Stack Traces in Haskell

Arash Rouhani

Chalmers University of Technology

Master thesis presentation
March 21, 2014

Contents

- Motivation
- Background
- The attempt in August 2013
- Contribution

An old problem . . .

- Try running this program:

```
main = print (f 10)
f x = ... g y ...
g x = ... h y ...
h x = ... head [] ...
```

An old problem . . .

- Try running this program:

```
main = print (f 10)
f x = ... g y ...
g x = ... h y ...
h x = ... head [] ...
```

- You get

```
$ runghc Crash.hs
Crash.hs: Prelude.head: empty list
```

An old problem . . .

- Try running this program:

```
main = print (f 10)
f x = ... g y ...
g x = ... h y ...
h x = ... head [] ...
```

- You get

```
$ runghc Crash.hs
Crash.hs: Prelude.head: empty list
```

- But you want

```
$ runghc Crash.hs
Crash.hs: Prelude.head: empty list
  in function   h
  in function   g
  in function   f
  in function   main
```

. . . with new constraints

- Should have very low overhead
- If you hesitate to use it in production, I've failed
- Not done for Haskell before, all earlier work have an overhead.

Background contents

- Is stack traces harder for *Haskell*?
- Will the implementation only work for *GHC*?

Laziness

- Consider the code

```haskell
myIf :: Bool -> a -> a -> a
myIf True  x y = x
myIf False x y = y

-- Then evaluate
myIf True 5 (error "evil crash")
```

- Will the usage of `error` make this crash?

Laziness

- Consider the code

```haskell
myIf :: Bool -> a -> a -> a
myIf True  x y = x
myIf False x y = y

-- Then evaluate
myIf True 5 (error "evil crash")
```

- Will the usage of `error` make this crash?
- No, `(error "evil crash")` is a *delayed computation*.

Case expressions

- Consider the code

```
case myBool of
  True  -> this
  Flase -> that
```

- So is pattern matching just like `switch-case` in C?

Case expressions

- Consider the code

```haskell
case myBool of
  True  -> this
  Flase -> that
```

- So is pattern matching just like `switch-case` in C?
- NO!

Case expressions

- Consider the code

```
case myBool of
  True  -> this
  Flase -> that
```

- So is pattern matching just like `switch-case` in C?
- NO!
- `myBool` can be a delayed computation, aka a *thunk*

History of GHC

- Compiles Haskell to machine code since 1989
- The only Haskell compiler people care about

Usage

- Compile and run (just like any other compiler)

```
$ ghc --make Code.hs
...
$ ./a.out
123
```

The magical function

- My work assumes the existence of

```haskell
getDebugInfo :: Ptr Instruction -- Pointer to runnable machine code
             -> IO DebugInfo -- Haskell function name etc.
```

The magical function

- My work assumes the existence of

```haskell
getDebugInfo :: Ptr Instruction -- Pointer to runnable machine code
             -> IO DebugInfo -- Haskell function name etc.
```

- This is a recent contribution not yet merged in HEAD
- Author is Peter Wortmann, part of his PhD at Leeds

The magical function

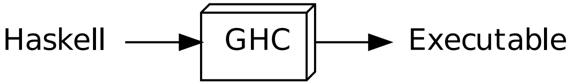
- My work assumes the existence of

```
getDebugInfo :: Ptr Instruction -- Pointer to runnable machine code
             -> IO DebugInfo -- Haskell function name etc.
```

- This is a recent contribution not yet merged in HEAD
- Author is Peter Wortmann, part of his PhD at Leeds
- *In essence, 95% of the job to implement stack traces was already done!*

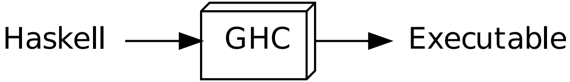
The compilation pipeline

- Well GHC works like this:

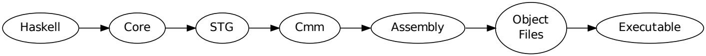


The compilation pipeline

- Well GHC works like this:

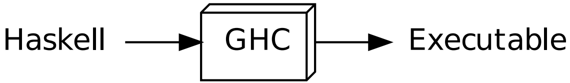


- Or rather like this

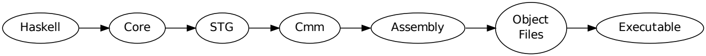


The compilation pipeline

- Well GHC works like this:



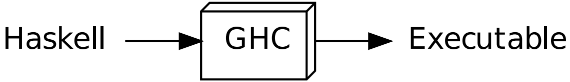
- Or rather like this



- We say that GHC has many *Intermediate Representations*

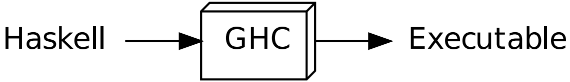
So there must be debug data!

