Hat Day 2005:
work in progress on the Hat tracing system for Haskell

Colin Runciman (Editor)

Technical Report YCS-2005-395
Department of Computer Science
University of York, UK
October 2005

Preface

News that the 2005 Ascot horse-racing meeting would take place in York not only guaranteed a week of traditionally exotic headgear in the city; it also prompted a group of us, working on a software system called HAT, to convene our own HAT DAY.

The HAT System: HAT is a software toolkit for obtaining and examining traces of Haskell 98 programs. The traces HAT works with are called *augmented redex trails*, and represent in detail how the results of a program were derived, including cross references to the source. The source-to-source transformation tool HAT-TRANS [6] lifts programs to self-tracing variants that write a trace to file as they run. Different trace-viewing programs, such as HAT-OBSERVE and HAT-TRAIL, support different ways of working with traces [24]. Early versions of HAT (the 1.x series) worked with the NHC98 compiler, but subsequent versions (the 2.x series) are more portable, working also with the widely-used GHC compiler. The current version, HAT 2.04, is freely available from `http://haskell.org/hat`.

The Working Papers: HAT DAY was an informal sequence of talks and discussion, mainly of work still in progress, and sometimes of ideas only just beginning to germinate. The short working papers in this report reflect the topics of the day: most are about new ways of viewing and working with traces, with or without results from prototype tools; but one paper describes a new method for generating traces, and another looks at a possible semantic model for tracing.

Acknowledgement: Our work on tracing functional programming has been supported by EPSRC research grants at York (GR/K64334 *Selective tracing of functional computations using graph reduction with redex trails* and GR/M81953/01 *Advanced redex trails: fully-fledged tracing technology for functional programs*), and at Kent (EP/C516605/1 *A theory of tracing pure functional programs*). EPSRC training awards have also supported some students doing HAT-related work.

Colin Runciman, York, October 2005

Contents

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].

Hat-Delta — One Right Does Make a Wrong

Thomas Davie & Olaf Chitil, University of Kent

Abstract: We outline two methods for locating bugs in a program. This is done by comparing computations of the same program with different input. At least one of these computations must produce a correct result, while exactly one must exhibit some erroneous behaviour. Firstly reductions that are thought highly likely to be correct are eliminated from the search for the bug. Secondly, a program slicing technique is used to identify areas of code that are likely to be correct. Both methods have been implemented. In combination with algorithmic debugging they provide a system that quickly and accurately identifies bugs.

1 Introduction

Program bugs often do not manifest themselves immediately. A user will often execute a program several times with correct results, only to later find a specific input that produces an erroneous behaviour. We aim to use the information that can be gathered from correct computations of the program to diagnose bugs in an erroneous computation.

We describe two new ways of exploiting information from correct computations when debugging the program. The extra information narrows the position of a bug and thus improves on earlier methods. The first method, based on finding repeated reductions, eliminates small sections of program computation from the search for bugs. After eliminating sections of computation a second method based on a program slicing can be used. Both methods provide heuristics which guide the debugger. The slicing method is less reliable in its predictions, but can remove larger numbers of questions when it is correct.

We use the two methods to locate bugs in Haskell programs. However, the technique can be applied in any situation where algorithmic debugging can be applied.

```
sort [] = []
sort (x:xs) = insert x (sort xs)

insert x [] = [x]
insert x (y:ys)
  | x < y     = x:y:ys
  | otherwise = insert x ys
```

Figure 1: Buggy program used in Figure 2

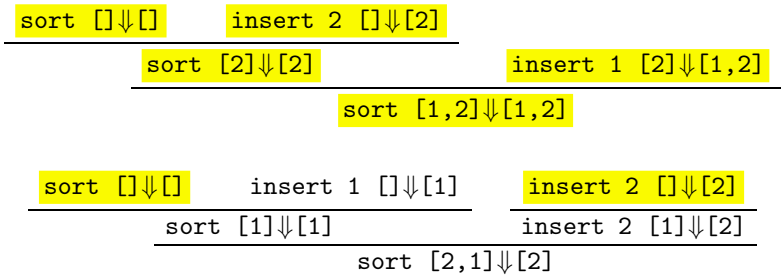

Figure 2: Computation graphs of program in Figure 1. The upper computation is correct, but the lower one is erroneous

2 Correct Subcomputations

When a computation produces a correct result, it is reasonable to assume that all its subcomputations produce correct results. This statement is not always true – it is possible that two bugs cancelled each other out. It is, however, remarkably hard to come up with non-trivial examples in which the statement is incorrect, and as such there is a very low chance of marking incorrect reductions correct. With this in mind, it is immediately possible to eliminate an area of computation graph from the search for a bug.

If a subcomputation appears in a correct computation graph, as well as that of an erroneous computation, it is possible to label it as a correct computation. The debugger can therefore eliminate it from the search for a bug. Not only the root reduction is said to be correct, but also all subcomputations involved in the reduction. In Figure 2 we can see that `sort [1,2]` computing the correct result has caused the computation of `sort []`, `insert 2 []`, `sort [2]` and `insert 1 [2]` to be considered correct as well. At present we consider a computation having a correct result to be a good indicator that all reductions within it are correct, and we label them as such (although we will allow the user to disable this behaviour). Experimental results will later tell us if such reductions should be trusted less.

Figure 2 shows both a correct and an erroneous computation of the program shown in Figure 1. Computation that occurred in the correct trace is

```
1   sort [] = []
2   sort (x:xs) = insert x (sort xs)
-
1   insert x [] = [x]
1   insert x (y:ys)
1     | x < y     = x:y:ys
0     | otherwise = insert x ys
```

Figure 3: The program shown in Figure 1, labeled.

marked as coloured text. While this technique finds a few correct subcomputations, and hence reduces the number of questions asked by an algorithmic debugger, it does not give very much extra information. In the above example, the number of questions asked by an algorithmic debugger is cut by only one.

Finding extra correct reductions gains relatively little – each correct reduction found in the erroneous computation graph cuts the number of questions asked by at most one. Often finding a correct reduction does not cut the number of questions at all.

3 Correct Program Slices

The 'correct subcomputation' method looks only at the computation graph, and ignores a large amount of data available from looking at the program as well. The source code used in executions that ran correctly is more likely to be correct than code that has never been executed. If some slice of the code was executed 50,000 times during the computation of the correct program, then it is a good guess that this slice is correct. This heuristic allows a significant narrowing of the area of the program in which the bug is likely to be.

Each reduction in the computation arises from a program slice. If a program slice is correct, then all reductions arising from this slice must be. Algorithmic debugging is based on this property – if a reduction is erroneous, then the program slice is incorrect. Our method uses the case where a reduction is known to be correct – in this case a slice from which it aries is *likely* though not certain to be correct. This method allows the debugger to find a new set of reductions that are likely to be correct.

In the example used before (Figure 1), we can apply this process, and arrive at the result shown in Figure 3. This figure adds labels indicating the number of times each line has been executed in the correct computation. This labeling suggests a buggy line, but a debugging session is needed to confirm it. An algorithmic debugger may now order its questions differently in the hope of finding the bug faster. Instead of traversing the graph in a breadth first manner (looking only at children of erroneous nodes), the

new algorithmic debugger will look at nodes in order of their likely erroneousness (based on the likely bugginess of the part of the program being executed). If a node is found to be definitely erroneous, the system reapplies the heuristic within that node's children, rather than continuing in order of likely erroneousness. This method results in this new debugging session:

```
insert 2 [1] is [2]
> No

Bug identified in 'insert x (y:ys)':
  | otherwise = insert x ys
```

First, the most likely erroneous reduction is the computation of `insert 2 [1]` to `[2]`, as there is no record that this section of the program has ever been used to produce a correct result. Hence the first question the algorithmic debugger asks. Second, the reduction of `insert 2 []` to `[2]` is known to be correct from the technique described in Section 2, so now the bug can be identified. Compared to ordinary algorithmic debugging, there is a clear advantage in using this technique. In this example, the total number of questions asked is cut from 4 to only 1 (the normal ordering would cause the algorithmic debugger to ask 4 questions). In more complex examples the number of questions asked can be cut by an even greater factor.

