Hat – The Haskell Tracer
Version 1.04
Users' Manual

The ART Team

21 May 2001

Contents

1 Introduction

Hat is a source-level tracer for Haskell (the *Ha*skell *T*racer). It is a tool that gives the user access to otherwise invisible information about a computation. Thus Hat helps locating errors in programs. However, it is also useful for understanding how a correct program works, especially for teaching and program maintenance. Hence we avoid the popular name "debugger". Note that a profiler, which gives access to information about the time or space behaviour of a program, is also a kind of tracer. However, Hat is not intended for that purpose. Hat measures neither time nor space usage.

Conventional tracers (debuggers) for imperative languages allow the user to step through the program computation, stop at given points and examine variable contents. This tracing method is unsuitable for a lazy functional language such as Haskell, because its evaluation order is complex, function arguments are usually unwieldy large unevaluated expressions and generally computation details do not match the user's high-level view of functions mapping values to values.

Tracing a program with Hat consists of two phases: First the specially compiled program runs as normal, except that additionally a trace is written to file. Second, after the program has terminated, the trace is viewed with a tool.

Hat can be used for programs that terminate normally, that terminate with an error message or that terminate when interrupted by the programmer.

The trace consists of high-level information about the computation. It describes each reduction, that is, the replacements of an instance of a left-hand side of an equation by an instance of its right-hand side, and the relation of the reduction to other reductions.

Because the trace describes the whole computation, it is huge. Hence the programmer uses tools to selectively view the fragments of the trace that are of interest. Currently Hat includes four tools, `hat-stack`, `hat-observe`, `hat-detect` and `hat-trail` for that purpose. Each tool shows fragments of the computation in a particular way, highlighting a specific aspect.

2 Obtaining the Trace of a Computation

To obtain a trace of a computation of a program, the program has to be compiled specially with `nhc98` and then executed.

2.1 Compilation

Compile all modules of the program with `nhc98` with the `-T` option; also specify `-T` at link-time. Using `hmake -T` does all the necessary compiling and linking automatically.

Tracing makes programs use more heap space. As a rough rule of thumb, traced programs require 3 times as much heap space as untraced ones. However, because traced programs allocate (and discard) much memory, it is useful to choose an even larger heap size to reduce garbage collection time. The preset heap size for untraced programs is 400KB and for traced programs 2MB. For example, you can set the heap size at compile (link) time with `-H10m` or for a specific execution of the program with `+RTS -H10M -RTS` to a ten megabyte heap.

Note that compilation does not insert the complete file paths of the source modules into the executable. The trace viewers assume that the source modules are in the same directory as the executable.

2.2 Execution

The executed traced program behaves exactly like the untraced program, except that it is (currently about 50 times) slower and additionally writes a trace to file. If it seems that the computation is stuck in a loop, then force halting by keying an interrupt (usually `Ctrl-C`). After termination of the program (normal termination or caused by error or interrupt) you can explore the trace with any of the programs described in the following sections.

The execution of a program *name* creates the trace files *name*`.hat`, *name*`.hat.bridge` and *name*`.hat.output`. The latter is a copy of the whole output of the computation. The first is the actual trace. It can easily grow to several hundred megabytes. To improve the runtime of the traced program you should create the trace file on a local disc, not on a file system mounted over a network. The trace files are always created in the same directory as the executable program.

2.3 Trusting

Hat enables you to trace a computation without recording every reduction. You can *trust* the function definitions of a module. Then the calls of trusted functions from trusted functions are not recorded in the trace.

Note that a call of an untrusted function from a trusted function is possible,because an untrusted function can be passed to a trusted higher-order function. These calls are recorded in the trace.

For example, you may call the trusted function `map` with an untrusted function `prime`: `map prime [2,4]`. If this call is from an untrusted function, then the reduction of `map prime [2,4]` is recorded in the trace, but not the reductions of the recursive calls `map prime [4]` and `map prime []`. However, the reductions of `prime 2` and `prime 4` are recorded, because `prime` is untrusted.

