DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46021 Valencia (SPAIN)

Informe Técnico / Technical Report

Ref. No.: DSIC-II/09/07 **Pages:**30
Title: Examine your laziness.
A lightweight procedural debugging technique for Haskell
Author(s): José Iborra and Simon Marlow
Date: April 2007
Keywords: Debugging, Lazy Evaluation, RTTI

Vº Bº
Leader of research Group Author(s)

Examine your laziness. A lightweight procedural debugging technique for Haskell

José Iborra[1]⋆ and Simon Marlow[2]

[1] Departamento de Sistemas Informáticos y Computación. Technical University of Valencia, Camino de Vera s/n, 46022 Valencia, Spain jiborra@dsic.upv.es
[2] Microsoft Research Ltd., Cambridge, U.K. simonmar@microsoft.com

Abstract. Writing debuggers for lazy functional languages is known to be difficult. While some recent progress has been made in this area, there is a dearth of practical tools which are applicable to real programs. We present a lightweight approach to procedural debugging, based on the traditional "stop-examine-continue" model using breakpoints. Our debugger sidesteps most of the problematic issues with laziness in a satisfactory way, delegating all compromising choices to the user. The debugger provides dynamic breakpoints with lazy, interactive inspection of the local bindings as the main feature. We have implemented a prototype on top of the GHC Haskell compiler, which is publicly available.

1 Introduction

Writing debuggers for lazy functional languages is known to be difficult. While some recent progress has been made in this area, there is a dearth of practical tools which are applicable to real programs. Why is it so difficult to come up with a practical debugger for lazy functional languages? This is not an empty statement: in the Haskell community there have been many attempts [3,10,5,12,22,16], which, in our opinion, fail to fill in the role of "simple, handy development tool", in one way or another.

We argue that, by delegating to the user all the problematic choices associated to laziness, it is possible to come up with an acceptable, conventional style debugging tool. The key aspects of our approach are twofold. First, we do not modify the evaluation model of Haskell for our debugger, but rather work our way around it by delegating to the user the responsibility of altering the lazy evaluation order. Second, in the same vein, we employ a program transformation approach to avoid modifying the runtime engine and the underlying compiler, which helps our approach to stay lightweight, flexible and easily embeddable.

These ideas have been tested with a prototype embedded in GHCi, the interactive environment included of the GHC tool set. The extended GHCi can,

⋆ This work has been supported by a participation as student in Google Summer of Code 2006, by the Spanish MEC under grant TIN 2004-7943-C04-02, and by the UPV under grant FPI-UPV 2006-01

by design, debug any program that can be loaded in it, which means that our tool supports the full GHC Haskell language. In addition it does not need any kind of compiler injected debug information, neither to the program nor to the libraries. This facility minimizes the configuration hassles imposed to the user, but also limits in some ways the debugging capabilities that GHCi can offer.

The present paper further develops ideas and work previously presented by David Himmelstrup [6], which introduced the *breakpoint* combinators and their embedding in GHCi. This work extends that effort with a refinement of the *breakpoint* combinators, a scheme for dynamic breakpoints, and the capability to lazily examine arbitrary values, which allows us to examine any local binding on a breakpoint. For that, we formulate a Run Time Type Information (RTTI) reconstruction scheme for GHC Haskell.

1.1 Debugging Haskell

Haskell programmers have been demanding an easy to set up debugger for a long time. In a web survey[3] conducted by the GHC team during the summer of 2005, one of the outstanding results was the need for a usable debugger.

Nowadays there are several debugging tools available for Haskell programmers, so this result may seem unjustified. Trace based debuggers such as Hat [22] and Buddha [16] rely on execution traces containing all the information about an execution to perform a postmortem analysis. The Haskell Object Observation Debugger (Hood)[5] is a more lightweight tool that provides lazy observations of evaluations, which can be used with debugging purposes, even though it is debatable that this makes Hood a debugger.

Even with all these choices, Haskell programmers seem to be unsatisfied. As a matter of fact, most of these tools have serious limitations in the subset of Haskell they support, the size of programs they can handle, or the complexity of installation/usage. For one reason or another, in practice they are not widely used by the community, as observed in another survey[4] conducted in 2006 for a retrospective article [7] on the Haskell language, where "only 3% of the respondents named Hat as one of the most useful tools and libraries".

Motivation for our approach The tools mentioned above share one common attribute: they explore alternatives to the traditional debugging technique of "stop-examine-continue". It is well known [21] that laziness poses a very serious challenge to this technique because of both the rather unpredictable evaluation order, and the absence of call frames in the stack. And even if there were call frames, the evaluation order would make the obtained call stack trace very different from the lexical call stack found in strict languages. The non lexical call stack induced by laziness would be unhelpful and confusing to the programmer, more details and examples of this are found in the HsDebug [3] paper.

[3] http://www.haskell.org/ghc/survey2005-summary.html
[4] http://www.cs.chalmers.se/~rjmh/Wash/Survey/Survey.cgi

Previous "stop-examine-continue" debugging efforts, such as HsDebug and Rectus [10], have addressed this problem by disabling tail call elimination, so that there is more information in the stack to start with, and then by hiding laziness, simulating a strict evaluation order, so that a lexical call trace can be obtained. These approaches however, require heavyweight changes to the compiler and the runtime component, and as a result none of them has found its way into mainstream.

Our goals are to stay lightweight and to produce an easily embeddable approach. Thus, we have opted for working our way around laziness without modifying the evaluation order nor the run time stack. Our approach provides few but useful debugging features, and sacrifices others such as call traces in the interest of being easy to set up and use.

Section 2 introduces the GHCi debugger, and showcases our approach. We tackle the problem of call frames in Section 3, whereas our RTTI based solution to the value printing problem is explained in Section 4. Related work is discussed in Section 6 and concluding remarks, together with an outline of future work lines, are developed in Section 7.

2 The GHCi debugger

2.1 Preliminaries

Haskell 98 A certain familiarity with Haskell syntax and type system is helpful, but not required, to follow the contents in this paper. Here we introduce the few required concepts. An algebraic datatype is declared with the following syntax:

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Just}\ a \mid \mathit{Nothing}$$

This declares the algebraic datatype *Maybe*. A value of type $\mathit{Maybe}\ a$ is either a value of type a tagged with the constructor *Just*, or the constructor *Nothing*. The constructors are functions with these types:

$$\mathit{Just} :: a \to \mathit{Maybe}\ a$$
$$\mathit{Nothing} :: \mathit{Maybe}\ a$$

Another example is the List datatype:

$$\textbf{data}\ [a] = a : [a] \mid [\,]$$
$$(:) :: a \to [a] \to [a]$$
$$[\,] :: [a]$$

The (:) constructor uses misfix notation; a value of type *List* is written as $(1 : 2 : 3 : [\,])$, but in practice Haskell offers syntactic sugar to write it as $[1, 2, 3]$.

A declaration of a renamed datatype, with the syntax

$$\textbf{newtype}\ \mathit{Set}\ a = \mathit{Set}\ [a]$$

introduces a new datatype *Set* whose representation is the same as $[a]$, with a single constructor *Set*. These renamed datatypes, from here on called *newtypes*, have always a single constructor with a single argument. The newtype is isomorphic to its argument and is used only during type checking; after this, the newtype constructor is eliminated and, once at run time, values have the same representation as the corresponding isomorphic datatype.

The GHC tool set GHC is a suite of Haskell tools which includes a compiler, an interactive environment, a package manager and a profiler. GHC supports the full Haskell 98 language plus a large number of extensions. We are concerned here with the GHC optimising Haskell compiler and the GHCi interactive environment. It is useful for this presentation to see the stages in the compile pipeline, as shown in Figure 1.