An extension to this method would refine the estimates of program correctness as the debugger proceeded. To implement this method, each time the user answers the algorithmic debugger with a 'yes', the system would gain a new correct sub-program. The new correct sub-program in turn gives it a new reduction that it knows to be correct, and adds to the total information about correctly executing parts of the program.

4 Combination of Methods

Finding correct subcomputations is not very effective at cutting the number of questions asked, however it can be used to greatly improve upon finding correct program slices. In finding correct subcomputations we are able to identify several reductions that are very likely to be correct. These reductions can then be used to provide further program slices that have been executed correctly, and provide more data for our second method to work with.

5 Related Work

Delta Debugging has been developed for imperative languages by Zeller and Cleve [9]. Their approach uses comparisons of two execution states at different points in time. The approach is hard to transfer directly to functional languages as it relies on comparing program state. Delta debugging was

however the inspiration for our comparative debugger. In *Scalable Statistical Bug Isolation* [15], the authors describe a method of performing statistical analysis on multiple program runs to identify the causes of program failure. The approach looks at the computation of predicates within programs and isolates predicates which appear to be good indicators of a certain bug occurring. This in turn allows them to isolate control flow patterns that cause erroneous behaviour.

6 Discussion and Future Work

Initial Implementation The two methods described in this paper have been implemented on top of the HAT tracer. First the HAT-DETECT debugger was re-written, allowing it to work correctly with the current HAT trace file format. After this, the new HAT-DETECT was used as a basis for the HAT-DELTA debugger. Experiments comparing HAT-DELTA with the original HAT-DETECT have so far shown that the number of questions asked is reduced by a vastly varying amount depending on the program, and the executions of that program. In some experiments, the number of questions is not cut at all, while in others the number is cut by a factor of ten. So although an initial implementation has been completed, there is still significant work to do in determining the best heuristics to use in order to provide a short search path in as many cases as possible.

Combination With Other Views The algorithmic debugging process consistently finds bugs, however it can often ask large numbers of questions, or questions the user finds difficult to answer. We have presented two methods for reducing the number of questions asked by an algorithmic debugger. However, the information gathered from the traces is independent of the algorithm used to view it. The information can be combined with other views, and provide the user with more information. Olaf Chitil has described a method of improving algorithmic debugging by allowing the user to navigate freely (and thus choose the most likely position of the bug themselves) [5]. The information gathered by comparative debugging could be combined with this view to provide hints to the user about what is buggy. In a similar way, the information can be combined with an observation based view. This view mechanism presents a listing of function applications and their results. An extension could highlight the likely erroneousness of these applications.

7 Conclusion

Large amounts of extra information can be gained by examining traces of programs that evaluate correctly. This information can be used to lower the number of questions asked by an algorithmic debugger using the methods

described in this paper. The proportion of questions removed from the debugging session can be as great as 90%, but it can be as little as 0%. There have been no observations made of what conditions are needed for delta debugging using this technique to be effective.

Using Trace Data to Diagnose Non-Termination Errors

Mike Dodds & Colin Runciman, University of York

Abstract: This paper discusses BLACK-HAT and HAT-NONTERM, two tools for locating and diagnosing non-termination errors in Haskell programs. Both of these tools give a small trace which is intended to illuminate the cause of the non-termination error. BLACK-HAT analyses programs which contain *black holes*, a particularly restricted kind of non-termination error, while HAT-NONTERM applies the approach used in BLACK-HAT to more general non-terminating programs. This paper discusses the traces generated by BLACK-HAT and HAT-NONTERM, as well as the approach used to generate these traces.

1 Introduction

Non-termination errors are one of the two kinds of error that can cause Haskell programs to fail. In general, non-termination errors cannot be detected at run time, and non-terminating programs must be terminated by hand. The source of the error may also be difficult to locate and fix in a large program. Section 3 below discusses discuss HAT-NONTERM, a tool for generating traces from non-terminating programs.

Black holes are a kind of simple non-termination error that *can* be detected at run time. A black hole occurs when the process of reducing a variable[1] to its value results in an expression which contains the same variable and strictly needs its value. Black holes can be extremely simple, as can be seen from this example:

```
a = b + 2
b = a * 3
```

It should be clear why the variable `a` cannot be reduced to a final result: getting the value of `a` requires a value for `b`, which requires the value of `a`. Section 2 below discusses Black Hat, which is intended to locate and diagnose these black hole errors.

[1] that is, a value-name defined with arity zero

2 Tracing Black Holes

BLACK-HAT is a tool for analysing programs that have failed as a result of a black hole. Programs compiled with HAT support generate an Augmented Redex Trail (ART) file which describes the execution of a program. BLACK-HAT uses this ART file to locate and diagnose the causes of black holes, by giving the series of reductions which lead to the reoccurrence of a variable. To begin with a simple example, let us consider the definition of `a`, discussed above. BLACK-HAT gives us the following analysis:

```
---- black-hat: simple-bh ----
> print a
> b + fromInteger 2
> a * fromInteger 3
```

BLACK-HAT's output shows a series of expressions generated by the program. Each line holds the result of rewriting one of the previous line's subexpressions. The rewritten subexpression is highlighted in each expression. We call this series of reductions and subexpressions the *black-hole path*.

The analysis for the `ab` program shows that `a` reduces to an expression containing `b`, which then reduces to an expression containing `a` again.

For small programs the black-hole path can be constructed quite easily by hand, so let us consider slightly more complicated example. The program below is a faulty program for generating Hamming numbers:

```
ham = merge ham' ham''
ham' = map (3*) ham
ham'' = map (2*) ham
```

It should be clear that there are in fact two black-hole paths for `ham` — one from evaluating `ham'`, and one from `ham''`. Which one will result in the error depends on the order of reduction in the program. BLACK-HAT generates the following analysis when applied to this program:

```
---- black-hat: ham-bh ----
> print ham
> merge ham' ham''
> map (fromInteger 2 *) ham
```

Because BLACK-HAT works with the ART graph, which is generated from the execution of the program, it can work out which of the two potential black holes actually caused the error. The analysis makes it clear that the black hole occurs through `ham''` — through the right-hand path through the result-subexpression graph.

BLACK-HAT successfully locates black hole errors in all examples on which it has been tested. With simpler programs, such as the ones discussed above,

BLACK-HAT also makes the source of the error obvious immediately. More complicated programs make this more difficult, as it becomes harder to follow the sequence of reductions and subexpressions. Also, programs that use Haskell's list comprehension can be difficult to understand, as they are transformed during compilation.

2.1 Black Holes in the ART Graph

The ART file produced by HAT corresponds to a rooted graph with the intermediate expressions generated while executing the program as nodes. Several different kinds of edges exist between expression nodes, but BLACK-HAT uses two in particular: the result edge which point from an expression to its result, and subexpression edges, which point from an expression to its subexpressions.

An expression node in an ART files is generated when an expression is evaluated at run-time. When an expression is being evaluated, before it's result has been generated, the expression's result pointer has a value of **entered**. However, in the case of a black hole, the execution terminates because an expression has been reduced to a variable which has also been entered. This means that the result pointer for the final executed expression points back to the variable which caused the black hole, and so the graph will contain a loop in the result and sub-expression edges. This loop corresponds to the black-hole path displayed by BLACK-HAT, and it can be located by a simple search of the ART graph.