You should trust modules in whose computations you are not interested. Trusting is desirable for the following reasons:

- to keep the size of the trace file smaller (main point)
  - to save file space
  - to avoid unnecessary detail when viewing the trace
- to reduce the runtime of the traced program (slightly)

If you want to trust a module, then compile it with the options `-T -trusted` (an object file that has been compiled without any tracing option cannot be used). By default the Prelude and the standard libraries are trusted.

3 Viewing a Trace

Although each tool gives a different view on the trace, they all have some properties in common.

The tools show function arguments in evaluated form, more precisely: as far evaluated as the arguments are at the end of the computation. For example, although in a computation the unevaluated expression `(map (+5) [1,2])` might be passed to the function `length`, the tools will show the function application as `length [1+5,2+5]` or `length [_,_]` (assuming the list elements were not evaluated).

Unevaluated subexpressions are sometimes shown or indicated by the underscore _.

In traces of aborted computations the bottom symbol ⊥ may appear. It indicates a subexpression that was under evaluation when the computation was aborted.

The following faulty program is used as example in the description of most viewing tools:

```
main = let xs :: [Int]
           xs = [4*2,5 `div` 0,5+6]
       in  print (head xs,last' xs)

last' (x:xs) = last' xs
last' [x] = x
```

4 Hat-Stack

For aborted computations, that is computations that terminated with an error message or were interrupted, hat-stack shows in which function call the computation was aborted. It does so by showing a *virtual* stack of function calls (redexes). So every function call on the stack caused the function call above it. The evaluation of the top stack element caused the error or during its evaluation the computation was interrupted. The shown stack is *virtual*, because it does not correspond to the actual runtime stack. The actual runtime stack enables lazy evaluation whereas the *virtual* stack corresponds to a stack that would be used for eager (strict) evaluation.

4.1 Usage

To use hat-stack enter

`hat-stack` *programname*

where *programname* is the name of the traced program.

4.2 Example

Here is an example output:

```
Program terminated with error:
    "No match in pattern."
Virtual stack trace:
    (last' [])          (Example.hs: line-6/col-16)
    (last' (5+6:[]))            (Example.hs: line-6/col-16)
    (last' ((div 5 0):5+6:[]))          (Example.hs: line-6/col-16)
    (last' (8:(div 5 0):5+6:[]))                (Example.hs: line-4/col-27)
    main                (Example.hs: line-2/col-1)
```

4.3 Further Information

Hat-trail can also show this virtual stack. Hat-stack is a simple tool that enables you to obtain the stack directly.

The description of hat-trail contains more details about the relations between the stack elements. Hat-stack shows ☑ and subexpressions of very large expressions as a dot (·).

5 Hat-Observe

Hat-observe shows a top-level function. That is, for a given top-level function name it shows all the arguments with which is was called during the computation together with the respective results.

5.1 Usage

To use hat-observe enter

`hat-observe [-v] [-r]` *variable* `[in` *variable'*`]` *programname*`[.hat]`

where *programname* is the name of the traced program, and *variable* and *variable'* are top-level functions/constants of the program.

`-v` verbose mode
: Normally unevaluated subexpressions of arguments or results are just shown as _. With this option they are shown in full.

`-r` recursive mode
: Recursive function applications are not shown.

If a second variable is given after the keyword `in`, then only the calls of the first variable from the right-hand-side of the second variable are shown.

5.2 Examples

```
hat-observe "last'" Example
```

shows

```
last' (8:_:_:[]) = _|_
last' (_:_:[]) = _|_
last' (_:[]) = _|_
last' [] = _|_
```

```
hat-observe -v "last'" Example
```

also shows the unevaluated subexpressions

```
last' (8:(div 5 0):5+6:[]) = _|_
last' ((div 5 0):5+6:[]) = _|_
last' (5+6:[]) = _|_
last' [] = _|_
```

```
hat-observe -r "last'" Example
```

only shows the single non-recursive call of `last'`

```
last' (8:_:_:[]) = _|_

hat-observe "last'" in "last'" Example
```

only shows the recursive calls of `last'`

```
last' (_:_:[]) = _|_
last' (_:[]) = _|_
last' [] = _|_
```

5.3 Further Information

A function may be called several times with the same arguments. Hat-observe shows these arguments and the result only once. Furthermore, because a function may not need full evaluation of its arguments, a function call may be more general than another one in that the arguments are less evaluated in the first than the second one. If the result is the same or the result for the less general arguments is less evaluated, than the application to the less general arguments is not shown.

