Stack Tracing in Haskell

An exploration of the design space

Bernie Pope, 2009

The University of Melbourne, Australia

Sad state of debugging support in Haskell

Typical advice from the Haskell cafe mailing list:

“Techniques that worked for Java don't work very well when debugging Haskell. Others will tell you about flags and possibly using the debugger but <u>I would count on eyeballing and printing as the least painful method</u>.”

Outline

- Motivation.
- Desirable features.
- Technical impediments.
- Existing solutions.
- Conclusions.

Why do we want stack tracing?

- To explain dynamic context.
- It is easy to get lost in GHCi's breakpoint debugger.
  - Programmers want to know "how did the computation get here?"
- Uncaught exceptions are a common source of errors.
  - They *ought* to be relatively easy to find.
  - Currently they can be hard to find.
  - Programmers want to know "how did the computation get here?"

Stack tracing considered beneficial to Facebook

- In the “Functional Programming at Facebook” CUFP talk on Friday:

  “Stack traces point the way to bugs” (using Erlang).

Desirable features

- Works with all programs.
- Is accessible (doesn't require contortions or extensive source modifications).
- Can be used within the GHCi debugger.
- Can be applied selectively to subparts of programs.
- Space and time efficient (with knobs to tune).
- Arguments of a function call can be optionally included in the trace.
- Output makes sense to mortals on sane code (bugs are ultimately found).

Some hurdles (the usual suspects)

- Lazy evaluation.
- CAFs.
- Higher-order functions.
- Performance costs.

Lazy evaluation is the main culprit

```
map f list
   = case list of
        []    -> []
        x:xs -> f x : map f xs
```

`list` might not be in WHNF.
The case statement might cause other redexes to be evaluated.

Lazy evaluation is the main culprit

```
map f list
   = case list of
        []   -> []
        x:xs -> f x : map f xs
```

Subterms of the body are suspended as thunks. If and when they are reduced depends on the external context of the call to **`map`**.

Higher-order functions are also tricky

```
map f list
   = case list of
        []    -> []
        x:xs -> f x : map f xs
```

Does the call to the function bound to **`f`** constitute a child of the call to **`map`**?

When *is* a function called in Haskell?

Maybe controversial.

Some options:

- In the context where it is first mentioned by name.
- In the context where it is saturated.
- All of the above and any other context where it receives an argument.
- In the context where it is reduced.

When *is* a function called in Haskell?

Maybe controversial.

Some options:

- In the context where it is first mentioned by name.
- In the context where it is saturated.
- All of the above and any other context where it receives an argument.
- In the context where it is reduced.

This is the view you get from cost centre stacks, and StackTrace.

When *is* a function called in Haskell?

Maybe controversial.

Some options:

- In the context where it is first mentioned by name.
- In the context where it is saturated.
- All of the above and any other context where it receives an argument.
- In the context where it is reduced.

This is the view you get from Hat-Stack, and stack traces in most conventional languages.

Existing solutions - main contenders

- Cost centre stacks.
- StackTrace (as seen at the Haskell Symposium on Thursday).
- Hat.
- Breakpoint debugger trace history.

Cost centre stacks (simplified for stack tracing)

code	stack	heap	context
mark c e	s	h	cs

$\rightarrow$

code	stack	heap	context
e	s	h	c:cs

Cost centre stacks - context annotations

code	stack	heap	context
mark c e	s	h	cs

code	stack	heap	context
e	s	h	c:cs

Annotate the expression e with some context information c. It is called scc in the profiler.

Cost centre stacks - the context stack

code	stack	heap	context
mark c e	s	h	cs

→

code	stack	heap	context
e	s	h	c:cs

The current context (stack). Not part of the normal semantics. Added for profiling where it is called the cost-centre stack.

Cost centre stacks - thunk allocation

code	stack	heap	context
let x = THUNK(e_2) in e_1	s	h	cs

→

code	stack	heap	context
e_1[y/x]	s	h[y := THUNK(e_2)$_{cs}$]	cs

Cost centre stacks - thunk allocation

code	stack	heap	context
let x = THUNK(e_2) in e_1	s	h	cs

→

The thunk heap object is annotated with the current context.

code	stack	heap	context
e_1[y/x]	s	h[y := THUNK(e_2)$_{cs}$]	cs

Cost centre stacks - thunk reduction

code	stack	heap	context
x	s	h[x := THUNK(e)$_{thunk\_cs}$]	cs

→

code	stack	heap	context
e	(Update x)$_{cs}$: s	h[x := BLACKHOLE]	thunk_cs

Cost centre stacks - thunk reduction

code	stack	heap	context
x	s	h[x := THUNK(e)$_{thunk\_cs}$]	cs

code	stack	heap	context
e	(Update x)$_{cs}$: s	h[x := BLACKHOLE]	thunk_cs

The context of the thunk is reinstated.

Cost centre stacks: CAFs are a pain

- CAFs are top-level thunks.
- Where do they get their context from?

Cost centre stacks: CAFs are a pain

```
div x@(I32# x#) y@(I32# y#)
    | y == 0                      = divZeroError
    | x == minBound && y == (-1)  = overflowError
    | otherwise                   = I32# (x# `divInt32#` y#)

divZeroError :: a
divZeroError = throw (ArithException DivideByZero)
```

```
ghc --make -prof -auto-all Main.hs
./Main +RTS -xc -RTS
<GHC.Err.CAF>Main: divide by zero
```

CAFs - can we just turn them into functions?

- Sharing of CAF reduction *can* be important for performance.
- An optimising compiler can introduce more CAFs by lifting constant expressions - so they can be more common than you think.
- Big question: can we preserve sharing (when needed) but still get useful traces?
- Maybe it is sufficient if a CAF receives its context from the first place it is evaluated?

Cost centre stacks - stack compression

- Necessary to keep the space usage in check.
- When a function is pushed on the context stack all previous instances of that function are removed from the stack. (Morgan, Jarvis, JFP 1998).
- Interesting to compare with the stack elision in StackTrace.
- More difficult if you want to allow arguments to function calls in the stack.

Hat (stack)

- Source-to-source transformation.
- Massive performance overheads, but big payoff.
- Detailed execution trace is saved to disk and can be debugged with many different tools.
- Somewhat difficult to maintain:
    - Need to transform all libraries.
    - Primitive functions need special treatment.
- It would be nice if Haskell had better support for source transformation tools.

Some things to do ...

- Work out the (desirable) semantics of stack tracing in Haskell.
    - What to do with higher-order function applications?
    - What to do with CAFs?
- Determine if the cost-centre stacks of profiling can be re-used.
- Look more closely at the continuation marks in Scheme (e.g. “*Modeling an algebraic stepper*” Clements, Flatt, Felleisen.)
- http://www.haskell.org/haskellwiki/Ministg