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. 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 traditional 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 using this portal to 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, the programmer might step through some suspect code using sample test data, stopping and examining variables at key points. Yet the analog for functional languages is not clear. What does it mean to step to a specific line or examine variables in a functional language?

In functional languages, listful style (and its generalization, structureful style) is strongly encouraged [2]. Listful and structureful algorithms are expressed in terms of pipelines of data transformers, glued together with intermediate data structures. The structureful style lends to the pervasive fine grain level of lazy functional programming. In this paper, we argue that the analog to breakpointing and examining variables for structureful 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 [10]. We also argue that functional programs written in other styles can also use this debugging paradigm to good effect.

Consider the 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 (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 that lets Haskell users get concise data structure information, like the information displayed above, about the structures in their Haskell programs.

This paper has been submitted to the Haskell Workshop 2000.

www.manaraa.com

Our overall debugging system is as 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 their Haskell program.
- The execution of the Haskell program produces a file, which traces the observations made by the combinators.
- These observations can be viewed using an off-line data structure browser, written in Java.

The structure of this paper is as follows. After giving some background context about debugging functional languages (Section 2), we describe our principal debugging combinator, giving several examples of usage (Sections 3 & 4). Then we explain how our debugging combinators work (Section 5), and describe our implementation of a debugging toolkit based around the debugging combinator (Section 6). Next we describe some related work (Section 7), and give possible future work (Section 8).

2 Background

The field of debugging lazy functional languages has been a fertile research area for several years. In this brief overview we explain how debugging information is gathered, how this information can be used, and discuss commonly implemented debugging combinators. For more detailed account of the area, Chapter 2 of Watson's thesis [15] is a great starting point.

2.1 Tracing Execution

The art of watching and remembering the reduction patterns of functional languages executes is called tracing. There are basically three ways of tracing the execution:

1. **Instrumenting the code with transformations**

   This is where the code is transformed to insert extra (side effecting) functions that record specific actions, like entering functions and evaluating structures. The transformation can either be done inside the compiler (and therefore compiler specific), or as a preprocessing pass (complicating the compilation mechanism). In practice, the transformations turn out to be specific to compiler.

   One example of tracing via transformations is the work by Sparud [11]. It is an option for the nhc compiler. Another recent example of instrumentation via transformation is the work by Watson [15]. Neither translation scheme are implemented for full Haskell.

2. **Using modified reduction engine**

   This is where the reduction engine is augmented to gather trace information, and is completely compiler specific. One example of such a reduction engine is the work by Nilsson [5], with a modified version of the G-machine.

3. **Using a modified interpreter**

   This is where a direct interpreter of the language to be debugged is constructed, which contains tracing code. This alternative is not popular, however, because the speed of this route does not typically permit debugging of non-trivial examples.

None of these techniques is particularly appealing. Using either of the first two will tie down the debugger to a specific compiler implementation, and an aim of this project is to provide a truly portable debugger. This paper introduces a new way of tracing. Section 5 will compare it to the general techniques above, but it is portable in practice.

2.2 User Interaction

Why can't we use traditional debugging technology, like gdb or Visual Studio, once we know how to gather an execution trace? Critical debugging concepts, however, do not map across to the lazy functional world.[1]

- There are no assignments to examine 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 its context), and the dynamic one (that *first* evaluated the closure).
- When a function is called, arguments might not yet be evaluated. Should the debugger do extra evaluations?
- Handling functional arguments also brings other differences.

Considerable effort has been put into getting passed the problems introduced by the functional computational model, and making good use of trace information.

One approach is algorithmic debuggers, which were originally applied to the problem of debugging Prolog programs [9]. Algorithmic debuggers compare the result of specific chunks of computation (like function calls) with what the programmer intended. By asking the programmer (an oracle) about expectations, the debugger can home in on a bug's location. Algorithmic debugging is sometimes called declarative debugging.

Performing post-mortem debugging by using a complete trace (for example) can allow the recovery of some of the more traditional familiar debugging features. One such feature is seeing arguments in their most evaluated form, and ignoring (to some extent) laziness [5]. Another variation of tracing, called Redex trails [11]. However, complete post-mortem traces can be very large.

