Step inside the GHCi debugger

by Bernie Pope ⟨bjpop@csse.unimelb.edu.au⟩

March 6, 2008

Major releases of GHC are highly anticipated events, especially because of all the exciting new features they bring. The 6.8 series was a particularly impressive example, which came with lots of goodies, including a shiny new debugger. In this article we take the debugger out for a test run, and see what it can do.

Introduction

Anyone who follows the Haskell mailing lists for long enough knows that certain topics are recurrent. A prominent example is debugging tools, and the apparent lack thereof. See the thread entitled "modern language design, stone age tools" [1], for a memorable lamentation on the matter.

Conventional wisdom says that lazy evaluation makes it difficult to apply traditional procedural debugging techniques to Haskell. Therefore, substantial effort has been spent on researching more suitable approaches; a key outcome of this work is Hat, the Haskell Tracer [2]. Hat provides several powerful tools, covering different kinds of debugging styles, but it comes with a large performance cost, and is not particularly well integrated with the predominant programming environment (GHC), thus hampering its usability.

Recently GHC gained its own debugging tool, which was released with version 6.8.1 of the compiler. The debugger is less sophisticated than Hat, but it makes up for that by being lightweight and tightly integrated with GHCi. The design of the debugger has been directed by a "keep it simple" philosophy, which has helped to keep development costs in check. An important consequence of this pragmatism is the use of conventional procedural technology: breakpoints, and execution stepping. While it is true that lazy evaluation does cause problems for this approach, some steps have been taken to ameliorate them. In particular,

the debugger supports printing of partially evaluated terms, and the user can selectively force evaluation of suspended sub-terms from the command line.

In this article we will take a short tour of the main features of the debugger, and hopefully encourage you to try it out on your own code. As with any programming tool, there are lots of little details that have to be learned by the user, however we make no attempt to be exhaustive. Please consult the GHC User's Guide [3] for a more comprehensive reference. If you are interested in implementation issues please see the paper from the 2007 Haskell Workshop [4].

Getting started

To get started you need a recent version of GHC; anything from 6.8.1 onwards will do. The examples in this article have been tested with 6.8.2, which is the latest stable release at the time of writing. The debugger is built into GHCi, and it is always enabled. You simply start GHCi like normal and you are ready to go. In the rest of the article, interaction with GHCi is indicated like so:

```
*Main> putStrLn "Hello World"
Hello World
```

The prompt is indicated by `*Main>`, and text entered by the user is set in italics to distinguish it from interpreter output. GHCi commands can be abbreviated to a unique prefix, but here they are written in full for the sake of clarity.

One proviso is that, whilst GHCi supports two compilation modes – byte code and object code – breakpoints are only supported in byte code. This makes object code invisible in the debugger (but it is otherwise fully functional). Thus, breakpoints cannot be set inside libraries (packages), because they are always compiled to object code. In practice, this feature can be useful because it allows us to selectively turn off debugging for certain "trusted" modules. This helps us focus on the suspicious code, and improves performance because object code is roughly ten times faster than byte code.

The debugging examples in this article are based on the small program in Figure 1. It reads a dictionary of computer terms in HTML format, and searches for the definition of an input term. The dictionary data comes from the **Dictionary of Computer Terms** [5], which can be downloaded as one large HTML file [6]. The program assumes that the dictionary is saved in a file called `Dictionary.html`. The HTML is parsed into a list of tags using the TagSoup package [7], which can be obtained from hackageDB [8]. The (rather naive) search algorithm has two parts. First, it scans the list of tags for the first occurrence of text which matches the search term. Second, it scans the remaining list for the next item of text. If

```
module Main where
import Text.HTML.TagSoup
import Maybe

dict = "Dictionary.html"

main = do
   putStr "Enter a term to search for: "
   term <- getLine
   html <- readFile dict
   let soup = parseTags html
   putStrLn $ display $ searchTerm term soup

searchTerm :: String -> [Tag] -> Maybe String
searchTerm term (TagText text : rest)
   | term == text = searchDef rest
   | otherwise = searchTerm term rest
searchTerm term (_ : rest) = searchTerm term rest
searchTerm term [] = Nothing

searchDef :: [Tag] -> Maybe String
searchDef (TagText def : _) = Just def
searchDef (tag : rest) = searchDef rest
searchDef [] = Nothing

display :: Maybe String -> String
display = maybe "No match found." id
```

