Hat – The Haskell Tracer
Version 2.00
Users' Manual

The ART Team

14 June 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, using `hmake` with either `nhc98` or `ghc`, and then run.

Compile the program using `hmake` and the `-hat` option; you may want to choose your compiler as well, e.g. `hmake -ghc -hat`.

What `hmake` does is this: all the modules of the program are 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. The generated modules are compiled and linked using an ordinary compiler with the extra option `-package hat`. The `hat` package contains interface files and a link-library that are needed by the transformed program.

You can invoke `hat-trans` and the compiler manually if you wish, but `hat-trans` 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 much easier simply to let `hmake` do all the work.

Note that the `hat-trans` preprocessor does not insert the complete file paths of the original source modules into the generated modules. The trace viewers assume that the source modules are in the same directory as the executable.

2.1 Compilation with nhc98

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 default heap size for an untraced program compiled by `nhc98` is 400KB; you will probably want to increase this to at least 2MB. For example, you can set the heap size at compile (link) time with `+CTS -H10m -CTS` or for a specific computation with `+RTS -H10m -RTS` to a ten megabyte heap.

2.2 Computation

The traced computation behaves exactly like the untraced one, except that it is slower (currently about 100 times slower in nhc98, 200 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.

2.3 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 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 'uneval' 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 reductions of the identifier are omitted in the display (`:set unique`). You can change this behaviour in order to see all reductions, even identical ones (`:set all`). In future there will also be an option to see only the most general reductions. 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, and how many times they were used.

4.6 Filtering Reductions

Even when only unique reductions are shown, some observations may still result in an excessively large number of displayed equations. 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 Observing Calls from a Specific Context

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`.

4.6.3 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 _ = []`.

Infix patterns are also supported, although the fixity and priority of the operator is not necessarily known, so always use explicit parentheses around such patterns.

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

4.6.4 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 Modes

There are several modes determining the relative verbosity of the output. `:set uneval on` shows unevaluated expressions in full, rather than abbreviating them to underscores. `:set strSugar off` turns off string sugaring, and `:set listSugar off` turns off sugaring for other kinds of list: in both cases, the effect is to reveal the explicit cons and nil structures.

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 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*.

Alternatively, 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>
 :source <n>        show the source application for equation <n>
 :Source <n>        show the source definition for identifier in eqn <n>
 :set               show all current mode settings
 :set <flag>        change one mode setting
   <flag> can be: uneval [on|off]      show unevaluated expressions in full
```