- Again:



So there must be debug data!

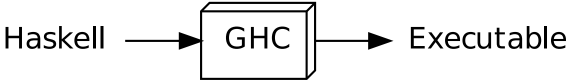
- Again:



- The intuition behind `getDebugInfo` is:

So there must be debug data!

- Again:



- The intuition behind `getDebugInfo` is:

- For this, we *must* retain debug data in the binary!

Lets get to work!

Successive recastings

Haskell
```
addition :: Int -> Int -> Int
addition x y = (x + y)
```

Core
```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType]
addition_r8m =
  \ (x_a9l :: GHC.Types.Int) (y_a9m :: GHC.Types.Int) ->
    GHC.Num.+ @ GHC.Types.Int GHC.Num.$fNumInt x_a9l y_a9m
```

Stg
```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType, Unf=OtherCon []] =
    sat-only \r srt:SRT:[(r9o, GHC.Num.$fNumInt)] [x_smq y_smr]
        GHC.Num.+ GHC.Num.$fNumInt x_smq y_smr;
```

Cmm
```
...
cmG:
  R2 = GHC.Num.$fNumInt_closure;   // CmmAssign
  I64[(old + 32)] = stg_ap_pp_info;   // CmmStore
  P64[(old + 24)] = _smo::P64;   // CmmStore
  P64[(old + 16)] = _smp::P64;   // CmmStore
  call GHC.Num.+_info(R2) args: 32, res: 0, upd: 8;   // CmmCall
...
```

x64 assembly
```
...
_cmG:
   movq %r14,%rax
   movl $GHC.Num.$fNumInt_closure,%r14d
   movq $stg_ap_pp_info,-24(%rbp)
   movq %rax,-16(%rbp)
   movq %rsi,-8(%rbp)
   addq $-24,%rbp
   jmp GHC.Num.+_info
...
```

Successive recastings

Haskell
```
addition :: Int -> Int -> Int
addition x y = (x + y)
```

Core
```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType]
  \ (x_a9l :: GHC.Types.Int) (y_a9m :: GHC.Types.Int) ->
    src<stages.hs:3:1-22>
    GHC.Num.+
      @ GHC.Types.Int
      GHC.Num.$fNumInt
      (src<stages.hs:3:17> x_a9l)
      (src<stages.hs:3:21> y_a9m)
```

Stg
```
addition_r8m :: GHC.Types.Int -> GHC.Types.Int -> GHC.Types.Int
[GblId, Arity=2, Str=DmdType, Unf=OtherCon []] =
    sat-only \r srt:SRT:[(r9o, GHC.Num.$fNumInt)] [x_smq y_smr]
        src<stages.hs:3:1-22>
        src<stages.hs:3:17>
        src<stages.hs:3:21> GHC.Num.+ GHC.Num.$fNumInt x_smq y_smr;
```

Cmm
```
...
  //tick src<stages.hs:3:1-22>
  //tick src<stages.hs:3:17>
  //tick src<stages.hs:3:21>
...
CmG:
  R2 = GHC.Num.$fNumInt_closure;   // CmmAssign
  I64[(old + 32)] = stg_ap_pp_info;   // CmmStore
  P64[(old + 24)] = _smo::P64;   // CmmStore
  P64[(old + 16)] = _smp::P64;   // CmmStore
  call GHC.Num.+_info(R2) args: 32, res: 0, upd: 8;   // CmmCall
...
```

x64 assembly
```
...
_cmG:
   movq %r14,%rax
   movl $GHC.Num.$fNumInt_closure,%r14d
   movq $stg_ap_pp_info,-24(%rbp)
   movq %rax,-16(%rbp)
   movq %rsi,-8(%rbp)
   addq $-24,%rbp
   jmp GHC.Num.+_info
...
```

