Programación Funcional Avanzada

Evaluación de desempeño

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Análisis de Desempeño

Evaluar, mejorar y aprovechar

- Casi siempre se puede escribir un programa Haskell con desempeño igual a un programa escrito en *whatever*.

Análisis de Desempeño

Evaluar, mejorar y aprovechar

- Casi siempre se puede escribir un programa Haskell con desempeño igual a un programa escrito en *whatever*.
- Evaluar el desempeño – *profiling*.
  - Proceso iterativo.
  - Instrumentar, correr, analizar, cambiar – enjuague y repita.

UNIVERSIDAD SIMÓN BOLÍVAR

Análisis de Desempeño

Evaluar, mejorar y aprovechar

- Casi siempre se puede escribir un programa Haskell con desempeño igual a un programa escrito en *whatever*.
- Evaluar el desempeño – *profiling*.
  - Proceso iterativo.
  - Instrumentar, correr, analizar, cambiar – enjuague y repita.
- Mejorar los algoritmos – investigación.
  - Estructuras de datos funcionales especializadas – Okasaki
  - Convertir algoritmo en flujo o transformación de datos.

UNIVERSIDAD SIMÓN BOLÍVAR

Análisis de Desempeño

Evaluar, mejorar y aprovechar

- Casi siempre se puede escribir un programa Haskell con desempeño igual a un programa escrito en *whatever*.
- Evaluar el desempeño – *profiling*.
  - Proceso iterativo.
  - Instrumentar, correr, analizar, cambiar – enjuague y repita.
- Mejorar los algoritmos – investigación.
  - Estructuras de datos funcionales especializadas – Okasaki
  - Convertir algoritmo en flujo o transformación de datos.
- Tipos de datos optimizados.
  - Conozca las librerías y estudie sus API – algo de eso el jueves.
  - Interactúe con una librería externa vía FFI.

La optimización prematura es la raíz de todos los males
– Donald Knuth

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Un ejemplo simple

```
import System.Environment
import Control.Monad

main = do
  [d] <- map read `liftM` getArgs
  print $ promedio [1..d]

promedio :: [Double] -> Double
promedio xs = sum xs / (fromIntegral $ length xs)
```

- Implantación directa del cálculo del promedio.
- El típico código "algebráico" simpático de explicar.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

- Compilamos de manera regular

```
$ ghc --make ave0.hs
```

- Probamos con diferentes tamaños.

```
$ time ./ave0 1e5
50000.5
real 0m0.023s
$ time ./ave0 1e6
500000.5
real 0m0.113s
$ time ./ave0 1e7
5000000.5
real 0m1.296s
```

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

- Compilamos de manera regular

```
$ ghc --make ave0.hs
```

- Probamos con diferentes tamaños.

```
$ time ./ave0 1e5
50000.5
real 0m0.023s
$ time ./ave0 1e6
500000.5
real 0m0.113s
$ time ./ave0 1e7
5000000.5
real 0m1.296s
```

- . . . la corrida con 1e8 la interrumpí después de 30 segundos.

Y U NO FAST?

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Primer paso – Ejecutar con estadísticas

```
$ ghc --make -rtsopts ave0.hs
$ ave0 +RTS -sstderr -RTS 1e7
5000000.5
   1,680,120,656 bytes allocated in the heap
   1,160,997,168 bytes copied during GC
     382,503,904 bytes maximum residency (10 sample(s))
      68,437,544 bytes maximum slop
             868 MB total memory in use
                 (0 MB lost due to fragmentation)
...
```

- 868Mb efectivamente empleados durante la ejecución.
- 1160Mb copiados durante recolección de basura – objetos que debían persistir y cambiaron de generación.

Profiling

Primer paso – Ejecutar con estadísticas

```
...
INIT    time    0.00s  (  0.00s elapsed)
MUT     time    0.43s  (  0.43s elapsed)
GC      time    0.84s  (  0.84s elapsed)
EXIT    time    0.00s  (  0.00s elapsed)
Total   time    1.26s  (  1.27s elapsed)
%GC     time   66.2%  (66.3% elapsed)
Alloc rate    3,935,011,902 bytes per MUT second
Productivity  33.8% of total user,
              33.7% of total elapsed
```

- `MUT` (*Main User Thread*) vs. `GC` (*Garbage Collector*).
- 66 % del tiempo recogiendo basura.
- Casi 4Gb de RAM por **segundo** de uso – WTF!

