A Lightweight Interactive Debugger for Haskell

Simon Marlow

Microsoft Research
simonmar@microsoft.com

José Iborra

Universidad Politécnica de
Valencia
jiborra@dsic.upv.es

Bernard Pope

The University of Melbourne
bjpop@csse.unimelb.edu.au

Andy Gill

Galois, Inc.
andy@galois.com

Abstract

This paper describes the design and construction of a Haskell source-level debugger built into the GHCi interactive environment. We have taken a pragmatic approach: the debugger is based on the traditional stop-examine-continue model of online debugging, which is simple and intuitive, but has traditionally been shunned in the context of Haskell because it exposes the lazy evaluation order. We argue that this drawback is not as severe as it may seem, and in some cases is an advantage.

The design focuses on *availability*: our debugger is intended to work on all programs that can be compiled with GHC, and without requiring the programmer to jump through additional hoops to debug their program. The debugger has a novel approach for reconstructing the type of runtime values in a polymorphic context. Our implementation is light on complexity, and was integrated into GHC without significant upheaval.

1. Introduction

In 2005, the GHC team surveyed its users, seeking input about their experiences with the compiler and the features which they most desired in future releases. By far the most common request was for a debugger. Wadler cites the lack of good debugging tools as one of the reasons that "Nobody uses Functional Languages" (Wadler 1998).

Haskell programmers have managed for a long time without extensive debugging support because:

- The type system eliminates a large class of bugs at compile time.

- Unit testing with Quickcheck (Claessen and Hughes 2000) and HUnit (Herington 2002) allow us to test individual components of our software, thus narrowing down the source of bugs and in many cases rendering a full debugger unnecessary.

- The interactive environments GHCi and Hugs let the user test individual functions in isolation, a kind of interactive unit testing.

Seasoned Haskell programmers often say that "when the code compiles, it usually works", which may well be true for "gurus" who follow good coding practices, but there are reasons that the above toolchain isn't always enough:

- New users often want a way to visualise and understand what is going on in their programs, as a way to learn the language. Even with an interactive interpreter, Haskell seems like a black box. Laziness in particular is an obscure concept for someone used to imperative languages. Existing debuggers even go to some trouble to hide laziness from the user. Experienced users can also struggle to comprehend the dynamic properties of complicated and unfamiliar code.

- Unit test failures can tell us that something is wrong in the program, but they do not always tell us *why*. Diagnosing the cause of such errors requires extra mental exertion from the programmer, which can be quite significant in the case of large systems.

- Whilst the Haskell type system is powerful, it does not catch all the defects in our code. Uncovered calls to partial functions are a common source of bugs. The best-known example is `head []`, which results in summary termination of the program and an unhelpful error message, stating only that `head` was called with a `[]` argument. Various techniques have been proposed to uncover this kind of mistake (Mitchell and Runciman 2007; Xu 2006), but the fact remains that the programmer often needs extra support to find the location of such errors.

GHC users resolutely requested a debugger, even though a variety of Haskell debuggers already existed, some even working with GHC. A large amount of research, including several PhDs, has gone into debugging functional languages in general and Haskell in particular (Nilsson 1998; Sparud 1999; Pope 2006; Ennals and Peyton Jones 2003a). The most prominent working debuggers for Haskell are Hat (Chitil et al. 2002) and Hood (Gill 2000). Hat is a sophisticated system, with a suite of powerful tools, based on solid research. So why is it apparently not being used by GHC users? The primary reason, in our opinion, is a lack of *accessibility*, namely:

- Hat employs a source-to-source transformation to turn a plain Haskell program into one with debugging support. Every module of the program, including all libraries, must be transformed in this way and recompiled in order to debug the application. A common complaint is that Hat doesn't come with support for a particular library. With the number of third-party libraries available for Haskell growing rapidly, the chance that a given program won't work with Hat is quite high.[1]

- Hat supports an extended subset of Haskell 98 (i.e. not all Haskell 98 features, plus some extensions to Haskell 98). Many widely-used features and extensions are not supported.

- Hat is a post-mortem debugger: you run the program, and then run tools that inspect the generated trace file. It therefore works only on complete programs, and the program must finish be-

---

[1] The Hat developers plan to address this issue in a future version.

fore it can be debugged. This approach has both advantages and disadvantages: it is liberating to be able to examine the entire computation without regard for evaluation order, but this requires storing the full history of the execution, which can be prohibitive. This implies that programs which do not terminate can't be directly debugged with Hat.

- It is not possible to inspect intermediate values in the trace using Haskell functions (e.g. `show`), only the default printer can be used. When the values are large, this may be inconvenient; for example, it is much easier to determine whether a large `Map` contains a particular key by invoking the `lookup` function than it is to inspect the runtime representation of the `Map`. In general, the lack of an interactive Haskell evaluator when debugging the program is a drawback.

- Hat imposes a significant run time cost: from 50 to 150 times slower execution. And moreover, due to the space taken by the trace, Hat cannot be used to debug medium sized programs (Silva 2007b).

- Hat doesn't come with GHC. This is something we could fix, but it doesn't seem appropriate without support for the whole GHC language. Also, the requirement to provide traced versions of all libraries is prohibitive.

There is a gaping hole in Haskell tools market for a debugger that offers a greater level of accessibility, perhaps trading some of the power of Hat for immediacy. This is what we have tried to achieve with our debugger, which has the following properties:

- It provides a simple, interactive, imperative-style debugging model, where the user can stop the execution of the program at a particular location in the source code and examine the values of variables. Execution can be single-stepped, and the debugger also provides a simple step-backwards facility to help find the source of exceptions.

- Any Haskell program that can be loaded into GHCi can be debugged (virtually any program that can be compiled with GHC can also be loaded into GHCi). There are no restrictions on language extensions or libraries that may be used by the program being debugged. Use of some extensions[2] may result in a degraded debugging experience, but in general they don't prevent the program as a whole from being debugged. Libraries are always compiled to machine-code and therefore may not be debugged, although it is possible to inspect runtime values constructed by library code.

- The debugger imposes a performance penalty on GHCi, of around 15% at compile-time and 45% at runtime (Section 4.7). We consider this acceptable, given that the debugger allows the program to be stopped at any subexpression. GHCi's interpreter is already a factor of 10-15 slower than compiled code, so an extra 45% here is not significant, and compile time tends to be more important than runtime performance during development.

- Because our debugger is integrated into GHCi, the user has the full power of an interactive Haskell evaluator to inspect intermediate values during debugging. Moreover, since the programmer is likely to be using GHCi as a development tool in any case, the debugger is available at their fingertips without having to switch tools.

- Our debugger has a programmatic interface, which is part of the GHC API. GHCi itself is a client of the GHC API, providing a text-based interface to GHC's interactive evaluation and debugging features. So while we are currently presenting

the GHCi debugger's textual user interface, this is by no means hard-wired into the implementation, and we fully expect tools with richer user interfaces to emerge in the future. The development environments Visual Haskell (Angelov and Marlow 2005), hIDE[3] and Shim[4] already use the GHC API; they can now provide integrated debugging features too.

- The implementation is lightweight: we share code with HPC, the Haskell Program Coverage tool (Gill and Runciman 2007), and there were no modifications to the compiler outside the HPC code, the bytecode compiler/evaluator, the GHC API, and the GHCi front-end.

Our debugger makes no attempt to hide the effects of laziness from the programmer: they see the real evaluation order that GHC is using. Ennals and Peyton Jones previously argued that conventional "stop, examine, continue" debuggers deserve more attention in the context of lazy functional languages (Ennals and Peyton Jones 2003a) — we take their approach a step further (or back!) in that we don't attempt to strictify the evaluation order as they did. It remains to be seen whether this approach is successful in the long term, but initial experience is promising, and we believe the design hits a useful point in the design space. Further comparison and discussion can be found in Sections 5 and 6.

