt{|===|}\ c_2) \\
x :: b & \texttt{|===|} & y :: b \quad \mid b \text{ is a basetype} & = \texttt{x == y} \\
x & \texttt{|===|} & y & = \texttt{False} \\
 & & & \\
\bot & \texttt{|\&\&|} & \texttt{False} & = \texttt{False} \\
p & \texttt{|\&\&|} & q & = \texttt{p \&\& q}
\end{array}$$

Figure 57: Definitions of parallel equality in the sum of products representation of Figure 17, and of parallel conjunction — where the $\bot$ argument pattern is matched by exception handling.

Figure 57 gives the type-generic definition `|===|` making the type class `Pareq` derivable. Its definition is similar to how the sequential comparison method (==) is derived, but replacing sequential conjunction (&&) by parallel conjunction (`|&&|`).

In an actual implementation of parallel conjunction, bottom can be detected by catching exceptions raised when trying to match other defining equations (Runciman, Naylor and Lindblad 2008). We can similarly implement a parallel version of other commonly used operators such as a parallel disjunction and replace sequential logical operators with their parallel equivalent.

To make use of parallel equality in a test property, the programmer, a compiler pass or a specialized transformation tool must ensure two things:

1. In the property `(==)` is replaced by, or redefined as, (`|==|`).

```
spec_idMatrix im n =  im n |==| idMatrix_ref n
```

2. Parallel equality is derived for the type of values compared in the property.

```
data IntMatrix = M [[Int]] deriving (Generic,ParEq)
```

To further reduce dependencies on unevaluated values sequential logic operators in the property are replaced by their parallel equivalent.

8.7 Quantifying unevaluated expressions

In Section 2.2.2 I discussed that we should interpret the computation statement

```
paint (Square Red _) = Paint Red _
```

from the program of Listing 4 as

$\forall x.\exists y.$ `paint (Square Red` x`) = Paint Square` y

So `paint (Square Red _) = Paint Red _` should be judged right. But can we determine this without consulting a human oracle? Although the test property in Listing 35 fully specifies `paint`, testing the property `spec_paint` for the recorded arguments gives an undefined result — even when `paint` is not restricted in its result.

Listing 35 A test property that fully specifies the defective program of Listing 4.

```
spec_paint p (Square c x) =
 p (Square c x) == Paint c (x*x)
spec_paint p (Rect c w h) =
 p (Rect c w h) == Paint c (w*h)
```

As we did for properties with more arguments than the subject function, we can generate values for unevaluated expressions in the recorded argument and obtain a set of completed arguments. Then we test if the property holds for all completed arguments:

```
quickCheck (\x -> spec_paint paint (Square Red x))
```

If this check succeeds, and the property fully specifies the subject function, we have *support for* a judgement that the statement is right. According to the user's preference, we may either judge statements that are supported by tests as (provisionally) right, or we may report the test outcome to the user and leave the judgement to them.

If we find a counter-example for a test property, we cannot judge the current statement to be wrong if we are testing with an unrestricted subject function. However, there is another option. The counter-example may provide a new faulty computation tree involving the same subject function but simpler than the subtree originally under investigation. If so, we can give the user the option of switching to the new tree as the scope for algorithmic debugging (see Section 8.5).

8.8 Two-oracle strategies

Current algorithmic debugging strategies aim to minimize the total number of questions (Section 2.2.3). For property-assisted algorithmic debugging, we could simply adopt one of these existing strategies. We select a statement as usual according to the strategy, and

1. use associated properties as oracle with a restricted subject function,
2. if inconclusive, use associated properties as oracle with an unrestricted subject function,
3. if inconclusive, and an argument of the subject function includes an unevaluated expression, use associated properties as test-case oracles with randomly generate values completing arguments,
4. if inconclusive, consult the human oracle.

However, as we now have two oracles, we might instead ask: how can we reduce the number of questions the human oracle has to answer?

A brute-force strategy sets the lower-bound on the number of statements the human oracle has to answer. The debugger tries the automatic steps 1–3 on *every* computation statement. If a computer with multiple processors is used some time can be saved by judging statements in parallel.

During this process sufficient information may be obtained to locate a defect. If not, we compare the complexities (e.g. as in Figure 56) of all subtrees with a wrong statement at the root, along with any candidate trees found in step 2 or 3, and select the simplest. Then algorithmic debugging using one of the standard strategies can locate the defect in the simplest subtree, asking the human oracle to provide missing judgements. When switching to a newly discovered tree we may use the brute-force strategy again on that tree.

The All-children strategy is a variation on the Top-down strategy that can be used when Brute-force is not feasible. When a computation statement has multiple children (a common case) we apply steps 1–3 to them all. If any of the children is found to be wrong, we repeat the process with the children of that statement. We only consult the human oracle when applying properties as oracle to the children of a wrong statement gives a mixture of right and inconclusive results.

8.9 Implementing property-assisted debugging

I implemented my method in a property-assisted extension of Hoed-pure. After running a program and connecting to the debugging session the user is offered three options:

- Manually judge a statement as right or wrong. Either the defect is now located or the next statement is shown. Which statement is next is determined by either the top-down or the divide-and-query strategy.

- Use properties as oracle for the current statement. If the result is conclusive, either the defect is located or the next statement is shown. If the result is inconclusive, advice is offered to the user who still has the first option.

- Use properties as oracle on many statements according to either the All-children strategy or the Brute-force strategy.

Using properties as oracle is implemented as follows. For each property applicable to a statement, we first generate a program in which the property is applied to a restricted subject function and the recorded argument values. For any unevaluated expression we generate `(error "unevaluated expression")`. This program is compiled, linked with the modules containing the property and the subject function, and run. If we cannot draw a conclusion from the result, we generate and run a further program where the property is applied to the unrestricted subject function. If this too is inconclusive, we substitute a fresh variable for each unevaluated expression and use QuickCheck to generate and run random tests.

Because a property may cause a function to diverge, we interrupt evaluation of the property after a set time limit.

I also use the Hoed-pure tracer to obtain the computation tree for *generated* programs. When a program uses a set of test-values, we disable the generation of the tree. If a counter-example is found, we re-run the program for the counter-example only, this time generating the computation tree.

8.10 Summary

I presented a new semi-automated method for defect location in functional programs, based on algorithmic debugging and property-based testing. It has often been suggested that a reference program or formal specification could be used as oracle in algorithmic debugging, but to my knowledge no previous work explains how either of these can be used soundly when the traced program is evaluated lazily.

My method re-uses properties to answer automatically some of the questions arising during algorithmic debugging, and to replace others by simpler questions. Properties may already be present in the code for testing; the programmer can also encode a specification or reference implementation as a property, or add a new property in response to a statement they are asked to judge.

During a debugging session, if it turns out that test properties are insufficient to locate a defect, then the programmer may choose to add a property in response to a computation statement presented by the debugger, or they may manually judge the statement. In the former case a debugging session not only locates the defect but also retains the information put into the debugging session by the programmer in the form of properties that can be used for future testing and debugging sessions.

To be applicable, a property need not fully specify a subject function, though I do assume that test properties are correct assertions. We may trust some functions involved in a property though we suspect others; most useful are properties where the subject function is the only suspected part of the property.

A property is associated with a subject function and used to help judge computation statements about it as right or wrong. Depending on the characteristics of the property, and on the outcome of property tests incorporating values from the computation statement, we may potentially draw different conclusions contributing to a judgement.

When properties associated with the subject function of a computation statement evaluate to `True`, if these properties together fully specify the subject function then the statement can be judged right, or if the properties only partially specify the subject function we can at least assist the user by confirming properties that hold.

When a property associated with the subject function of a computation statement evaluates to `False`, if the subject function in the property is not applied to any further extent than recorded in the computation statement then the statement can be judged wrong, or if the subject function is applied to a further extent we may find a simpler computation tree for investigation. As a conservative check for application of a subject function within a recorded computation, we provide a type-generic mechanism to constrain a function application to a specified result value.

Recorded values in non-strict evaluated programs may contain unevaluated expressions. I reduce dependencies on unevaluated expressions by introducing a parallel equality operator that can be used as a drop-in replacement for normal equality in a property. My definition for parallel equality is type-generic and automatically derivable for user defined types. When a property still gives an inconclusive result, randomly generated test-values can be used as substitutes for unevaluated subexpressions. A counter-example may lead to a simpler computation tree for investigation.

A statement may (provisionally) be judged right if a property fully specifies the statement's subject function and no counter-example can be found after testing has exhausted some specified resource.