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

¿Dónde se está consumiendo el tiempo?

- Compilar el programa para *time profiling*.
- Identificar **Centros de Costo** – funciones o sub-expresiones.
  - Automáticamente con `-auto-all`
  - Manualmente con `{-# SCC foo #-}`
- CAF – *Constant Applicative Forms*.
  - Cualquier valor sin argumentos es un *CAF*.
  - Calculados **una** sola vez – se comparten en el resto.
  - No "corren" – `-caf-all` mide el costo de su cálculo único.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

¿Dónde se está consumiendo el tiempo?

- Compilar el programa para *time profiling*.
- Identificar **Centros de Costo** – funciones o sub-expresiones.
  - Automáticamente con `-auto-all`
  - Manualmente con `{-# SCC foo #-}`
- CAF – *Constant Applicative Forms*.
  - Cualquier valor sin argumentos es un *CAF*.
  - Calculados **una** sola vez – se comparten en el resto.
  - No "corren" – `-caf-all` mide el costo de su cálculo único.

```
$ ghc -O2 -prof -auto-all -caf-all -rtsopts \
        -fforce-recomp --make ave0.hs
```

Hace falta instalar las versiones `-prof` de las librerías.

Profiling

¿Dónde se está consumiendo el tiempo?

- Ejecutar indicando al RTS que acumule contadores de *profiling*.

```
$ time ave0 +RTS -p -RTS 1e7
Stack space overflow: current size 8388608 bytes.
Use ‘+RTS -Ksize -RTS’ to increase it.
```

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

¿Dónde se está consumiendo el tiempo?

- Ejecutar indicando al RTS que acumule contadores de *profiling*.

```
$ time ave0 +RTS -p -RTS 1e7
Stack space overflow: current size 8388608 bytes.
Use ‘+RTS -Ksize -RTS’ to increase it.
```

- La instrumentación tiene un costo adicional en pila.

```
$ time ave0 +RTS -p -K1000M -RTS 1e7
5000000.5
real 0m4.987s
```

- La ejecución toma bastante más tiempo – resultados en `ave0.prof`.
  - *Profiling* es una evaluación a tiempo de desarrollo y pruebas.
  - Aproveche sus pruebas automatizadas.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

¿Dónde se está consumiendo el tiempo?

- Ejecutar indicando al RTS que acumule contadores de *profiling*.

```
$ time ave0 +RTS -p -RTS 1e7
Stack space overflow: current size 8388608 bytes.
Use ‘+RTS -Ksize -RTS’ to increase it.
```

- La instrumentación tiene un costo adicional en pila.

```
$ time ave0 +RTS -p -K1000M -RTS 1e7
5000000.5
real 0m4.987s
```

- La ejecución toma bastante más tiempo – resultados en `ave0.prof`.
  - *Profiling* es una evaluación a tiempo de desarrollo y pruebas.
  - Aproveche sus pruebas automatizadas.

Evite hacerlo en producción – se nota el impacto.

Profiling – Resumen General de Ejecución

¿Dónde se está consumiendo el tiempo?

```
total time  =         1.14 secs
                (1138 ticks @ 1000 us, 1 processor)
total alloc = 2,240,118,848 bytes
                (excludes profiling overheads)
```

- Tiempo real excluyendo costo de *profiling*.
- Cantidad total de memoria reservada durante la ejecución.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling – Resumen por Centro de Costo

¿Dónde se está consumiendo el tiempo?

```
COST CENTRE    MODULE     %time %alloc
main           Main        60.7   75.0
promedio       Main        39.3   25.0
```

- Seguido de un resumen detallado por centro de costo – demasiado grande para que quepa en una lámina.
- Muchos CAF son propios de GHC – generalmente no influyen.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

¿Dónde se está consumiendo el tiempo?

- La mayoría del tiempo se está invirtiendo en dos CAF
  - El que calcula la suma – `CAF:main_sum`.
  - Los relacionados con números punto flotante.
- ¿Qué podemos concluir?
  - Reserva de memoria para los números en punto flotante.
  - Liberación de memoria para los números en punto flotante.
  - La combinación nos está matando.

Estudiemos la forma en que se reserva memoria.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

¿Cómo se está utilizando la memoria?