6 Hat-Detect

Hat-detect is an interactive tool that enables locating semi-automatically an error in a program by answering a sequence of yes/no questions. Each question asked by hat-detect concerns the reduction of a redex – that is, a function application – to a value. You have to answer *yes*, if the reduction is correct with respect to your intentions, and *no* otherwise. After a number of questions hat-detect states which reduction is the cause of the observed faulty behaviour – that is, which function definition is incorrect.

At the moment hat-detect can only be used for computations that produce faulty output, not for computations that abort with an error message or are interrupted. Also it does not work correctly for programs that read input and for programs that handle infinite data structures.

6.1 Starting & Exiting

Start hat-detect by entering

`hat-detect` *programname* `[.hat]`

where *programname* is the name of the traced program.

To exit hat-detect enter `quit` or `q`.

6.2 Basic Functionality

Consider the following program:

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

sort xs = foldr insert [] xs

main = print (sort [3,2,1])
```

It produces the faulty output `[3,3,3]` instead of the intended output `[1,2,3]`.

The following is an example session with hat-detect for the computation. The y/n answers are given by the user:

```
1> main = IO (print (3:3:3:[]))   (Y/?/N): n

2> sort (3:2:1:[]) = 3:3:3:[]   (Y/?/N): n

3> insert 1 [] = 1:[]   (Y/?/N): y

4> insert 2 (1:[]) = 2:2:[]   (Y/?/N): n

5> insert 2 [] = 2:[]   (Y/?/N): y
Error located!
Bug found: "insert 2 (1:[]) = 2:2:[]"
```

The first question of a session always asks if the reduction of `main` is correct. Hat-detect indicates that `main` is reduced to an IO action and shows the action in an unfortunately strange way. Nontheless in the example the answer is obviously *no*.

Also the answer to the second question is obviously *no*. The third and the fifth reduction are correct, whereas the fourth is not. Note that hat-detect does not ask about any reduction of `foldr`, because it is trusted.

After the answer to the fifth question hat-detect determines the location of the error. The equation that is used to reduce the redex `insert 2 (1:[])` is wrong. Indeed, in the case `x > y` (note: `2 > 1`) the right-hand side should be `y : insert x ys`.

6.2.1 Postponing an answer

If you are not sure about the answer to a question you can answer *?*. Hat-detect proceeds as if the answer had been *no*. But if it cannot locate an error in one of the child reductions, then it will later ask you the question again.

6.2.2 Unevaluated Subexpressions

Reductions may contain underscores `_` that represent unevaluated subexpressions. To answer a question an underscore on the left-hand side of a reduction has to be read as "is the reduction correct for *any* value at this position?" and an underscore on the right-hand side has to be read as "is the reduction correct for *some* value at this position?".

6.3 Algorithmic Debugging

Hat-detect is based on the idea of algorithmic/declarative debugging. The reductions of a computation are related by a tree structure. The reduction of `main` is the root of the tree. The children of a reduction of a function application are all those reductions that reduce expressions occurring on the right-hand side of the definition of the function.

If a question about a reduction is answered with *no*, then the next question concerns the reduction of a child node. However, if the answer is *yes*, then the next question will be about a sibling or a remaining node closer to the root.

An error is located when a node is found such that its reduction is incorrect but the reductions of all its children are correct. That reduction is the source of the error.

6.4 Advanced Features

6.4.1 Single stepping