3 Tracing Non-Termination Errors

HAT-NONTERM applies the same approach as BLACK-HAT to more general non-termination problems involving functions with an arity greater than zero. That is, it generates a small trace from the ART which illuminates the source of the non-termination error.

HAT-NONTERM assumes that the execution of a non-terminating program will consist of two general phases. Firstly, a non-terminating program will execute the bug-free 'head' of the program. Then the faulty, non-terminating 'tail' of the program will execute until the program is interrupted by the user. Because a non-terminating program can be run for an arbitrary amount of time in the faulty section of the computation, the tail of the computation can always be made large relative to the head. For the same reason, in any non-terminating program, a trace can be generated where the number of function applications is much larger than the number of defined functions.

Because of these two facts, a non-termination will contain large numbers of calls to the same functions. These calls will be recursive calls originating from previous instances of this function, although the function arguments

may be different. This presents a way of displaying non-terminating functions. If the suspicious function involved in the non-termination can be identified, then the path from one instance of the function to the next can be displayed. The path between two calls to the same function may not always be the same, and so it would be a good idea to show the path between several calls to the function. We call this the *non-termination path*. For example, consider this faulty Fibonacci-number program:

```
fib 0 = 1
fib 1 = 1
fib n = fib n + fib (n-1)
```

HAT-NONTERM gives us the following analysis:

```
---- Hat Non-Term: fib-nt ----
suspicious function is fib in module Main
> print [(fib (fromInteger 5))]
> [fib (fromInteger 5)] | [False]
> [fib (fromInteger 5)] | [False] | [False]
> [fib (fromInteger 5)] + fib (fromInteger 5 - fromInteger 1)
> [fib (fromInteger 5)] | [False]
> [fib (fromInteger 5)] | [False] | [False]
> [fib (fromInteger 5)] + fib (fromInteger 5 - fromInteger 1)
```

This non-termination path correctly identifies the `fib` function as the faulty function, and displays the responsible sequence of subexpressions and results.

3.1 Non-Terminations in the ART Graph

The ART file explicitly records the fact that a program was interrupted by the user, rather than terminating normally. As with black holes, nodes that are being evaluated, or with subexpressions that are evaluating are marked **entered**. When a computation is interrupted by the user, all of the nodes that are marked **entered** are re-marked **interrupted**. There will also be a large number of other nodes with a result pointer which points through to an **entered** if the pointer is followed repeatedly. Taken together, these nodes will form a chain of expressions from the ART root to the expression which was interrupted by the user.

As was discussed above, the tail of a non-termination will consist of a large number of reductions involving a small set of functions. Some of these may be terminating functions, but unless the computation is interrupted inside one of these functions, the corresponding nodes will not be marked **interrupted**. This means that the majority of nodes which are marked

interrupted will be the functions involved with the non-termination loop. The set of suspicious functions which are involved in the non-termination can therefore be identified by the search process.

This method of selecting suspicious functions can be complicated by non-terminating programs that call expensive terminating functions. If the program is interrupted, it is likely to be interrupted during the execution of the expensive function, and so this function will be marked as **interrupted**, even though the function is not part of the non-termination path.

Once the set of suspicious functions have been identified, HAT-NONTERM selects a single function for tracing. Several heuristics are possible. The simplest is to select the interrupted function that appears last in the ART file. Another possibility is to take the interrupted function that appears most often in the ART file. Unfortunately, both of these ideas will fail with non-terminations that call expensive terminating functions. The execution of such a program is likely to be interrupted inside an instance of the expensive function. This means that the last-interrupted heuristic will incorrectly select it as the suspicious function. Similarly, if the expensive function is heavily recursive its applications may occur more often in the file than those of non-terminating recursive functions. The most-common heuristic would then also incorrectly select this function.

The solution used by HAT-NONTERM is to select the function whose applications appear furthest apart in the ART file. The functions involved in the non-termination will not only be marked **interrupted** at the point of interruption; they will also be **interrupted** all the way through the non-terminating program tail. This is in contrast to an expensive terminating function called by the non-terminating loop, which will terminate in all but the final case, and so will only be marked **interrupted** right at the end of the ART.

Once a suspicious function has been selected, HAT-NONTERM looks for a path through the ART graph which contains the required number of interrupted instances of the selected function.

3.2 Evaluating HAT-NONTERM

HAT-NONTERM generally deals well with simple non-terminations. However, it is quite easy to come up with examples where the tool's analysis is obscure or nonexistent. HAT-NONTERM suffers from the same problem as BLACK-HAT, in that the sequence of subexpressions and reductions can be difficult to follow. The tool can also have problems with functions that are passed infinite data structures — if a normal function is passed an infinite data structure, it may not terminate, but HAT-NONTERM will give no indication of why the data structure is infinite.

Another problem with HAT-NONTERM's approach is that it fails to correctly analyse non-terminating programs that generate data-structures as

they run. This is because these programs do not produce large numbers of **interrupted** nodes in the ART, as functions are constantly begin reduced to constructors. For example, consider the program:

```
repeat a = a : repeat a
```

Almost all of the instances of `repeat` will evaluate to a result where the constructor is a '`:`' node. Only the final instance of repeat will be interrupted when the program terminates. This means that HAT-NONTERM will fail to locate the non-termination in this program. One solution may be to follow the parent edges from the terminated expression up towards the ART root, and record the functions which are found along this path, but this approach has not yet been investigated.

It is interesting that HAT-NONTERM fails on 'structure-generating' non-terminating functions. These are quite different from non-terminating functions which do not generate data structures, and which therefore cause any function that evaluates them to fail. In contrast, Haskell's laziness means that structure-generating non-terminating functions can sometimes be used safely in terminating programs. For this reason, it may be inaccurate to always describe such functions as erroneous.

4 Conclusions

BLACK-HAT is generally successful in locating the causes of black holes in test programs. However, it can be difficult to interpret the trace data that it produces for complicated programs. HAT-NONTERM successfully locates and traces some simple non-terminating programs, but it is quite easy to find programs for which it generates poor or nonexistent analysis. Much more detail and analysis of BLACK-HAT and HAT-NONTERM can be found in [11], along with a number of larger examples.

A Natural Semantics for Tracing in Haskell

Yong Luo, University of Kent

Abstract: In this short paper, I present the early stage development of a natural semantics for tracing in Haskell. The semantics is deterministic and captures the essence of lazy evaluation. The output of a program in the semantics is the "same" as the output in Haskell. More precisely, if the output in the semantics is a value, then the output in Haskell is the same value. If the output in the semantics is an error message, then the output in Haskell is also an error message albeit two error messages are different. Since the semantics is deterministic, we can, in theory, always trace back to find the bugs when the result of a program is unexpected.

1 Introduction

The purpose of the paper is to give a deterministic semantics to capture the lazy computation in Haskell, and also to debug Haskell programs. The semantics have a clear and unique reduction order. The output of a program in the semantics is the "same" as the output in Haskell. More precisely, if the output in the semantics is a value, then the output in Haskell is the same value. If the output in the semantics is an error message, then the output in Haskell is also an error message albeit two error messages are different. In principle, one can follow the reductions and find bugs.

2 Basic Definitions

In this section, we define the language.

Arity

- The *arity* of a term is the maximum number of arguments that the term can have.

Patterns

- a variable is a pattern;
- $cp_1...p_n$ is a pattern if c is a constructor with arity n and $p_1, ..., p_n$ are patterns.

Terms

- a variable is a term;
- a constructor is a term;
- $\lambda x.M$ is a term if x is a variable and M is a term;
- MN is a term if M and N are terms;
- case N of $< p_1 = M_1, ..., p_n = M_n >$ is a term if N, M_1,..., M_n $(n > 0)$ are terms and p_1,..., p_n are patterns.

i.e.

$$M ::= V \mid C \mid \lambda V.M | MM \mid \text{case } M \text{ of } < P = M >$$

Weak head normal form and Normal form

The definitions of weak head normal form and normal form are slightly different from the standard ones. A weak head normal form is of the form $ca_1...a_n$ where c is a constructor. A normal form is of the form $ca_1...a_n$ where c is a constructor and $a_1...a_n$ are in normal form.

