# Source-Based Trace Exploration

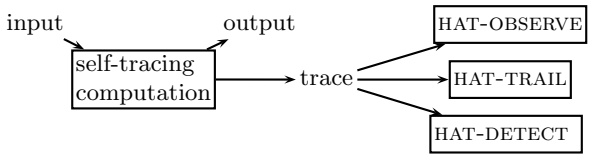
Olaf Chitil

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:




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-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 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` → `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 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 that {`IO`} denotes an IO-action value for which no informative representation is available.

In algorithmic debugging an oracle decides which nodes of the EDT are correct and which are incorrect. A node is correct if and only if its reduction of

```
main = putStrLn (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.

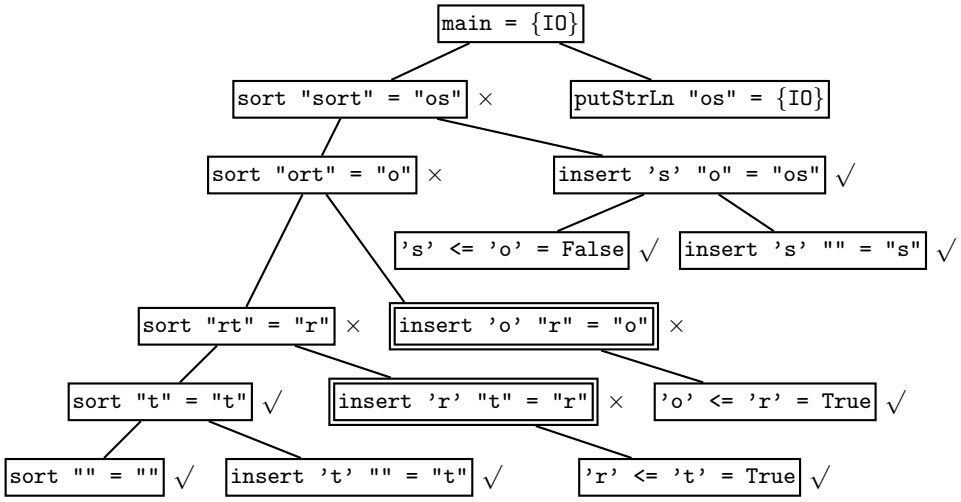



**Fig. 2.** Evaluation Dependency Tree for insertion sort.

a function agrees with the semantics we as programmers *intend* the function to have. 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. Hence the aim of algorithmic debugging is to find a faulty node. The definition of a faulty node is intuitive: if a function call yields an incorrect result, but all the calls made from this function call are correct, then the definition body must be faulty. In the EDT of Figure 2 all nodes except the IO-related ones have been declared as correct ($\surd$) or incorrect ($\times$). The double framed nodes are faulty. Both faulty nodes are caused by the same faulty part of the definition of `insert`.

A formal specification can be the basis of the oracle and the correctness of nodes can be considered in any order. However, most algorithmic debugging systems assume that the user is the oracle and implicitly traverses the EDT while answering questions about correctness with "yes" or "no". Entering "no" makes a child of the current node the new current node (If the node has no children, the aim of debugging has been reached, because the current node is

faulty). Entering "yes" makes the next yet unvisited sibling of the current node the new current node (if all siblings have been visited, then the next yet unvisited sibling of the parent is chosen, and so on). Usually, the user of an algorithmic debugging tool is not meant to be aware of these non-trivial navigation steps, but shall just answer the questions.

## 3 Source-Based Free Navigation through the Evaluation Dependency Tree

Basically HAT-EXPLORE is a tool for free navigation through an EDT. The EDT is a complete representation of a computation. While navigation via "yes"/"no" answers is fairly complex, it is straightforward to provide simple navigation through the tree via the cursor keys: up to the parent, down to the first child, and left and right to siblings. Most importantly, however, the program source can provide good orientation while traversing the EDT. The call-by-value structure of the EDT ensures that the EDT reflects the program structure. If $f \ldots = \ldots$ is the reduction of a node, then the redexes of its children are all instances of the definition body of the function $f$. Figure 3 demonstrates this property.

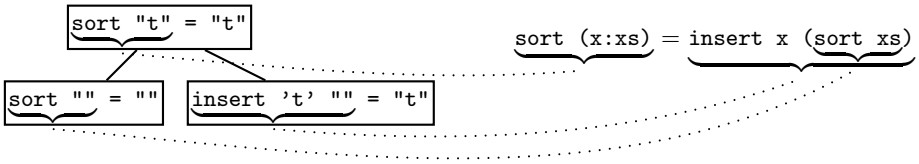



**Fig. 3.** Relationship between parent and children in EDT and program source.

The display of HAT-EXPLORE is divided into two parts: the current reduction and the source. In the source the *call site* of the redex of the current reduction is underlined.