Hat-detect can be used rather similarly to a conventional debugger. So the input *no* means "step into current function call" and the input *yes* means "go on to next function call". Note that this single stepping is not with respect to the lazy evaluation order actually used in the computation, but with respect to an eager evaluation order that "magically" skips over the evaluation of expressions that are not needed in the remaining computation.

6.4.2 Showing unevaluated subexpressions

By default hat-detect shows unevaluated subexpressions just as underscores `_`. For answering a question these unevaluated subexpressions are irrelevant anyway. However, by entering `v` you can switch to verbose mode which shows these unevaluated subexpressions. By entering `v` again you can switch the verbose mode off.

6.4.3 Going back to a question

The questions are numbered. By entering a number you can go back to any previous question. When you do that the answers to all questions are deleted.

6.4.4 Trusting

Hat-detect does not ask any question about the reductions of functions that are trusted as described in Section 2.3. However, you can trust further functions and thus avoid questions about them by entering `t` instead of `y` when being asked about a specific reduction of a function. By entering `u` you stop trusting all these functions again.

6.4.5 Observing a function

When being asked about a specific reduction of a function you can enter `o` to observe the function. All the arguments with which is was called during the computation together with the respective results are shown, just as by hat-observe. If all the reductions are correct, you will probably want to trust the function as described in the previous paragraph. If you find an erroneous reduction, you can select it with the cursor keys and restart hat-detect with that reduction.

6.4.6 Memoisation

By default hat-detect memoises all answers you gave. So, although the same reduction may be performed several times in a computation, hat-detect will only ask once about it. Hat-detect even avoids asking a question, if a more general question (containing more unevaluated expressions) has been asked before.

You can turn memoisation on/off by entering `m`.

6.4.7 Help

Enter `h` or any input that is not a command for hat-detect to obtain a short overview of the commands understood by hat-detect.

7 Hat-Trail

Hat-trail is an interactive tool that enables exploring a computation *backwards*, starting at the program output or an error message (with which the computation aborted). This is particularly useful for locating an error. You start at the observed faulty behaviour and work backwards towards the source of the error.

Every reduction replaces an instance of the left-hand side of a program equation by an instance of its right-hand side. The instance of the left-hand side "creates" the instance of the right-hand side and is therefore called its *parent*.

Consider our example from Section 3. The error message is caused by the redex `last' []`.
The parent of `last' []` is `last' (5+6:[])`.
The parent of `last' (5+6:[])` is `last' (5 `div` 0:5+6:[])`.
The parent of `last' (5 `div` 0:5+6:[])` is `last' (8:5 `div` 0:5+6:[])`.
The parent of `last' (8:5 `div` 0:5+6:[])` is `main`.
Also the parent of the `8` in the redex `last' (8:5 `div` 0:5+6:[])` is `4*2` whose parent is `xs`.
Hat-trail presents this information as shown in Figure 1.

Every subexpression (if it is not a top-level constant such as `main`) has a parent. In the example the parent of `(8:5 `div` 0:5+6:[])` is `xs`. The parent of every subexpression of an expression can be different.

7.1 Starting & Exiting

Start hat-trail by either entering

```
hat-trail programname[.hat]
```

where *programname* is the name of the program (the extension `.hat` is optional) or by entering