Note that $\lambda x.M$ is not a weak head normal form, nor a normal form. It is a function and must be applied. Otherwise it will reduce to an error message $error_1$. A case-expression of the form case N of $< p_1 = M_1, ..., p_n = M_n >$ is neither a weak head normal form nor a normal form.

A normal form of a term is also called the output of the term.

Reduction rules

There are two kinds of reduction rules. One is β-reduction, and the other is case-reduction. The β-reduction rule is the standard one.

$$(\beta) \qquad (\lambda x.M)N \longrightarrow M[N/x]$$

Now we analyse how to reduce a case-expression, case N of $< p_1 = M_1, ..., p_n = M_n >$. When N does not match any patterns $p_1, ..., p_{i-1}$ but matches the pattern p_i with a sequence of equations $< x_1 = b_1, ..., x_k = b_k >$, then

$$\text{case } N \text{ of } < p_1 = M_1, ..., p_n = M_n > \longrightarrow M_i[b_1/x_1, ..., b_k/x_k]$$

When N does not match any of the patterns, then it reduce to an error message $error_2$.

3 Weak head normal form and Pattern matching

This section shows how a weak head normal form of a term is derived, and how pattern-matching works. The rules for pattern matching and weak head normal form are simultaneously presented.

Weak head normal form

$$\frac{}{ca_1...a_n \Downarrow_w ca_1...a_n}$$

where c is a constructor.

$$\frac{(M[a_1/x_1, ..., a_m/x_m])a_{m+1}...a_n \Downarrow_w h}{(\lambda x_1.....\lambda x_m.M)a_1...a_m a_{m+1}...a_n \Downarrow_w h}$$

where M is not a λ-abstraction.

$$\frac{\begin{array}{c}M \text{ matches } p_i \text{ with } \overline{x = b} \\ (N_i[\overline{b/x}])a_1...a_n \Downarrow_w h\end{array}}{(\text{case } M \text{ of } \overline{p = N})a_1...a_n \Downarrow_w h}$$

where the sentence, M matches p_i with $\overline{x = b}$, means that M does not match $p_1, ..., p_{i-1}$ but matches the pattern p_i with a sequence of equations $< x_1 = b_1, ..., x_k = b_k >$

$$\frac{M \text{ doesn't match any } p_i}{(\text{case } M \text{ of } \overline{p = N})a_1...a_n \Downarrow_w error_2}$$

Pattern Matching

$$\frac{p = x}{M \text{ matches } p \text{ with } < x = M >}$$

This rule says that if p is a variable, then M matches p with $< p = M >$.

$$\frac{M \Downarrow_w error_2 \quad p \neq x}{M \text{ doesn't match } p}$$

where $p \neq x$ means p is not a variable.

$$\frac{\begin{array}{c}M \Downarrow_w ca_1...a_n \quad p = cq_1...q_n \\ a_1, ..., a_{i-1} \text{ matches } q_1, ..., q_{i-1} \text{ with } l_1, ..., l_{i-1} \\ a_i \text{ doesn't match } q_i\end{array}}{M \text{ doesn't match } p}$$

where l_j is a sequence of equations.

$$\frac{M \Downarrow_w ca_1...a_n \quad p = c'q_1...q_n \quad c \neq c'}{M \text{ doesn't match } p}$$

$$\frac{\begin{array}{c} M \Downarrow_w ca_1...a_n \quad p = cq_1...q_n \\ \overline{a} \text{ match } \overline{q} \text{ with } \overline{l} \end{array}}{M \text{ matches } p \text{ with } \bigcup \overline{l}}$$

where l_j is a sequence of equations. Note that when $n = 0$, $\bigcup \overline{l} = []$.

4 Normal form

This section contains the rules which show how the final result of a term is derived, in particular, how error messages are handled.

$$\frac{arity(c) = n \quad a_1 \Downarrow v_1, ..., a_n \Downarrow v_n \quad \overline{v} \neq error}{ca_1...a_n \Downarrow cv_1...v_n}$$

where c is a constructor and $\overline{v} \neq error$ means none of the values $v_i$is an error.

$$\frac{(M[a_1/x_1, ..., a_m/x_m])a_{m+1}...a_n \Downarrow v}{(\lambda x_1.....\lambda x_m.M)a_1...a_m a_{m+1}...a_n \Downarrow v}$$

where M is not a λ-abstraction.

$$\frac{\begin{array}{c} M \text{ matches } p_i \text{ with } \overline{x = b} \\ (N_i[\overline{b/x}])a_1...a_n \Downarrow v \end{array}}{(\text{case } M \text{ of } \overline{p = N})a_1...a_n \Downarrow v}$$

Error handling

There are two types of error messages; one is for the terms which are not fully applied, and another is for the failure of pattern matching.

$$\frac{}{\overline{\lambda x.M \Downarrow error_1}}$$

$$\frac{arity(c) > n}{ca_1...a_n \Downarrow error_1}$$

Note that when $arity(c) < n$, type-checking will detect the error.

$$\frac{\begin{array}{c} arity(c) = n \\ a_1 \Downarrow v_1, ..., a_{i-1} \Downarrow v_{i-1}, \ a_i \Downarrow error_j \quad v_1, ..., v_{i-1} \neq error \end{array}}{ca_1...a_n \Downarrow error_j}$$

where $j = 1, 2$.

$$\frac{M \text{ doesn't match any } p_i}{(\text{case } M \text{ of } \overline{p = N})a_1...a_n \Downarrow error_2}$$

5 Conclusion

This paper has presented a deterministic semantics for simple functional programs with lazy evaluation. The result of a program according to the semantics agrees with results obtained from the corresponding program in Haskell. In principle, one could follow the reduction sequence derived from the semantics to investigate the cause when a result is not expected. The semantic rules sketched here also represent a first step towards the development of a semantic model for a simplified version of Hat traces.

Visual Hat

Neil Mitchell, University of York

Abstract: This paper describes a new approach to visualizing the data contained in Hat traces. The aim is to cater for Windows users who are more familiar with graphical debugging tools.

1 Introduction and Motivation

My motivation with respect to Hat is very different from that of the other contributors. I am not a Hat developer, nor even a regular Hat user, but would very much like some additional help with debugging my Haskell code. I am exclusively a Windows user, and as a fan of visual interaction tools I use a console only rarely. The best debugger I have *ever* used is that which comes with Microsoft Visual Studio.

My typical Hat interaction goes along the following lines:
Compile with Hat Often the program needs modification before it will compile for tracing. Typically numeric literals need to be explicitly typed.
Create the trace This is usually pain free. I do not usually have a problem with 50 times slower execution (although faster would be better).
View the trace This is the stage at which I usually fail. A combination of interfaces I am not experienced with, along with a disconcerting text-mode, make it hard for me to extract the data that I know is in the trace. Having many tools to use makes it harder to know where to start.

The issues of compiling with Hat and creating the trace are being addressed by Tom Shackell. This paper is concerned with the final issue of viewing the trace, and argues for a radical direction change. What I propose will probably appeal more to Haskell beginners and less to experienced programmers.

While other people have developed graphical tools for Haskell development, most have focused on an environment for source code editing and compilation [2], or on graphical programming [12] – with little reference to debugging.

2 Graphical User Interfaces

2.1 General Requirements

The new tool should have a proper native Windows GUI that follows the standard conventions of Windows as much as possible. All modern graphical environments provide standard user interface elements such as scroll bars, and emphasize the mouse in favour of the keyboard. A console is not a suitable user interface.