Listing 1: A program which searches for the definition of a term in a HTML dictionary.

found, the second piece of text is assumed to be the definition of the term. Despite the simplicity of the code, it works correctly in many cases. For example, it returns the expected output on inputs such as `C++` and `Fortran`. However, it is not without fault. Searching for the definition of `C` (the programming language) returns `D`, which is not supposed to happen – we will use the debugger to look into this later in the next section. To be fair, it is not the most exciting bug in the world, but it does allow us to show off some of the debugger's functionality without getting bogged down in details.

Breakpoints and single stepping

We can step through a computation, one expression at at time, using the `:step` command:

```
*Main> :step main
```

The `:step` command has two modes of operation. In one mode – demonstrated here – it takes an expression as an argument. The expression is used to start a new computation with single stepping activated. In the other mode – which we shall see shortly – it takes no arguments, and continues single stepping from a breakpoint within an existing computation. As you would expect with single stepping, evaluation stops as soon as it reaches an expression which is associated with a breakpoint (regardless of whether the breakpoint is activated or not).

In our example, the first encountered breakpoint expression is the body of `main`. When the computation stops at a breakpoint, control returns to the GHCi command-line, which is indicated like so:

```
Stopped at Main.hs:(7,7)-(12,43)
_result :: IO () = _
[Main.hs:(7,7)-(12,43)] *Main>
```

Note that the prompt has changed to indicate that we are now inside a debugging session. The location of a breakpoint expression is indicated using its source code span. In this case, the body of `main` spans the coordinates `(7,7)-(12,43)` in the file `Main.hs`. Coordinates are written as line-column pairs.

The source code of the current breakpoint expression can be printed using the `:list` command:

```
[Main.hs:(7,7)-(12,43)] *Main> :list
6
7  main = do
8    putStr "Enter a term to search for: "
```

```
9     term <- getLine
10     html <- readFile dict
11     let soup = parseTags html
12     putStrLn $ display $ searchTerm term soup
13
```

The breakpoint expression is highlighted (using an underline in this article), and shown within a few lines of context.

Manually listing the code for every new breakpoint can become tedious. Fortunately, GHCi can automatically execute commands on our behalf whenever a breakpoint is reached, using the `stop` setting:

```
[Main.hs:(7,7)-(12,43)] *Main> :set stop :list
```

This instructs GHCi to execute `:list` whenever execution stops at a breakpoint (some people like to put this setting in their GHCi configuration file). Of course other actions are possible by changing the setting appropriately. The effect of the new `stop` setting can be seen when we take another step in the execution:

```
[Main.hs:(7,7)-(12,43)] *Main> :step
Stopped at Main.hs:8:3-39
_result :: IO () = _
7  main = do
8     putStr "Enter a term to search for: "
9     term <- getLine
```

Now the code of the breakpoint is automatically listed, which happens to be the first statement of the do-block in the body of `main`. Unlike previously, `:step` was not given any arguments in this instance, hence execution continued from the last breakpoint.

Most debuggers for imperative languages allow the user to choose between stepping **into** function calls, and stepping **over** them. It is not so easy to step over a function call in a lazy language because function calls are not always evaluated in a depth-first fashion. Consequently, the debugger does not provide this functionality. However, restricted versions of the `:step` command are available, which can limit the visited expressions to a module or a top-level declaration (`:stepmodule` and `:steplocal`).

By now you have probably noticed the appearance of the following line at each of the breakpoints:

```
_result :: IO () = _
```

This is a fresh variable (conjured up by GHCi) which is bound to the value of the current breakpoint expression. The type of the variable is displayed along with its

value. In this particular case `_result` has type `IO ()`. Values which have not yet been computed (called thunks) are printed as underscores, as is the case above. As we shall see shortly, the free variables of the current breakpoint expression are also bound at the debugging prompt, and they are displayed in a similar fashion.