The source code is parsed and transformed into an AST, identifiers are made unique by the renamer, the AST is type checked, and the desugarer translates it into Core, a typed intermediate language based on System F extended with equality coercions [19], where the optimizations are performed. The code is transformed into another intermediate language called STG, from which it is translated by one of the several back-ends to (almost) executable code, and finally to object code.

One of these back-ends translates to bytecodes. GHCi is built on top of this back-end and provides a read-eval-print-loop interactive environment, allowing the user to enter arbitrary expressions and evaluate them in a scope. This encourages a style of development in which the basic blocks are functions, and can be tested and reasoned about individually. GHCi compiles to bytecodes and interprets them in an abstract machine. In order to maximize interaction, the focus is on fast compilation times over time efficiency, and for this GHCi disables the optimisation stages in the compile pipeline. In addition, it is possible to mix interpreted modules with object-compiled modules freely, and there is no performance penalty for using compiled modules in GHCi.

2.2 The GHCi debugger

The GHCi debugger is a basic, always active "stop-examine-continue" debugger, integrated in an interactive environment. Any expression entered by the user is susceptible of hitting a breakpoint. When this happens, a new instance of the

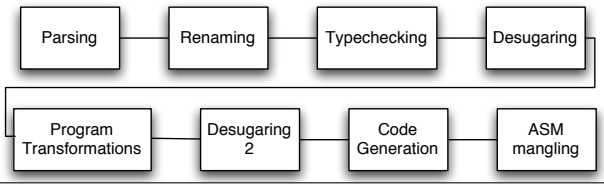

Fig. 1. Simplified GHC compiling pipeline

interactive environment is launched which makes available the local bindings in scope at the breakpoint.

Interaction with the debugger happens via a set of commands which extend the command set of GHCi. These commands allow the user to set and remove breakpoints at specific sites in the source code. Sites are points in the program where interesting events can occurr; we introduce sites in detail in section 3 The feature set of our debugging tool includes:

- Any program loadable in GHCi can be debugged (with the caveat that only bytecode compiled modules can have breakpoints).
- Breakpoints can be set on any subexpression located in a breakpoint site.
- When stopped at a breakpoint, the local bindings are made available and the user can evaluate any Haskell expression in this interactive environment.
- When stopped at a breakpoint, the intermediate structure of values can be examined, to see whether some of its subterms are suspended due to laziness, and it is possible also to selectively force subterms and observe their value.
- When stopped at a breakpoint, it is possible to alter the value of mutable variables.
- Step by step execution.
- Conditional breakpoints.
- It is not necessary to load all the modules that compose a program in debugging mode. The user can arbitrarily select the modules to be debugged.

An overview of the debugger We begin with an overview of the debugger, from the user's point of view.

```
loop :: String → Int → IO ()
loop filename i = handle (λ_ → return ()) $ do
                    let segname = printf "%s.03%i" filename i
                    putStrLn ("Reading file " ++ segname)
                    datum ← breakpoint (readFile segname)
                    f datum      -- some computation involving datum
                    loop filename (succ i)
```

Fig. 2. Example of a breakpoint

Breakpoints The user can manually set breakpoints using the *breakpoint* primitive. For example, consider the Haskell function in Figure 2. From a syntactic point of view *breakpoint* is a function of arity one. Its effect is to annotate an expression, interrupting the execution of the program whenever the wrapped expression is about to be evaluated. In the example, this will happen before reading the contents of the file pointed by *segname*.

```
> loop "datafile" 1
Stopped at a breakpoint in Loop.hs:5. Local bindings in scope:
  i :: Int, file :: [Char], size :: Integer,
  contents :: B.ByteString, filename :: [Char], segname :: [Char]
Loop.hs:5:20-37> show segname
"datafile.031"
Loop.hs:5:20-37>
```

Fig. 3. Output of the debugger

Dynamic breakpoints The previous example introduced breakpoint sites, and the *breakpoint* function. Our debugger does not require the user to manually insert breakpoints, the preferred behaviour is to dynamically set or remove breakpoints when the program is already loaded. For this, it is mandatory to load the program in the debugging tool.

Running the code in the debugger To load a program in the debugger, the user simply instructs GHCi to load the program as usual. To enable dynamic breakpoints, the user gives the `-fdebugging` option to GHCi. When this is active, the user can set breakpoints in the loaded program. For instance, we can set a breakpoint in the same place as the one set manually in the previous example by loading the program in GHCi and requesting a breakpoint in the first site to be found in line 5:

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.7, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( loop.hs, interpreted(debugging) )
Ok, modules loaded: Main.
*Main> :breakpoint add 5
```

Figure 3 shows the output produced when the breakpoint is hit. The prompt shows the source code location of the expression wrapped by *breakpoint*, as well as the list of local bindings (name and static type information).

The intention of the program is to process a list of files named "datafile.001", "datafile.002", etc, where we don't know a priori how many there are. The program tries to open one after another, composing the filename extension from the index given, until it encounters an IOException and then stops looping. This is the expected behaviour, but the programmer observes that the program stops without processing any file. Setting a breakpoint and examining the value of *segname* shows how the extension is not the expected, it should be `001` in-

stead of `031`. Looking two lines above, she finds out that the pattern in the first argument for *printf* had a bug[5].

3 Absence of call frames leads to instrumentation

In this section we tackle the problem of (lack of) call frames in lazy languages. A characteristic feature of the execution model induced by laziness is that the execution stack does not contain call frames. Call frames are important for debugging because they provide useful information about the execution context in run time.

The absence of call frames in the execution stack makes it difficult to obtain, at runtime, the list of local bindings in scope at a breakpoint. Either we restore call frames (or an analogous mechanism), or we need to know the local bindings for a breakpoint in advance. Our approach is to use a program transformation to inject the list of local bindings at every breakpoint, at compile time.

Central to this is the *breakpoint* combinator:

$$
\begin{aligned}
&\mathit{breakpoint} :: a \to a \\
&\mathit{breakpointCond} :: \mathit{Bool} \to a \to a \\
\\
&\mathit{breakpoint} = \mathit{id} \\
&\mathit{breakpointCond}\ \mathit{cond} = \mathit{cond}\ \text{`}\mathit{seq}\text{`}\ \mathit{breakpoint}
\end{aligned}
$$

Declaratively it is the identity function, but operationally it causes the program to stop its evaluation and display an interactive environment extended with the local bindings. The variant *breakpointCond* is just the conditional rendition, where the breakpoint is activated only if the first argument evaluates to True. For instance, a user can insert a breakpoint as follows:

$$\mathit{max}\ x\ y = \mathit{breakpointbold}\ (\mathbf{if}\ x > y\ \mathbf{then}\ x\ \mathbf{else}\ y)$$

The *breakpoint* function is nothing but a placeholder. The real work is made by the function *breakpointJump*, to which *breakpoint* is expanded by the compiler. *breakpointJump* is internally implemented by GHC and its evaluation in the interactive environment launches a new interactive session where the local bindings at the breakpoint site are made available.

The type of the jump function is more involved than that of vanilla *breakpoint*, as it explicitly receives the list of local bindings as well as their name and type information:

$$\mathbf{breakpointJump} :: \mathit{Ptr}\ () \to [\,\mathit{Opaque}\,] \to (\mathit{String}, \mathit{Int}) \to \mathit{String} \to a \to a$$

We explain this type through example. This is how the code above looks like after the expansion:

[5] it should be ``%s.%03i''

$$
\begin{aligned}
fn\ x\ y = \textbf{breakpointJump}\ ptr \\
&\quad [Opaque\ x, Opaque\ y] \\
&\quad (\texttt{"Test.hs"}, 1) \\
&\quad \texttt{"Test.hs:1"} \\
&\quad (\textbf{if}\ x > y\ \textbf{then}\ x\ \textbf{else}\ y)
\end{aligned}
$$

The first argument of *breakpointJump* carries the type and name information of the local bindings. We do not encode it in the source program; rather, since a debugging session is embedded in the normal interpreter session, it is simpler to inject a pointer to the relevant GHC data structures containing the relevant pieces of information. These can be accessed later by the debugger when the breakpoint is hit.

The second argument is a list with the values of the local bindings in the environment, packaged in an existential wrapper (*Opaque*). The rest of arguments to *breakpointJump* contain source location information, site number (explained in the next section) and finally the original expression.

The *breakpoint* expansion described above takes place in the desugaring step of the GHC compiler pipeline. At this stage, the code has already been type checked and annotated with full name and type information. In order to accommodate the desugaring of breakpoints a very modest extension to the desugarer was undertaken to allow tracking of local bindings in the environment.

3.1 Dynamic Breakpoints via Instrumentation

We employ a program transformation again to simulate dynamic breakpoints. The goal of dynamic breakpoints is to permit creation/deletion of breakpoints without the need to recompile the program. For this we transform the source program inserting conditional breakpoints at *breakpoint sites*. Most breakpoint based debuggers implicitly identify breakpoints with lines. Since Haskell favors expressions (as opposed to statements), we take a more flexible approach by associating breakpoints with breakpoint sites. These are points in the program where interesting events can occur. Breakpoints in our debugger can only be set at such sites, and thus these constitute also the unit of single-stepping.

These interesting sites are enumerated below.

- Binding sites, which include
  - function definitions
  - where expressions
  - let expressions (declarations and body)
  - alternatives in a case statement
  - lambda abstractions
- Monadic do-notation statements

We acknowledge that most breakpoint debuggers for expression-based languages, such as Tolmach's ML debugger [20], include function application as part of the set of breakpoint sites. Even though our approach in the current stage

does not, we are planning to do so in the future, for reasons elaborated later in this section.

The program transformation is performed, again, in the desugaring stage of the compiler. The instrumentation process is straightforward: it inserts a conditional version of vanilla breakpoints at every breakpoint site. As every breakpoint is assigned a unique site number by the *breakpointJump* expansion, this number is used to build a mapping between source code locations and breakpoints, stored in a global table. Every time a dynamic breakpoint site is hit, it performs a test with this table to check if the breakpoint is enabled. Consider the following example program:

```
main = do putStrLn "Input a string:"
          s ← getContents
          putStrLn (reverse s [])
  where reverse list acc = case list of
                             [] → []
                             (x : xx) → reverse xx x : acc
```

After being instrumented the result would be:

```
main = breakpoint_1 (
       do breakpoint_2 (putStrLn "Input a string:")
          s ← breakpoint_3 getContents
          breakpoint_4 (putStrLn (reverse s [])))
  where reverse list acc = breakpoint_5 (case list of
                             [] → breakpoint_6 acc
                             (x : xx) → breakpoint_7 (reverse xx (x : acc)))
```

These breakpoints are later expanded by the *breakpointJump* transformation, and the final program obtained is fed to the next stage in the compiler pipeline.

This *extremely* simple instrumentation scheme is made possible by lazy evaluation. Note how the argument to $breakpoint_1$ is the whole program. In a call-by-value language, the function is evaluated first, then the arguments, and finally the application of the arguments to the function. This means that, in a non lazy language, the whole program would be evaluated, and only then the breakpoint would happen. Obviously this would be far from the intended behaviour, and that is why other instrumentation based approaches, such as the ML Debugger, employ much more complex transformation schemes which involve large performance overheads[6]. In our case what happens is that the evaluation of the call to $breakpoint_1$ takes place first, while the evaluation of its argument, the whole program indeed, is suspended until needed. This means that the breakpoint is hit *right before* the evaluation of its argument, as intended. Laziness buys us extreme simplicity and, and as we show now, improved efficiency.

[6] of course, the ML debugger uses a number of other cunning techniques to greatly reduce this overhead

3.2 An even lazier breakpoint encoding

Dynamic breakpoints perform a test of enablement; if the breakpoint is not enabled, the rest of arguments to *breakpointJump* are *never used*. We can use laziness to our advantage by asking the compiler to be as lazy as possible when creating them. The idea is that, in the overwhelmingly common case that a breakpoint is not active, the arguments to *breakpointJump* will never get built, so that we only pay for the enablement test. In addition, we put all the arguments together in a tuple, except the site information required for the enablement test. This way, we allocate fewer suspended computations in the heap. The type and encoding of *breakpointJump* are now as follows, where the *lazy* function simply asks the compiler to be as lazy as possible when creating its argument. :

$$breakpointJump :: (String, Int) \rightarrow (Ptr\ (), [Opaque], String) \rightarrow a \rightarrow a$$

$$\begin{aligned} fn\ x\ y = breakpointJump\ &(\texttt{"Test.hs"}, 1) \\ &(lazy\ (ptr, [Opaque\ x, Opaque\ y], \texttt{"Test.hs:1"})) \\ &(\textbf{if}\ x > y\ \textbf{then}\ x\ \textbf{else}\ y) \end{aligned}$$

These small changes improve the overall efficiency of the breakpoint encoding mechanism by a factor of four.

3.3 Performance evaluation

There are two separate questions relative to performance overheads caused by instrumentation: overheads at compile-time and overheads at run-time. We have done extensive testing over the nofib [13] suite of benchmarks to provide quantitative measurements. 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". 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.

Compile-time How much longer will it take to load a program in debugging mode? To recap, the prototype extends the GHC compilation pipeline desugarer to:

- Keep track of the local bindings in scope in every point.
- Insert instrumentation breakpoints at breakpoint sites.
- Perform the transformation of *breakpoint* to *breakpointJump*.

The additional volume of code injected by our transformations does not go through the most expensive stages in the compilation pipeline: type checking and optimising. Type checking happens earlier in the pipeline, and the optimizer is

always disabled for GHCi, in order to speed up the load times of programs in the interactive environment. This contributes to make the overhead when compiling barely noticeable or, frankly, not at all. Our measurements over the nofib suite show an overhead of 20% or less in average, 90% in the worst case.

Run-time How much longer will it take to execute a program loaded in debugging mode? There are several factors that can generate some overhead every time a breakpoint site is evaluated. The direct costs are clear: a test to see if the user has enabled a breakpoint here, which involves one O(1) access to an array of breakpoints(per module), one O(log n) access to a hash table[7], and the construction of the arguments to *breakpointJump*. Luckily, laziness will prevent the evaluation of these arguments until they are really needed, which will happen only when the breakpoint is enabled; the overhead in this case comes from the allocation of a suspended closure in the heap for each argument. The indirect costs are far more subtle; the program transformations might alter the properties of the original program, altering the optimization of tail recursive functions for instance. Nevertheless indirect costs are bounded by the fact that the program-analysing optimizer is always disabled in GHCi, and therefore there is not much to lose.

To measure the overhead the nofib suite has been executed twice in GHCi, in normal and debugging mode. The results are shown in Figure 4 where T_C means the performance overhead at compile time produced by enabling `-fdebugging`; T_R the overhead at run time, LOC the *lines of code* metric and sites/loc the number of breakpoint sites per line. The results show that running times in debugging mode are 166% slower on average, with a worst case scenario of 657%. We include the name of the offending benchmarks in the worst cases. To give some perspective about these numbers, we note that when using GHCi instead of the native code compiler the overhead of interpretation averages 1000% in our measurements of the nofib suite.

Benchmarkset	$T_{C_{(avg)}}$	$T_{R_{(avg)}}$	$T_{R_{(max)}}$	LOC_{avg}	$\frac{sites}{loc}$
imaginary	20%	122%	311%(exp3_8)	22.6	0.38
real	30%	140%	657%(compress2)	1019.4	0.34
spectral	2.29%	197%	534%(partsof)	442.1	0.39

Fig. 4. Benchmark results

A user of GHCi usually takes advantage of the interactive environment by launching manually constructed expressions, focusing on one part of the program at a time. In addition, it is possible to individually select the modules to be instrumented and, as noted in Section 2.1, mix interpreted code with compiled code. In practice all this means that a user experiences no performance problems when using GHCi, and we expect that this remains the same even when

[7] where n is the number of modules loaded in the debugger

debugging is enabled. Thus, we are confident that the performance overheads do not diminish the usefulness of our approach.

3.4 Limitations

The main limitation of this scheme is related to exceptions and other unexpected conditions. We would like to be able to debug such scenarios, but right now we are limited to setting a breakpoint in the *error* function; *error* is defined in the Haskell prelude as

$$error\ msg = <raise\ exception>$$

One could insert a breakpoint in *error*, but there, of course, only the error message is available for examination! Ideally, we would be interested in examining the evaluation that led to the error, not the error itself. If there was a trace of the call stack, we would be able to go up in the trace and examine the call that led to the error. But the aforementioned lack of call frames prevents us from doing precisely that.

Another approach could be expanding the set of breakpoint sites so that it captures function applications too. By storing a trace of the most recent application contexts, it would be possible to use it for examining the call that immediately preceded the error condition. Of course in this case we should not present this trace to the user, since what it contains has nothing to do with the lexical call trace obtained in a strict language. Our trace would contain a lazy call trace which would only be confusing for the user. A decisive issue with this solution is the impact that the additional volume of breakpoints would have on performance.

3.5 Improvements and extensions

In practice, the debugger tries to minimize the number of sites inserted. Adjacent sites in the source tree are merged; in particular this happens when an expression binding starts with a let declaration: the site associated to the expression binding is logically (but not physically) adjacent to the site associated to the body of the let definition.

There is a subtlety with breakpoints in top level constants. Constants are evaluated only once by GHC operational semantics, and their value is updated in place; so for instance, if a source program defines:

$$deg\_factor = 2 * pi\ /\ 360$$

deg_factor will be evaluated only the first time it is accessed. If we wrap the expression in a breakpoint, the breakpoint too would trigger only the first time *deg_factor* is accessed, but not in future accesses. This is not necessarily a problem, but it can certainly be very confusing to a user who might expect to observe accesses to the constant using a breakpoint.

3.6 Other approaches

The traditional option to cope with the lack of call frames has been to not to. Rectus [10] builds its own lexical call stack via a heavyweight program transformation which modifies the original program enforcing a different evaluation order, creating its own decorative stack with call frames. HsDebug [3] modifies the GHC runtime system, which does not maintain a traditional call stack, so that it introduces decorative call frames; in addition to that HsDebug employs an altered evaluation order called *optimistic evaluation* to obtain a lexical call stack. The motivation is to allow easier and more versatile recovery of local bindings, and extended options for dynamic breakpointing. Essentially, their scheme would no longer require program transformation for injecting locals and dynamic breakpoints. The corresponding issues that need to be considered instead include, among others, providing source code location information and avoiding stack exhaustion in absence of the tail call optimisation.

4 Bindings examination demands a form of RTTI

4.1 Preliminaries

Types, type erasure, and unboxed values We say that a type τ_1 is more polymorphic than other τ_2 if there exists a substitution σ such that $\sigma\tau_1 = \tau_2$. A *monotype* is a type with no type variables.

Obviously at runtime all non-function values have monotypes. However due to the fact that GHC does *type erasure*, these types are effectively unknown. Type erasure is the process of discarding the type information in the artifacts produced by the compiler in the interest of time and space performance. This means that even though Haskell is a statically typed language, at run time of a GHC compiled program there is no available type information.

Unboxed values or unboxed types are a GHC extension to Haskell 98 for typing raw machine values. The set of valid unboxed types is predefined and closed, simple examples include `Int#`, `Char#`, `Float#` or `Word#`. Most often unboxed types are not directly manipulated by the user, instead they constitute somehow the operational mechanism underlying. For instance, these are the definitions of some standard Haskell 98 datatypes in GHC:

```
data Int   = I# Int#
data Char  = C# Char#
data Float = F# Float#
data Word  = W# Word#
```

Since unboxed values are wrapped by a datatype constructor, one says that they are *tagged* or *boxed*.

GHC Evaluation Model The most spread evaluation models for lazy functional languages are based on graph reduction techniques. One can imagine that

in the same way a functional program is a big expression, its runtime equivalent is a big graph with computations in the nodes and edges given by subterm relations. The graph is in-place updatable and in principle all the node computations are unevaluated; the reduction of the graph executes them until the initial node and all its subterms are fully evaluated.

GHC Heap Layout The main structure used by the GHC runtime is the *heap closure*, which constitutes the nodes of the evaluation graph. Each closure contains a header followed by a number of arguments; both the head and the arguments are one machine word in length each. The key element of the header is the pointer to an *info table*. In the info table there is additional information about the closure, including the closure type.

GHC deals with several types of closures; the relevant ones for this presentation are values, functions, and *thunks*. A value closure represents a non-function value, built by a full application of a data constructor. The arguments of a value closure correspond to the arguments of the associated data constructor and can be of two types: primitive (unboxed) values, or pointers to other closures (which constitute edges in the evaluation graph). A Function closure represents a function value; the arguments of a function closure are pointers to the values in scope at the function definition site (and constitute edges in the graph too). Finally thunk closures, or simply thunks, represent *unevaluated* computations. Overall, the process of reducing a graph involves forcing the thunks needed to make the top level value completely evaluated; forcing a thunk roughly means executing the associated computation and in-place updating the node with the resulting closure.

Info tables are closure type specific, containing diverse pieces of information for each closure type. But of course, info tables contain no type information as all. Type information is gone at runtime due to *type erasure*.

4.2 Examining variable values

Displaying values in a sensible manner is highly non trivial, even if we consider solely the problems of user-defined polymorphic datatypes and first class functions. But in addition, we must deal with the fact that some of these values will be unevaluated, and forcing their evaluation, which is necessary to show them, would be a serious alteration of the program being debugged. This alteration can easily cause non termination in a program execution that would ordinarily terminate, and the debugger can get stuck in an infinite computation when trying to display some value which was not supposed to be evaluated in the first place.

Other efforts such as Rectus or HsDebug have addressed this by simulating strictness instead of enforcing it, usually by setting a maximum bound of reductions after which no further attempts are done. Our approach is based on the idea of delegating these decisions to the user: we don't force anything, it is the user's responsibility. For this, we make the local bindings available in an interactive environment. We have extended it with a command similar to the "Display

intermediate reduced term" *dirt* primitive, described by Naish and Barbour [11]. Our primitive extends *dirt* with the capability of recovering the type information, as we will see in the next section. The debugger uses it to examine the bindings in scope, without forcing them, in a user friendly way.

There is another reason to work in an interactive environment. In a higher-order programming language, often these bindings will be themselves functions. Our approach cannot intensionally "print" functions, but the interactive environment allows the user to explore them extensionally, by calling them with a set of inputs and observing the outputs.

In order to make these values available in the interactive environment, the debugger must first give them proper types, otherwise the very own type system will forbid their examination, since in the interactive environment one can only construct and evaluate legally typed expressions. We need to recover the monomorphic type of values:

$$
\begin{aligned}
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] = 0 \\
&\mathit{length}\ (x : xs) = 1 + \mathit{length}\ xs
\end{aligned}
$$

The function *length* above calculates the *length* of a list; it has a parametrically polymorphic type on the elements of the list. For a breakpoint set on *length*, the debugger *aims* to find out the types of `x` and `xs`, but this information is unavailable at compile time, and hardly is at run time. GHC Haskell does not provide any Run Time Type Information (RTTI) mechanism, as other typed functional languages like [2] do. In the object oriented setting RTTI is also known as reflection or introspection and is a very common feature among such programming languages.

More generally, in the non-lazy setting there seem to be mainly two approaches to this problem. On the one hand, since all polymorphically typed values are created in calls to polymorphic functions, and at the call site the arguments are monomorphically typed, the ML debugger [20] uses a cunning technique based on the idea of traversing the call stack backwards examining the context until a full type is recovered. Let us show this with a simple example:

$$c = \mathit{length}\ [1, 2, 3, 4]$$

Inside the definition of *length* we know nothing about the type of `x`, however in the call to *length* inside the body of c there is the information we were looking for. With this technique, it is possible to reconstruct the type of `x` in this particular application of the *length* function.

The second approach is based on inspecting the store containing the object-code representation of values and decoding these to find out the types. This can be done usually with the help of debugging information of some kind, embedded in the compile time result, or with help of some RTTI mechanism provided by the language runtime. This approach is more low level and hence less portable than the former one, even though it is the one employed in the 99% of cases.

How do these techniques translate to our lazy setting? It is unfeasible to apply the first one, since there is no call stack. With regard to the second, it is certainly possible to recreate some form of RTTI for GHC, although there are some difficulties. Mainly, dealing with the fact that in the heap there are not only values, but most often delayed computations. In the following, we show a technique that manages this in a rather satisfactory way without any extra run time information.

4.3 Run Time Type Information in a lazy language

In this section we describe a mechanism for providing a partial RTTI mechanism, which the debugger uses to recover the types of local bindings in a breakpoint. Our RTTI mechanism is capable of returning type information for fully forced values in almost all cases, whereas when values are only partially evaluated an approximate type is returned. Such approximate types take the form of polymorphic types, with variables in the holes that RTTI could not fill. The RTTI primitive works by walking through the values in the heap and, in addition to type information, it provides generic pretty printing and evaluation information, telling how much of a value has been evaluated so far.

The debugger uses this RTTI mechanism to provide the user with a representation of the value, with placeholders in place of thunks, allowing manipulation and forcing of these thunks at will. After the user has forced some of the thunks, the debugger can use the RTTI mechanism again to obtain the desired full type.

Informal presentation Our RTTI mechanism is based on the following fact. It turns out that it is possible to recover some partial type information out of the info table of a value closure. Info tables for value closures are allocated statically during the execution of a program, and uniquely associated to a specific data constructor. That is, there is a single info table for every data constructor declared in the program source code. The RTTI mechanism is based on the capability of determining, for a value closure, its data constructor.

Thunks, function and closures of type other than value and primitive value are black boxes in this regard. Hence, if some subterm is suspended, it might be not possible to fully recover the type using our scheme. In this case, when the full type is not available, the RTTI mechanism returns a polymorphic type with type variables in the holes, as outlined before. The recovery of types for functions is more severely restricted since there is no way to 'force' a function in order to give it a more defined type; in this case we are stuck with a permanent hole, i.e. a polymorphic variable.

Compensating for program transformations The first consideration after obtaining the data constructor is to compensate the difference between what the user declared, and what the compiler ended up producing. Intermediate encodings and optimizations in the compiler can alter the type signatures of constructors.

Specifically, we need to ignore extra arguments due to type class dictionaries[8][14] and existential dictionaries. We benefit from integration with the compiler here, since it provides a mapping from the source to the final type signatures for data constructors; the mapping is consulted and those arguments that were not in the original signature are ignored by the RTTI mechanism.

