Hat-Explore
Version 2.04
Users' Manual

Olaf Chitil

May 4, 2005

1 Introduction

HAT-EXPLORE is a tool for free, source-based navigation through a computation. Like a conventional debugger hat-explore highlights our current position in the computation in the program source and shows a stack backtrace of function calls. In contrast to conventional debuggers we are free of the sequential evaluation order when stepping through the computation. From any function call we can go down to any further function called by it, to a function call in the same function definition, or upwards to the caller of the current function call.

Because arguments and the result are shown for each function call, it is easier to determine which function is incorrect. You can also mark reductions as correct/incorrect which enables the tool to pinpoint the bug to a smaller and smaller slice of the program.

2 The Screen Layout

The display of HAT-EXPLORE is divided into two parts: the call stack and the source. The stack shows a sequence of reductions, where each reduction called the function applied in the reduction below. The last reduction on the call stack is called the current reduction, the reduction that is currently in focus. In the source the *call site* of the redex of the current reduction is underlined.

The following shows the display from the middle of a session:

```
==== Hat-Explore 2.04 ==== Call 1/2 ============================
 1. main = {IO}
 2. sort "sort" = "os"
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
---- Insert.hs ---- line 1 to 9 --------------------------------
main = putStrLn (sort "sort")

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

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

Optionally the *definition site* of the function of the redex can also be highlighted.

3 Navigation through the Computation

You navigate through computation via the cursor keys: up to the caller of the current reduction, down to the first callee, and left and right to siblings.

In the program source the call sites of the siblings are also highlighted but not underlined like the current redex (display shortened):

```
==== Hat-Explore 2.04 ==== Call 2/2 ============================
5. sort "t" = "t"
---- Insert.hs ---- line 1 to 9 --------------------------------
main = putStrLn (sort "sort")

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

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

So, given the state of the last screenshot, pressing the left cursor key yields (display shortened):

```
==== Hat-Explore 2.04 ==== Call 1/2 ============================
5. insert 'r' "t" = "r"
---- Insert.hs ---- line 1 to 5 --------------------------------
main = putStrLn (sort "sort")

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

A move to the caller via cursor key up or to a callee via cursor key down usually requires a complete change of the displayed source, because the call sites are further away. So pressing cursor key down yields:

```
==== Hat-Explore 2.04 ==== Call 1/1 ============================
6. 'o' <= 'r' = True
---- Insert.hs ---- line 6 to 9 --------------------------------

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

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

```
==== Hat-Explore 2.04 ==== Call 1/2 ============================
4. sort "ort" = "o"
---- Insert.hs ---- line 4 to 7 --------------------------------
sort [] = []
sort (x:xs) = insert x (sort xs)

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

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

4 Algorithmic Debugging

HAT-EXPLORE supports algorithmic debugging, that is, error-location based on declarations of the user on which reductions are correct. We can declare if the current reduction is correct or incorrect with respect to our intentions and also change and take back any previous such declaration. The tool uses several colours

for highlighting: correct reductions are green, incorrect ones are yellow, unknown/undeclared ones are blue. When the tool identifies a reduction as faulty, it is highlighted in red.

Let us work step by step through an example session for the faulty insertion sort program. The tool starts with the reduction of `main`. (There is no call site of `main`, hence its definition is underlined.)

```
==== Hat-Explore 2.04 ==== Call 1/1 ============================
 1. main = {IO}


---- Insert.hs ---- lines 1 to 3 -------------------------------
main = putStrLn (sort "sort")

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

We cannot say if this reduction is correct, but only press cursor down to look at the children:

```
==== Hat-Explore 2.04 ==== Call 1/2 ============================
 1. main = {IO}
 2. putStrLn "os" = {IO}

---- Insert.hs ---- lines 1 to 3 -------------------------------
main = putStrLn ( sort "sort" )

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

The first child is a reduction of a trusted function and hence assumed to be correct. So we press cursor right to look at the second child:

```
==== Hat-Explore 2.04 ==== Call 2/2 ============================
 1. main = {IO}
 2. sort "sort" = "os"

---- Insert.hs ---- lines 1 to 3 -------------------------------
main = putStrLn ( sort "sort" )

