Hat-Explore: Source-Based Trace Exploration

Olaf Chitil, University of Kent

Abstract: Experience shows that users of the HAT viewing tools find it hard to keep orientation and navigate to a point of interest in the trace. Hence this paper describes a new viewing tool where navigation through the trace is based on the program source. The tool combines ideas from algorithmic debugging, traditional stepping debuggers and dynamic program slicing.

1 Introduction

HAT [24] still has a number of shortcomings. One of these is that it is often hard to navigate through large computations. By using the existing viewing tools together and calling one tool from the other we can in principle quickly reach any point in the trace. However, the questions: "where am I in the trace?" and "how do I get to the point I want to see in the trace?" often occur. We require orientation guides.

One candidate for an orientation structure immediately springs to mind: the program source. We are likely to be familiar with the source, because we wrote it, read it beforehand and/or will have to modify it. All expressions in the trace originate from the source. Usually the source is far shorter than the huge computation trace.

None of the existing viewing tools take advantage of the source. All HAT viewing tools display only expressions and equations of the traced computation. The tools just allow opening a source browser with the cursor positioned at the redex or at the definition of the function of current interest.

HAT-EXPLORE is a new HAT viewing tool that allows simple, free navigation through a trace while providing orientation based on the program source. HAT-EXPLORE combines ideas from algorithmic debugging, traditional stepping debuggers and dynamic program slicing.

2 Functionality

The Screen Layout The display of HAT-EXPLORE is divided into two parts: the call stack and the source. The stack shows a sequence of reductions, where each reduction called the function applied in the reduction below. We say that function `f` calls function `g`, if the *application* of `g` appears in the definition body of `f`; so this stack resembles the runtime stack of an eager evaluator, not a lazy one. The last reduction on the call stack is called the current reduction, the reduction that is currently in focus. In the source the *call site* of the redex of the current reduction is underlined.

Navigation through the Computation We navigate through a computation via the cursor keys: up to the caller of the current reduction, down to the first callee, and left and right to siblings. In the program source the call sites of the siblings are highlighted (but not underlined).

Algorithmic Debugging HAT-EXPLORE supports algorithmic debugging, that is, error-location based on declarations by the user about which reductions are correct. We can declare if the current reduction is correct or incorrect with respect to our intentions and also change any previous such declaration. HAT-EXPLORE uses several colours for highlighting: correct reductions are green, incorrect ones are yellow, unknown/undeclared ones are blue. When the tool identifies a reduction as faulty, it is highlighted in red.

Example Let us work step by step through an example session for the faulty insertion sort program. The tool starts with the reduction of `main`. (There is no call site of `main`, hence its definition is underlined.)

```
==== Hat-Explore 2.04 ==== Call 1/1 ==========================
 1. main = {IO}


---- Insert.hs ---- lines 1 to 3 -----------------------------
main = putStrLn (sort "sort")

sort :: Ord a => [a] -> [a]
```

We cannot say if this reduction is correct, but only press cursor down to look at the children:

```
==== Hat-Explore 2.04 ==== Call 1/2 ==========================
 1. main = {IO}
 2. putStrLn "os" = {IO}

---- Insert.hs ---- lines 1 to 3 -----------------------------
main = putStrLn ( sort "sort" )

sort :: Ord a => [a] -> [a]
```

The first child is a reduction of a trusted function and hence assumed to be correct. So we press cursor right to look at the second child:

```
==== Hat-Explore 2.04 ==== Call 2/2 ========================
 1. main = {IO}
 2. sort "sort" = "os"

---- Insert.hs ---- lines 1 to 3 ----------------------------
main = putStrLn ( sort "sort" )

sort :: Ord a => [a] -> [a]
```

This reduction disagrees with our intentions and hence we press 'w' to declare the reduction as wrong:

```
==== Hat-Explore 2.04 ==== Call 2/2 ========================
 1. main = {IO}
 2. sort "sort" = "os"

---- Insert.hs ---- lines 1 to 3 ----------------------------
main = putStrLn ( sort "sort" )

sort :: Ord a => [a] -> [a]
```

To find out why the reduction is wrong we have to look at the children, so we press cursor down:

```
==== Hat-Explore 2.04 ==== Call 1/2 ========================
 1. main = {IO}
 2. sort "sort" = "os"
 3. insert 's' "o" = "os"
---- Insert.hs ---- lines 3 to 5 ----------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)
```

