Debugging Haskell by Observing Intermediate Data Structures

Andy Gill
Oregon Graduate Institute
andy@cse.ogi.edu
http://www.cse.ogi.edu/~andy

Abstract

Haskell has long needed a debugger. Although there has been much research into the topic of debugging lazy functional programs, no robust tool has yet come from the Haskell community that can help debug full Haskell - until now. This paper describes a portable debugger for full Haskell, building only on commonly implemented extensions. It is based on the concept of observation of intermediate data structures, rather than the more traditional stepping and variable examination paradigm used by imperative debuggers.

1 Introduction

Debuggers allow you to see inside your program while running, and help you understand both the flow of control and the internal data and structures that are being created, manipulated and destroyed. The art of debugging is viewing your program through this portal, letting you locate the difference between what the computer has been told to do, and what the programmer thinks the computer should be doing.

When debugging an imperative program using traditional debugging technology (like gdb or Visual Studio) the programmer might step through some suspect code using sample test data, stopping and examining internal structures at key points. Haskell programs can have an imperative veneer, using the IO monad, and it should be possible to use typical debugging technology for such parts of a Haskell program. But when debugging other parts of Haskell, we cannot straightforwardly use the same debugging technology to render internal information, because many of the hooks that are used to provide the user with debugging facilities do not map neatly across to the lazy functional world.

- There are no variables to observe changing during execution.
- The concept of sequences of actions or executing a specific line number does not exist.
- Any closure has two parents, the static one (that build the closure and give context), and the dynamic one (that *first* evaluate the closure). A stack trace becomes a parent tree.
- When a function is called, its arguments might not yet be evaluated. Should the debugger do extra evaluations?

In this paper, we argue that the analog to breakpointing and examining variables for functional programs is observing intermediate data structures as they are passed between functions. This argument can be considered a generalization of the "debugging via dataflow" idea proposed by Sinclair [7].

Consider this Haskell function

```
natural :: Int -> [Int]
natural
  = reverse
  . map (`mod` 10)
  . takeWhile (/= 0)
  . iterate (`div` 10)
```

The first step to understanding this listful function is to run the function with some example data.

```
Main> natural 3408
[3,4,0,8]
```

This tells us what the function does, but not how the function works. To understand this function, we need to visualize the hidden intermediate structures behind the function, and inside the pipeline of (lazy) intermediate lists. ($ is a combinator for infix application)

```
natural 3408
➔ reverse
  . map (`mod` 10)
  . takeWhile (/= 0)
  . iterate (`div` 10)
  $ 3408
➔ reverse
  . map (`mod` 10)
  . takeWhile (/= 0)
  $ (3408 : 340 : 34 : 3 : 0 :_)
➔ reverse
  . map (`mod` 10)
  $ (3408 : 340 : 34 : 3 : [])
➔ reverse
  $ (8 : 0 : 4 : 3 : [])
➔ (3 : 4 : 0 : 8 : [])
```

Displaying steps like this gets garrulous quickly. Yet the critical information - the intermediate structures - can be concisely expressed.

```
-- after iterate (`div` 10)
 ( 3408 : 340 : 34 : 3 : 0 : _ )
-- after takeWhile (/= 0)
 ( 3408 : 340 : 34 : 3 : [] )
-- after map (`mod` 10)
 ( 8 : 0 : 4 : 3 : [] )
-- after reverse
 ( 3 : 4 : 0 : 8 : [] )
```

We want to build a portable debugger (in the form of a Haskell library) that lets Haskell users get concise data structure information, like the information displayed above, about the structures in their Haskell programs. Even though our debugger answers only this one question - what are the contents of specific intermediate structures, because structures in Haskell are both rich and regular, even this simple question can be the basis of a powerful debugging tool.

Our overall debugging system is follows:

- We provide a Haskell library that contains combinators for debugging. (Taking this form allows the user to debug full Haskell.)
- The frustrated Haskell programmer uses these debugging combinators to annotate their code, and re-runs the Haskell program.
- The execution of the Haskell program runs as normal; there are no behavioral changes because of the debugging annotations.
- The structures that have been marked for observation are displayed on the users console on termination of their program.

Other versions of the debugging library allow for other debugging setups, like offline observations of data-structures.

2 Debugging Combinators

We introduce our new debugging combinator in terms of an improvement of the current state of the art in full Haskell debugging, which is using an unsafe function called *trace*.

2.1 trace – A Reprise

All current Haskell implementations come with this (non-standard) function, which has the type:

```
trace :: String -> a -> a
```

The semantics of trace is to print (a side effect) the first argument, and return the second argument. There are three main problems with using trace for debugging.

The first problem with trace is *the incomprehensibleness of output*. Augustsson and Johnsson had a variation of trace in their LML compiler [1]. Their conclusion about trace was that it was generally difficult to understand the "mish-mash" of output from different instances of trace. This is partly because the strictness of the first argument of trace might itself trigger other traces, and partly due to the unintuitive ordering of lazy evaluation. The "mish-mash" problem could perhaps be tackled using a post-processor on the output.

The second problem with trace is that *inserting it into Haskell code tends to be invasive*, changing the structure of code. For example, consider a variant of sum, which displays its own execution using trace.

```
tracing_sum xs = trace message res
   where
      res = sum xs
      message = "sum " ++ show xs ++
                    " = " ++ show res
```

Running tracing_sum using Hugs gives:

```
Main> tracing_sum [1,2,3]
sum [1,2,3] = 66
Main>
```

We have observed the behavior of sum, but needed to make non-trivial code changes to do so.