RTTI reconstruction Recovering the type of a value closure is addressed as a simple problem of type inference. Type inference focuses on recovering the principal type [1], i.e. the most general or the most polymorphic type. We replace the concept of polymorphism with the concept of incompletely defined type, and the rest of the procedure largely is the same. Our type inference algorithm deals with the equivalent of user provided annotations but in a *dual* way. In our case, the annotations come from statically inferred type information. And since we can safely assume that these are always correct and always most general, our type inference algorithm can override them when needed to obtain a concrete type, contrarily to traditional type inference where user provided annotations are not necessarily most general and must be always respected.

Type inference style constraints are generated by walking the heap, starting from the closure of interest. Suppose that we are examining a value t, and the static type environment tells us that t is a list, i.e. $t :: [a]$. This is akin to a user provided type annotation [9]. We want to find out the complete type of t. At this point in the dynamic program execution, the values of this list are partially evaluated as follows, where underscores represent suspensions:

$$t = \mathit{Just}\ \_ : (\mathit{Just}\ (1 : \_) : \_)$$

That is, the first element is only partially evaluated, the second element is a slightly more evaluated, and the remaining list is suspended. Figure 5 shows

[8] Type class constraints on functions are typically desugared into an extra argument containing a dictionary of instances, an implementation mechanism for late binding

[9] Note that it wouldn't be a valid type annotation in GHC, because it is not true; however in our dual scheme it is perfectly valid, since it is *more polymorphic* than the inferred type

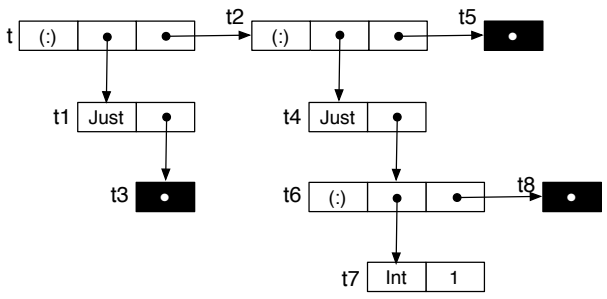

Fig. 5. Heap representation of an intermediate value

how `t` is represented in the heap and how it can be examined to extract type constraints. Closures are represented as black boxes in the diagram. The set of constraints is generated by walking all the subterms, as follows:

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

Recall the definition of the list datatype and the type of its two constructors:

$$\begin{aligned} &\textbf{data}\ [a] = a : [a] \mid [] \\ &(:) :: a \to [a] \to [a] \\ &[] :: [a] \end{aligned}$$

The first constraint corresponds to the static piece of type information we had, while the others come from the heap. In these, the left-hand side of a constraint is formed from the heap representation, while the right-hand side comes from the data constructor signature. Solving the equations of the example, via the standard mechanism of (syntactic) unification, gives $t :: [\mathit{Maybe}\ [\mathit{Int}]]$.

Note that there are two sources of constraints: run time heap examination and compile-time static information about types. In the example, the static information was $t :: [\alpha]$, in this case unhelpful, but in the general case, when the argument is not sufficiently evaluated, static information can provide a more concrete type than at run time. Had t been a thunk, it is better to use the less polymorphic type $[\alpha]$ instead of just α.

Unification modulo newtypes In section 2.1 we described the Haskell 98 **newtype** language construct, and how newtypes constructors are eliminated once type checking is completed. Even though there is no trace of newtypes in the heap, newtypes are present in the signatures of data constructors which we use to generate constraints. This means there are additional equations to consider when doing unification for type reconstruction. In the example above, the newtype declaration for *Set* gives rise to the equation $\mathit{Set}\ a = [a]$.

Consider a slightly modified version of the RTTI reconstruction example of Figure 5, in which the static type information is $t :: \mathit{Set}\ \alpha$. We proceed as before, walking the heap to collect the typing constraints. But since *Set* is defined as a newtype, we will not find it in the heap. Instead, what we find is exactly the same as before, raw lists. The representation of t in the heap is the same as before, even if its type now is different. The new set of constraints is the same as before:

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

Now we cannot launch unification directly, because the two equations for t simply do not unify: type $Set\ \alpha_1$ does not unify with type $[\alpha_2]$. In the next section we give a set of type checking rules that address this problem.

Even worse, there are cases in which it is not possible to recover the newtypes, no matter what we do, without further help from the compiler. Consider the previous example in which the type of t was $t :: [Maybe\ [Int]]$. If instead we had $t :: [Maybe\ (Set\ Int)]$, the lack of evidence for the inner Set prevents from recovering this type; what would be obtained is the wrong $t :: [Maybe\ [Int]]$.

Since the newtype mechanism is considered to be a special case of datatype where Haskell is free to optimize away the constructors, an acceptable workaround might be a program transformation that replaces newtypes by equivalent data declarations, thus disabling the optimization without changing the semantics of the code. Unfortunately, this only affects newtypes declared in the code being debugged. Declarations living elsewhere, such as the ones in libraries or compiled code, are set in stone and cannot be disabled. It would be necessary to have a special compilation mode for debugging which generates the modified artifacts, and include duplicated versions of every library compiled in this mode. This would be an acceptable solution, although it seems to be against our initial goals. Also, it is not clear if this equivalence holds in every case. For the moment we have not explored it in depth, but we do not discard doing so in the future.

Definition In the following we present a set of type checking rules that informally describe our algorithm for Run Time Type Inference in the presence of newtypes. Its main goals are two: to recover the type of a closure, and to 'guess' its newtypes as far as possible.

The key observation is that when we reconstruct the type of a constructor application, by definition we always obtain a newtype-less type. Recovering a potential newtype wrapping can only by guided by

- the type signature of the (data constructor of the) term that contains it.
- a compile-time type annotation.

Therefore, newtype information always flows from the top of a term tree to the bottom.

The language for types is defined in Figure 6. We will use this type system to talk about the language defined in Figure **??**. A type can be a variable, a type constructor A applied to 0 or more type arguments, a function type, or a newtype construct $n_\mu(\rho)$, which denotes a newtype with tag μ isomorphic to the type ρ. For example, the Haskell type `Set a` would be represented as $n_{Set}([a])$.

$$
\begin{array}{lll}
\text{Environments} & \Gamma ::= \Gamma,\, x : \tau \mid \cdot \\
\text{Types} & \rho ::= \alpha \mid \rho \to \rho \mid A\,\overline{\rho} \mid n_\mu(\rho) \\
\text{Tags} & \mu ::= B \mid \beta \\
\\
\text{Type variables} & \alpha \\
\text{Tag variables} & \beta
\end{array}
$$

Fig. 6. Syntax of the type language

A tag can be a base tag B corresponding to a user newtype declaration, or a newtype tag variable.

Figure 7 shows the rules for the RTTI type system. The main judgement takes the form $\Gamma \vdash \mathtt{t} : \rho$, which means "in environment Γ the term `t` has type ρ". We assume that the type signature of a constructor, in rule APP, has had its universally quantified variables instantiated by a capture-avoiding substitution. This allows us to avoid explicitly dealing with quantified variables in our presentation of the type system, in the interest of clarity. Rule THUNK says that the type of a hole is free. The auxiliary set of rules NEW define an ordering or instantiation $<$ over newtypes, such that a newtype can be used in the place of the type it is isomorphic to.

Rule APP handles value closures, i.e. constructor applications. It makes a very important assertion, which is that the type of the arguments of a constructor signature is exact except by the result type. Since we are doing typechecking over data, the arguments correspond to the subterms. This implies that the signature of the parent term ultimately determines the type of its subterms, however it only correctly determines its own type "up to newtype instantiation". The most accurate type is the smallest one according to the newtype instantiation relation $<$ seen as an ordering.

$$
\boxed{\Gamma \vdash t : \rho}
$$

$$
\frac{}{\Gamma \vdash \square : \rho}\,\text{THUNK}
$$

$$
\frac{\Gamma \vdash c : \rho_1 \to \ldots \to \rho_n \to \rho \qquad \Gamma \vdash u_i : \rho_i \quad \rho < \rho'}{\Gamma \vdash c\; u_1 \ldots u_n : \rho'}\,\text{APP}
$$

$$
\boxed{\rho < \rho}
$$

$$
\frac{}{\alpha < n_\mu(\alpha)}\,\text{NEW\_V}
$$

$$
\frac{\overline{\rho} < \overline{\rho}'}{A\;\overline{\rho} < n_\mu(A)\;\overline{\rho}'}\,\text{NEW\_A}
$$

$$
\frac{}{n_\mu(\rho) < n_{\mu'}(n_\mu(\rho))}\,\text{NEW\_N}
$$

Fig. 7. Type checking rules

