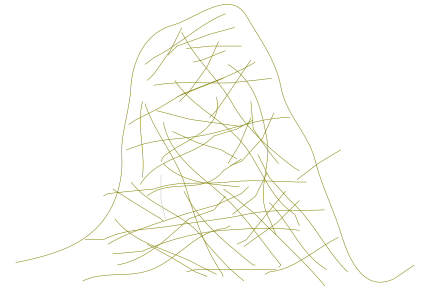
Finding the Needle
Stack Traces for GHC



Tristan O.R. Allwood
Imperial College - tora@doc.ic.ac.uk

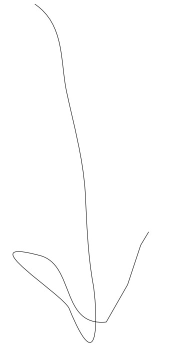
Simon Peyton Jones
Microsoft Research - simonpj@microsoft.com

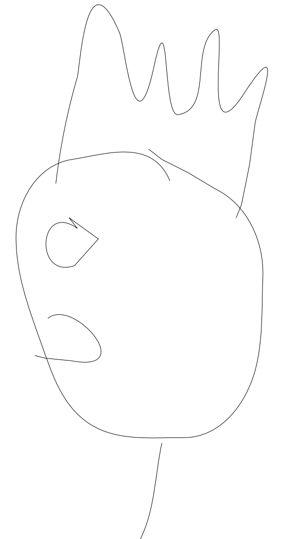
Susan Eisenbach
Imperial College - susan.eisenbach@imperial.ac.uk

Prelude

The adventures of Jake

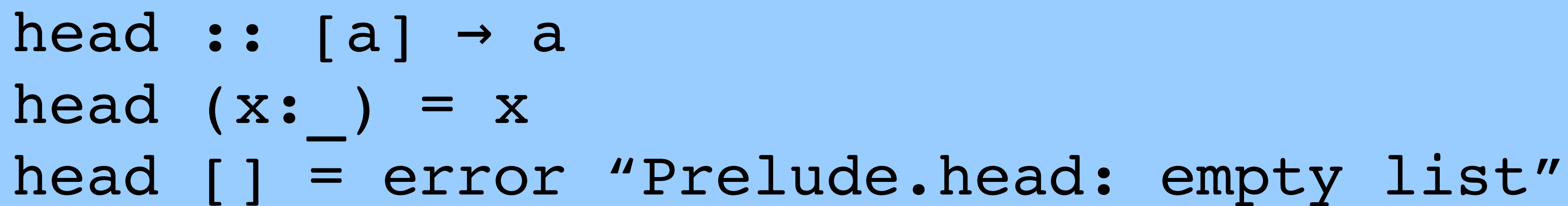
Jake





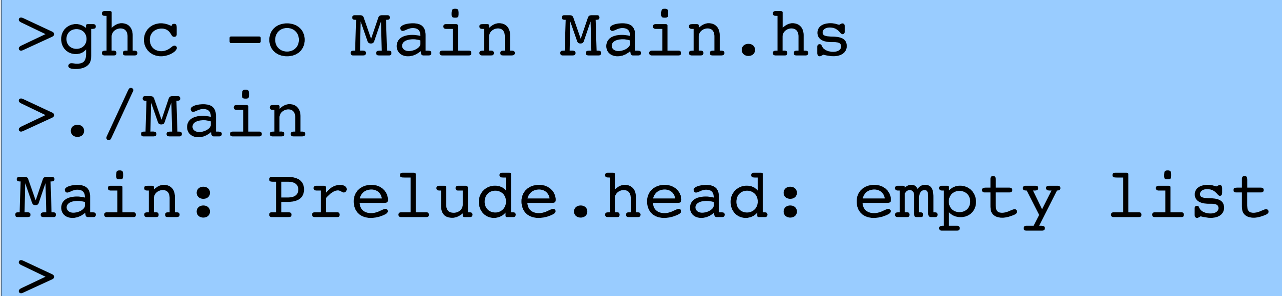
Somewhere nice and sunny...

Prelude

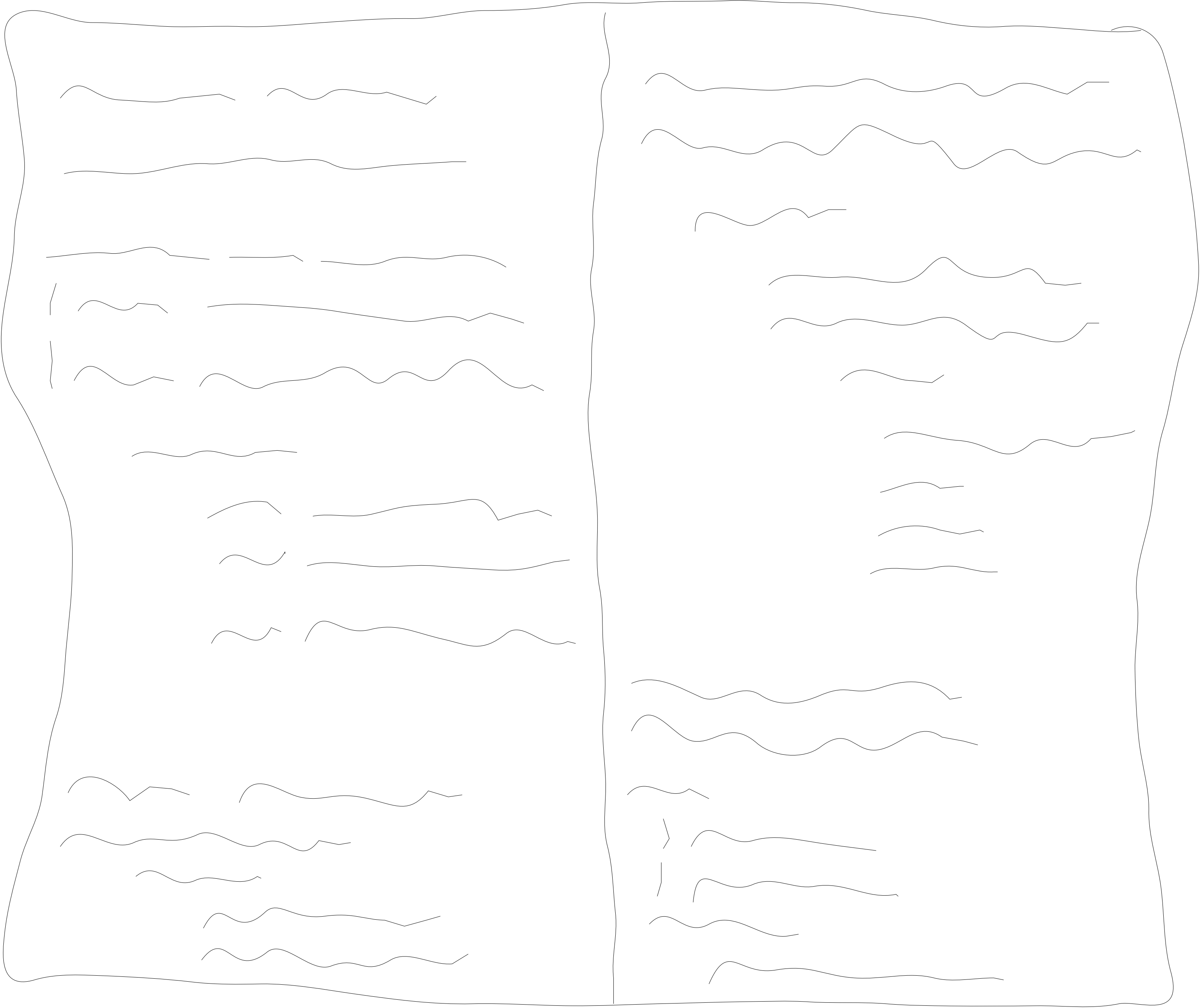


```
head :: [a] → a
head (x:_) = x
head [] = error “Prelude.head: empty list”
```