The third problem is *trace changes the strictness of the things it is observing*, because trace is hyper-strict in its first argument. Consider a tracing version of fst.

```
tracing_fst pair = trace message res
   where
      res = fst pair
      message = "fst " ++ show pair ++
                    " = " ++ show res
```

Using this version of fst is problematic, because of the strictness of tracing_fst.

```
Main> tracing_fst (99,undefined :: Int)
fst (99,
Program error: {undefined}
Main>
```

2.2 Introducing observe

The function trace can be really useful for debugging Haskell, but the bona fide shortcoming is that trace is at too low a level. Building combinator libraries is a common way to build a low-level primitives, giving interfaces that are both friendlier and more intuitive.

What form could a higher level debugging combinator take? Using the examples in the introduction as evidence, we argue that it should take the form of a function that allows us to observe data structures in a transparent way. As a way of achieving this, consider the Haskell fragment:

```
consumer . producer
```

Imagine if the Prelude function id *remembered* its argument. We could insert strategically placed id's, and id would tell us what got passed from the producer to the consumer.

```
consumer . id . producer
```

We argue that a higher level combinator for debugging should take this form, and both passing an argument transparently, and observing and remembering it. To facilitate multiple observations in one program, we use a string argument, which is a label used only for identification purposes. The type of our principal debugging combinator is

```
observe :: (Observable a) => String -> a -> a
```

In the above (point-free) example, we could write:

```
consumer . observe "intermediate" . producer
```

This has identical semantics to `consumer . producer`, but the observe squirrels away the data structure that gets drawn through it, putting it into some persistent structure for later perusal. As far as the execution of Haskell program is concerned, observe (with a label) is just a version of id. Notice that observe can be used to observe *any* expression, not just the intermediate values inside a point-free pipeline; we will see examples of both styles later.

observe has a type class restriction on the object being observed. This does not turn out to be as big a problem as might be thought.

We provide instances for all the Haskell98 base types (Int, Bool, Float, etc), as well as many containers (List, Array, Maybe, Tuples, etc). We will return to the specifics of this restriction in Section 5.2, because the type class mechanism provided the framework that enables observe to work.

How does observe compare with respect to the three weakness of trace?

- trace sometimes produced a "mish-mash" of output. In our system, we provide renderings, using pretty printer, of the specific observations made by observe. This is possible because observe provides a structured way of looking at Haskell objects.
- Unlike advanced uses of trace, minimal code changes are required to observe an intermediate structure.
- Finally and critically, the strictness of the observed structure is not changed, because observe does not do any evaluation of the object it is observing. Observation of an infinite list, or a list full of ⊥ is perfectly valid, as we shall see shortly.

3 Examples of using observe

Now we look at several examples of observe being used, before explaining how to implement observe in Section 5.

3.1 Observing a finite list

As a first example consider:

```
ex1 :: IO ()
ex1 = print
    ((observe "list" :: Observing [Int]) [0..9])
```

If we run this IO action inside the debugging context (explained in Section 6.1), we would make the observation

```
-- list
 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : []
```

We have successfully observed an intermediate data structure, without changing the value or semantics of the final Haskell program.

We use the observe type synonym to allow us to be explicit about what type we think we are observing.

```
type Observing a = a -> a
```

However, using this explicit typing is optional. We could have equally well written

```
ex1 = print (observe "list" [0..9])
```

This definition however relies on the default mechanism choosing an Int or Integer list. Typically the type of observe is fully determined by its context, but we sometimes include the type signature with our examples to make explicit to the reader what type is being observed.

3.2 Observing an intermediate list

observe can be used partially applied, which is the typical use scenario when observing inside a point-free pipeline.

```
ex2 = print
   . reverse
   . (observe "intermediate" :: Observing [Int])
   . reverse
   $ [0..9]
```

This observe makes the following observation

```
-- intermediate
 9 : 8 : 7 : 6 : 5 : 4 : 3 : 2 : 1 : 0 : []
```

3.3 Observing an infinite list

Both the lists we have observed so far were finite. As an example of an observation on an infinite list, consider:

```
ex3 :: IO ()
ex3 = print
      (take 10
        (observe "infinite list" [0..])
      )
```

Here we observe an infinite list, starting at 0, which has the first 10 elements taken from it, and is then printed. Running this example allows us to make the observation

```
-- infinite list
 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : _
```

We can see that 10 elements have been evaluated, but the tail of the 10th cons has not been evaluated, rendered using the notation "_". If more of the list were extracted, we would see more cons cells, etc.

3.4 Observing lists with unevaluated elements

So what about unevaluated elements of the list? What if we were to take the length of finite list?

```
ex4 :: IO ()
ex4 = print
      (length
        (observe "finite list" [1..10])
      )
```

This gives the observations as

```
-- finite list
_ : _ : _ : _ : _ : _ : _ : _ : _ : _ : []
```

What if the elements were ⊥?

```
ex5 :: IO ()
ex5 = print
     (length
      ((observe "finite list" :: Observing [()])
         [ error "oops!" | _ <- [0..9]]
      )
     )
```

This gives exactly the same debugging output as ex4. Because we never evaluate the elements, it did not matter what they were, even if the elements were bottom. We needed to give them *some* non-polymorphic types, so we can actually observe them, though.

What about if only some elements are observed?

```
ex6 :: IO ()
ex6 = let xs = observe "list" [0..9]
      in print (xs !! 2 + xs !! 4)
```

This example gives

```
-- list
 _ : _ : 2 : _ : 4 : _
```

We can use observe to both see data inside intermediate structures, and also as a tool to see how much of a lazy structure is actually evaluated, *without fear of changing the evaluation order.* This is where the power of observe lies.

