Type Generic Observing

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 frameworks. Furthermore we present an extension to specify per observation point how much to observe of a value. We discuss especially functional values and behaviour based on class membership in generic programming frameworks.

1 Introduction

Tracing intermediate values helps to understand what is going on in your program. For lazy functional languages Gill presented a method to observe values in a lightweight manner while preserving semantics [3]. His approach is implemented in the library HOOD for Haskell. However, *how* values are observed has to be stated for each type 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. In Section 2.2 we discuss these issues in detail.

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.

With this in mind we have developed Hoed, an improved version of the HOOD tracing library. Hoed can be installed with `cabal update; cabal install Hoed`. In this paper we make the following 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 use our definition as case study to compare the three type-generic programming frameworks: Generic Deriving Mechanism (Section 4.1), Scrap Your Boilerplate (Section 4.2) and Template Haskell (Section 4.3).
– We include an extension that allows us to define in a generic manner, per observation point, how much of a value to observe. (Section 5.1).
– Our novel application of generic programming demonstrates that there is a real demand for the support of higher kinded types in generic programming frameworks and that the current approaches leave room for future research (Section 5).

2 A Closer Look at HOOD

HOOD is a library to observe the evaluated parts of intermediate values [3]. 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. Adding observations doesn't change the semantics of a program. The `runO` function formats and prints observed values. Here is a typical usage of these combinators:

```
main = (runO . putStrLn . show)
     $ floatToRational 0.6 (observe "sternbrocot" sternbrocot)
```

Our program converts floating point value 0.6 into rational value 3/5. Our solution is composed of two parts: the first part constructs an infinite ordered binary tree with all rational values, in the second part we descend into the tree finding better and better approximations until a rational value that is equal to our floating point value is found. For completeness our `Tree` type also has a `Leaf` constructor, even though the tree we define is infinite and has no leafs.

```
-- API of Stern-Brocot library, implementation omitted in paper
data Tree a  = Node a (Tree a) (Tree a) | Leaf a
sternbrocot     :: Tree Rational
floatToRational :: Float -> Tree Rational -> Rational
```

We would like to know how the value 3/5 is computed and what the values of the intermediate results are. We cannot do `print sternbrocot` or `let s = sternbrocot in trace (show s) s` because this would force evaluation of the whole infinite tree. With HOOD we can observe a value and only print those parts that have been evaluated. After termination of the program, `runO` prints the representation of the observed value:

```
Node (Rational 1 1)
     (Node (Rational 1 2) _
       (Node (Rational 2 3) (Node (Rational 3 5) _ _) _ )
     )
     _
```

```
-- 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
```

Fig. 1: Essential parts of the HOOD API.

2.1 Defining How Values Are Observed

Values of different types need to be observed in different ways. For this purpose the function `observe` uses the `observer` method. The `observer` method is part of the class `Observable`. We need to define instances of the class `Observable` for the types of all values that we want to observe.

To implement our own `observer`, 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 components of the observed value. Later we take a closer look at numbering components and the context, for now it is enough to know that this is used to connect various parts of the observation.

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 shallow representation of the value is recorded now, and that other observers will do the same for the components of the value when these are evaluated. The helper function (`<<`) can be used to count and number the components of a value and to apply `observer` to each component: To observe the tree of our example the definition would be:

```
observer (Node x t1 t2)
  = send "Node" (return Node << x << t1 << t2)
```

A user can easily change the lazy semantics of `observe` with their own `observer` instance. For example using `show` on the components of the type can result in a non-terminating program:

```
observer n@(Node x t1 t2)
  = send (show x  ++ ", " ++ show t1 ++ show t2) (return n)
```

2.2 How HOOD Works

Before we discuss how a generic `observer` works, we begin with a brief overview of the `observer` mechanism in HOOD. Assume we want to observe a value such