- RTS puede generar *gráficos* mostrando uso del *heap*.
- Se compila igual – se ejecuta con opciones diferentes
  - Consumo por centro de costo.
  - Consumo por módulo.
  - Consumo por constructor.
  - Consumo por tipo de datos.
- Ayudan a detectar *space leaks*
  - Iniciar con el gráfico "estándar" por centro de costo.
  - Refinar con los otros tres.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Consumo de memoria por centro de costo

- Se ejecuta usando `-hc`

```
$ ave0  +RTS -p -K1000M -hc -RTS 1e7
```

  - Tarda *aún más* que la corrida anterior.
  - Resultados depositados en `ave0.hp`

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Consumo de memoria por centro de costo

- Se ejecuta usando `-hc`

```
$ ave0  +RTS -p -K1000M -hc -RTS 1e7
```

  - Tarda *aún más* que la corrida anterior.
  - Resultados depositados en `ave0.hp`

- Convertimos a gráfico

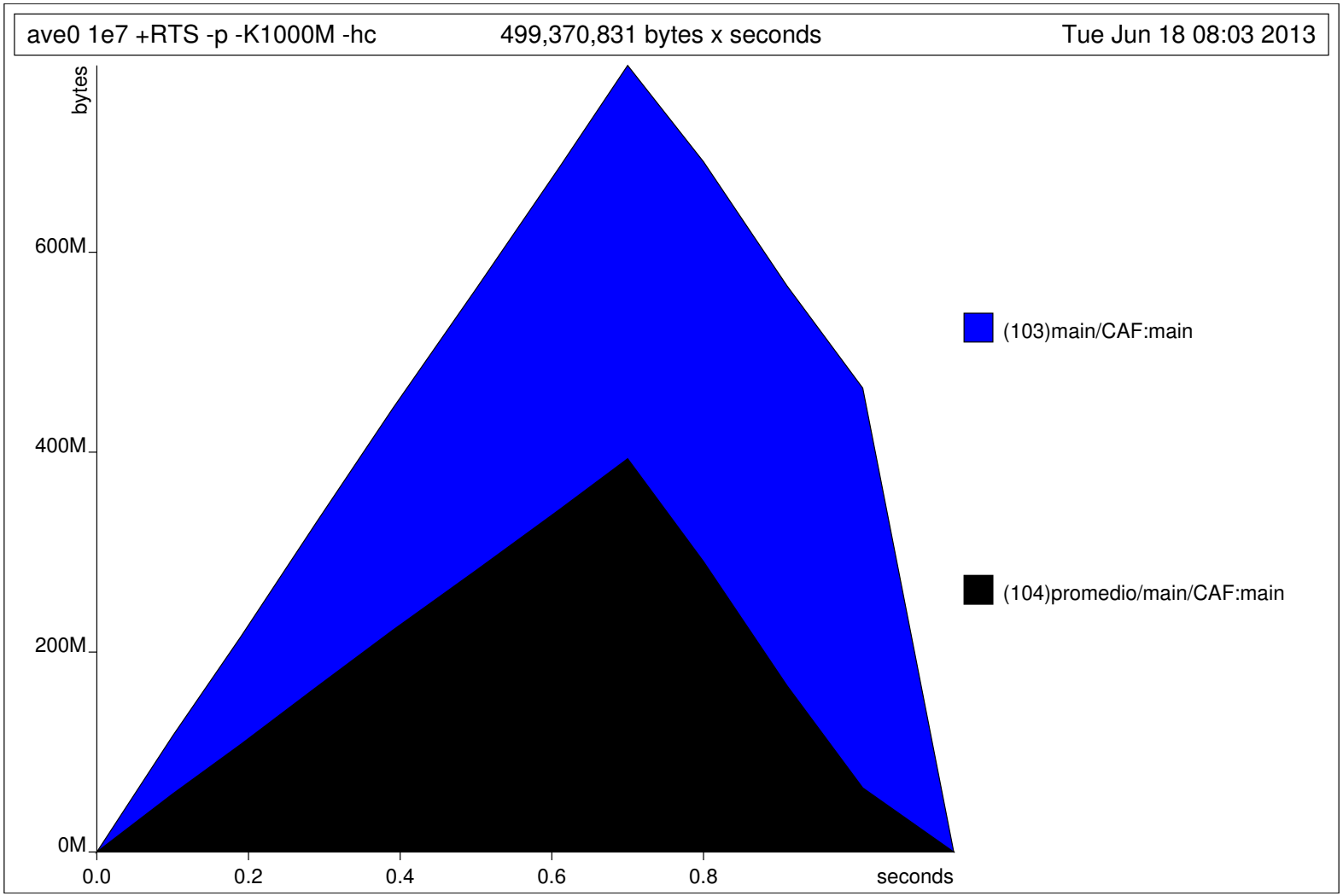
```
$ hp2ps -e8in -c ave0.hp
$ epstopdf ave0.ps
```

