Tema 10: Evaluación perezosa

Informática (2009–10)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 10: Evaluación perezosa

1. Estrategias de evaluación
2. Terminación
3. Número de reducciones
4. Estructuras infinitas
5. Programación modular
6. Aplicación estricta

Tema 10: Evaluación perezosa

Estrategias de evaluación

- Para los ejemplos se considera la función

```
mult :: (Int,Int) -> Int
mult (x,y) = x*y
```

- Evaluación mediante paso de parámetros por valor (o por más internos):

```
      mult (1+2,2+3)
=     mult (3,5)          [por def. de +]
=     3*5                 [por def. de mult]
=     15                  [por def. de *]
```

- Evaluación mediante paso de parámetros por nombre (o por más externos):

```
      mult (1+2,2+3)
=     (1+2)*(3+5)         [por def. de mult]
=     3*5                 [por def. de +]
```

Evaluación con lambda expresiones

- Se considera la función

```
mult' :: Int -> Int -> Int
mult' x = \y -> x*y
```

- Evaluación:

```
    mult' (1+2) (2+3)
=   mult' 3 (2+3)          [por def. de +]
=   (λy → 3*y) (2+3)       [por def. de mult']
=   (λy → 3*y) 5           [por def. de +]
=   3*5                    [por def. de +]
=   15                     [por def. de *]
```

Tema 10: Evaluación perezosa

Procesamiento con el infinito

- Definición de infinito

```
inf :: Int
inf = 1 + inf
```

- Evaluación de infinito en Haskell:

```
*Main> inf
  C-c C-cInterrupted.
```

- Evaluación de infinito:

$$
\begin{array}{lll}
 & \text{inf} & \\
= & 1 + \text{inf} & \text{[por def. inf]} \\
= & 1 + (1 + \text{inf}) & \text{[por def. inf]} \\
= & 1 + (1 + (1 + \text{inf})) & \text{[por def. inf]} \\
= & \ldots &
\end{array}
$$

Procesamiento con el infinito

- Evaluación mediante paso de parámetros por valor:

$$\begin{aligned} & \texttt{fst (0,inf)} \\ = \quad & \texttt{fst (0,1 + inf)} && \text{[por def. inf]} \\ = \quad & \texttt{fst (0,1 + (1 + inf))} && \text{[por def. inf]} \\ = \quad & \texttt{fst (0,1 + (1 + (1 + inf)))} && \text{[por def. inf]} \\ = \quad & \ldots \end{aligned}$$

- Evaluación mediante paso de parámetros por nombre:

$$\begin{aligned} & \texttt{fst (0,inf)} \\ = \quad & \texttt{0} && \text{[por def. fst]} \end{aligned}$$

- Evaluación Haskell con infinito:

```
*Main> fst (0,inf)
0
```

Tema 10: Evaluación perezosa

Número de reducciones según las estrategias

- Para los ejemplos se considera la función

```
cuadrado :: Int -> Int
cuadrado n = n * n
```

- Evaluación mediante paso de parámetros por valor:

	cuadrado (1+2)	
=	cuadrado 3	[por def. +]
=	3*3	[por def. cuadrado]
=	9	[por def. de *]

- Evaluación mediante paso de parámetros por nombre:

	cuadrado (1+2)	
=	(1+2)*(1+2)	[por def. cuadrado]
=	3*(1+2)	[por def. de +]
=	3*3	[por def. de +]
=	9	[por def. de *]

Evaluación perezosa e impaciente

- En la evaluación mediante paso de parámetros por nombre los argumentos pueden evaluarse más veces que en el paso por valor.
- Se puede usar punteros para compartir valores de expresiones.
- La evaluación mediante paso de parámetros por nombre usando punteros para compartir valores de expresiones se llama evaluación perezosa.
- La evaluación mediante paso de parámetros por valor se llama evaluación impaciente.
- Evaluación perezosa del ejemplo anterior:

```
      cuadrado (1+2)
=     x*x con x = 1+2     [por def. cuadrado]
=     3*3                 [por def. de +]
=     9                   [por def. de *]
```

- Haskell usa evaluación perezosa.

Tema 10: Evaluación perezosa

Programación con estructuras infinitas

- `unos` es una lista infinita de unos.

```
unos :: [Int]
unos = 1 : unos
```

- Evaluación:

$$
\begin{array}{lll}
 & \text{unos} & \\
= & 1 : \text{unos} & \text{[por def. unos]} \\
= & 1 : (1 : \text{unos}) & \text{[por def. unos]} \\
= & 1 : (1 : (1 : \text{unos})) & \text{[por def. unos]} \\
= & \ldots &
\end{array}
$$

- Evaluación en Haskell:

```
*Main> unos
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...
```

Evaluación con estructuras infinitas

- Evaluación impaciente:

```
    head unos
=   head (1 : unos)                [por def. unos]
=   head (1 : (1 : unos))          [por def. unos]
=   head (1 : (1 : (1 : unos)))    [por def. unos]
=   ...
```