---

[1] Of course, Haskell can have an imperative veneer, using the IO monad, and it should be possible to use typical debugging technology for such a Haskell program.

www.manaraa.com

Typically, a lazy functional language debugger will offer combinations of these (and other) approaches to the perplexed programmer. We take another approach, and argue that we should offer debugging at the Haskell expression level. As a whole, our debugger answers only one question: what are the contents of this structure? Since structures in Haskell are both the and gular, even this simple question can be the basis of a powerful debugging tool.

2.3 Combinators for Observing Intermediate Structures

All current Haskell implementations come with a (non-standard) function, trace. This has type:

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

The semantics of trace is to print (as a side effect) the first argument, and return the second argument. One problem with trace is that inserting it into Haskell code tends to be invasive, changing the structure of code, as well as changing the strictness because trace is hyper-strict in its first argument. (Tracing, in general, is more powerful than the basic tracing mechanism provided by the function, trace – a better name for trace might be hark!)

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.

We argue that a combinator, a next-generation trace, is easier to use and has much cleaner side effects. This new combinator is the subject of the next section.

3 A New Debugging Combinator

What form could a debugging combinator take? Using the example 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
```

Now imagine a version of the Prelude function id (called ident here), that remembered its argument. We could then write:

```
consumer . ident . producer
```

ident is identical to the implementation of id in the Haskell Prelude in terms of operational and denotation, *except it remembers (as a side effect) the data structure that gets drawn through it*. As far as the Haskell program is concerned, ident is just a version of id.

To facilitate multiple observations in one program, we add 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
```

Instead of the above example, we could write:

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

This has identical semantics to consumer . producer, but the intermediate observation is squirreled away, using the label "intermediate", to some persistent structure for later perusal.

observe has a type class restriction on the object being observed. This is not as restrictive a problem as might be thought. We provide instances for all the Haskell 98 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 provides the framework that enables observe to work.

Some readers might have noticed that the type of observe is practically the same as the type for the trace function. There are major differences, however, between the two:

- trace outputs its string as a side effect, while observe uses the string argument for a label. Typically this argument to observe is a constant string, as in the above example.
- trace totally ignores its second argument, just returning it, while observe examines its second argument, but in a *completely lazy, demand driven manner*. That is, observe only looks at its second argument as much as the context that observe is being evaluated in. Observation of an infinite list, or a list of , is perfectly valid, as we shall see shortly.
- trace outputs to stderr, while observe sends its output to a file, which is perused by a separate tool.

Now we look at several examples of observe being used.

3.1 Observing a finite list

As a first example, consider:

```
ex1 :: IO ()
ex1 = print
     ((observe "list" :: Observe [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 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 Observe 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 a list of Integer. Typically, the type of observe is fully determined by its context, but sometimes we include the type signature with our examples to make explicit to the reader what type is being observed.

المنارة للاستشارات
www.manaraa.com

3.2 Observing an intermediate list

Observe can be used partially applied, which is the typical use scenario when observing inside pipeline.

```
ex2 = print
   . reverse
   . observe "intermediate" :: Observe [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 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 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 the 10 elements have been evaluated, but the tail after the 10th cons has not been evaluated, rendered using the notation '_'. If more elements 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 take the length of a finite list?

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

This gives the observations

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

What if the elements were ⊥?

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

This gives exactly the same debugging output as ex4. Because we never evaluate the elements, it does not matter what they were, even if the elements were bottom. We needed to give them *some* non-polymorphic type, 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 the intermediate structures and as a tool to see how much of a structure is actually evaluated, *without fear of changing the evaluation order.* This is where the power of observe lies.