  - Opciones por omisión – PS blanco y negro, papel A4 apaisado.
  - Opciones "LaTeX friendly" – EPS a colores al tamaño especificado.
  - Para incluirlas hay que usar `epstopdf`.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Consumo de memoria por centro de costo



UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Consumo de memoria por constructor

- Se ejecuta usando `-hd` – resultados en `prof.hp`

```
$ ave0  +RTS -p -K1000M -hd -RTS 1e7
```

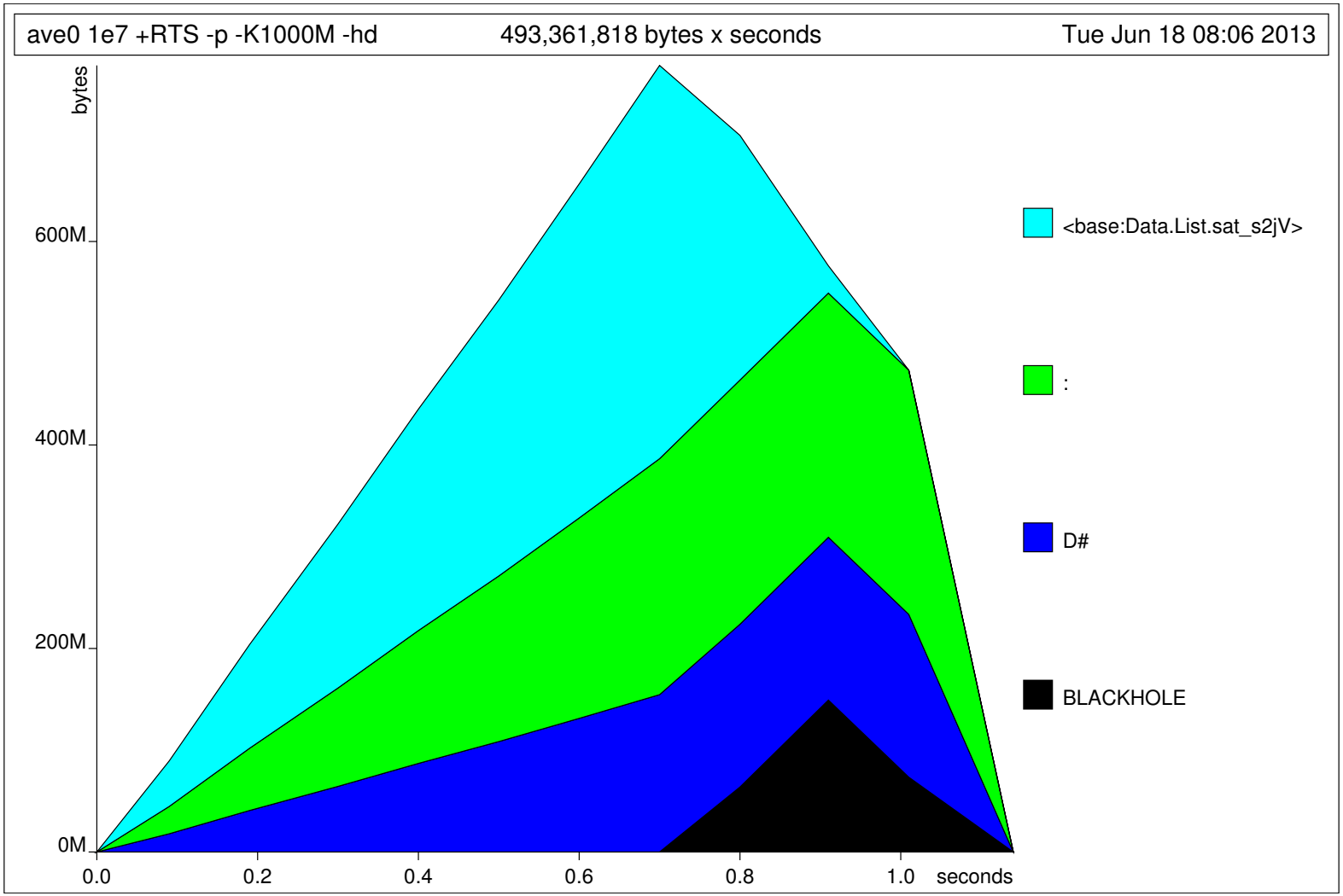
- Convertimos a gráfico

```
$ hp2ps -e8in -c ave0.hp
$ epstopdf ave0.ps
```

