CS 457/557: Functional Languages

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, …

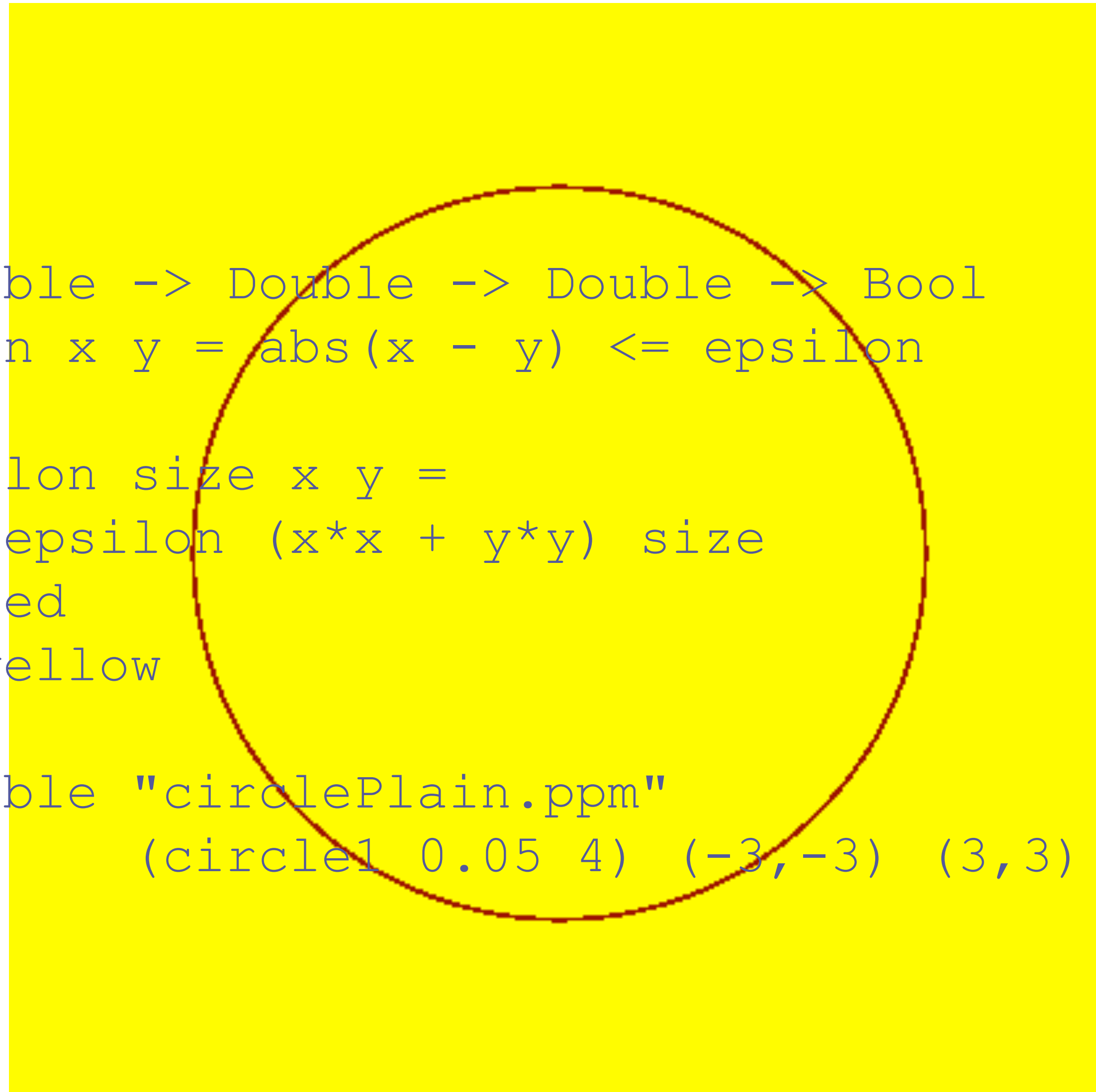
Profiling Tools:

- 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

Case Study: Profiling PPM

A Circle:



```
close :: Double -> Double -> Double -> Bool
close epsilon x y = abs(x - y) <= epsilon

circle1 epsilon size x y =
   if close epsilon (x*x + y*y) size
      then red
      else yellow

go1 = mapDouble "circlePlain.ppm"
                (circle1 0.05 4) (-3,-3) (3,3) (420,420)
```

Making Circles in Hugs:

```
Main> main
^C{Interrupted!}

Main> :set +s +g
Main> main
{{Gc:913203}}{{Gc:913215}}{{Gc:913203}}{{Gc:
   913206}}{{Gc:913219}}{{Gc:913209}}{{Gc:
   913206}}{{Gc:913207}}{{Gc:
   913207}}^C{Interrupted!}

(6164225 reductions, 8422432 cells, 9 garbage
   collections)
{{Gc:913207}}Main>
```

Making Circles with GHC:

```
prompt$ ./Main +RTS -sstderr
676,614,088 bytes allocated in the heap
    202,664 bytes copied during GC (scavenged)
    114,632 bytes copied during GC (not scavenged)
    548,864 bytes maximum residency (1 sample(s))
…
  MUT   time    1.19s  (  1.21s elapsed)
  GC    time    0.01s  (  0.01s elapsed)
  Total time    1.20s  (  1.22s elapsed)
…
  Productivity  99.2% of total user, 97.5% of total elapsed

prompt$
```

Bigger Circles with GHC:

```
prompt$ ./Main +RTS -sstderr
3,016,207,804 bytes allocated in the heap
      899,336 bytes copied during GC (scavenged)
      466,292 bytes copied during GC (not scavenged)
    3,153,920 bytes maximum residency (2 sample(s))
…
  MUT    time    5.26s  (  5.35s elapsed)
  GC     time    0.04s  (  0.05s elapsed)
  Total time     5.30s  (  5.40s elapsed)
…
  Productivity  99.3% of total user, 97.5% of total elapsed

prompt$
```

Increasing grid size to (1024,768)

Preparing to Profile:

```
prompt$ ghc --make Main -prof -auto-all -fforce-recomp -o Main
prompt$ ./Main +RTS -sstderr -p
4,688,711,540 bytes allocated in the heap
  2,201,188 bytes copied during GC (scavenged)
  1,235,956 bytes copied during GC (not scavenged)
  3,153,920 bytes maximum residency (2 sample(s))

  MUT    time     9.10s  (  9.23s elapsed)
  GC     time     0.06s  (  0.08s elapsed)
  Total  time     9.16s  (  9.31s elapsed)

prompt$
```