3.5 Using more than one observe

One program can contain many specific instances of observe. We might rewrite the natural example from the introduction ("…" refers to more text, shown in output comments)

```
natural :: Int -> [Int]
natural
 = (observe "after reverse"   :: Observing [Int])
 . reverse
 . (observe "after map …"     :: Observing [Int])
 . map (`mod` 10)
 . (observe "after takeWhi …" :: Observing [Int])
 . takeWhile (/= 0)
 . (observe "after iterate …" :: Observing [Int])
 . iterate (`div` 10)
```

Running this the example data, 3408, gives:

```
-- after iterate (`div` 10)
 (3408 : 340 : 34 : 3 : 0 : _)
-- after takeWhile (/= 0)
 ( 3408 : 340 : 34 : 3 : [] )
-- after map (`mod` 10)
 ( 8 : 0 : 4 : 3 : [] )
-- after reverse
 ( 3 : 4 : 0 : 8 : [])
```

This is exactly what we were looking for in our introduction!

4 Advanced uses of observe

We have seen how observe is a powerful tool for seeing what before was hidden. We now look at a number of other ways of using observe for debugging, beyond simply looking inside pipelines.

4.1 Observing Functions

If we can observe base types (like Int and Bool), and can observe containers (like tuples and lists), can we also observe Haskell functions?

What does it mean to observe a function? We argue that to observe a function is to observe a finite mapping from (observable) arguments to (observable) results. So, for our observational purposes, functions are just a bag of argument-result pairs, one for each time the observed function is invoked.

Functions are observed only in the specific ways they are used. Function arguments (or results) might contain unevaluated aspects, like several of the examples in Section 2.1.

What does this mean in practical terms? Let's look at an example:

```
ex7 = print
  ((observe "length" :: Observing ([Int] -> Int))
      length [1..3]
  )
```

This allows the following observation

```
-- length
  { \ (_ : _ : _ : []) -> 3
  }
```

We notice a number of things about this example.

- observe now takes three arguments, the label, the observed entity (the length function), and the argument to length. Remember that "observe <label>" is styled id, and just returns its argument. The effect on the Haskell program can be explained using simple rewriting

```
  (observe "length" :: Observing ([Int] ->Int))
       length [1..3]
   -- remove the type annotation
=  observe "length" length [1..3]
   -- turn observe into id
=  id length [1..3]
   -- id takes one argument
=  (id length) [1..3]
   -- which is simply returned
=  (length) [1..3]
```

  This line of reasoning also works with further arguments, and observe can successfully observe multiple argument functions.

- Rather than render functions as a bag of pairs, we take liberties and use more Haskell like syntax when printing debugging output.

- The length function did not look at the arguments, specifically the elements of the list. This is now a reflecting the state of the list itself. Someone else might have evaluated the elements, but we will never see this by observing length, because *the observation on length is only concerned with the way the argument and result is used specifically by length* in that context.

Observing functions is general and powerful. We place observe at the caller site, and can see the effect that specific function has from this context, including higher order functions.

```
ex8 = print
    ((observe "foldl (+) 0 [1..4]"
      :: Observing ((Int -> Int -> Int)
              -> Int -> [Int] -> Int)
     ) foldl (+) 0 [1..4]
    )
```

```
-- foldl (+) 0 [1..4]
 { \ { \ 6 4 -> 10
    , \ 3 3 -> 6
    , \ 1 2 -> 3
    , \ 0 1 -> 1
    }
    0
    ( 1 : 2 : 3 : 4 : [])
    -> 10
 }
```

Notice by observing foldl, we have also observed its arguments, including a functional one. We can see exactly how higher-orderness is this example is used.

We can make great use of observing functions when examining pipelines. Returning to our natural example, we can now observe the individual transformers, rather than the structures between them.

```
natural :: Int -> [Int]
natural
 = observe "reverse"          reverse
 . observe "map (`mod` 10)"    map (`mod` 10)
 . observe "takeWhile (/= 0)"  takeWhile (/= 0)
 . observe "iterate (`div` …)" iterate (`div` 10)
```

Notice there is no "." between the observes and the original code. We give the output from "iterate …" and "takeWhile …", the others are similar in style.

```
-- iterate (`div` 10)
 { \ { \ 3  -> 0
    , \ 34  -> 3
    , \ 340  -> 34
    , \ 3408  -> 340
    } 3408
     -> 3408 : 340 : 34 : 3 : 0 : _
 }
-- takeWhile (/= 0)
 { \ { \ 0  -> False
    , \ 3  -> True
    , \ 34  -> True
    , \ 340  -> True
    , \ 3408  -> True
    } (3408 : 340 : 34 : 3 : 0 : _)
     -> 3408 : 340 : 34 : 3 : []
 }
```

This is a clear summary of what the transformers we are doing. iterate took an integer (3408), and produced a stream of decreasing numbers, of which the first 5 were evaluated. We also see how the functional argument to iterate was used. takeWhile turned an infinite list into a finite list, when it found the element 0.

4.2 Observing the State Monad

We can use observe to look at the state inside the state monad. State monads typically have a state transformer function that takes a complete state, and returns a new state. Let's call this function modify.

```
modify :: (State -> State) -> M ()
```

We can observe the state at a specific point using the function observeM.

```
observeM :: String -> M ()
observeM label
  = modify (observe label :: Observing State)
```

By placing observeM at appropriate places, we can take snapshots of the state. Other combinators can be built to look inside other monads, like the reader monad and writer monad.

observeM was instrumental to the debugging of a Haskell model of a pretty printer used in the Java based data structure browser presented in Section 6.2. Quickcheck [2] was used to find problematic counter examples, and observeM opened up the inner workings of the faulty Haskell model. One problem with the original Haskell model was that an update of the state in the monad was not being done correctly, and this manifested itself in the form of unevaluated components inside the state that were expected to contain evaluated data-structures.

