Draft Hat Tutorial: Part 1

Colin Runciman

March 15, 2002

1 Introduction

This tutorial is intended as a practical introduction to the Hat tools[1] for tracing Haskell 98 programs. It introduces the basic ideas and explains with worked examples how to use the tools.

Readers are encouraged to follow the tutorial using an installation of Hat. This first version of the tutorial assumes nhc98 (Version 1.12), hmake (Version 3.02) and Linux.

There are several Hat tools for examining traces, but the tutorial will consider only the two used most: `hat-trail` and `hat-observe`. Even for these tools not every option and command is discussed. For a more comprehensive reference see the Hat User Manual.

The tutorial makes use of a small example program — at first a correctly working version, later one with faults deliberately introduced. The intended behaviour of the program is very simple: it should sort the letters of the word 'program' using insertion sort. The working program is given[2] in Figure 1.

2 Hat Compilation and Execution

To use Hat, the Haskell program to be traced must first be compiled with the `-T` option. The `-T` flag is interpreted suitably by `hmake`:

```
$ hmake -T Insort
nhc98 -T -c -o Insort.T.o Insort.hs
nhc98 -T -o Insort Insort.T.o
```

A program compiled for tracing can be executed just as if it had been compiled normally.

```
$ Insort
agmoprr
```

The main difference from untraced execution is that as `Insort` runs it records a detailed trace of its computation in a file `Insort.hat`. The trace is a graph of program expressions encoded in a special-purpose binary format.

Two further files `Insort.hat.output` and `Insort.hat.bridge` record the output and associated references to the trace file.[3]

[1] Available from `http://www.cs.york.ac.uk/fp/hat/`.

[2] The program may also be found in the file `Insort.hs`.

[3] Trace files do not include program sources, but they do include references to program sources; modifying source files could invalidate source links from traces.

```
sort ::  Ord a => [a] -> [a]
sort []      =  []
sort (x:xs)  =  insert x (sort xs)
insert ::  Ord a => a -> [a] -> [a]
insert x []      =  [x]
insert x (y:ys)  =  if x <= y then x :  y :  ys
                    else y :  insert x ys
main  =  putStrLn (sort "program")
```

Figure 1: An insertion sort program.

3 Hat-trail: Basics

After a program compiled for tracing has been run, creating a trace file, special-purpose tools are used to examine the trace. The first such tool we shall look at is hat-trail. The idea of hat-trail is to answer the question 'Where did that come from?' in relation to values, expressions, outputs and error messages. The immediate answer will be a parent application or name. More specifically:

- *errors*: the application or name being reduced when the error occurred (eg. `head []` might be the parent of a pattern-match failure);
- *outputs*: the monadic action that caused the output (eg. `putStr "Hello world"` might the parent of a section of output text);
- *non-value expressions*: the application or name whose defining body contains the expression of which the child is an instance (eg. `append [1,2] [3,4]` might be the parent of `append [2] [3,4]`);
- *values*: as for non-value expressions, or the application of a predefined function with the child as result (eg. `[1,2]++[3,4]` might be the parent of `[1,2,3,4]`).

Parent expressions, and their subexpressions, may in turn have parents of their own. The tool is called hat-trail because it displays trails of ancestral redexes, tracing effects back to their causes.

Hat-trail sessions and requests

A hat-trail session can be started from a shell command line, or from within existing sessions of hat tools. The immediate result of the shell command

```
$ hat-trail Insort
```

is the display of a terminal window with an upper part headed `Output` and a lower part headed `Trail`:

```
Output: ------------------------------------------------------
agmoprr

Trail: -------------------------------------------------------
```

The line of output is highlighted[4] because it is the current selection.

Requests in `hat-trail` are of two kinds. Some are single key presses with an immediate response; others are command-lines starting with a colon and only acted upon when completed by keying return. A basic repertoire of single-key requests is:

return	add to the trail the parent expression of the current selection
backspace	remove the last addition to the trail display
arrow keys	select (a) parts of the output generated by different actions, or (b) subexpressions of expressions already on display

And a basic repertoire of command-line requests is:

`:source`	show the source expression of which the current selection is an instance
`:quit`	finish this hat-trail session

It is enough to give initial letters, `:s` or `:q`, rather than `:source` or `:quit`.

Some `Insort` trails

To trace the output from the `Insort` computation, keying return alters the `Trail` part of the display to:

```
Trail: ----------------------- Insort.hs line: 10 col: 8 ------
<- putStrLn "agmoprr"
```

The source reference is to the corresponding application of `putStrLn` in the program. Giving the command `:s` at this point creates a separate source window showing the relevant extract of the program.[5]

Back to the `Trail` display. Keying return again:

```
Trail: ----------------------- Insort.hs line: 10 col: 1 ------
<-putStrLn "agmoprr"
<-main
```

That is, the line of output was produced by an application of `putStrLn` occurring in the body of `main`.

