Hat – The Haskell Tracer
Version 1.12
Users' Manual

The ART Team

14 March 2002

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 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 computation, 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 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.

Hat is an offline tracer: First the specially compiled program runs as normal, except that additionally a trace is written to file. Second, after the computation has terminated, the trace is viewed with a number of browsing tools.

Hat can be used for computations that terminate normally, that terminate with an error message or that are interrupted by the programmer (because they do not terminate).

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-observe, hat-trail, hat-detect, and hat-stack – 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, either with `nhc98` or `ghc`, and then run.

2.1 Compilation with nhc98

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 computations use more heap space. As a rough rule of thumb, traced computations require 3 times as much heap space as untraced ones. However, because traced computations 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 computations is 400KB and for traced computations 2MB. For example, you can set the heap size at compile (link) time with `-H10m` or for a specific computation 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 Compilation with ghc

Before compiling a program, all its modules must be transformed to tracing versions with the pre-processor `hat-trans`. This preprocessor generates a new module (prefixed with the letter 'T') for each original module. Compile and link the generated modules in the normal way using `ghc` with the extra option `--package hat`.

The `hat-trans` pre-processor generates and reads its own special kind of module interface files (`.hx` files) and therefore modules must be transformed in the same dependency order as normal compilation. Hence, it is often easier simply to let `hmake` do all the work: `hmake -ghc -hat` does all the transformation, compiling, and linking automatically.

2.3 Computation

The traced computation behaves exactly like the untraced one, except that it is slower (currently about 50 times slower in nhc98, 300 times slower in ghc), 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 computation (normal termination or caused by error or interrupt) you can explore the trace with any of the programs described in the following sections.

The computation 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 computation 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.

2.4 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 for tracing as normal but with the extra option `-trusted`. (A plain object file 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.

3.1 Arguments in Most Evaluated Form

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 show the function application as `length [1+5,2+5]` or `length [_,_]` if the list elements are unevaluated.

3.2 Special Expressions

Unevaluated expressions Tools do not usually show non-value subexpressions. The underscore _ represents these unevaluated expressions. (The 'verbose' option can be set interactively if you wish to replace underscores with the full representation of the unevaluated expression.)

λ-abstractions A λ-abstraction, as for example `\xs-> xs ++ xs`, is represented simply by `(\..)`.

The undefined value $\perp$ 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 $\perp$ (`_|_` in ASCII form).

A typical case where we obtain $\perp$ 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 aborts with an error message saying that a black hole has been detected. The trace of the computation contains several $\perp$'s.

Trusted Expressions The symbol {_} is used to represent an expression that was not recorded in the trace, because it was trusted.

3.3 Combination of Viewing Tools

Each tool gives a unique view of a computation. These views are complementary and it is productive to use them together. From each of the three tools hat-observe, hat-trail and hat-detect you can at any time change to any of the other two tools, starting there at exactly the point of the trace at which you left the other tool. So after using one tool to track a bug to a certain point you can change to another tool to continue the search or confirm your suspicion.

3.4 The Running Example

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-Observe

Hat-observe enables you to observe the value of top-level variables, that is, functions and constants. Hat-observe shows all reductions of a variable that occurred in the traced computation. Thus for a function it shows all the arguments with which the function was called during the computation together with the respective results.

It is possible to use hat-observe in batch-mode from the command line, but the main form of use is as an interactive tool. The interactive mode provides more comprehensive facilities for filtering the output than batch mode.

4.1 Starting & Exiting

To start hat-observe as an interactive tool, simply enter

`hat-observe` *prog*`[.hat]`

at the command line, where *prog* is the name of the traced program.

4.2 The Help Menu

Enter `:h` (`:help`) to obtain a short overview of the commands understood by hat-observe. All commands begin with a '`:`', and can be shortened to any prefix of the full name.

4.3 Observing for Beginners: Using the Wizard

If you use hat-observe for the first time, you might want to start by using the observation *wizard*. Simply enter the command `:observe` with no other arguments. The tool will then ask questions about the reductions you are interested in. Eventually, it will show the resulting query and start the observation. This way you can quickly learn what queries look like.