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Consumo de memoria por constructor



Profiling

¿Qué pasa con nuestro programa?

- Para calcular la suma hay que construir la lista.
- Para construir la lista hay que reservar `Doubles`.
- No se pueden liberar – `length` aún debe calcular la longitud.
- La flojera del lenguaje nos está afectando.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

¿Qué pasa con nuestro programa?

- Para calcular la suma hay que construir la lista.
- Para construir la lista hay que reservar `Doubles`.
- No se pueden liberar – `length` aún debe calcular la longitud.
- La flojera del lenguaje nos está afectando.
- Reescribimos usando `foldl'` que es más estricto.

```
promedio xs = s / fromIntegral n
  where
    (n,s)      = foldl' go (0,0) xs
    go (n,s) x = (n+1,s+x)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling – toma dos

- Compilamos con optimización.

```
$ ghc --make -rtsopts ave1.hs
```

- Probamos con diferentes tamaños.

```
$ time ./ave1 1e5
50000.5
real 0m0.047s
$ time ./ave1 1e6
Stack space overflow: current size 8388608 bytes.
...
$ time ./ave1 +RTS -K200M -RTS 1e6
500000.5
real 0m0.410s
$ time ./ave1 +RTS -K200M -RTS 1e7
5000000.5
real    0m4.381s
```

UNIVERSIDAD SIMÓN BOLÍVAR

Wait. . . what?

¿No y que `foldl'` era la cura de todos los males?

- Correr con estadísticas produce resultados desalentadores

```
3,224,754,000 bytes allocated in the heap
3,172,520,784 bytes copied during GC
  887,830,864 bytes maximum residency (11 sample(s))
   11,411,032 bytes maximum slop
         1572 MB total memory in use
           (0 MB lost due to fragmentation)
%GC time       79.2%  (79.2% elapsed)
Productivity   20.8% of total user,
               28.8% of total elapsed
```

- Consume más memoria que al principio.
- Es más ineficiente que al principio – ¡se la pasa recogiendo basura!
- ¡Correr con *profiling* sobre 1e7 requiere 6Gb RAM!

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Eliminando flojera

- `foldl'` es estricto en la *función* – ¡pero la función produce tuplas!
- "Estricto" quiere decir *Weak Head Normal Form*
  - WHNF para una tupla es `(,)`
  - ¡Tuplas con *thunks* de sumas de `Double`!

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Eliminando flojera

- `foldl'` es estricto en la *función* – ¡pero la función produce tuplas!
- "Estricto" quiere decir *Weak Head Normal Form*
  - WHNF para una tupla es `(,)`
  - ¡Tuplas con *thunks* de sumas de `Double`!
- Usaremos *bang patterns* para forzar los *thunks* de las tuplas

```
{-# LANGUAGE BangPatterns #-}

promedio xs = s / fromIntegral n
  where
    (n,s)         = foldl' k (0,0) xs
    k (!n,!s) x = (n+1,s+x)
```

Un buen resultado

- Ahora, sin miedo

```
time ./ave2 +RTS -sstderr -RTS 1e8
5.00000005e7
real 0m5.221s
```

- Y las estadísticas son excelentes

```
28,624 bytes maximum residency (1 sample(s))
22,992 bytes maximum slop
     1 MB total memory in use
    (0 MB lost due to fragmentation)
%GC time       2.4%  (2.3% elapsed)
Productivity  97.6% of total user,
              97.4% of total elapsed
```

Ciclo cerrado con espacio constante de pila.

UNIVERSIDAD SIMÓN BOLÍVAR

Profiling

Para comparar

- Resumen de ejecución general.

