Programación Funcional Avanzada

Instituto de Computación, Facultad de Ingeniería
Universidad de la República, Uruguay

Heap Profiling

Memoria en Tiempo de Ejecución

Heap

- colección de clausuras
  - valores (funciones, datos)
  - thunks (suspensiones no evaluadas aún)
- garbage collector

Fugas de espacio / Space leaks

- Estructuras de datos que crecen más, o viven más, de lo previsto
- más garbage collection $\Rightarrow$ más tiempo de ejecución

Ejemplo: sumatoria

Una primera solución:

$$
\begin{aligned}
&\mathit{main} = \mathit{print}\ \$\ \mathit{sum}_1\ [1\,..\,10000000] \\
&\mathit{sum}_1\ [\,] \qquad\quad = 0 \\
&\mathit{sum}_1\ (x : \mathit{xs}) = x + \mathit{sum}_1\ \mathit{xs}
\end{aligned}
$$

o, usando *foldr*:

$$\mathit{sum}_1 = \mathit{foldr}\ (+)\ 0$$

Problemas:

- utiliza mucho stack (aumentar con `+RTS -K`)
- demora mucho

¿Qué está pasando?

Estadísticas de Runtime

- Ejecutar usando flag `-s`
  `./Sum1 +RTS -s`

- Estadísticas de ejecución:
  - Tiempo consumido en ejecución del programa y garbage collector
  - Memoria alojada
  - Tamaño máximo del heap

Profiling

Tres pasos:

- Compilar usando la opción de profiling
  `ghc --make -prof -fprof-auto -rtsopts Sum1.hs`
- Ejecutar usando alguna de las opciones de profiling
  `./Sum1 +RTS -p -hc -K500M`
  - básico: `-p`
  - heap: `-hc`
- Examinar la información de profiling generada
  - básico: `Sum1.prof`
  - heap: `Sum1.hp` (se puede generar `.ps` con `hp2ps`)

Ver: `http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html`

Problema

$$
\begin{aligned}
& \mathit{sum}_1\ [1,2,3,4,...] \\
& \equiv \{\, \mathit{Def.de}\ \mathit{sum}_1 \,\} \\
& 1 + \mathit{sum}_1\ [2,3,4,...] \\
& \equiv \{\, \mathit{Def.de}\ \mathit{sum}_1 \,\} \\
& 1 + (2 + \mathit{sum}_1\ [3,4,...]) \\
& \equiv \{\, \mathit{Def.de}\ \mathit{sum}_1 \,\} \\
& 1 + (2 + (3 + \mathit{sum}_1\ [4,...])) \\
& \equiv \\
& ...
\end{aligned}
$$

Se tiene que hacer la recursión sobre toda la lista antes de hacer la primera suma.

Solución 2: recursiva por cola (tail-recursive)

Intentar disminuir el uso de stack y memoria

$$main = print\ \$\ sum_2\ [1\,..\,10000000]$$

$$sum_2\ xs = sum_2Aux\ 0\ xs$$
  where
$$sum_2Aux\ acc\ [\,] = acc$$
$$sum_2Aux\ acc\ (x : xs) = sum_2Aux\ (acc + x)\ xs$$

o, usando *foldl*:

$$sum_2 = foldl\ (+)\ 0$$

Problema

$$
\begin{aligned}
&\mathit{sum}_2\ [1,2,3,4,...] \\
&\equiv \{\mathit{Def.de\ sum}_2\} \\
&\mathit{sum}_2\mathit{Aux}\ 0\ [1,2,3,4,...] \\
&\equiv \{\mathit{Def.de\ sum}_2\mathit{Aux}\} \\
&\mathit{sum}_2\mathit{Aux}\ (0+1)\ [2,3,4,...]) \\
&\equiv \{\mathit{Def.de\ sum}_2\mathit{Aux}\} \\
&\mathit{sum}_2\mathit{Aux}\ ((0+1)+2)\ [3,4,...]) \\
&\equiv \\
&... \\
&\equiv \{\mathit{Def.de\ sum}_2\mathit{Aux}\} \\
&(...((0+1)+2)+...)\ [\,] \\
&\equiv \{\mathit{Def.de\ sum}_2\mathit{Aux}\} \\
&(...((0+1)+2)+...)
\end{aligned}
$$

Se construye la suma completa antes de empezar a reducirla, pasándola en el acumulador.

Solución 3: forzar la suma

Utilizando aplicación estricta

$$
\begin{aligned}
&\mathit{main} = \mathit{print}\ \$\ \mathit{sum}_3\ [1\,..\,10000000] \\
&\mathit{sum}_3\ \mathit{xs} = \mathit{sum}_3\mathit{Aux}\ 0\ \mathit{xs} \\
&\quad \text{where} \\
&\qquad \mathit{sum}_3\mathit{Aux}\ \mathit{acc}\ [\,] = \mathit{acc} \\
&\qquad \mathit{sum}_3\mathit{Aux}\ \mathit{acc}\ (x : \mathit{xs}) = (\mathit{sum}_3\mathit{Aux}\ \$!\ \mathit{acc} + x)\ \mathit{xs}
\end{aligned}
$$

o, usando *Data.List.foldl'*:

$$
\mathit{sum}_3 = \mathit{foldl}'\ (+)\ 0
$$

Solución 4: BangPatterns

Extensión del lenguaje que permite hacer funciones estrictas en ciertas variables

```
{-# LANGUAGE BangPatterns #-}
main = print $ sum4 [1..10000000]
sum4 xs = sum4Aux 0 xs
  where
    sum4Aux !acc []     = acc
    sum4Aux !acc (x:xs) = sum4Aux (acc + x) xs
```

Solución 5: tipos de datos estrictos

Definición alternativa de *Integer*:

$$
\begin{aligned}
&\text{data } \mathit{StrictInteger} = \mathit{SI} \; ! \; \mathit{Integer} \\
&\mathit{main} = \mathit{print} \; \$ \; \mathit{sum}_5 \; [1 \,..\, 10000000] \\
&\mathit{sum}_5 \; \mathit{xs} = \mathit{sum}_5\mathit{Aux} \; (\mathit{SI} \; 0) \; \mathit{xs} \\
&\quad \text{where} \\
&\qquad \mathit{sum}_5\mathit{Aux} \; (\mathit{SI} \; \mathit{acc}) \; [\,] \qquad\quad = \mathit{acc} \\
&\qquad \mathit{sum}_5\mathit{Aux} \; (\mathit{SI} \; \mathit{acc}) \; (x : \mathit{xs}) = \mathit{sum}_5\mathit{Aux} \; (\mathit{SI} \; (\mathit{acc} + x)) \; \mathit{xs}
\end{aligned}
$$

Un constructor es estricto en los argumentos indicados con la anotación "!"

Regla General

Si se esperan resultados parciales, o se quiere utilizar listas infinitas, usar *foldr*
Ejemplos: *map*, *filter*

Si el operador es estricto, usar *foldl'*
Ejemplos: *sum*, *product*

En otro caso, usar *foldl*
Ejemplo: *reverse*

Pasando a GHC la flag de optimización `-O` se realiza **strictness analysis**