One of the most useful aspects of the debugger is that the prompt provides the full power of GHCi. At each breakpoint the `_result` of the expression and its free variables are in scope. Having the debugger integrated into the interpreter means that we can manipulate those intermediate values as first class citizens. That is to say, we can not only print those values, but we can compute with them too! For example, `_result` is currently bound to an `IO` action, and although it cannot be printed as a term (because it is abstract), we can run the action and observe its effect simply by evaluating it:

```
[Main.hs:8:3-39] *Main> _result
Enter a term to search for:
```

The ability to compute with intermediate values provides a powerful mechanism for observation. We can issue customised commands for inspecting values, using the full expressiveness of Haskell. This is especially useful for large and/or complex structures which are too unwieldy to print in entirety. It is also worth noting that when the values bound at the prompt are functions, they can be called in the normal way, allowing us to observe their behaviour on whatever arguments we choose.

An interesting feature of the debugger is that it supports nested breakpoints. This is necessary because we can evaluate arbitrary expressions at the debugging prompt, including ones which themselves have breakpoints. For example, from the current breakpoint we can start single stepping in a different expression:

```
[Main.hs:8:3-39] *Main> :step display (Just "foo")
Stopped at Main.hs:27:10-35
_result :: Maybe String -> [Char] = _
26  display :: Maybe String -> String
27  display = maybe "No match found." id
28
... [Main.hs:27:10-35] *Main>
```

The ellipsis in the new prompt indicates that we have stopped at a nested breakpoint, and the previous breakpoint is now pending. An arbitrary number of nestings is supported, and GHCi keeps a stack of pending computations. When the top computation on the stack is completed, the next one down is resumed. Obviously it can become difficult to keep track of the stack of computations in your head, therefore GHCi provides a command for printing it out:

```
... [Main.hs:27:10-35] *Main> :show context
--> main
  Stopped at Main.hs:8:3-39
--> display (Just "foo")
  Stopped at Main.hs:27:10-35
```

The current debugging computation can be discarded using the `:abandon` command:

```
... [Main.hs:27:10-35] *Main> :abandon
[Main.hs:8:3-39] *Main>
```

Notice the change in the prompt which indicates we have returned to the earlier breakpoint, and no more breakpoints are pending. Abandoning from this point discards the whole debugging session and take us back to the normal GHCi prompt:

```
[Main.hs:8:3-39] *Main> :abandon
*Main>
```

For any sizable computation, single stepping is only going to get us so far, and we need to be able to take leaps as well. As would be expected, the debugger allows us to toggle breakpoints at specific program locations, and continue execution until an active breakpoint is encountered. The `:break` command activates breakpoints, providing a number of ways to refer to program locations, such as function names, line numbers, and source spans. One line of code can contain multiple breakpoint expressions, so we have to be careful with how the locations are specified. The rules for picking an appropriate breakpoint location are somewhat intricate; see the section on setting breakpoints in the User's Guide for a thorough treatment.

As noted earlier, there is a bug in the example program which is triggered when we try to search for the definition of the term `C`. It returns the string `D`, when we expect to see something which defines the C programming language.

To diagnose the bug we want to find out what happens just after the text tag containing `C` is found in the list of HTML tags. Therefore, a good place to set a breakpoint is on line 16, on the expression '`searchDef rest`'. At this point we know the term has been found, and we can try to determine what goes wrong in the search for the definition:

```
*Main> :break 16 20
Breakpoint 0 activated at Main.hs:16:20-33
```

In this case it is necessary to specify both the line number **and** the column number of the expression, because there is another breakpoint expression on the same line: the left-hand-side of the guarded equation. If we were to specify the line number

alone, GHCi would choose the leftmost subexpression that begins and ends on that line, which is not the one we want. Therefore, the column number is needed to disambiguate. We have chosen the starting column of the expression, but any column inside its span will do.

Now we can run the program and wait for the breakpoint. There are no special commands to invoke, we just do it in the normal way:

```
*Main> main
```

The program behaves as usual, and we are prompted for input:

```
Enter a term to search for: C
```

After a short amount of time the breakpoint is reached:

```
Stopped at Main.hs:16:20-33
_result :: Maybe String = _
rest :: [Tag] = _
15  searchTerm term (TagText text : rest)
16      | term == text = searchDef rest
17      | otherwise = searchTerm term rest
```

Note that, in addition to `_result`, the variable `rest` is bound at the prompt, because it is free in the current breakpoint expression.