4.4 Making Simple Observations

Observations of a function are made with the `:observe` command, or for simplicity, just by entering the name of the function at the prompt. For instance, enter `:observe` *f*, or simply *f*, to obtain all reductions of *f*.

To avoid redundant output, equivalent or less general reductions of the identifier are omitted in the display. A reduction of an identifier is considered more general than another if all its arguments on the left-hand-side are less defined (due to lazy evaluation) and/or if its result on the right-hand-side is more fully defined.

4.5 Exploring What to Observe

If you forgot the correct spelling of a function identifier you want to observe or you do not know the program well, you may want to see a list of all function identifiers which can be observed. With the `:info` command you can browse the list of all top-level function identifiers which were used during the computation.

4.6 Filtering Reductions

Although only the most general reductions are shown, some observations may still result in an excessively large number of unique reductions. You only want to see those reductions in which you are particularly interested. There are several ways to decrease the number of reductions shown.

4.6.1 Non-Recursive Mode

Hat-observe can omit recursive calls of the given function. If all the top-most calls of a function are correct, then all its recursive calls within the function itself are *likely* to be correct as well. If there are any erroneous recursive calls, their incorrect behaviour at least had no effect on the result of the top most calls. To omit recursive calls of a function, the `:set recursive off` command may be used. To see recursive calls again, use `:set recursive on`.

4.6.2 Generalise Equations

A function may be called several times with the same arguments. Hat-observe alwyas 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, then the display of the application to the less general arguments can be omitted. Use the `:set generalise on` command to omit the less general equations.

4.6.3 Observing Calls from a Specific Function

Another way to restrict the number of reductions being observed is by observing only calls made from within a specific calling function. If you are interested in all calls of `map` from the function `myMapper`, try `:observe map in myMapper`.

4.6.4 Specifying Reductions with a Pattern

You can significantly reduce the number of observed applications by observing only reductions that are instances of a given pattern. With a pattern you can specify in which reductions you are particularly interested.

You can enter a pattern for the whole equation or any prefix of it. A pattern for an equation consists of a pattern for the left-hand-side followed by a `=` and a pattern for the result. The `=` and result pattern may be omitted, as may any of the trailing argument patterns.

If you wish to skip one argument in the pattern, use an underscore. An underscore `_` in a pattern matches any expression, value, or unevaluated. The bottom symbol `_|_` may also be used in patterns, and matches only unevaluated things.

Examples:

- To see all applications of `map` where its first argument is `foo`, enter `:observe map foo`. However, to see all applications of `map` where its *second* argument is `foo`, enter `:observe map _ foo`.
- To see all applications of `filter` using first argument `odd` and resulting in an empty list, enter `:observe filter odd _ = []`.

Attention: infix patterns are not currently supported. Always use the prefix form of function/constructor applications, and enclose each application in parentheses.

Special syntax for strings and lists is supported, e.g. `"Hello world!"` for a string and `[1,2,42]` etc. for lists.

4.6.5 Combination of Filters

Of course, all methods previously described can be mixed with each other, as in the following examples.

```
:observe (map _ [1,2,3]) in myMapper
:observe (filter even (:  1 _)) = _|_ in myFunction
:observe (fibiterative _ _) = 0
```

4.7 Verbose Mode

The command `:set verbose on` can be used to switch to a mode where unevaluated expressions are shown in full, rather than as an underscore.

4.8 Browsing a List of Reductions

After successfully submitting a query in any of the described ways, the tool searches the given trace file. Depending on the size of the file and the number of reductions found, the search may take a considerable time. Progress will be indicated during the scan of the file. After the scan of the file, additional time might be spent on filtering the most general reductions matching the given pattern.