Profiling has overheads …

Inside Main.prof:

```
total time  =        1.06 secs   (53 ticks @ 20 ms)
total alloc = 2,705,945,792 bytes  (excludes profiling
overheads)
```

COST CENTRE	MODULE	%time	%alloc
quant8	**PPM6**	**47.2**	**62.1**
lift	PPM6	35.8	29.3
cmap	Colour	3.8	1.5
new	PPM6	3.8	0.1
go1	DemoPPM	3.8	0.0
clip	Colour	1.9	0.0
close	DemoPPM	1.9	1.4
circle1	DemoPPM	1.9	2.4
iterDouble	PPM6	0.0	3.1

... continued:

				individual		inherited	
COST CENTRE	MODULE	no.	entries	%time	%alloc	%time	%alloc
MAIN	MAIN	1	0	0.0	0.0	100.0	100.0
CAF	Main	222	2	0.0	0.0	0.0	0.0
main	Main	228	1	0.0	0.0	0.0	0.0
go1	DemoPPM	247	0	0.0	0.0	0.0	0.0
mapDouble	PPM6	248	0	0.0	0.0	0.0	0.0
CAF	DemoPPM	149	2	0.0	0.0	100.0	100.0
go1	DemoPPM	229	1	3.8	0.0	100.0	100.0
circle1	DemoPPM	239	786432	1.9	2.4	3.8	3.8
close	DemoPPM	240	786432	1.9	1.4	1.9	1.4
mapDouble	PPM6	230	1	0.0	0.0	92.5	96.2
lift	**PPM6**	**236**	**786432**	**35.8**	**29.3**	**88.7**	**92.9**
quant8	**PPM6**	**244**	**0**	**47.2**	**62.1**	**47.2**	**62.1**
cclip	Colour	237	786432	0.0	0.0	5.7	1.5
cmap	Colour	238	786432	3.8	1.5	5.7	1.5
clip	Colour	245	0	1.9	0.0	1.9	0.0
iterDouble	PPM6	235	1	0.0	3.1	0.0	3.1
new	PPM6	231	1	3.8	0.1	3.8	0.1

This slide has been edited to fit your screen ...

quant8 and lift:

```
quant8 :: RealFrac n => n -> Word8
quant8 x = floor $ x * 0xFF

lift :: IOUArray Int Word8 -> (Double -> Double -> Colour)
         -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case cclip (f x y) of
      (Colour r g b) -> do writeArray arr next     (quant8 r)
                           writeArray arr (next+1) (quant8 g)
                           writeArray arr (next+2) (quant8 b)
                           return (next +3)
```

We run quant8 3 times for every pixel on the grid!

Inside the Colour library:

```
module Colour where
data Colour = Colour {redPart, greenPart, bluePart :: Double}
    deriving (Eq, Show)

cmap :: (Double -> Double) -> Colour -> Colour
cmap f (Colour r g b) = Colour (f r) (f g) (f b)
…
clip   :: (Num n, Ord n) => n -> n
clip n  = max 0 (min 1 x)

cclip  :: Colour -> Colour
cclip   = cmap clip

black   = Colour 0 0 0
blue    = Colour 0 0 1
green   = Colour 0 0.5 0
…
```

Refactor the Colour library:

```
module Colour where
type Colour  = Color Double
data Color n = Color {redPart, greenPart, bluePart :: n }
    deriving (Eq, Show)

cmap :: (n -> n) -> Color n -> Color n
cmap f (Color r g b) = Color (f r) (f g) (f b)
…
clip   :: (Num n, Ord n) => n -> n
clip n  = max 0 (min 1 x)

cclip  :: (Num n, Ord n) => Color n -> Color n
cclip   = cmap clip

black, blue, green :: Colour
black   = Colour 0 0 0
blue    = Colour 0 0 1
green   = Colour 0 0.5 0
…
```

Update the definition of lift:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
          -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case cclip (f x y) of
      (Color r g b) -> do writeArray arr next     r
                          writeArray arr (next+1) g
                          writeArray arr (next+2) b
                          return (next +3)
```

Eliminates calls to quant8 …

Adjust definition of circle1:

```
circle1 epsilon size x y =
   if close epsilon (x*x + y*y) size
      then colquant red
      else colquant yellow

colquant             :: Color Double -> Color Word8
colquant (Color r g b) = Color (quant8 r) (quant8 g) (quant8 b)
```

… which get moved here instead

Time to Run!

```
prompt$ ./Main +RTS -sstderr -p
4,724,860,788 bytes allocated in the heap
  2,892,388 bytes copied during GC (scavenged)
  1,241,516 bytes copied during GC (not scavenged)
  3,153,920 bytes maximum residency (2 sample(s))

  MUT   time    9.13s  (  9.23s elapsed)
  GC    time    0.06s  (  0.09s elapsed)
  Total time    9.19s  (  9.32s elapsed)

prompt$
```

Disappointment: (slightly) worse than before ☹

… continued:

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

MAIN           MAIN          1          0    0.0    0.0   100.0  100.0
 CAF           Main        222          2    0.0    0.0     0.0    0.0
  main         Main        228          1    0.0    0.0     0.0    0.0
  go1          DemoPPM     250          0    0.0    0.0     0.0    0.0
   mapDouble   PPM6        251          0    0.0    0.0     0.0    0.0
 CAF           DemoPPM     149          1    0.0    0.0   100.0  100.0
  go1          DemoPPM     229          1    0.0    0.0   100.0  100.0
   circle1     DemoPPM     239     786432    3.2    2.2    51.6   66.9
    colquant   PPM6        241     786432    6.5    1.5    48.4   63.3
     quant8    PPM6        245          0   41.9   61.8    41.9   61.8
    close      DemoPPM     240     786432    0.0    1.4     0.0    1.4
   mapDouble   PPM6        230          1    0.0    0.0    48.4   33.1
    lift       PPM6        236     786432   38.7   28.1    38.7   29.9
     cclip     Colour      246          0    0.0    0.0     0.0    1.8
      cmap     Colour      247     786431    0.0    1.8     0.0    1.8
    iterDouble PPM6        235          1    3.2    3.1     3.2    3.1
    new        PPM6        231          1    6.5    0.1     6.5    0.1
```

Re-adjust definition of circle1:

```
circle1 epsilon size x y =
   if close epsilon (x*x + y*y) size
      then qred
      else qyellow

qred    = colquant red          -- CAFs
qyellow = colquant yellow       -- (Constant Applicative Forms)

colquant                :: Color Double -> Color Word8
colquant (Color r g b) = Color (quant8 r) (quant8 g) (quant8 b)
```

Make quantized red and yellow once only ...