While the paper is primarily a report on the development of a practical debugging tool for Haskell, it also makes the following technical contributions:

- We describe the way in which Haskell code is annotated with source locations and free variables on every subexpression, such that the annotations are maintained accurately throughout the compiler with almost no changes to the intermediate compilation stages (Section 4.2).

- We describe the way in which annotated code is compiled to bytecode such that there is only a 15% overhead in compile-time and 45% overhead in execution (Section 4.4).

- Our debugger includes a novel technique for recovering the types of polymorphic values at runtime, which is essential for debugging polymorphic code (Section 3).

- We describe an API for debugging Haskell that is exposed via the GHC API, and on which our GHCi front-end is built (Section 4.1).

2. Using the debugger

In this section we introduce the basic features of the debugger, and then demonstrate its use on a moderately sized program.

First, let us start with a simple example, the implementation of `Data.List.lines`:

```
lines     :: String -> [String]
lines "" =  []
lines s  =  let (l, s') = break (== '\n') s
            in  l : case s' of
                            []      -> []
                            (_:s'') -> lines s''
```

To debug a program we simply load it into GHCi in the normal way.

---

[2] implicit parameters, GADTs in some corner cases, and phantom types

[3] `http://haskell.org/hide/`

[4] `http://shim.haskellco.de/trac/`

Let's investigate why `lines` behaves differently for leading and trailing newlines:

```
*Main> lines "\na"
["","a"]
*Main> lines "a\n"
["a"]
```

The GHCi debugger allows us to set a *breakpoint* in the program, which is a location in the source code where execution will stop, so that we can inspect the values of local variables. A breakpoint can be set on any expression in the program, or a top-level function:

```
*Main> :break lines
Breakpoint 1 activated at lines.hs:(4,0)-(8,24)
```

Execution stops at the breakpoint when we invoke `lines`:

```
*Main> lines "a\n"
Stopped at lines.hs:(4,0)-(8,24)
_result :: [String]
[lines.hs:(4,0)-(8,24)] *Main>
```

GHCi displays a notice about the breakpoint, and the prompt is changed to reflect the current source location. A special variable called `_result` is bound to the value of the expression at the breakpoint, allowing us to manipulate it at the command line if so desired. The breakpoint is at the outer level of `lines`, so we cannot inspect the arguments until pattern matching has occurred. So we use `:step` to take a single evaluation step:

```
[lines.hs:(4,0)-(8,24)] *Main> :step
Stopped at lines.hs:(6,10)-(8,24)
_result :: [String]
s' :: [Char]
l :: [Char]
[lines.hs:(6,10)-(8,24)] *Main>
```

Execution stops inside the second equation of `lines`, at the outermost expression in the body of the `let`. The debugger provides a `:list` command which displays the source code around the current breakpoint, with the current expression highlighted. We demonstrate its use in the extended debugging example in the next part of this section.

We can now inspect the values of the local variables `s'` and `l`, that were bound by the `let` expression:

```
[lines.hs:(6,9)-(8,23)] *Main> (l,s')
("a","\n")
```

revealing that the line has been split as expected. If we single-step a couple more times, we see that execution proceeds to the second branch of the `case`:

```
[lines.hs:(6,13)-(8,23)] *Main> :step
Stopped at lines.hs:8:15-23
_result :: [String]
s'' :: [Char]
```

and we can display the value of `s''`:

```
[lines.hs:8:15-23] *Main> s''
""
```

Clearly the recursive call will now enter the base case of `lines`, returning the empty list. This explains why `lines` drops a trailing newline from the input.

2.1 A larger debugging example

We will now explore more of the functionality provided by the GHCi debugger by tackling one of the programs from the buggy version of the NoFib benchmark suite (Silva 2007a).[5] The program we have chosen is called `infer`, by Wadler, which implements Hindley-Milner style type inference, and is part of the "real" subset of test cases. Due to an (intentional) error introduced into the program, it crashes with a stack overflow when attempting to type check this term: `let id = \x.x in id id`. The expected output is the type `a -> a`, or something equivalent. We suspect that the program is diverging because it ought to produce an answer immediately, but instead it computes for a couple of seconds before raising the error.

The first thing to do is load the program into GHCi:

```
Prelude> :load Main.hs
```

The ideal way to diagnose this kind of bug is to run the program until it crashes, and then trace backwards through the call stack. Unfortunately, lazy evaluation means that a "lexical" stack is not available. We can work around this limitation using the debugger's tracing facility, combined with its ability to stop at a breakpoint when an exception is raised. All we have to do is raise an appropriate exception when we think the program has entered its divergent computation (this feature is not enabled by default, so we enable it by setting the flag `-fbreak-on-exception`).

Now we run the program in trace mode:

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

Once it has been running for a few seconds, we want to interrupt it and examine its trace. Due to the nature of our program, if it is diverging, it will do so almost immediately after it has started. So we can simply wait a couple of seconds and then interrupt it by typing `control-C`. This halts the program execution, and the debugger stops at a breakpoint:

```
Stopped at <exception thrown>
_exception :: e
[<exception thrown>] *Main>
```

Now we can inspect the trace. The `:history` command prints the top twenty logged locations by default. The first three locations are shown below:

```
-1  : MaybeM.hs:(13,0)-(14,32)
-2  : StateX.hs:7:38-82
-3  : StateX.hs:4:26
...
```

Each location is numbered from −1 downwards. The idea being that the current breakpoint location is 0, and −1 is the previous one, and so forth. We can explore the history by taking a step backwards with `:back`. This takes us to the program location which was visited just prior to the exception:

```
Logged breakpoint at MaybeM.hs:(13,0)-(14,32)
_result :: Maybe y
[-1: MaybeM.hs:(13,0)-(14,32)] *Main>
```

The location corresponds to the expression spanned by the coordinates (13,0)-(14,32) in the file `MaybeM.hs`. We can ask the debugger to list this section of the code:

```
:list
12  thenM ::  Maybe x -> (x -> Maybe y) -> Maybe y
13  (Just x) 'thenM' kM =   kM x
14  Nothing 'thenM' kM  =   Nothing
15  failM ::  Maybe x
```

The listing shows the line before and after the section to provide some helpful context.

Traversing backwards through the trace we eventually arrive at this breakpoint:

```
Logged breakpoint at InferMonad.hs:(82,25)-(86,61)
_result :: Infer [MonoType]
n :: Int
```

Listing the code reveals that it corresponds to `freshesI`, which is supposed to produce a list of `n` fresh type variables. We can print out the value of `n` like so:

```
:print n
n = 3523
```

This is a surprisingly big number — we do not expect that our example needs this many variables.

Now we have an important clue as to the cause of our bug, but why did `n` get so big? We search through the trace a bit more, but it is full of state monad functions which don't reveal any more clues.

We are suspicious of the large `n` from above, so we decide to start again, this time with a focus on on `freshesI`. First, we must set the breakpoint on the suspicious function:

```
:break freshesI
Breakpoint 0 activated at
   InferMonad.hs:(81,0)-(86,61)
```

We no longer want to keep the trace history of the program, so we run it as normal, by calling `main` at the prompt. The program executes as usual and we hit a breakpoint:

```
Stopped at InferMonad.hs:(81,0)-(86,61)
_result :: Infer [MonoType]
```

From here we want a more fine-grained view of the program's behaviour, so we decide to take a single step of reduction:

```
:step
Stopped at Infer.hs:17:43-52
_result :: Int
xxs :: [TVarId]
```

With a little bit more stepping we find ourselves wading through a sea of state monad code, which only leads to frustration. Fortunately there is a way to eliminate some of this noise. We can compile the state monad module to object code rather than byte code. GHCi can happily execute programs which are a mixture of object code and byte code, but it does not record breakpoints for compiled code. Re-compiling the state monad module to object code effectively disables all potential breakpoints in that module, so single stepping does not get bogged down in uninteresting parts of the program. We re-load the program and begin single stepping after the breakpoint on `freshesI`, and we soon notice that with each invocation the value of `n` increases, when it ought to decrease and eventually reach zero.