The first n (default 10) observed reductions are then displayed. More reductions can be displayed by pressing the `RETURN` key. The system indicates the availability of additional equations by prompting with `--more-->` instead of the usual command prompt. If more equations are available but you do not wish to see them, typing anything except the plain

RETURN key will cause you to leave the equation display mode and go back to the normal prompt.

The number of equations displayed per group can be altered by using the `:set group` n command. The default is 10 reductions at a time. The reductions are numbered - this is to facilitate selection of an equation for use within the other hat tools.

Attention: because hat-observe uses lazy evaluation to determine the list of the most general reductions, there may be a delay during which more reductions are determined.

4.9 Display of Large Expressions

Sometimes expressions may contain very large data structures which clutter the display. In order to cope with them the cutoff depth of the display can be adjusted. This cutoff value determines the nesting depth to which nested sub-expressions are printed: any subexpression beyond this depth is shown as a dark square. The cutoff depth is adjusted using the command `:set cutoff` n.

In certain circumstances, you simply want to increase or decrease the cutoff by a small amount. There are 'shortcut' commands `:+` n and `:-` n to increase or decrease the cutoff by n respectively. If n is omitted, then it is assumed to be one.

A data structure may be infinite. Because an *infinite* data structure is the result of a *finite* computation, it must contain a cycle. The following example demonstrates how such a cycle is shown.

```
cyclic = 1:2:3:4:cyclic
main = putStrLn (show (take 5 cyclic))
```

If you observe `cyclic`, then you obtain

```
cyclic = (cyc1 where cyc1 = 1:2:3:4:cyc1)
```

4.10 Invoking other Viewing Tools

You may eventually find an erroneous reduction. There are several ways in which you can proceed at this point.

The first way is to start observing functions used in the definition body of the erroneous function. You will need to check the source code for functions which might have caused the wrong result. If you suspect a function f to have caused the incorrect behaviour of g, it is a good idea to try `:observe` f `in` g.

A second way to proceed is to switch to the Algorithmic Debugging tool `hat-detect` at this point. The command `:detect` n starts a separate `hat-detect` session for equation number n in a new window (currently only works under Unix). See section 6 for information on `hat-detect`.

Finally you have the choice to use `hat-trail` on a reduction you have observed. Use the command `:trail` n to start a separate instance of `hat-trail` for equation number n.

4.11 Quick reference to commands

All the commands that are available in hat-observe are summarised in the following table.

```
--------------------------------------------------------------------------
 <query>            observe the named function/pattern
 <RETURN>           show more observations (if available)
 :observe <query>   observe the named function/pattern
 :info              see a list of all observable functions
 :detect <n>        start hat-detect on equation <n>
 :trail <n>         start hat-trail browser on equation <n>
 :set               show all current mode settings
 :set <flag>        change one mode setting
   <flag> can be: verbose [on|off]:    unevaluated expressions shown in full
                  generalise [on|off]: show only most general equations
                  recursive [on|off]:  show recursive calls
                  group <n>:           number of equations listed per page
                  cutoff <n>:          cut-off depth for deeply nested exprs
 :+[n]                short-cut to increase cutoff depth by <n> (default 1)
 :-[n]                short-cut to decrease cutoff depth by <n> (default 1)
 :help              show this help text
 :quit              quit
--------------------------------------------------------------------------
```

5 Hat-Trail

Hat-trail is an interactive tool that enables you to explore 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*.

Using the symbol ← to represent the relationship "is the parent of", here is an illustrative list showing the parent of every subexpression from the example in Section 3.

`last' []` ← the error message
`last' (_:[])` ← `last' []`
`last' (_:_:[])` ← `last' (_:[])`
`last' (8:_:_:[])` ← `last' (_:_:[])`
`main` ← `last' (8:_:_:[])`
`4*2` ← `8`
`xs` ← `4*2`

Every subexpression (if it is not a top-level constant such as `main`) has a parent. In the example the parent of `(8:_:_:[])` is `xs`. The parent of each subexpression in an expression can be different from the parent of the expression itself.