So far, so good; but what about the sorting? How do we see where `putStr`'s string argument `"agmoprr"` came from? By making that string the current selection and requesting its parent:

backspace	(removes `main`),
right-arrow	(selects `putStrLn`),
right-arrow	(selects `"agmoprr"`),
return	(requests parent expression)

[4] In this tutorial highlighted text or expressions are shown boxed; the Hat tools actually use colour for highlighting.

[5] The only thing to do with a source extract is to look at it: tracing with Hat does not involve annotating or otherwise modifying program sources.

```
Trail: ---------------------- Insort.hs line: 7 col: 19 -----
<- putStrLn "agmoprr"
<- insert 'p' "agmorr" | if False
```

The string `"agmoprr"` is the result of inserting `'p'`, the head of the string `"program"`, into the recursively sorted tail. More specifically, the string was computed in the else-branch of the conditional by which `insert` is defined in the recursive case (because `'p' <= 'a'` is `False`).

And so we could continue. For example, following the trail of string arguments:

```
<- insert 'p' "agmorr" | if False
<- insert 'r' "agmor" | if False
<- insert 'o' "agmr" | if False
<- insert 'g' "amr" | if False
<- insert 'r' "am" | if False
<- insert 'a' "m" | if True
<- insert 'm' []
```

But let's leave hat-trail for now.

```
:quit
```

4 Hat-observe: Basics

The idea of hat-observe is to answer the question 'How was that applied, and with what results?', mainly in relation to a top-level function. Answers take the form of a list of equational observations, showing for each application of the function to distinct arguments what result was computed. The user has the option to limit observations to particular patterns of arguments or results, or to particular application contexts.

Hat-observe sessions and requests

A hat-observe session can be started from a shell command line, or from within existing sessions of hat tools.

```
$ hat-observe Insort
hat-observe>
```

In comparison with hat-trail, there is more emphasis on command-lines in hat-observe, and the main user interface is a prompt-request-response cycle. Requests are of two kinds. Some are observation queries in the form of application patterns: the simplest observation query is just the name of a top-level function or defined value. Others are command-lines, starting with a colon, similar to those of hat-trail. A basic repertoire of command-line requests is

`:info`	list the names of functions and other defined values that can be observed
`:quit`	finish this hat-observe session

Again it is enough to give the initial letters, `:i` or `:q`.

Some `Insort` observations

A common way to begin a hat-observe session is with an `:info` request, followed by initial observation of central functions.

```
hat-observe> :info
<=                    insert                main
putStrLn              sort
hat-observe> sort
1 sort "program" = "agmoprr"
2 sort "rogram" = "agmorr"
3 sort "ogram" = "agmor"
4 sort "gram" = "agmr"
5 sort "ram" = "amr"
6 sort "am" = "am"
7 sort "m" = "m"
8 sort [] = []
```

Here the number of observations is small. Larger collections of obervations are presented in blocks of ten (by default).

```
hat-observe> <=
1 'a' <= 'm' = True
2 'r' <= 'a' = False
3 'g' <= 'a' = False
4 'o' <= 'a' = False
5 'p' <= 'a' = False
6 'r' <= 'm' = False
7 'g' <= 'm' = True
8 'o' <= 'g' = False
9 'r' <= 'g' = False
10 'p' <= 'g' = False
--more-->
```

Keying return in response to `--more-->` requests the next block of observations. Alternatively, requests in the colon-command family can be given. Any other line of input cuts short the list of reported observations in favour of a fresh `hat-observe>` prompt.

Observing restricted patterns of applications

Viewing a block at a time is not the only way of handling what may be a large number of applications. Obervations can also be restricted to applications in which specific patterns of values occur as arguments or result, or to applications in a specific context. The full syntax for obervation queries is

```
identifier pattern* [= pattern] [in identifier]
```

where the * indicates that there can be zero or more occurrences of an argument pattern and the `[...]` indicate that the result pattern and context are optional. Patterns in observation queries are simplified versions of constructor patterns with _ as the only variable. Some examples for the `Insort` computation:

```
hat-observe> insert 'g' _
1 insert 'g' "amr" = "agmr"
2 insert 'g' "mr" = "gmr"
hat-observe> insert _ _ = [_]
1 insert 'm' [] = "m"
2 insert 'r' [] = "r"
hat-observe> sort in main
1 sort "program" = "agmoprr"
hat-observe> sort in sort
1 sort "rogram" = "agmorr"
2 sort "ogram" = "agmor"
3 sort "gram" = "agmr"
4 sort "ram" = "amr"
5 sort "am" = "am"
6 sort "m" = "m"
7 sort [] = []
```

Enough on hat-observe for now.

```
hat-observe> :quit
```

5 Tracing Faulty Programs

We have seen so far some of the ways in which Hat tools can be used to trace a correctly working program. But a common and intended use for Hat is to trace a faulty program with the aim of discovering and understanding its faults. A faulty computation has one of three outcomes:

1. termination with a run-time error, or
2. termination with incorrect output, or
3. non-termination.