What happened?

```
...
```

```
...
_cmG:
   movq %r14,%rax
   movl $GHC.Num.$fNumInt_closure,%r14d
   movq $stg_ap_pp_info,-24(%rbp)
   movq %rax,-16(%rbp)
   movq %rsi,-8(%rbp)
   addq $-24,%rbp
   jmp GHC.Num.+_info
...
```

x64 assembly

```
...
```

```
...
_cmG:
   movq %r14,%rax
   movl $GHC.Num.$fNumInt_closure,%r14d
   movq $stg_ap_pp_info,-24(%rbp)
   movq %rax,-16(%rbp)
   movq %rsi,-8(%rbp)
   addq $-24,%rbp
   jmp GHC.Num.+_info
...
```

x64 assembly

- Did we just drop the debug data we worked so hard for?

This is a solved problem, of course!

- DWARF to the rescue!

```
< 1><0x0000008d>    DW_TAG_subprogram
                      DW_AT_name                 "addition"
                      DW_AT_MIPS_linkage_name    "r8m_info"
                      DW_AT_external             no
                      DW_AT_low_pc               0x00000020
                      DW_AT_high_pc              0x00000054
                      DW_AT_frame_base           DW_OP_call_frame_cfa
< 2><0x000000b3>      DW_TAG_lexical_block
                        DW_AT_name                 "cmG_entry"
                        DW_AT_low_pc               0x00000029
                        DW_AT_high_pc              0x0000004b
< 2><0x000000cf>      DW_TAG_lexical_block
                        DW_AT_name                 "cmF_entry"
                        DW_AT_low_pc               0x0000004b
                        DW_AT_high_pc              0x00000054
```

This is a solved problem, of course!

- DWARF to the rescue!

```
< 1><0x0000008d>    DW_TAG_subprogram
                      DW_AT_name                    "addition"
                      DW_AT_MIPS_linkage_name       "r8m_info"
                      DW_AT_external                no
                      DW_AT_low_pc                  0x00000020
                      DW_AT_high_pc                 0x00000054
                      DW_AT_frame_base              DW_OP_call_frame_cfa
< 2><0x000000b3>      DW_TAG_lexical_block
                        DW_AT_name                    "cmG_entry"
                        DW_AT_low_pc                  0x00000029
                        DW_AT_high_pc                 0x0000004b
< 2><0x000000cf>      DW_TAG_lexical_block
                        DW_AT_name                    "cmF_entry"
                        DW_AT_low_pc                  0x0000004b
                        DW_AT_high_pc                 0x00000054
```

- DWARF lives *side by side* in another section of the binary. Therefore it does not interfere.

Introduction to the Execution Stack

- GHC *chooses* to implement Haskell with a stack.

Introduction to the Execution Stack

- GHC *chooses* to implement Haskell with a stack.
- It does not use the normal "C-stack"

Introduction to the Execution Stack

- GHC *chooses* to implement Haskell with a stack.
- It does not use the normal "C-stack"
- GHC maintains its own stack, we call it the *execution stack*.

Similar but not same

- Unlike C, we do not push something on the stack when entering a function!

Similar but not same

- Unlike C, we do not push something on the stack when entering a function!
- Unlike C, we have cheap green threads, one stack per thread!

What is on it then?

- Recall this code:

```
case myBool of
  True  -> this
  Flase -> that
```

What is on it then?

- Recall this code:

```
case myBool of
  True  -> this
  Flase -> that
```

- How is this implemented? Let's think for a while . . .

What is on it then?

- Recall this code:

```
case myBool of
  True  -> this
  Flase -> that
```

- How is this implemented? Let's think for a while ...
- Aha! We can push a *continuation* on the stack and jump to the code of `myBool`!

What is on it then?

- Recall this code:

```
case myBool of
  True  -> this
  Flase -> that
```

- How is this implemented? Let's think for a while . . .
- Aha! We can push a *continuation* on the stack and jump to the code of `myBool`!
- We call this a *case continuation*.

Peter's demonstration

- In August 2013 Peter Wortmann showed a proof of concept stack trace based on his work.

Peter's demonstration

- In August 2013 Peter Wortmann showed a proof of concept stack trace based on his work.
- My master thesis is entirely based on Peter's work.

The stack trace . . .

- For this Haskell code:

```haskell
main :: IO ()
main = do a
          print 2

a, b :: IO ()
a = do b
       print 20

b = do print (crashSelf 2)
       print 200

crashSelf :: Int -> Int
crashSelf 0 = 1 `div` 0
crashSelf x = crashSelf (x - 1)
```

. . . is *terrible*!