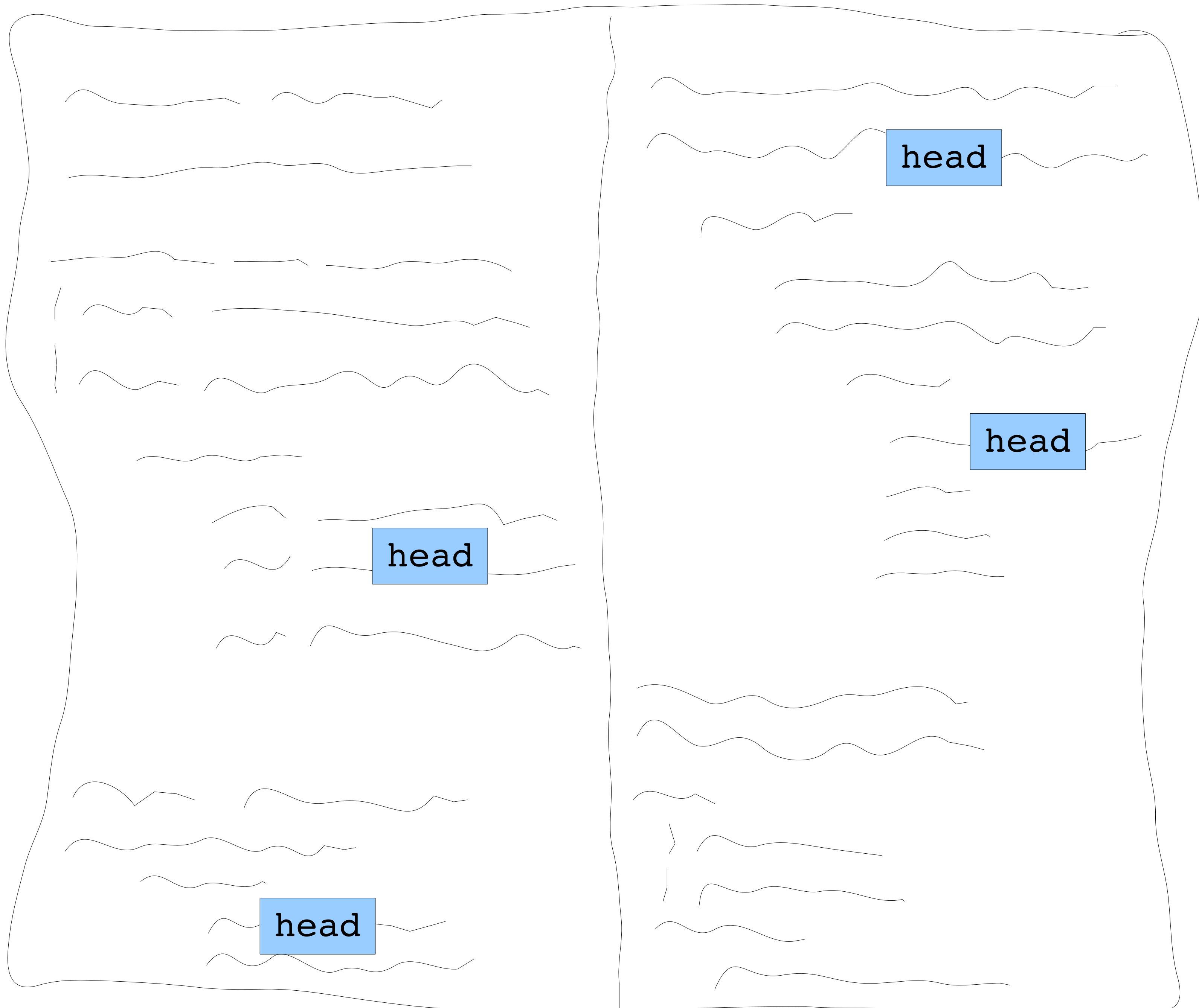
Jake could not have predicted the terror that was about to strike...



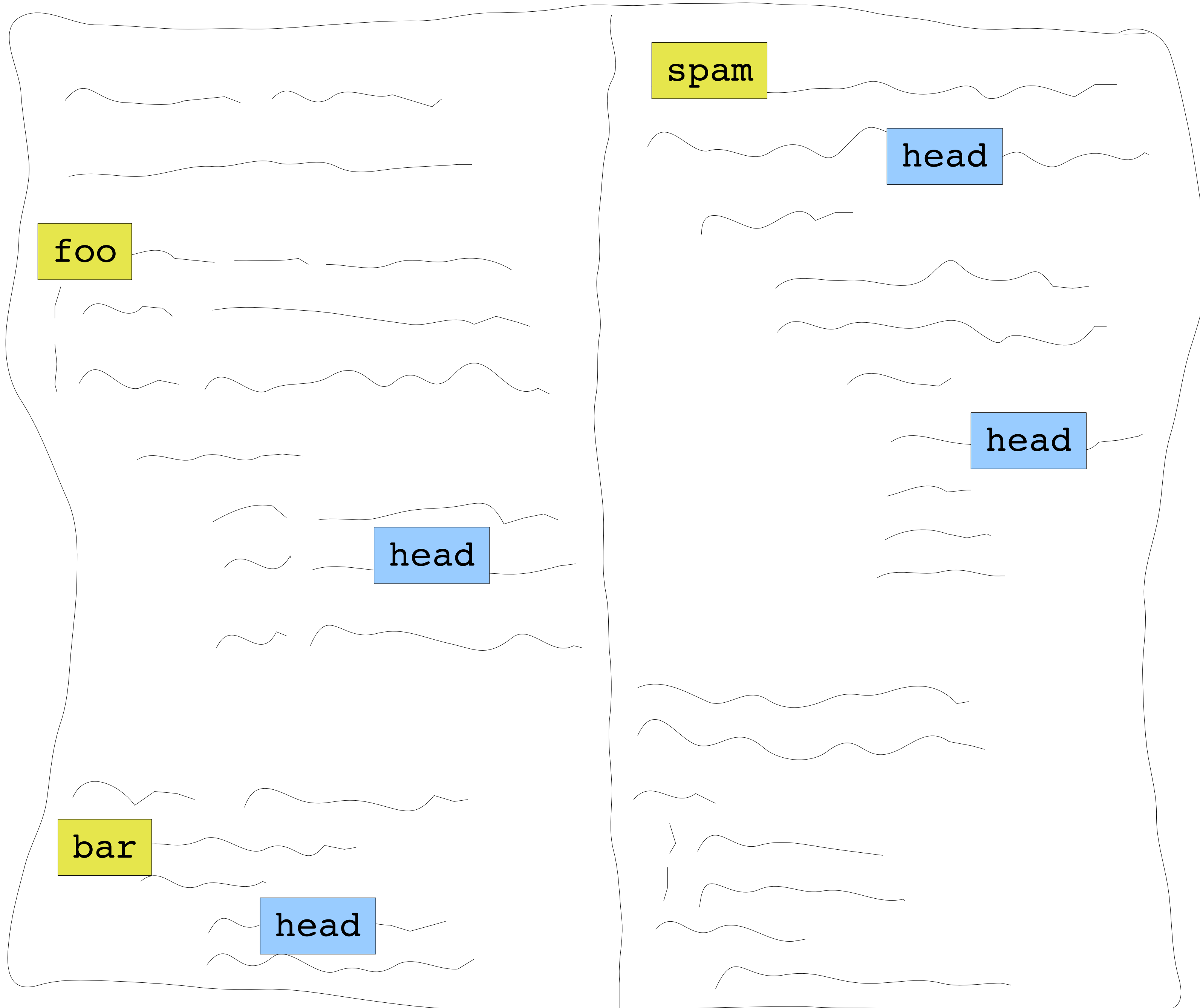
Jake tentatively peeked inside Main...



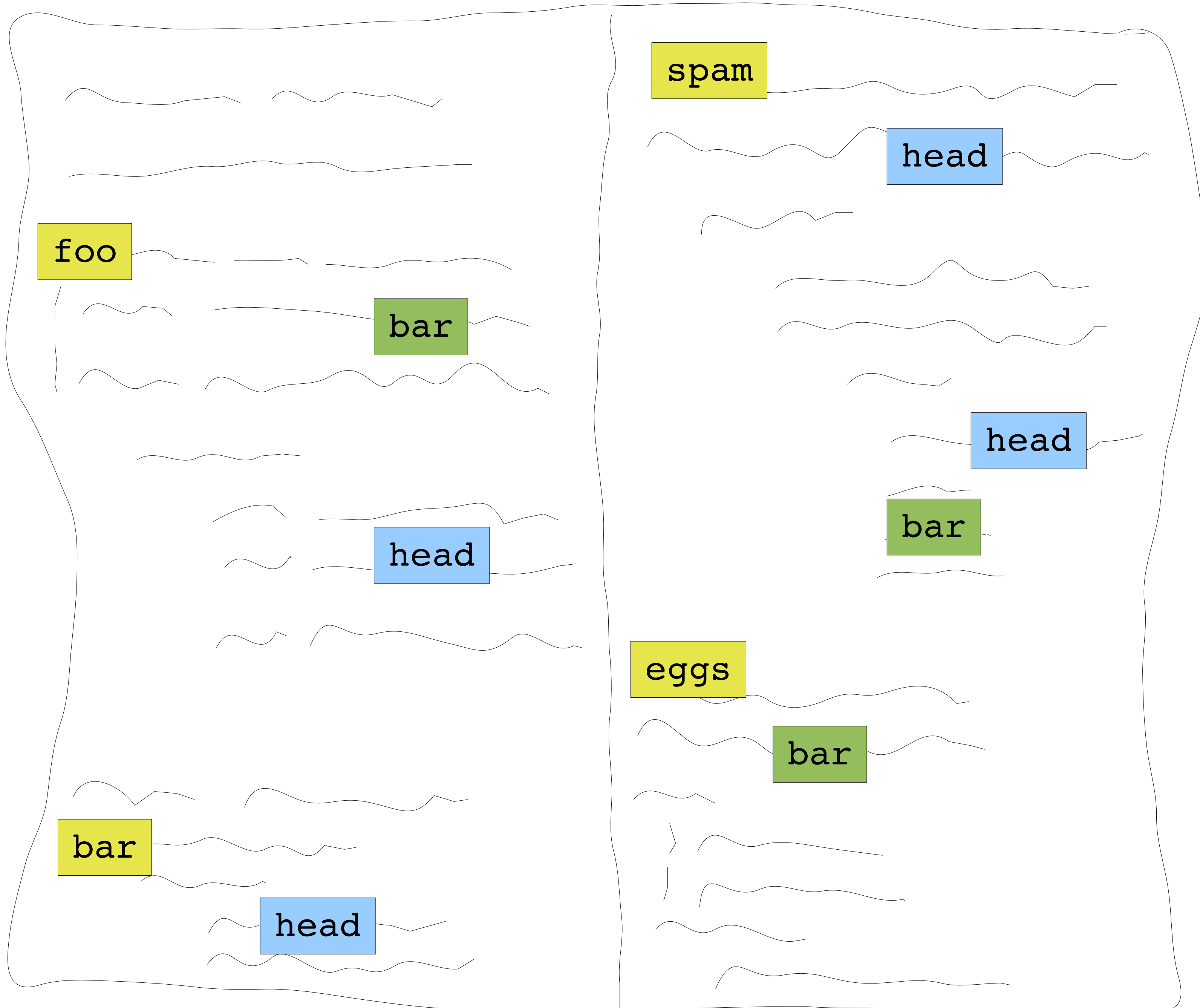
As he stared, things started to focus...