A variant of `Insort` given[6] in Figure 2 has three faults, each of which alone would cause a different outcome, as indicated by comments. In the following sections we shall apply the Hat tools to examine the faulty program, as if we didn't know in advance where the faults were.

5.1 Tracing a Run-time Error

We compile the faulty program for tracing, then run it:

```
$ hmake -T BadInsort
...
$ BadInsort
No match in pattern.
```

[6]The program may also be found in the file `BadInsort.hs`.

```
sort ::  Ord a => [a] -> [a]
-- FAULT (1):  missing equation for [] argument
sort (x:xs)  =  insert x (sort xs)
insert ::  Ord a => a -> [a] -> [a]
insert x []      =  [x]
insert x (y:ys)  =  if x <= y
                    -- FAULT (2):  y missing from result
                    then x :  ys
                    -- FAULT (3):  recursive call is same
                    else y :  insert x (y:ys)
main  =  putStrLn (sort "program")
```

Figure 2: A faulty version of the insertion sort program.

Two questions prompted by this error message are:

- What was the application that didn't match?
- Where did that application come from?

Using hat-trail

Both questions can be answered by using hat-trail to trace the derivation of the error.

```
$ hat-trail BadInsort

Error: ------------------------------------------------------
No match in pattern.
```

Keying return to see the application that spawned the error:

```
Trail: ------------------- BadInsort.hs line: 3 col: 25 -------
<- sort []
```

The `:source` command shows the site of the offending application to be the recursive call in `sort`. If necessary the ancestry of the `[]` argument or the `sort` application could be traced back further: for example, selecting the `[]` argument (right-arrow twice) shows its origin to be the string literal `"program"` in `main`.

Using hat-observe

Although hat-trail is usually the first resort for tracing run-time errors, it is instructive to see what happens if instead we try using hat-observe.

```
$ hat-observe BadInsort
hat-observe> :info
insert                main                  putStrLn
sort
```

There are no observations of `<=`. How can that be? What is happening to ordered insertion?

```
hat-observe> insert
1 insert 'p' _|_ = _|_
2 insert 'r' _|_ = _|_
3 insert 'o' _|_ = _|_
4 insert 'g' _|_ = _|_
5 insert 'a' _|_ = _|_
6 insert 'm' _|_ = _|_
```

The symbol `_|_` here indicates an undefined value. Reading the character arguments vertically `"program"` seems to be misspelt: is there an observation missing between 4 and 5? There are in fact two separate applications `insert 'r' _|_ = _|_`, but duplicate observations are not listed (by default).

The `insert` observations explain why there are no `<=` applications. In all the observed applications, the list arguments are undefined. So neither of the defining equations for `insert` is ever matched and, as the `<=` comparison is on the right-hand side of the recursive equation, it is never reached.

Why are the `insert` arguments undefined? They should be the results of `sort` applications.

```
hat-observe> sort
1 sort "program" = _|_
2 sort "rogram" = _|_
3 sort "ogram" = _|_
4 sort "gram" = _|_
5 sort "ram" = _|_
6 sort "am" = _|_
7 sort "m" = _|_
8 sort [] = _|_
```

With the exception of the last observation, these `_|_` results are exactly the results obtained from `insert` as already observed. But hat-observe, unlike hat-trail, is not concerned with such relationships.

In short, the story so far from hat-observe is quite simple: everything is undefined! What about the other two items in the info list, `putStrLn` and `main`?

```
hat-observe> putStrLn
1 putStrLn _|_ = IO (putStrLn _|_)
hat-observe> main
1 main = IO (putStrLn _|_)
```

These observations at least confirm that the program does compute an I/O action, but the output string is undefined.

5.2 Tracing a Non-terminating Computation

Suppose we correct the first fault, by restoring the equation

```
sort [] = []
```

and recompile. Now the result of running `BadInsort` is a non-terminating computation, with an infinite string `aaaaaaa...` as output. It seems that `BadInsort` has entered an infinite loop. The computation can be interrupted[7] by keying control-C.

```
$ BadInsort
Program interrupted. (^C)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa$
```

Questions this time include:

- What parts of the program does the infinite loop involve?
- How did it come about in the first place?