```
hat-trail
```

In the second case you still have to select the name of the program in a file selector box that appears when you select 'Connect to trace' in the File menu.

You can view the trace of a different computation by first selecting 'Disconnect' in the File menu and then use 'Connect to trace' for the new trace.

The browser is exited by selecting "Exit" in the "File" menu.

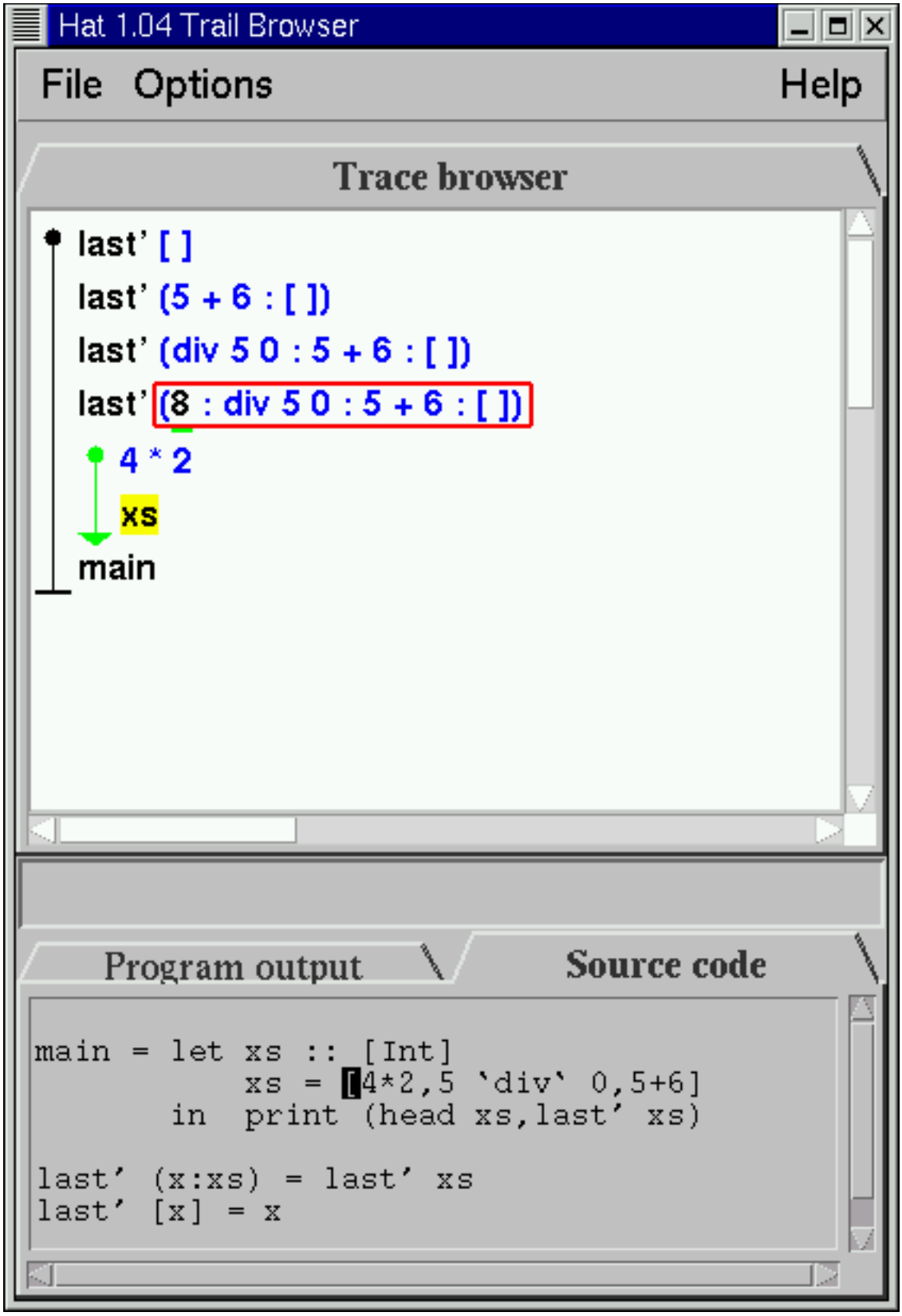

Figure 1: hat-trail

7.2 Basic exploration of a trace

The browser window mainly consists of three panes:

- The trace pane.
  This is the most important pane. In it you explore the trace. With the mouse you can demand to be shown more or less information about parts of the trace. Different kinds of highlighting are used to show how expressions relate to each other.

- The program output pane.
  Here you can select a part of the program output to show its parent redex in the trace pane for further exploration.

- The source code pane.
  Here the source code of the traced program is shown. In the trace pane you can demand to see a specific point of the source code which is then shown in the source code pane.

Additionally the browser window has a menu bar at the top and a message panel between the trace pane and the program output and source code pane.

To save screen space the program output pane and the source code pane share the same space in the browser window. Only one of them can be active any time. By clicking on the tab above the two panes you can select which one should be active.

If a run-time error has occurred, or the computation has been interrupted, the trace pane initially displays the expression under evaluation at the time. Otherwise you first have to select a section of the program output to obtain an expression in the trace pane.

7.2.1 The program output pane

Any output produced by the traced program is shown in the program output pane. The output is divided into sections; there is one section of output for each output action performed by the program. You select a section of the output just by moving the mouse pointer over it. The selected section is shown in blue. By clicking over a section you cause the parent redex for that section to be displayed in the trace pane.

7.2.2 Selecting an expression in the trace pane

You select an expression in the trace pane just by moving the mouse pointer over it. The currently-selected expression is marked by a red box around it. You can select any subexpression of a displayed expression. Hence you select an expression `sqrt y` by moving the mouse pointer on the space between `sqrt` and `y` (the invisible application). If you move the mouse pointer on `sqrt`, then you only select the expression `sqrt`. If you move it on `y`, then you only select `y`. Quickly selecting exactly the expression that you desire may take practice.

7.2.3 Viewing a parent

At the start the trace pane contains only a single line with a redex and an arrow to its left. When you click with the *left* mouse button on any selected subexpression of the redex, the parent of the subexpression is shown in the line below.

If you left-click on the whole redex, then the parent is shown exactly below the selected redex and the arrow on the left is extended appropriately. If you left-click on the whole redex

that just appeared, then its parent is shown below and the arrow is extended again. You can continue left-clicking on whole redexes until the redex is `main` or another top-level constant. These do not have parents. To indicate that the end has been reached, the arrow is replaced by a horizontal line.

If you left-click on a proper subexpression of a redex, then its parent will be shown in the line below as well. However, the parent will be indented further to the right. On its left a new arrow in a new colour appears. The selected expression is underlined in the same colour.

