A Type Generic Definition for Debugging Lazy Functional Programs by Value Observation

Maarten Faddegon and Olaf Chitil

University of Kent, UK

Abstract

Observing intermediate values helps to understand what is going on when your program runs. Gill presented an observation method for lazy functional languages that preserves the program's semantics. However, users need to define for each type how its values are observed: a laborious task and strictness of the program can easily be affected. Here we define how any value can be observed based on the structure of its type by applying generic programming. Furthermore we present an extension to specify per observation point how much to observe of a value.

Keywords: tracing, debugging, lazy evaluation, Haskell

1. Introduction

Consider the 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. A *lazy evaluation* strategy evaluates expressions to values as they are needed: first `gen` runs as little as possible, then `sel` runs and `gen` is suspended until `sel` requires a new input value.

A lazily evaluated program can be modularized into a generator part (`gen` in the example above) that constructs a large number of possible answers and a selector part (`sel` above) that selects the right answer. Well structured code is easier to write because each module becomes simpler, and reduces future programming costs because it provides modules that can be re-used [1].

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 will always still be made, requiring debugging. Consider the defective program in Figure 1. The generator function `mkTree` 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 our `Tree` type also has a `Leaf` constructor, even

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

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

toFloat :: Tree -> Float -> Rational
toFloat (Node x left right) y
 | delta <= 0 = x
 | delta > 0  = toFloat left  y
 | otherwise  = toFloat right y
 where delta = (fromRational x) - y

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

Figure 1: A defective program for finding a rational representation of a floating point number using an sorted infinite tree of rational numbers.

though the tree we define is infinite and has no leafs. The rational $\frac{1}{3}$ (represented as `1 :% 3` in the program) is for example the simplest approximation of 0.3 when the maximum deviation is 0.05. However, when we run our program it prints the unexpected result `3 :% 10` instead. How do we find out if the defect is in the definition of the generator function or in the definition of the selector function?

When we could inspect the intermediate values that are used as inputs for the selector function we 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 we derive `Show` for the `Tree` type and try to reveal the intermediate values with

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

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

Gill presented therefore a method that preserves the semantics for lazy evaluation and reveals only the intermediate values that are actually demanded by the selector function. His approach is implemented in the library HOOD for Haskell. Using Gill's method on the `toFloat` function in our program we obtain the values in Figure 2, where _ is used to represent a part of the tree that is not used in this context of the evaluated fuction. From these recorded values we can use the specification of the Stern-Brocot tree [cite??] to conclude that the part of the tree evaluated in the context of `toFloat` is correct. Hence, the defect is not in the generator function but must be in the selector function `toFloat` and not in the generator function `mkTree`.

1.1. The Problem

We can derive a `Show` instance for the type of a value we want to `print`. However, *how* HOOD observes intermediate values has to be stated for each type

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

Figure 2: Recorded arguments and result values of `toFloat`.

in a specific definition. Because the HOOD library comes with many of these definitions, observing values of common types works well. However when users define their own data types they also need to define how these are observed. There are two reasons why it is undesirable that users need to define instances for their data types:

First of all writing these definitions is a laborious and boring task making HOOD less accessible.

The second and maybe even more important problem is that the strictness of the program can be changed when not enough care is given to the definition. This is bad: non-termination can be introduced by tracing a program.

While considering how to give a generic definition to observe values, we realised that HOOD lacks a mechanism to *not* observe values of a certain or unknown type. Partial observation can be beneficial for two reasons: First of all, when fully observing a function, the formatted output of an observation may be cluttered by values that are not needed to understand the working of the function. Secondly, fully observing a value with a parameterised type requires the addition of class predicates to the type of the function in which the observation is made. The change then may require further type changes or additional instance declarations wherever the function is called. If such changes are required beyond the boundaries of one module, they are too intrusive and thus practically infeasible.

1.2. Contributions

With this in mind we have developed Hoed, an improved version of the HOOD tracing library. Hoed can be installed with `cabal install Hoed`. Some of the features described in this paper are also merged into HOOD version 0.3. In this paper we make the following main contributions:

- We define how any value can be observed based on the structure of its type and generalise HOOD with this definition (Section 4).
- We include an extension that allows us to define in a generic manner, per observation point, how much of a value to observe. (Section 5).

Furthermore, we explain how value observation tracing works (Section 2) and how value observation tracing is implemented in HOOD (Section 3).

2. Value Observation Tracing

To explain what is exactly recorded in the trace by an `observe` annotation we define a small language and with a typed natural semantics we specify how typed expressions from our language are lazily evaluated with generation of a trace. We used a similar, but untyped, language and semantics in our previous publication on constructing a computation tree for algorithmic debugging from

a value observation trace [2], to make this paper self-explanatory we summarize some parts of the language description from our previous paper. For the purpose of explaining our method we 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.

2.1. Language

Figure 3 gives the language for which we define value observation tracing. Our language contains two sorts of values: data constructors and functional values. Integer values are just data constructors with arity 0.

When the programmer wants to record the value to which an expression e evaluates in the trace, they use the `observe` expression to annotate e with a label f. Evaluating an `observe` expression introduces `obs` expressions and $\texttt{obs}_\lambda$ values, `obs` and $\texttt{obs}_\lambda$ should not be part of a program.