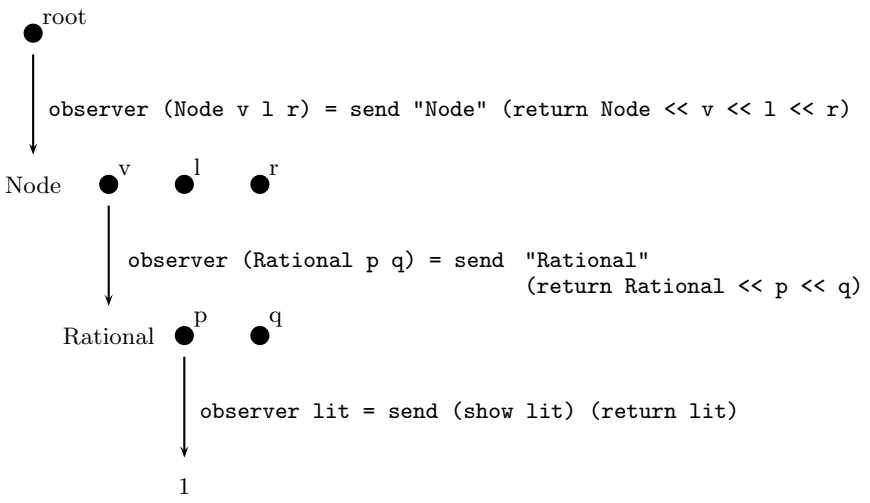

Fig. 2: Application of observer instances when our Stern-Brocot tree is evaluated.

as the `sternbrocot` tree. The diagram of Figure 2 illustrates three steps in an example evaluation. Each arrow represents an evaluation step, at each of these steps an `observer` method emits a message. Next to each arrow the code of the corresponding `observer` instance is shown.

The goal of each `observer` instance is twofold, as outlined in Figure 3.

First of all the `observer` records a message with a *shallow representation* of the value. In the three instances above the representations are "`Node`", "`Rational`" and the result of `(show lit)`. The `show` function in the last example does not change the semantics because the value of `lit` has no internal structure, but in general we need to be careful with `show`, because the representation it produces is often not shallow.

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

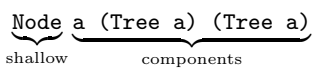

Fig. 3: Example of decomposing a value into a shallow part under observation now, and its components which are observed later (if evaluated).

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

Components may or may not be evaluated, in arbitrary order. We assign "port" numbers to each of the observers e.g. 0 for '`v`', 1 for '`l`' and 2 for '`r`' to identify which message is associated with which field in the data constructor. These numbers are stored in the `Parent` type

```
data Parent = Parent { observeParent :: Int, observePort :: Int}
```

To assign increasing port numbers to components, a state monad `ObserveM` and the function `thunk` are used. When `thunk` is evaluated the appropriate `observer` instance is applied. The `(<<)` function, used for hand written instances, applies `thunk` to the argument on the right.

```
thunk :: (Observable a) => a -> ObserverM a
fn << a = do { fn' <- fn ; a' <- thunk a ; return (fn' a') }
```

3 Using Type Generic HOOD

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].

3.1 Three Data Generic Frameworks

We use the following frameworks:

- 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 [14].
- *Scrap Your Boilerplate* (SYB) adds the derivable classes `Typeable` and `Data` with methods to map over subtypes. Generic functions are defined on types of the `Data` class [10].
- *Template Haskell* (TH) is a language extension that allows us to define functions on a meta-level that are evaluated at compile-time and construct code from types, functions or other expressions [21].

Each framework relies on some language extension and additional libraries. The Glasgow Haskell compiler[1] (GHC) implements all three frameworks; GDM is also provided by the Utrecht Haskell compiler[2] (UHC).

With the three frameworks we give give four implementations (see Figure 4). Three alternative implementations extending HOOD with the ability to derive how values are observed from their types. A fourth implementation, in TH only, additionally allows us to define up to which type a value is observed.

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

```
-- 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     = ...

-- For the Scrap Your Boilerplate framework we define an observe method
-- for all types of the (derivable) class Data:
observe :: (Data a) => String -> a -> a

-- 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
```

Fig. 4: API of our four type-generic HOOD extensions.