Run Again …

```
prompt$ ./Main +RTS -sstderr -p
1,910,189,200 bytes allocated in the heap
      579,940 bytes copied during GC (scavenged)
      243,876 bytes copied during GC (not scavenged)
    3,153,920 bytes maximum residency (2 sample(s))

    MUT    time     3.20s  (   3.26s elapsed)
    GC     time     0.02s  (   0.03s elapsed)
    Total  time     3.22s  (   3.29s elapsed)

prompt$
```

That's more like it! ☺

Profiling Data:

COST CENTRE	MODULE	no.	entries	individual		inherited	
				%time	%alloc	%time	%alloc
MAIN	MAIN	1	0	0.0	0.0	100.0	100.0
CAF	Main	222	2	0.0	0.0	0.0	0.0
main	Main	228	1	0.0	0.0	0.0	0.0
go1	DemoPPM	254	0	0.0	0.0	0.0	0.0
mapDouble	PPM6	255	0	0.0	0.0	0.0	0.0
CAF	DemoPPM	149	3	0.0	0.0	100.0	100.0
go1	DemoPPM	229	1	0.0	0.0	100.0	100.0
circle1	DemoPPM	239	786432	0.0	6.0	0.0	9.8
close	DemoPPM	240	786432	0.0	3.8	0.0	3.8
mapDouble	PPM6	230	1	0.0	0.0	100.0	90.2
lift	**PPM6**	**236**	**786432**	**75.9**	**76.5**	**86.2**	**81.5**
cclip	Colour	247	0	0.0	0.0	10.3	5.0
clip	Colour	249	0	10.3	0.0	10.3	0.0
cmap	Colour	248	786431	0.0	5.0	0.0	5.0
iterDouble	PPM6	235	1	6.9	8.5	6.9	8.5
new	PPM6	231	1	6.9	0.2	6.9	0.2

Another look at lift:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
          -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case cclip (f x y) of
      (Color r g b) -> do writeArray arr next     r
                          writeArray arr (next+1) g
                          writeArray arr (next+2) b
                          return (next +3)
```

Yikes!

Anything stand out here?

Another look at lift:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
          -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case cclip (f x y) of
      (Color r g b) -> do writeArray arr next     r
                          writeArray arr (next+1) g
                          writeArray arr (next+2) b
                          return (next +3)
```

Yikes!

Anything stand out here?

Not much of a circle …

Oops, clipping broke the program!

Eliminate Clipping:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
          -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case (f x y) of
      (Color r g b) -> do writeArray arr next     r
                          writeArray arr (next+1) g
                          writeArray arr (next+2) b
                          return (next +3)
```

Now what happens to performance?

Another Run:

```
prompt$ ./Main +RTS -sstderr -p
1,805,257,208 bytes allocated in the heap
      467,876 bytes copied during GC (scavenged)
      232,576 bytes copied during GC (not scavenged)
    3,153,920 bytes maximum residency (2 sample(s))

    MUT    time     3.01s  (  3.20s elapsed)
    GC     time     0.02s  (  0.03s elapsed)
    Total time      3.03s  (  3.23s elapsed)

prompt$
```

improvements, but modest

Adding Cost Centers:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
          -> Double -> Double -> Int -> IO Int
lift arr f x y next =
    case {-# SCC "fxy" #-} (f x y) of
      (Color r g b) -> {-# SCC "act" #-}
                       do writeArray arr next      r
                          writeArray arr (next+1) g
                          writeArray arr (next+2) b
                          return (next +3)
```

Pragma

Pragma

More Profiling Data:

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

MAIN            MAIN       1            0     0.0     0.0    100.0   100.0
 CAF            Main     222            2     0.0     0.0      0.0     0.0
  main          Main     228            1     0.0     0.0      0.0     0.0
  go1           DemoPPM  253            0     0.0     0.0      0.0     0.0
   mapDouble    PPM6     254            0     0.0     0.0      0.0     0.0
 CAF            DemoPPM  149            3     0.0     0.0    100.0   100.0
  go1           DemoPPM  229            1     0.0     0.0    100.0   100.0
   circle1      DemoPPM  238       786432     0.0     6.4      9.8    10.5
    close       DemoPPM  239       786432     9.8     4.1      9.8     4.1
   mapDouble    PPM6     230            1     0.0     0.0     90.2    89.5
    lift        PPM6     236       786432     0.0     0.0     80.5    80.2
     act        PPM6     243       786432    80.5    80.2     80.5    80.2
     fxy        PPM6     237       786432     0.0     0.0      0.0     0.0
    iterDouble  PPM6     235            1     4.9     9.1      4.9     9.1
    new         PPM6     231            1     4.9     0.3      4.9     0.3
```

Adding More Cost Centers:

```
lift :: IOUArray Int Word8 -> (Double -> Double -> Color Word8)
          -> Double -> Double -> Int -> IO Int
lift arr f x y next =
 case {-# SCC "fxy" #-} (f x y) of
  (Color r g b) -> {-# SCC "act" #-}
                  do {-# SCC "act0" #-}writeArray arr next     r
                     {-# SCC "act1" #-}writeArray arr (next+1) g
                     {-# SCC "act2" #-}writeArray arr (next+2) b
                     return (next +3)
```

Even More Profiling Data:

				individual		inherited	
COST CENTRE	MODULE	no.	entries	%time	%alloc	%time	%alloc
MAIN	MAIN	1	0	0.0	0.0	100.0	100.0
CAF	Main	222	2	0.0	0.0	0.0	0.0
main	Main	228	1	0.0	0.0	0.0	0.0
go1	DemoPPM	256	0	0.0	0.0	0.0	0.0
mapDouble	PPM6	257	0	0.0	0.0	0.0	0.0
CAF	DemoPPM	149	3	0.0	0.0	100.0	100.0
go1	DemoPPM	229	1	0.0	0.0	100.0	100.0
circle1	DemoPPM	238	786432	0.0	6.4	10.0	10.5
close	DemoPPM	239	786432	10.0	4.1	10.0	4.1
mapDouble	PPM6	230	1	10.0	0.0	90.0	89.5
lift	PPM6	236	786432	0.0	0.0	60.0	80.2
act	**PPM6**	**243**	**786432**	**0.0**	**10.1**	**60.0**	**80.2**
act2	**PPM6**	**250**	**786432**	**10.0**	**23.9**	**10.0**	**23.9**
act1	**PPM6**	**247**	**786432**	**30.0**	**23.9**	**30.0**	**23.9**
act0	**PPM6**	**244**	**786432**	**20.0**	**22.2**	**20.0**	**22.2**
fxy	**PPM6**	**237**	**786432**	**0.0**	**0.0**	**0.0**	**0.0**
iterDouble	PPM6	235	1	0.0	9.1	0.0	9.1
new	PPM6	231	1	20.0	0.3	20.0	0.3

And on we (could) go …

Running without Profiling:

```
prompt$ ghc --make -fforce-recomp Main
prompt$ ./Main +RTS -sstderr
1,222,949,592 bytes allocated in the heap
    217,640 bytes copied during GC (scavenged)
     91,700 bytes copied during GC (not scavenged)
  3,153,920 bytes maximum residency (2 sample(s))

  MUT   time    1.96s  (  1.99s elapsed)
  GC    time    0.01s  (  0.02s elapsed)
  Total time    1.97s  (  2.01s elapsed)

prompt$
```

From 5.4 to 2.0 seconds …

Case Study: Profiling a Parser

Form Follows Function:

```
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
```

The Parser Monad:

```
data Parser a = P { applyP :: String -> [(a, String)] }

parse        :: Parser a -> String -> [a]
parse p s     = [ v | (v, ””) <- applyP p s ]

instance Monad Parser where
  return x  = P (\s -> [(x,s)])
  P p >>= f = P (\s -> [(y,s2) | (x,s1) <- applyP p s,
                                 (y,s2) <- applyP (f x) s1 ])

(|||)  :: Parser a -> Parser a -> Parser a
p ||| q = Parser (\s -> applyP p s ++ applyP q s)

string :: String -> Parser String
string ""     = return ""
string (c:cs) = …
```

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:

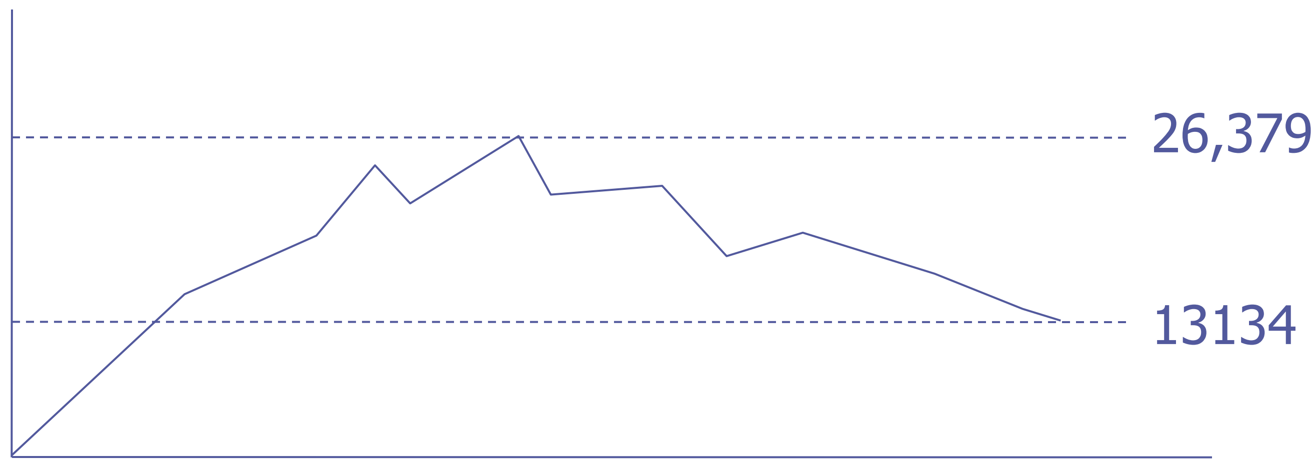
```
$ hugs -h100000 +gs
…
Hugs> length [1..200000]
{{Gc:86831}}{{Gc:86830}}{{Gc:86832}}{{Gc:86833}}{{Gc:86828}}…
{{Gc:86828}}{{Gc:86829}}{{Gc:86828}}{{Gc:86828}}200000
(4200054 reductions, 5598125 cells, 64 garbage collections)
{{Gc:86866}}Hugs> :q