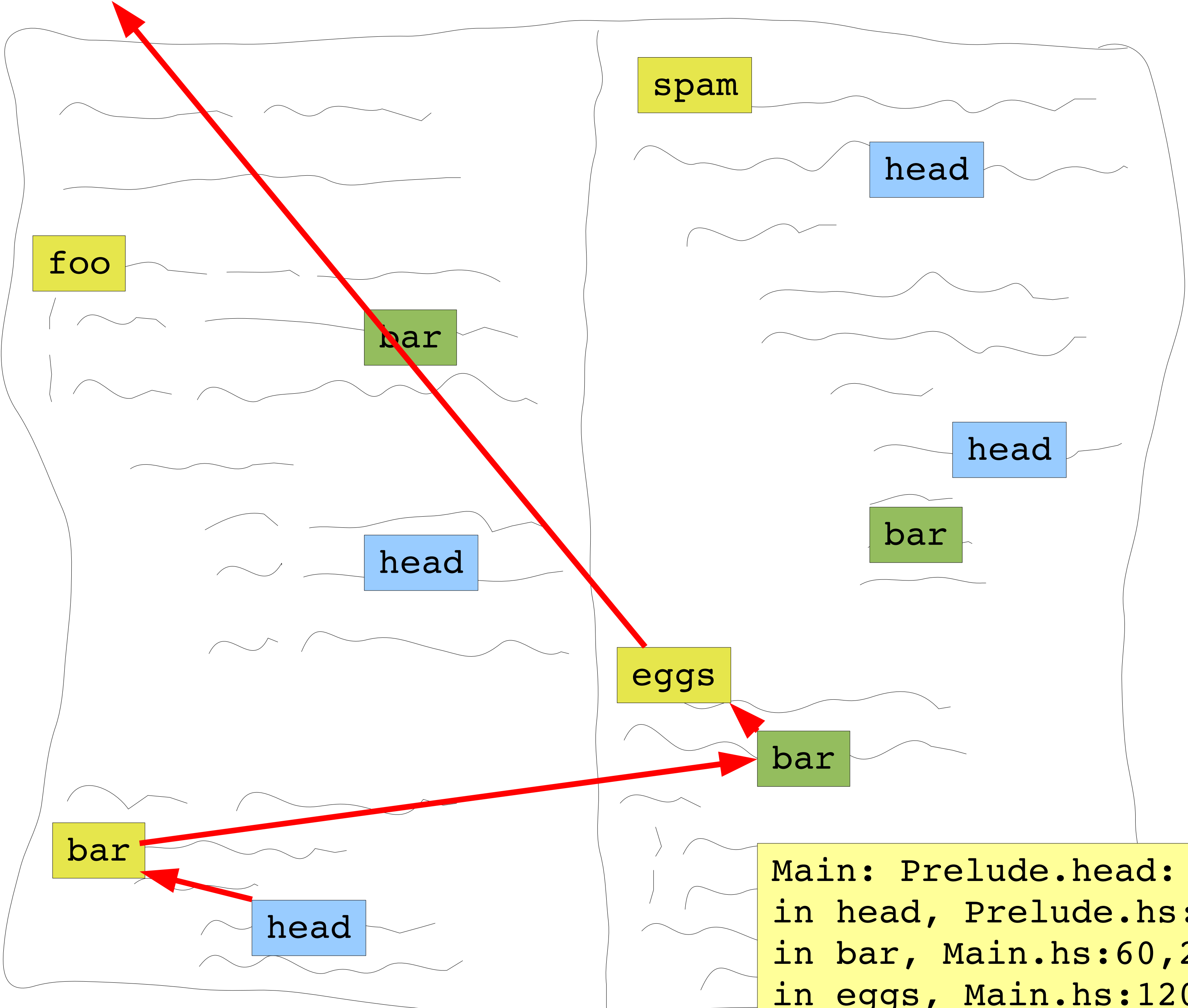
Slowly more context appeared...



But then he saw a complication...

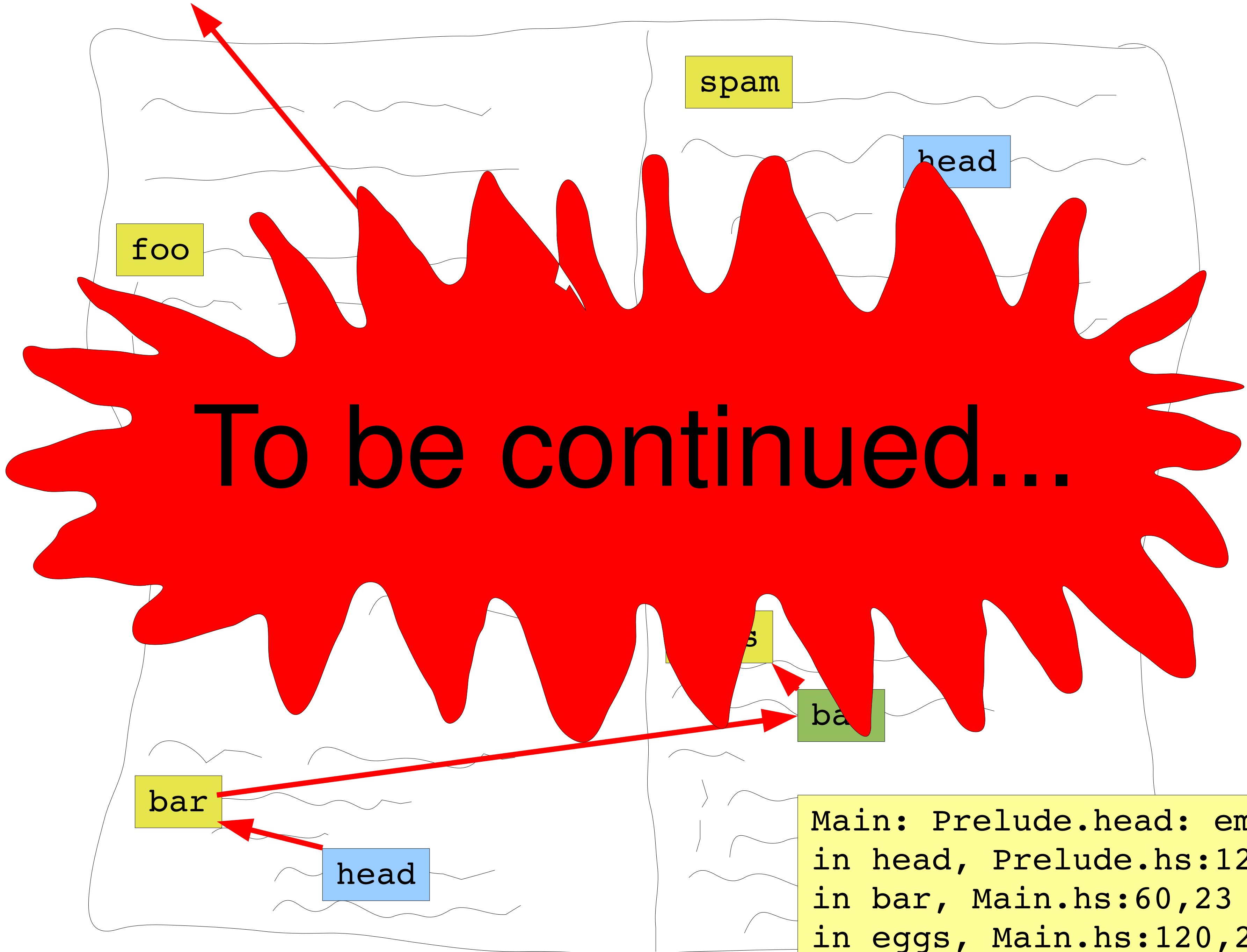


If only there was a way...*

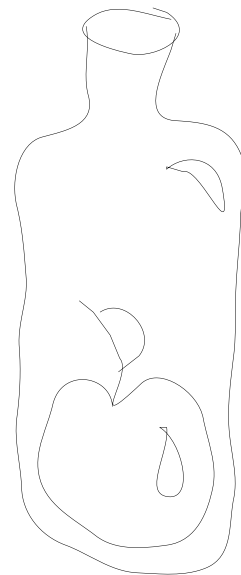



```
Main: Prelude.head: empty list
in head, Prelude.hs:12,23
in bar, Main.hs:60,23
in eggs, Main.hs:120,22
in main, Main.hs:5,5
```