Generic Deriving Mechanism 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 a = Node a (Tree a) (Tree a) | Leaf a deriving (Generic)
```

To derive an `observer` instance users add an `Observable` instance declaration for their type without a definition of the method:

```
instance (Observable a) => Observable (Tree a)
```

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.

Scrap Your Boilerplate With SYB we define an `observer` method for values from types of the `Data` class. This class is defined for types of the `Typeable` class. Both can be derived:

```
data Tree a = Node a (Tree a) (Tree a) | Leaf a
  deriving (Typeable, Data)
```

Note that this approach makes it impossible to define any ad-hoc instances that describe how a value should be observed. In Section 5 we discuss that this causes problems for types for which `Data` instances are difficult to define.

```
$(observedTypes "sternbrocot1"
  [ [t| forall a . Observable a => Tree a |], [t| Rational |] ])
$(observedTypes "sternbrocot2" [[t| forall a . Tree a |]])
```

(a) In the same program we specify per identifier (e.g. `"sternbrocot1"`) which types are to be observed.

```
f1 = floatToRational 0.6 ($(observe "sternbrocot1") sternbrocot)
f2 = floatToRational 0.6 ($(observe "sternbrocot2") sternbrocot)
```

(b) We use (almost) the same `observe` annotation as we did before. But in each case values of different types are observed depending on the specification above.

```
-- sternbrocot1
  Node
    (Rational 1 1)
    (Node (Rational 1 2) _ (Node (Rational 2 3)
                                 (Node (Rational 3 5) _ _)
                                  _))
-- sternbrocot2
  Node <?> (Node <?> _ (Node <?> (Node <?> _ _) _)) _
```

(c) Formatted output from the two example observations above. The symbol "`<?>`" indicates an evaluated but not observed component.

Fig. 5: Specifying how much of the Stern-Brocot tree we want to observe.

Template Haskell We define a template to generate `Observable` instances from a type. The user can apply a template to a type and "splice" the result 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.

3.2 Partial Observations

We explained in the Introduction that there are situations where we want to observe parts of a value. With TH 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.

We need to add two sorts of annotations to the code under observation. First of all, for each observation point we make a list of types whose values we want to be observed. Parametrised components are observed when we add an `Observable` class predicate for the type variable (Figure 5a). 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 (Figure 5c).

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

4 Three Type Generic Implementations

Now we discuss alternative type generic definitions of the `observer` function. Ideally this function would be applicable to values of any type (as per type signature below), in practice we still need some (derivable) class predicates.

```
observer :: a -> Parent -> a
```

For all our solutions we decompose the behaviour of `observer` into three parts: render a shallow representation of the value, as a side-effect record this representation, and observe components of the value.

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

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

```
shallowShow     :: a -> String
observeChildren :: a -> ObserverM a
```

4.1 Generic Derived Observers

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 [14].

The product-sum representation has its roots in type theory: representing a tuple or a record as the product of its components, and representing a variant type (e.g. `Node` and `Leaf` in `Tree`) as the sum of its variants.

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"
```

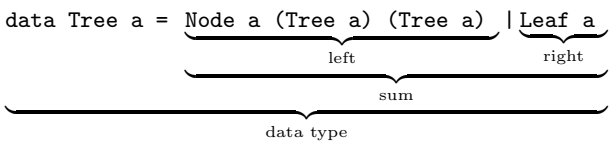

Fig. 6: Choice between data constructors of the `Tree` type encoded as the sum of `Node` and `Leaf`.