There is a version which we will return to in section. If we had inlined the right-hand side of the definition of xs into the print statement, we would have *two* observers, as shown in ex7.

```
ex7 :: IO ()
ex7 = let xs = [0..9]
     in print ((observe "list" xs) !! 2
               + (observe "list" xs) !! 4)
```

Now we get this observation

```
-- list
{ (_ : _ : 2 : _)
, (_ : _ : _ : _ : 4 : _ )
}
```

Remember, this is the same list observed from two different places, with individual contexts. For this to work, though, we state that the compiler did inline observe, nothing observe does effect the execution of the *Haskell* code itself in any way.

3.5 Multiple observes

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

Running this example on 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!

المنارة للاستشارات
www.manaraa.com

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 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 observation purposes, functions are just a set 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 3.

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

```
ex8 = print
  ((observe "length" :: Observe ([Int] -> Int))
     length [1..3]
  )
```

This allows the following observation

```
-- length
let fn ( _ : _ : _ : []) = 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 just a stylized id, and id just returns its argument. The effect of the Haskell program can be explained using simple rewriting

```
 (observe "length" :: Observe ([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 returns
=  (length) [1..3]
```

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

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

- The length function did not look at its arguments, specifically the elements of the list. This is in a way 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 of length is only concerned with the way the arguments and results are used specifically by length* in that context.

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

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

```
-- foldl (+) 0 [1..4]
let fn { let fn 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 a higher-order one 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 observe and the original code. We give the output from iterate and takeWhile; the others are similar in style.

```
-- iterate (`div' 10)
let fn { let fn 3408 = 340
             fn 340 = 34
             fn 34 = 3
             fn 3 = 0
         }
         3408 = (3408 : 340 : 34 : 3 : 0 : _)

-- takeWhile (/=0)
let fn { let fn 3408 = True
             fn 340 = True
             fn 34 = True
             fn 3 = True
             fn 0 = False
         }
         (3408 : 340 : 34 : 3 : 0 : _)
         = (3408 : 340 : 34 : 3 : [])
```

This is a clear summary of what the transformers were 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.

المنارة للاستشارات
www.manaraa.com

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 specific points using the function observeM.

```
observeM :: String -> M ()
observeM label
   = modify (observe label :: Observe 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.

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 observation of the returned object. Consider this example:

```
ex10 :: IO Int
ex10 = print
     ((observe "getChar" :: Observe (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:

```
ex11 :: String -> IO ()
ex11 str
 = print
   (observe "putChar" :: Observe (Char -> IO ()))
     putChar str
   )
```

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

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

4.4 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, for example 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 happen to pair through the Haskell execution? All expressions start as unevaluated thunks.

```
… pair = <thunk> -- start
```

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

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

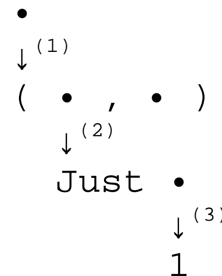
Now the fst function returns the first component of the tuple, and this element is further evaluated by print.

```
… 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.



We can now explain the key idea 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 of the evaluation to weak head normal form, *and also inform an offline agent that the reduction has taken place*. All thunks (including internal thunks) are therefore replaced with functions that, when evaluated, trigger the 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.

www.manaraa.com

5.1 Communicating the Shape of Data Structures

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

- What evaluation happened (location)
- What the evaluation reduced to (3, Nothing, etc)

So the example above would pass the following information to side-effecting function.

Step	Location	Constructor
(1)	root	tuple constructor with two children
(2)	first thunk inside (1)	The Just constructor with one child
(3)	first thunk inside (2)	The integer 1

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

• root

When we accept step (1), giving

(•(1.1) , •(1.2))

Here (1.1) represents the first thunk of the constructor produced by step (1), (1.2) represents the second child of the same reduction. When we accept step (2), giving

(Just •(2.1), •(1.2))

Here (2.1) represents the first (and only) thunk of the constructor produced by step (2). We finally accept step (3), giving

(Just 1, •(1.2))

Notice we give information about inner thunks we are updating using both the parent constructor, and the child index number.

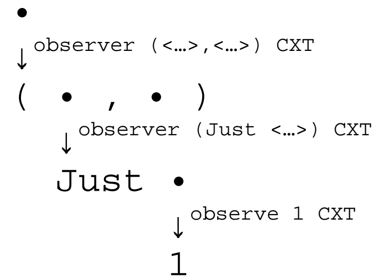
We now look at how to insert our message passing functions into our data structures.

5.2 Using Type classes for rewriting

We use the type class mechanism to force repeated calls to worker function observer as when structures get 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 above, we have 3 calls to observer.



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

The Observable instance for 2-tuples is