Careful inspection of the code for `freshesI` shows that `n` is erroneously incremented on each recursive call, when it ought to have been decremented. Hence, the function never reaches the base case of zero, and it diverges. Here is the offending piece of code:

```
freshesI 0 = returnI []
freshesI n = freshI 'thenI' (\x -> freshesI (n+1)
                    'thenI' (\y -> returnI (x:y)))
```

3. Runtime Value Inspection

In this section we describe how the GHCi debugger provides runtime value-inspection. When a program hits a breakpoint, there may be local variables in scope at the breakpoint site, and we must provide a way to inspect the values of those variables.

GHCi itself has no built-in way to inspect values; when you type an expression at the prompt, the result is displayed by the `show` method of the `Show` class. Therefore an expression's type must be an instance of `Show`, if it is to be displayed. This doesn't work in general for displaying the values of local variables at a breakpoint because we may have insufficient type information available. In a polymorphic function, local variables can have types that involve type variables, so we don't know which instance of `Show` is the right one — an instance may not even exist for some types.

At runtime all values except for functions and exceptions are monomorphic. GHC uses *type erasure*, so there is no explicit type information at runtime, and we have no direct access to the instantiations of the type variables (in contrast to languages which do have full runtime type information, such as Mercury (Dowd et al. 1999) and the .NET framework).

For example, consider the usual definition of `map`:

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

when stopped in the right-hand side, we may see something like this:

```
Stopped at map.hs:5:15-28
_result :: [b]
x :: a
f :: a -> b
xs :: [a]
```

all the local variables have the polymorphic types that were inferred at compile time.

These type variables are special: they are not implicitly universally quantified, as is the convention in normal Haskell types. They are regarded by GHC as unknown runtime types, and GHC's type checker will refuse to unify an unknown runtime type with anything except itself. For example, even if we happen to know that `x`'s type really is `Char`, we cannot use that information:

```
> Data.Char.ord x
<interactive>:1:14:
   Couldn't match expected type 'Char'
       against inferred type 'a' (a rigid variable)
     'a' is an unknown runtime type
```

We therefore cannot inspect the value of `x` by evaluating an expression, since GHC has no information about the type of `x`. For this reason the GHCi debugger includes generic value-inspection support, in the form of the `:print` command, that performs four main functions:

1. it allows the inspection of partially-evaluated values without forcing evaluation,

2. it prints the values of variables that have partially- or completely-unknown types,

3. it reconstructs as much of the real type as possible, based on the runtime value (this is called "runtime type inference"),

4. it propagates any knowledge obtained about the type to other variable bindings in the environment.

It is essential that we provide a way to view values without forcing any evaluation, since forcing evaluation prematurely may affect the future execution of the program. It must be possible to observe the execution of the program without modifying it. Moreover, subterms of the runtime value may be non-terminating or contain exceptions. We note that our `:print` command is similar to the *dirt* "Display Intermediate Reduced Term" primitive, described by Naish and Barbour (Naish and Barbour 1996); `:print` extends *dirt* with the capability of recovering the type information.

3.1 Approaches to runtime type reconstruction

In the strict, functional setting, we distinguish two main approaches to runtime type reconstruction.

The first approach is based on a backwards traversal of the call stack. All polymorphically typed values are created in calls to polymorphic functions, but at the call site the arguments are monomorphic. The ML debugger (Tolmach and Appel 1995) uses the information in the call stack to re-construct the calling context and from that recover type information. Consider this example:

```
c = map ord ['a','b','c']
```

Inside the definition of `map` we know nothing about the type of `x`. However, the call to `map` inside the body of `c` implies that the type of `x` is a character. We can generalize this technique for all cases. However, the debugger must be able to explore the function call stack, and have access to the binding-time call chain of every variable in scope.

The second approach is based on decoding the types by inspecting heap representations. This can be done by adding debugging information to the compiled code, such as explicit type tags. This approach is more low level and hence less portable than the first, although it is popular in practice, for example the Mercury debugger (Dowd et al. 1999) uses an instrumentation–based variation.

We employ the latter, since the lazy call stack does not always contain sufficient information about the calling context to reliably reconstruct types.

3.2 Run-time Type Inference

Runtime type inference occurs at any point, launched on demand upon a given runtime term t. Conceptually it takes place in two stages.

First, a type τ is inferred for t; this is reminiscent of standard type inference, except that the term is a simple structure containing only constructor applications:

$$\text{Terms} \quad t \quad ::= \quad \square \qquad \text{function, or non-value}$$
$$| \quad c_n \; \bar{t} \quad \text{constructor application}$$

Unevaluated expressions and functions in the runtime term are represented by $\square$.

Second, τ is unified with τ', the type of t that was inferred at compile-time. Note that τ' may involve type variables.

The substitution that results from unifying τ with τ' can be applied to τ' to give a refined type for t, and also to the types of other runtime terms in the environment. Any type variables remaining after applying the substitution correspond to unknown runtime types.

There is a key difference between runtime type inference and compile-time type inference that deserves mention. In standard Hindley-Milner type inference (Damas and Milner 1982), it is always safe to infer a type more specific than the principal type; the program will not go wrong (it may fail to typecheck, however). In runtime type inference, it is not in general safe to infer a type more specific than the most general type[6]. The type variables in a runtime type should be thought of as existentially quantified, not universally quantified: they represent a particular type, but we don't know what it is yet.

As described above, our debugger *combines* the results of compile-time type inference and runtime type inference to obtain as much type information as possible without compromising safety.

---

[6] We are working on a formalisation of this type system, but it wasn't ready in time for this paper.
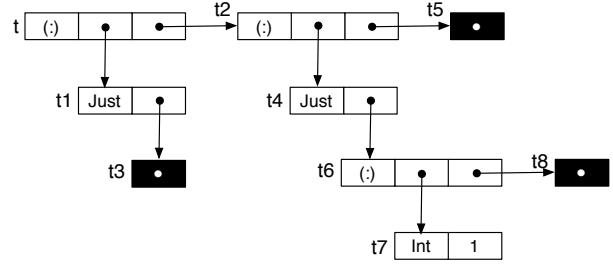

Figure 1. Heap representation of an intermediate value

A significant complication is that, at runtime, there is no way to infer types constructed using `newtype`. For example, given the declaration

```
newtype T = T Int
```

a value of type `T Int` is indistinguishable at runtime from a value of type `Int`. We cover our solution to this problem in Section 3.4.

Termination is also problematic with cyclic terms. We address this issue in Section 3.5.

Our algorithm produces not just a refined type for the term, but also a source term corresponding to the runtime term that can be displayed to the user. The source term may contain holes representing unevaluated expressions; in the user interface these holes are assigned fresh variable names so that they may be referred to in subsequent expressions. For the sake of simplicity we ignore these minor issues in the forthcoming discussion.

3.3 An example

Consider the following partially-evaluated runtime term:

$$\texttt{t} = \text{Just} \; \square : (\text{Just} \; (1 : \square) : \square)$$

Figure 1 shows how `t` is represented in the heap, where black boxes denote unevaluated expressions. Suppose that the static type environment tells us that `t` has partial type `[a]`.