Current strategies for algorithmic debugging focus on minimizing the total number of questions. As we now have two oracles, the human programmer and test properties, I propose new strategies for exploring the computation tree. To achieve the aim that the human programmer judges fewer or simpler statements, these strategies may opt to judge more statements in total.

I have implemented my method in a property-assisted extension of Hoed-pure. The task of associating properties with subject functions and abstracting these functions as property arguments is currently performed by hand; it is routine and could readily be automated.

9 Case Studies

Here I present five case studies of defective programs:

1. A defective program with a higher order function
2. A defective solver of a chess puzzle
3. A defective pretty printer
4. A defective video game
5. A defective window manager

The first two cases studies are constructed programs and the final three studies are existing programs. I locate the defect in case study 2, 3 and 5 using algorithmic debugging with properties as oracle. In study 5 I use only the properties written by the original developers.

9.1 A defective higher-order function

Listing 36 gives a library with a parity test and a higher order function for filtering elements from a list (top-half of figure), and properties and specifications of the functions in the library (bottom-half). Using QuickCheck we find that `spec_odds` does not hold and that `spec_isEven` does hold.

To find the location of the defect we might trace the evaluation of `spec_odds` applied to the counter example `[3,4]` (as found by QuickCheck), the computation tree is given in Figure 58. Note how functional arguments are represented as a finite mapping from argument to result. Higher order functions in computation trees are well

Listing 36 Top: a defective program for filtering elements of a list based on parity. The code contains a higher-order polymorphic function. Bottom: properties to test the program and assist in locating the defect.

```
odds :: [Int] -> [Int]
odds xs = filter (not . isEven) xs

isEven :: Int -> Bool
isEven x = (x .&. 1) == 0    ← Bit 1 is 0 for even values

filter :: (a -> Bool) -> [a] -> [a]
filter pred []    = []
filter pred (x:xs)                   | Defect: guards’ definitions
  | pred x    = filter pred xs    ←| swapped
  | otherwise = x : filter pred xs  |
```

```
spec_odds xs x =
 x ‘elem‘ xs ==> if odd_ref x then p else not p
 where p = x ‘elem‘ (odds xs)

spec_isEven x = isEven x == even_ref x

even_ref x = (x ‘div‘ 2) ∗ 2 == x

odd_ref x = not (even_ref x)

prop_filter_t p xs x =
 x ‘elem‘ xs && p x ==> x ‘elem‘ (filter p xs)

prop_filter_f p xs x =
 x ‘elem‘ xs && not (p x) ==>
 not (x ‘elem‘ (filter p xs))
```

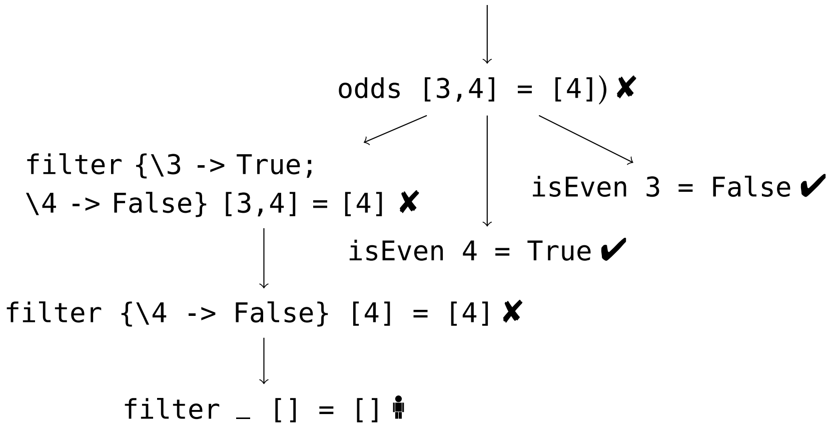


Figure 58: Computation tree with observed higher order function from Listing 36.

understood (Nilsson 1998; Wallace et al. 2001; Caballero, López-Fraguas and Rodríguez-Artalejo 2001; Braßel and Siegel 2008; Chitil and Davie 2008). With the recorded functional values and QuickCheck properties our tool judges the computation statements (marked with ✔ or ✘ in Figure 58) and leaves only one last statement for the human oracle to judge as right (marked with 🕴) to locate the defect.

9.2 A defective n-queens problem solver

Here I discuss locating the defect in the solver from Section 8.1 of the the n-queens problem, place n queens on an $n \times n$ board in such a way that no queen threatens any other (by occupying the same row, column or diagonal), in more detail.

One property we expect of the function `queens`, intended to compute a list of solutions, is that no solution has a row-number that occurs more than once. That is, for all n the following function should evaluate to `True`:

```
prop_queens_sets n = all isSet (queens n)
```

The function `isSet` tests if every element in the list to which it is applied occurs exactly once. Using QuickCheck (Claessen and Hughes 2000b), we get:

```
> quickCheck prop_queens_sets
*** Failed! Falsifiable: 4
```

When $n = 4$, the property does not hold for `queens` as defined in Listing 31. Indeed, the first item in `queens 4` is `[1,1,1,1]` — the incorrect solution of Figure 54 with all queens in the bottom row. How do we use properties as oracle to find this defect?

	properties as oracle			human oracle	
	right	wrong	advice	right	wrong
Top-down	3	6	0	0	0
Divide-and-query	2	4	0	0	0
All-children	7	6	1*	0	0
Brute-force	11	8	5*	0	0

∗) Property produced advice but defect could be determined without actually consulting the human oracle.

Figure 59: Computation statements that need to be judged to locate the defect in the n-queens solver of Listing 31.

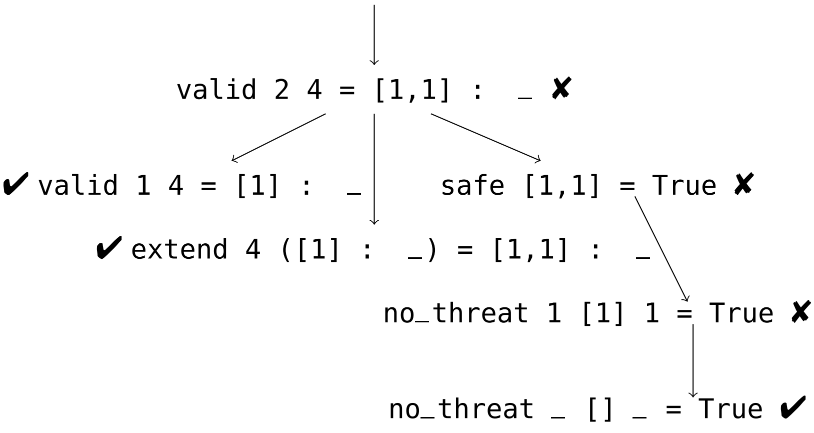

valid 2 4 = [1,1] : _ ✘
✔ valid 1 4 = [1] : _
safe [1,1] = True ✘
✔ extend 4 ([1] : _) = [1,1] : _
no_threat 1 [1] 1 = True ✘
no_threat _ [] _ = True ✔

Figure 60: A subtree of the full computation tree of `queens 4`. The statements with ✔ are right and statements with ✘ are wrong.

First we mark the functions defined in Listing 31 as suspected and we associate with each function test properties that fully specify the function. We evaluate `prop_queens_set` with the argument 4 that testing found as counter-example for the property. The computation tree for this evaluation has 28 computation statements.

Details under the top-down strategy The debugger starts by considering the statement at the root of the tree:

```
queens 4  = [1,1,1,1] : _
```

This is the same function for which prop_queens_sound detected a defect. So it is no surprise that prop_queens_sound with a restricted subject function and 4 evaluates to

False: the statement is wrong. The debugger therefore selects a child of the current statement to consider next:

```
valid 4 4 = [1,1,1,1] : _
```

With the arguments from this statement the property prop_valid_ sound with a restricted subject function evaluates to False: another wrong statement. The debugger goes a step deeper into the tree, next considering the following child of the current statement:

```
valid 3 4 = [1,1,1] : _
```

Again the property prop_valid_ sound fails with a restricted subject function, as the statement is wrong. A child of the current statement is:

```
valid 2 4 = [1,1] : _
```

One property can be used to judge many computation statements that show similar faulty behaviour. As before, testing prop_valid_ sound shows the statement is wrong. The debugger selects the child statement:

```
valid 1 4 = [1] : _
```

This time prop_valid_sound evaluates to bottom with an restricted subject function. Unrestricted, valid 1 4 evaluates to [[1], [2], [3], [4]] and texttprop_valid_sound holds. The property prop_valid_complete is also associated with valid and together these two properties fully specify the function. The completeness property takes a board as extra argument: on testing with a restricted subject function, the property is found to hold for 100 test-case boards, and is provisionally judged right.