If only there was a way...*



The Problem



```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
    = (fst (mkPair [])) / 2
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
    = (fst (mkPair [])) / 2
    = (fst (head [], False)) / 2
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
    = (fst (mkPair [])) / 2
    = (fst (head [], False)) / 2
    = head [] / 2
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
    = (fst (mkPair [])) / 2
    = (fst (head [], False)) / 2
    = head [] / 2
    = error .... / 2
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
    = (fst (mkPair [])) / 2
    = (fst (head [], False)) / 2
    = head [] / 2
    = error .... / 2
    = error “Prelude.head: empty list”
```

```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
    = (fst (mkPair [])) / 2
    = (fst (head [], False)) / 2
    = head [] / 2
    = error .... / 2
    = error “Prelude.head: empty list”
```

```
Prelude.head: empty list
in halfFst
in main
```

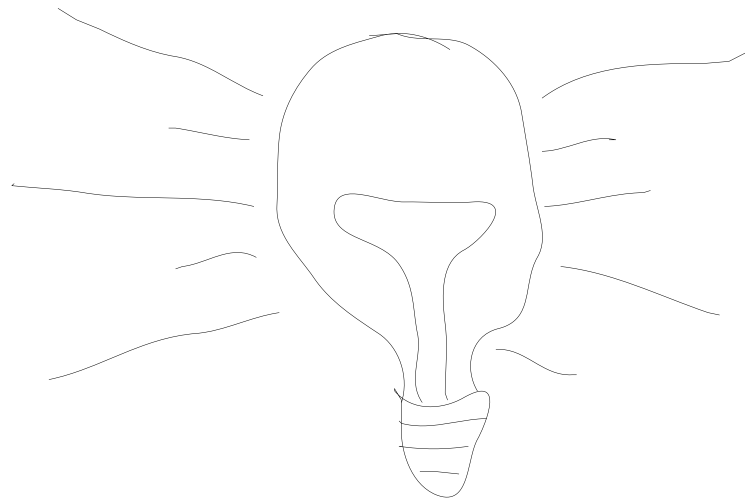
```
main xs = halfFst (mkPair xs)

mkPair xs = (head xs, False)
halfFst tup = (fst tup) / 2
```

```
main []
  = halfFst (mkPair [])
    = (fst (mkPair [])) / 2
    = (fst (head [], False)) / 2
    = head [] / 2
    = error .... / 2
    = error "Prelude.head: empty list"
```

```
Prelude.head: empty list
in halfFst
in main
```

```
Prelude.head: empty list
in mkPair
in main
```

The big idea



```
recip :: Int → Int
recip x = if x == 0 then error “Urk foo”
                                        else 1 / x

bargle x = ... (recip x) ...
```

```
recip :: Int → Int
recip x = if x == 0 then error “Urk foo”
                             else 1 / x

bargle x = ... (recip x) ...
```

```
recip :: Int → Int
recip = recip_deb emptyStack
```

```
recip :: Int → Int
recip x = if x == 0 then error "Urk foo"
                            else 1 / x

bargle x = ... (recip x) ...
```

```
recip :: Int → Int
recip = recip_deb emptyStack

recip_deb :: Stack → Int → Int
```

```
recip :: Int → Int
recip x = if x == 0 then error “Urk foo”
                            else 1 / x

bargle x = ... (recip x) ...
```

```
recip :: Int → Int
recip = recip_deb emptyStack

recip_deb :: Stack → Int → Int
recip_deb stack x
  = if x == 0 then (error_deb stack' “Urk foo”)
              else 1 / x
```

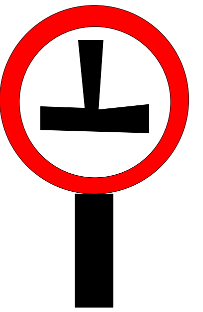
```
recip :: Int → Int
recip x = if x == 0 then error “Urk foo”
                            else 1 / x

bargle x = ... (recip x) ...
```

```
recip :: Int → Int
recip = recip_deb emptyStack

recip_deb :: Stack → Int → Int
recip_deb stack x
  = if x == 0 then (error_deb stack' “Urk foo”)
              else 1 / x
    where
      stack' = push “in recip:Ex.hs:14,25” stack
```

```
recip :: Int → Int
recip x = if x == 0 then error “Urk foo”
                             else 1 / x

bargle x = ... (recip x) ...
```

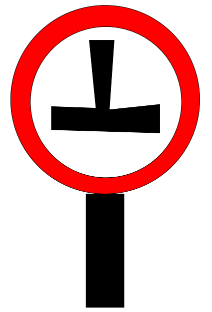
```
recip :: Int → Int
recip = recip_deb emptyStack

recip_deb :: Stack → Int → Int
recip_deb stack x
  = if x == 0 then (error_deb stack' “Urk foo”)
                else 1 / x
    where
      stack' = push “in recip:Ex.hs:14,25” stack

bargle_deb stack x = ... (recip_deb stack' x) ...
  where
    stack' = push “in bargle:Ex.hs:17,32” stack
```

error



```
error :: [Char] → a
error m = throwStack (\s → ErrorCall (m ++ show s))
```

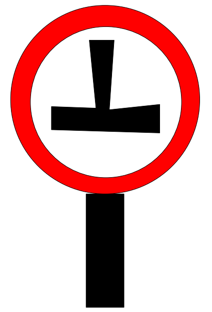
error



```
error :: [Char] → a
error m = throwStack (\s → ErrorCall (m ++ show s))
```

```
module Stack where
  emptyStack :: Stack
  push :: Location → Stack → Stack
  throwStack :: Exception e => (Stack → e) → a
```

error

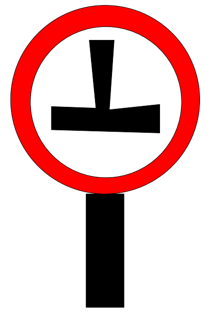


```
error :: [Char] → a
error m = throwStack (\s → ErrorCall (m ++ show s))
```

```
error :: [Char] → a
error = error_deb emptyStack
```

```
module Stack where
  emptyStack :: Stack
  push :: Location → Stack → Stack
  throwStack :: Exception e => (Stack → e) → a
```

error

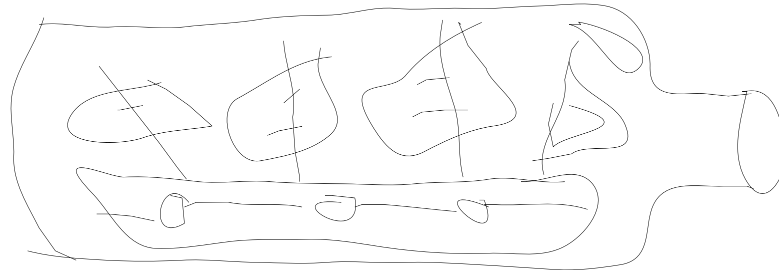


```
error :: [Char] → a
error m = throwStack (\s → ErrorCall (m ++ show s))
```

```
error :: [Char] → a
error = error_deb emptyStack

error_deb :: Stack → [Char] → a
error_deb stack m = throw (ErrorCall (m ++ show stack))
```

```
module Stack where
   emptyStack :: Stack
   push :: Location → Stack → Stack
   throwStack :: Exception e => (Stack → e) → a
```

User guided Stack Traces



```
ghc.exe: panic! (the 'impossible' happened)
  (GHC version 6.11.20081202 for i386-unknown-mingw32):
        idInfo
```

```
ghc.exe: panic! (the 'impossible' happened)
  (GHC version 6.11.20081202 for i386-unknown-mingw32):
        idInfo
```

```
varIdInfo :: Var → IdInfo
varIdInfo (GlobalId {idInfo_ = info}) = info
varIdInfo (LocalId {idInfo_ = info}) = info
varIdInfo other_var = pprPanic "idInfo" (ppr other_var)
```

```
ghc.exe: panic! (the 'impossible' happened)
  (GHC version 6.11.20081202 for i386-unknown-mingw32):
        idInfo
```

```
import GHC.ExplicitCallStack.Annotations

{-# ANN varIdInfo Debug #-}
varIdInfo :: Var → IdInfo
varIdInfo (GlobalId {idInfo_ = info}) = info
varIdInfo (LocalId {idInfo_ = info}) = info
varIdInfo other_var
  = throwStack (\s → pprPanic ("idInfo\n" ++ show s)
                              (ppr other_var) :: SomeException)
```

```
ghc.exe: panic! (the 'impossible' happened)
  (GHC version 6.11.20081202 for i386-unknown-mingw32):
        idInfo
in varIdInfo, basicTypes/Var.lhs:238,30
in idInfo, basicTypes/Id.lhs:168,10
```

```
import GHC.ExplicitCallStack.Annotations

{-# ANN varIdInfo Debug #-}
varIdInfo :: Var → IdInfo
varIdInfo (GlobalId {idInfo_ = info}) = info
varIdInfo (LocalId {idInfo_ = info}) = info
varIdInfo other_var
  = throwStack (\s → pprPanic ("idInfo\n" ++ show s)
                              (ppr other_var) :: SomeException)
```