Runtime type-inference proceeds as follows. For each closure we generate a constraint, where by convention the right hand side contains the data constructor type, and the left side is built from the types of the subterms in the heap.

The signatures of the constructors involved are:

```
(:)     :: a -> [a] -> [a]
[]      :: [a]
Just    :: a -> Maybe a
```

The set of constraints for Figure 1 is generated by walking all the subterms, obtaining:

$$
\begin{aligned}
\texttt{t} &= [\alpha_1] \\
t1 \rightarrow t2 \rightarrow t &= \alpha_2 \rightarrow [\alpha_2] \rightarrow [\alpha_2] \\
t3 \rightarrow t1 &= \alpha_3 \rightarrow \text{Maybe } \alpha_3 \\
t4 \rightarrow t5 \rightarrow t2 &= \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \\
t6 \rightarrow t4 &= \alpha_5 \rightarrow \text{Maybe } \alpha_5 \\
t7 \rightarrow t8 \rightarrow t6 &= \alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \\
t7 &= \text{Int}
\end{aligned}
$$

where the first equation comes from the compile-time type information. Solving the equations via the standard mechanism of (syntactic) unification returns a solution substitution that contains the types for all the closures, including `t :: [Maybe [Int]]`. Now we unify this with the compile-time type `t :: [a]`, obtaining the desired substitution `a ↦ Maybe [Int]`, which can be applied to the runtime environment to refine the types of the things inside it.

3.4 RTTI and newtypes

Newtypes are eliminated, by the compiler, after the standard compile–time type checking pass is completed. This means that there is no trace of newtype constructors in the heap. The only places where newtypes appear are the signatures of data constructors used in the right hand sides of constraints, and compile-time type signatures. Thus, we must consider additional implicit equations when solving the constraints. The declaration of a newtype

```
newtype Set a = Set [a]
```

gives rise to the equation `Set a = [a]`.

Consider a slightly modified version of the example in Section 3.3, but now the compile-time type information for `t` is `t :: Set` α. We proceed as before, walking the heap to collect the typing constraints. The representation of `t` in the heap has not changed.

$$
\begin{aligned}
t &= \text{Set } \alpha_1 \\
t1 \rightarrow t2 \rightarrow t &= \alpha_2 \rightarrow [\alpha_2] \rightarrow [\alpha_2] \\
t3 \rightarrow t1 &= \alpha_3 \rightarrow \text{Maybe } \alpha_3 \\
t4 \rightarrow t5 \rightarrow t2 &= \alpha_4 \rightarrow [\alpha_4] \rightarrow [\alpha_4] \\
t6 \rightarrow t4 &= \alpha_5 \rightarrow \text{Maybe } \alpha_5 \\
t7 \rightarrow t8 \rightarrow t6 &= \alpha_6 \rightarrow [\alpha_6] \rightarrow [\alpha_6] \\
t7 &= \text{Int}
\end{aligned}
$$

But now we cannot launch unification directly, because the two equations for `t` do not agree: type `Set` α_1 does not unify with type $[\alpha_2]$, even though we know that both are isomorphic.

Our problem can be expressed succinctly thus: we want to apply newtype equations where necessary such that the unification of static and runtime types succeeds, and we wish to do so as few times as possible. Our algorithm therefore works as follows: we attempt to unify the constraints generated by inference, and if unification fails, then we attempt to apply newtype equivalences to the terms that did not unify in order to make unification succeed. Doing so is hard in general: the rewrite system implied by newtype equations is not necessarily terminating. Nevertheless, our heuristics work well on common examples. We intend to explore a more formal treatment of this part of the system in future work.

3.5 Termination and efficiency

The number of constraints generated by the RTTI algorithm is proportional to the size, in number of closures, of the term being processed. Given that this number is finite, and from the results on termination of unification in the literature (see (Baader and Snyder 2001) for an extensive report) we conclude that unification over the set of constraints terminates. However, the process of generating the constraints, which is performed earlier or at the same time as the unification, may not terminate when conducted over cyclic data structures. It should be possible to keep a log of the nodes visited [7] to recover termination, however we have not pursued this option for the time being.

We have studied and implemented two refinements that take into account the availability of a completely reconstructed type in order to improve termination and efficiency. We briefly give an account of the general ideas.

The first refinement consists in generating and solving constraints in breadth-first fashion, unification being done step-wise. As soon as a full type is recovered, the process stops and returns it. It might seem odd that we need to act this way, but keep in mind that this is not exactly the same as a type inference problem: here we are usually interested only in the type of the top level term, while most of the constraints generated are necessary only for typ-

ing the subterms. Moreover, this refinement improves termination even on cyclic data structures in most cases. In practice, this probably covers all the reasonable cases. But indeed, if launched upon a cyclic structure with a fully unrolled spine and suspended contents, the breadth-first RTTI would still loop [8].

The second refinement concerns the scenario where we are interested in recovering the type of every subterm. In such case, we must obviously walk the full subterm tree. We do so in depth first fashion, and as soon as a fully monomorphic type has been recovered, we propagate it down the tree replacing unification by the much more efficient matching. Since it is still possible that type variables will appear further down the tree, this needs to be done carefully.

3.6 Practical Concerns

We have described how, given a partially-evaluated runtime term, together with some static type information, we are able to reconstruct a partial type.

There are some tricky issues with actually implementing this scheme, however. How do we inspect the partially-evaluated structure of a runtime term from a Haskell program (GHCi), and obtain the type signatures of the constructors used in the term?

Firstly, our runtime system provides a primitive operation `unpackClosure#` for inspecting a closure. A closure has two parts: the "info pointer" which points to a structure, the "info table", that describes the layout of the closure and the code to evaluate it, and the "payload", containing the fields of the closure. The `unpackClosure#` primitive returns the info pointer and the payload in a safe manner.

If we establish by inspecting its info table, that a closure corresponds to a data constructor, the next step is to retrieve its type signature. To this end, we have extended the info table for a data constructor with a field containing the fully-qualified *name* of the constructor. The name is unique within the program, so this is enough to retrieve all the information about the data constructor from GHC's internal data structures.

This was the only change we had to make to GHC's compilation strategy to enable debugging. The cost in terms of space taken up by the constructor name in the binary is small: there are only a few data constructors compared to the number of functions and thunks, which make up the majority of the code. It's conceivable that in the future we could add information to the info tables for other closures which would enable the debugger to reconstruct more of the type for, say, a function or thunk, but we haven't explored this extension as of yet.

Data constructor fields The fields of a data constructor in the heap do not necessarily match the fields of the data constructor as it appears in the source code, for two reasons:

- Existential quantification can cause extra type-class dictionaries to be stored in the constructor (we call these "existential dictionaries").
- Strict fields may be unpacked. For example, a strict field of type `(a,b)` may be represented by two fields of type `a` and `b` respectively, rather than a single pointer to the pair.

For each data constructor, GHC maintains a record of the types of the fields in the source-code representation of the constructor, and also the types of the fields in the runtime representation. When reconstructing types, we must be careful to use the latter when matching types against field values.

---

[7] by taking the address of a closure as its identity, and keeping the GC from moving things around during the RTTI process

[8] In our current implementation this is handled cheaply with a depth limit

4. Implementation

The debugger is crafted to be as light on complexity as possible; GHC is already a complex system, and we are resistant to introducing any new distributed invariants into the system.

We realised early on that the debugger must be integrated with the GHCi environment, for the following reasons:

- When inspecting values in the heap, the relationship between the source code representation of a data constructor and the runtime representation in the heap may be complex, as we described in Section 3.6. Understanding the runtime representation requires knowledge of how GHC derives the representation, so the best way to understand it is to ask GHC itself.