```
==== Hat-Explore 0.3 ==== Call 2/2 ==============================
sort "t" = "t"
---- Insert.hs ---- line 1 to 9 ---------------------------------
main = putStrLn (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)
```

Optionally the *definition site* of the function of the redex can also be highlighted, but usually definition site and call site are far apart in the source and

having more than one source window would be confusing. The call site is a smaller, more specific fragment of the source than the definition site. Additionally, this fragment is directly surrounded by the call sites of the redexes of the siblings of the current reduction. The call sites of the siblings are also highlighted but not underlined like the current redex. When we change the current reduction via left or right cursor keys, only underlining changes in the source. So, given the state of the last screenshot, pressing the left cursor key yields (display shortened):

```
==== Hat-Explore 0.3 ==== Call 1/2 ==============================
insert 'r' "t" = "r"
---- Insert.hs ---- line 1 to 5 --------------------------------
main = putStrLn (sort "sort")

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

In contrast, a move to the parent via cursor key up or to a child via cursor key down usually requires a complete change of the displayed source, because parents and children are further away. So pressing cursor key down yields:

```
==== Hat-Explore 0.3 ==== Call 1/1 ==============================
'o' <= 'r' = True
---- Insert.hs ---- line 6 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)
```

Pressing cursor key up once returns to the last but one screen. Pressing cursor key up again yields:

```
==== Hat-Explore 0.3 ==== Call 1/2 ==============================
sort "ort" = "o"
---- Insert.hs ---- line 4 to 7 --------------------------------
sort [] = []
sort (x:xs) = insert x (sort xs)

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

The call site of a parent or child can be in a different module. HAT-EXPLORE lazily loads a module source when it is needed and displays it.

## 4 A Stack for Context

Experience shows that after some navigation we still often lose orientation. We know the call site of the current reduction, but a single call site is possibly used

very often in a computation. More contextual information about the current reduction is needed. So a stack of parents is added to the display of HAT-EXPLORE. It shows the descendants chain of reductions from `main = {IO}` down to the current reduction as last element. Every time we move down to a child, this child is pushed on the stack; every time we move up to a parent, an element is popped from the stack. Hence the stack is displayed upside down, with the top element in the bottom line.

```
==== Hat-Explore 1.0 ==== Call 1/2 ==============================
 1. main = {IO}
 2. sort "sort" = "os"
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
---- Insert.hs ---- line 3 to 9 ---------------------------------
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)
```

In practice reductions are much larger than in the small sorting example; a single reduction may cover several lines. Hence only a small number of reductions can be shown at a time. Experience shows that in most cases the last few reductions are sufficient for orientation in the EDT.

## 5 Source-Based Algorithmic Debugging

HAT-EXPLORE still supports algorithmic debugging. We can declare if the current reduction is correct or incorrect with respect to our intentions and also change and take back any previous such declaration. The tool 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.

Let us work step by step through an example session for the faulty insertion sort program. The tool starts with the reduction of `main`.