$ 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>
```

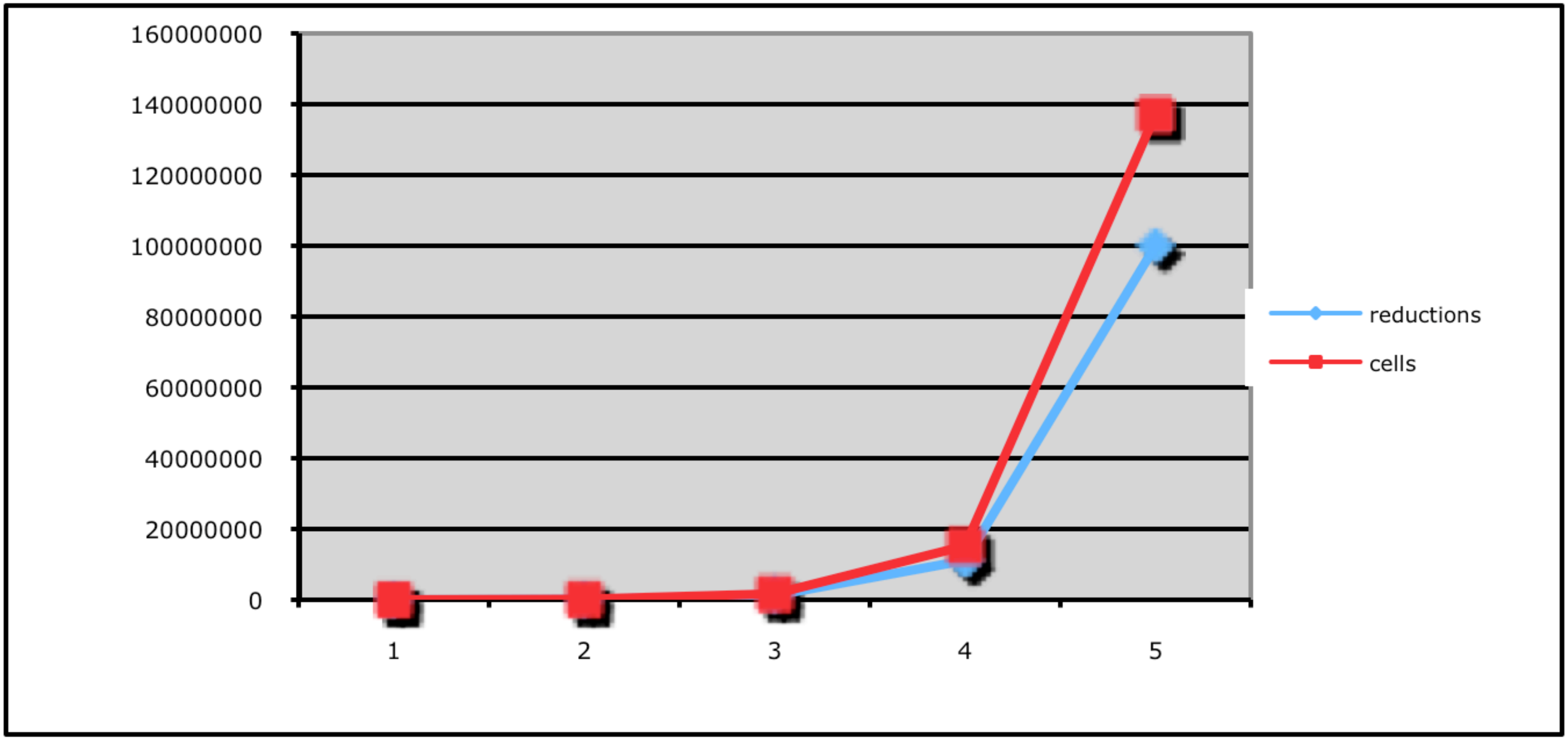
Rapid increases in reductions and cell counts

```
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:

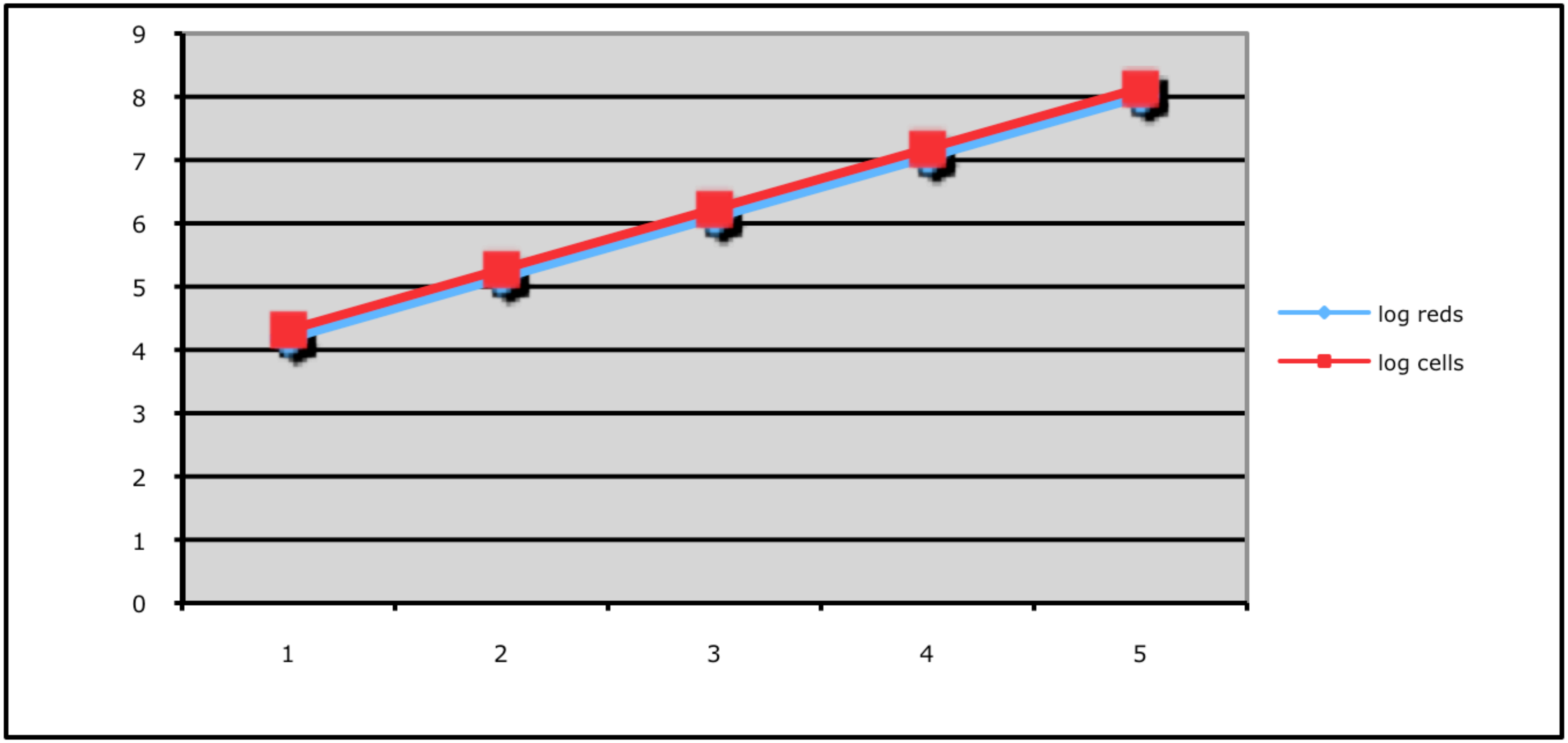
```
$ hugs -h26379 +sg
Hugs> :l altParsing.lhs
Parsing> :gc
Garbage collection recovered 6462 cells
Parsing> parse expr "1"
[1]
(1367 reductions, 1881 cells)
{{Gc:6304}}Parsing> parse expr (addParens 1 "1")
{{Gc:6218}}{{Gc:6213}}{{Gc:6217}}[1]
(15073 reductions, 20665 cells, 3 garbage collections)
{{Gc:6281}}Parsing> parse expr (addParens 5 "1")
{{Gc:6044}}{{Gc:6072}}{{Gc:6066}}{{Gc:6076}}{{Gc:6072}}{{Gc:
6081}}{{Gc:6063}}{{Gc:6085}}{{Gc:6068}}{{Gc:6090}}{{Gc:6062}}...
{{Gc:6113}}{{Gc:6078}}{{Gc^C:6048}}{Interrupted!}

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

Analysis (1):



parens	reductions	cells
1	15060	20628
2	137062	187767
3	1234954	1691736
4	11115840	15227127
5	100043656	137045268

Analysis (2):



parens	reductions	cells	log reds	log cells
1	15060	20628	4.177824972	4.314457123
2	137062	187767	5.136917065	5.273619267
3	1234954	1691736	6.091650781	6.228332591
4	11115840	15227127	7.045942287	7.18261797
5	100043656	137045268	8.000189554	8.136864044

Why Exponential Behavior?

Recall this grammar …