As this statement is judged right, the debugger returns to the parent statement in the tree and considers another child of computation statement valid 2 4 = [1,1] : _ :

```
extend 4 ([1] : _) = [1,1] : _
```

Property prop_extend_sound evaluates to True. On the other hand, property prop_extend_complete is inconclusive, whether or not the subject function is restricted, as it demands the unevaluated expression in the argument. The debugger therefore generates values that could complete the argument, and for all these completed argument values the property is found to hold. Together the two properties fully specify extend, so the statement is judged right. The debugger now tries the third child of the statement valid 2 4 = (1 : 1 : []) : _, which is:

```
safe [1,1] = True
```

Property `spec_safe` evaluates to `False` with a restricted subject function. So the statement is wrong and the debugger again steps a statement deeper into the tree:

```
no_threat 1 [1] 1 = True
```

Property `spec_no_threat` with restricted subject function evaluated to False: another wrong statement, and the only child is:

```
no_threat _ [] _ = True
```

Here `spec_no_threat` evaluates to `True`. Indeed this statement is right. And with that the debugger has located the defect. Figure 60 gives an overview of the last six steps.

Strategy Comparison We have seen how application of the top-down strategy, with specifying properties as oracle, finds the defect in our implementation of a solver for the n-queens problem. Nine statements are judged automatically. No statement has to be judged by a human oracle.

Figure 59 lists for each of the two conventional strategies, *Top-down* and *Divide-and-query*, and for each of our two new strategies, *All-children* and *Brute-force*, how many questions can be answered by properties as oracle and how many questions are left for the human oracle to answer. All strategies find the defect without consulting the human oracle. So the conventional strategies, designed with a single oracle in mind, outperform the Brute-force and All-children strategies.

We conclude that evaluation of test properties as an oracle can indeed reduce, and even eliminate, the work of the human oracle in the ideal situation where for each suspected function we have properties that fully specify the function.

In practice, however, the available properties are unlikely to be so comprehensive. They may not fully specify their subject functions. Some subject functions may not even have any associated and applicable properties. Such observations are part of the motivation for the next case study, involving a real-world Haskell program.

9.3 A defective pretty-printer

Within the implementation of Hoed-pure I used version 1.0 of the library FPretty (Olaf Chitil 2012) to pretty-print computation statements over several lines with appropriate

Listing 37 Part of XMonad's code with a defect.

```
data StackSet i l a sid sd =
  StackSet { current  :: !(Screen i l a sid sd)
           , visible  :: [Screen i l a sid sd]
           , hidden   :: [Workspace i l a]
           , floating :: M.Map a RationalRect
           }
data Screen i l a sid sd =
  Screen { workspace :: !(Workspace i l a)
         , screen :: !sid
         , screenDetail :: !sd }
data Workspace i l a =
  Workspace { tag :: !i
            , layout :: l
            , stack :: Maybe (Stack a) }
data Stack a = Stack { focus  :: !a
                     , up     :: [a]
                     , down   :: [a] }

view :: i -> StackSet i l a s sd
          -> StackSet i l a s sd
view i s
  | i == currentTag s = s
  | Just x <- List.find ((i==).tag.workspace)
                        (visible s)
    = s { current = x, visible = current s
           : List.deleteBy (equating screen) x
               (visible s) }
  | Just x <- List.find ((i==).tag) (hidden  s)
    = s { current = (current s) { workspace = x }
        , hidden = workspace (current s)
          : (hidden s) }
  | otherwise = s
```

```
shiftWin (NonNegative 1) ’d’
 (StackSet
  (Screen (Workspace (NonNegative 2) _ (Just (Stack ’c’ [] "z")))
    2 1)
  ((Screen (Workspace (NonNegative 0) _ (Just (Stack ’d’ [] [])))
    1 -2)
   : (Screen (Workspace (NonNegative 3) _ (Just (Stack ’v’ [] [])))
      3 -1)
   : (Screen (Workspace (NonNegative 4) _ (Just (Stack ’w’ [] "i")))
      0 -2)
   : [])
  (( Workspace (NonNegative 1) _ (Just (Stack ’n’ [] [])))
   : (Workspace (NonNegative 0) _ Nothing)
   : (Workspace (NonNegative 4) _ Nothing) : []) _)
 = StackSet
  (Screen (Workspace (NonNegative 2) _ (Just (Stack ’c’ [] "z")))
    2 1)
  ((Screen (Workspace (NonNegative 0) _ Nothing) 1 -2)
   : (Screen (Workspace (NonNegative 3) _ (Just (Stack ’v’ [] [])))
      3 -1)
   : (Screen (Workspace (NonNegative 4) _ (Just (Stack ’w’ [] "i")))
      0 -2)
   : [])
   (( Workspace (NonNegative 1) _ (Just (Stack ’d’ [] "n")))
   : (Workspace (NonNegative 2) _ (Just (Stack ’c’ [] "z")))
   : ( Workspace (NonNegative 1) _ (Just (Stack ’n’ [] [])))
   : (Workspace (NonNegative 0) _ Nothing)
   : (Workspace (NonNegative 4) _ Nothing) : []) _
```

Figure 61: Some questions can be difficult to answer for the programmer. The answer to this question is *wrong* because `WorkSpace` with tag 1 (in **bold**) occurs once in the argument but twice in result.

```
insertUp ’d’
 (StackSet (Screen
  (Workspace (NonNegative 1) _ (Just ( Stack ’n’ [] [])))) 2 1)
  ((Screen (Workspace (NonNegative 0) _ Nothing) 1 -2)
   : (Screen (Workspace (NonNegative 3) _ (Just (Stack ’v’ [] []
     ))) 3 -1)
   : (Screen (Workspace (NonNegative 4) _ (Just (Stack ’w’ []
     "i"))) 0 -2)
   : [])
   ((Workspace (NonNegative 2) _ (Just (Stack ’c’ [] "z")))
   : (Workspace (NonNegative 1) _ (Just (Stack ’n’ [] [])))
   : (Workspace (NonNegative 0) _ Nothing)
   : (Workspace (NonNegative 4) _ Nothing) : []) _)
 = StackSet (Screen
    (Workspace (NonNegative 1) _ (Just ( Stack ’d’ [] "n"))) 2 1)
  ((Screen (Workspace (NonNegative 0) _ Nothing) 1 -2)
  : (Screen (Workspace (NonNegative 3) _ (Just (Stack ’v’ [] [])))
    3 -1)
  : (Screen (Workspace (NonNegative 4) _ (Just (Stack ’w’ [] "i")))
    0 -2)
  : [])
  ((Workspace (NonNegative 2) _ (Just (Stack ’c’ [] "z")))
  : (Workspace (NonNegative 1) _ (Just (Stack ’n’ [] [])))
  : (Workspace (NonNegative 0) _ Nothing)
  : (Workspace (NonNegative 4) _ Nothing) : []) _
```

Figure 62: Some questions can be difficult to answer for the programmer. The answer to this question is *right*: the first argument inserted in the current stack (in **bold**) and StackSet is unchanged otherwise. Although there are two WorkSpaces with tag 1 in the resulting StackSet, the statement is not wrong because the two WorkSpaces also occur in the second argument.

```
view (NonNegative 1)
 (StackSet
  (Screen (Workspace (NonNegative 2) _ (Just (Stack ’c’ [] z̈)̈))
    2 1)
  ((Screen (Workspace (NonNegative 0) _ Nothing) 1 -2)
   : (Screen (Workspace (NonNegative 3) _ (Just (Stack ’v’ [] [])))
     3 -1)
   : (Screen (Workspace (NonNegative 4) _ (Just (Stack ’w’ [] ï)̈))
     0 -2)
   : []
  )
  (( Workspace (NonNegative 1) _ (Just (Stack ’n’ [] [])))
   : (Workspace (NonNegative 0) _ Nothing)
   : (Workspace (NonNegative 4) _ Nothing) : []
  ) _ )
 = StackSet
  (_Screen (Workspace (NonNegative 1) _ (Just (Stack ’n’ [] []))) 2 1)
  ((Screen (Workspace (NonNegative 0) _ Nothing) 1 -2)
   : (Screen (Workspace (NonNegative 3) _ (Just (Stack ’v’ [] []))) 3 -1)
   : (Screen (Workspace (NonNegative 4) _ (Just (Stack ’w’ [] "i"))) 0 -2)
   : []
  )
  ((Workspace (NonNegative 2) _ (Just (Stack ’c’ [] "z")))
  : ( Workspace (NonNegative 1) _ (Just (Stack ’n’ [] [])))
  : (Workspace (NonNegative 0) _ Nothing)
  : (Workspace (NonNegative 4) _ Nothing) : [])
  _
```

Figure 63: Some questions can be difficult to answer for the programmer. The answer to this question is *wrong* because WorkSpace with tag 1 (in **bold**) occurs once in the argument but twice in result.