2.2. Types

The expressions from our language are typed with the types from Figure 4. There is one base type `Integer`. 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
```

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

Figure 3: Syntax of the core language.

T	$::= T_1 \to T_2$	function type
	$\mid$ Integer	base type
	$\mid T_1 :+: T_2$	sum type, to encode choice
	$\mid T_1 :*: T_2$	product type, to encode structured data
	$\mid$ M c	meta type

Figure 4: Syntax of the types in our language.

is represented in our type syntax as

(M `True`) :+: (M `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

(M `:%`) :*: `Integer` :*: `Integer`

2.3. A Trace of Events

A value is observed by adding events as defined in Figure 5 to the trace, thus the trace is a sequence of events. 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 (P_c 2 2) "`4`" 0
4: Con (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, contain a field p. This field both identifies which event is their parent and what the role of the child event is. For example the parent field P_c 2 1 of event 4 above tells us that event 4 is the first argument to 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 $d =$ `ss` c is the representation of that constructor. A constructor may be the parent of up to a children, each child event with a parent field P_c i m such that $1 \leq m \leq a$. Because lazy evaluation may not evaluation some data constructor arguments the event i : Con p d a does not necessarily have all m children.

We 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 argument value. The argument value is optional, consider for example the program

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

which evaluates to the following trace:

trace	$\mathcal{T}$	$::= t_0, \ldots, t_n$	sequence growing right
event number	i	$\in \{0, \ldots, n\}$	refers to an event in the trace
trace event	t	$::= i{:}\text{Root } f$	root with function identifier f
		$\mid i{:}\text{Con } p\ d\ a$	value is data constructor
		$\mid i{:}\text{Lam } p$	value is an abstraction
		$\mid i{:}\text{MapsTo } p$	function application
parent	p	$::= \text{P } i$	parent is event $i{:}\text{Root } f$ or $i{:}\text{Lam } p'$
		$\mid \text{P}_\text{c}\ i\ m$	argument m; parent is constructor event $i{:}\text{Con } p'\ d\ a$ with $m \leq a$
		$\mid \text{P}_\text{a}\ i$	argument
		$\mid \text{P}_\text{r}\ i$	result
string	s	$::= \text{f}$	user label
		$\mid \text{d}$	data constructor representation

Figure 5: Syntax of the trace and its events.

1: Root "`f`"
2: Lam (P 1)
3: MapsTo (P 2)
4: Con (P_r 3) "`7`" 0

2.4. Semantics

We give a set of evalutation rules in Figure 6 that define how a heap Γ_1, that is a finite list from variables to expressions, a trace $\mathcal{T}_1$ and an expression e are evaluated to Γ_z, $\mathcal{T}_z$ and value v. Expression e and resulting value v have the same type T. For this we use the following notation:

$$\Gamma_1, \mathcal{T}_1 : e \Downarrow \Gamma_z, \mathcal{T}_z : v :: T$$

The initial heap and trace are empty. The resulting trace $\mathcal{T}_z$ 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 rules Lam, Con, Let, Var, App, Case and Prim are similar to the rules with the same names in Launchbury's semantics for lazy evaluation [3]. We added a trace as an additional global state that is passed and possibly changed by the dependencies but not by the rules themselves. Like Launchbury we require that all bound variables of an expression are distinct. The $\hat{v}$ in the Var rule indicates that all bound variables in v are renamed to fresh ones. For y_1 to y_n in ObsCon and y in ObsApp we also pick fresh variables. In the Prim rule $\underline{\oplus}$ is the total semantic function associated with the syntactic operator $\oplus$.

The last four rows define how the trace is constructed. For an application of a constructor $c\ x_1 \ldots x_n$ the ObsCon rule adds a Con event and continues observing the arguments $x_1, \ldots, x_n$ of the constructor using the pseudo function `obs`.

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

$$\Gamma, \mathcal{T} : \lambda x.e \Downarrow \Gamma, \mathcal{T} : \lambda x.e :: T_2 \to T_2 \quad \text{Lam}$$

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

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

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

$$\frac{\Gamma_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{T}_2 : \lambda x.e' :: T_1 \to T_2 \qquad \begin{array}{l}\Gamma_2, \mathcal{T}_2 : e'[y/x] \Downarrow \\ \Gamma_3, \mathcal{T}_3 : v :: T_2\end{array}}{\Gamma_1, \mathcal{T}_1 : e\ y \Downarrow \Gamma_3, \mathcal{T}_3 : v :: T_2}\ \text{App}$$

$$\frac{\begin{array}{ll}\Gamma_1, \mathcal{T}_1 : e \Downarrow \Gamma_2, \mathcal{T}_2 : c_k\ x_1 \ldots x_{m_k} & \Gamma_2, \mathcal{T}_2 : e_k[x_i/y_i]_{i=1}^{m_k} \Downarrow \Gamma_3, \mathcal{T}_3 : v \\ :: T_1 + T_2 + \ldots T_n & :: T_1 + T_2 + \ldots T_n\end{array}}{\Gamma_1, \mathcal{T}_1 : \texttt{case}\ e\ \texttt{of}\ \{c_i\ y_1 \ldots y_{m_i} \to e_i\}_{i=1}^n \Downarrow \Gamma_3, \mathcal{T}_3 : v :: T_1 + T_2 + \ldots T_n}\ \text{Case}$$

$$\frac{\Gamma_1, \mathcal{T}_1 : e_1 \Downarrow \Gamma_2, \mathcal{T}_2 : v_1 :: T_1 \qquad \Gamma_2, \mathcal{T}_2 : e_2 \Downarrow \Gamma_3, \mathcal{T}_3 : v_2 :: T_2}{\Gamma_1, \mathcal{T}_1 : e_1 \oplus e_2 \Downarrow \Gamma_3, \mathcal{T}_3 : v_1 \underline{\oplus} v_2 :: T_3}\ \text{Prim}$$

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

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

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

$$\frac{\begin{array}{ll}\Gamma_1, \mathcal{T}_1 : e \Downarrow & \Gamma_2[y \mapsto \texttt{obs}\ (\text{P}_\text{a}\ i)\ x :: T_1], \\ & \mathcal{T}' \lessdot (i : \text{MapsTo}\ p) : \\ \Gamma_2, \mathcal{T}_2 : \texttt{obs}_\lambda\ p\ v_\lambda :: T_1 \to T_2 & \texttt{obs}\ (\text{P}_\text{r}\ i)\ (v_\lambda\ y) \Downarrow \\ & \Gamma_3, \mathcal{T}_3 : v :: T_2 \qquad i = |\mathcal{T}_2|\end{array}}{\Gamma_1, \mathcal{T}_1 : e\ x :: T_2 \Downarrow \Gamma_3, \mathcal{T}_3 : v :: T_2}\ \text{ObsApp}$$

Figure 6: A typed natural semantics for lazy evaluation with generation of a trace.

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

IO actions such as `getChar` and `putChar` are similar to functions but either the result or the argument is opaque: we record that it is there but we cannot observe its value. For handling them see the original HOOD paper [4].

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

-- The class and method users need to implement
class Observable a where
  observer :: a -> Parent -> a

-- Helper functions to implement an observer method
send :: String -> ObserverM a -> Parent -> a
(<<) :: (Observable a) => ObserverM (a -> b) -> a -> ObserverM b
```

