Debugging in Haskell

Lasse Folger

June 29, 2016

Abstract

In this paper we will give an overview on techniques to debug Haskell programs. At first the problems are presented which occur, when debugging Haskell code in a imperative way (printf). After that the debugging tool Hood is presented to observe the evaluation of a Haskell program. Furthermore will GHCI as debugging tool be presented. Additionally are offline tracers shown with the example of Hat.

1 Introduction

Haskell is a purely functional language with static type system. The type system decreases the amount of potential run time errors, since the compiler only allows well typed programs. In a addition no implicit type casting is performed, which gives the programmer a better understanding of the program. It reduces the chances for technical problems such as type casting errors and pattern matching fails. The implicit memory management also reduces the chance for run time errors.

However this does not imply compiled programs are free of bugs. There may still be issues with business logic. To find such bugs in Haskell is usually harder than in other languages. Indeed Haskell is referentially transparent, which is a advantage with regards to comprehensibility, but its evaluation strategy is a lazy one. This means expressions only evaluated when necessary. The problem with this is that the evaluation of expressions seems to be done in a *weird* manner. To see the negative effects on debugging consider the following implementation of insertion sort:

```
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys) = if x <= y
                  then x : ys
                  else y : insert x ys

main = putStrLn (sort "program")
```

This evaluates to:

```
Prelude> main
agop
```

The result is not the expected one, since some letters are missing. The compiler accepts this program, because there is no type error. To find the bug in a piece of code (even if it is this short) can be tough. The approach in a strict imperative language would be to use something like the following:

```
 ...
insert :: Ord a => a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys) =
  Debug.Trace.trace ("insert" ++ show x ++
                     "in" ++ (show (y:ys)))
                    if x <= y
                     then x : ys
                     else y : insert x ys
...
```

The function *trace* is of the type $String \to a \to a$ and it prints its first argument before returning the second argument. This kind of *printf*-debugging does not work well in Haskell. At first the compiler would reject this program, because he cannot derive a instance of *Show* from the instance of *Ord*. We wanted the sort function to be generic, so we declared the type $Ord\ a \Rightarrow [a] \to [a]$. If we try to use the function *show* on the elements of the input the compiler tries to find the correct implementation of show. This does not work, since not every type, which is an instance of *Ord*, needs to be an instance of *Show*. This can (for testing purposes) be fixed by changing the type for the time we are debugging.

Another problem is the fact that we force the evaluation of the parameters of insert by printing them. This modifies the semantics of our code which is not expected from debugging code. In this example no problem occurs by doing this, since the list is evaluated anyways, but in other cases this can cause problems. For example when working with infinite lists or expressions like:
last (map (div 5) [0..5]).

Even if we remove the type error we run in another problem. The output of the modified program looks the following:

```
*Main> main
...
insert 'r' in "go"
insert 'r' in "o"
insert 'p' in "agor"
ainsert 'p' in "gor"
ginsert 'p' in "or"
oinsert 'p' in "r"
p
```

The disadvantages of Haskell with regards to debugging are shown here. The function *main* demands the evaluation of *sort* step by step. This is the reason the debug output is mixed up with the output of the original program. When it tries to print the result of *sort* "*program*", the expression is evaluated to the so called *head normal form*.

The head normal form is a the evaluation of the expression until a constructor (base values such as integers or characters are considered as constructors) is the top level part of the expression. While *sort "program"* is not in head normal form the evaluation is demanded. Doing this the first outputs from trace are generated, but not all. The head normal form is something like (_ : _), where _ indicates, that this value is not evaluated yet. Since print knows now it is a not empty list it wants the first value of the list in a evaluated form. Since this is a character it is already in a evaluated form and printed. After that the print function is applied to the rest of the list, for which the evaluation of the rest list is demanded. This leads to the next debug outputs and so on.

From the perspective of imperative programmer not only the lazy evaluation is a problem while debugging, but also the fact that Haskell is purely functional. This implies there are no global variables. To understand the problem, lets take a look on the output from above:

```
*Main> main
insert 'a' in "m"
insert 'r' in "a"
insert 'g' in "ar"
insert 'g' in "r"
insert 'o' in "ag"
insert 'o' in "g"
...
```

If we look upon the last two lines where the trace information for the insertion of *o* are printed it looks like the *a* in *ag* went missing. This is not the case. The natural way of inserting an element at the correct position is, to iterate on the list until the fitting position is found. To do such iterations, recursion is the technique used in functional languages. But this leads to the fact, that only parts of the result are visible.

To be exact at first *insert o [a,g]* is called. Since *g* cannot be added in front of *a*, a new list will be build with the following structure *a : (insert o [g])*, which produces the next output, namely *insert o in [g]*. This explains why *a* "went missing". The recursive call of insert does not know anything about the *a*, so there is no way to print it at this point, neither is the *a* lost. This makes it difficult to understand, when things are actually missing, because the were forgotten, and when things currently not available, because of recursion.

