CS 410/510: Advanced Programming

Profiling in Haskell

Mark P Jones
Portland State University
What makes a good program?

- Qualitative factors:
  - Correctness
  - Maintainability, readability, understandability, portability, flexibility, ...
  - Use of appropriate abstractions and idioms
  - ...

- Quantitative factors:
  - Performance, Predictability, ...
  - Time, Memory, Disk, Bandwidth, ...
Understanding Program Behavior:

- High-level languages abstract away from the underlying machine

- This can make it very difficult to understand what is happening when a program executes

- Analytic techniques can predict asymptotic trends

- Hard to model complexities of memory, timing, stack, cache, disk, buffers, network, latencies, bandwidth, concurrency, branch prediction, ...
Profiling Tools:

Two broad approaches:
- Instrumentation
- Sampling

Standard Advice:
- Focus on writing qualitatively good code first
- Once that’s working, use profiling tools to identify performance hot-spots and obtain quantitatively good code
Form Follows Function:

expr, term, atom :: Parser Int

expr = term "+" expr  -- return (l+r)
  | term "-" expr    -- return (l-r)
  | term

term = atom "*" term  -- return (l*r)
  | atom "/" term     -- return (l `div` r)
  | atom

atom = "-" atom  -- return (negate x)
  | "(" expr ")"    -- return n
  | number
Form Follows Function:

expr, term, atom :: Parser Int

expr = do l <- term; string "+"; r <- expr; return (l+r)
    ||| do l <- term; string "-"; r <- expr; return (l-r)
    ||| term

term = do l <- atom; string "*"; r <- term; return (l*r)
    ||| do l <- atom; string "/"; r <- term; return (l `div` r)
    ||| atom

atom = do string "-"; x <- atom; return (negate x)
    ||| do string "("; n <- expr; string ")"; return n
    ||| number
Parsing Examples:

Parsing> parse expr "1+2"
[3]
Parsing> parse expr "(1+2) * 3"
[]
Parsing> parse expr "(1+2)*3"
[9]
Parsing> parse expr "((1+2)*3)+1"
[10]
Parsing> parse expr "(((1+2)*3)+1)*8"
[80]
Parsing> parse expr "((((1+2)*3)+1)*8)"
[80]
Parsing>
Execution Statistics in Hugs:

Mechanisms:
- Enable the collection of execution statistics using `:set +s`
- Turn on messages when garbage collection occurs using `:set +g`
- Change total heap size (when loading Hugs) using `hugs --hSize`

Measures:
- **Cells**: a chunk of memory
- **Reductions**: a single rewrite step
Collecting Statistics:

Parsing> :set +s
Parsing> 1
1
(22 reductions, 32 cells)
Parsing> 2
2
(22 reductions, 32 cells)
Parsing> 3
3
(22 reductions, 32 cells)
Parsing> 1+2
3
(26 reductions, 36 cells)

Parsing> length "hello"
5
(56 reductions, 75 cells)
Parsing> length "world"
5
(56 reductions, 75 cells)
Parsing> id 1
1
(22 reductions, 32 cells)
Parsing> (\x -> x) 1
1
(23 reductions, 32 cells)
Parsing>
Observing Garbage Collection:

Parsing> :set
TOGGLES: groups begin with +/- to turn options on/off resp.
s Print no. reductions/cells after eval
...
OTHER OPTIONS: (leading + or - makes no difference)
hnum Set heap size (cannot be changed within Hugs)
...
Current settings: +squR -tgl.QwkIT -h1000000 -p"%s> " -r$$ -c40
...
Parsing> length [1..200000]
{{Gc:979946}}{{Gc:979945}}{{Gc:979947}}{{Gc:979946}}{{Gc:979947}}200000
(4200043 reductions, 5598039 cells, 5 garbage collections)
{{Gc:979983}}Parsing>
Observing Garbage Collection:

$ hugs -h100000 +gs
...
Hugs> length [1..200000]
{{Gc:86831}}{{Gc:86830}}{{Gc:86832}}{{Gc:86833}}{{Gc:86828}}...
{{Gc:86828}}{{Gc:86829}}{{Gc:86828}}{{Gc:86828}}{{Gc:86828}}200000
(4200054 reductions, 5598125 cells, 64 garbage collections)
{{Gc:86866}}Hugs> :q

$ hugs -h8M +gs
...
Hugs> length [1..200000]
200000
(4200054 reductions, 5598125 cells)
{{Gc:7986866}}Hugs> :q
Observing Garbage Collection:

$ hugs -h26378
...
ERROR "/Users/user/local/lib/hugs/packages/hugsbase/Hugs/Prelude.hs"
- Garbage collection fails to reclaim sufficient space
FATAL ERROR: Unable to load Prelude

$ hugs -h26379
...
Hugs> :set +sg
Hugs> length [1..200000]
{{Gc:13208}}{{Gc:13213}}{{Gc:13208}}{{Gc:13205}}{{Gc:13209}}...
{{Gc:13203}}{{Gc:13209}}200000
(4200054 reductions, 5598125 cells, 424 garbage collections)
{{Gc:13245}}Hugs>
Observations:

- Note that: $100000 - 86866 = 13134 = 26379 - 13245$
- So we can conclude that Hugs:
  - uses 13134 cells for internal state
  - needs at least 26379 cells to load

Possible profile of memory usage during startup:
Heap size, Residency, Allocation:

- **Heap size** measures maximum capacity
- **Residency** measures amount of memory that is actually in use at any given time
- Haskell programs allocate constantly (and, simultaneously, create garbage)
- **Total allocation** may exceed heap size
Back to Parsing:

Parentheses seem to be part of the problem, so let’s stress test:

```
addParens n s = if n==0
    then s
    else "(" ++ addParens (n-1) s ++ ")"
```

 Parsing> [ addParens n "1" | n <-[0..5] ]
["1","(1"","((1))","(((1)))","((((1))))","((((((1))))))")"
 Parsing>
Rapid increases in reductions and cell counts

 Parsing> :set +s
 Parsing> parse expr (addParens 1 "1")
 [1]
 (15060 reductions, 20628 cells)
 Parsing> parse expr (addParens 2 "1")
 [1]
 (137062 reductions, 187767 cells)
 Parsing> parse expr (addParens 3 "1")
 [1]
 (1234954 reductions, 1691736 cells, 1 garbage collection)
 Parsing> parse expr (addParens 4 "1")
 [1]
 (11115840 reductions, 15227127 cells, 15 garbage collections)
 Parsing> parse expr (addParens 5 "1")
 [1]
 (100043656 reductions, 137045268 cells, 139 garbage collections)
 Parsing>
Memory is not the problem here:

$ hugs -h26379 +sg
Hugs> :l altParsing.lhs
Parsing> :gc
Garbage collection recovered 6462 cells
Parsing> parse expr "1"
[1]
(1367 reductions, 1881 cells)
{{Gc:6304}} Parsing> parse expr (addParens 1 "1")
{{Gc:6218}} {{Gc:6213}} {{Gc:6217}}[1]
(15073 reductions, 20665 cells, 3 garbage collections)
{{Gc:6281}} Parsing> parse expr (addParens 5 "1")
{{Gc:6113}} {{Gc:6078}} {{Gc^C:6048}} {Interrupted!}

(16505831 reductions, 22610720 cells, 3713 garbage collections)
{{Gc:6048}} Parsing>

17
Analysis (1):

<table>
<thead>
<tr>
<th></th>
<th>parens</th>
<th>reductions</th>
<th>cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>15060</td>
<td>20628</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>137062</td>
<td>187767</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1234954</td>
<td>1691736</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>11115840</td>
<td>15227127</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>100043656</td>
<td>137045268</td>
</tr>
</tbody>
</table>
### Analysis (2):