Figure 7: Essential parts of the HOOD API.

3. Implementing Value Observation Tracing in Haskell

HOOD implements value observation tracing in Haskell [4]. HOOD is unobtrusive: it is just a library and requires no changes to the run-time system. Figure 7 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.

Consider adding the following underlined annotations to the example program of Figure 1:

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

We run our program as usual. During the evaluation of the program the `observe` annotation records, as defined in the semantics of Section 2, the event trace listed in Figure 8. Finally the function `runO` formats the collected events and prints the value representation of Figure 2. Note how for example event 7 records an arity of 3 but only has two childs (event 7 and 12) from which we conclude that the expression that describes the right-tree was not demanded in this context, and hence a _ is printed. We describe exactly how a trace of events is translated to a value representation in [2].

A user can define their own type and data constructors, like the `Tree` type with the `Node` and `Leaf` data constructors in the example of Figure 1. Because values of different types need to be observed in different ways, the HOOD library implements the ObsCon rule with a method named `observer`. This method is

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 8: Trace obtained by evaluating introductory example.

part of the class `Observable`. The programmer needs to define instances of the class `Observable` for the types of all values they want to observe.

Let us consider for example evaluating an observed expression e of type `Tree`, by the ObsCon rule, that is either

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

or

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

To implement this behaviour in Haskell the programmer writes an instance of the `observer` method. To add an event to the trace the HOOD library provides the function `send`. The `send` function takes the message to record, the value "wrapped" in the `ObserverM` monad and the context. The `ObserverM` state monad is used to number the constructor arguments of the observed value. Later we take a closer look at numbering constructor arguments and the context, for now it is enough to know that this is used to connect various parts of the observation.

The goal of each `observer` instance is twofold.

First of all the `observer` records a message with a string representation of the data constructor of the value, such as "`Node`" or "`Leaf`". For base types, such as `Integer` the message is the result of `(show lit)`, for base types the `show` function does not change the semantics because the value of has no internal structure, but in general we need to be careful with `show`.

Secondly the `observer` should put further `observer`s on the constructor arguments of the value: for example the `Tree observer` adds an `observer` to all three constructor arguments in our example above, resulting in the second observation when `v` is evaluated.

To place `observer`s on the constructor arguments of a value, the value is decomposed by pattern matching, e.g. into the constructor `Node` and the arguments `v`, `l` and `r` in the `Tree observer` above. From the decomposed parts a transformed value with `observer`s is inserted one level deeper.

The data constructor's arguments may or may not be evaluated, in arbitrary order. To identify which message is associated with which argument *port*

numbers are assigned to each of the arguments' `observer`s. For example when observing the value `Node` x_1 x_2 x_3 the observer of argument x_1 is assigned port number 1, x_2 gets port number 2 and x_3 port number 3.

To assign increasing port numbers to constructor arguments, a state monad `ObserveM` and the function `thunk` are used. When `thunk` is evaluated the appropriate `observer` instance is applied.

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

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)

thunk :: (Observable a) => a -> ObserverM a
thunk a = ObserverM $ \ parent port ->
                (observer a (Parent
                              { observeParent = parent
                              , observePort   = port
                              })
                ,port+1)
```

Using the monad can become rather involved for constructors with many arguments. Therefore the helper function (<<) can be used when defining an observer instance:

```
(<<) :: (Data a) => (Data b)
     => ObserverM (a -> b) -> a -> ObserverM b
fn << a = do fn' <- fn
             a' <- thunk a
             return (fn' a')
```

To write a correct `observer` implementation we need to have some understanding of how lazy evaluation works and have some basic understanding of HOOD's internals. We need to define the method `observer` such that only a representation of the value's data constructor is recorded now, and that other observers will do the same for the data constructor arguments when these are evaluated. The helper function (`<<`) can be used to count and number the constructor arguments of a value and to apply `observer` to each argument: To observe the tree of our example the definition would be:

```
instance Observable Tree where
 observer (Node x1 x2 x3)
  = send "Node" (return Node << x1 << x2 << x3)
 observer Leaf
  = send "Leaf" (return Leaf)
```

4. Type Generic Value Observation Tracing

A programmer can easily change the lazy semantics of `observe` with the definition of their `observer` instance. For example using `show` on the arguments of the constructor can result in a non-terminating program:

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

A second problem of hand written `observer` instances is that the repetitive task of writing observers is a burden to the programmer. Furthermore the observers are vulnerable to change, when debugging a program we are likely to want to make small changes to our data structures which require us to adapt our observers as well.

To make HOOD easier to use and less prone to misuse we extended HOOD allowing the user to derive how a value is observed from its type. Data generic programming techniques are a well researched area resulting in a multitude of libraries and language extensions. A fairly complete overview is given in [5].

The *Generic Deriving Mechanism* (GDM) adds the derivable class `Generic` with methods to convert to and from a type representation. A generic function is defined on this type representation [6]. The Glasgow Haskell compiler[1] (GHC) and the Utrecht Haskell compiler[2] (UHC) implement GDM.

With GDM we define how `observer` can be derived from a type representation. This representation is defined for instances of the `Generic` class. The `Generic` class is derivable:

```
data Tree = Node Rational Tree Tree | Leaf
  deriving (Generic)
```

To derive an `observer` instance users add an `Observable` instance declaration for their type without a definition of the method. Either as a standalone declaration (e.g. when the type was defined in a library):

```
instance Observable Tree
```

Or the instance is derived in the type declaration:

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

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.

4.1. Implementation

For the Generic Deriving framework we provide a default implementation of observer:

```
class Observable a where
        observer         :: a -> Parent -> a
        default observer :: (Generic a, GObservable (Rep a))
                         => a -> Parent -> a
        observer x c     = ...
```

[1] `http://www.haskell.org/ghc`
[2] `http://www.cs.uu.nl/wiki/UHC`

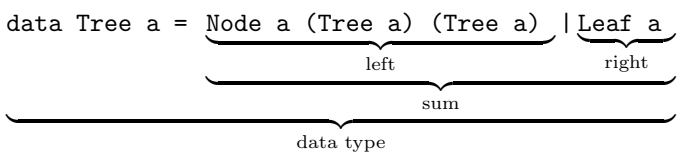


Figure 9: Choice between data constructors of the `Tree` type encoded as the sum of `Node` and `Leaf`.

A type generic function is implemented with the *Generic Deriving Mechanism* (GDM) by converting the observed value to a product-sum representation, manipulating this representation and converting back from the changed representation. To convert a value into a type representation its type should be of the `Generic` class, which is derivable [6].

The product-sum representation has its roots in type theory: representing the choice between constructors (e.g. `Node` or `Leaf` for values of type `Tree`) as the product of the choices and representing a variant type (e.g. `Rational`, `Tree` and `Tree` for the `Node` constructor) as the sum of its variants.

4.1.1. Encoding 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 we derive `Generics` for a type. We would for example for our `Tree` generate the types `NodeConstr` and `LeafConstr` such that:

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

4.1.2. Encoding Product and Sum

Here we summarise the product-sum representation[3] as used in GDM:

- To encode choice between data constructors of the same type GDM uses the sum type. When there are more than two constructors, the sum type can be nested.

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

[3] We simplified the actual representation of GDM, the full representation is presented by Magalhães et al. in [6].

- To encode structured data the product representation is used.

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

Let us consider how a value of the `Tree` type would be encoded. A value with constructor `Node` has three arguments, this is encoded with the product-representation. Our `Tree` type can either be `Node` or `Leaf` (see Figure 9), the choice between these data constructors is encoded with `L1` for `Node`-values and `R1` for `Leaf`-values. For example assume we want to encode a simple tree with two leafs and one node. The values `x`, `y` and `z` are stored in the tree. We do not elaborate on how these are encoded but just label their representations as `q`, `r` and `s`:

```
encode (Node x (Leaf y) (Leaf z))
  ↦ L1 (M1 (q :*:  R1 (M1 r) :*:  R1 (M1 s)))
```

$$\texttt{shallowShow}\ () = \epsilon$$

$$\texttt{observeChildren}\ () = ()$$

Records are another special case of the tuple type where a set of field labels index the tuple [7]. Currently HOOD does not trace field labels, but with our approach it would be trivial to extend it to do so. The definitions below reflect the current situation:

OC: Explain what $L_F : T$ is

$$\texttt{shallowShow}\ x :: L_F : T = \epsilon$$

$$\texttt{observeChildren}\ x :: L_F : T = \texttt{observer}\ x :: T$$

4.1.3. Base types

Base types such as Int and Float "have no internal structure as far as the type system is concerned" [8] Therefore we can use `show` to produce the representation without forcing further evalution:

$$\texttt{shallowShow} = \texttt{show}$$

And there are no further arguments to be observed:

$$\texttt{observeChildren}\ b = b$$

4.1.4. Implementing a Generic Observer with GDM

For each value that we want to observe with our generic observer we use GDM's `from`-function to construct a product-sum representation. Above we introduced GDM's fixed set of types in which it represents a `Generic` value.

We introduce a class `GObservable` with method `gobserver` and for each of GDM's representation-types we define an instance of `GObservable`: with the sum representation we query the meta-information; using the meta information we find the constructor names and record these; and with the product representation we observe the arguments of the value.

The observer applied to one of the arguments can either be another ad-hoc instance of `observer` provided by the programmer, or again the default

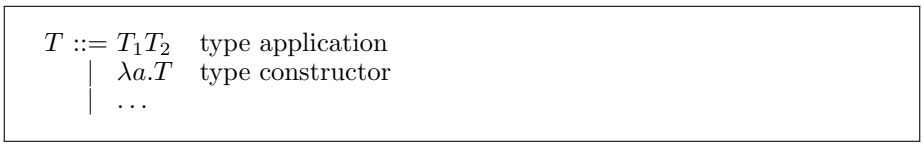


Figure 10: Type constructors and applications.

`observer`. The returned type representation (with observed constructor arguments) is decoded to the original type with GDM's `to`-function. Figure **??** shows a schematic overview of applying the generic `observer` to the type representation of a `Node` from our `Tree`.

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

5. Type Constructors and Value Observation Tracing

In the first part of this paper we ignored type constructors (Figure 10), however, type constructors are an essential part of writing polymorphic library functions.

5.1. Introducing the Case Study

Consider the tree library of Figure 11 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 we can define such a function polymorphically. Instead of an actual type we 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 we use our library to find the list of possible outcomes of flipping a coin n times. When we flip a coin once it can be heads or tails, we represent that with the following data type:

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

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

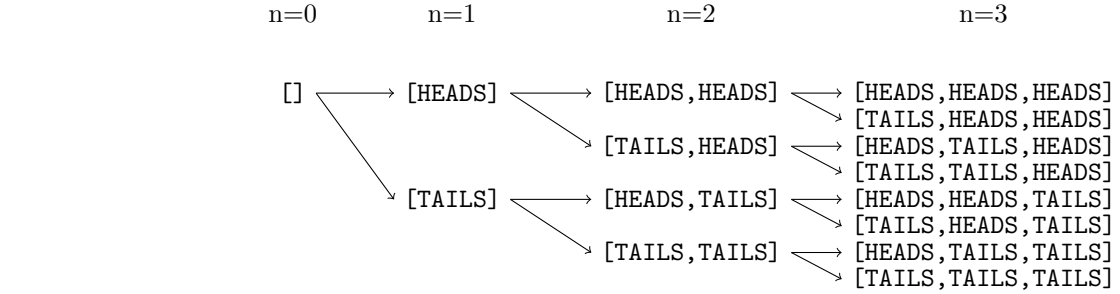
```
[Tails,Heads]
```

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