To engage this problem a wrapper for the function could be build to outsource the recursion and generating only one trace output per insertion. But doing this is pretty laborious in comparison to imperative languages, where the so called *printf-debugging* is the easiest approach.

You can still guess where the bug in the given program is by looking at the output. In the first output line the insertion of *a* into the list *[m]* is logged, while the second line records the insertion of *r* into the list *[a]*. Since we know, due to our implementation, that a recursive call would always have the same first parameter, it is clear that the first and the second row must describe two different calls evoked from *sort*. If we now apply our knowledge of the implementation, we know that the second parameter of the second call is the result of the first call. This implies that the result of *insert a [m]* is *[a]*. Obviously this is not correct. Now we can review our code, which is executed when *insert a [m]* is called, to find the error:

```
insert x (y:ys) =
  if x <= y
   then x : ys              --where is the y?
   else y : insert x ys
```

This example shows that debugging Haskell code in way well known from imperative languages is not satisfactory for Haskell for several reasons. Sometimes we have to adjust type annotations to satisfy the compiler. This seems to be minor problem, but this can mean changing ever type annotation in the whole program.[1] This means more work for the debugger.

The second Problem is the functional nature of Haskell. There is nothing like global data structures or global variables which can be accessed from every point of the code. In addition the only way to use constructs similar to for or while loops in Haskell is recursion. Due to this the programmer needs to keep track if the the output was produced in a recursive call, a call from another function or a combination of these. Furthermore it is quite common in Haskell for values to be functions which cannot be printed that easily, if it is even possible.

The last and most difficult problem is the lazy evaluation. It not only leads to mixing of debugging output and program output, because it is not obvious when the print (trace) function is evaluated. It also means that actual parameters in functions are (most likely) unevaluated. To print these values their evaluation is forced, which can lead to errors or infinite loops.

In conclusion the well known imperative ways of debugging are not satisfactory for Haskell. In the following sections we will present different ways to debug Haskell code. Starting in the following section we will look further into the idea of printf debugging to understand where it can and where it cannot be used. We will also take look into the debugging tool *Hood*. In Section 3 we will see how to use GHCI for debugging. The main part of this paper is presented in Section 4, where we will take a look on the most advanced debugging tools for Haskell, namely the offline tracer like Hat or Hoed. Afterwards we will talk about related work before concluding this paper in the last section.

2 Printf and Hood

Now we will look at two different kinds of direct outputs comparable to a *printf*. At first we will take a deeper look into *trace*, then we will look into *Hood*. [2]

2.1 Trace

We showed problems with *trace* in the previous section. We mentioned the problem that expressions always being evaluated before they are printed. Lets take a deeper look in this problem. This only seems to be a performance issue, because we sometimes evaluate expressions completely, where a partial evaluation would be sufficient. Since there are no side effects such as direct memory manipulation in Haskell, it does not matter when a expression is evaluated, if

[1] In the example we not only had to change the type of *insert*, but also the type of *sort*

[2] We compare these to printf since the produced outputs are unconditional and no user interaction is needed. In later Sections we will give examples of other ways to produce debugging outputs.

the expression is evaluated anyways. Normally we do not know which expressions are evaluated in the run of a program, so there is no guarantee, when we force the evaluation of an expression (e.g. by calling *trace*), that this is correct. We may run in infinite loops or exceptions that would not occur by a normal run of a program. Consider following example:

```
listWithProp :: (Int -> Bool) -> [Int]
listWithProp f = filter f [1..]
```

This functions returns a (infinite) list of numbers with a specific property. For example *listWithProp odd* will return a list of all odd numbers. Thanks to laziness, we can use this function as a generator. If we use this function in our program it may lead to problems if we try to use trace. When we try to *print* this generator, we will run in a infinite loop (though the list can be finite). Especially when using Library functions, we cannot know if we are allowed to force the evaluation the results.

Besides the infinite loops there are *hidden* exceptions as a potential problem when forcing the evaluation of an expression. Thanks to the static type system and implicit memory management there are no type errors or segmentation faults at run time, but there could occur errors like division by 0 errors or pattern matching errors. There are programs which have expressions that lead to such errors, but never evaluate these expressions. By forcing the evaluation of such expressions these *hidden* faults may emerge and lead to crash of the program. For example consider the following program:

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

mapTree :: Tree a -> (a -> b) -> Tree b
mapTree (Node tl el tr) f = Node (mapTree tl f)
                                 (f el)
                                 (mapTree tr f)
mapTree (Leaf el)       f = Leaf (f el)

rightMostEl :: Tree a -> a
rightMostEl (Node _ _ tr) = rightMostEl tr
rightMostEl (Leaf el)     = el
```

This is a simple implementation of a tree with two functions. *mapTree* applies a function all elements of a tree and *rightMostEl* gives the element on the right most branch of the tree. Observe the following expression:

```
rightMostEl $ mapTree (Node (Leaf 0) 1 (Leaf 2))
                      (div 100)
```

This expression would not fail under normal circumstances, but if we want to trace *rightMostEl*, we run into the problem that this tree and all function applications within are evaluated. Since there is a wrong function application, namely *div 100 0*, a *divide by zero* exception would be raised.

It is true that this examples are special cases, but they can occur. The programmer needs to keep in mind that he could change the semantics of his program just by tracing data structures. Using trace needs always special attention to make sure the traced expressions are evaluated anyways or the evaluation cannot lead to errors.

The problem with the mixed output order due to lazy evaluation remains. Especially when using *trace* in two different function (which may interact with each other) the output order could be confusion and needs a close analysis.

Another way of debugging would be using the *-Note* function from the Safe library. These function wrap a lot of the predefined functions which can lead to run time errors such as *haed, tail, fromJust*. The predefined function may lead to run time errors, because there are in the manner of types correct input which cannot be matched. For example the application *head []* would lead to the following error: **** Exception: Prelude.head: empty list*. This error message is not very useful, since we know there was a bad call of head, but not exactly which call of head lead to this exception.

The Safe library provides wrapper for these functions, for example *headNote* with the following type: $headNote :: String \to [a] \to a$. The given String is part of the error message when the function application fails:

```
Prelude> Safe.headNote "help" []
*** Exception: Safe.headNote [], help
```

This offers the possibility to locate the failed call. Otherwise it could be hard to determine from where the call of head which led to the exception was done, especially when there are a many head calls in the code.

2.2 Hood

Until now we only thought about ways of using imperative debugging in Haskell and which problems occur due to the special properties of Haskell. Now we will turn to use these special properties of Haskell to our advantage. We know about Haskell that there are no side effects, because Haskell is purely functional. This implies there are no variables in the sense of memory cells which can be modified by special instructions. This means that the result of a function only depends on its input and not on its environment. This property is also known as referential transparency.

The core idea of Hood is to *log* the reductions done to evaluate a Haskell expression. After the evaluation is complete these logged information is printed. Thanks to the referential transparency there is no need to produce outputs at run time, since there is no *state* at run time that could be necessary to understand if a reduction is correct or not. There are no such things as global variables which are modified and accessed by functions to evaluate their results.

On the other hand by logging the reductions in comparison to the passed values no evaluation is forced. In other words, there is no modification in the semantics of the program. Reading reductions would be tough. To engage this problem Hood provides a readable representation of the logged data for further analysis.

First lets take a look how Hood is used. Hood is a library and the source code needs to be modified to select which parts should be observed. Take a look at the following example: [3]

```
import Debug.Hood.Observe
```

[3] In this case we need the explicit type annotation to make sure there is a *Observable* instance for the type. [0..9] type is $(Num\ t, Enum\ t) \Rightarrow [t]$. There is no guarantee that there is also an instance for Observable.

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

In this first example we just want to take a look on a single expression, namely the list that is passed as the result from the first *reverse* to the second *reverse*. This is done by the function *observe*. The type of observe is: $observe :: Observable\ a \Rightarrow String \rightarrow a \rightarrow a$. The Observable class defines how data types are internally observed. This type class can be derived for Generic[4] data types. For example:[5]

```
data MyType a = MyConstr a Int String deriving Generic
instance (Observable a) => Observable (MyType a)
```

Since the aim of this paper is to give an overview of the features of Hood and not to show its implementation, we will not dig deeper into the Observable class here.

The *String* that is passed is part of the output to identify where the given data where observed. In order to evaluate an expression and print the observed reductions you need to use the function *runO* with the type: $runO :: IO\ a \rightarrow IO\ ()$. The type $IO\ a$ is demanded, because *runO* needs an IO type to start the evaluation and print its recorded logs which looks like this:

```
*Main> runO ex2
[0,1,2,3,4,5,6,7,8,9]

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

The first line is the output produced from the print statement. After that the observations are reported. This is where we find our keyword *intermediate* and the object it observed. Note that the object will not be evaluated at the point it is passed to observe. The reason the whole list is printed, is the call of *print* int *ex2*; it forced the evaluation of the whole list.

Look at the following two example outputs. The first is generated by replacing *print* with *(print . head)* and the second by replacing *reverse* in the second line with *head*:

```
*Main> runO ex3
0

-- intermediate
   _ : _ : _ : _ : _ : _ : _ : _ : _ : 0 : []


*Main> runO ex4
9
```

[4] https://hackage.haskell.org/package/base-4.9.0.0/docs/GHC-Generics.html

[5] This example was taken from https://hackage.haskell.org/package/hood-0.3/docs/Debug-Hood-Observe.html

```
-- intermediate
   9 : _
```

In the first expressions the **structure** of the whole list was evaluated due to the two calls of reverse, but the values were not evaluated which is indicated by "_". Only the last element is evaluated, since this element was printed.

The second example shows more obvious what happened. The list is only evaluated to head normal form to allow the pattern matching of *head*. Since the tail of the list is not further investigated, it is shown as unevaluated. The head on the other is evaluated by the function *print*.