Listing 38 Some of XMonad's properties.

```
prop_shift_win_I (n :: NonNegative Int)
                 (w :: Char) (x :: T) =
    n ‘tagMember‘ x && w ‘member‘ x
    ==> invariant $ shiftWin (fromIntegral n) w x

prop_view_I (n :: NonNegative Int) (x :: T) =
    invariant $ view (fromIntegral n) x

invariant (s :: T) = nub ws == ws
  where
  ws = concat [ focus t : up t ++ down t
                | w <- workspace (current s)
                  : map workspace (visible s)
                  ++ hidden s
                , t <- maybeToList (stack w)]
```

indentation. I noticed that the library sometimes indents more than I expected and 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. I annotated all 12 top-level functions in the library. Then I pretty-printed an example, during which 15327 events were collected. These events translated to 65 computation statements which Hoed-pure organised in a computation tree with 65 edges and branch factor 1.8. I found the defect after judging 11 statements. After I found the defect, I 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.

9.4 A video game

The game Raincat (Game Creation Society, Carnegie Mellon 2014) consists of approximately 2500 lines of Haskell code and uses libraries such as OpenGL that are not written in Haskell. Raincat cannot easily be traced with the Haskell tracer Hat.

The user of Raincat can use the mouse cursor to click on rectangular buttons. When the user of Raincat clicks somewhere the coordinates of the mouse cursor are compared to the position and size of the buttons to check whether a button is clicked. This check is performed by a function `pointInRect`. I introduced a defect in this function such that many of the user’s clicks on a button are incorrectly rejected (see Listing 39).

While my debugger is most useful on pure computations, it proved no problem for Hoed-cc or Hoed-pure to generate a computation tree for a program that also contained IO.

Because Raincat is an interactive game, its trace is different for every run. Some parts of the code are executed when a timer times-out or when the cursor is moved. Hence I cannot compare debugging sessions in detail. To reduce the size of the computation tree, and hence the amount of questions that need to be judged, it proved worthwhile to perform a scenario that makes Raincat to misbehave and to immediately terminate the program after the unexpected behaviour is observed.

Listing 39 Defective function from Raincat to determine if a point is inside a rectangle.

```
pointInRect (x, y) (Rect x’ y’ w h) =
  x >= x’ && x <= (x’ + w) && y <= y’ && y >=  (y’ + h)
                                ↑          ↑
```

Comparison only matches when $y = y'$ because comparison operators (<=) and (>=) are swapped.

9.5 A defective window manager

XMonad[1] is a dynamically tiling X11 window manager written in Haskell. Our discussion here is based on version 0.11.1 of XMonad, which has a core of around 1300 lines of code. As property-based testing was used during the development of XMonad, the code comes with 112 already-defined properties (not included in the core line-count). Listing 38 gives some examples of these properties.

XMonad organizes windows in a stack. A stack with a tag is a *workspace*. XMonad can manage one or more screens, each screen has a unique id and can display one workspace at a time. All screens and the current active workspace are organized in a *stack set*. There can be more workspaces than screens, and the workspaces that are

[1] http://xmonad.org

not visible on a screen are *hidden*. (The user presses the alt-button together with one of the numbers 0-9 to select which of the workspaces to display on the current screen — a feature is also referred to as *virtual desktops*). The top half of Listing 37 shows the data types used in XMonad to represent these abstractions.

Introducing a defect Listing 37 also shows the definition of the function `view`, *the only part of the code we changed*. The `view` function is applied to an identifier `i` and a `Stackset s`. The intention is that the function finds the `Workspace` whose identifier is equal to `i` and sets that `Workspace` in focus. This is implemented as follows. The first guard checks if the current workspace has identifier `i`, in which case we can just return `s` as it is. The second guard checks if a workspace with identifier `i` is already shown on one of the screens, in which case the screen with that workspace is made the current screen. The third guard checks if one of the hidden workspaces (say workspace `x`) has identifier `i`, in which case `x` is set as the workspace of the current screen. The underlined expression in this part of the definition is where we introduced a defect: the previously current workspace is now also hidden, but we forget to remove `x` from the set of hidden workspaces. The final guard covers the case where the user tries to switch to a workspace that does not exist.

Detecting and locating the defect We take the role of a programmer who notices that windows sometime appear twice on their screen. Our first step is to run XMonad's extensive property-based test-suite. Testing finds counter-examples for no less than 20 properties!

- prop_shift_win_I
- prop_shift_I
- prop_swap_right_I
- prop_swap_left_I
- prop_swap_master_I
- prop_delete_local
- prop_delete_insert
- prop_delete_I
- prop_insertUp_I
- prop_greedyView_reversible
- prop_view_greedyView_I
- prop_view_local
- prop_view_reversible
- prop_view_I
- prop_invariant
- prop_focus_I
- prop_focusDown_I
- prop_focusMaster_I
- prop_focusUp_I
- prop_ensure

These failing properties give us some indication of where the defect is likely to be —in something that shifting, swapping and focussing a window have in common— but they do not tell us exactly where the defect is.

We arbitrarily pick the failing property prop_shift_win_I, and its counterexample, to trace and debug.

I annotated the 9 functions in the code related to the failing property and 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. The 11 statements describe three applications of findTag, one application of insertUp, two applications of member, one application of shiftWin and four applications of view. The statement of Figure 61 is at the root of the recorded computation tree.

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. 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.

Properties suitable for algorithmic debugging are provided by the XMonad authors for all but the `member` function: this function only occurs in properties that also contain other suspected functions. None of the functions is specified completely by their properties.

Details under the top-down strategy Figure 65 shows part of the recorded computation tree. Top-down debugging considers the statement at the root first, and by evaluating `prop_shift_win_I` we determine that this statement is wrong. The next statement is an application of `insertUp`. The properties we have for this statement do not fully specify the function. Therefore the human oracle is needed to confirm that the statement is right. The final statement we consider with the top-down strategy is an application of `view`. Using property `prop_view_I` (see Listing 38) the debugger determines that this statement is wrong, and because the statement has no children the defect is located in `view`.

Strategy Comparison We have seen that using the top-down strategy we find the defect after judging just three statements, for one of which the human oracle is consulted.

Figure 66 lists for each of four strategies the number of judgements made using properties as oracle, and the number requiring a human oracle, before the defect in XMonad is located. Although the All-children and Brute-force strategies both require more statements to be judged than the usual human-oracle strategies, they each locate the defect without needing to consult the human programmer.

We compared for the deliberately introduced defect described above, and for five more defects introduced in other functions, how many statements have to be judged to locate the defect (Figure 64). The best strategy is one that consults the human oracle least. If several strategies require the same number of human judgements, then the best is one that requires the smallest number of judgements by properties as oracle. By these criteria, for four cases in Figure 66 the All-children and Divide-and-query strategies are equally good.

As we explained in Section 8.3.2, when a statement is right but the subject function's properties do not fully specify it, successful property tests cannot give a conclusive judgement. Using a conventional top-down strategy, the only option may be to consult the human oracle. However, the All-children and Brute-force strategies queue statements. Often statements in the queue do not have to be considered at all, because there is a chain of wrong statements from the root of the computation tree leading to the defective statement.

Suppose a parent statement is wrong and has n children, of which one is wrong

	defect	optimal	properties as oracle			human oracle	
starting point	location	strategy	right	wrong	advice	right	wrong
`prop_shift_win_I`	`view`	All-children	0	3	4	0	0
`prop_greedyView_local`	`greedyView`	Divide-and-query	0	1	0	0	0
`prop_allWindowsMember`	`member`	Divide-and-query	0	1	0	1	0
`prop_allWindowsMember`	`findTag`	Divide-and-query	0	2	0	0	0
`prop_insertUp_I`	`findTag`	Divide-and-query	0	3	0	0	0
`prop_swap_all_r`	`reverseStack`	Divide-and-query	0	1	0	0	1

Figure 64: Computation statements that need to be judged to locate a defect in each of six faults introduced in XMonad.

