# HsDebug : Debugging Lazy Programs by Not Being Lazy

Robert Ennals
Computer Laboratory, University of Cambridge
Robert.Ennals@cl.cam.ac.uk

Simon Peyton Jones
Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

## 1 Motivation

Debugging has long been recognised as one of the weaknesses of lazy functional languages. Conventional (strict, imperative) languages almost invariably use the "stop, examine, continue" paradigm (Section 2), but this approach just does not work well for with lazy evaluation. This difficulty has led to fascinating research in novel debugging techniques (Section 7).

We argue that conventional debugging techniques have perhaps been dismissed too quickly. We present two alterations to the evaluation model of lazy functional languages that allow conventional debugging techniques to be successfully applied. Transient tail frames allow tail-calls to be visible to the debugger without affecting space complexity (Section 4). Optimistic Evaluation causes evaluation to only use laziness when absolutely necessary, thus preserving the termination behaviour of Lazy Evaluation, while reducing its confusing effect on program state (Section 5).

We have implemented these ideas in HsDebug, an extension to the GHC tool set (Section 6). Our debugger is, by design, "cheap and cheerful". Its results are not as predictable, nor as user-friendly, as those of (say) Hat — but they come cheap. HsDebug can debug an entirely un-instrumented program, and it can do a lot better if the compiler deposits modest debug information (much like a conventional debugger). Furthermore, an arbitrary subset of the program can be compiled with debug information – in particular, the libraries need not be. Furthermore, program transformation and optimisation are unaffected.

## 2 How the Dark Side Do It

A debugger has long been one of the standard tools that is shipped with any strict, imperative, programming language. The vast majority of these debuggers follow a "stop, examine, continue" model of debugging, as used by GDB [19]. Such debuggers are characterised by the following features:


*The Haskell Workshop'03* August 28th, 2003, Uppsala, Sweden.


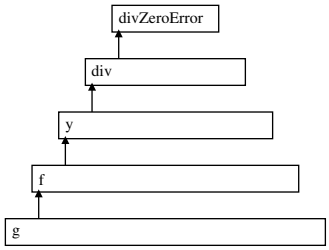


**Figure 1. Strict Evaluation**

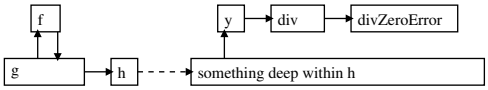


**Figure 2. Lazy Evaluation with Tail Call Elimination**

- The programmer can request that execution stop at a *breakpoint*. A breakpoint may correspond to a point in the source code of the program. Alternatively, it may be the point at which some logical property becomes true.
- When a program is in its stopped state, the programmer can examine the state of the program. From this state, the programmer is able to obtain an understanding of how the program came to be in the state that it is.
- The programmer can call functions within the program and can directly manipulate the program state.
- Once the programmer has finished examining the state and adjusting their breakpoints, they can request that execution continues until the next breakpoint is hit.

One of the most important pieces of information that a debugger allows the programmer to observe is the call stack. In a strict language, the nesting of frames on the call stack will correspond directly to the nesting of function call in the program source code. Consider the following program:

```
f x = let y = (3 `div` x) in Just y

g = case f 0 of Just y -> h y
```

If this program is executed strictly, then the call stack will be as illustrated in Figure 1. When the division by zero error occurs, the stack will clearly show that this took place inside the evaluation of $y$, in the call to $f$ made from $g$. It is likely that the stack will also hold the argument that $f$ was called with.

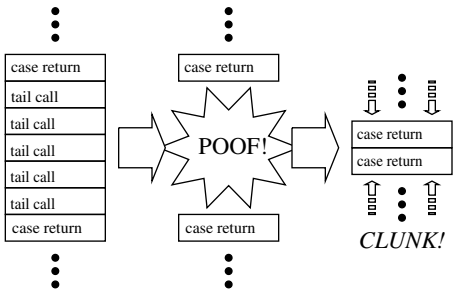


**Figure 3. Tail call frames disappear if we have too many**

## 3 Failing to Debug Lazy Programs

What happens if we try using this style of debugging for a lazy functional language such as Haskell [13]? If the same program given earlier were to be evaluated in a typical lazy language implementation, then the call stack would be as illustrated in Figure 2. Lazy evaluation has scrambled the execution order and tail call elimination has removed stack frames that would have provided useful information. The result is a mess that is very difficult to debug from.