5.1 Starting & Exiting

Start hat-trail by entering

```
hat-trail prog[.hat]
```

at the command line, where *prog* is the name of the program (the extension `.hat` is optional).

You can quit this browser at any time by typing the command *:quit*.

5.2 The Help Menu

The *:help* command offers short explanations of the main features of hat-trail, similar to the quick reference of Section 5.7.

5.3 Basic Exploration of a Trace

The browser window mainly consists of two panes:

- The program output (and error) pane.
  Here you can select a part of the program output (or an error message, if there was one), to show its parent redex in the trace pane for further exploration.
- The trail pane.
  This is the most important pane. In it you explore the trace. With the cursor keys you request information about different parts of the trace. Coloured highlighting is used to show the current and previous selections.

You can pop up a further, very important, window on demand:

- The source code window.
  Here part of the source code of the traced program is shown. In the trail pane, you can ask to see a specific point in the source code (e.g. where exactly a function in a particular expression was applied), and the cursor in the source code window is placed at the relevant site in the appropriate source file. This is not an editor window, just a viewer – type any key to close it.

5.3.1 The program output (and error) pane

Any output (or error message) produced by the traced program is shown in the top 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 with the cursor keys: left/up and right/down. The selected section is shown with a coloured highlight. If the output is very large, only a portion of it is displayed at a time – moving left/up at the top of the screen, or right/down at the bottom, 'pages' through the output. Press the Return key to start exploring the parent redex for the selected section in the lower trail pane.

5.3.2 The trail pane

Within the trail pane, the display shows a simple 'stack' of parent expressions, one per line. Each expression line is the parent of the highlighted subexpression on the line before it. Within a line, you can navigate to a subexpression using the cursor keys by one of two methods, described below. Pressing the Return key asks for the parent of the currently selected subexpression, and it is shown on a new line. Pressing the Delete or Backspace key removes the current line and goes back to the previous selection in the stack.

Selecting a subexpression in the trail pane There are two methods of navigating within an expression to highlight a specific subexpression. The simplest method just uses the right and left cursor keys. Repeatedly pressing the right cursor key follows a pre-order traversal of the underlying expression tree. Thus, first an application is highlighted, then the function part, then each argument, and so on recursively depth-first. The left cursor key follows the reverse order.

Alternatively, you can navigate by explicit levels within the tree. The up cursor key moves outwards from a subexpression to the surrounding expression. The down key moves inwards from an application to the function position. The ‘¡’ and ‘¿’ keys move left and right amongst subexpressions at the same level, for instance between a function and its arguments.

Folding away part of an expression When you are looking at a large expression, it is sometimes difficult to see its gross structure because of all the detail. At these times, it is helpful to be able to shrink certain subexpressions to hide the detail. This is represented in the display as a small dark box.

You can explicitly shrink any selected subexpression to a dark box, or expand a selected box to its full representation, with the ‘-’ and ‘+’ keys.

Like in the other tools, there is a standard cutoff depth for deeply nested expressions. The automatically cutoff expression is denoted by the same dark box as a manually hidden expression. Use the `:set cutoff` *n* command to change the cutoff depth, or the shortcuts `:+` *n* and `:-` *n* to increase and decrease the cutoff depth – if *n* is omitted, the cutoff is increased or decreased by one.

5.3.3 The source code window

Most functions and constants are used at more than one position in the program. Hence, when viewing an expression in hat-trail, it can be very helpful to know exactly which application site is under examination. There are a number of direct links to the source code available.

First, the filename, line, and column number of the currently selected application or value is always visible in a status line at the top of the trail pane. You can use this to help you navigate within your preferred editor or viewer.

Secondly, the command `:source` pops up a simple viewer with the cursor sitting directly over the application site in the relevant file. The command can be abbreviated to `:s`, and you may find that this is often quicker and more convenient than using an external editor or viewer.

Thirdly, if you want to look at the *definition* of the selected function or constant rather than its individual application site, the command `:Source` again pops up the simple viewer, this time with the cursor on the definition line.