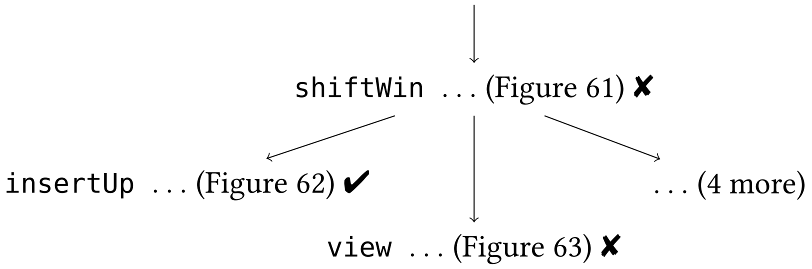


Figure 65: A subtree of the the computation tree recorded evaluating XMonad's property `prop_shift_win_I` applied to a counter-example.

	properties as oracle			human oracle	
	right	wrong	advice	right	wrong
Top-down	0	2	0	1	0
Divide-and-query	0	2	0	1	0
All-children	0	3	4^*	0	0
Brute-force	0	3	6^*	0	0

$*$) Property produced advice but defect could be determined without actually consulting the human oracle.

Figure 66: Computation statements that need to be judged before locating the defect in XMonad.

and $n-1$ are right. If all properties are partial specifications, then using the Top-down strategy the human oracle may be consulted up to $n-1$ times before the properties as oracle finally judge the n^{th} child wrong. But using a strategy that places the children in a queue, properties as oracle may find that the n^{th} child is wrong without consulting the human oracle, and then the other children can be removed from the queue.

9.6 Summary

Comparison of the trees of Hoed-cc and Hoed-pure in case studies confirmed that a tree purily generated from the value observation trace requires fewer answers from the oracle compared to a tree generated with cost centre stacks. In one case I also found that the Hoed-cc tree gives a less precise defect location compared to algorithmic debugging with the Hoed-pure tree.

I compared conventional algorithmic debugging strategies with the two new strategies from Section 8.8 that queue statements for which properties as oracle is inconclusive. Conventional strategies ask the oracles fewer questions in total and therefore perform best in the ideal case where our properties fully specify their subject function. With properties that partially specify their subject function, such as those for XMonad, a strategy that queues questions for which properties as oracle do not have a conclusive answer asks the oracles more questions in total, but fewer questions have to be answered by the human oracle.

In our case studies properties as oracle, combined with the right strategy, judged enough statements to locate the defect. Consulting the human oracle was not necessary. Our methods can indeed make test properties suitable as an oracle for algorithmic debugging. In this way, algorithmic debugging can be applied to some programs for which a human oracle alone would be overwhelmed by the size or number of questions.

10
Related work

Methods for locating the defect in a misbehaving program's code are studied by researchers for decades because defective programs cause all kinds of problems and lead to huge economic costs for society (Tassey 2002). Zeller (2009) gives an overview of many debugging techniques across different paradigmes. Throughout this thesis I also use Zeller's terminology of defect, infection and failure.

Here I give an overview of the work most closely related to mine. I focus on algoritmic debugging, computation tree construction for algorithmic debugging of lazy functional programs and debugging methods of lazy functional programs in general. I also discuss profiling, used in my computation tree construction method with cost centre stacks, and generic programming frameworks which can be used to implement my type-generic definition for value observation tracing.

10.1 Kinds of defects

Gerhart and Yelowitz (1976) divide defects in three classes: a defective specification, a defectively constructed program and a defect in a program's correctness proof.

The specification should express unambiguously and completely what the requirements of the program are. A common error in the specification is failure to be complete. When parts of the input or output are split in separate sets it is easy to not cover the whole domain or range. Other defects are the use of terms with an implicit understanding, and slips in the translation from concepts to formal specification.

Program construction is the implementation of a program that satisfies the requirements. Defects result from the assumptions and informal reasoning inherent to the nature of writing code. Shapiro (1983) lists incorrect output, missing output and non-termination as the three symptoms of an error in the implementation of a program.

Proving is the use of mathematical systems to show that the implemented program is correct according its specification. The most common error in proving is failing to define what should be proved in order to guarantee correctness. Skipping steps in the proof, or using an approach that is too informal are other causes for defects.

Shapiro (1983, Section 1.3.1) concludes that "a program can be proven correct formally only with respect to another formal description of its intended behaviour, [proving] does not solve the problem of program debugging, but simply reduces it to the problem of debugging specifications"

10.2 Algorithmic debugging

Shapiro (1983) invented the algorithmic debugging method for finding defects in Prolog programs. Shapiro's method records applications of first order functions and their results in a computation tree. He uses the human programmer as oracle to judge the statements in the tree. Later work generalized algorithmic debugging to deal with concepts such as higher-order functions, laziness and loops.

10.2.1 Computation tree tracing for Haskell

The algorithmic debuggers Freja (Nilsson and Sparud 1997; Nilsson 1998), Hat (Wallace et al. 2001) and Buddha (Pope 2006) use different but complex implementations to obtain a computation tree for a Haskell execution.

Freja (Nilsson and Fritzson 1992; Nilsson and Sparud 1997; Nilsson 1998) 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 (Sparud and Runciman 1997; Wallace et al. 2001; Chitil, Runciman and Wallace 2003) 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, Runciman and Wallace (2001) compare an old version of Hat with Freja and HOOD. 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 (Pope 2005, 2006) 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 provides an extensional representation of functional values, that is a finite map from argument to result values. Adding a language feature would require extending the source-to-source transformation and possibly the primitive.

10.2.2 Computation tree tracing for other languages

Shapiro constructed computation trees for the logic language Prolog (Shapiro 1983). Algorithmic debugging has since been applied to many other languages; I give a few notable examples.

Fritzon et al. generalized computation tree tracing to languages with side effects (Fritzson et al. 1992). An algorithmic debugger with a framework to record side-effects in the computation tree is for example available for Java (Caballero, Hermanns and Kuchen 2007; Cabrera and Silva 2010). Tail call optimization is forbidden and higher order functions are not supported.

Cabrera (2016) defined several techniques to make algorithmic debugging more suitable for imperative programs by converting loops into recursive function calls. He proposes a new method of searching the computation tree that asks the oracle fewer questions. Cabrera also proposes a hybrid technique for that combines traditional stepwise debugging for imperative programs with algorithmic debugging.

Algorithmic debugging is also applied to strict functional languages such as Erlang (Caballero et al. 2014). The implementation is complex and uses a specific run-time system to transform all code, including libraries, during evaluation of the program.

10.2.3 Display of functional values

Davie and Chitil (2006a) list three ways of displaying a functional value:

First of all, the name of a function as used in the function definition. Note that many languages allow function definition without a name, the anonymous or lambda function.

Secondly, as a finite map of argument-result values as used before by Gill (2000) in HOOD and by Pope (2006) in his algorithmic debugger Buddha.

Finally, as the function body. An hybrid approach is imaginable where, depending on the situation, one of these three methods is chosen.

Furthermore Pope and Naish (2002) discuss displaying partial application. For example the list of functional values returned by `map (+) [1,2]` could be shown as the function name followed by the value given as first argument: `[(+) 1, (+) 2]`.

10.2.4 Automated oracles

Papers on algorithmic debugging tend to focus on the construction and soundness of computation trees. The possibility in principle of using an automated oracle may be mentioned, but it is rarely investigated in practice. Actual implementations of algorithmic debuggers rely on a human oracle.

Drabent and Nadjm-tehrani (Drabent and Nadjm-Tehrani 1989) describe an algorithmic debugger for Prolog where the user can next to right/wrong also respond with an assertion (property). An assertion is also just Prolog and can contain calls to trusted library functions. Assertions are parametrised by recorded (first order) inputs and outputs. However, for many computation statements in non-strict functional programs these assertions are not powerful enough to derive a judgement (even when we do not consider unevaluated expressions, a function as argument can easily lead to evaluating parts of the code that were not part of the traced computation).

Fritzson, Auguston and Shahmehri (1994) define a specific language to define properties in that describe the desired behaviour of statements with side-effects in imperative languages.

Claessen et al. (2003) find that detecting a failure (and thus the existence of a defect in the code) with property-based testing and finding the defect with tracing play well together. Interaction with the trace uses a human oracle.

Tamarit et al. (2016) keep a database of computation statements and their judgement collected during debugging sessions of defective Erlang programs. Erlang is a strict functional language, hence the values in the statements are always fully evaluated. The statements can both be used to answer questions in future debugging sessions and for unit testing.