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`
- 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 components, this is encoded with the product-representation. Our `Tree` type can either be `Node` or `Leaf` (see Figure 6), 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)))
```

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 components of the value.

The observer applied to one of the components can either be another ad-hoc instance of `observer` provided by the programmer, or again the default `observer`. The returned type representation (with observed components) is decoded to the original type with GDM's `to`-function. Figure 7 shows a schematic overview of applying the generic `observer` to the type representation of a `Node` from our `Tree`.

[3] We simplified the actual representation of GDM, the full representation is presented by Magalhães et al. in [14].

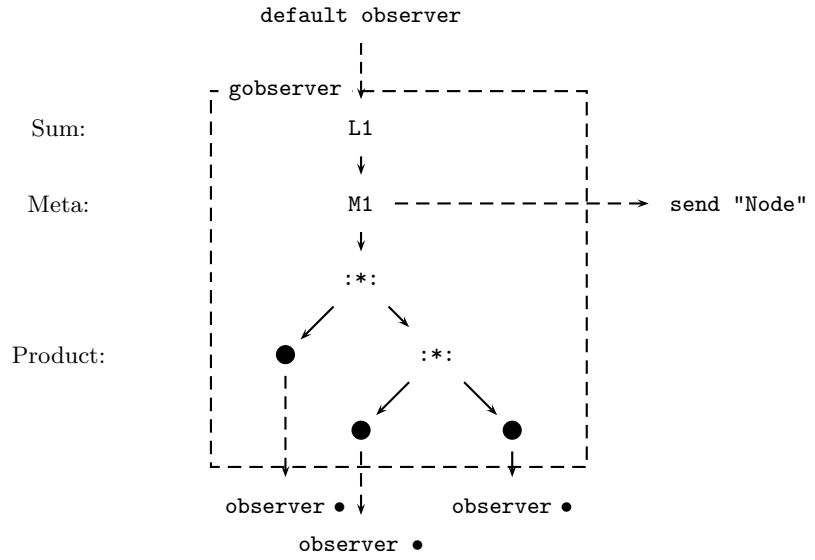

Fig. 7: Using GDM's type representation to observe an example value of our `Tree` type. The dots represent the components of the value, to which in turn instances of `observer` are applied.

```
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)
```

4.2 Scrap Your Observers

Implementing a type generic function with *Scrap Your Boilerplate* (SYB) is done by defining this function in terms of the `Typeable` and `Data` classes. The `Data` class provides methods to query, map and fold over the components of a value. The `Typeable` class provides a method to safely cast values. With SYB we can define a generic transformation by first extending a simple function such that it works over many types and then passing the type-extended function as an argument to a generic data traversal combinator, such as a query, map or fold function [10].

The class `Data` plays a central role in the SYB design pattern. Instances of the `Data` class are easy and regular to define and can be generated by a compiler when instructed by a deriving-clause [11].

Our goal is to develop a generic `observer` that takes the value of any type that belongs to the `Data` class. Our generic function should have the same behaviour as the `observer` instances discussed before, that is: to create a shallow representation of the value and to add intermediate observations to its components.

```
observer :: Data a => a -> Parent -> a
observer x = send (shallowShow x) (observeChildren x)
```

We do not use the class `Observable` in our SYB implementation. An alternative implementation could provide an `Observable` instance for types of the `Data` class. This however can lead to incoherent instances when we try to define an ad-hoc `Observable` instance for a type that already has a `Data` instance.

shallowShow We start with defining how a shallow representation is produced. In [11] a generic `show` is implemented. To get the name of the constructor the methods `toConstr` and `showConstr` are defined for all types `a` of the `Data` class:

```
toConstr   :: Data a => a -> Constr
showConstr :: Constr -> String
```

Applying `toConstr` to a value of base types such as `Int` results in a special `Constr` representing that value. We use a composition of these two methods to produce a shallow representation of any value of the `Data` type:

```
shallowShow :: Data a => a -> String
shallowShow = showConstr . toConstr
```

observeChildren SYB provides two methods to map over a value from the `Data` class: `gmapT` to apply a function to all immediate components of a value and `gmapM` to perform a monadic transformation on all immediate components of a value [10]. The latter is what we need to define `observeChildren`: by applying `thunk` the components will be observed and numbered. We have more to say on `gmapM` in Section 5.2.

```
observeChildren :: Data a => a -> ObserverM a
observeChildren = gmapM thunk
```

4.3 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.

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 [21].

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)
```

observeChildren We define the `observerChildren` template in a way that is syntactically close to the earlier SYB definition: we apply `thunk` to all components with a generic monadic map. The definition of `gmapM` with TH behaves similar to the `gmapM` of SYB but operates on templates.