The interface should allow the user to explore the trace in many ways. Anything that is visible and can sensibly be clicked should be a link. It should be possible for the user to go forward and backward through their interaction, without losing information they may have once found.

2.2 Graphical User Interfaces for Existing Tools

The tool I propose has two goals: to introduce a new way of working with the trace, and to use a proper graphical interface. While the new way of viewing the trace is not appropriate for a console tool, the existing console tools could have a graphical interface added to them.

The existing tools already have interface concepts that exist in GUI toolkits. For example, the screen is split into various panels which can be scrolled, and it is possible to cycle through options with the arrow keys. These features could be captured in a GUI providing an alternative interface on the existing tools.

It is possible to use Haskell to write a GUI for these elements. Unfortunately a compiled binary for Windows for Hello World in a GUI toolkit such as wxHaskell [14] is 8Mb. At this size, it is probably unreasonable to supply a toolset consisting of 6 separate viewers.

3 Concept and Examples

A functional program is all about an expression which evaluates to a result. In most programs this expression is `main` and the result is `{IO}`. The interface of the tool will show redexes at the top and values at the bottom. The user can click on either the top or the bottom to see more detail about a particular expression.

When clicking on a redex such as `main`, this might expand to `putStrLn "Hello World!"`. Clicking on a value such as `"Hello World!"` might give the expression `"Hello " ++ "World!"`. Every level of expression can be selected and viewed in more detail.

3.1 Values

Examples of values are:

- `"hello"`
- `['h','e','l','l','o']`
- `('h':'e':'l':'l':'o':[])`
- `{IO}`
- `Just True`

If a value is currently shown in the top part of the display, and it is clicked, then that value moves to the bottom and the top displays the path by which that value was derived.

If syntactic sugar has been applied to a value, then clicking it will remove the sugar. For example, `"ab"` goes to `['a','b']`, then on to `('a':'b':[])`. If a part of a value is clicked, for example if `True` is clicked within `Just True`, then the `True` is put to the bottom and the source of this particular True is placed at the top.

`{IO}` is a special type of value, and the behaviour on selecting it has not been considered in detail. One possible option would be to decompose the value into the actions that occurred.

3.2 Redexes

Some examples of redexes are:

- `main`
- `map (+1) [1,2,3]`

When these are clicked, a single reduction is performed. For example `main` might be reduced to `putStrLn "Hello World!"`. The second example would be reduced to `(+1) 1 :  map (+1) [2,3]`.

3.3 Collapsing

When a value is too large to fit on the screen, ellipses can be used as an indicator. By clicking the ellipses, further information will be displayed.

3.4 Selecting Using a Mouse

One of the hardest operations with the existing tools is selecting a sub-expression. The current solution is to use the left and right keys to cycle through the possible options. Unfortunately, these options are presented in an order which some users find unintuitive.

One possible solution is the use of underlines. Consider the list `(1:2:[])`

$$\underline{1} : \underline{2} : \underline{[]}$$

The underlines clearly delimit the different sub-expressions. Admittedly, a large number of underlines might confuse the user, and some constraints may be needed to limit the number that appear.

4 Feedback from Hat Day

It was mentioned that the original redex-trail browser (a predecessor to Hat) had several of the design features proposed here. In an attempt to learn from the mistakes of the past, some of the flaws that were mentioned as having caused problems in the previous GUI tool were:

Java The original tool was written in Java [22], which at the time was not a mature programming language for GUI programming. Fortunately since this time GUI programming languages have matured as GUIs have become more and more common. In particular, Java now has several mature GUI frameworks including Swing. C# is also available, which has major commercial backing along with friendly development tools, and has turned creating a GUI into an easy task.

Complexity The original tool had many features, leaving the users confused. By moving to more individual tools that were more tightly focused it was felt that the user could regain control. Certainly complexity is an issue that will have to be kept in mind while designing the tool, preferring fewer more easily accessible features over a complicated user interface.

GUI Not everyone is convinced that graphical interfaces are a good interaction method for debugging. I believe graphical interfaces can be highly beneficial, having used several good visual debuggers. Only time will tell if this method suits Haskell.

5 Conclusion

It is hard to determine whether a tool of the kind proposed here would be useful in practice without a working implementation.

There has been some progress on writing the tool described in this paper. An initial implementation in C# has been started, but has not been finished. The main disadvantage of the current implementation is that it does not reuse the existing Haskell and C routines for handling Hat traces.

Deriving Program Coverage from Hat Traces

Colin Runciman, University of York

Abstract: This note describes a small addition to the HAT toolkit, a program called HAT-COVER which reports the source-program coverage of a run recorded in a HAT trace.

1 Introduction

There were two reasons for writing HAT-COVER. First, it meets a practical need. Surprisingly, there is no currently distributed tool that reports source coverage for test runs of Haskell programs. The most widely used tool for testing Haskell programs is QuickCheck, and "the major limitation of QuickCheck is that there is no measurement of test coverage" [7]. As a HAT trace is a source-linked record of all the evaluation that occurs in a program run, it contains information about which parts of the program have been used. Although QuickCheck and HAT have been combined before [8], tracing was only used to investigate *failing* test-cases. If instead we trace the execution of *all* test cases, we should be able to derive information from the trace about which parts of the program have been covered by the tests.

The second reason for writing HAT-COVER was to investigate, by a small case-study, the claim that a HAT trace can readily be put to uses other than those for which it was originally designed. Part of the mantra for HAT is "multiple views" [24], and one way of thinking about HAT-COVER is that it gives another view, albeit a rather restricted one, of the trace information. The HAT-COVER program ought to be straightforward to write. Its intended output is just a source listing highlighting parts of the traced program actually used. But do the structure of the trace, and the library of basic routines for accessing traces, allow a short simple program to be written? And can a simply designed program also be efficient enough to process even large traces rapidly and without excessive demands for heap memory?

Section 2 describes HAT-COVER from a user's perspective. Section 3 explains how the program works. Section 4 evaluates its effectiveness. Section 5 concludes.

2 Using HAT-COVER

This section describes HAT-COVER from a user's perspective. It is based on the HAT-COVER manual page, with the addition of an example.

Synopsis

`$ hat-cover` *option... tracefile sourcefile...*
Following usual HAT convention, a *tracefile* may be specified with or without a `.hat` extension, and each *sourcefile* may be specified with or without a `.hs` extension.

Description

HAT-COVER prints to standard output a listing of each Haskell *sourcefile*, highlighting expressions for which at least one source reference occurs in the HAT *tracefile*. If no *sourcefile* is specified then HAT-COVER lists *all* sources in which there are traced expressions.

The highlighted expressions in the output are *maximal* expressions to which there are source references in *tracefile*. However, case-expressions and conditionals are deliberately *not* treated as candidates for highlighting, and there is no trace representation for entire let-expressions (or right-hand sides including a where-clause). These exclusions ensure that HAT-COVER decides separately whether to highlight each alternative, branch or local-definition body — *unless* the case-expression, conditional or let-expression to which it belongs is a subexpression of a larger expression that is a candidate for highlighting.

In general, one cannot infer from the highlighting of an expression that every subexpression within it was needed at some point in the traced computation. Extra detail in coverage information can be obtained if necessary by lifting out key subexpressions as the right-hand-sides of local definitions.

Options

`-hion=`*onchars*, `-hioff=`*offchars*
These options reset the character-sequence codes used to switch highlighting on and off in the output.

Bugs and Limitations

Some uses of variables as expressions are not available as candidates for highlighting — though any entire right-hand-side, or any alternative, branch or let-body is always a candidate. Some trailing brackets in expressions are not included in highlighting. These problems are consequences of unavailable or inaccurate information in current HAT traces.