So a parent of a whole redex is shown further down along the same arrow. The parent of a proper subexpression is displayed with a new arrow. The colour of underlinings and arrows indicates which subexpression belong to which parent.

As a shortcut for obtaining the parent of a whole redex you may simply left-click on the tip of its arrow.

7.2.4 Folding away part of a trace

The trace pane may be scrolled, but it quickly becomes cluttered nonetheless. Hence those parts of the trace that are no longer of interest need to be removed from the display.

By left-clicking on an expression for which the parent is already on display, the parent and any of its ancestors are removed from the display.

After you folded away the ancestors of a subexpression that subexpression will be underlined with a dashed line. This dashed line is a reminder that you have already looked at ancestors of the subexpression.

7.2.5 The source code pane

Usually it is not enough just to see the relationship between the values and redexes in a computation. Some coupling to the source code is needed.

If you *right*-click on an expression in the trace pane, then the source file where that instance of the expression was created is loaded and shown in the source code pane, and the cursor moves to the corresponding location in the file.

Note that, if the selected expression is a variable or constant, then the cursor shows this variable or constant in the source code. However, if the selected expression is more complex, then the source may contain variables where the selected expression has values. The selected expression is an *instance* of the source code expression.

To see the *definition* of a variable or data constructor, you right-click on it in the trace as before, but with the *shift* key pressed.

7.2.6 Contraction of a large subexpression

In the trace pane every redex is shown on a single line. However, some redexes are very large. They may for example contain lists with 1000 elements. In the case of cyclic structures it is even crucial that displaying is interrupted at some stage.

Hence, whenever an expression becomes deeper than a certain level, subexpressions are replaced by placeholders. A placeholder looks like an open box, □. By *middle*-clicking on the placeholder you can expand its contents, again just up to a certain depth. Conversely, you can contract any expression to a placeholder by *middle*-clicking on it. This is useful when you want to suppress the display of large uninteresting subexpressions.

Similarly, strings are displayed specially. A string is usually shown as in Haskell, for example `"Hi"`. This representation makes it impossible to sensibly select a substring, for

example `"i"`. However, you can *middle*-click on the string and thus change its representation to separate the first character, for example `'H':"i"`. Thus you can select subexpressions of a string, but the representation is also more verbose. By *middle*-clicking on a longer representation you can change it back to a string representation.

7.2.7 Special expressions

λ-abstractions In the trace pane a λ-abstraction, as for example `\xs-> xs ++ xs`, is represented simply by `(\)`. You have to right-click on `(\)` as described in Section 7.2.5 to see the λ-abstraction itself.

The undefined value ⊥ If the computation is aborted because of a run-time error or interruption by the user, then evaluation of a redex may have begun, but not yet resulted in a value. We call the result of such a redex *undefined* and denote it by ⊥ in the trace pane.

A typical case where we obtain ⊥ is when in order to compute the value of a redex the value of the redex itself is needed. The occurrence of such a situation is called a *black hole*. The following example causes a black hole:

```
a = b + 1
b = a + 1

main = print a
```

When the program is run, it will abort with an error message saying that a black hole has been detected. The trace of the computation will contain several ⊥'s.

Control-flow constructs The control-flow in a function is determined by conditional expressions (`if then else`), `case` expressions and guards. It is often desirable to see why a certain branch was taken in such a control-flow construct. For example, the problem in a function definition might not be that it computes a wrong return value, but that a test is erroneous which makes it select a branch that returns the wrong value.

Hence in hat-trail a redex may not simply be a function application but may be augmented with control-flow information. In general a redex is of the form:

$$\textit{control-flow}_1 \lhd \ldots \lhd \textit{control-flow}_k \lhd \textit{function application}$$

A *control-flow* item is any of the following three

- `if` *expression*
  for a conditional expression
- `case` *expression*
  for a `case` expression
- `|` *expression*
  for a guard

and the symbol ⊲ can be pronounced 'within'. For example, for the program

```
abs x | x < 0 = -x
      | otherwise = x

main = print (abs 42)
```

the parent of the result value `42` is

```
| True ◁ | False ◁ abs 42
```

This redex states that the second branch in the definition of `abs` was taken. The last guard was evaluated to `True` whereas the previous guard was evaluated to `False` (note that in hat-trail a trace is explored backwards). You may ask for the parent of `False` and learn that it was created by the redex `42 < 0`.

7.2.8 Pattern bindings

A program equation with a single variable or a pattern with variables on the left hand side is a pattern binding. The parent of a variable defined by a pattern binding is not the redex that called it, but the redex on whose right-hand-side the pattern binding occurs. Hence variables defined by top-level pattern bindings (i.e. constants) do not have parents.

So usually the parent of an expression is the function call that would have led to the evaluation of the expression if eager evaluation were used. However, this relation breaks down for pattern bindings.

7.3 Advanced exploration of a trace

You can gain a lot of information by just moving the mouse pointer over expressions in the trace pane. Expressions that are related to the currently-selected expression are highlighted in various ways.

7.3.1 Parents that are already shown

Many expressions have the same parent. Showing the same parent twice leads to unnecessary clutter in the trace pane. Hence, if the parent of the currently-selected expression is on display, then it is high-lighted with a *yellow background* colour. This gives you a signal that it is unnecessary to demand the parent.

7.3.2 Siblings

As just stated many expressions have the same parent. To show you which expressions have the same parent as the currently-selected expressions, these expressions are displayed in *blue* colour instead of the normal black colour.

7.3.3 Trusting

Section 2.3 describes trusting of modules as a means to obtain a smaller trace.

In general the result of a trusted function may be an unevaluated expression from within the trusted function. Such an expression is shown as a dashed box, ⍁. It cannot be expanded like a placeholder, □, but it has a parent. For example, for the program

```
main = print (take 5 (from 1))
```

the parent of the result value `[1,2,3,4,5]` is