```
==== Hat-Explore 2.00 ==== 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 2.00 ==== 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 2.00 ==== 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 2.00 ==== 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 2.00 ==== 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 2.00 ==== 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 2.00 ==== 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 2.00 ==== 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 2.00 ==== 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 rely on algorithmic debugging at all but just use declarations of (in)correctness as memory hints.

## 6 Program Slicing

Algorithmic debugging is based on the principle that if a node of the EDT is incorrect, then a faulty node must be amongst this node and its descendants, that is, the bug is in that sub-EDT of the EDT. If a sub-EDT of this sub-EDT has a correct node as root, that sub-EDT can be subtracted, the faulty node must be in the remaining sub-EDT. During algorithmic debugging the faulty sub-EDT is cut smaller and smaller, until it is reduced to a single node, the faulty node. HAT-EXPLORE marks the definition of the function reduced in the faulty node. However, that happens only rather late, after the faulty node has been identified. So in addition, HAT-EXPLORE can mark the definitions of all functions that are reduced in the nodes of the current faulty sub-EDT. These definitions comprise the faulty slice.

In the example session of the previous section a faulty slices is marked in *italics*. When `sort "sort" = "os"` is declared as wrong, the definition of `sort` and `insert` become the faulty slice. When `insert 'o' "r" = "o"` is declared as wrong, the definition of `sort` is subtracted from the faulty slice, leaving only the definition of `insert`.

While we declare nodes as correct or incorrect, the faulty sub-EDT and thus the slice of definitions that must contain a fault keep shrinking. 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. While traversing an EDT we often skip declaring the correctness of a node; for example, because it might be hard (large input or output) or impossible (values of abstract data types) to determine. Figure 4 shows a partially annotated EDT where the nodes of the faulty sub-EDT are marked.

A faulty sub-EDT of a partially annotated EDT is defined as a minimal connected subgraph such that for any completion of the annotation the sub-EDT contains a faulty node. So an unannotated EDT has no faulty sub-EDT, because all nodes might be correct. In general an annotated EDT can have several (disjoint) faulty sub-EDTs. HAT-EXPLORE marks the faulty sub-EDT that contains the currently viewed node or, if the current node is outside of any faulty sub-EDT, the next faulty sub-EDT above the current node.