This clash between "stop, examine, continue" debugging and lazy languages is considered to be so severe that, as far as we know, nobody has ever made a serious attempt to implement such a debugger for a lazy language. In many ways, this rejection of conventional debugging models has been a good thing, as it has led to the development of several extremely powerful alternative approaches (see Section 7). However, we believe that conventional debugging techniques should not be written off. In the sections that follow, we explain a series of tweaks to lazy evaluation that have allowed us to produce an effective "stop, examine, continue" debugger for Haskell.

## 4 Eliminating Tail Call Elimination

One simple way to increase the amount of information available from stacks is to disable tail-call elimination. This will provide us with extra stack frames that tell us more about the call chain that has taken place. This idea is not new; Strict languages that implement tail recursion often allow tail recursion to be disabled when a program is being debugged. For example, the CMU Common Lisp [4] environment disables tail call elimination if the debug level is set greater that "2".

While turning off tail call elimination makes debugging easier, it will also cause some programs to use massively more stack space than they would otherwise use. A tail recursive loop that would normally consume no stack space will now push a stack frame on every iteration.

Fortunately, there is no need for a program to fail with an "out of stack" error if it has a stack full of tail call return frames. The only purpose of tail call return frames is decorative; thus it is perfectly ok to delete them. Our solution is thus to delete all tail frames every time we run out of stack or perform a garbage collection. Figure 3 illustrates this concept.

It is important that we delete tail frames at a garbage collection, even if we are not short on stack. This is because our tails frames record all the arguments passed to a call, and so may be holding onto heap objects.

## 5 Optimistic Evaluation

Disabling tail call elimination makes debugging significantly easier, but it does not quite bring us to the point at which a "stop, examine, continue" debugger becomes usable. Lazy evaluation will still scramble the evaluation order, causing expressions to be evaluated on stacks that are different to the stack in which the expression was defined.

The obvious way to prevent lazy evaluation making debugging harder is to not use lazy evaluation. If we evaluate the program strictly, we will find that our stack trace mirrors the nesting of expressions and so find debugging much easier. Unfortunately, lazy evaluation is used for a reason. If we attempt to evaluate a lazy program strictly, then we are likely to find that the program runs massively slower, or even fails to terminate.

Fortunately, it turns out that we can avoid the problem of non termination by introducing the concept of abortion. We can set a time limit on how long we allow ourselves to evaluate the right hand side of a `let` before giving up and returning to the caller. This idea has previously been used to increase the performance of lazy evaluation, both by Eager Haskell [6, 5] and also by Optimistic Evaluation [1]. We have implemented our debugger on top of Optimistic Evaluation, but the same techniques could also be applied to Eager Haskell.

## 6 HsDebug

HsDebug is a "stop, examine, continue" debugger for Haskell. It has been implemented as part of GHC and currently lives on the Optimistic Evaluation branch in the GHC public CVS. HsDebug relies on Optimistic Evaluation and Transient Tail Frames, both of which have been added to GHC. While HsDebug has a long way to go before it becomes as powerful a tool as GDB, it is already very useful. The current feature set includes the following:

- Any program compilable with GHC can be debugged
- Breakpoints can be set in any Haskell function
- The original arguments of all calls on the stack can be inspected
- Closures on the heap can be pretty printed
- Functions and Thunks can be pretty printed - giving their source location, and free variables.
- Exceptions can be intercepted
- The program can be single-stepped

All in all, HsDebug feels very similar to GDB and should feel familiar to anyone who is already comfortable with GDB. HsDebug is currently very rough round the edges, and source level debugging is currently very incomplete, but it is already showing itself to be a useful tool.

Programs compiled for HsDebug run slightly slower than normal. This is partly due to the need to turn off some of the more confusing code transformations and partly due to the extra overhead of pushing and removing tail frames. In most cases this speed reduction is insignificant. The worst slowdown that we have managed to provoke (a tight, tail-recursive inner loop) slows down by around a factor of 3.