At this point we know that the search term has been located, and from the behaviour of the program, we know that the wrong definition is eventually returned. To help us understand the cause of the problem, we want to know what tags appear just after the search term, up until the erroneous definition is found (the next `TagText` token). We could print out the value of `rest`, but as a precaution, we first check its length:

```
[Main.hs:16:20-33] *Main> length rest
302250
```

Clearly it is far too big to print the whole list. However, we only really want to view a prefix of the list, up until the next `TagText`. We can find out exactly how big that prefix is:

```
[Main.hs:16:20-33] *Main> length (fst (break (~== TagText "") rest))
6
```

This expression splits `rest` into two parts – (1) everything before the first `TagText` token, and (2) everything else – and it returns the length of the first part. It uses the partial match operator `~=` from the TagSoup library, along with `fst`, `break` and `length` from the Prelude. Fortunately the prefix is short, so we can safely print it out:

```
[Main.hs:16:20-33] *Main> take 7 rest
[TagClose "a",TagClose "span",TagClose "dt",TagOpen "dt" [],
TagOpen "span" [("class","sect1")],TagOpen "a" [("href","#d")],
TagText "D"]
```

Obviously we are looking in the wrong place in the dictionary for the definition of the C programming language. A quick search for `"#d"` in the original HTML file reveals the cause of the problem: `TagText "C"` first appears in the file in the alphabetic index of terms near the beginning. No wonder the next `TagText` corresponds to `D`. The bug stems from an incorrect initial assumption about the structure of the HTML file. We can fix the bug in a number of ways, for example, by providing a more specific set of tags in the pattern match of `searchTerm`. However, the appropriate fix is something of a moot point in the context of this article, and we will leave it open for debate.

Normally the bug would not be found so easily, and it is likely that we will have multiple active breakpoints. The computation at the current breakpoint can be resumed using `:continue`, and as expected, execution continues until the next activated breakpoint is encountered or the computation ends. The current set of activated breakpoints can be displayed using `:show breaks`, and breakpoints can be deactivated using `:delete`.

Before this section concludes, there is one last issue to discuss, relating to the variables that are bound at a breakpoint. Notice in the previous breakpoint, that the variable `rest` was bound at the prompt, but, perhaps unexpectedly, the local variables `term` and `text` were not bound. Attaching the set of **all** in-scope variables to a breakpoint seems like a useful thing for the debugger to do, but it introduces unfortunate performance penalties (see Section 4.3 of [4] for details). However, attaching only the free variables of the breakpoint expression avoids these costs, and a design decision was made accordingly. In some cases we really need to see the values of other variables, besides the free ones, to understand what is happening in the program. Thankfully there is a simple hack to work around the problem. We can fake the needed free variables using `const`. For example, here is a modified version of line 16 of the example program that includes `term` and `text` as free variables in the right-hand-side expression:

```
  | term == text = const (searchDef rest) (term, text)
```

This modification does not change the behaviour of the program in any significant way because `const` returns its first argument as its result, discarding its second argument altogether; lazy evaluation ensures that no extra work is done. An optimising compiler could easily undo the ruse by inlining the definition of `const`, but happily GHCi does not. Having to manually modify the program source code for this purpose is fairly distasteful, and hopefully a more palatable solution will emerge in future versions of the debugger.

Printing values and forcing evaluation

Due to lazy evaluation, it is quite often the case that the variables bound at breakpoints are thunks, or are only partially evaluated. This is particularly so for `_result`, which is normally bound to a yet-to-be-evaluated expression. In the example above, when we wanted to print the value of something, we just evaluated a Haskell expression containing the appropriate variable. However, this can have undesirable consequences:

- It could raise an exception or cause an unwanted side-effect.
- It might require a lot of unnecessary computation.
- It might trigger nested breakpoints (which can be confusing if you aren't expecting them).
- It will probably change the order in which things are evaluated, which may not always be desirable.