We press 'c' to declare the reduction as correct and then press cursor right to look at the second child:

```
==== Hat-Explore 2.04 ==== Call 2/2 ========================
 1. main = {IO}
 2. sort "sort" = "os"
 3. sort "ort" = "o"
---- Insert.hs ---- lines 3 to 5 ----------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x ( sort xs )
```

We press 'w' to declare the reduction as wrong and then press cursor down to inquire further:

```
==== Hat-Explore 2.04 ==== Call 1/2 ==========================
 2. sort "sort" = "os"
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
---- Insert.hs ---- lines 3 to 5 -----------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)
```

We press 'w' to declare the reduction as wrong:

```
==== Hat-Explore 2.04 ==== Call 1/2 ==========================
 2. sort "sort" = "os"
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
---- Insert.hs ---- lines 3 to 5 -----------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x ( sort xs )
```

So the reduction `insert 'o' "r" = "o"` is faulty. We have located the fault, it must be in the definition of `insert`. If we are not convinced, we can still press cursor down to see that `insert 'o' "r" = "o"` has only a single child, a reduction of a trusted function, which is assumed to be correct:

```
==== Hat-Explore 2.04 ==== Call 1/1 ==========================
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
 5. 'o' <= 'r' = True
---- Insert.hs ---- lines 7 to 9 -----------------------------
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x <= y then x : ys else y : (insert x ys)
```

Declaring the (in)correctness of the current reduction is separate from navigation; it does not automatically navigate to a new reduction. Thus we are free to declare (in)correctness of reductions in any order. In practice it is often much easier to recognise an incorrect reduction than being sure that a reduction is correct. HAT-EXPLORE allows us to look at all children of a redex, determine that one of them is incorrect, and continue exploring that reduction, without having to consider the correctness of its siblings. We might not even use algorithmic debugging at all but simply navigate freely through the computation; we will do so in particular when there is no error but we aim to understand how the traced program works.

Program Slicing HAT-EXPLORE optionally marks the definitions of all functions within which the fault must be. These definitions comprise the faulty slice. With increasing information about correct and incorrect reductions the faulty slice shrinks until the faulty reduction has been identified. The shrinking of the faulty slice shows us that we are making progress, it

may quickly exclude large parts of the program, possibly parts that had been wrongly suspected, and when the faulty slice has become small we may spot the fault straight away without even having to continue algorithmic debugging to its end. In the preceding screen-shots the faulty slice is *emphasised*.

The faulty slice does not have to encompass whole definitions. When a reduction $f \ldots = \ldots$ is faulty, it is unnecessary to add the whole definition of function f to the faulty slice. For a specific reduction usually only parts of the definition body of the reduced function are evaluated because of pattern matching, conditionals and lazy evaluation. The fault can only be in that part of the definition.

Code Coverage By declaring the root reduction of the computation, `main = {IO}`, as incorrect and asking HAT-EXPLORE to mark only the evaluated faulty slice, we can obtain the slice of the program that was evaluated at all during the whole computation.

3 Conclusions

HAT-EXPLORE is a new trace viewing tool for the HAT system that enables us to navigate freely and intuitively through the trace of a Haskell 98 program. The display of the source together with a stack of reductions for the context give good orientation. The tool combines algorithmic debugging with program slicing and the user interface of a traditional stepping debugger.

Feedback on the HAT-day was mostly positive. However, the user interface was considered too complex: users might be confused by numerous expressions highlighted in several colours. It was also considered confusing that when the fault had been located, the *call site* was highlighted in red in the source code, not the *definition site* which is faulty and needs to be modified. Considering that slicing is too slow in practice for non-trivial computations (it requires a traversal of most of the trace), this feature might best be moved to a separate, non-interactive viewing tool.

HAT-EXPLORE demonstrates that it is relatively easy to extend the HAT system by a new viewing tool for which it was not designed originally. During the development of HAT-EXPLORE it became clear that the implementations of most viewing tools include functionalities that are likely to be useful for future tools and hence should be moved into separate libraries.

A more detailed description of HAT-EXPLORE is given in [5].

Bibliography

[1] Stephen R. Adams. Efficient sets — a balancing act. *Journal of Functional Programming*, 3(4):553–562, 1993.