Example

Figure 1 shows an example of a listing produced by HAT-COVER. The listed program tests whether a proposition is a tautology, and it works by interleaving simplification and case analysis of the left-most variable. Notice some parts of the program that are *not* covered: the equations defining `eval`, `varOf` and `subst` for negated arguments are never used. Even though `eg` contains three implications, and in one alternative `eval` simplifies implications to negations, that alternative is itself never used.

3 How HAT-COVER works

Any expression in a program can be specified by the source-file it belongs to and the character positions (line and column) where it begins and ends. This is exactly the information encoded in each source-reference node of a HAT trace. The essence of what HAT-COVER needs to compute is the *cover set* of maximal source references attached to traced expressions, restricted to specified modules and excluding case-expressions and conditionals. The relevant modules can then be listed, with highlighting controlled by the cover set.

After HAT-COVER has interpreted its arguments to obtain `moduleNames`, and performed `openHatFile`, it computes the cover set by a straightforward linear scan inspecting every trace-node and accumulating relevant source references:

```
do
  nodes <- nodeSequence
  let srs =
    [ sr |
    (fn, nt) <- nodes, isCover nt,
    let srn = getSrcRef fn, srn /= Filenode 0,
    let sr = readSrcRef srn, line sr /= 0,
    null moduleNames || filename sr `elem` moduleNames ]
  printCover (hiOn, hiOff) (addAll srs [])
```

All kinds of applications, variables and constants explicit in the source satisfy `isCover`, but case-expressions and conditionals do not.

The application `addAll srs []` inserts successive source references into an initially empty cover set. The set is a forest of intervals of line-column pairs, with a simple binary search-tree for each source file.

```
type Cover      = [(String, Tree (Interval LineColumn))]
data Tree       = Leaf | Fork (Tree a) a (Tree)
type Interval a = (a,a)
type LineColumn = (Int,Int)
```

It is tempting to define `addAll srs cvr = foldr add cvr srs` and to define `add` in the most obvious way using explicit list- and tree-recursion.

```
data Prop = Lit Bool | Var Char | Not Prop | Prop :=> Prop
infixr :=>

eg = Var 'p' :=> (Var 'p' :=> Var 'q') :=> Var 'q'

eval :: Prop -> Prop
eval (Lit b)   = Lit b
eval (Var v)   = Var v
eval (Not p)   = case eval p of
                   Lit b -> Lit (not b)
                   p'    -> Not p'
eval (p :=> q) = case (eval p, eval q) of
                   (Lit b, q') -> if b then q'
                                       else Lit True
                   (p', Lit b) -> if b then Lit True
                                       else Not p'
                   (p',    q') -> p' :=> q'

varOf :: Prop -> Char
varOf (Var v)   = v
varOf (Not p)   = varOf p
varOf (p :=> _) = varOf p

subst :: Char -> Bool -> Prop -> Prop
subst _ _ (Lit b)   = Lit b
subst v b (Var w)   = if v==w then Lit b
                      else Var w
subst v b (Not p)   = Not (subst v b p)
subst v b (p :=> q) = subst v b p :=> subst v b q

taut :: Prop -> Bool
taut p  = case eval p of
          Lit b -> b
          p'    -> let v = varOf p' in
                   taut (subst v True  p') &&
                   taut (subst v False p')

main = print (taut eg)
```

Figure 1: An example of HAT-COVER output.

But if `addAll` is so-defined HAT-COVER takes ages to run for all but small traces and it exhausts memory if applied to large traces. For stack economy *tail recursion* is essential, and for heap economy the forest must be kept in *normal form* to avoid chains of suspended updates.

```
addAll srs cvr | normal cvr =
  case srs of
  [] -> cvr
  (sr:srs') -> addAll srs' (add sr cvr)
```

Use of a non-recursive `normal` predicate as a guard, and *normalising constructors* in a function like `add` is a more flexible and earlier alternative to strictness annotations [20].

4 Evaluation and Future Work

The opening motivation suggests two questions: Is HAT-COVER concise and efficient? Does it provide the coverage information needed?

Conciseness and Efficiency

The current HAT-COVER program is about 200 lines of Haskell — in addition to a few pre-existing routines from the `LowLevel` HAT library and the standard Haskell libraries. Despite an unsophisticated data representation, it handles even quite big traces with tolerable efficiency. For example, running the MATE chess-end-game solver on a tough two-move problem generates a 235 MB binary trace file, with over 15 million nodes in the trace graph; HAT-COVER computes a coverage listing for the `Move` module (the largest and very heavily used) in about 3 minutes on a slowish PC.

If one wanted to make HAT-COVER more efficient, apart from rewriting it in C, options include:

Use more sophisticated data structures for a finite map from strings to ordered sets — eg. balanced trees of some kind [1]. *Cautions*: typically there are very few source modules so fancy finite maps may not repay their overheads; inserting an interval that encloses one already in the set requires pruning and grafting yet must preserve the balance invariant.

Use integer codes instead of full strings and source-reference information — for faster comparisons. *Caution*: although addresses of module nodes and source-reference nodes in the trace provide a natural integer reference, they do not directly support appropriate ordering and inclusion tests.

Guarantee once-only insertion of references in cover sets by setting *mark bits* directly in the trace file. *Cautions*: this cannot be done using only routines currently in the HAT library; also, the I/O costs might be greater than the savings in cover-set processing.

Level of Information

The coverage information that HAT-COVER provides could fairly be described as rudimentary. In the trade-off between the refinement of results and the complexity and cost of computing them, simpler options have almost always been preferred.

The slicing method of HAT-EXPLORE [5] can give a more refined view of needed parts, for example distinguishing between evaluated and unevaluated arguments at the point of call. But slicing may be too expensive as a routine method for deriving coverage from large traces. A HAT-COVER user can force finer distinctions by making local definitions for critical expressions.

If one wanted to make HAT-COVER more informative, without abandoning the simple linear-scan approach, options include:

Derive a summary measure of coverage in each module — ie. 100% coverage of module A but only 60% of module B. The measure could be based on comparison with an upper bound for the cover set of a module determined directly from the source. (Remember that "measurement of test coverage" is what the QuickCheck authors really wanted.)

Distinguish different frequencies of coverage, generalising a single form of highlighting to a range of intensities or colours indicating the relative frequency of trace references to different expressions. Computing multisets rather than sets is easy; the trickier part may be determining the most helpful scale of values.

5 Conclusions

It is rather straightforward to extract basic coverage information from HAT traces. The exercise can be completed using only a few already available routines to access the trace from a short Haskell program. An unsophisticated prototype, based on a linear scan of the trace to extract source references, is in most respects quite workable; but it is essential to define some parts of the program carefully to avoid excessive memory demands. There is scope for further refinement of the linear-scan approach to improve both efficiency and the quality of information obtained.

HAT-COVER as described in this note is available as part of HAT 2.04.

The Hat G-Machine

Tom Shackell & Colin Runciman, University of York

Abstract: The HAT system provides a method for tracing a lazy functional program. However HAT works by transforming the source program into a much larger self-tracing variant that runs between 15 and 100 times slower and uses several times as much memory. We show how equivalent traces can be generated much more efficiently by modifying the underlying abstract machine. Our approach shows that it is only necessary to add a few extra machine instructions and change the interpretation of a few others in order to generate HAT traces efficiently.

1 Introduction

It is often said that an advantage of functional languages is that they make it more likely that programmers will write correct code. However, even functional programs sometimes have bugs. In order to correct them it is often necessary to know why the program gave a particular answer. It can also be useful to understand why a program gave a particular answer for reasons other than debugging, for example, in order to gain a better understanding of the program. This can be particularly important for people modifying code that they did not write.

One method to debug and understand a program is to produce a trace of everything that the program did. Such an approach is used in systems such as HAT [24] as well as in algorithmic debugging systems such as Freja [16] and Buddha [18]. This paper will look principally at the HAT system.