```
instance (Observable a,Observable b)
                  => Observable (a,b) where
        observer (a,b) = sendObservePacket "," (do
                  a <- thunk a
                  b <- thunk b
                  return (a,b))
```

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

```
sendObservePacket :: String
               -> MonadObserver a
               -> ObserveContext
               -> a
```

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

```
thunk :: (Observable a) => a -> MonadObserver a
```

thunk includes a call back to observer, at the correct type for the sub-thunk.

We now walk through an example to illustrate observer actually rewriting structures. We use a simplified variation of our earlier example.

```
main = let pair = observe "pair" (42,88)
       in print (fst pair)
```

We expect this to make a 2-tuple, and take the first element, 42. We do not expect observe to tell anything about the second element of the tuple, because it is not evaluated.

observe is simply a wrapper to observer.

```
main = let pair = observer (42,88) {…root…}
       in print (fst pair)
```

Here we are using {…root…} to notate a context that contains some information, including this is root. Now the first time someone evaluates pair, pair evaluates to

```
… pair = observer <thunk> {…root…}
```

observer is strict in its first argument, reducing to

```
… observer (<thunk>,<thunk>) {…root…}
```

We can now execute the function observer for tuples.

المنارة للاستشارات
www.manaraa.com

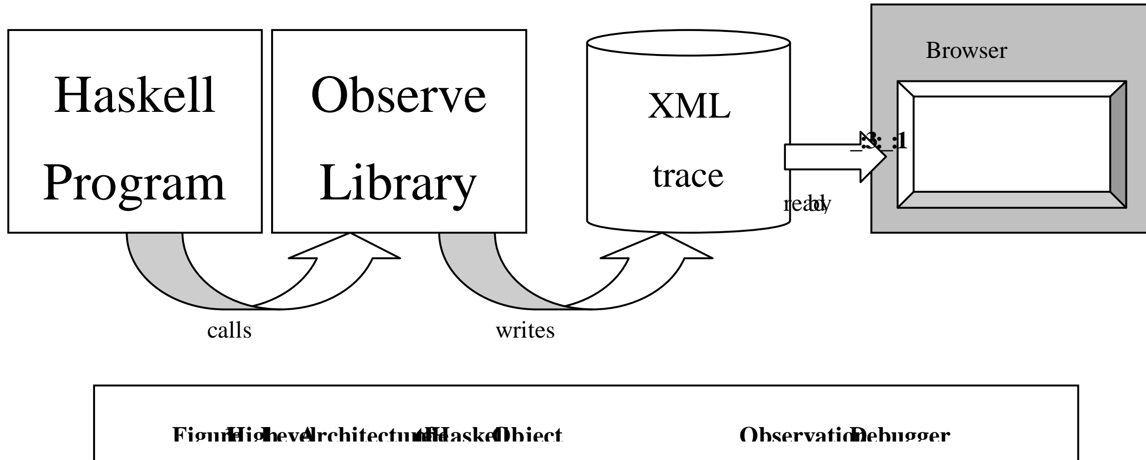


Figure: High-level Architecture of the Haskell Object Observation Debugger

```
… sendObservePacket "," (do
                a <- thunk <thunk>
                b <- thunk <thunk>
                return (a,b)) {…root…}
```

sendObservePacket now issues side effect, and passes the context (root), the constructor name (","), and the number of children (2). The thunk inside the monad evaluates to new calls to observer, and sendObservePacket returns the tuple.

```
… ( observer <thunk> {…pId = X, portNo = 0…}
  , observer <thunk> {…pId = X, portNo = 1…}
  )
```

thunk lazily its argument, with respect to the monadic aspect of the evaluation, even though we've noted the 2 thunks, we've not actually demanded any more evaluation of the context wanted. To continue, the first element of the pair is selected (via fst).