To avoid these potential problems, the debugger provides the `:print` command that conserves the state of evaluation of its argument. For example, suppose we were at the last breakpoint in the example above, and we wanted to observe the value of `rest`. Initially it is a thunk, and if we ask `:print` to display its value, we get this output:

```
[Main.hs:16:20-33] *Main> :print rest
rest = (_t1::[Tag])
```

The output is fairly uninformative at the moment because `rest` is unevaluated, but crucially it shows the term's current state of evaluation without forcing it further. An important side-effect of `:print` is that it binds any thunks in the printed term to fresh variables, which enables us to refer them as separate entities. In this case `_t1` is bound to the one and only thunk, though it is admittedly a boring example because it is just an alias for `rest` – more interesting examples follow.

Oftentimes we will want to selectively evaluate some thunks in order to make sense of the term being printed. This can be done using the `seq` primitive function:

```
[Main.hs:16:20-33] *Main> seq _t1 ()
()
```

Operationally, `seq` forces the evaluation of its first argument to weak head normal form (WHNF) before returning its second argument.[1] Here we are only interested in the evaluation side-effect of `seq`, so the value of the second argument is immaterial, and `()` plays the role of placeholder perfectly.

[1] Informally, a term is in WHNF if it is: a manifest function, a partial application, a saturated application of a data constructor, a nullary constructor, or a primitive value. The arguments of applications and the bodies of functions do not have to be in WHNF themselves.

The use of `seq` to evaluate parts of a term is sufficiently common to warrant its own command. GHCi does not provide such a command, but we can define it ourselves using `:def` macro facility:

```
:def seq (\arg -> return ("seq (" ++ arg ++ ") ()"))
```

(It is a good idea to put this definition in your GHCi configuration file.) This defines a new command called `:seq` (you can call it what you want), that takes an argument string `arg`, and executes '`seq (arg) ()`'. The parentheses around `arg` ensure syntactic correctness in case `arg` is a compound term. The use of `return` in the definition is needed because `:def` requires an `IO` function.

Printing `rest` again reveals that it is now slightly more evaluated:

```
[Main.hs:16:20-33] *Main> :print rest
rest = TagClose (_t2::String) : (_t3::[Tag])
```

Now we have an application of the list constructor to two arguments. The head element of the list is an application of `TagClose` to a thunk of type `String`. The tail of the list is a thunk of type `[Tag]`. The newly uncovered thunks are bound to fresh variables, allowing us to continue selectively evaluating parts of the term, like so:

```
[Main.hs:16:20-33] *Main> :seq _t2
()
[Main.hs:16:20-33] *Main> :print rest
rest = TagClose ('a' : (_t4::[Char])) : (_t5::[Tag])
```

As you can see, the presence of variables in the output can often inhibit readability, so a simpler alternative is provided by `:sprint`, which just prints underscores in the place of thunks:

```
[Main.hs:16:20-33] *Main> :sprint rest
rest = TagClose _ : _
```

Normally `:print` uses basic Haskell syntax to display terms. It can also be configured to use the `Show` instance for the type of the term – if one is available, and the term is fully evaluated – with the command `:set -fprint-evld-with-show`.

```
[Main.hs:16:20-33] *Main> :seq _t4
()
[Main.hs:16:20-33] *Main> :print rest
rest = TagClose ['a'] : (_t6::[Tag])
[Main.hs:16:20-33] *Main> :set -fprint-evld-with-show
[Main.hs:16:20-33] *Main> :print rest
rest = TagClose "a" : (_t7::[Tag])
```

Notice that after the setting was made, the string argument of the `TagClose` constructor was printed as `"a"`, using the `Show` instance for `String`, whereas it was previously printed as `['a']`.

Sometimes we just want to print the final value of a term, without having to `seq` all the thunks manually. There is a danger of triggering new breakpoints if we just evaluate the term at the command line, which can be quite irritating. Therefore the debugger provides the `:force` command, that evaluates a term completely, and prints it out, but with all breakpoints temporarily disabled. Some care needs to be taken with `:force`, because it might cause a lot of computation to happen, and may even result in divergence.