4.3 Observing the IO Monad

Can we observe an IO action? An IO action has two parts, the action (which is opaque), and the result of the action, which we can observe. We render an IO action using the pseudo-constructor **<IO>**, followed by an observation on the returned object. Consider this example:

```
ex9 :: IO Int
ex9 = print
    ((observe "getChar" :: Observing (IO Char))
     getChar
    )
```

It would render as

```
-- getChar
<IO> 'x'
```

We read this as "some side effect happened, resulting in the value 'x' being returned". As another example, consider:

```
ex10 :: Char -> IO ()
ex10 ch
 = print
   (observe "putChar"
               :: Observing (Char -> IO ()))
     putChar ch
   )
```

```
-- putChar
let fn 'x' = <IO> ()
```

We read this as "we have a function that takes 'x', does some side-effect stuff, and returns unit".

One great possible use of observing the IO monad is for remembering reads and writes to mutable variables (IORefs and MVars). In this way, functional programs written in an imperative manner can be debugged using observe.

4.4 Multiple Observations

One weakness of observe is that there is no way of tying together the different observations inside one function. Two invocations of natural would result in each observation being recorded, giving a set containing two structures for each label.

So if we call natural with 3408, and later with 123, we have two observations for each label. We return to the list of observations example for brevity, the problem (and solution) carries over to functions trivially.

```
-- after iterate (`div` 10)
 { (3408 : 340 : 34 : 3 : 0 : _)
 , (123 : 12 :1 : 0 : _) }
-- after takeWhile (/= 0)
 { ( 3408 : 340 : 34 : 3 : [] )
 , (123 : 12 :1 : []) }
-- after map (`mod` 10)
 { ( 8 : 0 : 4 : 3 : [] )
 , (3 : 2 :1 : []) }
-- after reverse
 { ( 3 : 4 : 0 : 8 : [])
 , (1 : 2 :3 : []) }
```

Now there is nothing tying together the data that share the same pipeline, apart from manual observations. There is no guarantee (because of lazy evaluation) that the data will be ordered like this example. In order to allow individual pipelines to have a way of tying observation together, we provide another combinator.

```
observations :: (Observable a)
                => String -> (Observer -> a) -> a
data Observer
  = Observer (forall a .(Observable a)
                     => String -> a -> a)
```

We have now left the Haskell98 camp, because we are using rank-2 polymorphism. observations passes a local version of observe, allowing scoped versions of observe to be used when debugging. An example use of this combinator is

```
natural :: Observer -> Int -> [Int]
natural = observations "natural" natural'
  $ \ (Observer observe) ->
    (observe "after reverse"  :: Observing [Int])
  . reverse
  . (observe "after map …"    :: Observing [Int])
  . map (`mod` 10)
  . (observe "after takeWhi …":: Observing [Int])
  . takeWhile (/= 0)
  . (observe "after iterate …":: Observing [Int])
  . iterate (`div` 10)
```

At this point, we are getting diminishing returns because we have made a number of changes to the code to get use these combinators. Notice we can't just return a local observe but need to wrap it inside the constructor, Observer, because observe must have a fully polymorphic type.

The example outputs…

```
-- natural
  { \ 3408  -> 3 : 4 : 0 : 8 : []
  }
  -- after reverse
    3 : 4 : 0 : 8 : []
  -- after map
    8 : 0 : 4 : 3 : []
  -- after takeWhile
    3408 : 340 : 34 : 3 : []
  -- after iterate
    3408 : 340 : 34 : 3 : 0 : _

-- natural
  { \ 123  -> 1 : 2 : 3 : []
  }
  -- after reverse
    1 : 2 : 3 : []
  -- after map
    3 : 2 : 1 : []
  -- after takeWhile
    123 : 12 : 1 : []
  -- after iterate
    123 : 12 : 1 : 0 : _
```

This is more structured record of what happened.

4.5 Summary of using observe

We have seen many examples of observe successfully observing internal, sometimes intermediate, structures. It is both general and flexible, working in many different practical settings, such as: observing how functions are used, observing state inside monads, and observing IO actions.

5 How does observe work?

We have demonstrated that observe can be used as a powerful debugging tool, but we still need to answer the question of how to implement observe in a portable way. This section introduces this new mechanism.

Take as an example this Haskell fragment.

```
ex12 = let pair = (Just 1,Nothing)
       in print (fst pair)
```

What steps has pair gone through in the Haskell execution? All expressions start as unevaluated thunks.

```
… pair = <thunk> -- start
```

First print is hyper-strict in its argument, so it starts the evaluation of the expression "(fst pair)". This causes pair to be evaluated via fst, returning a tuple with two thunks inside it.

```
… pair = (<thunk>,<thunk>) -- after step 1
```

Now the fst function returns the first component of the tuple, and this element is further evaluated by print.

```
… pair = (Just <thunk>,<thunk>) -- after step 2
```

And finally, the thunk inside the Just constructor is evaluated, giving

```
… pair = (Just 1,<thunk>) -- after step 3
```

This evaluation can be illustrated diagrammatically, showing the three evaluation steps that this structure went through.

```
•
↓(1)
( • , • )
  ↓(2)
  Just •
       ↓(3)
       1
```

We can now explain the key ideas behind the implementation of observe.