# 7 Other Approaches to Lazy Debugging

## *7.1 Tracing*

Most previous work on debugging of Lazy programs has focused on tracing. Systems such as Freja [12, 11, 8], Buddha [14] and Hat [17, 22] augment a program so that it creates a trace as it executes. This trace gives a history of the evaluation that took place. For each value (e.g. 5), a link can be provided to the redex that evaluated to produce that value (e.g. $3+2$)) and to the larger evaluation that this evaluation was part of (e.g. $f(3)$).

Once such a trace has been built up, it can be explored in many different ways. Hat allows one to look at any object on the heap and find out how it came to be created. Other work allows evaluations to be observed in the other in which they would have taken place under strict evaluation, creating a similar environment to a traditional debugger [9, 10].

Hat and Buddha run the program to completion before exploring the debug trace. While this simplifies the implementation, it makes debugging of IO awkward. In a traditional debugger, one can step over IO actions and observe the effect that the actions have on the outside world. This is made significantly more difficult if all actions take place before debugging starts. Indeed, Pope [14] says that it is assumed that Buddha will only be applied to the sub-parts of a program that do not perform IO operations. Freja does better in this respect by building its trace while debugging.

One drawback of trace based debugging approaches is performance. If every evaluation is to be logged, then a very large amount of information must be recorded. Not only does the recording of such information take time - it also takes space. Freja works round this problem by only storing a small amount of trace information and then re-executing the program if more is needed, however this is quite tricky to implement, particular when IO is involved. There has also been considerable work on reducing the amount of trace information generated for redex trails [18].

HsDebug is definitely less powerful and less elegant than trace based debuggers. It is however considerably simpler, considerably faster, and does not require extra space.

## *7.2 Cost Centre Stacks*

Cost Centre Stacks [7, 16] extend a program so that it maintains a record of the call chain that the current expression would have, were it being evaluated strictly. The information obtainable from a cost centre stack is thus very similar to that available from the real stack under Optimistic Evaluation. It is also quite similar to the trail created by Freja. Cost Centre Stacks were developed for use in profiling however it is plausible that a debugger could be written that made use of them. Such a debugger could show the user the current cost-centre stack rather than the actual execution stack, providing the user experience of strict evaluation, without having to actually evaluate the program strictly. We believe that this approach may be worth exploring.

## *7.3 HOOD*

HOOD [2] can be seen as an extension of traditional "printf debugging". The programmer adds annotations to the program that allow intermediate program states to be observed. HOOD goes a lot further than "printf" debugging by allowing lazy values and functions to be observed only to the extent to which they have been used. A sophisticated viewer application [15] allows the programmer to view and manipulate traces resulting from an execution. While HOOD is extremely powerful, the need to add manual annotations can make it awkward to use.

## *7.4 Time Travel Debugging*

Time Travel Debuggers extend the "stop, examine, continue" model further, by allowing the program to run backwards to a breakpoint as well as forwards. Often one will find that the reason for something going wrong is that part of the program state has become invalid. In such a case, it can be extremely useful to run the program backwards from the point at which something went wrong, to the point at which the state became invalid. Examples of time travel debuggers include the excellent O'Caml debugger [3] and the now sadly defunct SML/NJ debugger [21, 20]. Many of the advantages of Time Travel Debugging can also be achieved by Tracing, and vice versa.

# 8 Conclusions

HsDebug works. While it does not have the elegance of other lazy debuggers, it is fast, it is simple, and it can debug any GHC program. While HsDebug still has a lot of rough edges, it is already usable and demonstrates quite effectively that the traditional imperative approach to debugging can be successfully applied to lazy functional languages.


# Acknowledgments

We are grateful for the generous support of Microsoft Research, who have funded the studentship of the first author. We would also like to thank Simon Marlow, Henrik Nilsson and Alan Lawrence, who have provided useful suggestions.