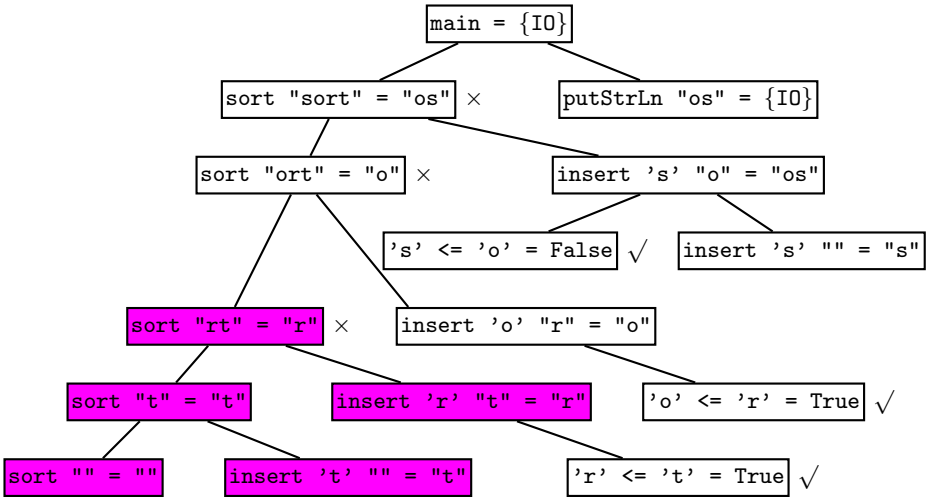



**Fig. 4.** A Faulty Sub-EDT.

## 7 Smaller Faulty Slices and Code Coverage

The faulty slice can be made smaller without additional input from us. Keeping the faulty sub-EDT unchanged, we can determine a smaller faulty slice. When the faulty sub-EDT contains a reduction $f \ldots = \ldots$, it is not necessary 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 that was actually evaluated for that particular reduction. Evaluated parts of the definition are the call sites of the children of the node plus demanded constants, data constructor applications and literals.[1] HAT-EXPLORE optionally only shows this smaller faulty slice. In our example program the "else" branch was never evaluated for the current, incorrect reduction.

```
==== Hat-Explore 2.03 ==== Call 2/2 | faulty slice | executed ===
 1. main = {IO}
 2. sort "sort" = "os"
 3. sort "ort" = "o"
---- Insert.hs ---- line 3 to 9 ---------------------------------
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)
```

[1] If a constant is evaluated, it is impossible to determine if it was demanded for the currently considered reduction or a different part of the computation, because

Unfortunately it is no longer true that the fault has to be within the faulty slice. The fault may also be within the patterns on the left-hand-sides of the defining equations.[2] The fault might even be that an equation that should be there is missing. This last possibility cannot be expressed well by marking any slice at all.

By declaring the root reduction of the EDT, `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:

```
==== Hat-Explore 2.03 ==== Call 1/1 | faulty slice | executed ===
 1. main = {IO}


---- Insert.hs ---- line 1 to 9 --------------------------------
main = putStrLn (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)
```

So HAT-EXPLORE can serve as a code coverage tool.

## 8 Trusting

HAT supports a notion of trusting modules. The computation of these modules is not traced [3]. By default all Haskell standard libraries are trusted. The reduction of a trusted function is still recorded in the trace. For example, `length "hi" = 2` may be recorded, but not its recursive call `length "i" = 1`. So leafs of the EDT can be reductions of trusted functions. HAT-EXPLORE assumes by default that these reductions are correct.

Trusted functions can be higher-order and the functional arguments may be normal untrusted functions [10, 5], for example `map myInc [1,2,3] = [2,3,4]`. In that case the reduction of the trusted function can have children, namely

constants are shared. For most data constructor applications and literals, the entry in the HAT trace contains no indication if they were ever demanded in the computation. To be on the safe side, in all such cases the expression has to be included in the slice, if the surrounding expression construct is included.

[2] The Hat trace does not include any information on the pattern matching process. For an unsuccessful match it cannot be determined which parts of a pattern were used and exactly where matching failed. The trace has no information on locations of patterns in the source. Nonetheless, HAT works fine for computations that abort with a pattern match failure, as Section 10 demonstrates.

the reductions of the passed untrusted functions. So `map myInc [1,2,3] = [2,3,4]` has the children `myInc 1 = 2`, `myInc 2 = 3` and `myInc 3 = 4`. In general, trusting causes parts of an EDT to be "cut out", even out of the middle of the tree. If a trusted reduction has children, it cannot assumed to be correct by default.

The children of trusted higher-order functions have call sites within trusted modules. Displaying these call sites would contradict the idea of a trusted module whose implementation is irrelevant.[1] So when the current reduction is the child of a trusted reduction, HAT-EXPLORE highlights the call site of the trusted parent instead of the child; it does so in a different style to indicate the different situation. The children of such a reduction without call site are again reductions with call site. So there is no danger of us losing orientation because we might have to make a long sequence of navigation steps without highlighting of call sites.

```
==== Hat-Explore 2.03 ==== Call 2/4 | faulty slice | executed ===
 1. main = {IO}
 2. sort "sort" = "os"
 3. foldr insert [] "sort" = "os"
 4. insert 'r' "t" = "r"

---- FoldrInsert.hs ---- line 3 to 9 -----------------------------
sort :: Ord a => [a] -> [a]
sort xs = foldr insert [] 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)
```

## 9 Constants

A constant definition, such as `nats = [0..]`, has to be handled specially in the construction of an EDT. In a computation the definition body is only evaluated once and the value is shared by all calls (i.e. uses) of the constant in the program. The algorithmic debugger Freja [5] does not include the reduction of a constant at its call site, but produces a forest of EDTs, one EDT per constant definition (the definition of `main` is a constant definition). This approach would complicate free navigation. Hence in HAT-EXPLORE there is only a single EDT with the EDT of a constant inserted at its call sites. The EDT of the constant is shared by all call sites, so that the EDT is no longer a tree but a directed graph. Navigation into the EDT of a constant is natural. Where to go back up is also uniquely identified by the information in the stack.

