A Lightweight Interactive Debugger for Haskell

Simon Marlow
José Iborra
Bernard Pope
Andy Gill
We don’t need no debugger!

• We have

  GHCi and Hugs!

  A type system!

  QuickCheck and HUnit!

  NO BUGS!
but...

• A debugger was still the #1 requested feature in the GHC survey (2005)
• New users often want a way to visualise execution to understand what’s happening
• When you missed out a test, diagnosing the failure is hard: deep bugs
• head [] 😞
But we have debuggers!

- There exist debugging tools for Haskell, but for various reasons they are often too hard to use:
  - limited language support
  - require recompiling libraries
  - time/space overhead
  - etc. (see paper for comparison)
Goals

• We want a debugger that:
  – Is always available
  – works with everything that GHC can compile
  – doesn’t add significant overheads

• We might sacrifice some functionality to achieve accessibility. Aim for a high power-to-weight ratio.

• This won’t necessarily supercede other Haskell debuggers: the goal is to provide some debugging functionality that everyone can use easily.

• Secondary goals:
  – can be re-used in IDE frontends
Real programmers want to set breakpoints

• Our debugger is “online”: you debug the running program, as opposed to “post-mortem” debugging (eg. Hat).
• The basic model is “execute, stop, inspect, continue”.
• Advantages:
  – we can let you evaluate arbitrary expressions involving runtime values (e.g. look up a key in a finite map).
  – no limit on the runtime of the program
• Disadvantages:
  – we can’t easily go back in time
  – evaluation order is exposed (possibly an advantage?)
• Our debugger is integrated into GHCi
**walkthrough**

`GHCi`, version 6.8.0.20070919: http://www.haskell.org/ghc/ :? for help
Loading package base ... linking ... done.

```
Prelude> :load foldl
[1 of 1] Compiling Test             ( foldl.hs, interpreted )
Ok, modules loaded: Test.
*Test> :list foldl
10
  11 foldl f c xs = go c xs
  12    where go c []     = c
  13          go c (x:xs) = go (f c x) xs

*Test> :break foldl
Breakpoint 1 activated at foldl.hs:(11,0)-(13,34)
*Test> :show breaks
[0] Test foldl.hs:(11,0)-(13,34)
*Test> foldl (+) 0 [1..5]
Stopped at foldl.hs:(11,0)-(13,34)
_result :: t1 = _
  [foldl.hs:(11,0)-(13,34)] *Test>
  [foldl.hs:(11,0)-(13,34)] *Test> :list
10
  11 foldl f c xs = go c xs
  12    where go c []     = c
  13          go c (x:xs) = go (f c x) xs
```

breakpoints can be set in any interpreted module, on any function definition or expression.
**Walkthrough**

```haskell
result :: a = _
c :: a = _
go :: a -> [b] -> a = _
x :: [b] = _
```

we can take a single step, which executes until the next breakpoint location - next expression or function call.

this is the expression to be evaluated next

new bindings for variables in scope at the breakpoint. Underscore means “unevaluated”

we can evaluate expressions involving the local variables
some variables have polymorphic types. What does that mean? At runtime, c is bound to a value with a monomorphic type.

The compiler doesn’t know the runtime type of c, so it doesn’t know which Show instance to use.

‘a’ is a runtime type variable

Defaulting does not apply to these type variables.

Note that ‘length xs’ worked, however.
:print displays a value without knowing its type, and without forcing evaluation.

:force is like :print, but forces evaluation.

the type of ‘c’ is now known!

and the type information has been propagated to the other bindings (type variable ‘a’ is now bound to ‘Integer’)

‘b’ was resolved to Integer earlier by ‘length xs’.

GHCi automatically propagates type information as it becomes available.
most of the types are concrete, because GHCi has inspected the values of the free variables to determine their types.
**walkthrough**

```haskell
foldl f c xs = go c xs
  where go c [] = c
  go c (x:xs) = go (f c x) xs
```

**note that ‘c’ is unevaluated now. foldl is building up a chain of thunks.**
walkthrough

We can single-step the evaluation of ‘c’ separately, :step takes an expression to evaluate as an argument

oops, not like that!

but this works

“…” in the prompt indicates that we are now in a nested debugging session.

lists the current debugging sessions
continue the evaluation of c

result of `c `seq` ()` is ()

continue the original evaluation, result is 15
walkthrough

let’s try something else… first delete all breakpoints

foldl has a commonly-encountered problem with stack overflow
(I’m using a reduced stack limit here).
walkthrough

*Test> :set -fbreak-on-error

*Test> :trace foldl (+) 0 [1..20000]
Stopped at <exception thrown>
_exception :: e = stack overflow

[<exception thrown>] *Test> :history
-1 : foldl (foldl.hs:13:26-30)
-3 : foldl (foldl.hs:13:26-30)
-4 : foldl (foldl.hs:13:26-30)
-5 : foldl (foldl.hs:13:26-30)
-6 : foldl (foldl.hs:13:26-30)
-7 : foldl (foldl.hs:13:26-30)
-8 : foldl (foldl.hs:13:26-30)
-10 : foldl (foldl.hs:13:26-30)
-12 : foldl (foldl.hs:13:26-30)
-14 : foldl (foldl.hs:13:26-30)
-16 : foldl (foldl.hs:13:26-30)
-17 : foldl (foldl.hs:13:26-30)
-20 : foldl (foldl.hs:13:26-30)
...