Recovering the type of a value closure is addressed as a problem of type inference. Type inference focuses on recovering the principal type [1], the type that makes the smallest commitment about the values of type variables, the most polymorphic type. We replace the concept of polymorphism with the concept of incompletely defined type, and the rest of the procedure largely is the dual: the goal is to recover the most defined, i.e. least polymorphic, type. Our type inference algorithm deals with the equivalent of user provided annotations in the *dual* way too. In our case, the annotations come from compile time type information, and since we can safely assume that these are always correct and always most general, our type inference algorithm can and does instantiate them when needed to obtain a more concrete type, contrarily to traditional type inference where user provided annotations are not necessarily most general and can never be overriden.

The type system presented here is peculiar in that it is does not satisfy the preservation of typing under substitutions. It is easy to see that this is the case, since type variables in this system generally represent *unknown* types, and not polymorphic types. An additional restriction is that it cannot type a solitary thunk. The type of a thunk in isolation is always an existentially quantified type variable. Typing thus has to start from a constructor application value. As another manifestation of duality, the system is stable under the converse of substitutions, i.e. if $\rho' = \sigma\,\rho$ where σ is a substitution, is a valid type for a term t, then ρ is also a valid type. Under this consideration, talking about principal types has also the dual meaning.

The rules are in syntax directed form and a type inference algorithm is extracted naturally from them. Basically, the rules are applied to a run time value to generate the set of constraints. We have no proofs for it, but conjecture that unification over this set of constraints yields always the principal type in the sense informally sketched before, and does so in finite time. However, the process of generating the constraints, which is performed earlier or at the same time as type inference, does not necessarily terminate[10] when conducted over ciclic data structures. We sketch a solution[11] in the next section.

Implementation First, some definitions. We divide an arrow type into segments, which are the biggest components which do not themselves contain arrows. The *result* is the rightmost segment, and all other segments are *arguments*. We define the *subterm relation* among closures as follows: closure `t1` is a subterm of closure `t2` iff `t2` contains a pointer to `t1`. A newtype declaration `newtype N = N t` induces an equivalence relation $N = t$ on types, and thus we talk of equality *up to newtypes*. However, we are not interested in enforcing this congruence, because there is no a priori single representant for a equivalence class. We sometimes need to *upgrade* `t` to the equivalent type `N`.

Every individual constraint `lhs = rhs` produced in a RTTI session involves a pair of types where one of them, by convention `lhs`, represents a reconstructed

[10] since in a naive implementation, it loops

[11] which does not resort to FFI tricks for comparing pointers

type for a runtime term, and the other one (`rhs`) comes either from a constructor signature instantiated with fresh variables, or from static type information. An important observation is that the type reconstructed for a given constructor application is, by definition, not a newtype, since it comes from the result type of a data constructor signature. Reconstructing a potential newtype wrapping can only be guided by:

- the type signature of the (data constructor of the) term that contains it, *i.e.* the parent term.
- a compile-time type annotation, if this is the term corresponding to the top level closure.

This implies that newtype information always flows from the top of a term tree down to the bottom. The arguments in the type signature of a term's constructor accurately describe the type of its subterms regarding newtypes. In constrast, as we have seen, the result type is correct only up to newtypes.

Keeping this in mind, if the constraints are arranged in descending order according to the subterm relation, it turns out that every type variable resolved in an unification step is accurate with respect to newtypes. That is, it will never arise the need to upgrade its value in the substitution in a forthcoming unification step.

Now, since we haven't changed the set of constraints, and given that syntactic unification is unaware of the newtype equivalence relation, a unification step will fail in the presence of a newtype. For instance, the constraints of the example are already in subterm ordering, and the first failure happens at the result types of constraint two. We distinguish two cases, depending on whether the failure happens at a result type position:

1. If it does, then the `lhs` is a newtype and the `rhs` is not, by definition. It is safe to upgrade the type constructor in the `rhs` result type in order to make both sides unify.
2. If it does not, then since the `rhs` is correct regarding newtypes, the `lhs` is missing a newtype. But since we know that the `lhs`, thanks to the ordering of the constraints, is accurate regarding newtypes, then failure.

The procedure described does not involve any additional constraints, and relies heavily on the ordering of the constraints to be correct. Upgrading the types inside the computed substitution, *i.e.* upgrading the value of a variable x at constraint `b` after x has been given some value by a previous constraint `a`, would put in danger the correctness of unification. Since the `lhs` of the rules consist solely of variables, upgrades in `lhs` correspond always to this case, whereas upgrades in the `rhs`, which occurr only in the result type, are confined to a modification of the constraint and do not alter the substitution. The procedure never upgrades at the `lhs`, thus this gives a notion on the safety of the approach.

Of course, when the real type involves newtypes, we cannot in general guarantee to recover it from the runtime value. Our algorithm for RTTI allows newtypes to be introduced via static type information only.

4.4 Extensions

Although we haven't implemented it in our tool, the scheme can be extended to recover the arity of a function. Arity information can be found in the info table associated to a function closure. For this extension it is not necessary to extend the syntax of the heap language nor the set of typing rules; it suffices with injecting this information to the typing environment in the obvious way, whenever we encounter a function closure in the heap.

We can define an analogue of optimistic evaluation but tuned for our purposes of recovering monotypes. The idea is to perform reduction steps on those subterms of a value which are 1. untyped, and 2. suspended. It would probably be necessary to set a maximum depth that bounds the number of reductions needed. Such scheme would be provided as an option to the user, with a disclaimer warning about the alteration of semantics. In practice, this alteration would be acceptable in most cases, and this feature would make the debugging experience simpler and more similar to other debuggers, where you do not need to worry about types.

4.5 Alternate approaches

We have not explored the alternate approach of annotating closures and/or info tables with type information. This probably would have a serious impact in space and time efficiency, and might as well require a redesign of the intermediate languages used in the compiler pipeline, since the STG language is untyped. Given the amount of complications we foresee, it seems more adequate to work on improving the type reconstruction approach.

5 Controlling program execution

Debugged programs run inside the interactive environment; the user asks for the evaluation of an expression and this launches a computation. When a breakpoint is hit the computation is halted, the interactive environment resumes control, the local bindings are now in scope, and the user is allowed to enter arbitrary expressions. Once the user is satisfied, the interactive environment continues the program execution.

We outline now the implementation of the transition between the program and the interactive environment. Calls to *breakpointJump* are dynamically linked to an action which launches a new interactive environment initialized with the state of the top level interactive environment, and adds the local bindings from the breakpoint scope to the evaluation scope. The lifespan of this new interactive environment is rather short: it dies when the user requests to continue the execution. As soon as this happens, the embedded environment gives control back to the original computation, returning the expression wrapped by the breakpoint.

6 Related work

Instrumentation Automated instrumentation of source code is a old idea. Trace based debuggers such as Hat and Buddha instrument Haskell source programs to generate a trace of an execution containing all the possibly interesting information in order to analyze it postmortem. The ML time-travel debugger [20] instruments ML source programs with debugging primitives, which create and keep track of a lexical call stack and also allow to replay the execution of the program, going back to previous states, making it possible to simulate time travel. Our approach to debugging is similar in that we use instrumentation too, although we do not offer the time travel capabilities. The lazy setting makes it more difficult to create and handle a lexical call stack, and we have not yet pursued this goal.