- We automatically insert side-effecting functions in place of the labeled arrows in the diagram above, which both return the correct result or the evaluation to weak-head normal form, *and also inform (potentially offline) agent that the reduction has taken place*. All thunks (including internal thunks) are therefore replaced with functions that, when evaluated, trigger their informative side effect.
- We use the type class mechanism as a vehicle for this systematic (runtime) rewriting.

Next, we examine the details of both these ideas.

5.1 Communicating the Shape of Data Structures

We need to give enough information to our viewer to allow it to rebuild a local copy of our observed structure. What information might these side-effecting functions send?

- What evaluation happened (path location)
- What the evaluation reduced to ((:),(,),Nothing, etc)

So, in the example above we would pass the following information via our side effecting function.

Name	Location	Constructor
<>	root	tuple constructor with two children
<1>	first thunk inside root	The Just constructor with one child
<1.1>	first thunk inside the first thunk of the root	The integer 1

This information is enough to recreate the observed structure. We start with an unevaluated thunk.

$$\bullet^{\text{root}}$$

We then accept the first step (<>) giving

$$(\bullet^{\langle 1\rangle} , \bullet^{\langle 2\rangle})$$

Here <1> represents the first thunk inside the constructor produced by the first step, and <2> represents the second thunk from this same reduction. We then accept the next thunk (<1>) giving

$$(\texttt{Just } \bullet^{\langle 1.1\rangle}, \bullet^{\langle 2\rangle})$$

Here <1.1> represents the first (and only) thunk of the constructor produced by the thunk labeled <1>. Finally, we accept information about <1.1> giving

$$(\texttt{Just 1}, \bullet^{\langle 2\rangle})$$

By default, if we know nothing about a thunk, it's unevaluated, like <2>. We now look at how to insert our message passing functions into our data structures.

5.2 Inserting intermediate observations

We use a worker function, observer, to both tell our (potentially offline) agent about reductions happening, and place further calls to new instances of observer on all the sub-thunks. One possible type for such a function is:

```
observer
  :: (Observable a) => [Int] -> String -> a -> a
```

The [Int] is used to represent the path from the root, as seen in the above example. observe can be defined in terms of this function.

```
observe = observer []
```

Let us consider the generic case for observer, over a pseudo-constructor. This is also acts as an informal semantic for observe.

```
data Cons = Cons ty1 … tyn

observer path label (Cons v1 … vn)
  = unsafePerformIO
    { send "Cons" path label
    ; return (
        let y1 = observer (1:path) label vn
          …
            yn = observer (n:path) label vn
        in Cons y1 … yn)
    }
```

We can notice a number of things about the function from this pseudo code.

- observer is strict in its constructor argument. This is not a contradiction from the claim that observe does not effect strictness of what it is observing, in the same way that

  ```
  forall xs :: [a] . foldr (:) [] xs = xs
  ```

  For observer to look at its constructor argument, it must itself be in the process of being evaluated to WHNF.
- The only place observer can get stuck (evaluate to ⊥) is when invoking send. There is (reasonable) presumption that this will not block or fail.
- The path is used in a strict fashion (assuming send is strict).
- observe can change the space behavior of programs, because it loses any sharing in its replication.

If we assume that the path is constant string, and send does not get stuck, simple equational reasoning can show that

```
forall (cons :: Cons) . cons = observe "lab" cons
```

for any cons of the above form.

- Strict fields just re-trigger evaluation of already evaluated things.
- We can consider base types (Int, Integer, etc) to be large enumerated types, and capture them by the above claim about constructors in general.

Functions are captured by a different instance:

```
observer path label fn arg
 = unsafePerformIO $ do
  { send "->" path label
  ; return (
    let arg' = observer (1:path) label arg
        res' = observer (2:path) label (fn arg')
      in res')
  }
```

This is a simplification (because observer actually needs to generate a unique reference for each function invocation), but does capture the behavior as far as the Haskell evaluation is concerned. Again, we use reasoning like that above to claim that

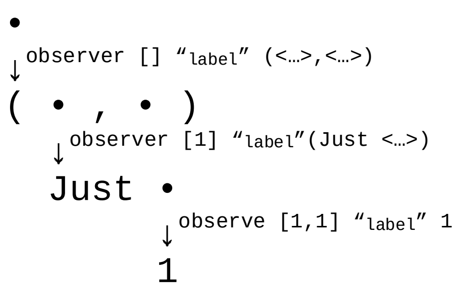
```
forall fn arg . fn arg = observe "lab" fn arg
```

5.3 The Observable Class

We use the type class mechanism to implement the various repeated calls to the worker function, observer, as and when structure gets evaluated. We have a class Observable, and for each observable Haskell object, we have an instance of this class.

```
class Observable a where
    observer :: a -> ObserveContext -> a
```

Reusing our diagram from Section 5.1 above, we have 3 calls to observer.



The first call uses the 2-tuple instance of Observable, the second uses the Maybe instance, and the third uses the Int instance. Each call is given a context, which contains information about where this thunk is in relation to its parent node.

In our implementation, we use a combinator, send, to capture the common idioms used when writing instances of observer. The Observable instance for 2-tuples is:

```
instance (Observable a,Observable b)
        => Observable (a,b) where
  observer (a,b) = send ","
                    (return (,) << a << b)
```

If observer is called on the 2-tuple type, it sends a packet of information, saying it has found a tuple, and sets up two new thunks that are the components of the tuple. The type of send is

```
send :: String
     -> MonadObserver a
     -> Parent
     -> a
```

MonadObserver is a lazy state monad that both counts the total number of sub-thunks this constructor has, and provides a unique context for the sub-thunks. Parent is simply a name for the context.