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

This reduction disagrees with our intentions and hence we press 'w' to declare the reduction as wrong:

```
==== Hat-Explore 2.04 ==== Call 2/2 ============================
 1. main = {IO}
 2. sort "sort" = "os"

---- Insert.hs ---- lines 1 to 3 -------------------------------
main = putStrLn ( sort "sort" )

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

To find out why the reduction is wrong we have to look at the children, so we press cursor down:

```
==== Hat-Explore 2.04 ==== Call 1/2 ============================
 1. main = {IO}
 2. sort "sort" = "os"
 3. insert 's' "o" = "os"
---- Insert.hs ---- lines 3 to 5 -------------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)
```

We press 'c' to declare the reduction as correct and then press cursor right to look at the second child:

```
==== Hat-Explore 2.04 ==== Call 2/2 ============================
 1. main = {IO}
 2. sort "sort" = "os"
 3. sort "ort" = "o"
---- Insert.hs ---- lines 3 to 5 -------------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x ( sort xs )
```

We press 'w' to declare the reduction as wrong and then press cursor down to inquire further:

```
==== Hat-Explore 2.04 ==== Call 1/2 ============================
 2. sort "sort" = "os"
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
---- Insert.hs ---- lines 3 to 5 -------------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x (sort xs)
```

We press 'w' to declare the reduction as wrong:

```
==== Hat-Explore 2.04 ==== Call 1/2 ============================
 2. sort "sort" = "os"
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
---- Insert.hs ---- lines 3 to 5 -------------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x ( sort xs )
```

So the reduction `insert 'o' "r" = "o"` is faulty. We have located the fault, it must be in the definition of `insert`. If we are not convinced, we can still press cursor down to see that `insert 'o' "r" = "o"` has only a single child, a reduction of a trusted function, which is assumed to be correct:

```
==== Hat-Explore 2.04 ==== Call 1/1 ============================
 3. sort "ort" = "o"
 4. insert 'o' "r" = "o"
 5. 'o' <= 'r' = True
---- Insert.hs ---- lines 7 to 9 -------------------------------
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x <= y then x : ys else y : (insert x ys)
```

Declaring the (in)correctness of the current reduction is separate from navigation; it does not automatically navigate to a new reduction. Thus we are free to declare (in)correctness of reductions in any order. In practice it is often much easier to recognise an incorrect reduction than being sure that a reduction is correct. HAT-EXPLORE allows us to look at all children of a redex, determine that one of them is incorrect, and continue exploring that reduction, without having to consider the correctness of its siblings. We might not even rely on algorithmic debugging at all but just use declarations of (in)correctness as memory hints.

5 Program Slicing

HAT-EXPLORE marks a faulty reduction and the definition of the faulty function when it has been identified. However, that happens only rather late, after enough information about correct and incorrect reductions is available. Hence in addition, HAT-EXPLORE offers to mark the definitions of all functions within which the fault must be. These definitions comprise the faulty slice. With increasing information about correct and incorrect reductions the faulty slice shrinks until the faulty reduction has been identified.

This feature is turned off by default and has to be turned on by pressing `f` *for faulty slice. Computation of the faulty slice can be proportional to the size of the trace, hence it is only practically useful for small traces.If* HAT-EXPLORE *seems to freeze, this is probably because it takes very long to compute the slide.*

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

While we declare nodes as correct or incorrect, the slice of definitions that must contain a fault keep shrinking. The shrinking of the faulty slice shows us that we are making progress, it may quickly exclude large parts of the program, possibly parts that had been wrongly suspected, and when the faulty slice has become small we may spot the fault straight away without even having to continue algorithmic debugging to its end. In practice we also often skip declaring the correctness of a node; for example, because it might be hard (large input or output) or impossible (values of abstract data types) to determine.

Exploration with HAT-EXPLORE may uncover several independent faults, each of which has a faulty slice. HAT-EXPLORE marks the faulty slice that belongs to the current reduction or, if the current reduction is correct, the faulty slice belonging to the next faulty reduction upwards on the stack.

6 Smaller Faulty Slices

The faulty slice can be made smaller without additional input from us. When a reduction $f \ldots = \ldots$ is faulty, it is not necessary to add the whole definition of function f to the faulty slice. For a specific reduction usually only parts of the definition body of the reduced function are evaluated because of pattern matching, conditionals and lazy evaluation. The fault can only be in that part of the definition that was actually evaluated for that particular reduction. Evaluated parts of the definition are the call sites of the children of the node plus demanded constants,

data constructor applications and literals.[1] HAT-EXPLORE optionally only shows this smaller faulty slice. Pressing `p` toggles between partial and full slices. In our example program the "else" branch was never evaluated for the current, incorrect reduction.