```
total time  =         12.10 secs
                (12103 ticks @ 1000 us, 1 processor)
total alloc = 25,600,119,080 bytes
                (excludes profiling overheads)
```

- Resumen de ejecución por centro de costo

```
COST CENTRE     MODULE     %time %alloc
main            Main        53.1   65.6
promedio.go     Main        23.6   12.5
promedio.(...)  Main        23.3   21.9
```

  - %time sigue siendo proporcional – "hay que hacer las cuentas"
  - %alloc sigue siendo proporcional – "hay que manipular los datos"
  - Manipulación estricta – no hay acumulación de memoria.

Profiling y concurrencia

- Combinar `-threaded` y `-prof` – ¡*profiling* de programas concurrentes!
- *Profiler* reducirá la escalabilidad.
  - Estado compartido para las estadísticas tiene un bloqueo global.
  - Estadísticas del manejador de memoria **no** usan bloqueos
    los resultados podrían ser inexactos dependiendo del programa.
  - Cuento corto – usar `-fno-prof-count-entries`.
- La preocupación suele ser cuánto trabajan y no tanto en qué trabajan.

Necesitamos una herramienta complementaria diferente.

UNIVERSIDAD SIMÓN BOLÍVAR

Análisis de Programas Concurrentes

¿Qué hacen los hilos?

- **Threadscope** es una herramienta especializada para el análisis de programas concurrentes y paralelos en Haskell.
  - ¿La carga de trabajo está balanceada entre los hilos?
  - ¿El recolector de basura está "molestando"?
  - ¿Tengo la cantidad adecuada de *sparks*?
- Ejecutable instrumentado genera *bitácora de eventos concurrentes* – `threadscope` muestra el análisis gráficamente.

UNIVERSIDAD SIMÓN BOLÍVAR

Instrumentando un ejecutable

Sudoku Solver "inocente"

- Compilar con instrumentación de código

```
$ ghc -threaded -eventlog --make sudoku1.hs
```

- Ejecutar el programa en múltiples *cores* y registrando eventos – la bitácora de eventos será `sudoku1.eventlog`

```
$ sudoku1 +RTS -N -ls -RTS 1000.txt
```

- Analizar la bitácora

```
$ threadscope sudoku1.eventlog
```

UNIVERSIDAD SIMÓN BOLÍVAR

Aaaaaaawwwwww!

This is a screenshot

UNIVERSIDAD SIMÓN BOLÍVAR

WYSIWYG

- Resumen visual de los eventos en cada hilo
  - Ejecución.
  - Recolector de basura – paralelo o secuencial.
  - Creación, migración, *wakeup* y destrucción de hilos.
  - Ciclo de vida de *sparks* – requiere GHC 7.3

WYSIWYG

- Resumen visual de los eventos en cada hilo
  - Ejecución.
  - Recolector de basura – paralelo o secuencial.
  - Creación, migración, *wakeup* y destrucción de hilos.
  - Ciclo de vida de *sparks* – requiere GHC 7.3
- Presentación visual y textual conjunta
  - *Timeline* muestra los eventos visualmente.
  - *Events* muestra la descripción textual de los eventos.
  - Seleccionar en uno, desplaza el otro apropiadamante.
- Zoom in/out – detalle de eventos.

UNIVERSIDAD SIMÓN BOLÍVAR

WYSIWYG

- Resumen visual de los eventos en cada hilo
  - Ejecución.
  - Recolector de basura – paralelo o secuencial.
  - Creación, migración, *wakeup* y destrucción de hilos.
  - Ciclo de vida de *sparks* – requiere GHC 7.3
- Presentación visual y textual conjunta
  - *Timeline* muestra los eventos visualmente.
  - *Events* muestra la descripción textual de los eventos.
  - Seleccionar en uno, desplaza el otro apropiadamante.
- Zoom in/out – detalle de eventos.
- Salvar el panel gráfico como una imagen
  - PNG con transparencia.
  - PDF LaTeX *friendly*.

UNIVERSIDAD SIMÓN BOLÍVAR

WYSIWYG

- Resumen visual de los eventos en cada hilo
  - Ejecución.
  - Recolector de basura – paralelo o secuencial.
  - Creación, migración, *wakeup* y destrucción de hilos.
  - Ciclo de vida de *sparks* – requiere GHC 7.3
- Presentación visual y textual conjunta
  - *Timeline* muestra los eventos visualmente.
  - *Events* muestra la descripción textual de los eventos.
  - Seleccionar en uno, desplaza el otro apropiadamante.
- Zoom in/out – detalle de eventos.
- Salvar el panel gráfico como una imagen
  - PNG con transparencia.
  - PDF LaTeX *friendly*.

Threadscope permite explorar la ejecución –
el programador es responsable de los ajustes.

"Mi algoritmo es más rápido"

¿Comparado con qué?

- Aplicar las técnicas de *profiling* usualmente
  - Obliga a controlar la flojera del lenguaje.
  - Sugiere reconsiderar los algoritmos.
- Ante alternativas. . . ¿cuál es mejor?
- *Benchmarking*
  - Comparar uno o más algoritmos sobre un mismo problema – mismos parámetros, mismo conjunto de datos.
  - Mediciones precisas de *tiempo*.
  - Garantizar que las mediciones sean estadísticamente sólidas.

UNIVERSIDAD SIMÓN BOLÍVAR

"Mi algoritmo es más rápido"

¿Comparado con qué?

- Aplicar las técnicas de *profiling* usualmente
  - Obliga a controlar la flojera del lenguaje.
  - Sugiere reconsiderar los algoritmos.
- Ante alternativas. . . ¿cuál es mejor?
- *Benchmarking*
  - Comparar uno o más algoritmos sobre un mismo problema – mismos parámetros, mismo conjunto de datos.
  - Mediciones precisas de *tiempo*.
  - Garantizar que las mediciones sean estadísticamente sólidas.

Todo esto es provisto por `Criterion`

UNIVERSIDAD SIMÓN BOLÍVAR

Fibonacci, otra vez

Recursivo e iterativo