The Haskell debugger Rectus [10] follows an approach based on instrumentation to simulate strictness and recreate an internal lexical call stack. The program transformations required by Rectus are rather complex and its evaluation mechanism imposes a very serious overhead, estimated by the authors around 10000%, although efficiency did not seem to be a factor in Rectus design. Another weakness of their approach is that the scheme used for printing the locals at a breakpoint does not manage types and often requires the user to explicitly cast them; a segmentation fault strikes in the event of a wrong coercion. We can imagine that Rectus could make good use of the RTTI mechanism introduced here to solve this problem.

Lazy languages Our debugger's user features, which include breakpointing, querying and interactive evaluation, are fairly ordinary. What makes our system unusual is how it integrates these features in a lazy pure functional language.

Rectus manages to provide these features too, and even more, such as call stack traces, but is almost prohibitively slow. There has been research on formal strictness simulation techniques such as optimistic evaluation [4] which make it possible to lift these performance restrictions. For instance the HsDebug prototype used them to provide an advanced procedural debugging tool [3] for GHC. Sadly these techniques are fairly heavyweight and do not seem to be under active research at the moment.

The Haskell Object Observation Debugger (Hood) [5] can be compared to our tool in the way it provides observations. Contrary to what its name says Hood is not a debugger; the programmer annotates its code with the *observe* combinator to produce an advanced flavor of traces, known as diagnostic writes, especially adapted to lazy languages. One drawback of using Hood is that the programmer is forced to modify the original code, and thus needs to recompile manually every time a new observation is added.

Higher-order We have been focusing on laziness as the main factor to consider in the design of a debugger for Haskell. Another factor with a very high impact is higher-order programming, which is a style of programming based on crafting functions from smaller ones, called combinators, used as building blocks. Parsing

and combinator libraries would be the prime example. It can be frustrating to debug this kind of programs with a traditional debugger, where it is not possible to examine values representing functions. The declarative Haskell debugger Buddha used reification and advanced program transformation [15] techniques to provide intensional examination of functions. It is also possible in Buddha to extensionally examine all the applications of the functions after the program execution. In contrast, our approach provides access to the function itself: while it is not possible to do intensional examination, having access to the interactive environment permits to observe the function extensionally by applying arbitrary inputs and examining the output.

Other declarative languages We have already mentioned a number of debuggers for Haskell. Somogyi and Henderson present a procedural debugger [17] for the declarative language Mercury [18]. Their approach is very similar to ours: they use sites too, which are called events in their terminology, and instrumentation to insert breakpoints at sites. The main difference is that they instrument the code generated at the end of the compiler pipeline, in this case C. They benefit from an already existing RTTI mechanism for Mercury to solve the problem of typing and examining locals, and from the availability of a call stack to easily locate locals in breakpoints. Mercury is a strict language, so this is not surprising. Our work shows that it is possible to do the same in a lazy language, jumping through a few extra hoops.

MacLarty presents a declarative debugger [9] for Mercury that relies on the technology developed for the Mercury procedural debugger to efficiently compute the traces in a piecemeal fashion; seeing how to combine our debugger with a declarative debugging effort in Haskell is certainly an interesting research problem.

7 Concluding remarks

We have used source code instrumentation and advanced heap examination to build a very simple debugger based on traditional techniques for a complex language with unusual dynamic semantics and an optimizing compiler. We have made available a prototype[12] that implements our ideas on a mainstream compiler. In contrast with the elegance of tracing based systems our approach may appear crude, but its general compatibility and interactivity make up for its apparent lack of charms. The main contribution is to show that it is indeed possible to build a simple "stop-examine-continue" in the presence of laziness; laziness has proven to be both a blessing and a curse.

The implementation of the prototype in GHC is extremely simple, about 2000 lines of modifications to GHC in Haskell code; versus around 70000 for the GHC compiler as a whole. But of course, had we lacked a platform on top of which to build, the task would have required far more resources.

[12] http://haskell.org/haskellwiki/Ghci/Debugger

7.1 Future work

We have seen how the RTTI mechanism has some weaknesses related to newtypes and especially functions. In section 4 we have outlined some roads to explore with regard to the second point. The other option to recover type information, call stack analysis as done by the ML debugger, requires the availability of a lexical call stack. An lexical call stack thus would be interesting for this reason and for a better understanding of the debugged program. Call stack traces are a key feature of "stop-examine-continue" debuggers and we would certainly like to have them in our solution. There is already an initiative to provide them via a lightweight program transformation, but it is still early to present the results.

Another route of work is improving the mechanism for breakpoints. The shortcomings of the current encoding are overhead, inability to retrieve the local bindings at an arbitrary execution point, transformation of the source program into a semantically but not operationally equivalent one, and finally due to the overhead, unavailability of sites in function applications. Sites on function applications are interesting because they are the key to debugging error conditions. Handling error conditions is a first priority; some ideas have been outlined in section 3.4, but further research is needed. Thus, future work includes the design of an improved mechanism for breakpoints with reduced overhead, which permits having breakpoint sites in function applications. Alternatively, improving the current mechanism would be desirable too.

Finally, it would be very desirable to provide intensional, syntactical examination of lambdas, in the style of Buddha.

7.2 Acknowledgements

Thanks to everyone who read early drafts of this paper, including German Vidal, Bernie Pope, Josep Silva, and Santiago Escobar for their help and comments. Finally, a thankful note is in order to David Himmelstrup for his help in the early days of this project.

References

1. Luis Damas and Robin 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.
2. Tyson Dowd, Zoltan Somogyi, Fergus Henderson, Thomas Conway, and David 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.
3. Robert Ennals and Simon 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, 2003. ACM Press.

4. Robert Ennals and Simon 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, 2003. ACM Press.
5. Andy Gill. Debugging Haskell by observing intermediate data structures. In *Haskell Workshop*. ACM SIGPLAN, September 2000.
6. David Himmelstrup. Interactive debugging with ghci. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 107–107, New York, NY, USA, 2006. ACM Press.
7. Paul Hudak, John Hugues, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. 2007. To Be published.
8. Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, 2007.
9. Ian MacLarty, Zoltan Somogyi, and Mark Brown. Divide-and-query and subterm dependency tracking in the mercury declarative debugger. In *AADEBUG*, pages 59–68, 2005.
10. Oleg Murk and Lennart Kolmodin. Rectus: Locally eager haskell debugger. 2006.
11. Lee Naish and Tim Barbour. Towards a portable lazy functional declarative debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.
12. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Department of Computer and Information Science, Linköpings universitet, S-581 83, Linköping, Sweden, May 1998.
13. Will 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.
14. John Peterson and Mark Jones. Implementing type classes. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 227–236, New York, NY, USA, 1993. ACM Press.
15. B. Pope and L. Naish. Specialisation of higher-order functions for debugging. In M. Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier Science Publishers, 2002.
16. B. Pope and L Naish. A program transformation for debugging Haskell-98. *Australian Computer Science Communications*, 25(1):227–236, January 2003.
17. Zoltan Somogyi and Fergus Henderson. The implementation technology of the mercury debugger. *Electronic Notes in Theoretical Computer Science*, 30(4), 1999.
18. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *J. Log. Program.*, 29(1-3):17–64, 1996.
19. Martin Sulzmann, Manuel Chakravarty, and Simon Peyton-Jones. System F with type equality coercions. *submitted to POPL*, 2007.
20. Andrew P. Tolmach and Andrew W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.
21. Philip Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, 1998.
22. 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.

A An extended debugging session

The `qsort` pet function, which we will debug