```
… observer <thunk> {…pId = X, portNo = 0…}
```

We have thrown away the second element of the tuple, and with it the side effecting call to observer. In this way, because we do not evaluate observer over the second element of the tuple, we will never observe the content of this element, and never do premature evalutation because of observe.

The call to observe (that is inside the first element of the tuple) now evaluates its argument.

```
… observer 42 {…pId = X, portNo = 0…}
```

Which now reduces to

```
… sendObservePacket (show 42)
                (return 42)
                    {…pId = X, portNo = 0…}
```

Above, this issues a message (the number 42, child of X, port number zero), and returns the value, this time 42.

```
… 42
```

6 The Haskell Object Observation Debugger

We have an implementation of these ideas, incorporating them into full-scale debugging tool called Haskell Object Observation Debugger. We give a short overview of the tool here. A user manual is pending.

Figure gives the high-level architecture of the Haskell Object Observation debugger.

- The user is responsible for producing a log of the observations. The easiest way of doing this is to import the Observe library, which exports several debugging functions, including observe.
- Using the Observe library produces a log in XML format.
- The browser then reads the XML audit of observations, and creates the structures (much like we did one in Section 5.1)

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) => (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)

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 combinators that help with these operations.

المنارة للاستشارات
www.manaraa.com

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

If observe is executed without the observation mechanism being turned on, it behaves like before (passing its second argument), but simply does not record the observation.

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 of things at the command line level.

Though Observe.lhs itself is portable (only needing unsafePerformIO), we also provide versions of Observe.lhs for specific compilers. While Classic Hugs98 uses the standard Observe.lhs, both GHC and STG Hugs use extended versions that provide extra functionality for observing Exceptions. Catching, observing and rethrowing exceptions allows you to observe exactly where in your data structure an error is raised, and 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.

The trace is put in 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 significantly better footprint size than a straightforward binary format, and we have plans for future versions that use pre-compressed trace.

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, as we saw in Section 3.4, observe is not referentially transparent with regard to possible observations it might make. Compiler optimizations might move observe around, changing what is observed.

  This does not turn out to be a problem in practice. The transformation shown in Section 3.4 and other problematic transformations, though technically valid, change the sharing behavior of a 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, and this occurrence should be obvious when it is happening.

  This glitch with observe turns out to be a problem in GHC, Classic Hugs and STG Hugs. If other Haskell compilers have a problem with inappropriate sharing of observe, this can be fixed easily by adding special cases to the sharing optimization. It is easier to add special cases than a whole debugger!

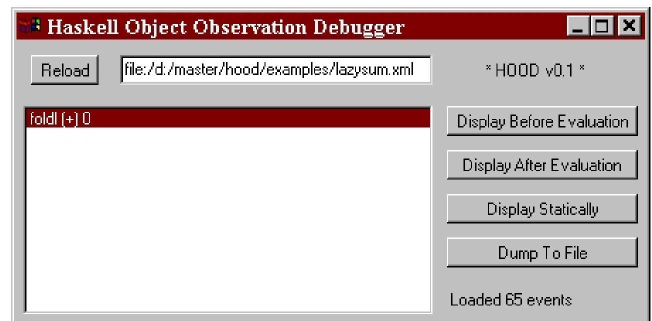
- Hugs does not re-evaluate top-level updatable values, called Constant Applicative Forms (CAF's), between specific invocations of expressions at the command line prompt. This is a good thing in general, but it means that if you want to observe structures inside CAF's, you need to reload the offending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