Hat produces traces in way that is portable across different compilers, but the traced computation uses much more memory and typically runs between 15 and 100 times slower than the untraced case. This paper describes how the same traces can be produced much more efficiently by modifying the underlying abstract machine.

1.1 The difficulties of tracing lazy functional languages

The method traditionally used for tracing and debugging in strict functional and imperative languages is to step through the execution sequentially and

examine the runtime state of the system, such as the contents of the heap and values of variables on the stack [23]. Such an approach is less effective for lazy functional languages, however, as the order of evaluation is demand driven and so is often un-intuitive to the programmer. In a lazy language the execution of a piece of work is suspended until its evaluation is necessary in forming the output of the program.

Producing a complete trace of the execution of the program means that debugging becomes a post-mortem activity. The advantage of this approach is that it allows the execution to be explored independently of evaluation order. It also means that the runtime state of the program cannot be altered during the tracing process, which is in keeping with the declarative nature of functional programs.

HAT works by using source level transformation: the original Haskell program is translated into a new Haskell program. When this new program is evaluated it does the same computation as the original but as a side effect also produces a trace of the evaluation. Such an approach has the advantage of being portable across different compilers and platforms. However, it also makes the source code several times larger, increases compile times, can greatly increase the memory usage and can greatly reduce the runtime performance of the program. For some large applications such a performance loss makes tracing impractical.

1.2 Tracing using a modified abstract machine

An alternative approach to transforming the program source is to use a modified compiler and abstract machine. The program is compiled with additional debugging information and then as execution proceeds the abstract machine generates a trace of the computation. Such an approach is not portable across different compilers but it has the advantage that it has much less interpretive overhead when compared with tracing by program transformation.

2 The G-Machine

The G-Machine is an abstract machine for implementing lazy functional languages by graph rewriting and is described by Augustsson and Johnsson in [13]. Suspended computations are represented in the heap as graphs. When the value of the suspended computation becomes necessary the function described by the graph is executed and the graph is rewritten with its result. This process continues until the final result of the computation has been obtained.

Results reported in this paper concern a modified version of the nhc compiler [19], which is implemented based on the Spineless G-Machine [4].

Functions are compiled as supercombinators, which are top level function definitions with no free variables. The definition for a function is compiled into a series of SG-Machine instructions. When a closure is evaluated the SG-code builds a graph representing the body of the function in the heap and then returns and updates the closure.

3 The Hat trace structure

The HAT trace is a graph that provides a detailed record of the graph reduction process when a program is executed. A trace can be viewed by the Hat tools to examine what the computation did [8]. The trace principally records applications of functions to particular arguments. The trace also records the parent application, in whose body this one was created, and what the result of this application is.

4 Modifying the SG-Machine for tracing

The correspondence between the HAT traces and the graph rewriting semantics strongly suggests that a HAT trace could be generated by a modified SG-Machine. The basic idea is that when new nodes are built in the heap a corresponding node is built in the trace. Similarly whenever a heap nodes is updated with a result, the corresponding trace node is updated with the corresponding result. For updates to be possible every heap node needs to know what its corresponding trace node is; this is achieved by adding a field for the file position of the trace node in the heap node.

4.1 The trace stack

Although the trace of a function closely matches its representation in the heap there are things recorded in the trace which have no representation in the heap, for example case expressions. In order to accommodate these features an extra stack is added to the abstract machine, the trace stack.

5 Results

The back end of the nhc compiler has been modified to generate the new instructions and to add extra information such as function names and source code positions.

The performances of Untraced nhc, HAT G-Machine and HAT-TRANS are compared using the following programs from the no-fib benchmark suite [2].

[2] http://www.dcs.gla.ac.uk/fp/software/ghc/nofib.html

Table 1: Performance results

Program	Compile-time measures		Run-time measures		
	Object Code(KB)	Time(s)	Mem(KB)	Time(s)	GCs
Primes-NT	282	0.47	3	0.88	121
Primes-HGM	709	0.53	6	1.91	220
Primes-HT	1753	10.43	20	66.92	9065
Queens-NT	207	0.49	8	0.17	27
Queens-HGM	488	0.55	14	0.61	48
Queens-HT	1755	10.80	103	8.86	1147
WheelSieve-NT	283	0.54	7	0.16	22
WheelSieve-HGM	713	0.66	13	0.47	41
WheelSieve-HT	1764	11.20	23	9.30	1234

Queens	A program to solve the n-queens puzzle (n=8).
Wheel1	An efficient program to list the first n primes (n=1000).
Primes	A naïve program to find the n^{th} prime (n=500).

5.1 Compile-time results

The compile-time results are shown in the first part of Table 1. Results are shown for two indicators: the size of the final binary and the time to compile the program.

Table 1 shows that the object code size for the HAT G-Machine is always within a factor of 3 of the untraced case, which is substantially smaller than the code generated by HAT-TRANS.

5.2 Run-time results

Run-time results are shown in the second part of Table 1. The results are shown for a number of key performance indicators.

Mem	peak live heap memory after a garbage collection
Time	complete program runtime in seconds
GCs	number of garbage collections

The memory usage of HAT G-Machine programs is always within a factor of two of the untraced program; in comparison programs generated by HAT-TRANS use between 3 and 12 times the memory needed in the untraced case. In terms of run-time performance the HAT G-Machine is between a factor of two and four slower than the untraced program; programs generated by HAT-TRANS typically being around a factor of 50 slower.

6 Related work

The most closely related work is that of Henrik Nilsson in his Freja system [16, 17]. Like the HAT G-Machine, Freja generates a trace of a lazy functional language by modifying the abstract machine. However there are a number of key differences: Freja generates simpler EDT structures rather than the richer HAT trace structure; Freja generates the traces 'piecemeal' in memory rather than in a file as HAT does; Freja is based on the standard G-Machine rather than the spineless G-Machine of nhc.

Many of the same ideas are present. For example Nilsson also augments the heap nodes with additional information, although the information Freja stores is quite different.

7 Conclusions and further work

7.1 Conclusions

The work on the HAT G-Machine has shown that by making minor changes to the SG-Machine code generated, and by extending the interpretation of SG-Machine code, it is indeed possible to generate HAT traces. With some additional effort this scheme can be extended to deal with the computations over trusted functions.

The expected efficiency gains from tracing at the abstract machine level are indeed realised. The HAT G-Machine produces traces more than an order of magnitude faster than corresponding programs generated by HAT-TRANS, with only a 2 to 4 fold increase in execution time compared to untraced code. It also substantially reduces the memory consumption, compile time and object code size when compared to the HAT-TRANS approach.

7.2 Further work

This paper is an abbreviated version of the paper [21]. The full paper gives examples of SG-Machine instructions, the exact structure of the HAT trace, more motivating examples as well as the full operational semantics of the new and modified instructions.

The unwanted differences between the traces produced by HAT-TRANS and those produced by the HAT G-Machine need to be resolved. It will be necessary to characterise more precisely the expected correspondence between the traces generated by the two methods. The traces generated by the HAT G-Machine have not yet been checked extensively against the full range of viewing tools.

In due course it is our aim to merge the HAT G-Machine work into the released nhc98 system.

Story and History — a Value-Based View of Hat Traces

Malcolm Wallace, University of York

Abstract: Most views of HAT traces are either expression- or reduction-oriented. They show (either forwards or backwards) how computation proceeded, and which computations depended on which other computations. This paper suggests a complementary view, fixing on a particular value, and examining what happens to it throughout its lifetime. This view will turn out to be particularly useful when tracing 'difficult' or 'para-functional' features of a program, such as I/O and concurrency.

1 Introduction

The HAT system provides a number of different views of the trace of a Haskell program [24]. For instance, HAT-OBSERVE shows function applications and their ultimate results; HAT-TRAIL allows backward exploration through the reduction graph of expressions; HAT-DETECT performs algorithmic debugging by traversing the evaluation dependency tree of an expression; HAT-ANIM animates the forward sequence of the reduction of an expression [10].