```
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
```

Matching "1" as an term:

- First, we match it as a term … and then find that it's not followed by a "+"

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

- So then we match it again as a term … and find that it's not followed by a "-"

  do **l <- term**; string "-"; r <- expr; return (l-r)

- Then, finally we can match it as a term without any following characters

  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
[1]
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
$
```

```
 Time and Allocation Profiling Report  (Final)

           altParsing +RTS -p -RTS

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

COST CENTRE                  MODULE        %time %alloc

CAF                          Main          100.0  100.0


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

MAIN         MAIN                    1           0    0.0    0.0   100.0  100.0
 CAF         Main                  154          19 100.0  100.0   100.0  100.0
 CAF         GHC.Handle             92           4    0.0    0.0     0.0    0.0
```

Alas, not a very insightful report,
in this case ...

Heap Profiling:

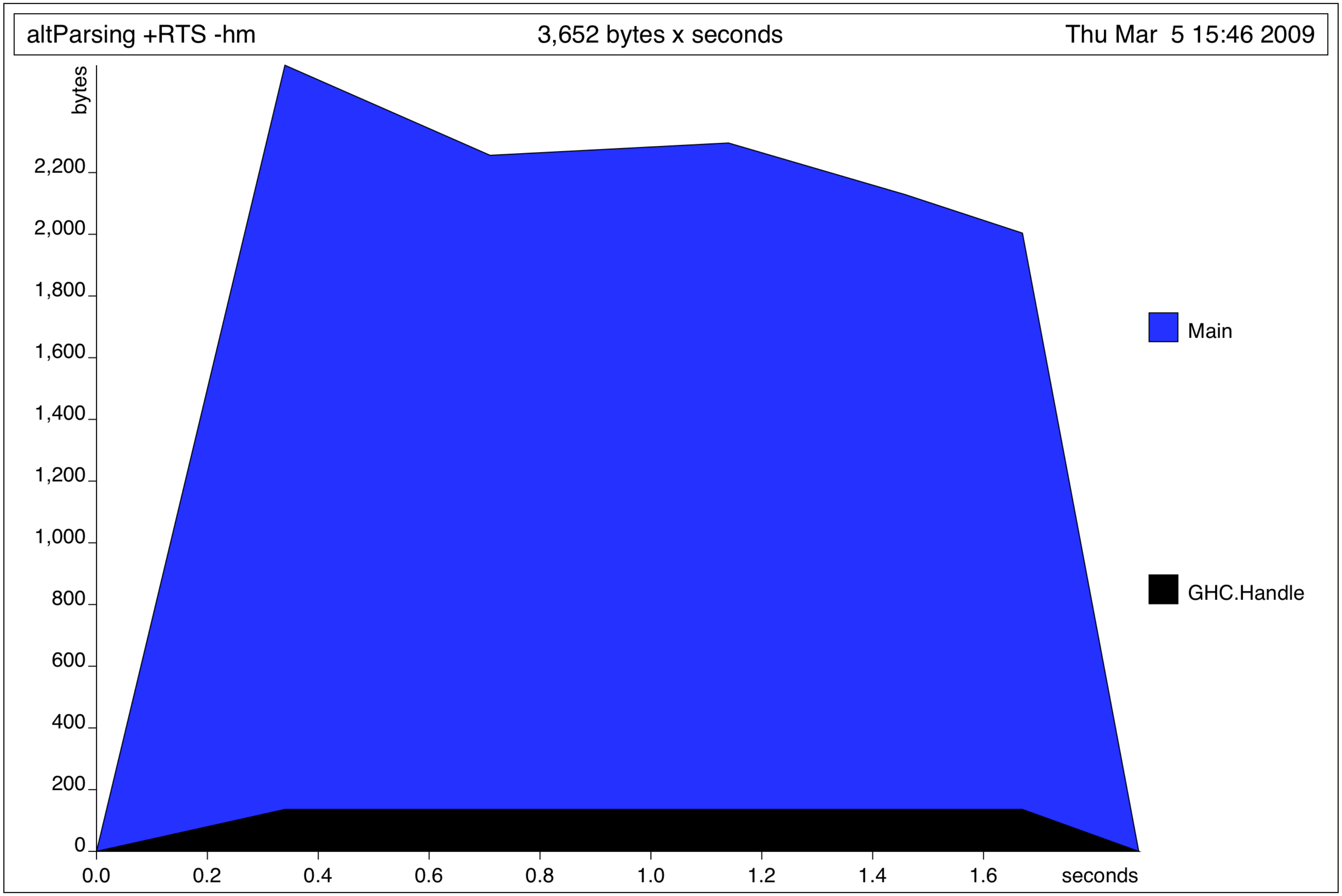
- A technique for measuring heap usage during program execution