```
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) = []
```

Figure 11: 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.

states: `Heads` : s_i and `Tails` : s_i. The states of up to three coin-flips can be represented in a tree as follows:



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 we want to inspect the argument and result value of the

applications of `breadFirst`. This will reveal the structure of the tree visited (helping us to decide if the definition of `breadFirst` 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 our `Tree` type in our 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 `observer` method expects `a` in `Tree a` to be `Observable`. Thus the type declaration becomes

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

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

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

4. But now the changed type declaration is exposed to modules outside our 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 `Type a` to which `depth`. In our 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, we rather not force the programmer to make changes outside a library module.

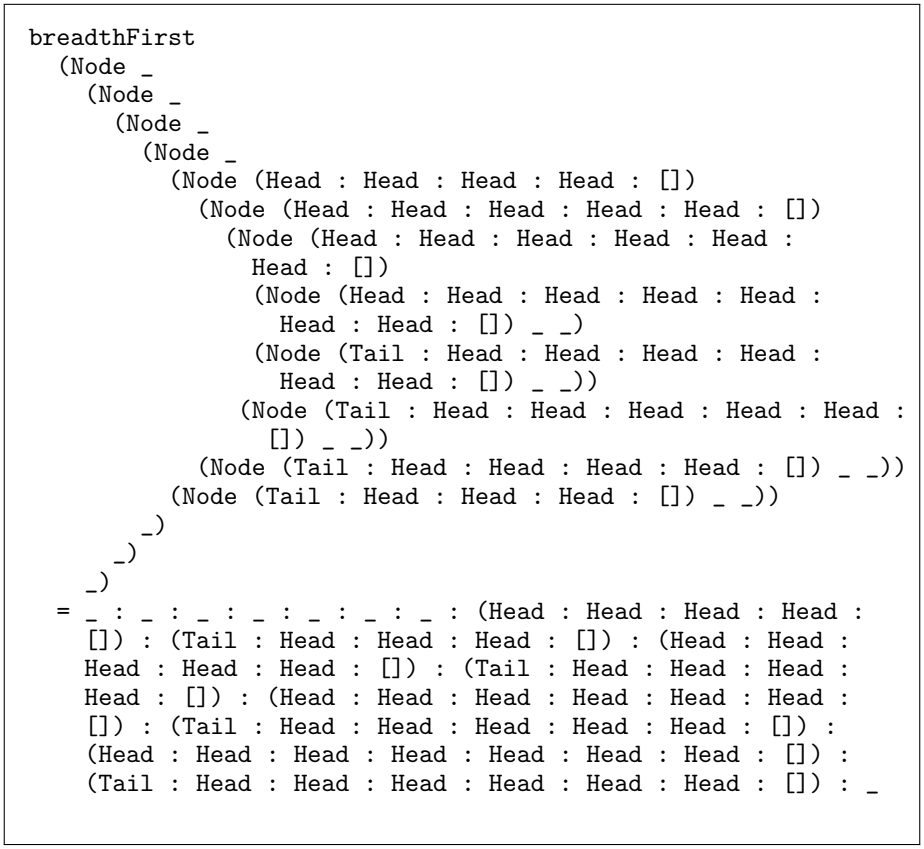
Furthermore, the now also traced coin-flip states do not provide much information to the programmer to understand their code, in fact they clutter the output printed after evaluating the program (Figure 12). What if we used `<?>` for values in the tree and _ for expressions unevaluated in this context? Compare the value representation on the left in Figure 13 with the value representation of Figure 12.

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
```



```
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 : []) :
    (Tail : Head : Head : Head : Head : Head : []) : _
```

Figure 12: Observing a defective breadth first implementation.

5.2. Type Generic Programming is Not Enough

Our definition should decide if a subvalue should be observed based on whether the type of a value is an instance of the `Observable` class. Given the type representations

```
Node a (Tree a) (Tree a)

Observable b => Node b (Tree b) (Tree b)
```

we need a function `instanceOf` such that `instaceOf a Observable = False` and `instaceOf b Observable = True`. To our knowledge there is no type generic programming framework for Haskell that allows this. Hence we resort to meta-programming.

Type generic programming is type-safe after compiling the library with the type generic definitions; meta-programming is type-safe after compiling the program that uses the meta-language annotations.