Several examples of real instances are included in the Appendix.

6 The Haskell Object Observation Debugger

We have implemented these ideas, incorporating them into a full-scale debugging tool we call the Haskell Object Observation Debugger. We give a short overview of the tool here. A user manual is available online. In essence, HOOD is used as follows:

- The user is responsible for importing the Observe library, which exports several debugging functions, including observe, and adding strategic observes to their code.
- Using the Observe library produces an internal trace of what was observed.
- At the termination of the running the code being debugging, some code in the Observe library recreates the structures, much like was done in Section 5.1, and the structures are displayed to the user.

6.1 The Observe Library

The Observe library is implementation of the observe combinators, some supporting combinators, and many instances for various Haskell types. Observe provides:

Base Types: Int, Bool, Float, Double, Integer, Char, ()

Constructors: (Observable a) => [a], (Maybe a)

(Observable a, Observable b) => (a,b), (Array a b), (Either a b)

(...) => 3-tuple, 4-tuple, 5-tuple

Functions: (Observable a, Observable b) => (a -> b)

IO Monad: (Observable a) => IO a

Extensions: Exceptions (error, etc) – with GHC and STG Hugs

In order to do debugging you need to be inside debugging mode. When this mode is turned on, the trace log file is created, and the system is ready for receiving observations. When the mode is turned off, the trace log file is closed. We provide a combinator that helps with these operations.

```
runO :: IO a -> IO ()
```

This turns on observations, runs the provided IO action, turns off observations, and returns. In a Haskell program with main you might write

```
main = runO $ do
        .. rest of program ..
```

To help with interactive use, we provide two extra combinators.

```
printO :: (Show a) => a -> IO ()
printO expr = runO (print expr)

putStrO :: String -> IO ()
putStrO expr = runO (putStr expr)
```

These are provided for convenience. For example, in Hugs you might write

```
Module> printO (observe "list" [0..9])
```

Because this version of print starts the observations, you can use it at the Hugs prompt, and make observations on things at the command line level.

Though Observe.lhs is itself fairly portable (needing only unsafePerformIO and IORef) we also provide versions of Observe.lhs for specific compilers. Classic Hugs98 uses rank-2 polymorphism in one place of the implementation, and uses MVars to allow debugging of concurrent programs. GHC and STG Hugs also use extended versions that provide extra functionality for observing Exceptions and handling threaded execution. Catching, observing and rethrowing exceptions allows you to observe exactly where in your data structures an error is raised, and perhaps later can also be used for debugging programs that blackhole.

In the Appendix we give code fragments from the Observe library, which include many more examples of instances for the Observable class. If users want to observe their own structures, then they need to provide their own instances. However, as can be seen, this is quite straightforward.

There are a couple of important caveats about having observe as a function provided by a library, rather than a separate compilation/interpretation mode.

- observe is referentially transparent with regard to the execution of the Haskell program, but observe is not referentially transparent with regard to possible observations it might make. Compiler optimizations might move observe around, changing what is observed. Here is an example problem

  ```
  let v = observe “label” <expr>
  in … v … v …
  ```

  This might be transformed into

  ```
  … observe “label” <expr>
       … observe “label” <expr> …
  ```

  This does not turn out to be a problem in practice. This transformation and other problematic transformations, though technically valid, change the sharing behavior of the program. Compilers do not like to change these sorts of properties without fully understanding the ramifications of doing so. Furthermore, the worst that can happen is a single structure is observed a number of times. If this occurred, it should be obvious what is happening.

  This glitch with observe turns out not to be a problem in GHC, Classic Hugs and STG Hugs. If any other Haskell compiler has a problem with inappropriate sharing of observe, this can be fixed, even by adding special cases to the sharing optimization. It is a lot easier to add special cases than a whole debugger!

- Hugs does not re-evaluate top level updatable values, called Constant Applicative Forms (CAFs), between specific invocations of expressions at the command line prompt. This is a good thing in general, but it also means that if you want to observe a structure inside a CAF you need to reload the offending CAF each time you want to observe it. This is just a minor annoyance in practice; perhaps a Hugs flag turning off caching of CAF's between expression evaluations could be added.

6.2 Using the HOOD browser

We have an extension to the released version of HOOD, that includes a browser that allows dynamic viewing of structures.

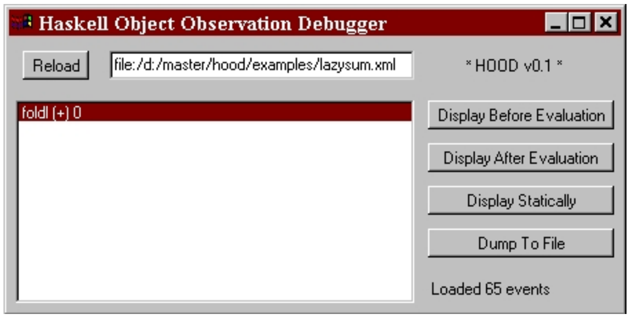
In this new version, a modified version of the Observe library puts the tracing information into a file called observe.xml. Though it might seem that XML is a poor choice for an intermediate format, off the shelf compression tools result in surprisingly good quality of compression (around 90%), which gives a significantly better footprint size than a straightforward binary format, and we have plans for a future version that uses a pre-compressed trace or pipe the trace directly between program and browser.

