Source-Based Trace Exploration

Olaf Chitil
University of Kent, UK

Abstract. Tracing a computation is a key method for program comprehension and debugging. HAT is a tracing system for Haskell 98 programs. During a computation a trace is recorded in a file; then the user studies the trace with a collection of viewing tools. Different views are complementary and can productively be used together. Experience shows that users of the 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 Hat and Its Views

A tracer gives us access to otherwise invisible information about a computation. It is a tool for understanding how a program works and for locating the source of runtime errors in a program. HAT is a tracer for the lazy functional language Haskell 98. HAT combines the tracing methods of several preceding systems [13, 3, 4]. Tracing a computation with HAT consists of two phases, trace generation and trace viewing:

- **HAT-TRACE** 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 \texttt{factorial 3 = 42}?” which we have 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 us to follow redex trails; we explore a computation backwards, from an effect — such as output or a runtime error — to its cause.

First, a special version of the program runs. In addition to its normal input/output behaviour it writes a trace into a file. Second, after the program has terminated, we study the trace with a collection of viewing tools:

- **HAT-TECT** 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 \texttt{factorial 3 = 42}?” which we have 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 us to follow redex trails; we explore a computation backwards, from an effect — such as output or a runtime error — to its cause.
Trace viewing consists of us selecting expressions whose parent, the function call that generated the expression, is then displayed. An example with selected expressions underlined: $42 \rightarrow 3*14 \rightarrow 2*7 \rightarrow \text{factorial } 2 \rightarrow \text{factorial } 3$.

- **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 viewing tool gives a different view of a computation; in practice, the views are complementary and can productively be used together [2]. The trace as concrete data structure liberates the views from the time arrow of the computation. HAT provides valuable insights into long computations of real-world programs.

Nonetheless, HAT 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.

Surprisingly, 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.

This paper describes a new HAT viewing tool, HAT-EXPLORE, 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. The following sections describe in several steps the design of HAT-EXPLORE and some implementation issues. HAT-EXPLORE is part of the HAT distribution which is available from http://haskell.org/hat.

## 2 Algorithmic Debugging

Algorithmic debugging is based on the representation of a computation as an Evaluation Dependency Tree (EDT) [6][5]. Each node of the tree 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. Figure 2 shows the EDT of the sorting program given in Figure 1. Note