Uncaught exceptions now trigger a breakpoint
:trace evaluates as normal, but remembers the 50 most recent steps
:history shows the history of evaluation steps when stopped
every step in the history is evaluating ‘f c x’
The :trace/:history functionality is a poor substitute for a proper lexical call stack.

Maintaining a proper lexical call stack is hard, but :trace/:history were easy to implement.

Still, it is useful for finding the source of ‘head []’, pattern-match failure, stack overflow, and infinite loops.

For infinite loops, hit Control-C which generates an exception, then use :history.
A problem with implementing a debugger is during execution answering the question “where in the source code am I right now?”. Approach 1: annotate each expression in the object code with the source expression it originated from.

- Problem: with extensive optimisation and transformation, it is hard to maintain this information accurately, and it requires careful attention to all the transformations.
- cf. debugging optimised C with gdb.
Annotating compiled code

- Approach 2: use a systematic source-to-source transformation to yield a program with the same semantics, but with additional side-effects, e.g. reporting the current source location
  - Used by several existing Haskell debuggers, also the HPC coverage tool.
  - Two challenges:
    - the annotations are side-effects, we don’t want those side-effects to be lost or moved by transformations in the compiler
    - do this without impacting performance too much - we want the debugger to be “always on”
Ticks

- Has the same semantics as E
- `tick<module,n>` is a side-effect:
  - in coverage it records that this expression was evaluated
  - in the debugger it checks whether this breakpoint is enabled
- GHC’s middle-stage optimisations are already side-effect-aware; they won’t
  - eliminate the side-effect, or
  - perform it speculatively
- but otherwise transformations are unaffected.
Compared to coverage analysis...

- The debugger puts ticks in different places
  - not on variables (too many ticks)
  - in let-expressions, coverage puts it here:

```haskell
  case tick<module,n> of
    DEFAULT -> let x = E in E'
```

- the debugger puts it here:

```haskell
  let x = E in
  case tick<module,n> of
    DEFAULT -> E'
```

- so that x is in scope at the breakpoint
Compiling ticks

• Ideally we want ticks to imply no extra allocation. e.g. in

```haskell
let f = \x -> case tick<module,n> of
    DEFAULT -> E
in ...
```

• f gives rise to a single byte-code-object (BCO), the first
  instruction of which implements the tick.
• Sadly, not all ticks are quite so easy to implement. e.g.

```haskell
let x = E in
    case tick<m,n> of DEFAULT -> E'
```

• Every tick must be at a safe-point, i.e. at the beginning of a
  BCO, because execution may be suspended (just like a GC).
  We need to manufacture an extra safe point:

```haskell
let x = E in
let z = case tick<m,n> of DEFAULT -> E'
in z
```
Performance

- Interpreted code already runs 15-20x slower than optimised compiled code (but compiles 2-3x quicker)
- Performance of interpreted code is affected:
  - at compile-time, we need to insert ticks (+15%)
  - at runtime, check breakpoint status at each breakpoint, and extra safe-points (+45%)
    - But safe points do unnecessary updates, we expect to be able to reduce this
  - debugger cannot be disabled, but individual modules can be compiled rather than interpreted.

Figure 3. NoFib running times
The GHC API

• The debugger is implemented mostly in Haskell; breakpoints are implemented using threads.

• Debugging functionality is exposed via the GHC API.

• GHCi is just a textual front-end built on top of this API, IDEs could also talk directly to the GHC API to provide interactive debugging.
Future Work

• High priority: a real lexical call stack
• Get feedback from GHC 6.8.1 users - is exposing the evaluation order confusing or helpful?
• Performance can be improved
  – breakpoints in compiled code should be possible
• API interface means it can be built into an IDE
• Concurrency debugging
Conclusion

- We have an instant always-on debugger that works with everything that GHCi can compile.
- Functionality is limited compared to other debuggers - no backtrace, no algorithmic debugging etc.
- Online debugging has some benefits over offline debugging:
  - evaluate arbitrary expressions involving runtime values
  - no need to store the entire trace of the program
  - no need to switch tools if you’re already using GHCi