```
rfib :: Integer -> Integer
rfib 1 = 1
rfib 2 = 1
rfib n = rfib (n-1) + rfib (n-2)

ifib :: Integer -> Integer
ifib n = go n (1,1)
  where go 1 (n1,n2) = n2
        go n (n1,n2) = go (n-1) (n1+n2,n1)
```

- Iterativo debe ser más rápido que recursivo – ¿cuánto más rápido?

UNIVERSIDAD SIMÓN BOLÍVAR

Preparando un *Benchmark*

¿Qué es un *Benchmark*?

- `Benchmark` es un tipo abstracto que representa una prueba – `Benchmarkable` es el *type class*
  - `Pure` – funciones puras.
  - `IO a` – cualquier acción I/O.

UNIVERSIDAD SIMÓN BOLÍVAR

Preparando un *Benchmark*

¿Qué es un *Benchmark*?

- `Benchmark` es un tipo abstracto que representa una prueba – `Benchmarkable` es el *type class*
  - `Pure` – funciones puras.
  - `IO a` – cualquier acción I/O.
- Las pruebas suelen tener un nombre para diferenciarlas.

```
bench :: Benchmarkable b => String -> b
      -> Benchmark
```

UNIVERSIDAD SIMÓN BOLÍVAR

Preparando un *Benchmark*

¿Qué es un *Benchmark*?

- `Benchmark` es un tipo abstracto que representa una prueba – `Benchmarkable` es el *type class*
  - `Pure` – funciones puras.
  - `IO a` – cualquier acción I/O.
- Las pruebas suelen tener un nombre para diferenciarlas.

```
bench :: Benchmarkable b => String -> b
      -> Benchmark
```

- `Criterion` provee métodos para forzar la evaluación.

```
whnf    :: (a -> b) -> a -> Pure
nf      :: NFData a => (a -> b) -> a -> Pure
whnfIO  :: NFData a => IO a -> IO ()
nfIO    :: NFData a => IO a -> IO ()
```

UNIVERSIDAD SIMÓN BOLÍVAR

Preparando un *Benchmark*

¿Qué es un *Benchmark*?

- Nuestras funciones son puras y sobre números – basta `whnf`
  - Para la función recursiva podríamos escribir

```
bench "rfib 10" $ whnf rfib 10
```

  - Para la función iterativa podríamos escribir

```
bench "ifib 10" $ whnf ifib 10
```

- Si quisiéramos probar una acción de I/O

```
bench "some I/O" $ print resultadoDeEvaluar
```

o bien

```
bench "some I/O" $ whnfIO accionDeIO
```

UNIVERSIDAD SIMÓN BOLÍVAR

Preparando un *Benchmark*

¿Cómo los ejecuto?