Observing only data structures alone is not sufficient for debugging, at least in a functional language. Hood allows also to observe functions. At this point it becomes essential that the idea of Hood is observing rather than printing objects, because there is no way to print an arbitrary function. Observing such a function is indeed possible. To do so we can write a wrapper for our function which handles the observations. Take a look at this example:

```
import Debug.Hood.Observe

ex1 = print
      $ map f
      $ zip [1..5] [0,4,2,7,3]

f = observe "mininum_function" f'
f' :: (Int,Int) -> Int
f' (a,b) = if a < b then a else b
```

It is pretty much the same as in the first example. Remember the type of *observe*: $observe :: Observable\ a \Rightarrow String \rightarrow a \rightarrow a$. The type is arbitrary as long as it is an instance of *Observable*. Luckily the instance for the function application is predefined, which allows us to observe any function which input and output are observable.
In other words: $Instance\ (Observable\ a, Observable\ b) \Rightarrow Oberable(a \rightarrow b)$ is defined in the module Observe.

Now lets continue with our example. If we run this, it produces this output:

```
*Main> runO ex1
[0,2,2,4,3]


-- mininum function
  { \ (1, 0)  -> 0
  , \ (2, 4)  -> 2
  , \ (3, 2)  -> 2
  , \ (4, 7)  -> 4
  , \ (5, 3)  -> 3
  }
```

As before the first line is the output from the expression. Then the observations follow. Here we can see how functions are observed. Each line shows a single application of the function f'. With this information we can easily see if our

function works as expected. The aforementioned referential transparency is the reason we can determine whether the function is correct or is not correct. No context information is needed. Of course we cannot determine if the function is absolute correct, because we would need the result for every possible input, but we can the determine if the function worked correct for the inputs which where given in this run.

This observation also respects the laziness of Haskell and do not forces the evaluation of the input or result of the function. In addition it is possible to observe anonymous functions. Furthermore can we do multiple observations in one run. The following example shows how it works:

```
...
ex5 = print
      $ observe "anonymous head" (\(x:xs) -> x)
      $ map f
      $ zip [1..5] [0,4,2,7,3]
```

which leads to:

```
*Main> runO ex5
0

-- anonymous head
  { \ ( 0 : _ ) -> 0
  }
-- mininum function
  { \ (1, 0) -> 0
  }
```

We can see in the input of the anonymous function "_" occurs which (we already know) indicates that the expression was not evaluated. In addition the minimum function was only evaluated once, since the other applications results were not demanded. Note that the result of a function is always evaluated, at least to head normal form. If a function is applied, it means a value is demanded which implies its evaluation to head normal form. Of course this only holds if the function does not lead to an error. Even errors can be handled quite elegant with Hood :

```
div' :: Int -> Int -> Int
div' = observe "div" div

ex6 = print
      . reverse
      . (observe "list")
      $ map (div' 100) ([0..5]::[Int])
```

The program runs into an exception, since there is a division by zero. It would be a pity, if the debugger were not be able to report the evaluation done until the exception occurred. Hood is able to catch exceptions and handle these like the following output shows:

```
*Main> runO ex6
[20,25,33,50,100,[Escaping Exception in Code :
```

```
                    divide by zero]

-- list
   (throw <Exception> ) :  100 :  50 :  33
                        : 25 :  20 : []

-- div
  { \ 100  -> { \ 0  -> throw <Exception>
      , \ 1  -> 100
      , \ 2  -> 50
      , \ 3  -> 33
      , \ 4  -> 25
      , \ 5  -> 20
      }
  }
```

We can see the print output, which leads, due to the evaluation of *div 100 0*, to an exception. Lets take a look at the Hood outputs. We can see in the output for *list* that except for the first element all elements where evaluated correctly. The first element which led to the exception was marked as $(throw\ <Exception>)$. Sadly there is no further information about the kind of exception thrown. On the other hand there is no need for this information, since an exception usually leads to the termination of the program and thus is displayed anyways.

In the output for *div* we can see two interesting things here. At first the application of a function with two parameters. This is shown as a function which result is a function. A function again is displayed as a list containing pairs of input and result. The second interesting observation is the fact we can easily locate the source of the exception. It emerges from applying *0* to (*div 100*). This kind of representation allows the programmer to realize which data structure or function where effected by the exception.

This concludes the overview of the most important features of the debugging system Hood. Other features and an outline of its implementation are presented in the paper of Andy Gill[5].

There are two other tools based on Hood or Hoods idea of debugging. At first there is GHood which is a graphical back end to Hood. This tool could help to understand more complex observations, which can be overwhelming in a shell. The other one, called Hugs, used the idea of Hood and build another implementation aiming for a better performance and slightly better output. [6]

3 Interpreter and Dynamic Breakpoints

In this section we will look upon the possibilities GHCI[7] offers for debugging. At first we will investigate the benefits of the interpreter in general, then we will consider dynamic breakpoints for debugging. Most of you possibly already know how to use the GHCI for debugging, but for the sake of completeness we will present it nevertheless.