The browser reads the XML file, and allows the user to browse the structures that were observed. To demonstrate our browser tool, take the example observation on foldl, from Section 4.1. We use runO inside main to turn on and off the observation machinery.

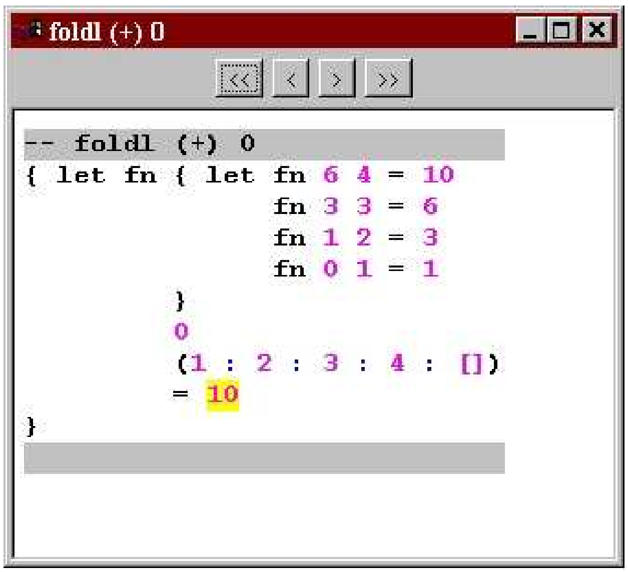
```
main :: IO ()
main = runO ex9
ex9 :: IO ()
ex9 = print
    ((observe "foldl (+) 0 [1..4]"
      :: Observing ((Int -> Int -> Int)
              -> Int -> [Int] -> Int)
     ) foldl (+) 0 [1..4]
    )
```

This produces the file called observe.xml. We now start our browser – the details are implementation dependent, but this can be done directly using JVM, or from inside Netscape Internet

Explorer. After the browser is started, it offers he se list of possible observations to look at.

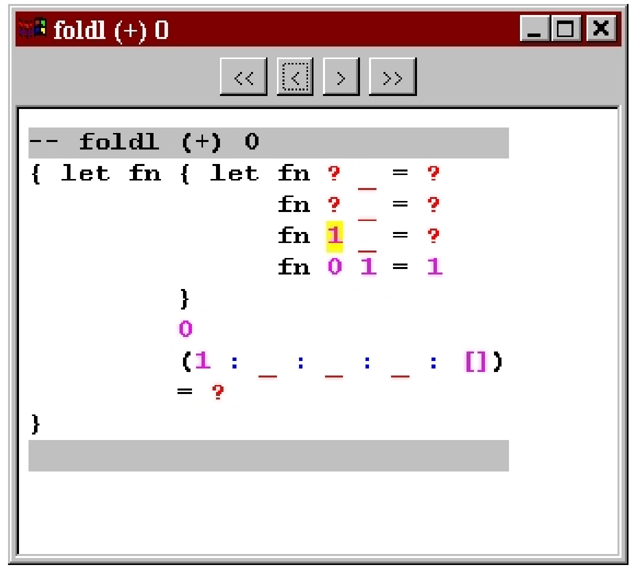


This shows we have loaded 65 events (observatio steps). We only have one observation ("foldl (+) 0"), and we hoose to display after evaluation, giving



This display uses colors to give information beside the raw text. We use purple for base types, blue for constructors, black for syntax, and yellow highlighting for the last expres sion changed. (Note this picture showing an alternative possible syntax for ren- dering functional values.)

This viewer has the ability to step forwards and ba ckwards through the observations, seeing what part of the ob servation was evaluated (demanded) in what order. Though in many cases we are not interested in this information, sometime s is invaluable. For example, if we step back few steps during our perusal of the foldl example, we see strange thing.



We use (red) '?' to signify an expression that ha s been entered (someone has requested its evaluation), but has not yet reached weak head normal form. We see we have a number of ques- tion marks, which correspond to the thunks that ref ent as a consequence of a lazy accumulating parameter, a wel l-known strictness bug.

This dynamic viewing of how structures and functions are used inside real contexts can bring a whole new level of understanding of what goes on when we evaluate functional program s, and could serve as a useful pedagogical tool.

7 Related Work

There are two previous pieces of work that use the explicit observing intermediate structures in debugging gi d.

- Hawk, a microprocessor architecture specification e mbedded language has a function called probe [4].

  ```
  probe :: Filename -> Signal a -> Signal a
  ```

  `probe` works exactly like observe on the Signal level, wh ere Signals are just lazy lists. However, `probe` is strict in the contents of the signals, so it can change the semant ics of a signal. Encouragingly, `probe` has turned out to be extremely useful in practice.

- The stream-based debugger in [9] let the user observe lazy streams as they are evaluated. The information gath ering mechanism was completely different. Their stream-ba sed de- bugger used a primitive (isWHNF a -> Bool) to make sure that they never caused extra evaluation when displaying struc- tures. We expect that we could emulate all the beha vior of this debugger (and more) in our new browser.

The work in this paper was undertaken because of th e success stories told by both these projects, and the hope that our generali- zation of both will be useful in practice when debu gging Haskell programs.

A complete description of the attempts to build debugging tools for lazy functional languages is not possible due to size limitations. Here is a short summary of the techniques; for more details about writing debuggers for Haskell, Watson's thesis [10] is a great starting point.

There are two basic approaches to instrumenting Haskell code:

- The first is where code is transformed to insert extra (side-effecting) functions that record specific actions, like entering functions and evaluating structures. The transformation can be done inside the compiler (and therefore compiler specific) or done as a preprocessing pass (complicating the compilation mechanism). In practice, such transformations turn out to be tied to specific compilers. One example of tracing via transformation is the work by Sparud [8] in his trace option for the nhc compiler.