- Purely from a usability perspective, having full interactive Haskell evaluation available while debugging is highly desirable, as we argued in Section 1.

We also had a goal in mind: accessibility. The debugger should work with everything and be always available, even if this means sacrificing functionality. In particular, this implies that any solution that requires recompiling the entire program and libraries is ruled out. GHC already forces libraries to be recompiled for profiling, and requiring yet another set of compiled libraries would be a painful burden. We considered using the profiling libraries for debugging: after all, our profiling system already includes cost-centre stacks, which give a form of lexical call stack which would be highly useful for debugging. However, GHCi only works with the non-profiled libraries, and modifying the bytecode compiler and interpreter to work with the profiling version of the runtime system would require a lot of work, so we didn't take this path. We may consider this as an alternative in the future, however.

The following sections describe the various parts of our implementation.

4.1 The debugger API

Our debugger implementation is completely independent of its user interface: debugging facilities are exposed by the GHC API, which provides a programmatic interface to GHC's compilation and interactive evaluation capabilities. GHCi consists of a text-based user interface built on top of the GHC API.

This is a useful factorisation, because it means that the GHC API can be reused in other contexts that need access to Haskell compilation or evaluation services. For example, interactive development environments can talk directly to the GHC API to obtain information about the source code on the fly; Visual Haskell (Angelov and Marlow 2005) does just that.

With our debugging services exposed via the GHC API, interactive development environments based on the GHC API can now provide the user with a more visual user interface to the debugger.

To give a flavour of the API for debugging, we have provided a cut-down version of it in Figure 2.

To execute a new statement interactively, the client of the GHC API invokes `runStmt` (this is what GHCi does when you type an expression or statement at the prompt). The `runStmt` call may return `RunBreak`, which indicates that execution stopped at a breakpoint. At this point, the client can use `getResumeContext` to find out where the breakpoint was (this is used by GHCi's `:show context` command). In general there may be a stack of breakpoints active, because after stopping at a breakpoint we may invoke another statement that itself stops at a breakpoint, which is why `getResumeContext` returns a list of `Resume`.

For each `Resume`, we can ask for information about the breakpoint: `resumeBreakInfo` extracts a value of type `BreakInfo`, which itself contains information such as the module and source-

```
runStmt :: Session -> String -> IO RunResult
resume  :: Session             -> IO RunResult

data RunResult
  = RunOk
  | RunFailed
  | RunException Exception
  | RunBreak BreakInfo

getResumeContext :: Session -> IO [Resume]

data Resume
resumeStmt      :: Resume -> String
resumeBreakInfo :: Resume -> BreakInfo

abandon :: Session -> IO ()
```

Figure 2. The GHC API for debugging

code location of the breakpoint (the details of the interface have been omitted for brevity).

Having stopped at a breakpoint, the client can choose to resume execution using `resume`, or abandon the current execution with `abandon`.

The full API contains support for listing all the breakpoints within a given module, enabling/disabling breakpoints, single-stepping (GHCi's `:step` command), tracing (`:trace`), and history (`:history`, `:back` and `:forward`).

4.2 Annotating the source code: ticks

A problem that faces every debugger is how to relate the compiled code back to the source code. This seems to require maintaining a relationship between the compiled code and the original source code at all stages during compilation. In GHC this would be futile: to start with, the translation from Haskell syntax into the simpler Core language already transforms the source code in complex ways, and then there are a host of transformation phases before finally generating byte code or machine code. The final program often bears very little resemblance to the original source code.

Fortunately the Haskell Program Coverage tool (Gill and Runciman 2007) had already solved this problem in an elegant and robust way. The key observation is that for establishing coverage, it is not necessary to reverse-engineer the location of the source code from which every expression in the resulting compiled code originated; and indeed if we tried to do so the results would undoubtedly be fragile.

Determining coverage requires knowing, for each nominated site in the original source program, whether the expression was entered at runtime or not. That is, if you can implement each expression E such that it has a side effect when entered at runtime, then that side effect can be used to update the table of coverage information for the current run. The coverage annotation pass works as follows: each expression in the Haskell source code E for which we need to collect coverage information is replaced by $(tick_{<module,n>} E)$, where n is a unique number for each particular site in any given Haskell module, and $module$ is the name of the source module. $tick_{<module,n>}$ is simply an annotation at this stage. The original source location for each tick is simple to find; for each module there is a list mapping tick numbers to source spans, recorded during the placement of the tick annotations to the Haskell source tree.

Then, during "desugaring" (the pass that transforms Haskell source into the intermediate Core language of GHC), the following translation happens:

$$D[\![tick_{<module,n>}\ E]\!] \implies \textbf{case}\ tick_{<module,n>}\ \textbf{of}$$
$$\textbf{DEFAULT} \rightarrow D[\![E]\!]$$

where $D[\![]\!]$ is the desugaring transformation, **case** in Core always implies evaluation, and **DEFAULT** is a case alternative that binds nothing. An identifier $tick_{<module,n>}$ is generated fresh for each site, and it is annotated with both the current module name and a unique number within the current module.

Now, $tick_{<module,n>}$ is a special identifier: it is regarded by GHC as having side-effects when evaluated, in much the same way as many primitive operations in GHC's intermediate language have side-effects.

GHC's optimiser is very careful with expressions whose evaluation may have side-effects. Side-effects normally occur only in the context of the IO monad, but it is still important to optimise IO code, and to do so without altering the intended ordering of the side-effects. So when the optimiser knows an expression has side-effects, it will cause the expression to be evaluated if and only if a normal-order evaluation would evaluate it. The expression will not be speculated (evaluated early); neither will it be eliminated if its result is never used. This is precisely the behaviour we need for analysing coverage: we want the results of coverage analysis to be deterministic, it's no good if the coverage results change when optimisation is turned on.

Note that GHC is not prevented from optimising code around a tick: all of GHC's optimisations are still valid, provided they respect side-effects, which they already do. Since each tick identifier contains the original module name, cross-module inlinings can be performed without concerns that the module-unique numbering system might have collisions. No changes to GHC's optimisations were required to get accurate coverage information, and the coverage results are consistent even when full optimisation is enabled.

4.3 Using ticks for breakpoint sites

Breakpoints have a lot in common with coverage ticks. We don't want the optimiser to speculate or eliminate our breakpoint sites; we want the breakpoint to be triggered if and only if the expression is really demanded. So our debugger implementation re-uses the tick mechanism to annotate the program with breakpoint sites. There are two main differences between the tick-annotation pass when used for breakpoints and when used for coverage:

- The choice of sites on which to place the ticks is somewhat different between coverage and breakpoints.

- Breakpoint sites are annotated with a list of free variables, whereas this information is not required for coverage.

A breakpoint site is added to every non-trivial subexpression, and to the body of every function equation, lambda expression, case alternative and let-expression. These sites were chosen as a reasonable compromise between utility and speed; we could certainly add breakpoint sites to every individual variable occurrence, for example, but to do so would impose a significant penalty at runtime and possibly have an adverse effect on the usability of single-stepping.

A breakpoint site in Core looks like this:

$$\textbf{case}\ tick_{<...>}\ x_1 \ldots x_n\ \textbf{of}$$
$$\textbf{DEFAULT} \rightarrow E$$

where $x_1 \ldots x_n$ are the free variables of the original source expression. Note that Core is a typed language, so in reality we have to make this expression type-correct by giving $tick_{...}$ a polymorphic type and passing an appropriate type argument; the details are not important.