- We get:

```
 0: stg_bh_upd_frame_ret
 1: stg_bh_upd_frame_ret
 2: stg_bh_upd_frame_ret
 3: showSignedInt
 4: stg_upd_frame_ret
 5: writeBlocks
 6: stg_ap_v_ret
 7: bindIO
 8: bindIO
 9: bindIO
10: stg_catch_frame_ret
```

. . . is *terrible*!

- We get:

```
0: stg_bh_upd_frame_ret
1: stg_bh_upd_frame_ret
2: stg_bh_upd_frame_ret
3: showSignedInt
4: stg_upd_frame_ret
5: writeBlocks
6: stg_ap_v_ret
7: bindIO
8: bindIO
9: bindIO
10: stg_catch_frame_ret
```

- We want:

```
0: crashSelf
1: crashSelf
2: print
3: b
4: a
5: main
```

Then what did Arash do?

- In addition to an unreadable stack trace, the time and memory complexity of stack reification can be improved.

Then what did Arash do?

- In addition to an unreadable stack trace, the time and memory complexity of stack reification can be improved.
- The stack reification in Peter Wortmann's demonstration is linear in time and memory.

Then what did Arash do?

- In addition to an unreadable stack trace, the time and memory complexity of stack reification can be improved.
- The stack reification in Peter Wortmann's demonstration is linear in time and memory.
- *Obviously*, if you throw a stack and then print it. It can not be worse than linear in time.

Then what did Arash do?

- In addition to an unreadable stack trace, the time and memory complexity of stack reification can be improved.
- The stack reification in Peter Wortmann's demonstration is linear in time and memory.
- *Obviously*, if you throw a stack and then print it. It can not be worse than linear in time.
- *But*, if you throw a stack and do *not* print it, a reification that is done *lazily* would be done in constant time.

So the problems to tackle are:

- Make stack traces readable

So the problems to tackle are:

- Make stack traces readable
- Make reification optimal complexity wise

So the problems to tackle are:

- Make stack traces readable
- Make reification optimal complexity wise
- Add a Haskell interface to this

We must understand the stack

- What is on the stack?

We must understand the stack

- What is on the stack?
- The C stack just have return addresses and local variables.

We must understand the stack

- What is on the stack?
- The C stack just have return addresses and local variables.
- The Haskell stack have many different kinds of members. Case continuations, update frames, catch frames, stm frames, stop frame, underflow frames etc.

We must understand the stack

- What is on the stack?
- The C stack just have return addresses and local variables.
- The Haskell stack have many different kinds of members. Case continuations, update frames, catch frames, stm frames, stop frame, underflow frames etc.
- 

Update frames

- Consider

```haskell
powerTwo :: Int -> Int
powerTwo 0 = 1
powerTwo n = x + x
  where x = powerTwo (n - 1)
```

Update frames

- Consider

```
powerTwo :: Int -> Int
powerTwo 0 = 1
powerTwo n = x + x
  where x = powerTwo (n - 1)
```

- In GHC, thunks are memoized by default

Update frames

- Consider

```haskell
powerTwo :: Int -> Int
powerTwo 0 = 1
powerTwo n = x + x
  where x = powerTwo (n - 1)
```

- In GHC, thunks are memoized by default
- This is done by update frames on the stack

Update frames

- Consider

```haskell
powerTwo :: Int -> Int
powerTwo 0 = 1
powerTwo n = x + x
  where x = powerTwo (n - 1)
```

- In GHC, thunks are memoized by default
- This is done by update frames on the stack
- Details omitted in interest of time

New policy for reifying update frames

- So instead of saying that we have an update frame, refer to its updatee.

```
 0: stg_bh_upd_frame_ret   ----------->     0: divZeroError
 1: stg_bh_upd_frame_ret   ----------->     1: crashSelf
 2: stg_bh_upd_frame_ret   ----------->     2: b
 3: showSignedInt          ----------->     3: showSignedInt
 4: stg_upd_frame_ret      ----------->     4: print
 5: writeBlocks            ----------->     5: writeBlocks
 6: stg_ap_v_ret           ----------->     6: stg_ap_v_ret
 7: bindIO                 ----------->     7: bindIO
 8: bindIO                 ----------->     8: bindIO
 9: bindIO                 ----------->     9: bindIO
10: stg_catch_frame_ret    ----------->    10: stg_catch_frame_ret
```