[6] `https://www.haskell.org/hugs/pages/users_guide/observe.html`
[7] `https://wiki.haskell.org/GHC`

3.1 Interpreter

Remember Haskell is referential transparency which ensures an expressions result only depends on the values of the subexpressions not on its environment. This allows us to use the GHCI as an debugging tool. With the aforementioned methods we may localized the bugged code, but not necessary found the bug.

For example lets assume we found a function to be erroneous, but cannot detect the error in the source code. We can now use the GHCI to check the function with different inputs. It is guaranteed that the call in out program which led to the bug behaves like the one were testing directly in GHCI. This helps to further isolate the incorrect code.

Recall the *sort* example from above:

```
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys) = if x <= y
                  then x : ys
                  else y : insert x ys

main = putStrLn (sort "program")
```

With the debugging techniques presented in Section 2 we localized *insert* as the source of error. We know that insert sometimes *deletes* letters, but we are not sure where exactly.

Out approach is to evaluate the function step by step to understand where the letter goes missing. Lets take input values which led to the error. For example *'g'* and *"ar"*. At first we can verify that the call of insert with this input values leads to the *deletion* of a letter:

```
*Main> insert 'g' "ar"
"ag"
```

Now we hope to find the bug by further investigating this application. Since the list was not empty, we are ensured the second rule was taken. [8] The next step would be to evaluate which branch of the if-statement was taken.

```
*Main> 'g' <= 'a'
False
```

In this case we know the *else-branch* was was evaluated. Besides the fact we remember *a* for building the result list, there is the recursive call of insert with the parameters *'g'* and *"r"*. Same as before the list is not empty; we investigate the second rule of insert:

```
*Main> insert 'g' "r"
"g"
```

And again we test the condition to know which branch was taken:

[8] We cannot use GHCI to simulate pattern matching. We need to do it our self to know which rule was taken.

```
*Main> 'g' <= 'r'
True
```

This indicates the *then-branch* was taken. In this example it is quite obvious that the expression in the then branch is not correct. But the expression in the then branch could have been more complex. To evaluate complex expressions by hand is destined to fail, especially when other function are called within these expressions. By using GHCI for this partial evaluation, it is ensured the evaluation of an expression is correct with respect to the semantics of Haskell. Now lets turn to a way of using GHCI for debugging you may not already know.

3.2 Dynamic Breakpoints [9]

Dynamic Breakpoints are a well known feature in imperative languages, for a good reason. They offer the possibility to stop the program if it reaches specific points in the code. After stopping the execution, the programmer is able to browse through the programs state. For example he is able to view local or global bindings. This can help to understand why a program is acting the way it does. He is even able to modify the bindings for testing purposes to see how the behavior changes by editing several variables. When the programmer finished his observations and modifications he can continue the program and it runs until it reaches a breakpoint (it may be the same) or it terminates.

In the context of Haskell breakpoints are not as useful as in imperative languages, since there is no state in a Haskell program, but they are not totally useless either. You are not able to modify bindings, when reaching a breakpoint. Nevertheless you can browse through existing bindings to verify if your expectations are met. Lets have look at the following example:

```
import Prelude hiding (filter)

qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) =
   qsort firsthalf ++ [pivotelement] ++ qsort secondhalf
  where pivotelement    = x
        firsthalf       = filter xs (<= pivotelement)
        secondhalf      = filter xs (> pivotelement)
        filter []     _ = []
        filter (y:ys) f =
            if f y then y:(filter ys f) else filter ys f


main = print $ qsort "programmer"
```

if we run this program it looks like this:

```
*Main> main
"aegmmoprrr"
```

If we want to understand how the program calculates this, we can use breakpoints. We first set a breakpoint, then we run our program:

[9] The structure of this subsection is based on the user guide `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html`

```
*Main> :break 6
Breakpoint 0 activated at break.hs:6:4-56
*Main> main
"Stopped at break.hs:6:4-56
_result :: [a] = _
firsthalf :: [a] = _
pivotelement :: a = _
secondhalf :: [a] = _
```

By using the *:break 6* we are setting a breakpoint in line six (in this case the concatenation). When we run *main*, the program is stopped at the specified point and the bindings known at that point are reported. The GHCI reports the bindings with type and evaluation as far as known at that point. It does not force its evaluation. If we are not sure which breakpoint we are at (in the case we set multiple breakpoints), we can use *:list*:

```
[break.hs:6:4-56] *Main> :list
5 qsort xs@(x:_) =
6   qsort firsthalf ++ [pivotelement] ++ qsort secondhalf
7  where pivotelement    = x
```

The command reports the line where the break point is and the lines before and after this line. Due to the layout of this paper you cannot see that the breakpoint line is marked in other color than the rest to see better where we stopped. After we know in which part of the code we are, we want to look around and explore our bindings:

```
*Main> firsthalf

<interactive>:6:1:
  No instance for (Show a) arising from a use of 'print'
  Cannot resolve unknown runtime type 'a'
  Use :print or :force to determine these types
  ...
  -- Defined in 'GHC.Real'
    ...plus 24 others
  In a stmt of an interactive GHCi command: print it
```

Since the evaluation is not far enough to know which type *firsthalf* has, we cannot print this value with the function *print . show* which is evoked every time a expression is executed in GHCI. Remember the call of show would force the evaluation of the expression. There is another way of showing what the program knows about the binding:

```
[break.hs:6:4-56] *Main> :print firsthalf
firsthalf = (_t1::[a])
```

:print is a special function of GHCI. This function displays the information about the binding as far as it knows. This means, if the value is evaluated, it would display its value as far as evaluated instead of its type.