```
import Criterion
import Criterion.Main

main = defaultMain [
         bench "rfib 10" $ whnf rfib 10,
         bench "ifib 10" $ whnf ifib 10
       ]
```

- Ejecutar varios *benchmark* en secuencia

```
defaultMain :: [Benchmark] -> IO ()
```

- Compilar como un programa convencional – incluir optimización.

UNIVERSIDAD SIMÓN BOLÍVAR

Ejecutando el *Benchmark*

- `Criterion` incorpora opciones para controlar la prueba.

```
$ crit0 -?
```

- Resultados de las pruebas
  - Cónsola – descripción de la prueba y parámetros.
  - Reporte HTML – con todos los jugueticos JavaScript.
  - Archivo CSV – analizar con R u hoja de cálculo favorita.
- Basta ejecutar

```
$ crit0 -o resultado.html -u resultados.csv
```

  - Opción `-o` para reporte visual – HTML con gráficos.
  - Opción `-u` para estadísticas – sólo en CSV.

Interpretando los resultados

`Criterion` se calibra automáticamente

```
warming up
estimating clock resolution...
mean is 1.192844 us (640001 iterations)
found 109725 outliers among 639999 samples (17.1%)
  4 (6.3e-4%) low severe
  109721 (17.1%) high severe
estimating cost of a clock call...
mean is 30.43798 ns (12 iterations)
found 2 outliers among 12 samples (16.7%)
  1 (8.3%) high mild
  1 (8.3%) high severe
```

- Antes de probar nada, evalúa la precisión del *tick* de reloj.
- Determina la mínima precisión – tiempo mínimo medible en prueba.

Interpretando los resultados

¡. . . prueba por prueba!

```
benchmarking rfib 10
mean: 2.012186 us, lb 1.986171 us,
                   ub 2.049102 us, ci 0.950
std dev: 157.3901 ns, lb 120.9249 ns,
                      ub 200.3766 ns, ci 0.950
found 15 outliers among 100 samples (15.0%)
  15 (15.0%) high severe
variance introduced by outliers: 69.713%
variance is severely inflated by outliers
```

- Cotas inferior y superior para media y desviación estándar – se incluye el intervalo de confianza para el 95 %
- Se evalúa la influencia de los *outliers* en la varianza para determinar si esta prueba es consistente o no.

UNIVERSIDAD SIMÓN BOLÍVAR

Interpretando los resultados

Bootstrapping y'all

```
benchmarking rfib 10
ran 1024 iterations in 2.571106 ms
collecting 100 samples, 490 iterations each,
           in estimated 123.0314 ms
analysing with 100000 resamples
mean: 2.104418 us, lb 2.059567 us,
                   ub 2.157552 us, ci 0.950
std dev: 250.4258 ns, lb 219.5444 ns,
                      ub 277.1237 ns, ci 0.950
```

- Detrás de cámaras – técnica de *bootstrapping*
  - Se genera una muestra de corridas – establece línea base.
  - Cada muestra incluye varias corridas – dentro de la precisión del reloj.
  - Se genera un "re-muestreo" más grande – confirma línea base.
- La opción `-v` muestra esos detalles – ¡parametrizables!

¿Y cuál resultó más rápido?

La única parte "oscura" de Criterion

- La prueba de `rfib` muestra

```
collecting 100 samples,
           490 iterations each,
           in estimated 123.0314 ms
```

esto es aproximadamente 3983 evaluaciones por segundo.

- La prueba de `ifib` muestra

```
collecting 100 samples,
           7292 iterations each,
           in estimated 123.0269 s
```

esto es aproximadamente 59271 evaluaciones por segundo.

UNIVERSIDAD SIMÓN BOLÍVAR

Quiero saber más...

- Sección sobre *Profiling* – Manual de GHC
- ThreadScope en el WiKi de Haskell
- ThreadScope Tour en el WiKi de Haskell
- How to profile a Haskell program en el WiKi de Haskell
- Documentación de Criterion

UNIVERSIDAD SIMÓN BOLÍVAR