History tracing and exceptions

Two of the more difficult to diagnose bugs in Haskell programs are unhandled exceptions (such as the ubiquitous pattern-match-failure bugbear), and infinite loops. Programmers using strict languages are accustomed to finding such bugs using stack traces, but such a feature is difficult to implement in a lazy language. In fact there are two kinds of stack traces that one might want in a lazy language: one based on the dynamic stack, which shows the current evaluation context, and another one based on a reconstructed lexical call stack, which reflects the dependencies of the source code. Neither of these kinds of trace is currently available in the debugger, although the developers are ruminating on some ideas at the moment. However, a basic kind of tracing facility is provided which is intended to provide some insight into these kinds of problems. When tracing is turned on, the debugger keeps a list of the N most recently visited breakpoint locations.[2] This list can be inspected at a breakpoint, thus providing information about what the program was doing just prior to when it stopped.

We will consider a simple example of tracing by introducing an infinite loop into the example program. Suppose that we inadvertently modified line 18 of the program from:

```
searchTerm term (_ : rest) = searchTerm term rest
```

to:

```
searchTerm term rest = searchTerm term rest
```

Clearly `searchTerm` will diverge if this equation is ever evaluated.

To diagnose this kind of bug we must be able to make an educated guess about when the program has entered the infinite loop. We can then turn the loop into an

[2] N is fixed at fifty at the moment, though it probably should be configurable.

exception by sending an interrupt signal to the program (Control-C on Unix) at an appropriate time. The debugger can stop on exceptions as if it had encountered an ordinary breakpoint, which allows us to inspect the trace to find out what the program was doing just before it was interrupted. The `-fbreak-on-error` flag tells GHCi to stop on a fabricated breakpoint when an uncaught exception is raised:

```
*Main> :set -fbreak-on-error
```

There is also a related flag called `-fbreak-on-exception` that causes execution to stop on a breakpoint whenever an exception is raised, regardless of whether it is eventually caught by the program.

Tracing imposes a significant runtime penalty, and so it is turned off by default. It is turned on by the `:trace` command, which is bimodal in the same way as `:step` – you can either start tracing a new computation, or you can begin tracing from within an existing one. In this example we will run the program from the beginning with tracing turned on:

```
*Main> :trace main
```

The program behaves as normal and we are asked to enter input:

```
Enter a term to search for: C
```

From here on the computation grinds away, and at some point we suspect a loop has been entered, so we interrupt the program by hitting Control-C:

```
^C
Stopped at <exception thrown>
_exception :: e = GHC.IOBase.DynException
                    (Data.Dynamic.Dynamic _
                     ghc-6.8.2:Panic.Interrupted)
[<exception thrown>] *Main>
```

Now the computation has stopped at an exception breakpoint and control has returned to the GHCi command line. Notice that the breakpoint is slightly different from a normal one:

- It does not have a source code location because the exception corresponds to an event rather than a particular expression in the program.
- A special variable called `_exception` is introduced which is bound to the value of the exception, allowing us to see which particular exception was thrown.

We can get an overview of the trace using the `:history` command:

```
[<exception thrown>] *Main> :history
-1  : searchTerm (Main.hs:18:23-42)
-2  : searchTerm (Main.hs:(15,0)-(19,27))
-3  : searchTerm (Main.hs:18:23-42)
-4  : searchTerm (Main.hs:(15,0)-(19,27))
-5  : searchTerm (Main.hs:18:23-42)
...
```

For the sake of the article, the output above is truncated at five entries, but in practice it shows the twenty most recent trace locations. The exact number of entries to be viewed can be specified as an argument to `:history`, and currently up to fifty are recorded.

Notice that the trace entries are numbered from `-1` backwards. The idea is that `-1` indicates the breakpoint just prior to the current one, and `-2` is the one before that, and so on. Each entry in the trace records all the information of the corresponding breakpoint, including bound variables, and we can traverse the list of entries using the `:back` and `:forward` commands. For example, taking two steps back and one forward brings us to entry number `-1`:

```
[<exception thrown>] *Main> :back
Logged breakpoint at Main.hs:18:23-42
_result :: Maybe String
rest :: [Tag]
term :: String
[-1: Main.hs:18:23-42] *Main> :back
Logged breakpoint at Main.hs:(15,0)-(19,27)
_result :: Maybe String
[-2: Main.hs:(15,0)-(19,27)] *Main> :forward
Logged breakpoint at Main.hs:18:23-42
_result :: Maybe String
rest :: [Tag]
term :: String
[-1: Main.hs:18:23-42] *Main>
```