5.3. Using Meta-Language Annotations

We define a template to generate `Observable` instances from a type (Figure 14 top). The user can apply a template to a type and "splice" the result

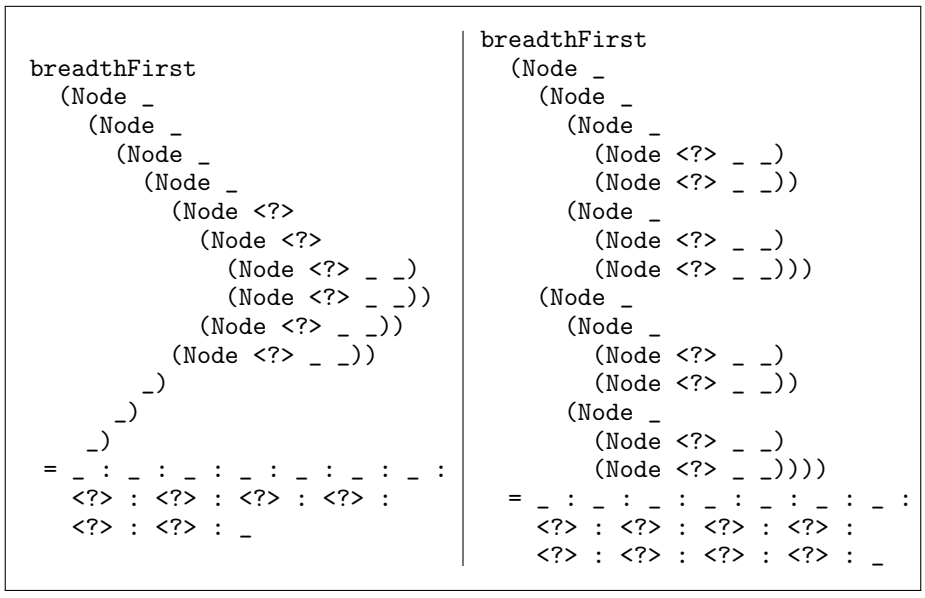


Figure 13: Observed tree data structure with unobserved elements of a defective implementation of breadth first search (on the left) and a sound implementation (on the right).

```
-- Our first implementation for the Template Haskell framework
-- provides a template to generate instances of Observable:
gobservableInstance :: Q Type -> Q [Dec]

-- Our second implementation for the Template Haskell framework
-- provides two templates. The first to specify which types
-- should be observed, and the second to observe a value:
observedTypes :: String -> [Q Type] -> Q [Dec]
observe       :: String -> Q Exp
```

Figure 14: Annotations to generate code for value observation tracing.

into the code under observation:

```
$(gobservableInstance [t| forall a . Tree a |])
```

Because our template offers just a way of generating code, it is again possible for advanced users to define their own `Observable` instances.

To partially observe a value we generate custom implementations of the whole observe mechanism to allow the user to specify per `observe`-annotation values of which types should and should not be observed with TH. In that case we need to add two sorts of annotations to the code under observation (Figure 14 bottom). First of all, for each observation point we make a list of types whose values we want to be observed. Parametrised constructor arguments are observed when we add an `Observable` class predicate for the type variable.

We associate each list with the label of an observation point. Secondly we add an `observe` call with the same label. The label doubles as identifier to find the list of types to be observed and to annotate the formatted output of the observation. Thus to generate Figure 12 the programmer uses the annotation

```
$(observedTypes "breadthFirst"
  [ [t| forall a . Observable a => Tree a|],
    [t| forall a . Observable a =>[a] |] ])
```

and for the output of Figure 13-left the annotation the programmer uses the annotation

```
$(observedTypes "breadthFirst"
  [ [t| forall a . Tree a|],
    [t| forall   a . [a] |] ])
```

instead. The `observe` and `observedTypes` annotations use the splice syntax from TH but are otherwise not heavier than the annotations we used previously.

5.4. Implementing Observer Templates

We define a type generic function in *Template Haskell* (TH) by defining a template that takes a type as argument to construct a type-specific function at compile-time.

We describe a template that from a type constructs an instance of the `Observable` class and thereby defines how values of that type are observed. We again follow the by now well known pattern of first defining templates to construct a shallow representation and afterwards define observation of child values.

5.4.1. TH Syntax

From templates we construct code that is spliced into our program at compile time. We define a template using either quasi-quote brackets (e.g. `[|thunk|]`) or directly using constructors from the TH library (e.g. `VarE thunk`). We can use ordinary Haskell code to combine and manipulate the templates.

With the splice notation (e.g. `$(gobservableInstance [t|MyData|])`) we construct and inject code into our program at compile time. Splicing code is not restricted to the top-level but can also be done from within templates. For a more comprehensive explanation we refer the interested reader to [9].

5.4.2. shallowShow

Our TH implementation of `shallowShow` operates on the type-representation to obtain the constructor name. This is similar to our GDM definition. However unlike the GDM definition we do not return the `String` itself but rather an expression-representation of the `String`. The expression-representation is evaluated at compile time and spliced as a snippet of code into the Haskell program.

```
shallowShow :: Con -> Q Exp
shallowShow (NormalC name _) = stringE (nameBase name)
```

5.4.3. observeChildren

We define the `observerChildren` template in a way that is syntactically close to the earlier type generic definition: we apply `thunk` to all constructor arguments with a generic monadic map `gmapM`. [4]

```
observeChildren :: Con -> [Q Exp] -> Q Exp
observeChildren = gmapM [| thunk |]
```

This function takes an expression and constructor as argument, and applies the expression to the fields in the constructor.

```
gmapM :: Q Exp -> Con -> [Q Exp] -> Q Exp
gmapM f (NormalC name fields) vars
  = let m []     = [| return $(conE name)        |]
        m (v:vs) = [| compositionM $f $(m vs) $v |]
    in m (reverse vars)
```