An alternative approach to defect location is based on the idea of annotating a function with a contract, that is the function's pre and post conditions (Findler and Felleisen 2002; Chitil 2012). To locate a defect contracts need to be full specifications and all suspected functions must be annotated. There is no hybrid approach that combines contracts with judgements from a human oracle.

Lazy SmallCheck (Runciman, Naylor and Lindblad 2008) also defines parallel logical-operators for use in properties but the library does not define a parallel equivalence method.

10.3 Other debugging methods

Many other methods for finding defects in lazy functional programs have been explored. Here I give an incomplete overview of approaches most closely related to mine.

10.3.1 Walking backward

Sparud and Runciman (1997) describes how to use a redex trail to start from a function application that resulted in an exception. We start from the redex whose reduction resulted in the exception. The redex consists of a function and one or more arguments which are the result of reduction of another redex: the dependencies of the current redex. The user repeatedly selects dependencies until the location of the bug in their code is revealed. Hat's augmented redex trails can be walked in similar fashion with hat-trail Wallace et al. (2001).

10.3.2 Generating redex trails with Hat

Computation based on graph reduction is achieved by repeatedly replacing one subgraph by another. At each step a redex is replaced by its reduct, parts no longer attached to the main graph are normally discarded.

Sparud and Runciman (1997, 1998) describe a system to generate an acyclic graph with the history of redex reductions called the redex trail. The redex trail is constructed by adding a link from each newly created node of the graph made to its parent redex. Old redexes stay connected and are not discarded. As a result the computation constructs its own trace.

This behaviour is achieved with a source to source transformation combined with a library. Some non-standard unsafe functions are used to trace, and some Haskell extensions are used to connect to the trace after termination (Chitil, Runciman and Wallace 2001).

Every value is wrapped in a special data type containing the original value and a trace for that value. The wrapping requires that either the whole program is annotated, or that values around trusted code are (un)wrapped appropriately. The wrapping and unwrapping can have the consequence that the relation between two reduced redexes of traced functions that are in each others lexical scope is not made. For example, assume annotated function definition `f x = h g x` with trusted higher order function `h` and annotated function `g`. No connection between the g-part of the redex trail and the f-part of the redex trail is made.

Lacking these connections would break most of the Hat viewers. Therefore Hat does annotate trusted code in such a way that reductions of redexes originating from the trusted code are not traced, but connections between traced reductions are made.

Shackell and Runciman (2005) investigate generating redex trails by modifying the underlying abstract machine instead of transforming the source. The goal of this approach is to make tracing faster. Their work looks promising but the paper lacks detail and to my knowledge no further research into this subject has been published.

The redex trail contains most of the information needed to construct an evaluation dependency tree (Chitil, Runciman and Wallace 2001). Wallace et al. (2001) extended Hat to generate an augmented redex trail that contains the information of the redex trails plus the information required to derive an evaluation dependence tree. The augmented redex trail is written to file and can be viewed in a number of different ways.

10.3.3 List of observations

Sinclair (1992) proposed debugging programs by observing the data flow: tracing intermediate values between functions. How the observing is to be done is not specified in this paper. GHC and most other Haskell compilers come with the `trace` function which allows users to print strings from otherwise pure functions. Careless use of `trace` however can change the order of evaluation.

HOOD is a library to trace the evaluated part of intermediate data structures (Gill 2000). With HOOD's `observe` function a list of values at an annotated point in the code can be obtained. With Hat a similar list can be extracted from the redex trail. This method does not guide the programmer to the bug in their program: they have the freedom to annotate their code where insight into intermediate values is required. Chitil, Runciman and Wallace (2001) suggest to apply a top-down strategy in case a wrong result is produced, and a bottom-up strategy when a program fails with a message that reveals the position.

Hugs keeps a type-representation of all values during runtime. HugsHood allows observation all values through type reflection and the user of HugsHood does not need to write instances of `Observable` to observe values of a user-defined type (Jones et al. 2004). Most other Haskell compilers do not provide run-time type information. It would therefore be hard to implement the Hugs debugging primitives in these compilers (Braßel et al. 2004). HugsHood extends Hood with an interesting "breakpoint" feature that shows the development of traces over time.

GHood extends HOOD with a graphical representation of the trace showing development over time (Reinke 2001).

COOSy is an adaptation of HOOD for the functional logic language Curry. COOSy's `observe` function takes a type description, somewhat similar to the list of types we specify in our Partial Observe from Template approach. Partly this was done because Curry lacks a class system, but like our extension it also enables the user to specify per observation up to which type to observe (Braßel et al. 2004). However, unlike COOSy, we also allow to observe into a polymorphic value, at the cost of needing to add a class predicate to the type signature of the value under observation.

10.3.4 Breakpoint-style debugging

An alternative to tracing is an interactive breakpoint-style debugger (Marlow et al. 2007). An interactive debugger is attractive, because its implementation is relative straightforward. Traced code in which breakpoints can be set and other code can be combined. 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.

10.4 Profiling

Sansom and Peyton Jones (1997) associate *cost centres* with expressions in Haskell and attribute cost of evaluating an expression to its cost centre. Soundness of the semantics for cost centres is proven. Later Morgan and Jarvis (1998) the idea of cost centres is generalized to a stack of cost centres

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, Peyton Jones and Eisenbach (2009) suggest an alternative scheme of stack compression that maintains linear space overhead while providing greater precision by telling us where labels are dropped.

The cost centre stack idea is further refined and implemented in GHC, soundness of the semantics of the GHC implementation is verified with QuickCheck by Marlow (2012). He presents a Launchbury based semantics and verifies with QuickCheck that inlining non-recursive functions does not affect the stack.

The GHC profiler uses cost centre stacks to attribute time and space to annotated expressions. Libraries are generally shipped already with a version compiled for profiling.

10.5 Generic programming frameworks

Hinze, Jeuring and Löh (2007) did a broad comparison of approaches to generic programming, and Rodriguez et al. (2008) defined a generic programming benchmark to compare 9 generic programming libraries Both were valuable sources of information writing this thesis.

Scrap Your Boilerplate allows querying and mapping over the components of a value (Lämmel and Peyton Jones 2003).

The Scrap Your Boilerplate With Class approach and the Smash Your Boilerplate variant are similar to SYB but introduce a dictionary-approach to add new methods to the Data class. (Lämmel and Peyton Jones 2005; Kiselyov 2006).

The Uniplate and Strafunsky libraries are variations on SYB offering different interfaces but neither allows mapping over more types compared to SYB (Mitchell and Runciman 2007; Lämmel and Visser 2001).

The Generic Deriving Mechanism (Magalhães et al. 2010) manipulates a value through its product-sum representation (in my implementation Hoed I implement the type-generic definition of Chapter **??** with the Generic Deriving Mechanism).

Macro-like expansion are possible with the meta-language Template Haskell (Sheard and Peyton Jones 2002).

The Generics for the Masses approach is captured completely in Haskell 98. A class is used that the user has to adapt for each new type. This approach is therefore not suitable to implement a type generic method (Hinze 2004; Lämmel and Peyton Jones 2005). Later work addressed this problem at the cost of introducing boilerplate code that was not in the original approach (Oliveira, Hinze and Löh 2006).

Hinze and Löh (2006) introduces the Lifted Spine View as a generalization of the SYB approach. Next to data constructors, with this view also type constructors can be represented. Unlike TH we cannot infer if a type is of a certain class, or if a type variable has a class predicate. To my knowledge, there is no working implementation of this approach.

PolyP is an extension to Haskell allowing the definition of type generic functions over types of kind $*$ and over higher kinded types as long as the types do not contain function spaces (Jansson and Jeuring 1997).

DrIFT allows the programmer to add directives to the program which create code from rules defined in a separate file (Winstanley and Meacham 2008). DrIFT's directives are comparable to splicing in TH, and its rules are comparable to the templates of TH. DrIFT is not as powerful as TH: data types with higher kinded type variables (e.g. `Tree a`) are not handled (Hinze, Jeuring and Löh 2007).

11 Summary and 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. I have presented two new lightweights method for generating a computation tree. The implementation supports all of Haskell and requires minimal maintance in case of future language extension.

My first methods constructs a computation tree for algorithmic debugging that needs only local annotations and GHC's profiling run-time system. Already when Marlow (2012) revisited trace stacks, he was aiming to use them for time and space profiling, coverage analysis and traditional debugging. Now my method provides insight into the relationship between computation trees and profiling.

However there is also room for improvement: surplus dependencies can lead the algorithmic debugger to asking unnecessary questions or to a sound but inaccurate conclusion. Furthermore, not all run-time environments support the cost centre stack extension.