Why do we need to annotate breakpoint sites with a list of free variables? Couldn't we just figure out the free variables later on

when compiling code for the breakpoint site? The primary reason is that the set of free variables of a source expression may very well be different from the set of free variables of the corresponding Core expression after it has undergone transformation and optimisation. GHC is free to invent new variable bindings, discard bindings by inlining or elimination of dead code, and rename existing variables. By adding variables to our tick annotations, we ensure that the free variables of the original source expression are not inadvertently optimised away, and we can keep track of their names in the final program. We must separately keep track of the names of the original source variables and the order in which they were added to the tick. This information is collected during the tick-annotation phase and carried through to the bytecode compiler.

Why do we only attach *free* variables, instead of collecting all the source variables that are in scope? We originally did the latter, but it has some undesirable side effects:

- Many more variables are attached to each breakpoint location, and this affects compilation performance (a large example we tried went from 30% slower to 50% slower compilation with in-scope variables rather than free variables).

- It prevents many optimisations that would otherwise be performed on the intermediate Core. For example, in a set of bindings in a where clause, the right-hand side of each binding would have a breakpoint site that refers to all the bindings (including itself), because all of these would be in scope. This prevents the binding group from being broken up and inlined, because the extra dependencies introduced by the breakpoint sites have forced it to be a single strongly-connected component.

Annotating breakpoint sites with just the *free* variables imposes no extra dependencies on the intermediate code than were already there, so transformations should be unaffected.

4.4 Compiling ticks to bytecode

The bytecode compiler takes a Haskell module and compiles it into a set of bytecode objects (BCOs). The input to the bytecode compiler is a set of bindings in A-normal form; that is, the right-hand side of an application is always a variable. The idea is that every closure in the program is represented by an explicit let-binding; this makes code generation easier.

Each BCO represents either the right-hand side of a **let**-binding, or the continuation of a case expression. For example, when generating bytecode for the Core expression

$$\textbf{let}\ x = E_1\ \textbf{in}\ E_2$$

a new BCO would be created to contain the code for E_1, and the current BCO would contain code to construct the closure for x, followed by the code for E_2. In the Core expression

$$\textbf{case}\ E\ \textbf{of}\ \{\ alts\ \}$$

a new BCO is created to contain the code for $alts$, and the current BCO would contain code to push $alts$ onto the stack (along with any free variables it requires), followed by the code for E.

For the debugger, we extended the compilation scheme slightly. When compiling the expression

$$\textbf{let}\ f = \lambda\,x_1 \ldots x_n \rightarrow \textbf{case}\ tick_{<module,n>}\ y_1 \ldots y_m\ \textbf{of}$$
$$\textbf{DEFAULT} \rightarrow E_1$$
$$\textbf{in}\ E_2$$

we create a new BCO as usual for the right-hand side of f, and additionally we annotate it with the breakpoint site $<module,n>$. The breakpoint site requires no extra bytecode, it is simply an an-

notation on the BCO object[9]. This scheme applies to non-function bindings too (when n is zero).

What about the breakpoint sites that don't occur as a **let**-binding? Recall that the program is in A-normal form, so the remaining places where breakpoint sites can occur are:

- A case alternative

$$\textbf{case } E \textbf{ of } \{ p \rightarrow \textbf{case } tick_{<...>} \textbf{ of } \{ \ldots \}\}$$

- The scrutinee of a case expression

$$\textbf{case } (\textbf{case } tick_{<...>} \textbf{ of } \{ \ldots \}) \textbf{ of } \{ \ldots \}$$

- The body of a `let` expression

$$\textbf{let } x = E \textbf{ in case } tick_{<...>} \textbf{ of } \{ \ldots \}$$

The *tick* expression in each of the above examples occurs somewhere in the middle of a bytecode sequence, rather than at the beginning. This presents a problem, because we can only break at a "safe point", where the stack is in a state that can be traversed by the garbage collector. Safe points in the interpreter occur at the beginning of a BCO only.

This means the above three breakpoint sites cannot be implemented directly, because they do not occur at safe points. We must manufacture extra safe points so that it is possible to break at these points. Fortunately this is relatively straightforward: all we need to do is transform the intermediate code so that the breakpoint site occurs as the right-hand side of a **let**-binding. For each breakpoint site that does not already occur as the right-hand side of a **let**-binding, we transform it thus:

$$\begin{array}{ccc} \textbf{case } tick_{<...>} \textbf{ of} & & \textbf{let } x = \textbf{case } tick_{<...>} \textbf{ of} \\ \{\ldots\} & \Longrightarrow & \{\ldots\} \\ & & \textbf{in } x \end{array}$$

This has the effect of forcing the code generator to insert an extra safe point, by building a new closure for the expression to be evaluated and then immediately evaluating it.

Unfortunately this also introduces some overhead, due to the extra closures being allocated. No doubt there is a more elaborate solution that could minimize the overhead introduced, perhaps by only creating the extra closure if the breakpoint is enabled. However, the current scheme is simple, and the overhead is manageable (see Section 4.7), so we don't consider this a serious problem.

4.5 Implementing breakpoints in the interpreter

When the running evaluation encounters a breakpoint site, the following must happen:

- The interpreter should check whether this breakpoint site has been enabled via the `:break` command, or we are in single-step mode. If not, execution continues as normal.

- Otherwise, control is passed back to GHCi, leaving a way in which the computation may be continued later, and providing access to the values of the free variables at the breakpoint site.

Recall that in our debugging API (Figure 2), the `runStmt` function begins the execution of a new statement. The statement either fails to compile (`RunFailed`), runs to completion (`RunOk`), raises an exception (`RunException`), or encounters a breakpoint (`RunBreak`). The function `resume` continues the most recent computation that stopped at a breakpoint.

The code for `runStmt` (somewhat simplified from the real implementation to convey the ideas more clearly) is given below:

---

[9] The current implementation represents the annotation as an instruction in the bytecode stream, but we plan to change this.

```
runStmt stmt = do
  status_mvar <- newEmptyMVar
  break_mvar  <- newEmptyMVar
  let  on_break info = do
         putMVar status_mvar (Break info)
         takeMVar break_mvar
  forkIO $ withBreakAction on_break $ do
            result <- try stmt
            putMVar status_mvar (Complete result)
  result <- takeMVar status_mvar
  case result of
    Complete (Left ex) -> return (RunException ex)
    Complete (Right r) -> return RunOk
    Break info -> do
      setResume session (break_mvar, status_mvar)
      return (RunBreak info)
```

Breakpoints are implemented using threads and `MVars`. The computation is run in a new thread, and it places its result in `status_mvar`. The `withBreakAction` function tells the interpreter that the action `on_break` should be invoked in the event of a breakpoint being hit.

The main thread waits for a result in `status_mvar`. If a breakpoint is encountered, the interpreter runs `on_break`, which tells the main thread via `status_mvar` that a breakpoint was hit, and then waits on `break_mvar`. The main thread sees `Break` as the return status, it records the information needed to resume the current computation in the `Session`, and returns `RunBreak` to the caller.

This scheme has the advantage that it requires very little special runtime support. Stopping at a breakpoint is handled using the existing blocking/resuming capabilities provided by `MVars`, so the runtime scheduler needs no modifications. All that is needed in the runtime is support in the bytecode interpreter to detect a breakpoint being hit (by examining the annotation on the BCO) and to run the `on_break` action.

4.6 Debugging exceptions

As we demonstrated in Section 2.1, the debugger can stop execution when an exception is thrown, even if the exception is thrown by compiled code (as it would be if the program evaluated `head []`, for example). This turned out to be almost trivial to implement: the runtime has a primitive for throwing exceptions (`raise#`), and in its implementation we simply invoke the breakpoint handler, if one is set. The breakpoint handler is the `IO` action given to `withBreakAction`, as before (Section 4.5). The effect is exactly as if the thread stopped at a breakpoint, except that there is no breakpoint source location. It is possible to inspect the history, as we did in the example.