6.2 Using the HOOD Browser

To demonstrate the browser, we take the example observation on foldl, from Section 4.1. We use runO inside main to turn on and off the observation machinery.

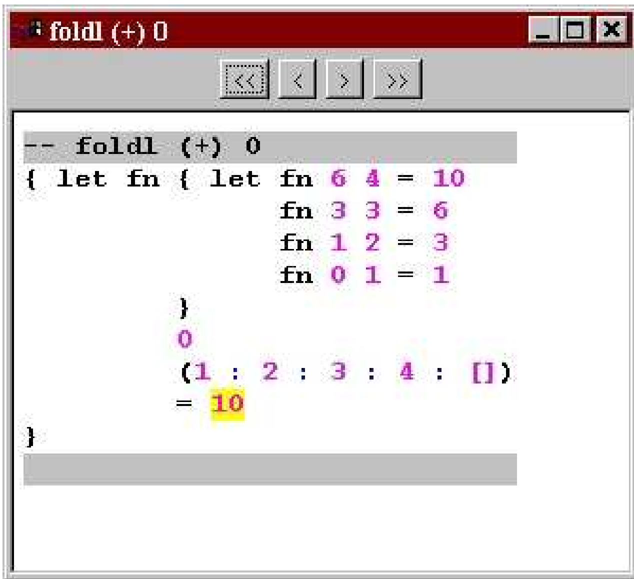
```
main :: IO ()
main = runO ex9
ex9 :: IO ()
ex9 = print
    ((observe "foldl (+) 0 [1..4]"
       :: Observe ((Int -> Int -> Int)
                -> Int -> [Int] -> Int)
     ) foldl (+) 0 [1..4]
    )
```

This produces a file called observe.xml. We start our browser (the details are implementation dependent, but this is done directly using JVM from inside Netscape or Internet Explorer). After the browser is started, it offers the user a list of possible observations to look at.



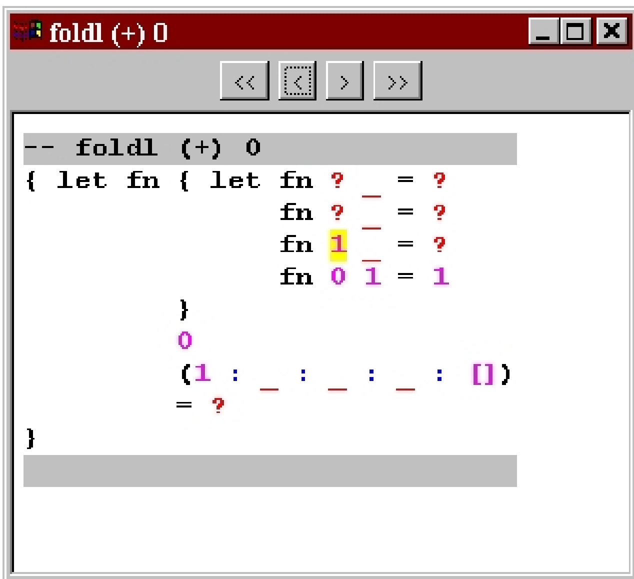
This shows we have loaded 65 events ("observation steps"). We only have one observation ("foldl (+) 0"), and we choose to display after evaluation, giving

www.manaraa.com



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 expression changed.

This viewer has the ability to step forwards and backwards through the observations, seeing what part of the observation was evaluated (demanded) in what order. Though in many cases we are not interested in this information, sometimes it is invaluable. For example, we step back two steps during our perusal of the foldl example, and see a strange thing.



We use (red) ? to signify expressions that have been entered (someone has requested its evaluation), but has not yet reached weak head normal form. We see we have a number of question marks, which correspond to the nasty chain of inters as a consequence of a lazy accumulating parameter, a well-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 programs, and could serve as a useful pedagogical tool.

7 Related Work

We have already discussed a number of debugging systems, and their general context to this work. There are two pieces of closely related work that should be given special attention.

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

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

  `probe` works exactly like observe at the Signal level, where Signals are just lazy lists. However, `probe` is strict in the contents of the signal, so changes the semantics of signals. Encouragingly, `probe` has turned out to be extremely useful in practice.

- Secondly, the stream-based debugger in [12] is also related, because it also puts emphasis on debugging by understanding intermediate structures, even if the tracing mechanism was completely different.

The work in this paper was undertaken because of the success stories told by both these projects, and the hope that a generalization of it will be useful in practice when debugging Haskell programs.

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. A semantics could clear up the caveats that were discussed in Sections 3, 4 and 6.1.

This debugging system could be made even more useful if the Observable class restriction was removed. It would be nice to be able 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 Observe.

The pretty printing algorithm used in the browser is robust (its actually a port of the Haskell algorithm published in [13]) but the printing structure is inflexible. For example, we still print strings as lists of characters, which is verbose. Another example is when displaying abstract syntax trees, concrete syntax is typically more concise. It would be nice to have a flexible scheme for providing rendering shortcuts like these.

المنارة للاستشارات
www.manaraa.com

Hood will be packaged and available for download real soon now. The source code is available from the same CVS repository as GHC and Hugs.

HOOD web page: **http://www.cse.ogi.edu/~andy/hood/**

