Prelude.head: empty list

# How do we get more information?

- Compile with –prof, run with +RTS –xc

```
<GHC.List.CAF>empty: Prelude.head: empty list
```

- Hmm

# More ideas

- GHCi debugger
  - Not really a stack trace, but can be useful
- mapException-type things
  - Requires work on the part of the user
- Tristan Allwood's "Finding the needle" (Haskell Symposium 2009)
  - automated source-to-source transformation guided by user annotations, difficulties with CAFs
- HPC
  - Look at which bits of your program are coloured
- Hat
  - A bit dormant

# A unified framework?

- Profiling, HPC, and the GHCi debugger
  - All require feeding source-code locations through to compiled code in some way
  - HPC and GHCi share some infrastructure (ticks), but profiling is done differently (cost centres)
  - We also want:
    - stack tracing
    - cheap flat annotations for parallel profiling
  - how can we share the infrastructure?
  - Profiling generates stacks of a kind (cost centre stacks), why don't they work for error reporting?

# Profiling: background

- Cost Centre profiling (Sansom thesis '94)
  - main innovation: cost centres abstract away from evaluation order, by saving and restoring the cost centre when evaluating a thunk
  - only flat profiling, no stacks (we later extended this to stacks)
  - identified a difference between "lexical scoping" and "evaluation scoping"
- Difference: in "main = map f xs"
  - Lexical scoping: [main,f]
  - Evaluation scoping: [main,map,f]
- Ways to think about it:
  - Eval scoping gives the stack that you would see in a strict language
  - Lexical scoping gives you the stack obtained by tracing from uses to (top-level) definitions in the source program, eval scoping gives you a dynamic call stack
  - In eval scoping, an annotation (scc) captures the costs of evaluating an expression to NF but not under lambdas, whereas lexical scoping also captures the repeated costs under lambdas
- Lots of other differences: e.g.
  - "f = foo . bar", f does not appear in the stack with eval
  - "let f x = ... in f 1 + f 2", no way to identify individual calls with lexical scoping
  - in general, lexical scoping makes fewer distinctions

# Lexical or evaluation scoping?

- Sansom advocated a hybrid scheme
- GHC implements stacks with lexical scoping (or tries to)

- Plan: switch to evaluation scoping
  - lexical scoping has undesirable properties:
    - the need to box higher-order arguments, “let x = y in f x” is not the same as “f y”!
    - lexical scoping restricts the transformations that apply
    - never managed to find a semantics that worked properly for CAFs
  - Eval scoping is far easier. Deleted lots of code from GHC.
    - Easy to explain: “this is the stack you’d get in a strict language”
    - one drawback: every lambda needs an annotation, whereas with lexical scoping only need one annotation at the top level
    - But, we can use this to give more information: where bindings get identified separately.

# Back to the unified framework

- Add two constructs to Core:

```
tick s E
```

- Where s is a source code location. A tick *counts entries.*

```
scc s E
```

- (for "set cost centre", naming subject to change)
- This is an annotation that *scopes over E* (in an evaluation-scoping way).
- For stack tracing, scc corresponds to pushing an item on the stack while evaluating E.

# How do we use these?

- HPC decorates the program with tick
- Profiling/auto-all decorates the program with both scc and tick
  - because profiling counts entries too
  - will have an option to use scc only, sacrificing entry counts for optimisation
- Flat (parallel) profiling uses scc only
- Stack tracing uses scc
- All decorations are added by one pass, parameterised by the strategy (formerly the Coverage module in GHC).
- Main point: by giving a semantics to these two constructs only, we can implement many different profiling/debugging features

# Transformations

- Important: we want as many optimisations to apply as possible, while retaining the correct semantics. e.g.

```
(scc s \x . e)  =>  \x . e
```

- no evaluation to do: lambda is in HNF, so can discard scc
  - but could not discard tick: remember it has to count the evaluation!
- Accept transformations that make a small change to the cost attribution. After all: all optimisations change the cost attribution in some way.
- Inlining or floating lambda: always valid (eval scoping only!)
- Inlining a redex: no!
- Important: the semantics tells us which transformations are valid.