Because constant definitions may be (mutually) recursive, the EDT may be cyclic. Algorithmic debugging only works for trees or acyclic graphs. It is currently the responsibility of the user to be aware that algorithmic debugging

[1] Hence the HAT trace also does not contain any such source location information.

may not be able to locate a faulty reduction within the computation of mutually recursive functions. The faulty slice is still correct, but it may never shrink further than a set of mutually recursive definitions.

## 10 Other Starting Points

Normally HAT-EXPLORE starts with the reduction of `main`. Although paths through the EDT are only logarithmic in the size of the tree, a reduction of interest may still be far away from the root.

Other viewing tools such as HAT-TRAIL and HAT-OBSERVE may give quicker access to a reduction of interest. It was simple to extend these tools so that we can directly switch from one of them to HAT-EXPLORE, starting at the reduction that we just investigated in the other tool.

Experience shows that faults are often not far (within the EDT) from the observed error. Hence the feature of HAT-TRAIL, to start directly at the reduction that raised a runtime error, has been added to HAT-EXPLORE. A slightly modified version of our insertion sort causes a pattern match failure. HAT-EXPLORE starts as follows, displaying the error value as `_|_` (bottom):

```
==== Hat-Explore 2.03 ==== Call 1/2 | faulty slice | complete ===
 4. sort "rt" = _|_
 5. sort "t" = _|_
 6. insert 't' [] = _|_
---- Insert.hs ---- line 1 to 9 --------------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)

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

## 11 Implementation

HAT-EXPLORE has been implemented in about 1000 lines of Haskell. It also uses a library for accessing the trace that is shared with other viewing tools.

The Hat trace is a complex graph of expression components. The reconstruction of an EDT from this structure is described in [13]. For the efficiency of HAT-EXPLORE it is important that a small part of an EDT can be constructed easily from reading only a small part of the trace. So both memory and time costs for the construction of the small part of an EDT that is demanded by the user in a single interaction step is independent of the generally huge size of the trace. Only determining the faulty slice is expensive. It requires traversing the whole faulty sub-EDT in the trace. Hence the user can turn off this feature.

The algorithmically most complex part of HAT-EXPLORE is the handling of source slices. A slice is a set of source locations, where a location consists of start

line and column and end line and column. HAT-EXPLORE comprises an abstract data type of slices with several functions for combining and subtracting slices. Slices are used to highlight parts of the source while excluding subexpressions. In an extreme case an application has to be highlighted, without highlighting its function and arguments. The slice for highlighting can be obtained by 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 may remain in the slice.

To support HAT-EXPLORE, HAT required two extensions: Originally the trace contained for each recorded expression and each defined function the filename, line and column where it starts in the source. Now HAT records a full location that also includes the line and column at which such an expression or definition ends. The lexer and parser had to be modified and the abstract syntax tree slightly extended. Second, now a trusted reduction in the trace has an explicit list of pointers to its children. In the past, HAT-DETECT used an incomplete approximation algorithm to determine children; to find all children for certain, a time consuming search through most of the trace would have been required. Only the definition of a single combinator in the HAT library of tracing combinators [3] had to be modified. Both extensions slightly changed the trace file format, but only few changes in a library for accessing the trace were needed to make all previously existing HAT viewers work with the file format. Overall, both extensions only needed a small number of changes to HAT and benefit other viewing tools besides HAT-EXPLORE.

HAT-EXPLORE has a simple textual user interface based on text interleaved with ANSI escape sequences for various forms of highlighting. This user interface is portable and was easy to implement. Nonetheless it has its limitations; in particular, different highlighting of nested expressions yields output that is hard to read. For this purpose multiple underlining similar to the old redex trail browser [11] would be more suitable.

