Source-Based Trace Exploration
— Work in Progress —

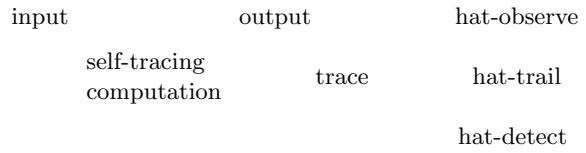
Olaf Chitil

University of Kent, UK

Abstract. HAT is a programmer's tool for generating a trace of a computation of a Haskell 98 program and viewing such a trace in various different ways. Applications include program comprehension and debugging. The trace viewing tools show expressions and equations of a computation, but they hardly refer to the source program. This disregard of the program is odd, because the computation follows from the program and the usually familiar source program can help orientation in a complex computation. Hence I started the development of new trace viewing tools that are based on showing the source program with various changing markings.

1 Introduction

A tracer gives the programmer access to otherwise invisible information about a computation. It is a tool for understanding how a program works and for locating errors in a program. HAT is a tracer for the lazy functional language Haskell 98, that combines the tracing methods of several preceding systems [6, 2, 3]. Tracing a computation with HAT consists of two phases, trace generation and trace viewing:



First, a special version of the program runs. In addition to its normal input/output behaviour it writes a trace into a file. The trace as concrete data structure liberates the views from the time arrow of the computation. A trace is a complex graph of expression components [6]. Second, after the program has terminated, the programmer studies the trace with a collection of viewing tools.

- HAT-DETECT provides algorithmic debugging, that is, semi-automatic localisation of program faults. Trace viewing consists of the system asking questions about the computation such as "Should `factorial 3 = 42`?" which the programmer has to answer with "yes" or "no". After a series of questions and answers the debugger gives the location of a fault in the program.

– HAT-TRAIL enables the programmer to follow redex trails; the programmer explores a computation backwards, from an effect — such as output or a run-time error — to its cause. Trace viewing consists of the programmer selecting expressions whose parent, the function call that generated the expression, is then displayed. An example with programmer-selected expressions underlined: `42` → `3*14` → `2*7` → `factorial 2` → `factorial 3`.
– HAT-OBSERVE allows the observation of functions. A functional value is displayed as a finite mapping from all the arguments the function was called with in the computation to the respective results, for example: {`factorial 0 = 7, factorial 1 = 7, factorial 2 = 14, factorial 3 = 42`}.

Each tracing method gives a different view of a computation; in practice, the views are complementary and can productively be used together [1].

All HAT viewing tools display only expressions and equations of the traced computation. They just allow opening a program source browser with the cursor positioned at the redex or at the definition of the function of current interest.

All expressions of the traced computation originate from the source program. The programmer is likely to be familiar with the source program, because they wrote it, read it beforehand and/or will have to modify it. A source program can provide a concise orientation structure for the often huge computation trace.

In both HAT-DETECT and HAT-TRAIL navigation through the trace actually follows the source program structure. In HAT-DETECT the order of questions follows a call-by-value call-structure, that is, after a question about a function call has been answered with "no", the following questions will be about function calls from the definition body of this function. In HAT-TRAIL a marked expression is part of an instance of a function body and its parent is the corresponding instance of the function application. Sadly this close relationship to the source program is currently lost on many programmers using HAT.

These are several arguments for extending the existing tools with a display of the program source, with various parts of interest marked. I started developing such a source-based extension of HAT-DETECT . Quickly it became apparent that the source display could provide far more than just an orientation point for the navigation through the questions of algorithmic debugging.

2 Source-Based Algorithmic Debugging

Algorithmic debugging is based on the representation of a computation as an Evaluation Dependency Tree (EDT). Each node is labelled with an equation, which is a reduction of a redex to a value. The tree is basically the proof tree of a natural semantics for a call-by-value evaluation with miraculous stops where arguments are not needed for the final result value. The call-by-value structure ensures that arguments are values, not complex unevaluated expressions, and that the tree structure reflects the program structure. So the redexes of children are all instances of the definition body of the function that is reduced in their parent node.

```
main = sort ``sort''

sort :: Ord a => [a] -> [a]
sort []     = []
sort (x:xs) = insert x (sort xs)

insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x <= y then x : ys else y : insert x ys
```

Fig. 1. A faulty insertion sort program

Fig. 2. Evaluation Dependency Tree for insertion sort

In algorithmic debugging the programmer states that some nodes of the EDT are correct, that is, the reduction of the function agrees with the semantics the programmer *intends* the function to have. Other nodes the programmer declares to be incorrect. A node that is incorrect but whose children are all correct is faulty. The definition of the function reduced in this node is faulty and needs to be modified. This localisation of a program fault is intuitive: if the reduction of a function is incorrect, but all the calls made from this function call are correct, then the definition body must be faulty.