5.4.4. observer

With `shallowShow` and `observeChildren` we now can implement `observer`. We generate the code for a class instance of `Observable` with TH. Types often have multiple data constructors. The `gobserverClauses` template generates an implementation of `observer` for each constructor of the given type.

```
gobserver :: Q Type -> Q [Dec]
gobserver t = do cs <- gobserverClauses t
                 return [FunD (mkName "observer") cs]
```

The body of a clause is a familiar pattern by now and uses a template named `gobserverBody`. However, our `gobserverBody` template requires a list of variable bindings `evars`. This is needed, because with TH we do not operate on a value representation, but generate actual Haskell code.

```
gobserverBody :: TyVarMap -> Con -> Q Exp -> [Q Exp] -> Q Body
gobserverBody tvm y c evars = normalB
  [| send $(shallowShow y) $(observeChildren tvm y evars) $c |]
```

5.5. Implementing Partial Observe from Template

The function type constructor has kind `* -> *`. A function is observed by collecting the argument-result pairs of its applications.

Up to now we assumed that all constructor arguments of an observed type are observable. In Section **??** we already gave reasons for sometimes desiring *not* to observe constructor arguments of a certain type or type variable. In this section we first explain how to generate customised partial `observe` functions, `observer` methods and `Observable` classes from template, then we discuss why we cannot provide a similar implementation with GDM or SYB.

In the previous section we generated an `observer` method instance by applying a template to a type. Now we want to be able to specify per observe

[4] See `http://github.com/MaartenFaddegon/Hoed` for full implementation.

annotation which constructor arguments of a value are observed. We define two templates:

First of all the `observedTypes` template, which takes a list of types into whose values an observation should descent. The template can be used more than once to make several different observations. This is possible, because the template generates a new "`Observable`"-like class, a set of "`observer`"-like instances and a new "`observe`"-like function.

Secondly the `observe` template is used to insert the appropriate "`observe`"-like function. The desired "`observe`"-like function is selected using the identifier that is passed both to the `observedTypes` and `observe` template. This identifier is also used to annotate the formatted output of the observation.

The templates we used before can be re-used here to implement the `observedTypes` template but instead of unconditionally applying `thunk` to all constructor arguments we need to choose between `thunk` to continue tracing deeper, or `nothunk` to stop tracing.

We introduce the `nothunk` function to distinguish constructor arguments that are evaluated but not traced and constructor arguments that are not evaluated.

```
observeChildren :: TyVarMap -> Con -> [Q Exp] -> Q Exp
observeChildren bs = gmapM (thunkObservable bs)

thunkObservable :: TyVarMap -> Type -> Q Exp
thunkObservable tvmap t
  = if isObservable tvmap t then [| thunk |] else [| nothunk |]

 if isObservable type then [| thunk |] else [| nothunk |]
```

To determine if a type is observable we identify two cases: if it is a type variable we check if the user added an `Observable` class predicate to the type. Otherwise we check if an instance of our custom class for the type exists. Both SYB and GDM lack the ability to perform these tests, we can therefore only give a TH implementation of this extension.

With GDM and SYB it is possible to derive functions that observe parts of a value based on the type of its constructor argument. However there is no mechanism to generate new class declarations with instances. Thus with these frameworks we would need to provide type descriptions or a set of functions to every `observe` application. Previous research has shown that this approach gives problems with values of polymorphic types [10].

6. Related Work

Much work was done before on tracing lazy functional languages and generic programming without which our work would not have been possible.

6.1. Tracing

Previous work on tracing Haskell provides a rich set of information but has seen limited use because systems such as Freja[11], Hat[12] and Buddha [13] require instrumentation of the whole program, including libraries, and are implemented only for subsets of Haskell [14].

With HOOD, Gill made tracing accessible to a larger set of users by presenting a portable library of tracing combinators. To deal with the `Observable` class restriction, users are required to understand lazy evaluation and how HOOD's internals work.

The Haskell interpreter Hugs[5] keeps a type-representation of all values during runtime. Hence Hugs provides a variant of Hood called HugsHood which allows observation of all values without class restriction through type reflection [15]. 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 [10]. HugsHood also extends Hood with an interesting "breakpoint" feature that shows the development of observations over time.

GHood extends HOOD with a graphical representation of the observation showing development over time [16].

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 (see Section 5). 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 values are observed [10]. 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.

GHC and most other Haskell compilers come with the `trace` primitive which allows users to print strings from otherwise pure functions. Compared to the `trace` function HOOD has three benefits: the given implementation does not change the strictness of things it is observing, produced trace output is more readable because of post-processing, and inserting the library combinators in Haskell code tends to be less invasive [4].

6.2. Generic Programming Frameworks

In this paper we discuss and compare the implementation of type generic observations with Scrap Your Boilerplate, Generic Deriving Mechanism and Template Haskell.

Previously Hinze et al. [5] did a much broader comparison of approaches to generic programming, and Rodriguez et al. [17] defined a generic programming benchmark to compare 9 generic programming libraries. Both were valuable sources of information for writing this paper. Our comparison is more modest in the sense that we only compare three approaches. Our contribution however is that we add two criteria of comparison derived from a real world application that previously were not, or not high on the agenda:

1. Define a generic function's behaviour based on class membership of the type of its argument.
2. Define a generic function over a functional value in terms of the applications of that functional value.

With the Scrap Your Boilerplate With Class approach and the Smash Your Boilerplate variant we can reintroduce the `Observable` class in our second implementation: using a dictionary we can explicitly define a default `observer`

[5] `http://www.haskell.org/haskellwiki/Hugs`

instance of `Data` types [18, 19]. We can provide a specific instance for function types, and advanced users can also again define their own instances.