One difficulty arises if we want to stop when the user types Control-C to interrupt the execution, as we did in the example. Control-C generates an *asynchronous* exception, which doesn't go via the `raise#` primitive. To work around this, we catch the Control-C exception in GHC and re-throw it as a synchronous exception, at which point the breakpoint handler will be invoked as normal.

4.7 Performance

We have used the regular version of the NoFib (Partain 1993) benchmark suite for measuring the overhead introduced by our modifications. The suite consists of around 90 programs ranging from small to fairly large in terms of lines of code (from 10 to 5800, excluding comments), and execution times in GHCi ranging from a few milliseconds to more than a minute. The programs are distributed in three categories consisting of synthetic benchmarks, real life programs, and those considered to be "somewhere in between".
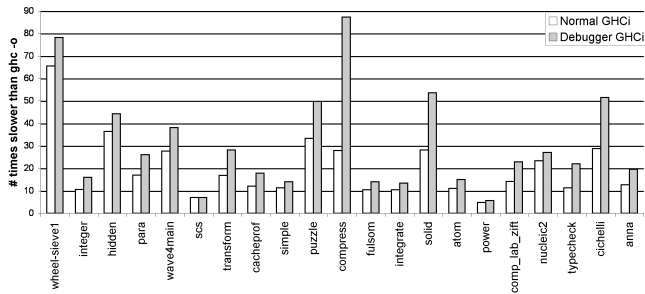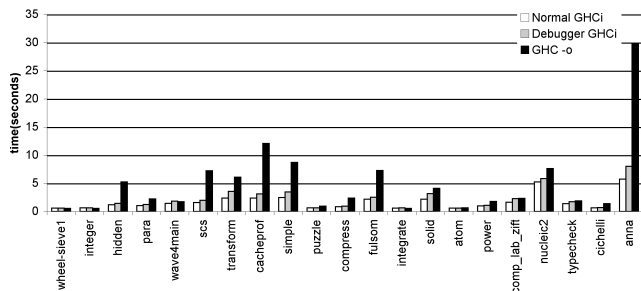
Figure 3. NoFib running times

Figure 4. NoFib compile-to-bytecode times

Examples include a strictness analyser for an abstract language, a brute-force perfect hash function, a type checker, a propositional formulae clausifier, a Haskell parser, and many others.

The test consisted of loading and running every program in the suite. Figure 3 compares the running times of the 20 longer running programs, between the object code compiled version, the evaluation in plain GHCi, and in GHCi extended with our debugger. Figure 4 compares the compile to bytecode times, for the same set of programs.

Over the full suite of programs, the debugging version of GHCi takes around 15% extra time on average to produce the bytecode for a program, while the evaluation time is increased around 45% on average. In the worst case we have seen, program `compress`, the increase in running time is of 200%.

5. Related Work

It has long been known that conventional debugging technology is difficult to apply to purely functional languages (Hall and O'Donnell 1985), especially so for lazy languages. In this section we consider the most recent advances which have been made, with a particular emphasis on those projects which have resulted in tools.

5.1 Diagnostic writes

Most Haskell implementations come with a debugging primitive `trace :: String -> a -> a`, for wrapping diagnostic writes around expressions. There are a number of reasons why `trace` is a poor substitute for a real debugger:

- One call to `trace` can invoke further calls to `trace`, resulting in an incomprehensible mixture of debugging messages.

- One is obliged to manually convert a value to a string, but this can change the semantics of the program being debugged.

- `trace` is limited to types which can be converted to strings, which prohibits the printing of functions and abstract data types.

The limitations of `trace` are addressed by the Haskell Object Observation Debugger (Hood). Hood is implemented as a Haskell library which provides a diagnostic writing facility called `observe`:

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

The advantages of `observe` over `trace` are as follows:

- Observations are demand driven, which means `observe` accurately records the extent to which a value was computed, without changing the semantics of the program.

- Observations are associated with a particular expression, and are tagged by a string. Thus, a program execution with multiple observed expressions produces a comprehensible output.

- `observe` is overloaded, which enables customised printing methods on a per-type basis.

- Functional values can be observed, and are printed in an extensional style (providing their domains and co-domains are observable).

Diagnostic writes are a cheap way to probe the behaviour of programs, but their effectiveness is limited in a couple of ways:

- Programs must be modified by hand.

- They encourage a style of debugging based on trial-and-error.

5.2 Breakpoint debuggers

The conventional wisdom is that breakpoint debuggers are ill-suited to non-strict functional languages because the order of computation steps *under lazy evaluation* is difficult for the user to follow. An animated dynamic trace of execution of Haskell programs shows the unintuitive behaviour of entering a function then later returning to the call site to evaluate arguments. Especially in recursive functions, this jumping back can be plain confusing to follow.

Ennals and Peyton Jones (2003a) have shown that step-based debugging is possible in a non-strict language if *optimistic evaluation* is employed instead of lazy evaluation. Optimistic evaluation causes function applications to be evaluated eagerly, whilst preserving non-strict semantics (Ennals and Peyton Jones 2003b). On occasion a branch of execution might be suspended if runtime profiling determines that it is too costly. Suspended computations can be resumed at a later stage if more of their value is needed.

Optimistic evaluation provides two main advantages for debugging:

1. The stacking of function calls usually resembles the nesting of applications in the source code. This makes it easier to see how calls are related to the program structure.

2. Argument values are (mostly) evaluated before the body of the function is entered, making them easier to display and comprehend.

In practice it would require a significant investment of effort to implement optimistic evaluation in the main branch of GHC. For this reason HsDebug has not progressed past the prototype stage.

Unsurprisingly strict functional languages have enjoyed breakpoint debuggers for many years. Notable examples are: the ML debugger of (Tolmach and Appel 1995) which features reverse program execution by periodically checkpointing program states — a feature which has been adopted in the OCaml debugger (Leroy et al. 2007); and the procedural debugger of Mercury (Somogyi and Henderson 1999), which is tightly integrated with the Mercury declarative debugger (MacLarty 2005).

5.3 Program tracers and Declarative Debugging

Declarative debugging (also called Algorithmic Debugging (Shapiro 1983)) is a technique for diagnosing logical errors in languages with a "declarative semantics". The evaluation steps of a computation are structured in a dependency tree, which is searched for nodes which correspond to program errors. An advantage of this approach is that the search can be highly automated. The main problem with declarative debuggers is that the computation tree must be generated before debugging can commence. Either the whole tree is saved in advance, which requires enormous amounts of memory for long program runs, or parts of the tree are generated on demand by re-executing parts of the computation again.

There has been considerable interest in building declarative debuggers for Haskell. The earliest example is Freja (Nilsson 2001), which uses an instrumented runtime environment to build a computation tree as a side-effect of program execution. Freja employs a piecemeal approach to building the computation tree, which means that only part of the tree is materialised at any one time (Nilsson 1999). Missing parts can be re-generated by running the program again from the beginning, at the cost of extra computation time. Perhaps the biggest limitation of Freja is that it only supports a subset of Haskell (everything except type classes and IO), and it only runs on a (now) outdated architecture. Buddha (Pope 2006) is a more recent declarative debugger, which is quite similar to Freja from the perspective of the user, but it is implemented by means of program transformation, in order to reduce the implementation cost by taking advantage of existing compiler technology. The downside of this approach is that the space-saving techniques of Freja are much more difficult to achieve, since the debugger is implemented at arm's length from the runtime system. Buddha supports the printing of functional values in two ways: using a term-based representation (like Freja does), and also using an extensional representation (like Hood does).