Other frames

- Many of the frames are interesting. But the most common one is probably case continuations, which luckily are unique and therefore useful when applying `getDebugInfo`

The problem

- On a crash, the stack is unwounded and the stack reified
- Control is passed to the first catch frame on the stack

The problem

- On a crash, the stack is unwounded and the stack reified
- Control is passed to the first catch frame on the stack
- Imagine the function

```
catchWithStack :: Exception e =>
       IO a                     -- Action to run
    -> (e -> Stack -> IO a) -- Handler
    -> IO a
```

- What can `Stack` be?
- Can it really be lazily evaluated?

The problem

- On a crash, the stack is unwounded and the stack reified
- Control is passed to the first catch frame on the stack
- Imagine the function

```
catchWithStack :: Exception e =>
       IO a                   -- Action to run
    -> (e -> Stack -> IO a) -- Handler
    -> IO a
```

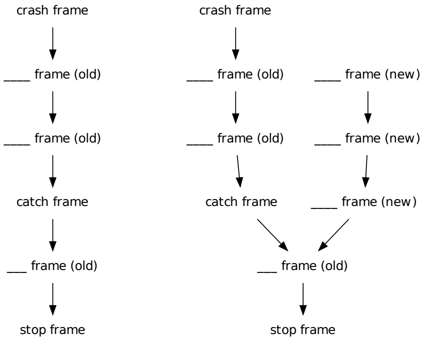
- What can `Stack` be?
- Can it really be lazily evaluated?
- We have to be really careful, the stack is a mutable data structure!

One idea

- Internally, the execution stack is a *chunked linked list*.
- What if we *freeze* the stack and continue our stack in a new chunk?

One idea

- Internally, the execution stack is a *chunked linked list*.
- What if we *freeze* the stack and continue our stack in a new chunk?



-

Why an Haskell interface?

- Compare
  - `gdb` style of stack traces
  - Catching an exception with the stack trace

Why an Haskell interface?

- Compare
  - `gdb` style of stack traces
  - Catching an exception with the stack trace
- The latter is *much* more powerful since we have control over it in Haskell land

Why an Haskell interface?

- Compare
  - `gdb` style of stack traces
  - Catching an exception with the stack trace
- The latter is *much* more powerful since we have control over it in Haskell land
- We can:
  - Print to screen
  - Email it
  - Choose to handle the exception based on if frame X is present on

Why an Haskell interface?

- Compare
    - `gdb` style of stack traces
    - Catching an exception with the stack trace
- The latter is *much* more powerful since we have control over it in Haskell land
- We can:
    - Print to screen
    - Email it
    - Choose to handle the exception based on if frame X is present on
- Definitely a requirement for software running in production

The final Haskell API

base-4.7.0.0: Basic libraries

GHC.ExecutionStack

This is a module for the efficient but inaccurate Stack Traces. If you can take a factor 2 of performance penalty. You should consider using GHC.Stack as the s

```
myFunction :: IO ()
myFunction = do
      stack <- reifyStack
      dumpStack stack
```

An ExecutionStack is a data wrapper around ByteArray#. The Array is a reified stack. Each element can be thought as the Instruction Pointers. For languag function. The ExecutionStack as described by the STG will only contain pointers to entry code of Info Tables.

Simple interface

`reifyStack :: IO ExecutionStack`

Reify the stack. This is the only way to get an ExecutionStack value.

`dumpStack :: ExecutionStack -> IO ()`

Pretty print the stack. Will print it to stdout. Note that this is more efficent than doing `print` as no intermediete Haskell values will get created

Complicated interface

ExecutionStack

`data ExecutionStack`

Constructors

`ExecutionStack`

`unExecutionStack :: ByteArray#`

Instances

`Show ExecutionStack`

`stackSize :: ExecutionStack -> Int`

The number of functions on your stack

`stackIndex :: ExecutionStack -> Int -> Addr#`

- Meh

Final remarks

- It seems possible to create an efficient first-class value of the execution stack that is available post mortem. If my ideas work out this will be *amazing*
- This work will not be so super-useful unless it incorporates with exceptions that Haskell is not aware of, like segmentation faults. Think foreign function calls and Haskell code like:

```
unsafeWrite v 1000000000 (0 :: Int)
```