# Status

- scc and tick implemented (actually one construct with flags internally)
- HPC and GHCi debugger adapted, working
- Profiling:
  - now obeys eval scoping
  - -auto-all decorates nested bindings too
  - entry counts are correct
  - more optimisations apply than before: profiled code should be closer to –O, -fhpc should generate faster code.
- all this might be in 7.4

```
COST CENTRE         MODULE                       no.     entries  %time %alloc   %time %alloc

MAIN                MAIN                         103           0    0.0    0.0   100.0  100.0
 res                Main                         207           1    0.0    0.0   100.0  100.0
  disp              Main                         239          20    0.0    0.0     0.0    0.0
   interleave       Main                         240         100    0.0    0.0     0.0    0.0
  unicl             Main                         232          20    0.0    0.0    34.6   55.5
   unicl.unicl'     Main                         233      106920    0.0    1.0    34.6   55.5
    insert          Main                         238          20    0.0    0.0     0.0    0.0
    tautclause      Main                         236      106920    0.0    3.4     0.0    3.4
    clause          Main                         234      106920    0.0    1.5    34.6   51.1
     clause.clause' Main                         235     1989000   11.5   46.7    34.6   49.6
      insert        Main                         237     1047960   23.1    2.9    23.1    2.9
  split             Main                         229          20    0.0    0.0     0.0    3.4
   split.split'     Main                         230      213820    0.0    3.4     0.0    3.4
  disin             Main                         228     2450280   57.7   40.7    65.4   40.7
   conjunct         Main                         231     2169580    7.7    0.0     7.7    0.0
  negin             Main                         227        4620    0.0    0.1     0.0    0.1
  elim              Main                         226        3980    0.0    0.1     0.0    0.1
  parse             Main                         209          20    0.0    0.0     0.0    0.0
   parse.(...)      Main                         210          20    0.0    0.0     0.0    0.0
    parse'          Main                         211         800    0.0    0.0     0.0    0.0
     while          Main                         222          60    0.0    0.0     0.0    0.0
      red           Main                         225          40    0.0    0.0     0.0    0.0
      spri          Main                         223          60    0.0    0.0     0.0    0.0
       opri         Main                         224          40    0.0    0.0     0.0    0.0
     parse'.(...)   Main                         216          60    0.0    0.0     0.0    0.0
      while         Main                         218         180    0.0    0.0     0.0    0.0
       red          Main                         221         120    0.0    0.0     0.0    0.0
       spri         Main                         219         180    0.0    0.0     0.0    0.0
        opri        Main                         220         180    0.0    0.0     0.0    0.0
     spri           Main                         214         160    0.0    0.0     0.0    0.0
      opri          Main                         215         140    0.0    0.0     0.0    0.0
     opri           Main                         213         160    0.0    0.0     0.0    0.0
```

```haskell
module Main where

import qualified Data.Map as Map
import Data.List (nub)
import Data.Char (digitToInt)

data Coord =  Coord !Int !Int
              deriving (Show,Eq,Ord)

(|+|) :: Coord -> Coord -> Coord
(|+|) (Coord x1 y1) (Coord x2 y2)
   = Coord (x1 + x2) (y1 + y2)

isAccessible :: Coord -> Bool
isAccessible = (<=25) . sumCoord
  where digits    = map digitToInt . show
        sumDigits = sum . digits
        sumCoord (Coord x y)
           = sumDigits x + sumDigits y

reachableCoords :: Coord -> [Coord]
reachableCoords c = map ((|+|) c) $
possibleMoves
  where possibleMoves =
          [Coord 1    0
          ,Coord (-1) 0
          ,Coord 0    1
          ,Coord 0    (-1)
          ]

type CoordMap = Map.Map Coord ()
emptyMap = Map.empty :: CoordMap
```

```haskell
walk2 :: [Coord] -> CoordMap -> Int -> Int
walk2 []        _    count = count
walk2 (x:stack) seen count
   | isOld x          = walk2 stack  seen  count
   | isAccessible x   = walk2 stack' seen' (count+1)
   | otherwise        = walk2 stack  seen' count
  where seen'   = Map.insert x () seen
        stack'  = reachableCoords x ++ stack
        isOld p = Map.member p $ seen

walk :: [Coord] -> CoordMap -> Int -> Int
walk []      _    count = count
walk surface seen count = walk new seen' count'
  where poss    = nub . concat
                .  map reachableCoords $ surface
        new     = filter (\p -> isNew p &&
                            isAccessible p) $ poss
        seen'   = foldr (\k m ->
                     Map.insert k () m) seen poss
        count'  = count + length new
        isNew p = not . Map.member p $ seen

main = print $ walk2 [Coord 1000 1000] emptyMap 0

main' = print $ walk [start] seen 1
  where start = Coord 1000 1000
        seen  = Map.insert start () emptyMap
```