- 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

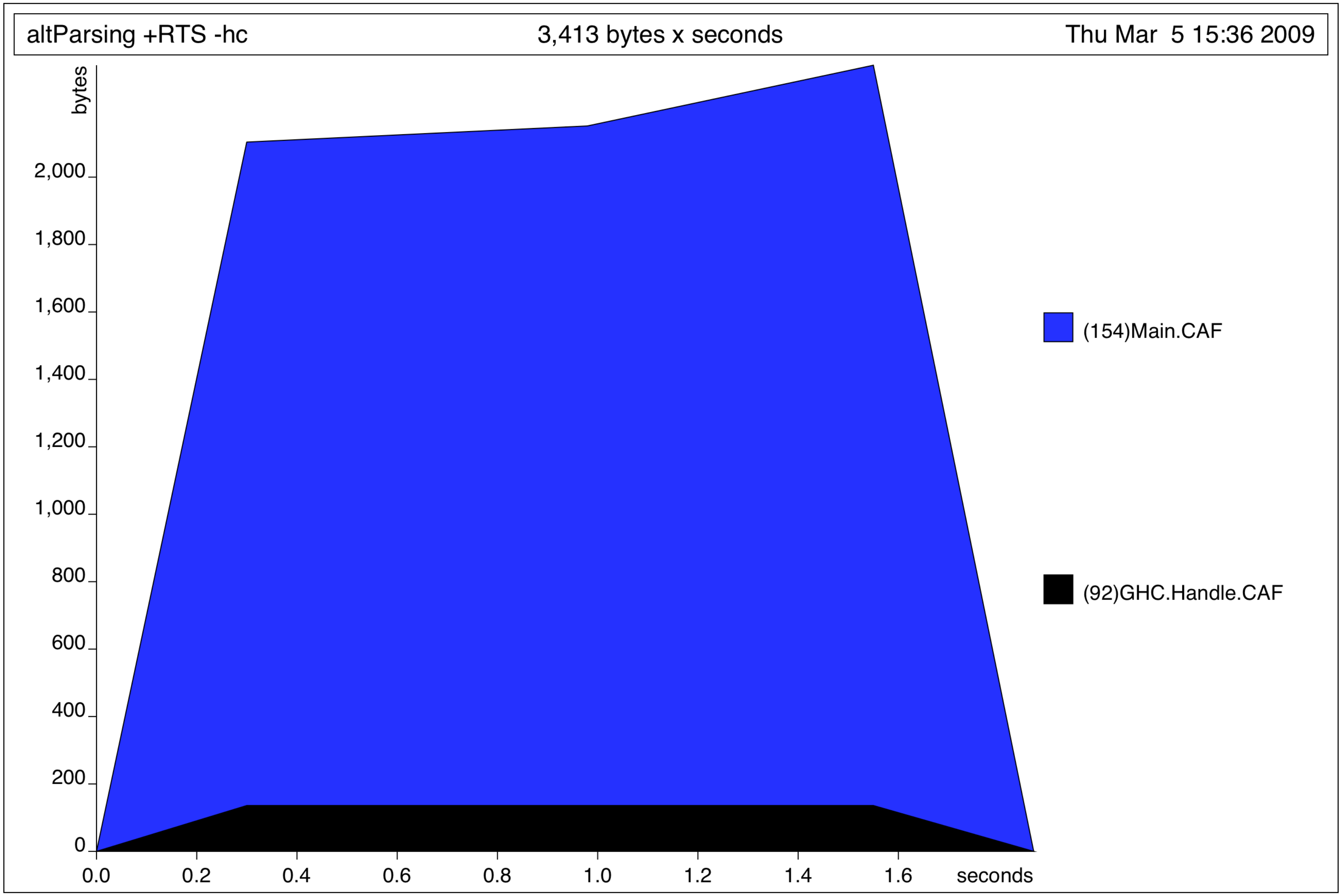
- Generates output.hp text file
- Produce a graphical version using hp2ps utility

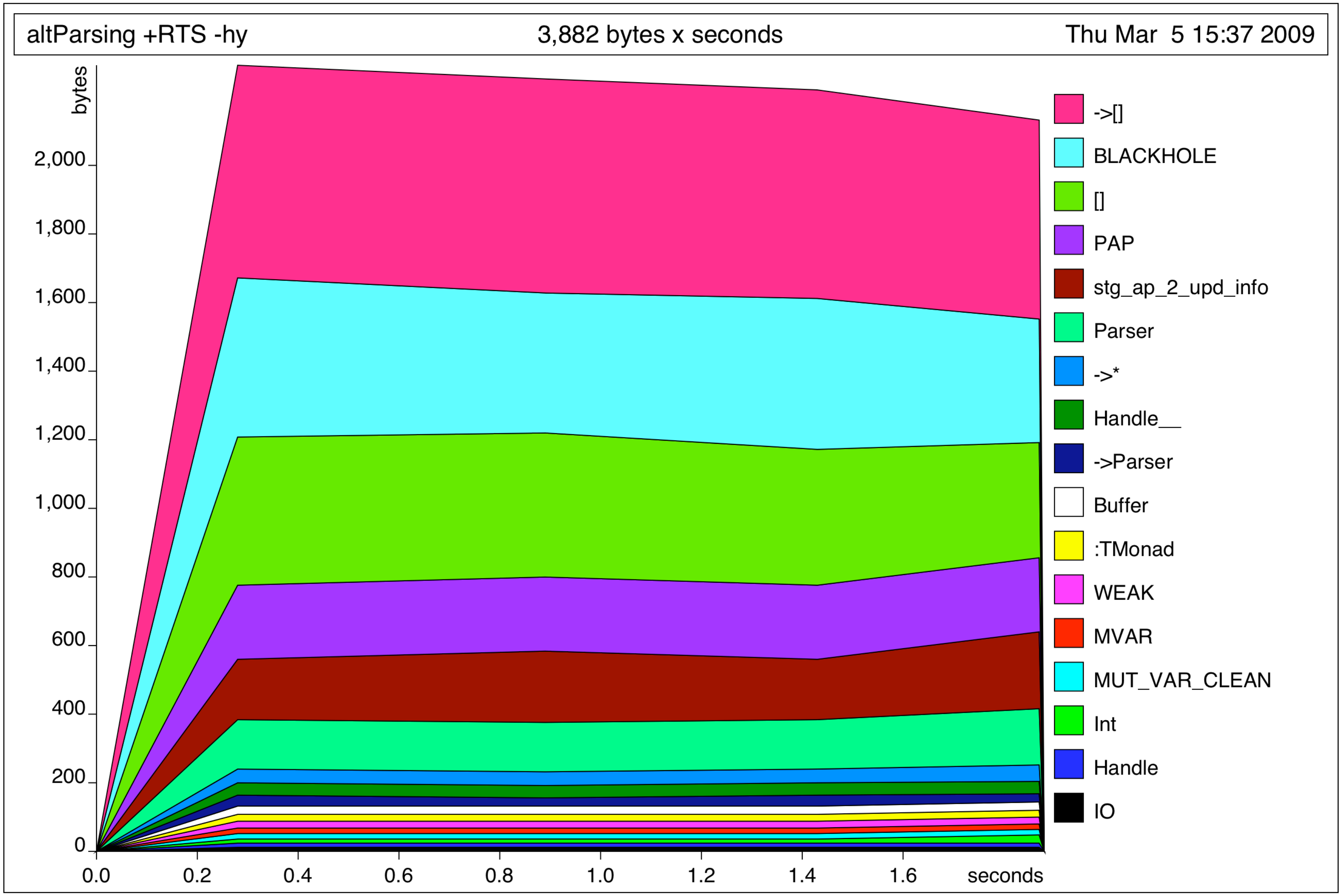
In Practice:

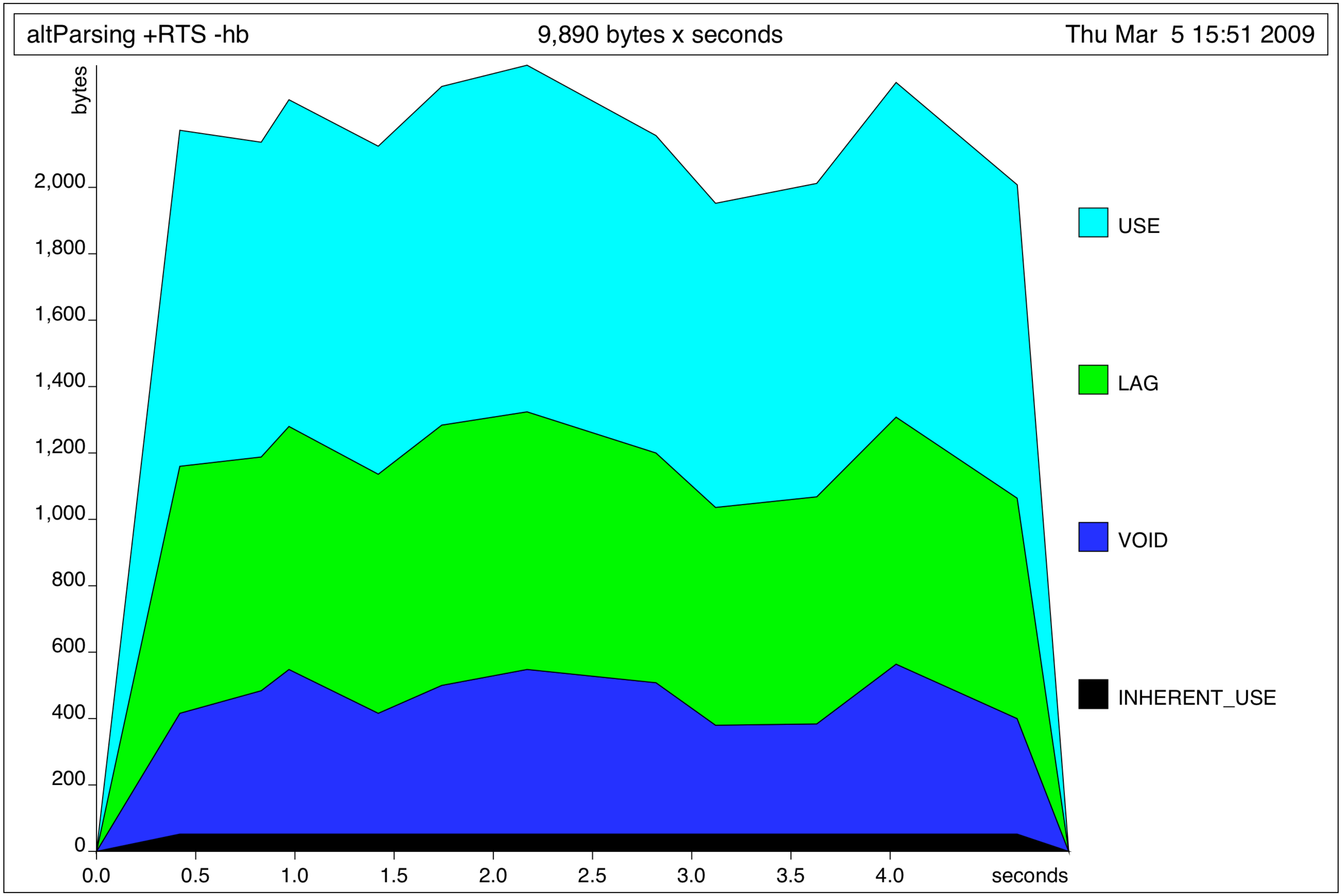
```
$ ghc --make –prof altParsing.lhs
$ ./altParsing +RTS -hc
[1]
$ ls
altParsing*      altParsing.hi    altParsing.lhs
altParsing.o     altParsing.hp
$ hp2ps –c altParsing.hp
$ open altParsing.ps
$
```



altParsing +RTS -hm
3,652 bytes x seconds
Thu Mar 5 15:46 2009
bytes
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
seconds
Main
GHC.Handle


altParsing +RTS -hc
3,413 bytes x seconds
Thu Mar 5 15:36 2009
bytes
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
seconds
(154)Main.CAF
(92)GHC.Handle.CAF


altParsing +RTS -hy
3,882 bytes x seconds
Thu Mar 5 15:37 2009
bytes
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
seconds
->[]
BLACKHOLE
[]
PAP
stg_ap_2_upd_info
Parser
->*
Handle__
->Parser
Buffer
:TMonad
WEAK
MVAR
MUT_VAR_CLEAN
Int
Handle
IO



altParsing +RTS -hb
9,890 bytes x seconds
Thu Mar 5 15:51 2009
bytes
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
seconds
USE
LAG
VOID
INHERENT_USE

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 –fhpc 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  = many1 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 into what our programs really do!