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`
  - `scc s E`

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

  \((\text{sc}c\ s\ \chi .\ e) \Rightarrow \chi .\ e\)

• no evaluation to do: lambda is in HNF, so can discard sc
  – 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
<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no.</th>
<th>entries</th>
<th>%time</th>
<th>%alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>103</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>res</td>
<td>Main</td>
<td>207</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>disp</td>
<td>Main</td>
<td>239</td>
<td>20</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>interleave</td>
<td>Main</td>
<td>240</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>unicl</td>
<td>Main</td>
<td>232</td>
<td>20</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>unicl.unicl</td>
<td>Main</td>
<td>233</td>
<td>106920</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td>insert</td>
<td>Main</td>
<td>238</td>
<td>20</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>tautclause</td>
<td>Main</td>
<td>236</td>
<td>106920</td>
<td>0.0</td>
<td>3.4</td>
</tr>
<tr>
<td>clause</td>
<td>Main</td>
<td>234</td>
<td>106920</td>
<td>0.0</td>
<td>1.5</td>
</tr>
<tr>
<td>clause.clause</td>
<td>Main</td>
<td>235</td>
<td>1989000</td>
<td>11.5</td>
<td>46.7</td>
</tr>
<tr>
<td>insert</td>
<td>Main</td>
<td>237</td>
<td>1047960</td>
<td>23.1</td>
<td>2.9</td>
</tr>
<tr>
<td>split</td>
<td>Main</td>
<td>229</td>
<td>20</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>split.split'</td>
<td>Main</td>
<td>230</td>
<td>213820</td>
<td>0.0</td>
<td>3.4</td>
</tr>
<tr>
<td>disin</td>
<td>Main</td>
<td>228</td>
<td>2450280</td>
<td>57.7</td>
<td>40.7</td>
</tr>
<tr>
<td>conjunct</td>
<td>Main</td>
<td>231</td>
<td>2169580</td>
<td>7.7</td>
<td>0.0</td>
</tr>
<tr>
<td>negin</td>
<td>Main</td>
<td>227</td>
<td>4620</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>elim</td>
<td>Main</td>
<td>226</td>
<td>3980</td>
<td>0.0</td>
<td>0.1</td>
</tr>
<tr>
<td>parse</td>
<td>Main</td>
<td>209</td>
<td>20</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>parse.(...)</td>
<td>Main</td>
<td>210</td>
<td>20</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>parse'</td>
<td>Main</td>
<td>211</td>
<td>800</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>while</td>
<td>Main</td>
<td>222</td>
<td>60</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>red</td>
<td>Main</td>
<td>225</td>
<td>40</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>spri</td>
<td>Main</td>
<td>223</td>
<td>60</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>opri</td>
<td>Main</td>
<td>224</td>
<td>40</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>parse'.(...)</td>
<td>Main</td>
<td>216</td>
<td>60</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>while</td>
<td>Main</td>
<td>218</td>
<td>180</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>red</td>
<td>Main</td>
<td>221</td>
<td>120</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>spri</td>
<td>Main</td>
<td>219</td>
<td>180</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>opri</td>
<td>Main</td>
<td>220</td>
<td>180</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>spri</td>
<td>Main</td>
<td>214</td>
<td>160</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>opri</td>
<td>Main</td>
<td>215</td>
<td>140</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>opri</td>
<td>Main</td>
<td>213</td>
<td>160</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
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

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)

<table>
<thead>
<tr>
<th></th>
<th>Tot time (elapsed)</th>
<th>Avg pause</th>
<th>Max pause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen 0</td>
<td>878 colls, 0 par</td>
<td>0.26s</td>
<td>0.0003s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.26s</td>
<td>0.0008s</td>
</tr>
<tr>
<td>Gen 1</td>
<td>9 colls, 0 par</td>
<td>0.17s</td>
<td>0.0194s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.17s</td>
<td>0.0577s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no.</th>
<th>entries</th>
<th>%time</th>
<th>%alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>1</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>Main</td>
<td>268</td>
<td>10</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>reachableCoords</td>
<td>Main</td>
<td>278</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>main</td>
<td>Main</td>
<td>274</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>walk2</td>
<td>Main</td>
<td>276</td>
<td>595394</td>
<td>69.5</td>
<td>48.2</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td></td>
<td>Main</td>
<td>279</td>
<td>595392</td>
</tr>
<tr>
<td>isAccessible</td>
<td>Main</td>
<td>277</td>
<td>535707</td>
<td>30.1</td>
<td>46.0</td>
</tr>
<tr>
<td>emptyMap</td>
<td>Main</td>
<td>275</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>GHC.IO.Handle.FD</td>
<td>204</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>GHC.IO.Encoding.Iconv</td>
<td>162</td>
<td>2</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>GHC.Conc.Signal</td>
<td>159</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
### Simon’s GHC –O2 –prof –auto-all

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no.</th>
<th>entries</th>
<th>%time</th>
<th>%alloc</th>
<th>individual %time</th>
<th>%alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN</td>
<td>MAIN</td>
<td>114</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>CAF</td>
<td>GHC.IO.Handle.FD</td>
<td>142</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>GHC.IO.Encoding.Iconv</td>
<td>134</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>GHC.Conc.Signal</td>
<td>124</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>CAF</td>
<td>Main</td>
<td>119</td>
<td>0</td>
<td>0.0</td>
<td>0.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>isAccessible</td>
<td>Main</td>
<td>232</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>emptyMap</td>
<td>Main</td>
<td>231</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>main</td>
<td>Main</td>
<td>228</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>walk2</td>
<td>Main</td>
<td>229</td>
<td>595394</td>
<td>33.3</td>
<td>41.8</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>compare</td>
<td>Main</td>
<td>237</td>
<td>3286902</td>
<td>0.7</td>
<td>0.0</td>
<td>0.7</td>
<td>0.0</td>
</tr>
<tr>
<td>reachableCoords</td>
<td>Main</td>
<td>234</td>
<td>148848</td>
<td>5.7</td>
<td>15.1</td>
<td>5.7</td>
<td>15.1</td>
</tr>
<tr>
<td>l+</td>
<td>l+</td>
<td>Main</td>
<td>235</td>
<td>595392</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>isAccessible.sumCoord</td>
<td>Main</td>
<td>233</td>
<td>178569</td>
<td>17.7</td>
<td>42.0</td>
<td>17.7</td>
<td>42.0</td>
</tr>
<tr>
<td>walk2.isOld</td>
<td>Main</td>
<td>230</td>
<td>595393</td>
<td>31.2</td>
<td>1.1</td>
<td>31.2</td>
<td>1.1</td>
</tr>
<tr>
<td>compare</td>
<td>Main</td>
<td>236</td>
<td>10678924</td>
<td>11.3</td>
<td>0.0</td>
<td>11.3</td>
<td>0.0</td>
</tr>
</tbody>
</table>
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