Starting point of the second method is my formal definition of the value observation 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.

The algorithmic debugging technique breaks down when the human oracle becomes overwhelmed by the size and number of computation statements. I have presented a new semi-automated method for defect location in functional programs, based on algorithmic debugging and property-based testing. It has often been suggested that a reference program or formal specification could be used as oracle in algorithmic debugging, but to my knowledge no previous work explains how either of these can be used soundly when the traced program is evaluated lazily. My method re-uses properties to answer automatically some of the questions arising during algorithmic debugging, and to replace others by simpler questions.

Properties may already be present in the code for testing; the programmer can also encode a specification or reference implementation as a property, or add a new property in response to a statement they are asked to judge. Properties that are added during a debugging session may be used again for further testing in the future.

I implemented all these techniques in the tracer and algorithmic debugger Hoed for Haskell. Case studies with defective programs from open-source projects show that Hoed makes algorithmic debugging useable for a much wider range of Haskell programs. Hoed is available for download from

```
http://hackage.haskell.org/package/Hoed
```

In conclusion, my contribution makes algorithmic debugging applicable to more programs because an implementation of my computation tree tracing method is less complex than existing methods. Furthermore, adding support for new language features is expected to require less effort compared to adding support to existing computation tree tracing approaches. Using test-properties as oracle I made algorithmic debugging scale to large and complex programs

12 Further work

My computation tree tracing method is implemented in a library with annotations for suspected functions. Adding an annotation to a function is a mechanical process and could be automated. I discussed different methods of dealing with constants and their dependencies, future work may explore which method is best.

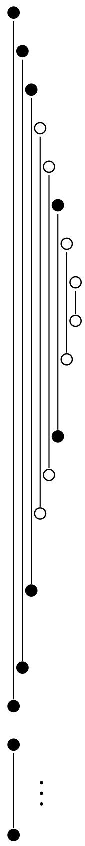
In some cases the programmer does not want to give a right/wrong judgement but for example investigate where the value a function is applied to comes from. Future work may explore producing Hat-like traces based on value observation. A first step into this direction is already made (Chitil, Faddegon and Runciman 2016).

A programmer might find it difficult to define the right properties for their functions. Research in this area has produced tools that help programmers to discover properties of a correct function (Claessen, Smallbone and Hughes 2010; Braquehais and Runciman 2016), future work may explore how the programmer can be assisted in finding properties of a (possibly) defective function.

In Chapter 5 I showed the relation between profiling and tracing and use profiling information for constructing a computation tree. In Chapter 6 I show a less invasive method for constructing a computation tree that requires no changes to the run-time system. Profiling currently still requires a specialized run-time system, and it would be worthwhile to investigate if also profiling can be implemented with a tracing library.

The profiling library could introduce additional additional *cost* events that are added to the trace e.g. upon allocating memory. The algorithm from Listing 21 can be modified to assign the cost of a cost event to the stack of labels from the current node to the root of tree. Consider for example the value observation trace with cost events 11, 15 and 17 in Figure 67. Cost A would be assigned to ⟨`plusOne`, `isOdd`⟩, cost B would be assigned to ⟨`modTwo`, `isEven`, `isOdd`⟩, and cost C to ⟨`isEven`, `isOdd`⟩.

Alternatively an increasing cost (e.g. the number of clock-ticks since starting the



1:	request	result of `isOdd`		t_1
2:	request	result of `isEven`		t_2
3:	request	result of `modTwo`		t_3
4:	request	argument of `modTwo`		t_4
5:	request	argument of `isEven`		t_5
6:	request	result of `plusOne`		t_6
7:	request	argument of `plusOne`		t_7
8:	request	argument of `isOdd`		t_8
9:	response	argument of `isOdd`	is 2	t_9
10:	response	argument of `plusOne`	is 2	t_{10}
11:	cost	A		
12:	response	result of `plusOne`	is 3	t_{12}
13:	response	argument of `isEven`	is 3	t_{13}
14:	response	argument of `modTwo`	is 3	t_{14}
15:	cost	B		
16:	response	result of `modTwo`	is 1	t_{16}
17:	cost	C		
18:	response	result of `isEven`	is `False`	t_{18}
19:	response	result of `isOdd`	is `False`	t_{19}
20:	request	result of `isOdd`		t_{20}
⋮	⋮	⋮		
38:	response	result of `isOdd`	is `False`	t_{38}

Figure 67: Trace with cost for `prop_notBothOdd 2` from Listing 19 on page 76.

program) could be added to every event. Then the cost of a span is equal to the difference between the recorded cost in the span's request and result event. Cost could be computed by subtracting cost of all nested spans and accumulated cost can be determined by subtracting the cost of spans nested in negative spans. Consider for example the events 1-20 in value observation trace of Figure 67, ignoring the cost events. The cost of `isOdd` is $(t_{20} - t_1) - (t_{18} - t_2)$. Similarly, the accumulated cost for `isOdd` is $(t_{20} - t_1) - (t_9 - t_8)$.

Bibliography

Allwood, T. O., Peyton Jones, S. and Eisenbach, S. (2009). Finding the needle: stack traces for GHC. In *Proceedings of the symposium on Haskell*, pp. 129–140.

Augustsson, L. (1999). Partial evaluation in aircraft crew planning. In *Partial Evaluation*, Springer, pp. 231–245.

Av-Ron, E. (1984). *Top-down diagnosis of Prolog programs*. Ph.D. thesis, Weizmann Institute.

Barendregt, H. P. and Barendsen, E. (1984). Introduction to lambda calculus.

Beizer, B. (1990). *Software testing techniques*. Van Nostrand Reinhold.

Bird, R. and Wadler, P. (1988). *Introduction to functional programming*, vol. 1. Prentice Hall New York.

Braquehais, R. and Runciman, C. (2016). Fitspec: Refining property sets for functional testing. In *Proceedings of the 9th International Symposium on Haskell*, New York, NY, USA: ACM, Haskell 2016, pp. 1–12.

Braßel, B. and Siegel, H. (2008). 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, pp. 183–200.

Braßel, B. et al. (2004). Observing functional logic computations. In *Practical Aspects of Declarative Languages*, Springer LNCS 3057.

Brehm, T. (2001). A toolkit for multi-view tracing of Haskell programs.

Caballero, R., Hermanns, C. and Kuchen, H. (2007). Algorithmic debugging of Java programs. *Electronic Notes in Theoretical Computer Science*, 177, pp. 75–89.

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

Caballero, R. et al. (2014). EDD: A Declarative Debugger for Sequential Erlang Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 581–586.

Cabrera, D. I. (2016). *Optimization Techniques for Algorithmic Debugging*. Ph.D. thesis, Universitat Politecnica de Valencia.

Cabrera, D. I. and Silva, J. (2010). An algorithmic debugger for Java. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, IEEE, pp. 1–6.

Chitil, O. (2005). Source-based trace exploration. In C. Grelck, F. Huch, G. J. Michaelson and P. Trinder, eds., *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, Springer, LNCS 3474.

Chitil, O. (2012). Practical typed lazy contracts. In *Proc. 17th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP'12)*, pp. 67–76.

Chitil, O. and Davie, T. (2008). 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, pp. 205–216.

Chitil, O., Faddegon, M. and Runciman, C. (2016). A Lightweight Hat: Simple, Type-Preserving Instrumentation for Self-Tracing Lazy Functional Programs. In *Implementation of Functional Languages*, presented and submitted to post-proceedings.

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

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

Chitil, O., Runciman, C. and Wallace, M. (2003). Transforming Haskell for tracing. In *Implementation of Functional Languages*, LNCS 2670, pp. 165–181.

Church, A. (1936). An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2), pp. 345–363.

Claessen, K. and Hughes, J. (2000a). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP 2000, pp. 268–279.

Claessen, K. and Hughes, J. (2000b). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proc. 5th ACM SIGPLAN Intl. Conf. on Functional Programming*, ICFP 2000, pp. 268–279.

Claessen, K., Smallbone, N. and Hughes, J. (2010). Quickspec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs*, Springer, pp. 6–21.

Claessen, K. et al. (2003). Testing and tracing lazy functional programs using QuickCheck and Hat. In *Advanced Functional programming*, Springer, pp. 59–99.

Davie, T. and Chitil, O. (2006a). Display of functional values for debugging. In *Proceedings of the 18th International Symposium on Implementation and Application of Functional Languages, IFL 2006*, pp. 326–337.