# A Example Debugging Sessions

In this appendix, we give a transcript from a real HsDebug session. The programming being debugged is the following:

```
 1: module Main where
 2:
 3: import System.IO
 4:
 5: main :: IO ()
 6: main = do
 7:   putStrLn "List length?"
 8:   str <- getLine
 9:   let c = read str
10:   let xs = ints c
11:   putStrLn $ "hd = " ++ (show $ head xs)
12:   putStrLn $ "tl = " ++ (show $ last' xs)
13:
14: ints 0 = []
15: ints n = n : (ints (n-1))
16:
17: last' :: [Int] -> Int
18: last' (x:xs) = last' xs
```

The following session demonstrates simple IO debugging, breakpoints, exception catching, stack backtraces, and printing of closures:

```
bash-2.03$ hsdebug ./demo3
(hsdebug) break Main.ints$
Breakpoint set at address 0x804932c
(hsdebug) continue
List length?
2
breakpoint hit: Main.ints$ (0x804932c)
args = (2)
(hsdebug) cont
breakpoint hit: Main.ints$ (0x804932c)
args = (1)
(hsdebug) c
breakpoint hit: Main.ints$ (0x804932c)
args = (0)
(hsdebug) c
hd = 2
Exception raised:
data: GHC.IOBase.PatternMatchFail.con <0>
        ["demo3.hs:2" ++ 0x402c80b4]
(hsdebug) print 0x402c80b4
"0: Non-exh" ++ 0x402c7fa8
(hsdebug) where
locals = ()
        args = (0x80a005d)
#0: Main.lvl2                   at demo3.hs:18
        update : 402c78cc
        args = ()
#1: Main.last'                  at demo3.hs:17
        args = ([])
#2: Main.last'                  at demo3.hs:17
        args = ([I# 1])
#3: Main.last'                  at demo3.hs:17
        args = ([I# 2, I# 1])
#4: Main.main                   at demo3.hs:6
#5: xs.s3g9                     at inlined "putStrLn"
        env = ([I# 2, I# 1])
    catch frame : GHC.TopHandler.topHandler.info
    startup code
end of stack
(hsdebug)
```

## References


[1] R. Ennals and S. Peyton Jones. Optimistic Evaluation: An adaptive evaluation strategy for non-strict programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2003.

[2] A. Gill. Debugging haskell by observing intermediate data structures. In *Proceedings of the 4th Haskell Workshop*, 2000.

[3] X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. *The Objective Caml system, documentation and user's guide*. INRIA, 1998. Available at `http://pauillac.inria.fr/ocaml/htmlman/`.

[4] R. MacLachlan, editor. *CMU Common Lisp Users Manual*. Carnegie Mellon University, 1992.

[5] J.-W. Maessen. Eager Haskell: Resource-bounded execution yields efficient iteration. In *The Haskell Workshop, Pittsburgh*, 2002.

[6] J.-W. Maessen. *Hybrid Eager and Lazy Evaluation for Efficient Compilation of Haskell*. PhD thesis, Massachusetts Institute of Technology, June 2002.

[7] R. Morgan and S. Jarvis. Profiling large-scale lazy functional programs. *Journal of Functional Programming*, 8(3), May 1998.

[8] 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, Nov. 2001.

[9] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming (PLILP '92)*, volume 631 of *Lecture Notes in Computer Science*, pages 385–399, Leuven, Belgium, Aug. 1992.

[10] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, July 1994.

[11] H. Nilsson and J. Sparud. The evaluation dependence tree: an execution record for lazy functional debugging. Technical report, Department of Computer and Information Science, Linkoping University, 1996.

[12] H. Nilsson and J. Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4(2):121–150, April 1997.

[13] S. Peyton Jones, R. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the programming language Haskell 98. `http:/haskell.org`, Feb. 1999.

[14] B. Pope. Buddha: A declarative debugger for Haskell. Honours thesis, Dept. of Computer Science, University of Melbourne, Australia, June 1998, 1998.

[15] C. Reinke. Ghood — graphical visualisation and animation of haskell object observations. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, 2001.

[16] P. Sansom. *Execution profiling for non-strict functional languages*. Ph.D. thesis, University of Glasgow, Sept. 1994.

[17] J. Sparud and C. Runciman. Tracing lazy functional computations using redex trails. In *PLILP*, pages 291–308, 1997.

[18] J. Sparud and C. Runciman. Complete and partial redex trails of functional computations. *Lecture Notes in Computer Science*, 1467, 1998.

[19] R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, 1991.

[20] A. P. Tolmach and A. W. Appel. Debugging standard ML without reverse engineering. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 1–12, New York, NY, 1990. ACM.

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

[22] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multipleview tracing for haskell: a new hat. In *Proceedings of the Haskell Workshop*, 2001.