# GHC 7.0.3 –O2

```
./ants +RTS -K32m -i0.01 -s
148848
     268,071,308 bytes allocated in the heap
      64,002,300 bytes copied during GC
       7,485,040 bytes maximum residency (7 sample(s))
         128,532 bytes maximum slop
              19 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:   505 collections,     0 parallel,  0.16s,  0.16s elapsed
  Generation 1:     7 collections,     0 parallel,  0.10s,  0.10s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    0.87s  (  0.87s elapsed)
  GC    time    0.25s  (  0.25s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    1.13s  (  1.13s elapsed)
```

# GHC 7.0.3 –O2 –prof –auto-all

```
./ants +RTS -K32m -i0.01 -s
148848
     495,854,100 bytes allocated in the heap
     102,483,988 bytes copied during GC
      28,129,260 bytes maximum residency (8 sample(s))
      15,826,960 bytes maximum slop
              54 MB total memory in use (0 MB lost due to fragmentation)

  Generation 0:   874 collections,     0 parallel,  4.13s,  4.14s elapsed
  Generation 1:     8 collections,     0 parallel,  0.12s,  0.12s elapsed

  INIT  time    0.00s  (  0.00s elapsed)
  MUT   time    5.44s  (  5.46s elapsed)
  GC    time    4.25s  (  4.26s elapsed)
  RP    time    0.00s  (  0.00s elapsed)
  PROF  time    0.00s  (  0.00s elapsed)
  EXIT  time    0.00s  (  0.00s elapsed)
  Total time    9.69s  (  9.73s elapsed)
```

# Simon's GHC –O2 –prof –auto-all

```
   474,715,900 bytes allocated in the heap
   132,442,188 bytes copied during GC
    23,746,884 bytes maximum residency (9 sample(s))
       154,652 bytes maximum slop
            47 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
Gen  0       878 colls,     0 par    0.26s    0.26s     0.0003s    0.0008s
Gen  1         9 colls,     0 par    0.17s    0.17s     0.0194s    0.0577s

INIT    time    0.00s  (  0.00s elapsed)
MUT     time    1.37s  (  1.37s elapsed)
GC      time    0.43s  (  0.43s elapsed)
RP      time    0.00s  (  0.00s elapsed)
PROF    time    0.00s  (  0.00s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    1.80s  (  1.80s elapsed)
```

# GHC 7.0.3 –O2 –prof –auto-all

```
COST CENTRE              MODULE                  no.    entries  %time %alloc

MAIN                     MAIN                      1          0    0.0    0.0
 CAF                     Main                    268         10    0.0    0.0
  reachableCoords        Main                    278          0    0.0    0.0
  main                   Main                    274          1    0.0    0.0
   walk2                 Main                    276     595394   69.5   48.2
    |+|                  Main                    279     595392    0.4    5.8
    isAccessible         Main                    277     535707   30.1   46.0
   emptyMap              Main                    275          1    0.0    0.0
 CAF                     GHC.IO.Handle.FD        204          2    0.0    0.0
 CAF                     GHC.IO.Encoding.Iconv   162          2    0.0    0.0
 CAF                     GHC.Conc.Signal         159          1    0.0    0.0
```

# Simon’s GHC –O2 –prof –auto-all

```
                                                                  individual
COST CENTRE                 MODULE                  no.     entries  %time %alloc

MAIN                        MAIN                    114           0    0.0    0.0
 CAF                        GHC.IO.Handle.FD        142           0    0.0    0.0
 CAF                        GHC.IO.Encoding.Iconv   134           0    0.0    0.0
 CAF                        GHC.Conc.Signal         124           0    0.0    0.0
 CAF                        Main                    119           0    0.0    0.0
  isAccessible              Main                    232           1    0.0    0.0
  emptyMap                  Main                    231           1    0.0    0.0
  main                      Main                    228           1    0.0    0.0
   walk2                    Main                    229      595394   33.3   41.8
    compare                 Main                    237     3286902    0.7    0.0
    reachableCoords         Main                    234      148848    5.7   15.1
     |+|                    Main                    235      595392    0.0    0.0
    isAccessible.sumCoord   Main                    233      178569   17.7   42.0
    walk2.isOld             Main                    230      595393   31.2    1.1
     compare                Main                    236    10678924   11.3    0.0
```

# What about stack tracing?

- Well, now we have the mechanism in Core
- Certainly need to compile code for stack tracing, with scc's added
  - don't really want to have to use profiling to get stack traces
    - also, profiling uses the wrong decoration strategy, for stack tracing we want *call sites*
  - need to make sure that errors in CAFs get useful stack traces
  - We could provide stack tracing in GHCi for interpreted code, because the program is already decorated for the GHCi debugger.
  - tricky bit: can we track call stacks in a program that is only partially annotated, and get partial information?
    - when evaluating an unannotated thunk, have to save/restore the stack.

# Work in progress!

- Comments/questions welcome