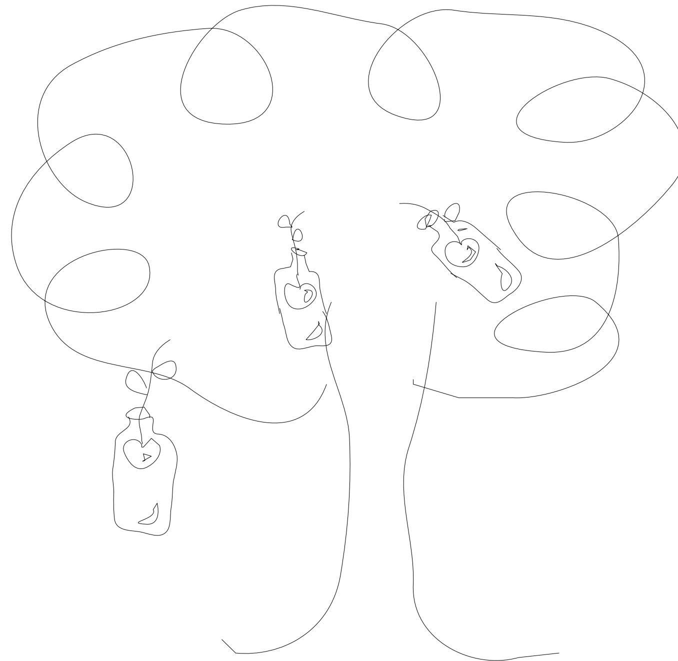
```
ghc.exe: panic! (the 'impossible' happened)
  (GHC version 6.11.20081202 for i386-unknown-mingw32):
        idInfo
in varIdInfo, basicTypes/Var.lhs:238,30
in idInfo, basicTypes/Id.lhs:168,10
in idInlinePragma, basicTypes/Id.lhs:633,37
```

```
import GHC.ExplicitCallStack.Annotations

{-# ANN varIdInfo Debug #-}
varIdInfo :: Var → IdInfo
varIdInfo (GlobalId {idInfo_ = info}) = info
varIdInfo (LocalId {idInfo_ = info}) = info
varIdInfo other_var
  = throwStack (\s → pprPanic ("idInfo\n" ++ show s)
                              (ppr other_var) :: SomeException)
```

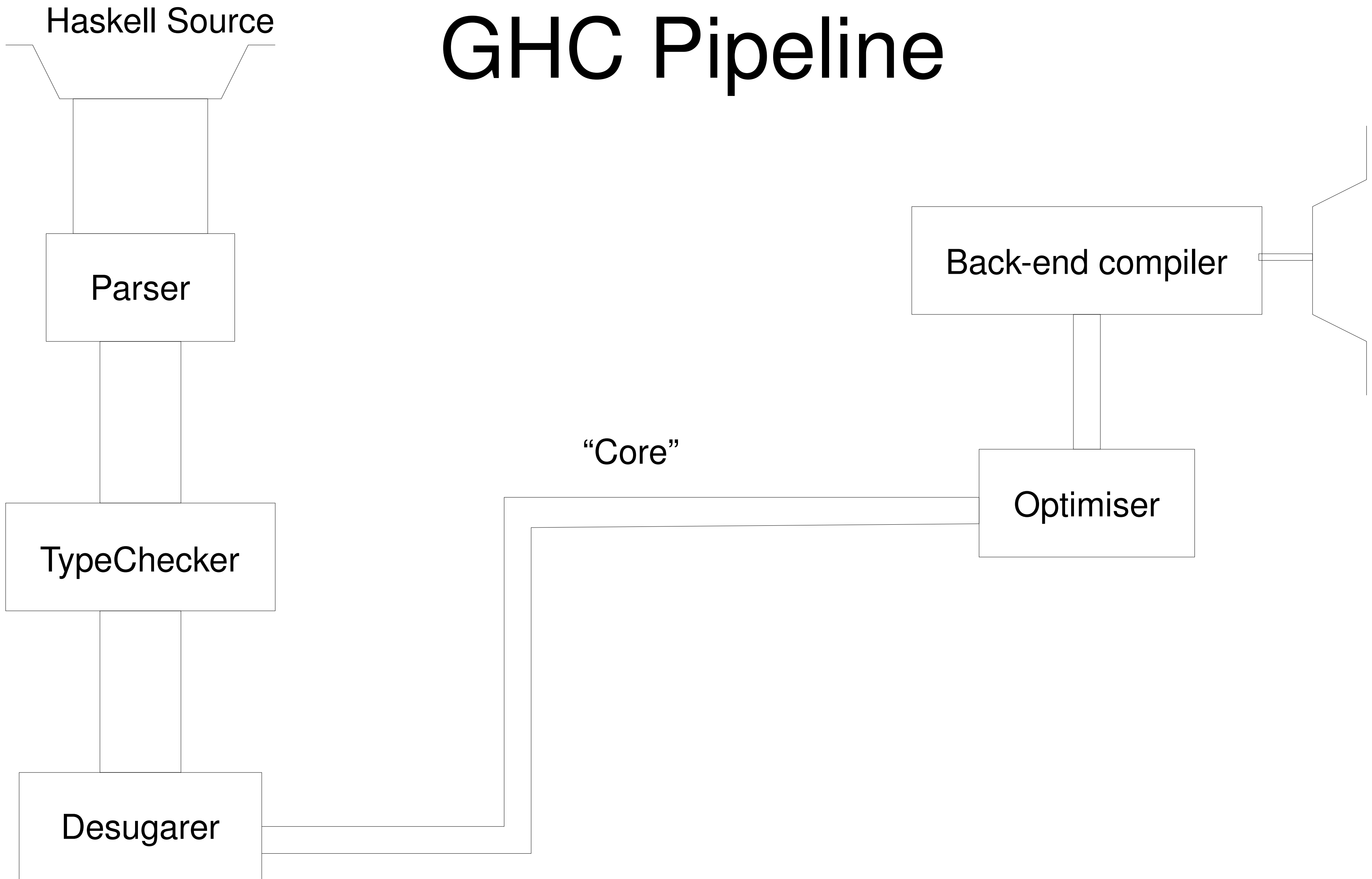
```
ghc.exe: panic! (the 'impossible' happened)
  (GHC version 6.11.20081202 for i386-unknown-mingw32):
        idInfo
in varIdInfo, basicTypes/Var.lhs:238,30
in idInfo, basicTypes/Id.lhs:168,10
in idInlinePragma, basicTypes/Id.lhs:633,37
in preInlineUnconditionally,
simplCore/SimplUtils.lhs:619,12
in simplNonRecE, simplCore/Simplify.lhs:964,5
in simplLam, simplCore/Simplify.lhs:925,13
in simplExprF', simplCore/Simplify.lhs:754,5
in simplExprF, simplCore/Simplify.lhs:741,5
```

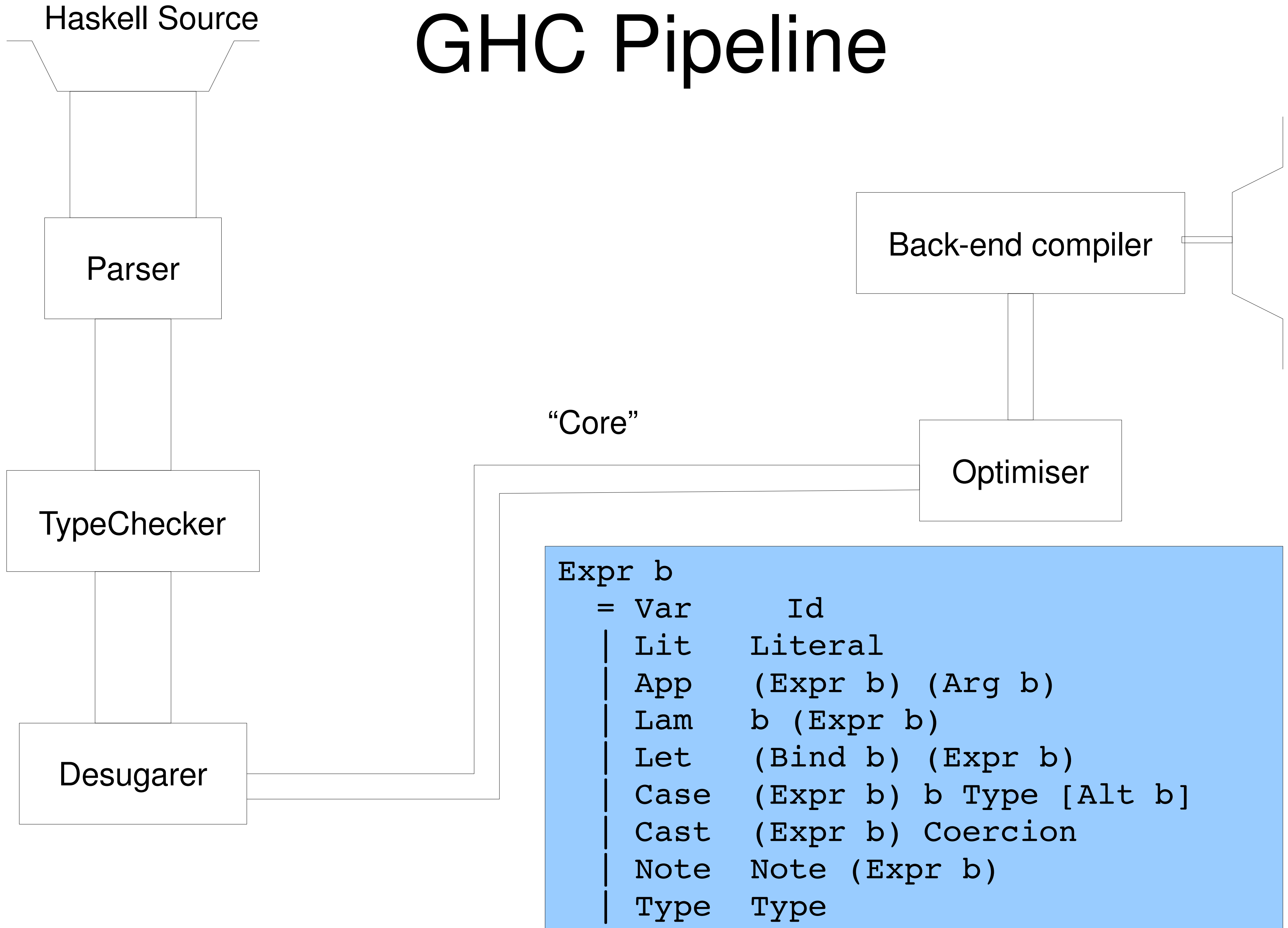
```
import GHC.ExplicitCallStack.Annotations

{-# ANN varIdInfo Debug #-}
varIdInfo :: Var → IdInfo
varIdInfo (GlobalId {idInfo_ = info}) = info
varIdInfo (LocalId {idInfo_ = info}) = info
varIdInfo other_var
  = throwStack (\s → pprPanic ("idInfo\n" ++ show s)
                              (ppr other_var) :: SomeException)
```