```
take 5 (1:2:3:4:5:⍁)
```

The parent of ⍁ is `from 1`, as for the whole expression `(1:2:3:4:5:⍁)`.

7.4 Record a tracing session in a script

A script is a recorded session of using the tracer. A script contains all actions taken by the user, and can also be annotated with comments.

7.4.1 Create a script

To create a new script select the "Create script" option in the "File" menu. A file selector box will ask you for the file name of the script. The extension ".scr" will be appended automatically to the file name, if you do not give it.

On the message panel between the trace pane and the program output and source panes the browser informs you that script recording is on. All your actions in exploring the trace will be recorded. You can also write a comment about the actions you just performed or you are going to perform by selecting the "Add script message" option in the "File" menu. A window will appear in which you can type your comment. Press "Ok" when you complete your comment and continue exploring the trace.

You end script recording by selecting the "End script" option in the "File" menu.

7.4.2 Run a script

To run a script select the "Run script" option in the "File" menu. A file selector box will ask you for the file name of the script.

Subsequently a window will appear. At the bottom of the window are four buttons:

Step Moves the script one step further. Every step performs a single action in the browser window, such as selecting an expression or showing a parent.

Run Steps automatically through the script, with a short time interval between each step.

Pause Interrupts a running script.

Done Finishes the script, the browser resumes normal operation.

Note that when a script is active, you cannot manually explore trails.

7.5 Further features

7.5.1 Select a font for the trace

You can select the font in which the trace is displayed by selecting the "Select font" option in the "Options" menu. A dialogue appears in which you can choose the font face, the style and the size. Note that you have to press "Enter" or "Return" to change the size. The effect of your choice is shown in the dialogue. You commit your choice by selecting "Ok".

7.5.2 The Help menu

The Help menu offers short explanations of the main features of hat-trail, similar to the quick reference of Section 7.7.

7.6 Some practical advice

- First-time users of hat-trail tend to quickly unfold large parts of the trace and thus clutter the screen and get lost. Think well, before you demand to see another parent. It is seldom useful to follow a long sequence of parents for whole redexes. Do not forget that you can ask for the parent of any subexpression. Choose the subexpression that interests you carefully. When locating an error, a wrong subexpression of an argument is a good candidate for further enquiry.

  In our experience usually less than 10 parents need to be viewed to locate an error, even in large programs.

- Use the links to the source as described in Section 7.2.5. The trace is designed to be of minimal size. The source gives valuable context information.

- Use the various forms of highlighting described in Section 7.3. The information conveyed by highlighting often makes viewing a parent superfluous.

- Avoid λ-abstractions in your program. Informative function names are very helpful for tracing.

7.7 Quick reference

A mouse click on a subexpression S in the trace panel has the following effect:

left	fold/unfold trace show the parent redex of S, if any; *or*, if the parent is already on display, remove it along with any of its ancestors also on display
middle	fold/unfold expression if S is a place-holder, expand it; *or*, if not, contract S to a place-holder
right	show source reference show where S was created in the source program, displayed in the source code panel.
shift-right	show where S is defined in the source program, displayed in the source code panel. (only for names, not arbitrary expressions)

Moving the mouse over expressions in the trace panel causes highlighting of expressions in various ways:

surrounded by red box	currently-selected expression
in blue text	expression with the same parent as the currently-selected expression
with yellow background	parent redex of the currently-selected expression (if it is on display)

Beyond the normal syntax for Haskell expressions, five special symbols may occur in trace expressions:

⊥ the undefined value, as usual;

□ a placeholder for a subexpression suppressed for the time-being (e.g. to avoid over-wide displays);

⍁ a placeholder for an expression that is not available because it is part of a *trusted* computation not recorded in the trace – however, the parent redex is available;

⊠ a placeholder for an expression that is not available – should rarely occur;

◁ shown between control-flow information for `case`, conditions or guards and the redex they belong to; it is pronounced 'within'.

8 Limitations of Functionality

Although Hat can trace nearly any Haskell 98 program, some program constructs are still only supported in a restricted way. See the Hat web page for further limitations and bugs.

8.1 Input/Output

Programs can use all standard IO actions, but in the trace the internal implementation of IO sometimes shows up. Hence the viewing tools sometimes show obscure expressions involving a data constructor `IO`.

8.2 List Comprehensions

List comprehensions are desugared by Hat, that is, their implementation in terms of higher-order list functions such as `foldr` is traced.

8.3 Labelled Fields (records)

Expressions with field labels (records) are desugared by Hat. So viewing tools show field names only as selectors but never together with the arguments of a data constructor. An update using field labels is shown as a `case` expression.

8.4 Strictness Flags

Strictness flags in data type definitions are ignored by Hat and hence lose their effect.