```
qsort [] = []
qsort (a:as) =  (qsort left) ++ [a] ++ (qsort right)
 where (left,right) = (filter (<=a) as, filter (>a) as)
```

The debugging session

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.7, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Prelude> :set -fdebugging
Prelude> :load Qsort
[1 of 1] Compiling Main             ( Qsort.hs, interpreted(debugging) )
Ok, modules loaded: Main.
*Main> :break add 2
Breakpoint set at main:Main:2:16
*Main> qsort [10,2,5,9,1,7,3,8,4]
Stopped at a breakpoint in Qsort.hs:2. Local bindings in scope:
  _result :: [a], as :: [a], a :: a, left :: [a], right :: [a]
Qsort.hs:2:16-51> :p a
a = (_t1::a)
Qsort.hs:2:16-51> :p as
as = [(_t2::a),(_t3::a),(_t4::a),(_t5::a),(_t6::a),(_t7::a),(_t8::a),(_t9::a)]
Qsort.hs:2:16-51> :sp as
as = [_,_,_,_,_,_,_,_,_]
Qsort.hs:2:16-51> as
This is an untyped, unevaluated computation. You can use seq to force its
    evaluation and then :print to recover its type
Qsort.hs:2:16-51> :t as
as :: [GHC.Base.Unknown]
Qsort.hs:2:16-51> seq a ()
()
Qsort.hs:2:16-51> :p a
a = 10
Qsort.hs:2:16-51> :p as
as = [(_t10::a),(_t11::a),(_t12::a),(_t13::a),(_t14::a),(_t15::a),(_t16::a),(_t17::a)]
Qsort.hs:2:16-51> :p left
left = (_t18::[a])
Qsort.hs:2:16-51> seq left ()
()
```

```
Qsort.hs:2:16-51> :p left
left = [2 | (_t19::[Integer])]
Qsort.hs:2:16-51> left
[2,5,9,1,7,3,8,4]
Qsort.hs:2:16-51> seq right ()
()
Qsort.hs:2:16-51> :p right
right = []
Qsort.hs:2:16-51> :continue
Returning to normal execution...
Stopped at a breakpoint in Qsort.hs:2. Local bindings in scope:
  _result :: [a], as :: [a], a :: a, left :: [a], right :: [a]
Qsort.hs:2:16-51> :p a
a = 2
Qsort.hs:2:16-51> seq left ()
()
Qsort.hs:2:16-51> :p left
left = [1 | (_t1::[Integer])]
Qsort.hs:2:16-51> left
[1]
Qsort.hs:2:16-51> seq right ()
()
Qsort.hs:2:16-51> :p right
right = [5 | (_t2::[Integer])]
Qsort.hs:2:16-51> right
[5,9,7,3,8,4]
Qsort.hs:2:16-51> :p as
as = [5,9,1,7,3,8,4]
Qsort.hs:2:16-51> :c
Returning to normal execution...
Stopped at a breakpoint in Qsort.hs:2. Local bindings in scope:
  _result :: [a], as :: [a], a :: a, left :: [a], right :: [a]
Qsort.hs:2:16-51> :p a
a = 1
Qsort.hs:2:16-51> :p as
as = []
Qsort.hs:2:16-51> :c
Returning to normal execution...
[1,2
Stopped at a breakpoint in Qsort.hs:2. Local bindings in scope:
  _result :: [a], as :: [a], a :: a, left :: [a], right :: [a]
Qsort.hs:2:16-51> :p a
a = 5
Qsort.hs:2:16-51> :p as
as = [9,7,3,8,4]
```

```
Qsort.hs:2:16-51> seq left ()
()
Qsort.hs:2:16-51> :p left
left = [3 | (_t1::[Integer])]
Qsort.hs:2:16-51> left
[3,4]
Qsort.hs:2:16-51> :breakpoint list
(1) Main : (2, 16)
Qsort.hs:2:16-51> :break del 1
Breakpoint deleted
Qsort.hs:2:16-51> :c
Returning to normal execution...
,3,4,5,7,8,9,10]
*Main>
```