There are many options for marking program parts of interest and supporting navigation in an algorithmic debugging viewer with source display.

– *Navigation:* The redex of the current node could be marked where it appears in a definition body. The whole definition body, which belongs to the function of the parent of the current node, could also be marked. Additionally all siblings of the current node could be marked in the definition body. These markings would improve orientation and simplify free navigation through

the EDT. The programmer could quickly navigate to EDT nodes of interest and declare only nodes as correct or incorrect where this judgement is certain. The markings of redexes in the definition body could even be different depending on if they were declared correct or incorrect.
- *Slicing:* If a node is incorrect, then either itself or one of its descendants must be faulty. With each node a function definition is associated. So with an incorrect node we can associate the function definitions of itself and all its descendant nodes. The fault causing the incorrect node must be in these function definitions. When the current node is incorrect, these function definitions could be marked. Their might be so few definitions that the programmer spots the fault straight away or the programmer might only be interested that certain definitions cannot be responsible for the incorrect node.
- *Coverage:* For a specific reduction usually only parts of the definition body of the reduced function are evaluated, both because of conditionals and lazy evaluation. The Hat trace contains this information and hence instead of marking full definitions only those parts that were demanded should be marked. Only those parts can be faulty.

Highlighting each marking in the source program in a different colour would create a very colourful but confusing display. Hence experiments are needed to determine the usefulness of each marking and their combinations.

3 Information Available in the Trace

The trace is a huge and complex graph structure; however, just from a local part of the trace a local part of an EDT can be constructed.

The trace generation of Hat follows a simple reduction model. In a reduction the application of a function is replaced by an instance of the function body. The same happens in the trace, except that the instance of the function application is not replaced, but connected to the newly added instance of the function body. In consequence a trace contains the full instance of a definition body, even if only parts of that are later evaluated. Evaluated parts of the definition body can easily be identified, because they are connected to instances of their function bodies. However, the trace does not contain any information about the definitions of functions that were never evaluated. To access any of these, a trace viewing tool would need to include a full Haskell front end (as the Hat transformation tool that generates self-tracing programs does) to process the source program. The trace also does not include any information on the pattern matching process and the locations of patterns in the source program.

Originally the Hat trace contained for each recorded expression and each defined function the filename, line and column where it starts in the source program. I extended Hat to record a full location that also includes the line and column at which such an expression or definition ends in the program source.

Some source program markings refer to whole expressions and definitions, but others refer to slices. Slices are collections of program constructs that may

not include all subexpressions. In an extreme case an application has to be marked, without marking its function and arguments. Such a marking can be obtained from subtracting the locations of the subexpressions from the location of the whole application. In the case of an application only the space between the function and the arguments is marked.

4 Related Work

Program slicing is a well-known technique for analysing and particularly debugging programs [5]. The set of all function definitions (or even just its evaluated part) of a subtree of the EDT is a dynamic slice in that sense, with the reduction of the root node as *slicing criterion*. The paper "Dynamic Slicing Based on Redex Trails" [4] is just about to appear. It describes a slicing method for a core language of the Haskell-like functional logic language Curry. Although the slicing criterion is also based on a reduction, these slices are not related to EDTs and the authors do not claim that a fault has to be within a slice. Their trace structure, although also called redex trail, differs in several points. In particular, parent pointers have a different meaning; they do not point to an EDT parent and hence it is unclear if an EDT can be reconstructed from this trace structure.

5 Conclusions

There is a need for extending current tracing tools for functional languages with a display of the program source with various program slices of current interest being marked. The outlined new source-based algorithmic debugging tool is still under heavy development. The tool combines algorithmic debugging with dynamic program slicing and coverage analysis. Only practical use can show in which directions the tool best should evolve.

Acknowledgements

This work relies heavily on previous work on the Haskell tracer Hat by Colin Runciman, Malcolm Wallace and Thorsten Brehm.

References

1. Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood — A comparative evaluation of three systems for tracing and debugging lazy functional programs. In Markus Mohnen and Pieter Koopman, editors, *Implementation of Functional Languages, 12th International Workshop, IFL 2000*, LNCS 2011, pages 176–193. Springer, 2001.
2. Olaf Chitil, Colin Runciman, and Malcolm Wallace. Transforming Haskell for tracing. In *Proceedings of the 14th International Workshop on Implementation of Functional Languages (IFL 2002)*, LNCS 2670, pages 165–181, 2003.

3. Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using QuickCheck and Hat. In *4th Summer School in Advanced Functional Programming*, LNCS 2638, pages 59–99, August 2003.
4. C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*. ACM Press, 2004. to appear.
5. Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
6. Malcolm Wallace, Olaf Chitil, Thorsten Brehm, and Colin Runciman. Multiple-view tracing for Haskell: a new Hat. In *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, UU-CS-2001-23. Universiteit Utrecht, 2001. Final proceedings to appear in ENTCS 59(2).