5.3.4 Special syntax

We have already mentioned that a dark box represents a subexpression that has been hidden from display, either due to automatic cutoff of deep nesting, or by explicit request of the user.

There are a number of other special syntactic representations in the display.

Lists A list where some elements are undefined or unevaluated is displayed as a sequence of nested applications of the normal list constructor `(:)`. However, fully evaluated lists are displayed using the more compact syntactic sugar of square brackets with elements separated by commas.

Strings Fully evaluated character strings are displayed differently again. A string is usually shown using the Haskell lexical convention of double quotes, for example `"Hi"`. However this representation makes it slightly more difficult to select a substring...

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.

A control-flow expression of this nature is shown in the trail as the value of the guard, condition or case discriminant, placed to the right of the expression within which it belongs and separated from it by a bar and a highlighted keyword, e.g. `| if False` or `| case EQ`.

Strictly speaking, the expression to the left of the bar is the parent of the expression on the right, but the tool displays them together for clarity since the guard, condition, or case makes most sense when understood in the context of its parent.

For example, in the program

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

main = print (abs 42)
```

the parent of the result value `42` is

```
abs 42 | False | True
```

This redex display 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`. You may ask for the parent of `False` and learn that it was created by the redex `42 < 0`.

Trusting Section 2.4 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 with the symbol {_}. It cannot be expanded like a dark box representing a cutoff expression, but it does have 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:{_})`.

Unevaluated expressions Unevaluated expressions are shown by default with the underscore symbol (_). Show you wish to see these expressions in full, you should switch on the verbose option with the command `:set verbose on`.

5.3.5 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.

5.4 Advanced Exploration of a Trace

5.4.1 Shared expressions

When you select a subexpression in the trail pane, sometimes not only this expression is highlighted but also some other occurrences of the subexpression. The reason is that the marked occurrences are shared. That is, they are not just equal, but they actually share the same space in memory. This operational observation can often help you to understand a computation.

5.5 Invoking other Viewing Tools

It is possible to invoke hat-observe and hat-detect immediately from hat-trail.

- The command `:observe` launches a new window with the hat-observe tool, which immediately searches for all applications of the currently selected function throughout the computation.
- The command `:location` launches a new window with the hat-observe tool, which immediately searches for applications of the currently selected function, but only at the same source location as the current selection. This is useful to narrow down your exploration to a specific site of interest.
- The command `:detect` launches a new window with the hat-detect tool, restricting the debugging algorithm solely to the currently selected equation and all its dependents.
- The command `:trail` starts a new window with a fresh instance of the hat-trail tool starting with the current selection. This can be useful if there are several redex trails you wish to explore and compare side-by-side.

5.6 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 8.3.5. The trail display is designed to be concise, so the source viewer gives valuable context information.
- Avoid λ-abstractions in your program. Informative function names are very helpful for tracing.

5.7 Quick reference to commands

All the commands that are available in hat-trail are summarised in the following table.