The information we have seen so far are not very helpful. We need to force the evaluation of some bindings to get benefits from the breakpoint. There are two function to evaluate bindings. You may already know the first, namely *seq*:

```
[break.hs:6:4-56] *Main> seq firsthalf ()
()
[break.hs:6:4-56] *Main> :print firsthalf
firsthalf = 'o' : (_t2::[Char])
[break.hs:6:4-56] *Main>
```

seq forces the evaluation of its first argument to head normal form and then returns its second argument. As we can see in the example above: we forced the evaluation of *firsthalf* to head normal form with *seq*. After we used *:print*, we could see the first value of the list *firsthalf* and its type. This is handy for subexpressions which are applied to functions or case statements, because we can determine which rule will be taken.

Sometimes we want to force the full evaluation of an (sub)expressions. In order to do so, we can use the function *:force*:

```
[break.hs:6:4-56] *Main> :force firsthalf
firsthalf = "ogamme"
```

The function behaves like *:print*, but it forces the evaluation of its argument before trying to print it. This leads to the previous mentioned risks of changing the semantic. This may evokes exceptions, infinite loops or other break point.

If we finished our browsing through the bindings, we can use *:continue* to let the program resume its evaluation until it terminated or another breakpoint occurs.

This completes the overview of the most important features that GHCI offers for debugging. If you want to look deeper in this possibilities, we recommend the official documentation at `https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html`.

4 Offline Tracer

The most powerful tools for debugging a Haskell program are *Offline Tracers*. The most popular and most advanced offline tracer is Hat which will be present in the section. Hat requires a automatized source to source code transformation, before an execution is able to produce Hats trace files. These trace files can be interpret by Hat in different ways.

Buddha is a offline tracer quite similar to Hat, but will not be presented here, since there is already a decent work about it [6] and its principles are the same as the ones of Hat.

The last well known offline tracer is Hoed which is less powerful than Hat. Hoed requires annotations in the code done by the user similar to Hood. When the program is executed information are gathered. This information is used to allow a html/browser based debugging.

4.1 Hat

Currently Hat is the most powerful and advanced debugging tool for Haskell. It defines around ten ways of exploring your program. This ranges from tracing over code coverage to algorithmic bug detection. All these analyze tools require the trace files generated by a transformed version of the program. For the sake

of space we will only look upon some of the features Hat offers for debugging. [10] Therefore consider the following version of the Insert Sort example with some more bugs:

```
sort :: Ord a => [a] -> [a]
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x []     = [x]
insert x (y:ys) = if x <= y
                  then x : ys
                  else y : insert x (y:ys)

main = putStrLn (sort "program")
```

We will now use Hat to debug this program step by step. At first we need to transform our program with *hat-make*. This will generate a transformed version of our program; a version that produces the trace information needed by Hat. The behavior of the transformed program is the same as the original one. By transforming our program, Hat will also compile the program. After we ran that program, the information for Hat are generated and we can start using it.

If we run the program as it is shown here the following happens:

```
~$ ./Hat/Sort

Error: No match in pattern.
```

It seems the pattern matching failed at some point. To find out which input to which function led to this error we can use *hat-trail* which starts at the output or error of the program and let us explore the computation backwards.

```
Error: ------------------------------
No match in pattern.

Trail: -------- hat-trail 2.9 -----
```

We can now interactively see which function application this error emerged from. Additionally we can see which function called the erroneous function.

```
Error: ------------------------------
No match in pattern.

Trail: ------ Sort.hs line: 4 col: 35 -----
<- sort []
<- sort "m"
<- sort "am"
<- sort "ram"
```

We could continue this until we reach the *main* function. But we can already see where the error lies. A look at the definition of sort let us recognize we forgot the rule for the empty list.

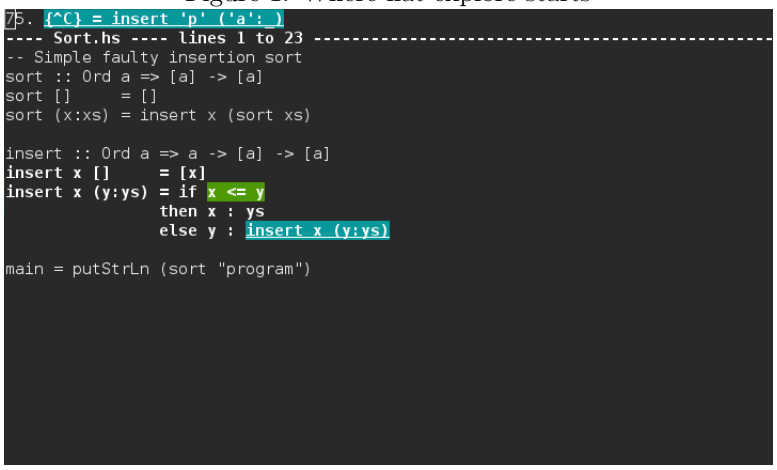
After fixing this bug, we try it again and get the following:

[10] There is a cabal package for Hat. By installing it, you will gain access to shell commands. In the following part of this subsection we will use these commands to identify the functions of Hat.

```
~$ ./Hat/Sort
aaaaaaaaaaaaaaaaaaaaaaa...^CInterrupted
```

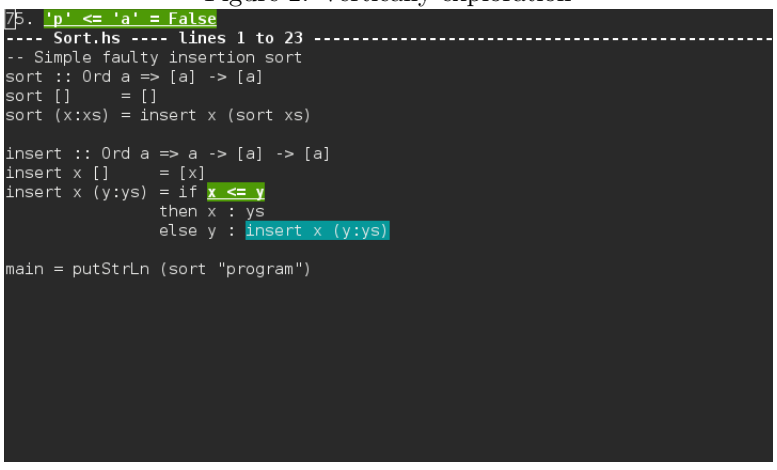
Somewhere in our program is a infinite loop. To identify where we use *hat-explore*. With hat-explore we can step through our program aided by syntax highlighting. To understand the full extend of this we will use screenshots at this point. Hat explore starts with the result shown in Fig.1.

Figure 1: Where hat-explore starts



Now we can navigate through the computation either vertically or horizontally. Horizontally means looking at the evaluation of the subexpressions such as the if condition in this example. Vertically means looking at the the expression the current one were called from. We start the exploration horizontally and get the result shown in Fig.2.

Figure 2: Vertically exploration



Here we can see evaluation of the condition which meets our expectations. Lets see what hapens if we explore vertically; it leads to the result shown in Fig.3.

Figure 3: Horizontally exploration

We can immediately see that the same piece of code where used for the evaluation. But what is more confusing is the first line. The duplication of *a* is the error we are looking for. This must be evoked by the code highlighted, since no other code where executed during this step and a second *a* occurs in the recursive call. After a quick look at the recursive call we see the problem. It should be *ys* instead of *(y:ys)*.

If we fix that bug and recompile and rerun it the following happens:

```
$ ./Hat/Sort
agop
```

This is the bug we already know from previous examples. Now we want to use Hat to localize the origin of this bug; more specifically *hat-detect*. Hat-detect allows an algorithmic bug localization. It asks the user repeatedly if a function application is correct (with respect to the intended semantic).

Hat constructs a *call-tree*[11] which logs for every function application which functions where called within this application. If a application is wrong and all its children are correct, then the rule used for this application have to be wrong (basic functions as + or * are considered to be correct). If we start hat-detect it presents us the first equation which we have to judge:

```
1  sort "program" = "agop"
hat-detect>_
```

Now its our turn to enter *n* for no (we think this equation is wrong) or *y* for yes (we think this equation is correct). We can also us *?n* or *?y*, if we are not sure about it. If we type *n* hat-detect would show us the next equation to be judged, until hat-detect finds the bug.

```
1  sort "program" = "agop"
hat-detect> n
2  sort "rogram" = "agor"
hat-detect> n
```

[11] This is not a official name.

```
3  sort "ogram" = "ago"
hat-detect> n
4  sort "gram" = "ag"
hat-detect> n
5  sort "ram" = "ar"
hat-detect> n
6  sort "am" = "a"
hat-detect> n
7  sort "m" = "m"
hat-detect> y
8  insert 'a' "m" = "a"
hat-detect> n
bug found at 4f7
    insert 'a' "m" = "a"
Done.
```

Now we know that the bug lies withing the rule of insert, which is used for *(insert 'a' "m")*. It does not seem that useful, but we not only know which inputs led to an error, but also which rule has to be wrong. It is not a subexpressions which the error emerges from, it is exactly that rule, where the bug is located. In our case it is the missed insertion of y in the *then* case. After fixing this bug the program works as intended.

Another interesting function of Hat is *hat-observe* which can be used similar to *Hood* which was presented in Section 2. Since there is a transformation of the whole program, every function will be observed and can be reported.