<table>
<thead>
<tr>
<th></th>
<th>parens</th>
<th>reductions</th>
<th>cells</th>
<th>log reds</th>
<th>log cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15060</td>
<td>20628</td>
<td></td>
<td>4.177824972</td>
<td>4.314457123</td>
</tr>
<tr>
<td>2</td>
<td>137062</td>
<td>187767</td>
<td></td>
<td>5.136917065</td>
<td>5.273619267</td>
</tr>
<tr>
<td>3</td>
<td>1234954</td>
<td>1691736</td>
<td></td>
<td>6.091650781</td>
<td>6.228332591</td>
</tr>
<tr>
<td>4</td>
<td>11115840</td>
<td>15227127</td>
<td></td>
<td>7.045942287</td>
<td>7.18261797</td>
</tr>
<tr>
<td>5</td>
<td>100043656</td>
<td>137045268</td>
<td></td>
<td>8.000189554</td>
<td>8.136864044</td>
</tr>
</tbody>
</table>
Why Exponential Behavior?

```
expr, term, atom :: Parser Int

expr = do l <- term; string "+"; r <- expr; return (l+r)
      ||| do l <- term; string "-"; r <- expr; return (l-r)
      ||| term

term = do l <- atom; string "*"; r <- term; return (l*r)
      ||| do l <- atom; string "/"; r <- term; return (l `div` r)
      ||| atom

atom = do string "-"; x <- atom; return (negate x)
      ||| do string "("; n <- expr; string ")"; return n
      ||| number

Recall this grammar ...
```
Matching "1" as an term:

- First, we match it as a term ... and then find that it’s not followed by a "+
  \[
  \text{do } l \leftarrow \text{term}; \text{ string } "\text{+}"; r \leftarrow \text{expr}; \text{ return } (l+r)\]

- So then we match it again as a term ... and find that it’s not followed by a "-
  \[
  \text{do } l \leftarrow \text{term}; \text{ string } "\text{-}"; r \leftarrow \text{expr}; \text{ return } (l-r)\]

- Then, finally we can match it as a term without any following characters
  \[
  \text{term}\]

- So we will match "1" as a term three times before we succeed ... or as an atom nine times ... or ...
Refactoring the Grammar:

expr, term, atom :: Parser Int

expr = do l <- term
    do string "+"); r <- expr; return (l+r)
    ||| do string "; r <- expr; return (l-r)
    ||| return l

term = do l <- atom
    do string "*"); r <- term; return (l*r)
    ||| do string "; r <- term; return (l`div`r)
    ||| return l

atom = ... as before ...
A Step Forward:

Parsing> :set +s
Parsing> parse expr (addParens 10 "1")
[1]
(3624 reductions, 6091 cells)
Parsing> parse expr (addParens 100 "1")
[1]
(42414 reductions, 83491 cells)
Parsing> parse expr (addParens 1000 "1")
[1]
(1321314 reductions, 3530491 cells, 3 garbage collections)
Parsing> parse expr (addParens 10000 "1")

(3899701 reductions, 11445375 cells, 12 garbage collections)
ERROR - Control stack overflow
Parsing>
Proﬁling with GHC:

- GHC provides a much broader and more powerful range of proﬁling tools than Hugs

- We have to identify a main program:
  
  module Main where
  main = print (parse expr "((((((1)))))")

- Compiling: ghc --make altParsing.lhs

- Running: ./altParsing +RTS –sstderr

- Still slow!
$ ./altParsing +RTS -sstderr

848,494,732 bytes allocated in the heap
  1,506,284 bytes copied during GC (scavenged)
    0 bytes copied during GC (not scavenged)
  24,576 bytes maximum residency (1 sample(s))

  1619 collections in generation 0 ( 0.02s)
    1 collections in generation 1 ( 0.00s)

  1 Mb total memory in use

INIT time 0.00s ( 0.00s elapsed)
MUT time 1.01s ( 1.03s elapsed)
GC time 0.02s ( 0.02s elapsed)
EXIT time 0.00s ( 0.00s elapsed)
Total time 1.03s ( 1.06s elapsed)

%GC time 1.7% (2.3% elapsed)

Alloc rate 836,673,373 bytes per MUT second

Productivity 98.2% of total user, 96.0% of total elapsed
Profiling Options:

- For more serious work, compile with the –prof flag
  
  `ghc --make -prof altParsing.lhs`

- Opens up possibilities for:
  - Time and allocation profiling
  - Memory profiling
  - Coverage Profiling
  - ...