## 12 Related Work

Using HAT-EXPLORE reminds one of using a *classical stepping debugger* for an imperative programming language, such as DDD[2]. The debugger highlights the current execution line. The user can perform one execution step, moving to a line which was called from the previous line. Alternatively, the user can go to the next line, skipping the execution of all function calls. So the source-based navigation model of HAT-EXPLORE has already been proven useful for imperative languages. Users of these stepping debuggers can build on previous experience when moving to HAT-EXPLORE. While the user steps through the computation HAT-EXPLORE also provides with each function call its result. In a side-effect free functional language the result fully describes the semantics of the function call. Thus it is far easier to locate the faulty program part than it is in a stepping debugger for an imperative language.

[2] `http://www.gnu.org/software/ddd/`

*Algorithmic debugging* [9] has been the starting point for HAT-EXPLORE. There exist several algorithmic debuggers for lazy functional languages [5, 13, 8]. They all allow more direct navigation through the EDT then via "yes"/"no" answers but they do not encourage free navigation. They do not use the source.

*Program slicing* is a well-known technique for analysing and particularly debugging programs [12]. The faulty slice of HAT-EXPLORE (both with full definitions and with evaluated expressions only) is a dynamic slice in that sense, with the reduction of the root node as *slicing criterion*. However, whereas program slicing is based on the control and data flow of a computation, the EDT expresses the control and data flow of a computation only in a limited form.

In [7] a slicing method for a core of the Haskell-like functional logic language Curry is described. 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 [1], although also called redex trail, differs in several points from the HAT trace. In particular, parent pointers have a different meaning; they do not point to an EDT parent and hence it is doubtful that an EDT can be reconstructed from this trace structure.

## 13 Conclusions and Future Work

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. Initial informal feedback from users has been positive.

The HAT system gives important insights into the internals of computations of Haskell programs. Nonetheless there is still much work to do. Features of several existing HAT viewers could be combined. In particular, it is possible to merge HAT-TRAIL and HAT-EXPLORE. However, the resulting tool might be too complex to use. Alternatively, HAT-TRAIL could be extended by source-based orientation facilities. HAT does not support all types of programs well. For example, tracing of IO intensive programs is limited because the IO monad is just treated as an abstract data type with unknown values; some higher-order programs rely on a complex control flow that is hard to visualise adequately.

This paper demonstrates that it is relatively easy to extend the HAT system by a new viewing tool for which it was not designed originally. HAT provides a modular framework for further exploration of tracing systems.

## Acknowledgements

This work relies heavily on previous work on the Haskell tracer HAT by Colin Runciman, Malcolm Wallace and Thorsten Brehm. I also thank the four referees for their constructive comments.

## References


1. Bernd Braßel, Michael Hanus, Frank Huch, and German Vidal. A semantics for tracing declarative multi-paradigm programs. In *Proceedings of the 6th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 179–190. ACM Press, 2004.
2. 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.
3. 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.
4. 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.
5. Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.
6. Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering: An International Journal*, 4(2):121–150, April 1997.
7. 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)*, pages 123–134. ACM Press, 2004.
8. B. Pope and Lee Naish. Practical aspects of declarative debugging in Haskell-98. In *Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 230–240, 2003.
9. E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.
10. Jan Sparud and Colin Runciman. Complete and partial redex trails of functional computations. In C. Clack, K. Hammond, and T. Davie, editors, *Selected papers from 9th Intl. Workshop on the Implementation of Functional Languages (IFL'97)*, pages 160–177. Springer LNCS Vol. 1467, September 1997.
11. Jan Sparud and Colin Runciman. Tracing lazy functional computations using redex trails. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proc. 9th Intl. Symposium on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS Vol. 1292, September 1997.
12. Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
13. 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).