Using hat-trail

The initial hat-trail display is:

```
Error: ------------------------------------------------------------
Program interrupted. (^C)
Output: -----------------------------------------------------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
```

We have a choice: we can follow the trail back either from the point of interruption (the initial selection) or from the output (reached by down-arrow). In this case, it makes little difference[8]; either way we end up examining the endless list of `'a'`s. Let's select the output:

```
Output: ------------------------------------------------------------
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...
Trail: ----------------- BadInsort.hs line: 7 col: 19 --------
<- putStrLn "aaaaaaaa▮"
<- insert 'p' ('a':_) | if False
```

Notice two further features of expression display:

- the symbol ▮ in the string argument to `putStrLn` indicates the lower reaches of a large structure (here the tail-end of a long string) that has been pruned from the display;
- the symbol `_` in the list argument to `insert` indicates an expression that was never evaluated.

The parent application `insert 'p' ('a':_) | if False` gives several important clues. It tells us that in the else-branch of the recursive case in the definition of `insert` the argument's head (here `'a'`) is duplicated endlessly to generate the result without ever demanding the argument's tail (shown only as `_`). This should be enough explanation to discover the fault if we didn't already know it.

[7] When non-termination is suspected, interrupt as quickly as possible to avoid working with very large traces.

[8] However, the trace from point of interruption depends on the timing of the interrupt.

Using hat-observe

Once again, let's also see what happens if we use hat-observe.

```
$ hat-observe BadInsort
hat-observe> :info
<=                    insert                main
putStrLn              sort
```

All the expected items are listed as observable. We know well enough from the overall output what `main` and `putStrLn` are doing, but what about `sort`?

```
hat-observe> sort
1 sort "program" = 'a':'a':'a':'a':'a':'a':'a':'a':'a':'a':■:■
2 sort "rogram" = 'a':_
3 sort "ogram" = 'a':_
4 sort "gram" = 'a':_
5 sort "ram" = 'a':_
6 sort "am" = "a"
7 sort "m" = "m"
8 sort [] = []
```

Observations 1 to 5 tell a similar story to hat-trail: the tails of the recursively computed lists are never demanded; at the outermost level, the head is repeated endlessly. Observation 6 points to a problem other than non-termination, but we shall ignore that for now. Observations 7 and 8 do not point to a problem at all.

There is one further clue in these observations: there are only eight of them, and the arguments decrease just as expected. This suggests that the recursive loop is elsewhere — in `insert` perhaps?

```
hat-observe> insert
1 insert 'p' ('a':_) = 'a':'a':'a':'a':'a':'a':'a':'a':'a':'a':■:■
2 insert 'r' ('a':_) = 'a':_
3 insert 'o' ('a':_) = 'a':_
4 insert 'g' ('a':_) = 'a':_
5 insert 'a' "m" = "a"
6 insert 'm' [] = "m"
searching ...  (^C to interrupt)
{Interrupted}
```

There is a distinct pause after observation 6. Many more observations would eventually be reported because hat-observe lists each observation that is distinct from, or more general than[9], those listed previously. When the computation is interrupted there are many different applications of the form `insert 'p' ('a':_)` in progress, each with results evaluated to a different extent.

But observation 1 is enough. As the tail of the argument is unevaluated, the result would be the same whatever the tail. It could be `[]`; so we know `insert 'p' "a" = "aaaa ..."`. This specific and simple failing case directs us to the fault in the definition of `insert`.

[9]Application A is more general than application B if their arguments and results agree where both are evaluated but A's arguments are less evaluated than B's or A's result is more evaluated than B's.

5.3 Tracing Wrong Output

Let's now correct the recursive call from `insert x (y:ys)` to `insert x ys`, recompile, then execute.

```
$ BadInsort
agop
```

Using hat-observe

Once again, we could reach first for hat-trail to trace the fault, but the availability of a well-defined (but wrong) result also suggests a possible starting point in hat-observe:

```
$ hat-observe BadInsort
hat-observe> insert _ _ = "agop"
1 insert 'p' "agor" = "agop"
```

Somehow, insertion loses the final element `'r'`. Perhaps we'd like to see more details of how this result is obtained — the relevant recursive calls, for example:

```
hat-observe> insert 'p' _ in insert
1 insert 'p' "gor" = "gop"
2 insert 'p' "or" = "op"
3 insert 'p' "r" = "p"
```

Observation 3 makes it easy to discover the fault by inspection.

Using hat-trail

If we instead use hat-trail, the same application could be reached as follows. We first request the parent of the output; unsurprisingly it is `putStrLn "agop"`. We then request the parent of the string argument `"agop"`:

```
Output: ------------------------------------------------------
[agop]

Trail: ----------------- BadInsort.hs line: 10 col: 26 -------
<- putStrLn ["agop"]
<- [insert 'p' "agor" | if False]
```

As in hat-observe, we notice the loss of `'r'` in `"agop"`. Back-spacing over this application of `insert`, we instead select the one-character tail of `"agop"`:

```
Trail: ----------------- BadInsort3.hs line: 9 col: 26 -------
<- putStrLn "ago[p"]
<- [insert 'p' "r" | if True]
```

(To be continued.)