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

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 Strange Types

Up to now we have discussed observing values which have a type such as `Int` or `Tree Rational` of kind `*`. The following types need further consideration:

- Type constructors such as `Tree` do not have any values. Therefore these cannot be observed directly. However, with type constructors we can create polymorphic types such as `Tree a`. In Section 5.1 we discuss how to observe values of type `Tree a` for any `a`.
- The function type constructor has kind `* -> *`. A function is observed by collecting the argument-result pairs of its applications. In Section 5.2 we discuss both the ad-hoc instance and the generic `observer` for function types.
- 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 [3].

5.1 Partial Observe from Template

Up to now we assumed that all components of an observed type are observable. In Section 3 we already gave reasons for sometimes desiring *not* to observe components 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 annotation which components 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 components we need to choose between `thunk` to continue tracing deeper, or `nothunk` to stop tracing.

```
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 component. 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 [1].

5.2 Observing Functions

In HOOD's output, a function is represented as a finite map of arguments to results. This map is built in two steps: a function is observed by observing the set of its applications and an application is observed by observing the argument and result. In other words: the components of a function are the applications of that function and the components of function application are the argument and result.

In the original HOOD implementation this is done via an ad-hoc implementation of the `observer` method. With TH we generate ad-hoc instances of `observer`, and with GDM we define a default observer; both frameworks allow us to keep the hand-written observer instance for function types.

Because `observe` in the SYB solution is defined over the values that have a type of the `Data` rather than the `Observable` class, we cannot use a hand-written observer and need to extend our generic definition with the function type. In the remainder of this section we discuss how we can map over the components of a function as defined above.

Monadic Map over Function Application Lämmel and Jones dismiss traversing into functional values as impossible unless the source code itself is traversed [10]. The monadic map function `gmapM` from Scrap Your Boilerplate is defined over types from the `Data` class, but for function type it does not do what we need.

```
gmapM :: Monad m => (forall b . Data b => b -> m b) -> a -> m a
instance Data (a->b) where gmapM g fn = return fn
```

While it is hard to define what the components of a function are, we can define what the components of function *application* are: the argument and the result. From these argument and result pairs we can construct a finite map to represent the function. We define function `apM` to traverse into the components of function application.

```
apM :: Monad m => (a -> m b) -> (b -> a) -> a -> m b
apM g fn = \arg -> do {arg <- g arg; g (fn arg)}
```

Even if we would provide an alternative definition of `gmapM` there would be a problem: `gmapM` constructs values of the same type as the input: `b` goes in and `m b` comes out. With our view on function types however we want to construct values of a different type: `b->a` goes in and, rather than `m (a->b)`, we want `a->m b` to come out.

Emulating A Monadic Map We can define a specific implementation of the `gmapT` instance for functional types, to emulate *a specific application* of `gmapM`. To get context information into our specific implementation of `gmapT` we can construct a special "transformer" function that actually does not transform anything but just returns the context.

```
funObserver :: (Data a) => a -> Parent -> a
funObserver y c = gmapT (mkT (\_-> c)) y

instance (Data a, Data b) =>  Data (a -> b) where
  gmapT g fn = observeFunChildren (g root) fn
  toConstr f = mkConstr (mkNoRepType "Fun") "Fun" [] Prefix
```

Because we use a different approach for function types, `observer` needs to detect which types are function types and use our special approach in those cases:

```
observer :: (Data a) => a -> Parent -> a
observer x c = if   isFun x
               then funObserver x c
               else send (shallowShow x) (observeChildren x) c
```

Concluding Remarks on Classes Compared to using an ad-hoc instance as we did with GDM and TH, our SYB `observer` for function types is more complicated. Furthermore, redefining the `gmapT` instance will prevent us from using HOOD in modules that use a conflicting SYB instance of `gmapT`. With GDM we define `Observer` instances while we define the `observer` behaviour in SYB in terms of the `Data` class. This prohibits defining ad-hoc instances with our SYB solution.

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 [16], Hat [22] and Buddha [18] require instrumentation of the whole program, including libraries, and are implemented only for subsets of Haskell [2].

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[4] 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 [8]. 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 [1]. 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 [19].

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 5.1). 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 [1]. 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.

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. [20] defined a generic programming

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

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` instance of `Data` types [9, 12]. 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 [13,15].

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 [4, 12]. Later work addressed this problem at the cost of introducing boilerplate code that was not in the original approach [17].

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 [6].

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 [7].

DrIFT allows the programmer to add directives to the program which create code from rules defined in a separate file [23]. 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 three different generic programming techniques: Scrap Your Boilerplate, Generic Deriving Mechanism and Template Haskell. From our experience we make three observations:

- Neither GDM nor SYB completely support functional values. But GDM and SYB-with-class can be extended with a hand-written ad-hoc `Observable` instance for the function type.
- Specifying per `observe` which types are observed currently requires the power of a meta-language.

– 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 SYB and GDM [5].

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.

	GDM	SYB	TH
function-type instances	ad-hoc	no	ad-hoc or template
type-safe	yes	yes	when using library
class-membership test	no	no	yes

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.

Acknowledgements

We thank Pedro Magalhães and Adam Vogt for their help answering our questions when implementing respectively the GDM and TH versions of HOOD discussed in this paper. We thank the anonymous reviewers for their useful feedback.

References

1. Bernd Braßel, Olaf Chitil, Michael Hanus, and Frank Huch. Observing functional logic computations. In *Practical Aspects of Declarative Languages*. Springer LNCS 3057, 2004.
2. Olaf Chitil, Colin Runciman, and Malcolm 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.
3. Andy Gill. Debugging Haskell by Observing Intermediate Data Structures. *Electronic Notes in Theoretical Computer Science*, 41, 2000. ACM SIGPLAN Workshop on Haskell.
4. Ralf Hinze. Generics for the masses. In *Proceedings of the International Conference on Functional Programming*. ACM Press, 2004.
5. Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*. Springer LNCS 4719, 2007.
6. Ralf Hinze and Andres Löh. "Scrap your boilerplate" revolutions. In *Mathematics of Program Construction*. Springer LNCS 4014, 2006.
7. Patrik Jansson and Johan Jeuring. PolyP — a polytypic programming language extension. In *Proceedings of the symposium on Principles of programming languages*. ACM Press, 1997.

8. Mark P. Jones, Alastair Reid, The Yale Haskell Group, and the OGI School of Science & Engineering. *The Hugs 98 User's Guide*, 1994–2004. `http://www.haskell.org/haskellwiki/Hugs`.
9. Oleg Kiselyov. Smash your boilerplate without class and typeable. `http://article.gmane.org/gmane.comp.lang.haskell.general/14086`, 2006.
10. Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM Press, 2003.
11. Ralf Lämmel and Simon Peyton Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *Proceedings of the International Conference on Functional Programming*. ACM Press, 2004.
12. Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate with Class: Extensible Generic Functions. In *Proceedings of the International Conference on Functional Programming*. ACM Press, 2005.
13. Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In *Practical Aspects of Declarative Languages*. Springer LNCS 2562, 2003.
14. José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A Generic Deriving Mechanism for Haskell. In *Proceedings of the Symposium on Haskell*. ACM Press, 2010.
15. Neil Mitchell and Colin Runciman. Uniform boilerplate and list processing. In *Proceedings of the Haskell workshop*. ACM Press, 2007.
16. Henrik Nilsson. *Declarative debugging for lazy functional languages*. PhD thesis, Linköpings universitet, 1998.
17. Bruno CdS Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In *Proceedings of Trends in Functional Programming*. Elsevier, 2006.
18. Bernard Pope. Declarative Debugging with Buddha. In *Advanced Functional Programming*, pages 273–308. Springer LNCS 3622, 2005.
19. Claus Reinke. GHood – Graphical Visualisation and Animation of Haskell Object Observations. In *Proceedings of the Haskell Workshop*, 2001.
20. Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing Libraries for Generic Programming in Haskell. In *Proceedings of the Symposium on Haskell*. ACM Press, 2008.
21. Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the Workshop on Haskell*. ACM Press, 2002.
22. Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, 2001.
23. Noel Winstanley and John Meacham. DrIFT Manual. `http://repetae.net/computer/haskell/DrIFT/drift.html`, 2008.