How does StackTrace do the rewriting?

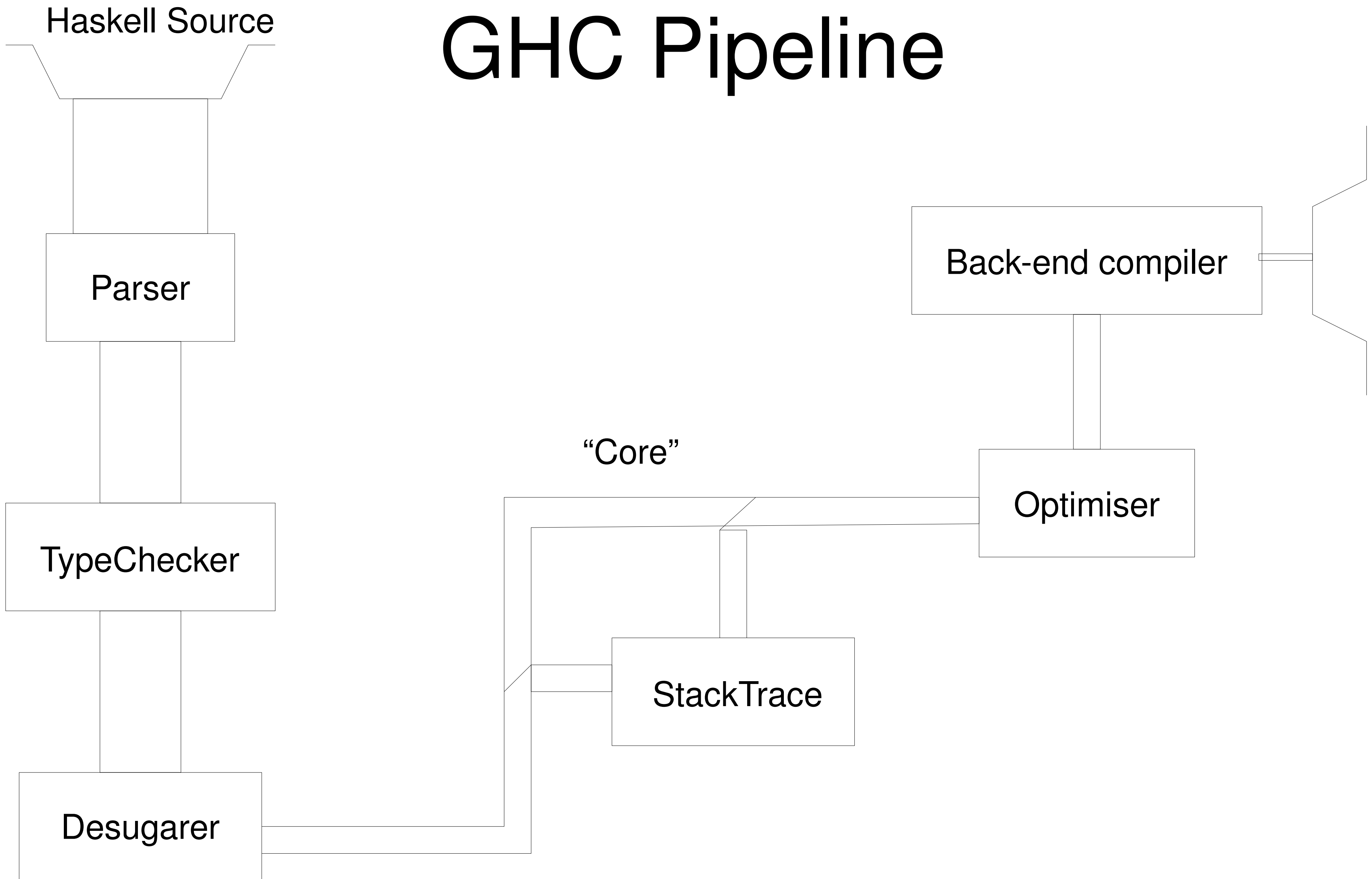


GHC Pipeline




GHC Pipeline
Haskell Source
Parser
TypeChecker
Desugarer
"Core"
Optimiser
Back-end compiler
Expr b
= Var Id
| Lit Literal
| App (Expr b) (Arg b)
| Lam b (Expr b)
| Let (Bind b) (Expr b)
| Case (Expr b) b Type [Alt b]
| Cast (Expr b) Coercion
| Note Note (Expr b)
| Type Type

GHC Pipeline



Equations

[| f = e |]	→ f = f_deb emptyStack → f_deb s = $[\|e\|]_s$	if f_deb is to be made
[| f = e |]	→ f = $[\|e\|]_{emptyStack}$	otherwise
$[\|throwStack\|]_s$	→ \f → throw (f s)	
$[\|x\|]_s$	→ x_deb (push l s)	if x_deb exists
$[\|x\|]_s$	→ x	otherwise
$[\|e1\ e2\|]_s$	→ $[\|e1\|]_s$ $[\|e2\|]_s$	
$[\|\backslash x \to e\|]_s$	→ \x → $[\|e\|]_s$	
$[\|case\ e1\ of\ p \to e2\|]_s$	→ case $[\|e1\|]_s$ of p → $[\|e2\|]_s$	
$[\|let\ x = e1\ in\ e2\|]_s$	→ let x = $[\|e1\|]_s$ in $[\|e2\|]_s$	

```
throwStack :: Exception e => (Stack → e) → a
```

Stacks, stacks, stacks

```
module Stack where
  emptyStack :: Stack
  push :: Location → Stack → Stack
  throwStack :: Exception e => (Stack → e) → a
```

Stacks, stacks, stacks

```
foo 0 = error “End of foo!”
foo n = bar (n – 1)

bar n = foo n

main = print (foo 1000)
```

```
>./Main
Main: End of foo!
in error, Main.hs:16,14
in foo, Main.hs:9,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
```

```
foo 0 = error “End of foo!”
foo n = bar (n – 1)

bar n = foo n

main = print (foo 1000)
```

```
in foo, Main.hs:9,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in main, Main.hs:6,15
```

```
>./Main
Main: End of foo!
in error, Main.hs:16,14
in foo, Main.hs:9,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
```

```
foo 0 = error "End of foo!"
foo n = bar (n – 1)

bar n = foo n

main = print (foo 1000)
```