```
==== Hat-Explore 2.03 ==== Call 2/2 | faulty slice | executed ===
 1. main = {IO}
 2. sort "sort" = "os"
 3. sort "ort" = "o"
---- Insert.hs ---- line 3 to 9 ---------------------------------
sort :: Ord a => [a] -> [a]
sort [] = []
sort (x:xs) = insert x ( sort xs )

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

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

7 Code Coverage

By declaring the root reduction of the computation, `main = {IO}`, as incorrect and asking HAT-EXPLORE to mark only the evaluated faulty slice, we can obtain the slice of the program that was evaluated at all during the whole computation. So HAT-EXPLORE can serve as a code coverage tool.

[1] If a constant is evaluated, it is impossible to determine if it was demanded for the currently considered reduction or a different part of the computation, because constants are shared. For most data constructor applications and literals, the entry in the HAT trace contains no indication if they were ever demanded in the computation. To be on the safe side, in all such cases the expression has to be included in the slice, if the surrounding expression construct is included.

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

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


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

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

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

8 Trusting

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

Trusted functions can be higher-order and the functional arguments may be normal untrusted functions, for example `map myInc [1,2,3] = [2,3,4]`. In that case the reduction of the trusted function can have children, namely the reductions of the passed untrusted functions. So `map myInc [1,2,3] = [2,3,4]` has the children `myInc 1 = 2`, `myInc 2 = 3` and `myInc 3 = 4`. In general, trusting causes parts of a computation tree to be "cut out", even out of the middle of the tree. If a trusted reduction has children, it cannot assumed to be correct by default.

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

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

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

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

9 Constants

A constant definition, such as `nats = [0..]`, has to be handled specially in the computation tree. In a computation the definition body is only evaluated once and the value is shared by all calls (i.e. uses) of the constant in the program. Because of this sharing the computation is not a tree but a directed graph. Navigation into the computation of a constant is natural. Where to go back up is also uniquely identified by the information in the stack.

Because constant definitions may be (mutually) recursive, the computation graph may be cyclic. Algorithmic debugging only works for trees or acyclic graphs. It is currently the responsibility of the user to be aware that algorithmic debugging may not be able to locate a faulty reduction within the computation of mutually recursive functions. The faulty slice is still correct, but it may never shrink further than a set of mutually recursive definitions.

10 Other Starting Points

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

Other viewing tools such as HAT-TRAIL and HAT-OBSERVE may give quicker access to a reduction of interest. Hence both of them have the option to switch to HAT-EXPLORE, starting at the reduction that we just investigated in the other tool.

Experience shows that faults are often not far from the observed error. Hence if a computation aborts with a runtime error, HAT-TRAIL starts directly at the reduction that raised the runtime error. For example, a slightly modified version of

our insertion sort causes a pattern match failure. HAT-EXPLORE starts as follows, displaying the error value as _|_ (bottom):

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

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

11 Command Overview

Pressing `h` yields the complete list of commands understood by HAT-EXPLORE:

```
cursor down    follow current call
cursor up      go back to caller of current call
cursor left    go to call further left in current definition body
cursor right   go to call further right in current definition body

c     declare current equation to be correct (wrt. intentions)
w     declare current equation to be wrong (wrt. intentions)
n     undo declaration of correctness (neither correct nor wrong)
a     amnesia - forget all declarations as correct or wrong

f     toggle between showing fault set or just current definition
p     toggle between showing used part of definition or full

<     change to alphabetically preceeding module
>     change to alphabetically succeeding module

t     scroll source window to top of code
u     scroll source window upwards
d     scroll source window downwards
b     scroll source window to bottom of code
```

```
 r              redraw everything after change of window size
 h or ?         display this help text

 q              quit


Meaning of colours:
green - correct, amber - wrong, blue - unkown, red - faulty.
```