We can print out the values of the bound variables, which can provide useful information about why the exception (or loop) is occurring:

```
[-1: Main.hs:18:23-42] *Main> :print rest
rest = TagOpen ('h' : (_t1::[Char])) (_t2::[Attribute]) :
                                     (_t3::[Tag])
```

For a simple example like this one it is not too hard to spot the loop in the trace, but real bugs can require a lot more investigative work. A common problem is that

the trace gets polluted by noise, because many innocent functions are called near the point of exception. A classic example is when combinator functions are used, such as those in a custom state monad. Generally the combinator functions are small and easy to verify for correctness, but they used to glue more complex parts of the program together. The trace can easily be awash with breakpoints for the combinator functions because they are called frequently. A useful workaround for this situation is to compile the module containing the combinators to object code. That way, no breakpoints are generated for the combinators, thus removing their entries in the trace. Of course this presupposes that the combinators are found in a separate module, and so some refactoring might be needed in practice.

Runtime type inference

So far we have shown the operation of the debugger in a fairly good light, but there is one thorny issue that needs to be discussed. As noted earlier, variables which are bound at a breakpoint have type information associated with them, but where does that information come from? As it happens, GHCi does not attach detailed source-level type information to terms at runtime. Therefore, type information has to be reconstructed from two sources:

1. The internal representation of a term.
2. The static type environment which results from typechecking the program.

The reconstruction of a term's type is called runtime type inference (RTTI). The possibility of thunks inside a term means that sometimes only partial type information can be gleaned from the representation alone. Also, terms can appear in polymorphic contexts – such as the arguments to polymorphic functions – which means that types inferred from the environment are not always concrete. A consequence of these limitations is that, in some circumstances, RTTI is unable to determine the most specific type for a term.

For a contrived example, suppose we have the two-tuple swap function defined in a file called `F.hs`:

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

We can take two steps in the execution of a call to `swap` like so:

```
*Main> :step swap (not True, not False)
Stopped at F.hs:2:0-19
_result :: (b, a) = _
[F.hs:2:0-19] *Main> :step
Stopped at F.hs:2:14-19
```

```
_result :: (b, a) = _
x :: a = _
y :: b = _
```

This takes us to a breakpoint corresponding to the body of `swap`. Notice that `x` and `y` are bound to thunks, and that their types are variables. We happen to know that `x` and `y` will evaluate to booleans, but RTTI is not privy to that information; it just sees thunks (which provide no type information) used within a polymorphic context.

We get an error if we try to evaluate one of those thunks at the command line:

```
[F.hs:2:14-19] *Main> x
<interactive>:1:0:
    Ambiguous type variable ‘a’ in the constraint:
      ‘Show a’ arising from a use of ‘print’ at <interactive>:1:0
    Cannot resolve unknown runtime types: a
    Use :print or :force to determine these types
```

GHCi inserts a call to `print` in front of `x` as part of its standard way of evaluating expressions. The `print` function calls `show`, and therefore the type of `x` must be an instance of the `Show` class. However, the type of `x` is a variable, which is not an instance of `Show`, hence the ambiguity in the type of the expression.

In general, inferred types become more specific as terms become more evaluated. Therefore, it often helps to `seq` a few thunks in a term to get better type information:

```
[F.hs:2:14-19] *Main> :seq x
()
[F.hs:2:14-19] *Main> :type x
x :: Bool
[F.hs:2:14-19] *Main> x
False
```

Forcing the evaluation of `x` turns the thunk into a data constructor (`False`), which provides enough information for RTTI to determine that the type of `x` is `Bool`.

A related problem occurs because of Haskell's `newtype` construct. Recall that `newtype` introduces a distinct type in the program, which has the same runtime representation as an existing type. For example:

```
newtype Velocity = MkVelocity Double
```

This creates a new type called `Velocity`, with a constructor `MkVelocity`. Owing to the semantics of `newtype`, the `MkVelocity` constructor is erased from the program

at runtime, and thus the runtime representation of `Velocity` is the same as `Double`. Obviously this has an effect on RTTI, because it uses the representation of terms as a source of type information. When the static type environment provides no extra information about a term, the debugger can infer the wrong type. For example it might infer `Double` instead of `Velocity`.