- Profiling code has overheads; not for production use
Cost Center Profiling:

- A technique for distributing costs during program execution

- Programmer creates “cost centers”:
  - by hand {-# SCC “name” #-}
  - for all top-level functions: -auto-all

- Program maintains runtime stack of cost centers

- RTS samples behavior at regular intervals

- Produce a summary report of statistics at the end of execution
In Practice:

$ ghc --make -prof -auto-all altParsing.lhs
$ ./altParsing +RTS -p
[1]
$ ls
altParsing* altParsing.hi altParsing.lhs
altParsing.o altParsing.prof
$
altParsing +RTS -p -RTS

total time = 0.54 secs (27 ticks @ 20 ms)
total alloc = 803,275,236 bytes (excludes profiling overheads)

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>%time</th>
<th>%alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAF</td>
<td>Main</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COST CENTRE</th>
<th>MODULE</th>
<th>no.</th>
<th>entries</th>
<th>%time</th>
<th>%alloc</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>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>CAF</td>
<td>Main</td>
<td>154</td>
<td>19</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>CAF</td>
<td>GHC.Handle</td>
<td>92</td>
<td>4</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Alas, not a very insightful report, in this case ...
Heap Profiling:

- A technique for measuring heap usage during program execution

- Compile code for profiling and run with argument +RTS option where option is:
  - -hc by function
  - -hm by module
  - -hy by type
  - -hb by thunk behavior

- Generates output.hp text file
- Produce a graphical version using hp2ps utility
In Practice:

```
$ ghc --make -prof altParing.lhs
$ ./altParing +RTS -hc
[1]

$ ls
altParing* altParing.hi altParing.lhs
altParing.o altParing.hp

$ hp2ps -c altParing.hp
$ open altParing.ps
$ 
```
Biographical Profiling (-hb):

- LAG phase: object created but not yet used
- USE: objects is in use
- DRAG: object has been used for the last time, but is still referenced
- VOID: an object is never used
Coverage Profiling:

- Used to determine which parts of a program have been exercised during any given run.

- Works by instrumenting code to get exact results.

- Provides two kinds of coverage:
  - Source coverage
    - Yellow – not executed
  - Boolean guard coverage
    - Green always true
    - Red always false
In Practice:

$ ghc --make -fhs pc altParsing.lhs
$ ./altParsing

[1]

$ ls
altParsing*  altParsing.hi  altParsing.lhs
altParsing.o  altParsing.tix
$
In Practice:

```
$hpc$ report altParsing

33% expressions used (138/409)
  0% boolean coverage (0/1)
    100% guards (0/0)
      0% 'if' conditions (0/1), 1 unevaluated
    100% qualifiers (0/0)
  66% alternatives used (4/6)
    0% local declarations used (0/6)
  54% top-level declarations used (18/33)
```

$
In Practice:

$ hpc markup altParsing
Writing: Main.hs.html
Writing: hpc_index.html
Writing: hpc_index_fun.html
Writing: hpc_index_alt.html
Writing: hpc_index_exp.html

$ open Main.hs.html
$ open hpc_index.html
$
Coverage of altParser:

```
> number :: Parser Int
> number = manyl digit
>       *** foldl1 (\a x -> 10*a+x)

A parser that evaluates arithmetic expressions:

> expr, term, atom :: Parser Int

> expr   = do l <- term; string "+"; r <- expr; return (l+r)
> | | do l <- term; string "-"; r <- expr; return (l-r)
> | | term

> term   = do l <- atom; string "*"; r <- term; return (l*r)
> | | do l <- atom; string "/"; r <- term; return (l `div` r)
> | | atom

> atom   = do string "-"; x <- atom; return (negate x)
> | | do string "("; n <- expr; string ")"; return n
> | | number
```
Summary:

- Profiling tools help us to understand the complex operational behavior of code
- Expert use of profiling tools requires significant use and experience
- But, even with limited experience, it is still possible to gain some interesting insight into what our programs really do!