- The second approach to gathering debugging information is augmenting reduction engines to gather the relevant information, and is completely compiler specific. One example of such a reduction engine is the work by Nilsson [5], who modified the G-machine reduction engine.

Using the raw debugging information gathered to help debug Haskell programs is a difficult problem, partly for the reasons already mentioned in the introduction. One important debugging *strategy* is algorithmic (or declarative) debugging [6]. Algorithmic debuggers compare the result of specific chunks of computations (like function calls) with what the programmer intended. By asking the programmer (or an oracle) about expectations, the debugger can home in on a bug's location. observe can be used to perform a manual version of algorithmic debugging.

8 Conclusions & Future Work

All previous work on debuggers for Haskell have only been implemented for subsets of Haskell, and therefore of limited use for debugging real Haskell programs. This paper combats the need for debugging real Haskell by using a portable library of debugging combinators, and develops a surprisingly rich debugging system using them.

There is work to be done with building semantics for observe. The semantics given in [3] would be a good place to start.

This debugging system could be made even more useful if the Observable class restriction was removed. It would be conceivable to have a compiler flag where Observable is passed silently everywhere, and therefore can be used without type restrictions, provided we supply a default instance for Observable. Alternatively, a reflection interface might be used to look at constructors in a polymorphic way, allowing the type class restriction to be totally eliminated.

HOOD has a web page: **http://www.haskell.org/hood**

The first version of Hood has been released, and is available from the web page. Future versions will include the graphical browser. The source code (including a copy of the graphical browser) is available from the same CVS repository as GHC and Hugs.

Acknowledgements

The idea for using Haskell type classes and unsafe operations to observe intermediate data structures arose from a conversation between Simon Marlow and the author in 1992, when we were both graduate students at Glasgow. Thanks Simon! Thank you also Magnus Carlsson, Tim Sheard, Richard Watson, and the anonymous referees, all of whom gave useful comments and suggestions.

References

[1] Augustsson, L., Johnsson, T. (1989) The Chalmers Lazy-ML Compiler. *The Computing Journal*. 32(2):127-139.

[2] Claessen, K. and Hughes, J. (2000) *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In ICFP 2000, Montreal, Canada.

[3] Launchbury, J. (1993) *A Static Semantics for Lazy Functional Programs*. Proc ACM Principles of Programming Languages, Charleston.

[4] Launchbury, J., Lewis, J. and Cook, B. (1999) *On embedding a microarchitectural design language within Haskell*. In ICFP 99

[5] Nilsson, H. (1998) *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden.

[6] Shapiro, E. (1982) *Algorithmic Program Debugging*. MIT Press.

[7] Sinclair, D. (1991) Debugging by Dataflow Summary. In *Proceedings of the 1990 Glasgow Workshop on Functional Programming*, Portree, Isle of Skye, pp 347-351.

[8] Sparud, J. (1995) *A Transformational Approach to Debugging Lazy Functional Programs*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Goteborg, Sweden.

[9] Sparud, J. and Sabry, A. (1997) Debugging Reactive Systems in Haskell, *Haskell Workshop*, Amsterdam.

[10] Watson, R. (1997) *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, School of Multimedia and Information Technology, Southern Cross University, Australia.

Appendix A – Haskell Code from Observe.lhs

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

type Observing a = a -> a

-- The base types

instance Observable Int       where { observer = observeBase }
instance Observable Bool      where { observer = observeBase }
instance Observable Integer   where { observer = observeBase }
instance Observable Float     where { observer = observeBase }
instance Observable Double    where { observer = observeBase }
instance Observable Char      where { observer = observeBase }

instance Observable ()        where { observer = observeOpaque "()" }

observeBase :: (Show a) => a -> Parent -> a
observeBase lit cxt = seq lit $ send (show lit) (return lit) cxt

observeOpaque :: String -> a -> Parent -> a
observeOpaque str val cxt = seq val $ send str (return val) cxt

-- The constructors

instance (Observable a,Observable b) => Observable (a,b) where
  observer (a,b) = send "," (return (,) << a << b)

instance (Observable a,Observable b,Observable c) => Observable (a,b,c) where
  observer (a,b,c) = send "," (return (,,) << a << b << c)

instance (Observable a,Observable b,Observable c,Observable d)
          => Observable (a,b,c,d) where
  observer (a,b,c,d) = send "," (return (,,,) << a << b << c << d)

instance (Observable a,Observable b,Observable c,Observable d,Observable e)
         => Observable (a,b,c,d,e) where
  observer (a,b,c,d,e) = send "," (return (,,,,) << a << b << c << d << e)

instance (Observable a) => Observable [a] where
  observer (a:as) = send ":"  (return (:) << a << as)
  observer []     = send "[]" (return [])

instance (Observable a) => Observable (Maybe a) where
  observer (Just a) = send "Just"    (return Just << a)
  observer Nothing  = send "Nothing" (return Nothing)

instance (Observable a,Observable b) => Observable (Either a b) where
  observer (Left a)  = send "Left"  (return Left  << a)
  observer (Right a) = send "Right" (return Right << a)

-- arrays

instance (Ix a,Observable a,Observable b) => Observable (Array.Array a b) where
  observer arr = send "array" (return Array.array << Array.bounds arr
                                                  << Array.assocs arr
                              )
-- IO monad

instance (Observable a) => Observable (IO a) where
  observer fn cxt =
        do res <- fn
           send "<IO>" (return return << res) cxt
```