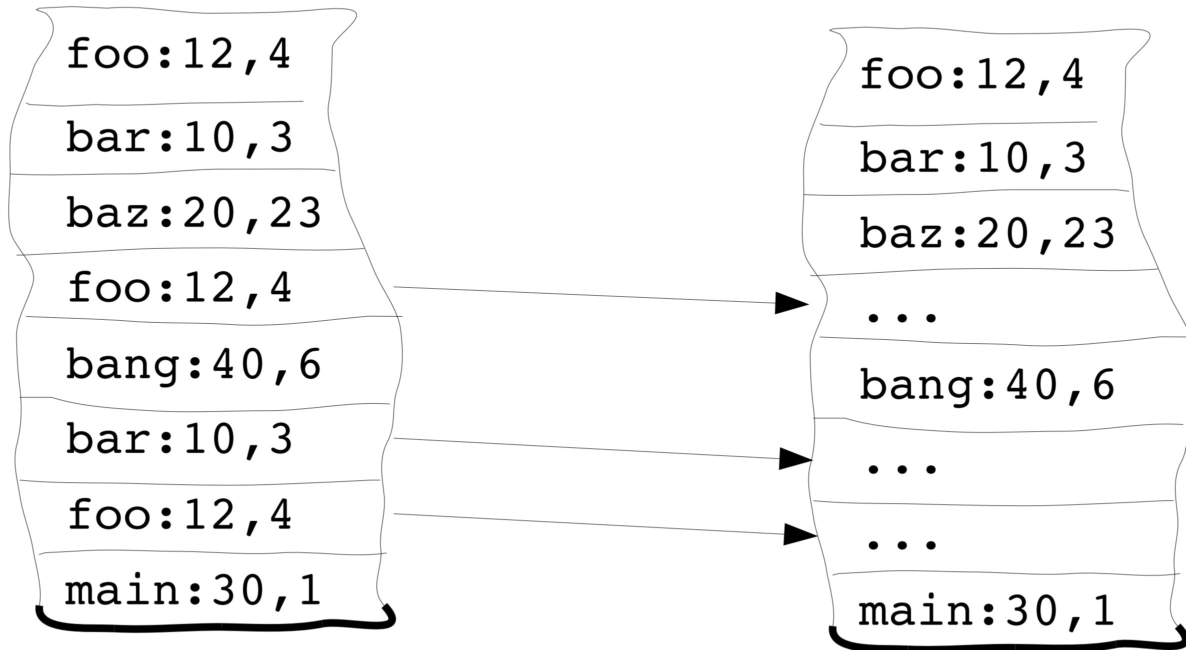
```
Main: End of foo!
in error, Main.hs:16,14
in foo, Main.hs:9,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
...
in main, Main.hs:6,15
```

```
in foo, Main.hs:9,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in bar, Main.hs:13,9
in foo, Main.hs:10,9
in main, Main.hs:6,15
```

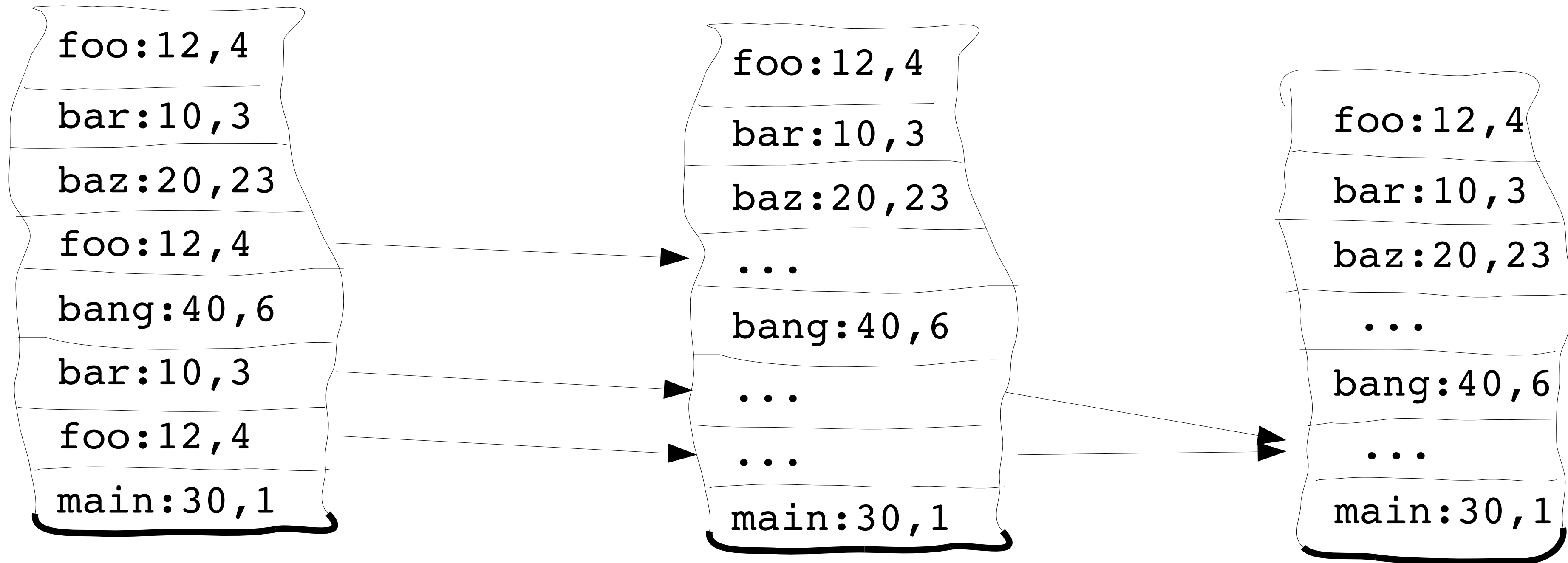
Intuition

foo:12,4
bar:10,3
baz:20,23
foo:12,4
bang:40,6
bar:10,3
foo:12,4
main:30,1

Intuition



Intuition



Properties

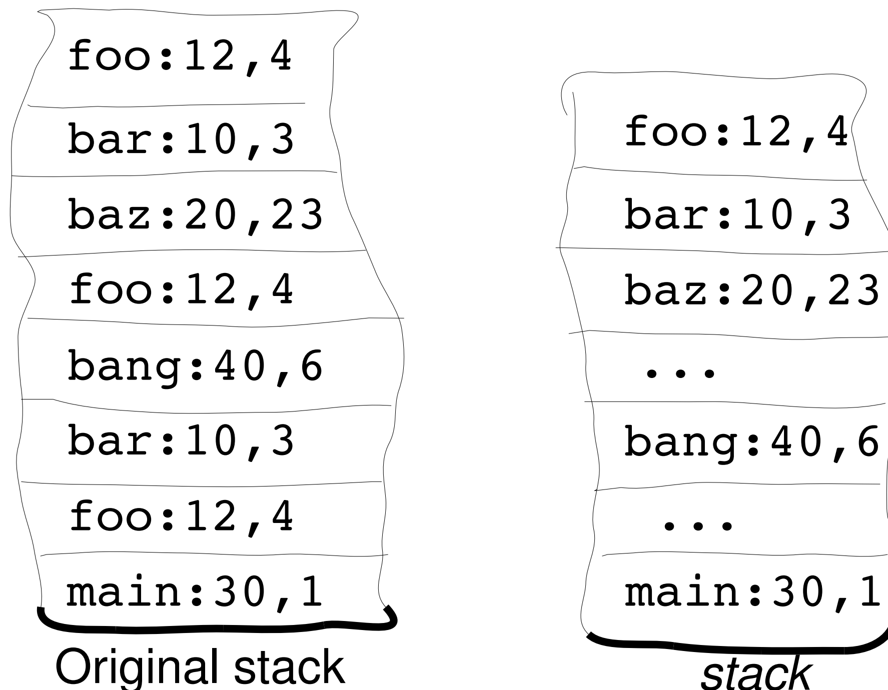
Any call site in the original stack is in the *stack*

Top of *stack* to the first “...” is same as original stack

“...”s only elide call sites that are above them in the *stack*

“...”s are never adjacent

The size of a *stack* is now statically bounded at twice the number of call sites in the program



See the paper for...

Stack implementation

Stack tail hash-consing

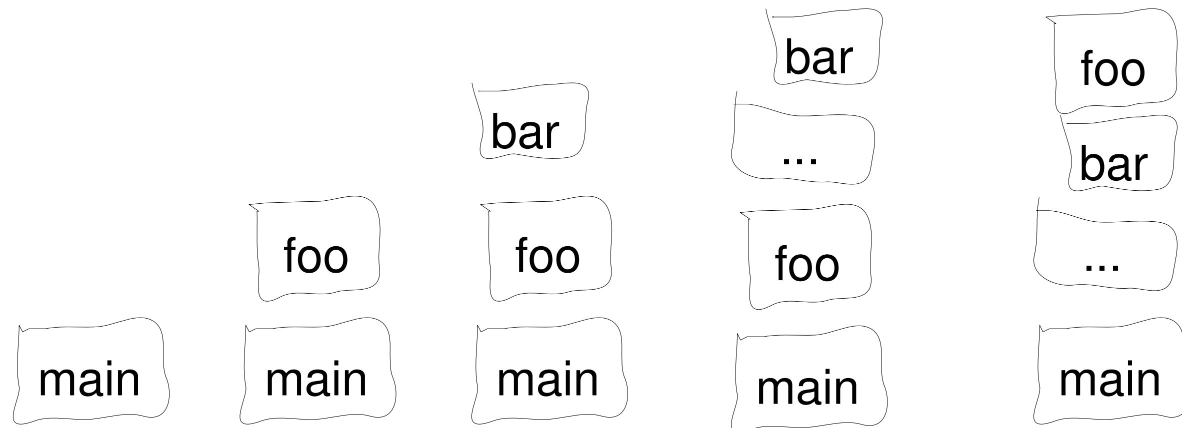
Gory details of how push works

See the paper for...