All of these views are based around expressions and reductions. They follow the flow of the computation, in some sense concentrating on the *arrows* of the graph, and where they take us – the viewer's focus moves around the trace graph. This paper suggests an alternative view, based more on values. Values in a pure functional language are immutable – fixed – so in that sense, the camera is fixed on a single *node* of the graph, and what is of interest in the movie is what happens to that node over time — its creation, the attachment of dependency arrows to it, and its eventual quiescence when it no longer plays a part in the computation.

2 The Idea

Consider this trivial program:

```
main          = print (select (simple 3))
```

```
select (_:x:_) = x
simple n       = (n-1) : n : (n+1) : []
```

The requirement is to view the sequence of all activity involving the particular value `3`. Now, because here the program does in fact print `3` as its output, one might expect HAT-TRAIL to give us exactly what we want, albeit backwards:

```
Output: ---------------------------------------------------
3
Trail: ----------------------------------------------------
<-  print 3 = IO
<-  select [_,3,_] = 3
<-  fromInteger 3 = 3
<-  main = IO
```

But in fact, there is quite a lot of information missing here, and some of the apparent connections between lines may be less than they seem. The actual backward sequence (or *history*) we would like to have found is as follows:

1. argument of: `print`
2. result of: `select`
3. pattern-match in: `select`
4. argument of: `cons (:)`
5. argument of: `simple`
6. parented by: `main`

HAT-TRAIL has only been able to reveal (for definite) the first, third, and sixth items. More exploration would be required to confirm the second and fourth items, and the fifth item cannot be found directly at all! (This last problem is due to a projection node in the trace.)

If we think about the forward sequence (or *story*), it is not just the linear reversal of the history, but has a branching structure. In the reverse of the sequence above, after becoming the argument to `simple`, the value `3` then has three independent things happen to it. In addition to becoming an argument of a cons, it also becomes an argument in the two expressions `(3-1)` and `(3+1)`, and in each of these arithmetic branches of the story, the value `3` ceases to take any further part. Once again, HAT-TRAIL does not easily reveal this information.

Other viewers, e.g. HAT-ANIM and HAT-OBSERVE, are no better at gathering the story/history information quickly and coherently. In principle, all the information is already available in the trace, but it currently requires considerable skill to use the trace-viewing tools to recover it.

So, I propose a new viewer, based on the central concept of a value and its story/history.

The activities noted in the trace that are potentially of interest within a story/history are:

- parented by: the creation of a value
- argument of: the value is passed to a function
- applied to: the value is a function (possibly anonymous), and is applied to some argument(s) with some result
- reduced to/from: the value undergoes some evaluation
- pattern-match in: the value is either decomposed, or bound to a variable, within a pattern-match arising on the LHS of a function, a lambda binding, or a case expression
- result of: the value is returned as the result of a function – either as a projection, or subsequent to being 'parented by' that function.

3 Motivating Examples

Here are two motivating examples to show that value-based views can provide interesting and useful information.

3.1 I/O

Generally speaking, I/O is difficult to trace. The result of either an output or input action is itself rather uninteresting – usually just a placeholder indicating an opaque monadic value. In order to be of any use at all, HAT currently records all character output separately from the trace proper, with a link backward from each character to the traced I/O action that produced it. Input characters *can* be found within the trace, although not directly as the result of the I/O action, and only if the input is in fact used by the computation. They appear as the arguments of some other functions, subsequent to their retrieval by the I/O action, and can be identified only because their parent redex is a redex whose result included the I/O action – a complicated indirection forced by the monadic nature of I/O.

There are several unfortunate consequences of this model. If an I/O action produces no visible output, e.g. `putStr ""`, it can be almost impossible to navigate to it within a viewer. Even worse, the action may have a side effect on the environment, yet produce no discernible input or output, e.g. `hSetBuffering NoBuffering`.

But value-based tracing provides a solution! A large number of I/O actions either use or return an opaque handle on the external resource being manipulated.

For instance, opening a file or socket creates a handle, reading from or writing to the resource is mediated by the handle, and often the handle is closed explicitly too. By examining the story of the handle itself, we are able to see all of these events in sequence, including any empty writes, or side-effecting actions like changing the buffer size.

I/O handles are not only used to mediate external resources – they are also commonly used for internal resources, in the form of global variables. An `IORef` (or `STRef`) is, for our purposes, exactly a handle on a mutable memory location, whose story and history can be examined, to discover the value stored there at different times. Likewise for mutable arrays, and any other mutable data structure whose access pattern must be constrained through references and a monad.

As a special case of I/O, concurrent programs tend to have bugs that are unique to the non-sequential nature of the paradigm, e.g. deadlock and livelock. These arise through faults in locking and signalling protocols. Until now, Hat has had no facilities to address finding such faults (see [3]). But an `MVar`, the basic unit for concurrent communication in Haskell, is once again just an opaque handle on a resource, and again, its story (perhaps interleaved with the story of other `MVar`s) can perhaps usefully reveal the behaviour of the protocol, illuminating the causes of concurrency problems.

3.2 Compilers

The following example has a rather different flavour. Consider a compiler, built in the traditional way with multiple passes. To better understand how the compiler works, a student may want to trace the compiler running over a small program, and for example, to watch the symbol table as entries are added to it during the identification phase, then as those entries are updated later during type checking. Or perhaps they want to see how the original text is lexed into chunks, parsed into an AST, then how some fragment of the AST is changed by various translation and optimisation phases, before finally being flattened to assembly or binary output. In each case, the objective is to follow the story of what happens to a value. However, unlike the I/O scenario, here the focus jumps from one immutable value to a different immutable value as the computation progresses, transforming one data structure into another.

As a smaller illustration of the same idea, consider the following trivial program:

```
main = print (jam 3)
jam x = filter (== x) [1,2,3,4,5]
```

Once again, we are interested in following what happens to the value `3` from the first line. Its full story is this:

1. parented by: `main`
2. argument of: `jam`
3. argument of: `(==)`

Intuitively, we know that `3` is used many more times within the `filter`. What has happened is that the value has become fused inside another value (a partial application of the equality operator), and it is the larger value we now want to follow. So, a sideways step is required from viewing the story of `3` to viewing the story of `(==3)`, which is this:

1. parented by: `jam`
2. argument of: `filter`
3. applied to : `1` (with result `False`)
4. applied to : `2` (with result `False`)
5. applied to : `3` (with result `True`)
6. applied to : `4` (with result `False`)
7. applied to : `5` (with result `False`)

4 Conclusions

There is considerable scope for new views of the Hat trace, based solely on the information already available, but taking a different approach to the existing browsers. I have outlined one possible new view, where the life story of a value is of central interest, as opposed to the flow of reduction common to most current views.

Many issues remain open for discussion:

- Nodes in the trace have identity, unlike computational values. That is, there can be multiple trace nodes denoting the same value, yet these nodes will be treated as separate values for the purposes of the viewer. This abandonment of referential transparency can be justified. Each node in the trace file corresponds to a unique parentage, and therefore a different lifestory. Although two values may be the same, their life stories throughout the computation can be different, and so it would be confusing to interleave their timelines arbitrarily, as if they were one. It also more closely reflects the lazy sharing model of the underlying implementation.

- No thought has yet been given to a user interface for such a viewer. To what extent should it use expressions, and to what extent descriptions (e.g. "argument of")? How will a branching story be represented to the user? Is it better to tell a story or a history, or some combination of both?

- As in the compiler example, it may turn out that following the story of a single value from start to finish is uninteresting. It will probably be useful to have the ability to jump from one value's story to another, following the value's transformation from argument to result of a function.

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).