```
---------------------------------------------------------------------------
 cursor keys   movement within current expression
 < and > keys  movement within current expression
 RETURN        show parent expression of selected expression
 BACKSPACE     remove most recently-added expression/equation
 -/+           shrink/expand a cutoff expression
 ^L            repaint the display if it gets corrupted
 ^R            repaint the display after resizing the window
 :source       look at the source-code application of this expression
 :Source       look at the source-code definition of current function
 :observe      use hat-observe to find all applications of this function
 :location     use hat-observe to find all applications at this call site
 :trail        start a fresh hat-trail with the current expression
 :detect       use hat-detect to debug the current expression
 :set          show all current mode settings
 :set <flag>   change one mode setting
   <flag> can be: verbose [on|off]:   unevaluated expressions shown in full
                  equations [on|off]: show equations, not just redexes
                  cycles [on|off]:    cope with cyclic structures
                  cutoff <n>:         cut-off depth for deeply nested exprs
 :+[n]         shortcut to increase cutoff depth
 :-[n]         shortcut to decrease cutoff depth
 :help  :?     show this help text
 :quit         quit
---------------------------------------------------------------------------
```

6 Hat-Detect

Hat-detect is an interactive tool that enables you to locate semi-automatically an error in a program by answering a sequence of yes/no questions. Each question concerns a reduction. 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.

6.1 Limitations

At the moment hat-detect does not handle IO actions properly. It can only handle computations that perform a *single* primitive *output* action such as `putStr` or `print`. Monadic binding operators (or do-notation) and input actions such as `read` lead to confusion.

Hence the recommended usage of hat-detect is to first use hat-observe to locate an erroneous reduction that does not involve IO and then to invoke hat-detect for this reduction as described in Section 4.10.

Also, currently hat-detect can only be used for computations that produce faulty output, not for computations that abort with an error message or are interrupted (in the latter cases hat-detect may indicate a wrong error location).

6.2 Starting & Exiting

Start hat-detect by entering

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

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

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

6.3 The Help Menu

Enter `:help` to obtain a short overview of the commands understood by hat-detect.

6.4 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:[]))   ? n
2  sort (3:2:1:[]) = 3:3:3:[]   ? n
3  insert 1 [] = 1:[]   ? y
4  insert 2 (1:[]) = 2:2:[]   ? n
5  insert 2 [] = 2:[]   ? y

Error located!
Bug found: "insert 2 (1:[]) = 2:2:[]"
```

The first question of the session asks if the reduction of `main` is correct. Hat-detect indicates that `main` is reduced to an IO action, and shows the action. The answer is obviously *no*. Further answers from the user show that the third and the fifth reductions are correct, whereas the second and fourth are not.

Note that hat-detect does not ask about any reductions of `foldr` here, mainly because it is trusted.

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

6.4.1 Postponing an Answer

If you are not sure about the answer to a question you can answer *?n* or *?y*. If you answer *?n*, then 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. Answering *?y* will postpone the question as well, but hat-detect will proceed as if the answer hat been *yes*. If it cannot locate an error in one of its brother reductions, then it will ask you the question again.

6.4.2 Unevaluated Subexpressions

Reductions may contain underscores _ that represent unevaluated subexpressions. A question with an underscore on the left-hand side of the reduction has to be read as "is the reduction correct for *any* value at this position?" and a question with an underscore on the right-hand side should be read as "is the reduction correct for *some* value at this position?". If there are several underscores in a reduction the values at these positions need not be the same.

6.5 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.

6.6 Advanced Features

6.6.1 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.

6.6.2 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 the `:set verbose on` command you can switch to verbose mode which shows these unevaluated subexpressions in full. Use `:set verbose off` to switch the verbose mode off again.

6.6.3 Going back to a question

The questions are numbered. By entering a number *n* you can go back to any previous question numbered *n*. When you do this, the answers to all intervening questions are deleted.

6.6.4 Trusting

Hat-detect does not ask any question about the reductions of functions that are trusted as described in Section 2.4. However, you can trust further functions and thus avoid questions about them. By entering `:trust` instead of `y` when being asked about a specific reduction of a function you trust this function. By entering `:untrust` you stop trusting *all* these functions again.

6.6.5 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 answered before.

You can turn memoisation on/off with the command `:set memoize on` or `:set memoize off`.

6.6.6 Invoking other Viewing Tools

Observing a function When being asked about a specific reduction of a function you can enter `:observe` to observe the function. The hat-observe tool will appear in a new window, showing all applications of the given function. This interface to hat-observe is particularly useful, if you are not sure whether to trust a function for Algorithmic Debugging. By observing all applications of the function you can decide whether the function can indeed be trusted or not. If you find an erroneous reduction in the observation, you can select it and in turn start a new Algorithmic Debugging session for this reduction.

Tracing arguments When `hat-detect` asks you about the reduction of an application, which obviously has a *wrong* argument, you should consider using `hat-trail` to investigate where this argument came from. By answering a question with `:trail` the Redex Trail browser is launched immediately from the Algorithmic Debugger.