Even if Hat is a powerful tool for debugging, there are also some disadvantages. Due to the collection of information there is a overhead, which slows down the computation. In addition more complex programs produce much bigger trace files. Since every reduction is logged the trace files can easily consume several megabytes. In these days it does not seem that much of a problem, but it should not be underestimated, because debugging complex software systems with Hat could be impossible.

There is another point why debugging Hat may not be the best option. In order to collect its information Hat needs access to all functions used within a program. Haskell is a strongly modular language which implies the excessive use of libraries (also called packages). Hat needs also access to these libraries. That is the reason you need to recompile all libraries used within you program with Hat.

This concludes the overview on Hat and some of its features. We will now turn over to the more lightweight offline tracer: Hoed.

4.2 Hoed

Another well known offline tracer is *Hoed*. Hoed is more lightweight than Hat, thus there is no complete transformation of your program. You need to annotate the parts you want to observe in a manner similar to Hood. This can mean more work, but in return the overhead of Hoed is much smaller and there is no need to recompile libraries or other trusted modules. For the sake of space we will not give a detailed presentation of Hoed here.

After the annotation of the code, arbitrary expressions can be tested and

debugged. The core feature of Hoed for debugging is the one presented in hat-detect. After the expression were evaluated a browser based debug session can be started. All the programmer have to do is to judge equations. To simplify this, a evaluation tree can be displayed which shows which functions where called to evaluate subexpressions.

Its seems that Hat is the far better tool for debugging, since it is has much more features. There are special program where those features are needed, but most of the time the features Hoed offers (or even Hood) are enough. Usually it is enough to know which rule is bugged, especially when you have input values for which the bug occurs. The additional features to find the origin of exceptions can also be done by GHC with special compile flags to print the stack trace, even if this output is much less comfortable than the one of Hat. In addition you always have to decide which tool serves the best for localizing a specific bug.

5 Related Work

There is not much work done in the field of Haskell debugging, but there is. In [4] a offline tracer is presented which has no need to translate all modules of a program. This is important for real-world Haskell programs, because many different libraries are used which may use language features the debugging system does not support.

The work [2] describes a approach for removing type errors systematically. To find type errors in complex programs can be a difficult task. The described approach is an interactive method to remove type errors efficiently.

[1] describes a method to significantly reduce the size of trace files generated by offline tracers such as Hat. Therefore a a new semantic is introduced which uses extra information for an partial strict evaluation.

A automatized way to find deadlocks in concurrent Haskell is presented in [3]. The approach is based on the redefinition of the IO monad. This allows a structured search for deadlocks.

6 Conclusion

This paper described several ways to debug Haskell programs. At first we have seen the difficulties of debugging Haskell code. Laziness gives a not intuitive way of evaluating expression. In addition the outputs generated by *trace* can occur in a not intended order. Also Haskells purely functional character is not suitable for *printf*-debugging, because most of the time some program data rests on the function stack and is not accessible. Additionally the problem occurs that we force the evaluation of expression by printing them.

After we saw the problems which occurs when using *trace*, Hood were presented. Hood does not force any evaluations. Hood observes reductions and function applications to report them after the run of a program.

In Section 3 we showed methods to use GHCI for debugging. First we used GHCI for evaluation of single functions or expressions, then we turned over and inspected breakpoints and their possibilities.

The most powerful tools for debugging were presented in Section 4. The *offline tracers* are tools which require a transformation of the program (automatized or manually) to log information while the program is running. These information is used afterwards to allow an interactive debugging.

There is no universal tool for debugging Haskell programs, but there are a lot of tools to do so. The programmer needs to know which tool suits which problem to efficiently debug his code. For most of the problems Hood would be enough to locate the bug, but for more complex bugs offline traces may be needed. Debug tools require always a modification of the code which is costly in the manner of time when done by hand. If the modification is automatized, the programmer needs to recompile the libraries, even if no debugging is needed for these.

To debug Haskell code without tools on the other hand is not satisfactory, since you always need to keep the function stack in head to fully understand the program. This is usually not possible, since the recursion depth can be very high in functional programs.

References

[1] Bernd Brassel, Michael Hanus, Sebastian Fischer, Frank Huch, and Germán Vidal. Lazy call-by-value evaluation. *SIGPLAN Not.*, 42(9):265–276, October 2007.

[2] Sheng Chen and Martin Erwig. *Guided Type Debugging*, pages 35–51. Springer International Publishing, Cham, 2014.

[3] Jan Christiansen and Frank Huch. Searching for deadlocks while debugging concurrent haskell programs. *SIGPLAN Not.*, 39(9):28–39, September 2004.

[4] Maarten Faddegon and Olaf Chitil. Algorithmic debugging of real-world haskell programs: Deriving dependencies from the cost centre stack. *SIGPLAN Not.*, 50(6):33–42, June 2015.

[5] Andy Gill. Debugging Haskell by observing intermediate data structures. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Haskell, Technical report of the University of Nottingham*, 2000.

[6] Bernard Pope. Declarative debugging with buddha. In *Advanced Functional Programming*, pages 273–308. Springer, 2004.