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 May 1993, when we were both graduate students at Glasgow. Thanks Simon! Thank you also to my colleagues at OGI who 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] Hughes, R.J.M (1989) Why Functional Programming Matters, *Computing Journal.* 32(2):98-107.

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

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

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

[6] O'Donnell, J., Hall, C. (1988) Debugging in applicative languages. *Lisp and Symbolic Computation,* 1(2):113-145

[7] Penney, A. (1999) *Augmented Trace-based Functional Debugging.* PhD Thesis. Department of Computer Science, University of Bristol, England.

[8] Runciman, C., Sparud, J. (1997) Tracing lazy functional computations using Redex trails. In *Proceedings of the 9th International Symposium of Programming on Programming Languages, Implementations, Logics and Programs.*

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

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

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

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

[13] Wadler, P. (1998) *A prettier printer.* Draft paper from http://www.cs.bell-labs.com/who/wadler/

[14] Wadler, P. (1998) *Why no one uses functional languages.* SIGPLAN Notices 33(8):23-27.

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

www.manaraa.com

Appendix A Haskell Code from Observe.lhs

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

observe :: (Observable a) => String -> a -> a
observe name a = generateContext name maxBound a

type Observing a = a -> a

-- Some Haskell Base types
instance Observable Int       where { observer = observe Lit }
instance Observable Bool      where { observer = observ eLit }
instance Observable Char      where { observer = observ eLit }
instance Observable ()        where { observer = observeL it }

observeLit :: (Show a) => a -> ObserveContext -> a
observeLit lit cxt =
        seq lit $
        sendObservePacket (show lit) (return lit) cxt

-- Some constructors
instance (Observable a,Observable b) => Observable (a,b) where
        observer (a,b) = sendObservePacket "," (do
                a <- thunk a
                b <- thunk b
                return (a,b))

instance (Observable a) => Observable [a] where
        observer (a:as) = sendObservePacket ":" (do
                a <- thunk a
                as <- thunk as
                return (a:as))
        observer [] = sendObservePacket "[]" (return [])

-- The thunk wrapper round observer
data MonadObserver a = MonadObserver { runMO :: Int  -> Int -> Int -> (a,Int) }

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

thunk :: (Observable a) => a -> MonadObserver a
thunk a = MonadObserver $ \ depth parent port ->
                ( observer a (ObserveContext
                                { observeParent = parent
                                , observePort   = port
                                , observeDepth  = depth
                                })
                , port+1 )

-- Now some side effecting utility functions
sendObservePacket :: String -> MonadObserver a -> O bserveContext -> a
sendObservePacket consLabel fn context = unsafePerformIO $
    do{ g <- readIORef observeGlobal
        ; case g of
          NoObserveGlobal
            -> error "The global observe state is not ena   bled"
          _ -> return ()
        ; let node = observeUniq g
        ; writeIORef observeGlobal (g { observeUniq = node  + 1 })
        ; let (r,portCount) = runMO fn (observeDepth conte xt - 1) node 0
        ; hPutStrLn (observeHandle g)
                   (xmlCons node context (showXmlString consLabe  l) portCount)
        ; return r
        }
```


المنارة للاستشارات
www.manaraa.com