6.7 Quick reference to commands

All the commands that are available in hat-detect are summarised in the following table.

```
---------------------------------------------------------------------------
 y   or  yes      you believe the equation is ok
 n   or  no       you believe the equation is wrong
 ?y  or  y?       you are not sure (but try ok for now)
 ?n  or  n?       you are not sure (but try wrong for now)
 <n>              go back to question <n>
 :set             show all current mode settings
 :set <flag>      change one mode setting
      <flag> can be: memoise [on|off]:  never ask the same question again
                     verbose [on|off]:  show unevaluated exprs in full
                     cutoff <n>:        set subexpression cutoff depth
 :observe         start hat-observe on the current function
 :trail           start hat-trail on the current equation
```

```
 :trust            trust all applications of the current function
 :untrust          untrust ALL functions which were previously trusted
 :help             show this help text
 :quit             quit
---------------------------------------------------------------------------
```

7 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.

7.1 Usage

To use hat-stack enter

`hat-stack` *programname*

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

7.2 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)
```

7.3 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 relationships between the stack elements.

8 Hat-Trail-In-Java

Hat-trail-in-java is now obsolete. It is an interactive tool very similar to hat-trail, but significantly slower and more complex. Like hat-trail, it enables you to explore 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-in-java 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.

8.1 Starting & Exiting

Start hat-trail-in-java by either entering

`hat-trail-in-java` *prog*`[.hat]`

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

```
hat-trail-in-java
```

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

At any time you can use "Open trace file" to view the trace of a different computation.

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

8.2 The Help Menu

The Help menu offers short explanations of the main features of hat-trail-in-java, similar to the quick reference of Section 8.9.

8.3 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.

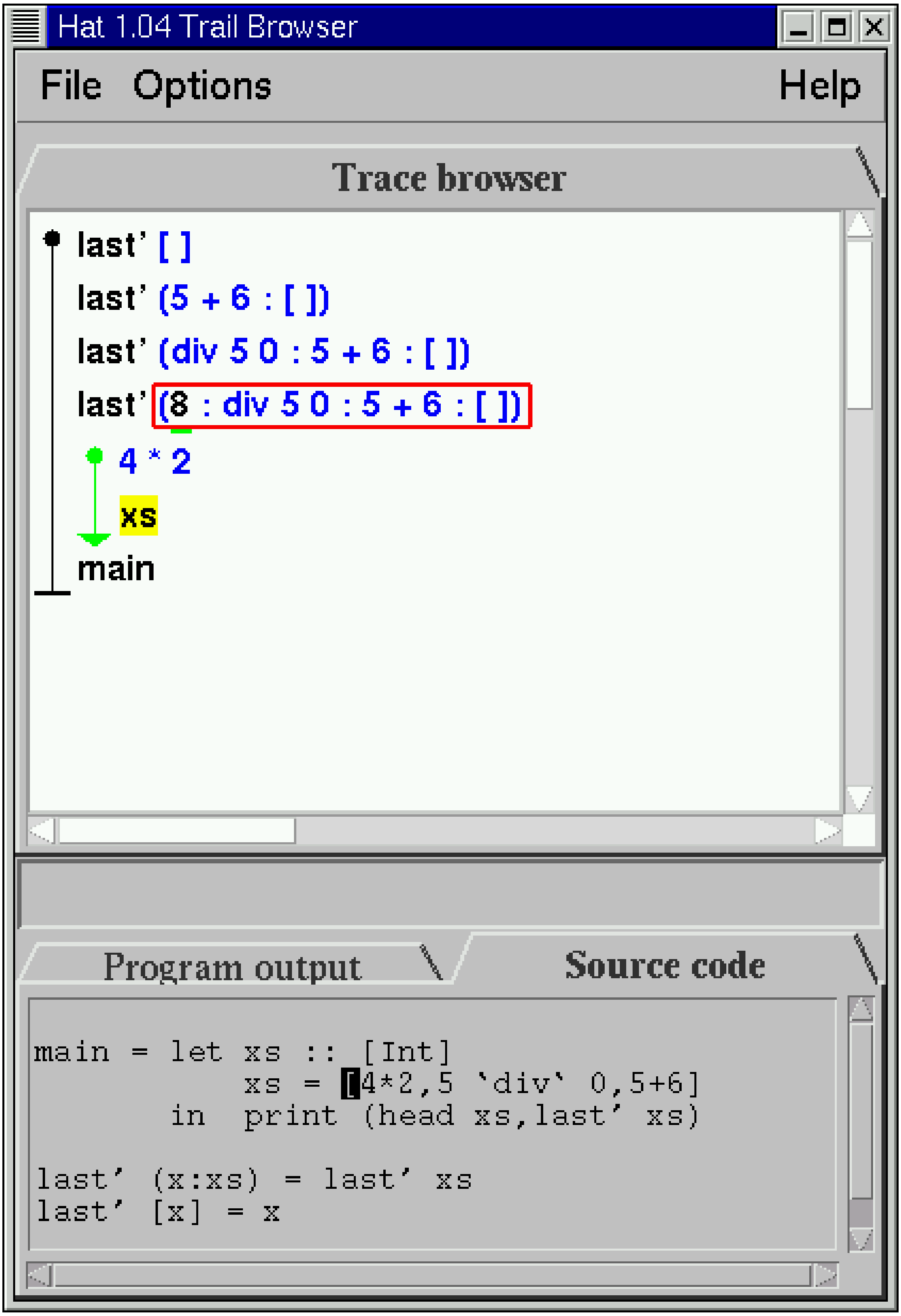

Figure 1: hat-trail

- 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.

8.3.1 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.

8.3.2 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. For example, 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.

8.3.3 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 belongs 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.

8.3.4 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.

8.3.5 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.

8.3.6 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.

When strings are very long, everything to their right can only be reached by cumbersome scrolling. In the menu "Options" you can select the item "Choose string-length limit" to set an upper boundary for the length of a string. Abbreviated strings are indicated by dots (...) in their middle. These abbreviated strings can still be expanded as described in the preceding paragraph.

8.3.7 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 the "Options" menu you can choose "Show case/guard/if expression" to augment redexes with control-flow information. An augmented redex is of the form:

function application ▷ *control-flow*$_1$ ▷ … ▷ *control-flow*$_k$

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 "and within that". For example, for the program

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

main = print (abs 42)
```