The Uniplate and Strafunsky libraries are variations on SYB offering different interfaces but neither allows mapping over more types compared to SYB [20, 21].

The Generics for the Masses approach is captured completely in Haskell 98. Because the class for generics needs to be adapted for each new type this approach is not suitable to implement a type generic `observer` method [22, 18]. Later work addressed this problem at the cost of introducing boilerplate code that was not in the original approach [23].

The lifted spine view allows representation of data constructors as well as type constructors. Unlike TH we cannot infer if a type is of a certain class, or if a type variable has a class predicate [24].

PolyP is an extension of 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 [25].

DrIFT allows the programmer to add directives to the program which create code from rules defined in a separate file [26]. DrIFTs 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 [5].

7. Conclusions and Future Work

In this paper we show how to overcome the restriction of hand-written `Observable` instances for datatypes of values that we want to observe. Furthermore we present a method to observe up to a certain data type or type variable, which makes HOOD easier to use in libraries and testing frameworks.

We implemented our idea with generic programming techniques and with a meta-language.

Specifying per `observe` which types are observed currently requires the power of a meta-language. With our partial-observe extension we explored a new domain of generic programming. We show that class membership testing, ignored in most previous work, deserves a dedicated study to guarantee type correctness to the writer of a generic library.

Typechecking our `Observable` templates gives no guarantee that correct code is produced under all circumstances. An error will be caught when the user of our library typechecks their code, but this is a much weaker guarantee compared to using a type generic programming framework [5].

Tracing lazy functional programs has seen much research in the past. It produced very informative systems with a high use barrier on the one hand and lightweight systems that provide less information on the other hand. Our contribution extends the out-of-the-box applicability of HOOD to a wider range of types. We however do not address the wide gap between the information provided by systems such as HAT compared to the information provided by HOOD; this calls for research on closing this gap while maintaining HOOD's ease-of-use.

References

[1] J. Hughes, Why functional programming matters, The computer journal 32 (2) (1989) 98–107.

[2] M. Faddegon, O. Chitil, Lightweight Computation Tree Tracing, in: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, 2016.

[3] J. Launchbury, A natural semantics for lazy evaluation, in: Proceedings of the symposium on Principles of programming languages, 1993, pp. 144–154.

[4] ACM SIGPLAN Workshop on Haskell.

[5] R. Hinze, J. Jeuring, A. Löh, Comparing approaches to generic programming in Haskell, in: Datatype-Generic Programming, Springer LNCS 4719, 2007.

[6] J. P. Magalhães, A. Dijkstra, J. Jeuring, A. Löh, A Generic Deriving Mechanism for Haskell, in: Proceedings of the Symposium on Haskell, ACM Press, 2010.

[7] R. Harper, Practical foundations for programming languages, 2013.

[8] B. C. Pierce, Types and programming languages, The MIT Press, 2002.

[9] T. Sheard, S. P. Jones, Template meta-programming for Haskell, in: Proceedings of the Workshop on Haskell, ACM Press, 2002.

[10] B. Braßel, O. Chitil, M. Hanus, F. Huch, Observing functional logic computations, in: Practical Aspects of Declarative Languages, Springer LNCS 3057, 2004.

[11] H. Nilsson, Declarative debugging for lazy functional languages, Ph.D. thesis, Linköpings universitet (1998).

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

[13] B. Pope, Declarative Debugging with Buddha, in: Advanced Functional Programming, Springer LNCS 3622, 2005, pp. 273–308.

[14] O. Chitil, C. Runciman, M. Wallace, Freja, Hat and Hood — a comparative evaluation of three systems for tracing and debugging lazy functional programs, in: Implementation of Functional Languages 2000, Springer LNCS 2011, 2001.

[15] M. P. Jones, A. Reid, T. Y. H. Group, the OGI School of Science & Engineering, The Hugs 98 User's Guide, `http://www.haskell.org/haskellwiki/Hugs` (1994–2004).

[16] C. Reinke, GHood – Graphical Visualisation and Animation of Haskell Object Observations, in: Proceedings of the Haskell Workshop, 2001.

[17] A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, B. C. d. S. Oliveira, Comparing Libraries for Generic Programming in Haskell, in: Proceedings of the Symposium on Haskell, ACM Press, 2008. doi:10.1145/1411286.1411301.
URL http://doi.acm.org/10.1145/1411286.1411301

[18] R. Lämmel, S. P. Jones, Scrap Your Boilerplate with Class: Extensible Generic Functions, in: Proceedings of the International Conference on Functional Programming, ACM Press, 2005.

[19] O. Kiselyov, Smash your boilerplate without class and typeable, http://article.gmane.org/gmane.comp.lang.haskell.general/14086 (2006).

[20] N. Mitchell, C. Runciman, Uniform boilerplate and list processing, in: Proceedings of the Haskell workshop, ACM Press, 2007.

[21] R. Lämmel, J. Visser, A Strafunski Application Letter, in: Practical Aspects of Declarative Languages, Springer LNCS 2562, 2003.

[22] R. Hinze, Generics for the masses, in: Proceedings of the International Conference on Functional Programming, ACM Press, 2004.

[23] B. C. Oliveira, R. Hinze, A. Löh, Extensible and modular generics for the masses, in: Proceedings of Trends in Functional Programming, Elsevier, 2006.

[24] R. Hinze, A. Löh, "Scrap your boilerplate" revolutions, in: Mathematics of Program Construction, Springer LNCS 4014, 2006.

[25] P. Jansson, J. Jeuring, PolyP — a polytypic programming language extension, in: Proceedings of the symposium on Principles of programming languages, ACM Press, 1997.

[26] N. Winstanley, J. Meacham, DrIFT Manual, http://repetae.net/computer/haskell/DrIFT/drift.html (2008).