In situations where we know what the type of a term should be, but RTTI produces something which is less specific, or incorrect, we can use `unsafeCoerce#` to obtain the desired type. For example, suppose that the RTTI had determined the type of a variable `v` to be `Double`, but we know it should be `Velocity`. We can coerce the type like so:

```
*Main> :set -fglasgow-exts
*Main> let v' = GHC.Prim.unsafeCoerce# v :: Velocity
```

It is a bit tedious to type all this out, so it is useful to create a canned version using GHCi's `:def` facility – we leave this as an exercise for the enthusiastic reader. The effect of the type conversion can be seen when we print out the values of `v` and `v'`:

```
*Main> v
3.2
*Main> v'
MkVelocity 3.2
```

It is anticipated that future versions of the debugger will include an improved mechanism for determining the types of values bound at breakpoints, which avoids some of the problems identified above.

Future work and conclusion

Hopefully by now you have a good idea about what the debugger is able to do, and you might consider using it in the future. The implementation is still relatively new, and there are many ways in which it could be extended. To a large extent, new features will be directed by feedback from users; the more experience people have with the tool, the better it will become.

Here is a non-exhaustive list of some ideas that might be developed in the future:

1. The debugger works with multi-threaded programs, but it isn't particularly targeted to the kinds of bugs that concurrency brings, such as race conditions. It would be useful to have more control over thread scheduling, and also the ability to look at the state of live threads, and perhaps even inspect their stacks.
2. The debugger – and indeed GHC in general – provides a full featured API. It is envisaged that other interfaces could be developed, and perhaps the debugger can be integrated into existing IDEs.

3. As noted above, stack tracing is a very desirable feature to have, but it is not yet available. One possibility is to reuse some of the infrastructure already provided by the runtime profiling system, but other options are also possible, such as some kind of program transformation done in the early stages of compilation.
4. Currently thunks are displayed as underscores, which provides no information about their values. We can force their evaluation using `seq`, but this is not always desirable. The runtime representation of byte-code-compiled thunks does have source and free variable information attached to it, and work is currently under way to make that information available to the user.

If you see the need for a particular feature in the debugger, don't be afraid to make a suggestion, and remember that GHC is open source, so contributions are always welcome.

Acknowledgments

Thank you to José Iborra (Pepe) and Simon Marlow for reading earlier drafts of this article, and making many helpful suggestions. Thank you to Wouter Swierstra for agreeing to accept the article and waiting so patiently while I dilly-dallied. Thank you to Microsoft Research for giving me the opportunity to work on the debugger during a most enjoyable internship in 2007, and thank you to Simon Marlow for being an excellent supervisor. Thank you to Peng Li and Srdjan Stipic for befriending me during the internship, and for helping me hack on GHC. I can highly recommend Microsoft's internship program to anyone who is eligible, and Cambridge is a great place to spend a few months, even in the winter. GHC is the product of many dedicated people, and we, the Haskell Community, benefit greatly from their efforts; thank you to all those who have worked so hard.

References

[1] Modern language design, stone age tools. `www.cse.unsw.edu.au/~dons/haskell-1990-2006/msg17236.html`.

[2] Hat - the haskell tracer. `www.haskell.org/hat/`.

[3] The glorious glasgow haskell compilation system user's guide. `www.haskell.org/ghc/docs/latest/html/users_guide/`.

[4] Simon Marlow, José Iborra, Bernard Pope, and Andy Gill. A lightweight interactive debugger for Haskell. In **Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell**, pages 13–24. ACM, New York, NY, USA (2007).

[5] Computer dictionary project. computerdictionary.tsf.org.za/.

[6] Computer dictionary project, single html file. computerdictionary.tsf.org.za/dictionary/terms/computerdictionary-all.html.

[7] Tag soup. www-users.cs.york.ac.uk/~ndm/tagsoup/.

[8] hackagedb. hackage.haskell.org/packages/hackage.html.