Declarative debuggers can be thought of as a special instance of the more general field of program tracing: the computation tree is just a particular view of a more detailed history of a program's execution (its trace). The tracing of lazy functional languages has received a reasonable amount of attention in the literature (Watson 1997; Gibbons and Wansbrough 1996; Goldson 1994), but none of these systems have progressed to usable tools, except for Hat (Chitil et al. 2002), which is currently the most substantial debugging tool available for Haskell. Like Buddha, Hat is based on a program transformation. The execution of transformed program behaves like its normal incarnation, but also records a detailed account of its reduction history (called a *redex trail*) to a log file. The redex trail can be used as the basis for a large variety of debugging tools, each providing a different — but complementary — way of reasoning about the behaviour of the program (Brehm 2001; Wallace et al. 2001). The main limitations of Hat are discussed in Section 1.

5.4 Run Time Type Information

We have already mentioned the RTTI features of the logic/functional language Mercury, described in (Dowd et al. 1999). In their solution, the compiler instruments polymorphic functions with additional arguments, one per type variable in the signature of the callee. These then receive a representation of the concrete type in a given invocation, filled in by the caller. The compiler also massages the code so that the type representations are propagated in the obvious way. One can see that this is similar to the dictionary–passing solution used for type classes by Haskell compilers. A priori, one could thus emulate their solution using instrumentation to require that every type variable in a function is an instance of some type class providing representation types, e.g. Data (Lämmel and Peyton Jones 2003) r Rep (Weirich 2006). The mercury solution in addition employs several tricks to reduce the overhead, e.g. reusing info

tables for representations. In comparison, our approach is more low level, since we inspect heap structures to recover type information. The advantage of avoiding program transformations is that there is no price to pay in efficiency if you don't use RTTI.

6. Conclusion and future work

In a sense, our debugger ignores conventional wisdom concerning how to build debuggers for lazy functional languages. Rather than trying to hide laziness and present a purely declarative view of the program, we let the programmer see the effects of laziness, we show the real order of evaluation that GHC uses, and we show the user to what extent their data structures are unevaluated.

This approach has both advantages and disadvantages: one advantage is that it is simpler to do it this way; we don't have to modify the execution model. Another advantage is that it helps the user to understand how laziness works: debugging laziness is sometimes necessary, for example when using techniques like lazy I/O or cyclic programming. A disadvantage is that, unless you really do want to see the effects of laziness, having the execution jump around all over the program can be distracting and confusing. Furthermore, the evaluation order that the programmer sees is determined to an extent by the whim of GHC, and it might be different tomorrow. These are certainly valid arguments, but in our opinion should not prevent us from building a tool that shows the programmer what is happening in their program at runtime — something is better than nothing, and sometimes worse is better.

In our limited experience with the debugger so far, we haven't found the exposure of laziness to be a significant problem. When stopped at a breakpoint, often it is just the values of the free variables that are important, rather than the lexical call stack. It would be nice to have access to a lexical call stack, but giving the user a history of evaluation steps is often enough to establish the context of an error.

There are plenty of ways we would like to extend this work. However we don't yet have the benefit of any significant user feedback to tell us which missing features are the most important. After the debugger is released with the next version of GHC we will collect user feedback and use it to decide where to go next.

We expect that most users will want some way to inspect the lexical call stack. We have done some modest investigations in this direction, but we haven't as yet found a solution that was both lightweight enough and gave predictable enough results to include in the debugger.

It should be possible to allow breakpoints in compiled code. The benefit from doing this would be mainly to reduce the factor of 10-15 overhead from compiling a program to bytecode compared to optimised machine-code compilation.

The current debugger includes no facilities for debugging concurrent programs; we expect this to be a high priority improvement for the future. Also a graphical interface on top of the underlying debugging commands would be useful, especially for visualising the single-stepping facility.

Acknowledgments

References

K. Angelov and S. Marlow. Visual Haskell: A full-featured Haskell development environment. In *Haskell '05: Proceedings of the*

2005 ACM SIGPLAN workshop on Haskell, pages 5–16. ACM Press, September 2005.

F. Baader and W. Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9, 0-262-18223-8.

T. Brehm. A toolkit for multi-view tracing of Haskell programs. Master's thesis, RWTH Aachen, 2001.

O. Chitil, C. Runciman, and M. Wallace. Transforming Haskell for tracing. In R. Pena and T. Arts, editors, *Implementation of Functional Languages: 14th International Workshop, IFL 2002*, volume 2670 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2002.

K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM Press, 2000.

L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

T. Dowd, Z. Somogyi, F. Henderson, T. Conway, and D. Jeffery. Run time type information in Mercury. In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 224–243, London, UK, 1999. Springer-Verlag.

R. Ennals and S. Peyton Jones. HsDebug: debugging lazy programs by not being lazy. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 84–87, New York, NY, USA, 2003a. ACM Press.

R. Ennals and S. Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 287–298, New York, NY, USA, 2003b. ACM Press.

J. Gibbons and K. Wansbrough. Tracing lazy functional languages. In *Proceedings of CATS'96: Computing the Australasian Theory Symposium*, Melbourne, Australia, January 1996.

A. Gill. Debugging Haskell by observing intermediate data structures. In *Haskell Workshop*. ACM SIGPLAN, September 2000.

A. Gill and C. Runciman. Haskell program coverage. In *Haskell Workshop*. ACM Press, September 2007.

D. Goldson. A symbolic calculator for non-strict functional programs. *Computer Journal*, 37(3):178–187, 1994.

C. Hall and J. O'Donnell. Debugging in a side effect free programming environment. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 60–68. ACM Press, 1985.

D. Herington. HUnit 1.0 user's guide. `http://hunit.sourceforge.net/HUnit-1.0/Guide.html`, 2002.

R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. 38(3):26–37, March 2003. In Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

X. Leroy, D. Rémy D. Doligez, J. Garrigue, and J. Vouillon. The Objective Caml system release 3.10, chapter 16. `http://caml.inria.fr/pub/docs/manual-ocaml/manual030.html`, 2007.

I. MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, The University of Melbourne, 2005.

N. Mitchell and C. Runciman. A static checker for safe pattern matching in Haskell. In *Trends in Functional Programming*, volume 6. Intellect, 2007.

L. Naish and T. Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.

H. Nilsson. Tracing piece by piece: Affordable debugging for lazy functional languages. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 36–47. ACM Press, 1999.

H. Nilsson. How to look busy while being as lazy as ever: The implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, May 1998.

W. Partain. The NoFib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

J. Silva. The Buggy Benchmarks Collection of Haskell Programs. Technical Report DSIC-II/13/07, Universitat Politècnica de València, 2007a. `http://www.dsic.upv.es/~jsilva/research.htm#techs`.

J. Silva. A comparative of algorithmic debuggers. In *Proc. of VI Jornadas de Programación y Lenguajes (PROLE'07)*, 2007b. `http://www.dsic.upv.es/~jsilva/papers/Prole07.pdf`.

Z. Somogyi and F. Henderson. The implementation technology of the Mercury debugger. *Electronic Notes in Theoretical Computer Science*, 30(4), 1999.

J. Sparud. *Tracing and Debugging Lazy Functional Computations*. PhD thesis, Chalmers University of Technology, Sweden, 1999.

A. Tolmach and A. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

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

M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23, 2001.

R. Watson. *Tracing Lazy Evaluation by Program Transformation*. PhD thesis, Southern Cross University, New South Wales, Australia, 1997.

S. Weirich. Replib: a library for derivable type classes. In *Haskell Workshop*, pages 1–12, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-489-8.

D. Xu. Extended static checking for Haskell. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 48–59. ACM Press, 2006.