the parent of the result value `42` is

```
abs 42 ▷ | False ▷ | True
```

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`. You may ask for the parent of `False` and learn that it was created by the redex `42 < 0`.

8.3.8 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.

8.4 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.

8.4.1 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.

8.4.2 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.

8.4.3 Trusting

Section 2.4 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:⍁)`.

8.5 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.

8.5.1 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.

8.5.2 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.

8.6 Further Features

8.6.1 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".

8.7 Invoking other Viewing Tools

It is possible to invoke hat-detect and hat-observe immediately from hat-trail-in-java.

- A double click with the right mouse button on an identifier immediately launches a new window with the hat-observe tool. It allows to observe all applications of the chosen identifier.
- A double click with the right mouse button on a redex launches a separate window with the hat-detect Algorithmic Debugging tool. Hat-detect then allows to analyse the evaluation of the chosen redex.

8.8 Some practical advice

- First-time users of hat-trail-in-java 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 8.3.5. The trace is designed to be of minimal size. The source gives valuable context information.
- Use the various forms of highlighting described in Section 8.4. 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.

8.9 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:

$\perp$ the undefined value, as usual;

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

$\boxslash$ 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;

$\boxtimes$ a placeholder for an expression that is not available – should rarely occur;

$\triangleright$ shown between a redex and its control-flow information for `case`, conditions or guards; it is pronounced "and within that".

9 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.

9.1 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`.

9.2 List Comprehensions

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

9.3 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.

9.4 Strictness Flags

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