Stack implementation

Stack tail hash-consing

Gory details of how push works

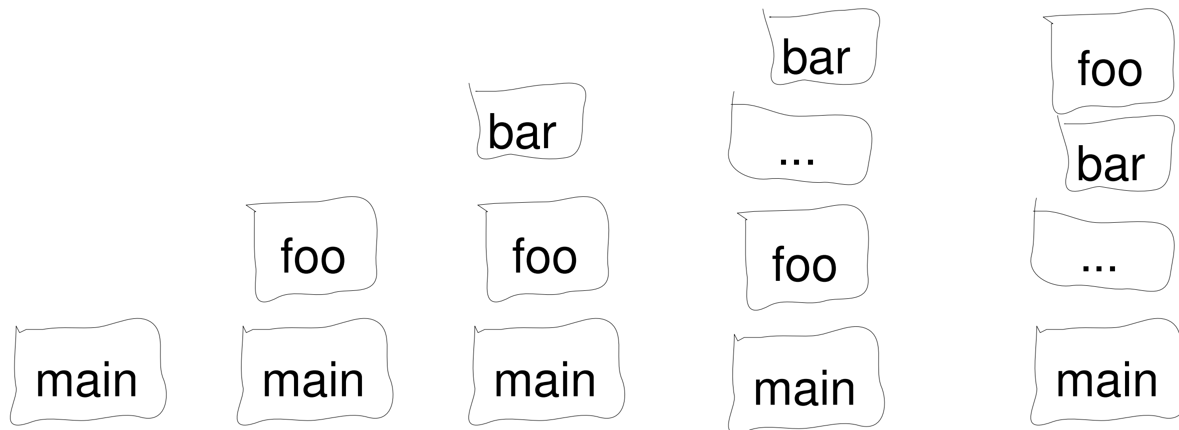


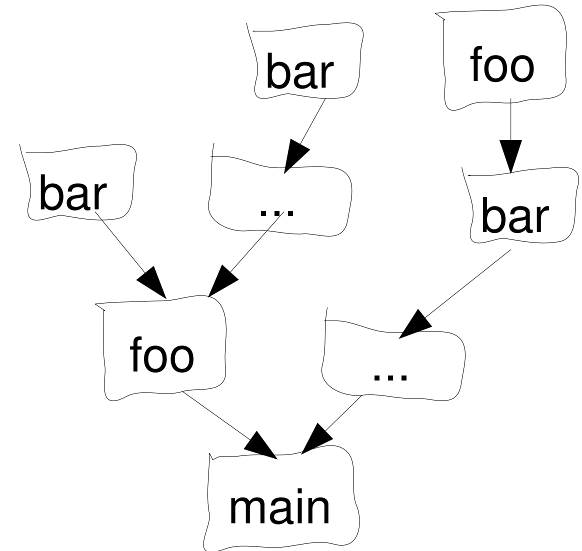
See the paper for...

Stack implementation

Stack tail hash-consing

Gory details of how push works





But wait, there's more!

Tricky cases / open problems:

- Desugaring recursive polymorphic functions
- CAFs
- TypeClasses
- Bootstrapping this into existence

But wait, there's more!

- Related Work
  - GHCi debugger
    - S. Marlow, J. Iborra, B. Pope, A. Gill
  - HPC
    - A. Gill, C. Runciman
  - JHC's SRCLOC_ANNOTATE
    - J. Meacham
  - HAT
    - M. Wallace, O. Chitil, T. Brehm, C. Runciman
  - GHC Cost Center Stacks
    - P. Sansom, S. Peyton Jones

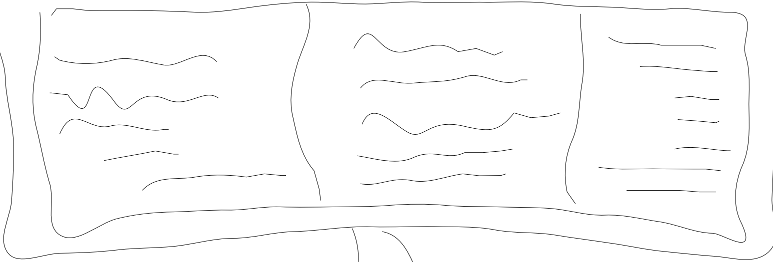
But wait, there's more!

Patches, paper, performance results & slides

`code.haskell.org/explicitCallStackPaper/`

Jake recompiled with an explicit Stack Trace

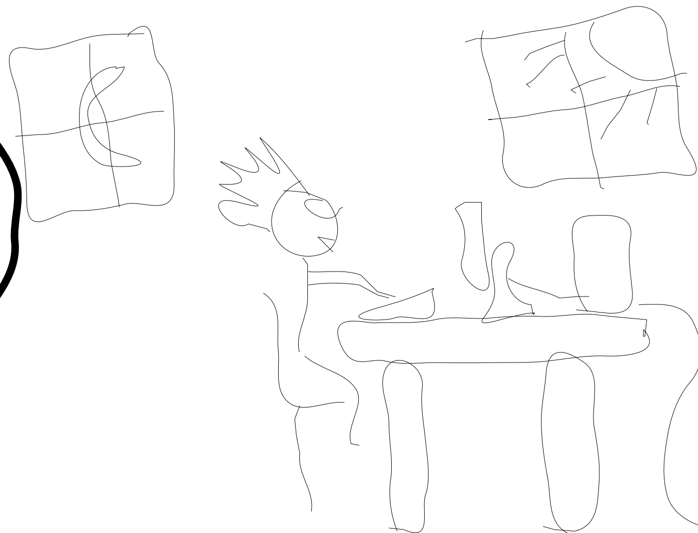
```
> ghc -fexplicit-call-stack-all -o Main Main.hs
```

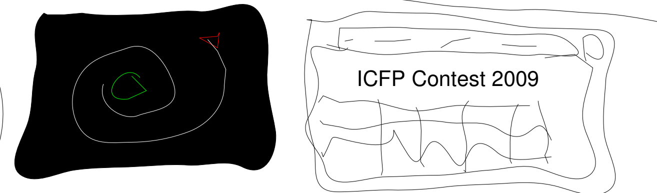


```
./Main
Main: Prelude.head: empty list
in head, Prelude.hs:12,23
in calibrateSensors.hs:20,24
in bar, Main.hs:60,23
in bang, Main.hs:120,22
in main, Main.hs:5,5
```

And re-ran his program to get the trace

After debugging for 3 days straight...






```
> ./Main
Simulation running...
Success! Satellite intercepted
Score: 201
```

...Jake got his program to work!

Thank-you for listening!

Tricky open problems

Desugaring woes

```
f 0 = error "Argh!"
f x = g (x – 1)
g x = f x
```

Desugaring woes

```
f 0 = error “Argh!”
f x = g (x – 1)
g x = f x
```

```
fg_tuple = \dict_num →
  let {
    dict_eq = getEqDict dict_num

    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error “Argh”
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → f_lcl

g = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

Desugaring woes

```
f 0 = error "Argh!"
f x = g (x – 1)
g x = f x
```

```
fg_tuple = \dict_num →
  let {
    dict_eq = getEqDict dict_num

    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error "Argh"
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → f_lcl

g = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

Desugaring woes

```
f 0 = error “Argh!”
f x = g (x – 1)
g x = f x
```

```
fg_tuple = \dict_num →
  let {
    dict_eq = getEqDict dict_num

    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error “Argh”
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → f_lcl

g = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

Desugaring woes

```
f 0 = error "Argh!"
f x = g (x – 1)
g x = f x
```

```
fg_tuple = \dict_num →
  let {
    dict_eq = getEqDict dict_num

    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error "Argh"
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → f_lcl

g = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

Desugaring woes

```
f 0 = error “Argh!”
f x = g (x – 1)
g x = f x
```

```
fg_tuple = \dict_num →
  let {
    dict_eq = getEqDict dict_num

    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error “Argh”
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → f_lcl

g = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

Desugaring woes

```
f 0 = error "Argh!"
f x = g (x – 1)
g x = f x
```

```
fg_tuple = \dict_num →
  let {
    dict_eq = getEqDict dict_num

    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error "Argh"
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → f_lcl

g = \dict_num → case (fg_tuple dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

Desugaring woes

```
f 0 = error “Argh!”
f x = g (x – 1)
g x = f x
```

```
fg_tuple_d = \s dict_num →
  let {
    dict_eq = getEqDict dict_num
    s' = push “in fg_tuple:Blah.hs...” s
    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error s' “Argh”
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f_d = \s dict_num → case (fg_tuple_d (push “in f:...” s) dict_num) of
    (f_lcl, g_lcl) → f_lcl

g_d = \s dict_num → case (fg_tuple_d (push “in g:...” s) dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

Desugaring woes

```
f 0 = error "Argh!"
f x = g (x – 1)
g x = f x
```

```
fg_tuple_d = \s dict_num →
  let {
    dict_eq = getEqDict dict_num
    s' = push "in fg_tuple:Blah.hs..." s
    f_lcl = \x → case (((==) dict_eq) 0 x) of
      True → error s' "Argh"
      False → g_lcl (((-) dict_num) x 1)
    g_lcl = \x → f_lcl x
  } in  (f_lcl, g_lcl)

f_d = \s dict_num → case (fg_tuple_d (push "in f:..." s) dict_num) of
    (f_lcl, g_lcl) → f_lcl

g_d = \s dict_num → case (fg_tuple_d (push "in g:..." s) dict_num) of
    (f_lcl, g_lcl) → g_lcl
```

```
error: yikes!
in fg_tuple:Blah.hs
in f:Blah.hs
in main
```

CAFs

```
e = expensive `seq` f

main = do
  print e
  print e
```

CAFs

```
e = expensive `seq` f

main = do
  print e
  print e
```

```
e_deb stack = expensive_deb stack1 `seq` f_deb stack2
   where
      stack1 = ...
      stack2 = ...
```