Davie, T. and Chitil, O. (2006b). One right does make a wrong. In *Pre-Proceedings of the Seventh Symposium on Trends in Functional Programming, TFP 2006.*

Drabent, L. and Nadjm-Tehrani, S. (1989). Algorithmic debugging with assertions. In *Meta-programming in logic programming*, Citeseer.

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

Faddegon, M. and Chitil, O. (2015). 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, pp. 33–42.

Faddegon, M. and Chitil, O. (2016). Lightweight computation tree tracing for lazy functional languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA: ACM, PLDI '16, pp. 114–128.

Faddegon, M. and Chitil, O. (2017). Type generic observation of intermediate data structures for debugging lazy functional programs. *Computer Languages, Systems & Structures.*

Faddegon, M. and Runciman, C. (2016). Test Properties as Oracle for Algorithmic Debugging of Lazy Functional Programs. In *Implementation of Functional Languages*, presented and submitted to post-proceedings.

Findler, R. B. and Felleisen, M. (2002). Contracts for higher-order functions. In *Proc. 7th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP'02)*, pp. 48–59.

Fritzson, P., Auguston, M. and Shahmehri, N. (1994). Using assertions in declarative and operational models for automated debugging. *Journal of Systems and Software*, 25(3), pp. 223–239.

Fritzson, P. et al. (1992). Generalized algorithmic debugging and testing. *ACM Lett Program Lang Syst*, 1(4), pp. 303–322.

Game Creation Society, Carnegie Mellon (2014). Raincat, `http://www.gamecreation.org/game/raincat`.

Gerhart, S. L. and Yelowitz, L. (1976). Observations of fallibility in applications of modern programming methodologies. *Software Engineering, IEEE Transactions on*, (3), pp. 195–207.

Gibbons, J., Lester, D. and Bird, R. (2006). Functional pearl: Enumerating the rationals. *Journal of Functional Programming*, 16, pp. 281–291.

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

Gill, A. and Faddegon, M. (2016). Private correspondence.

Graham, R. L., Knuth, D. E. and Patashnik, O. (1989). *Concrete Mathematics: A Foundation for Computer Science*, AIP Publishing, chap. 4. Number Theory. pp. 116–123.

Harper, R. (2013). *Practical foundations for programming languages.*

Hinze, R. (2004). Generics for the masses. In *Proceedings of the International Conference on Functional Programming*, ACM Press.

Hinze, R., Jeuring, J. and Löh, A. (2007). Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, Springer LNCS 4719.

Hinze, R. and Löh, A. (2006). "Scrap your boilerplate" revolutions. In *Mathematics of Program Construction*, Springer LNCS 4014.

Hudak, P. et al. (2007). A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ACM, pp. 12–1.

Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2), pp. 98–107.

Jansson, P. and Jeuring, J. (1997). PolyP—a polytypic programming language extension. In *Proceedings of the symposium on Principles of programming languages*, ACM Press.

Jones, M. P. et al. (1994 – 2004). *The Hugs 98 User's Guide.* url-http://www.haskell.org/haskellwiki/Hugs.

Kiselyov, O. (2006). Smash your boilerplate without class and typeable. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`.

Kozen, D. C. (2012). *Automata and computability.* Springer Science & Business Media.

Lämmel, R. and Peyton Jones, S. (2003). Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, vol. 38, ACM Press.

Lämmel, R. and Peyton Jones, S. (2005). Scrap Your Boilerplate with Class: Extensible Generic Functions. In *Proceedings of the International Conference on Functional Programming*, ACM Press.

Lämmel, R. and Visser, J. (2001). Generic Programming with Strafunski.

Launchbury, J. (1993). A natural semantics for lazy evaluation. In *Proceedings of the symposium on Principles of programming languages*, POPL 1993, pp. 144–154.

Magalhães, J. P. et al. (2010). A Generic Deriving Mechanism for Haskell. In *Proceedings of the Symposium on Haskell*, ACM Press.

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

Marlow, S. et al. (2007). A lightweight interactive debugger for Haskell. In *Proceedings of the Haskell workshop*, pp. 13–24.

Marlow, S. et al. (2010). *Haskell 2010 language report.*

Mitchell, N. and Runciman, C. (2007). Uniform boilerplate and list processing. In *Proceedings of the Haskell workshop*, ACM Press.

Morgan, R. G. and Jarvis, S. A. (1998). Profiling Large-Scale Lazy Functional Programs. *J Funct Program*, 8(3), pp. 201–237.

Naish, L. (1992). *Declarative debugging of lazy functional programs.* Department of Computer Science, University of Melbourne.

Naish, L. (1997a). A declarative debugging scheme. *Journal of Functional and Logic Programming*, 3.

Naish, L. (1997b). A declarative debugging scheme. *Journal of Functional and Logic Programming*, 3.

Nilsson, H. (1998). *Declarative debugging for lazy functional languages.* Ph.D. thesis, Linköpings universitet.

Nilsson, H. and Fritzson, P. (1992). Algorithmic debugging for lazy functional languages. In M. Bruynooghe and M. Wirsing, eds., *Programming Language Implementation and Logic Programming*, LNCS 631, PLILP '92, pp. 385–399.

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

Olaf Chitil (2012). FPretty, `http://hackage.haskell.org/package/FPretty`.

Oliveira, B. C., Hinze, R. and Löh, A. (2006). Extensible and modular generics for the masses. In *Proceedings of Trends in Functional Programming*, Elsevier.

Peyton Jones, S. and Lester, D. (1992). *Implementation Functional Languages: a tutorial.* Prentice Hall.

Peyton Jones, S. L. et al. (2003). *Haskell 98 language and libraries: the revised report.* Cambridge University Press.

Pierce, B. C. (2002). *Types and programming languages.* The MIT Press.

Pope, B. (2005). Declarative Debugging with Buddha. In *Advanced Functional Programming*, LNCS 3622, pp. 273–308.

Pope, B. (2006). *A Declarative Debugger for Haskell.* Ph.D. thesis, The University of Melbourne, Australia.

Pope, B. and Naish, L. (2002). Specialisation of higher-order functions for debugging. *Electronic Notes in Theoretical Computer Science*, 64, pp. 277–291.

Ray, B. et al. (2014). A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM, pp. 155–165.

Reinke, C. (2001). GHood – Graphical Visualisation and Animation of Haskell Object Observations. In *Proceedings of the Haskell Workshop.*

Rodriguez, A. et al. (2008). Comparing Libraries for Generic Programming in Haskell. In *Proceedings of the Symposium on Haskell*, ACM Press.

Runciman, C., Naylor, M. and Lindblad, F. (2008). Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Proc. 1st ACM SIGPLAN Haskell Symposium (Haskell'08)*, ACM, pp. 37–48.

Sansom, P. M. and Peyton Jones, S. L. (1997). Formally based profiling for higher-order functional languages. *ACM Trans Program Lang Syst*, 19(2), pp. 334–385.

Shackell, T. and Runciman, C. (2005). Faster production of redex trails: The Hat G-Machine. In *Proc. 6th Symposium on Trends in Functional Programming (TFP 2005)*, pp. 135–150.

Shapiro, E. Y. (1983). *Algorithmic program debugging.* MIT press.

Sheard, T. and Peyton Jones, S. (2002). Template meta-programming for Haskell. In *Proceedings of the Workshop on Haskell*, ACM Press.

Silva, J. (2007). A comparative Study of Algorithmic Debugging Strategies. In *Logic-Based Program Synthesis and Transformation*, LNCS 4407, pp. 143–159.

Sinclair, D. C. (1992). Debugging by Dataflow—Summary. In *Functional Programming, Glasgow 1991*, Springer, pp. 347–351.

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

Sparud, J. and Runciman, C. (1998). Complete and partial redex trails of functional computations. In *Implementation of Functional Languages*, Springer, pp. 160–177.

Tamarit, S. et al. (2016). *Debugging Meets Testing in Erlang*, Cham: Springer International Publishing. pp. 171–180.

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011).

Wadler, P. (1998a). Functional programming: An angry half-dozen. In *Database Programming Languages*, Springer, pp. 25–34.

Wadler, P. (1998b). Why No One Uses Functional Languages. *SIGPLAN Not*, 33(8), pp. 23–27.

Wallace, M. et al. (2001). Multiple-view tracing for Haskell: a new Hat. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop.*

Winstanley, N. and Meacham, J. (2008). DrIFT Manual. `http://repetae.net/computer/haskell/DrIFT/drift.html`.

Zeller, A. (2009). *Why Programs Fail, 2nd Edition.* Morgan Kaufmann.

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