[2] Krasimir Angelov and Simon Marlow. Visual Haskell: a full-featured Haskell development environment. In *Haskell'05: Proc. 2005 ACM SIGPLAN Haskell Workshop*, pages 5–16, Tallinn, Estonia, 2005. ACM Press.

[3] Thomas Böttcher and Frank Huch. A Debugger for Concurrent Haskell. In *Draft Proc. 14th Intl. Workshop on Implementation of Functional Languages (IFL'2002)*, pages 129–141, Madrid, Spain, 2002. Tech. Report 127-02, Dept. de Sistemas Informaticos y Programacion, Universidad Complutense de Madrid.

[4] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless G-Machine. In *Proc. 1988 ACM Conference on LISP and Functional Programming*, pages 244–258, Snowbird, Utah, USA, 1988. ACM Press.

[5] Olaf Chitil. Source-based trace exploration. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004*, LNCS 3474, pages 126–141. Springer, 2005.

[6] Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for tracing. In *Implementation of Functional Languages, 14th Intl. Workshop, IFL 2002, Revised Selected Papers*, pages 165–181. Springer LNCS 2670, 2003.

[7] K. Claessen and R. J. M. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. 5th Intl. ACM Conference on Functional Programming*, pages 268–279. ACM Press, 2000.

[8] K. Claessen, C. Runciman, O. Chitil, J. Hughes, and M. Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *Advanced Functional Programming, 4th International School (AFP 2002)*, pages 59–99. Springer LNCS 2638, 2002.

[9] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *Proc. 4th Intl. Workshop on Automated Debugging (AADEBUG 2000)*, Munich, Germany, 2000. `http://xxx.lanl.gov/abs/cs.SE/0012009`.

[10] Tom Davie. Animation of lazy evaluation in Haskell using Hat traces. BSc project dissertation, Dept. of Computer Science, University of York, 2004.

[11] Mike Dodds. Using trace data to diagnose non-termination errors. MEng project dissertation, Dept. of Computer Science, University of York, 2004.

[12] Keith Hanna. Interactive visual functional programming. In *Proc. 7th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP'02)*, pages 145–156, Pittsburgh, USA, 2002. ACM Press.

[13] T. Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Notices*, 19(6):58–69, June 1984.

[14] Daan Leijen. wxHaskell – a portable and concise GUI library for Haskell. In *Proc. ACM SIGPLAN 2004 Haskell Workshop*, pages 57–68, Snowbird, Utah, September 2004. ACM Press.

[15] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'2005)*, pages 15–26, Chicago, Illinois, June 2005. ACM Press.

[16] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping University, April 1998.

[17] Henrik Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *J. Funct. Program.*, 11(6):629–671, 2001.

[18] Bernard Pope and Lee Naish. Practical aspects of declarative debugging in Haskell 98. In *Proc. 5th ACM SIGPLAN Intl. Conf. on Principles and Practice of Declarative Programming (PPDP'03)*, pages 230–240, Uppsala, Sweden, 2003. ACM Press.

[19] Niklas Röjemo. Highlights from nhc: a space-efficient Haskell compiler. In *FPCA '95: Proc. 7th Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 282–292, La Jolla, USA, 1995. ACM Press.

[20] Colin Runciman. TIP in Haskell — another exercise in functional programming. In Rogardt Heldal, Carsten Kehler Holst, and Philip

Wadler, editors, *Proc. Glasgow Workshop on Functional Programming 1991*, pages 278–292. Springer Verlag BCS Workshops in Computing, 1992.

[21] Tom Shackell and Colin Runciman. Faster production of redex trails: The Hat G-Machine. In Marko van Eekelen, editor, *Proc. 6th Symposium on Trends in Functional Programming (TFP 2005)*, pages 135–150. Tartu University Press, Estonia, 2005.

[22] Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In *Proc. 9th Intl. Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, pages 291–308, London, UK, 1997. Springer-Verlag.

[23] Andrew Peter Tolmach. *Debugging standard ML*. PhD thesis, Princeton University, Princeton, NJ, USA, 1992.

[24] Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In Ralf Hinze, editor, *Proc. 2001 ACM SIGPLAN Haskell Workshop*, pages 151–170, Firenze, Italy, September 2001. Universiteit Utrecht UU-CS-2001-23. Final version in ENTCS 59(2).