- Evaluación perezosa:

```
    head unos
=   head (1 : unos)    [por def. unos]
=   1                  [por def. head]
```

- Evaluación Haskell:

```
*Main> head unos
1
```

Tema 10: Evaluación perezosa

Programación modular

- La evaluación perezosa permite separar el control de los datos.
- Para los ejemplos se considera la función

Prelude

```
take :: Int -> [a] -> [a]
take n _   | n <= 0   = []
take _ []             = []
take n (x:xs)         = x : take (n-1) xs
```

- Ejemplo de separación del control (tomar 2 elementos) de los datos (una lista infinita de unos):

	take 2 unos	
=	take 2 (1 : unos)	[por def. unos]
=	1 : (take 1 unos)	[por def. take]
=	1 : (take 1 (1 : unos))	[por def. unos]
=	1 : (1 : (take 0 unos))	[por def. take]
=	1 : (1 : [])	[por def. take]
=	[1,1]	[por notación de listas]

Terminación de evaluaciones con estructuras infinitas

- Ejemplo de no terminación:

```
*Main> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
```

- Ejemplo de terminación:

```
*Main> take 3 [1..]
[1,2,3]
```

- Ejemplo de no terminación:

```
*Main> filter (<=3) [1..]
[1,2,3  C-c C-c Interrupted.
```

- Ejemplo de no terminación:

```
*Main> takeWhile (<=3) [1..]
[1,2,3]
```

La criba de Erastótenes

- La criba de Erastótenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
	3		5		7		9		11		13		15	...
			5		7				11		13			...
					7				11		13			...
									11		13			...
											13			...

- Definición

```
primos :: [Int ]
primos = criba [2..]

criba :: [Int] -> [Int]
criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]
```

La criba de Erastótenes

- Evaluación:

```
take 15 primos ⇝ [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

- Cálculo:

```
  primos
= criba [2..]
= criba (2 : [3..])
= 2 : (criba [x | x <- [3..], x `mod` 2 /= 0])
= 2 : (criba (3 : [x | x <- [4..], x `mod` 2 /= 0]))
= 2 : 3 : (criba [x | x <- [4..], x `mod` 2 /= 0,
                                  x `mod` 3 /= 0])
= 2 : 3 : (criba (5 : [x | x <- [6..], x `mod` 2 /= 0,
                                       x `mod` 3 /= 0]))
= 2 : 3 : 5 : (criba ([x | x <- [6..], x `mod` 2 /= 0,
                                       x `mod` 3 /= 0,
                                       x `mod` 5 /= 0]))
= ...
```

Tema 10: Evaluación perezosa

Ejemplo de programa sin aplicación estricta

- `(sumaNE xs)` es la suma de los números de `xs`. Por ejemplo,

  `sumaNE [2,3,5]` $\rightsquigarrow$ `10`

```
sumaNE :: [Int] -> Int
sumaNE xs = sumaNE' 0 xs

sumaNE' :: Int -> [Int] -> Int
sumaNE' v []     = v
sumaNE' v (x:xs) = sumaNE' (v+x) xs
```

Ejemplo de programa sin aplicación estricta

- Evaluación: :

	sumaNE [2,3,5]	
=	sumaNE' 0 [2,3,5]	[por def. sumaNE]
=	sumaNE' (0+2) [3,5]	[por def. sumaNE']
=	sumaNE' ((0+2)+3) [5]	[por def. sumaNE']
=	sumaNE' (((0+2)+3)+5) []	[por def. sumaNE']
=	((0+2)+3)+5	[por def. sumaNE']
=	(2+3)+5	[por def. +]
=	5+5	[por def. +]
=	10	[por def. +]

Ejemplo de programa con aplicación estricta

- `(sumaE xs)` es la suma de los números de `xs`. Por ejemplo,

  `sumaE [2,3,5]` $\rightsquigarrow$ `10`

```
sumaE :: [Int] -> Int
sumaE xs = sumaE' 0 xs

sumaE' :: Int -> [Int] -> Int
sumaE' v []     = v
sumaE' v (x:xs) = (sumaE' $! (v+x)) xs
```

Ejemplo de programa con aplicación estricta

- Evaluación: :

```
    sumaE [2,3,5]
=   sumaE' 0 [2,3,5]            [por def. sumaE]
=   (sumaE' $! (0+2)) [3,5]     [por def. sumaE']
=   sumaE' 2 [3,5]              [por aplicación de $!]
=   (sumaE' $! (2+3)) [5]       [por def. sumaE']
=   sumaE' 5 [5]                [por aplicación de $!]
=   (sumaE' $! (5+5)) []        [por def. sumaE']
=   sumaE' 10 []                [por aplicación de $!]
=   10                          [por def. sumaE']
```

Comparación de consumo de memoria

- Comparación de consumo de memoria:

```
*Main> sumaNE [1..1000000]
*** Exception: stack overflow
*Main> sumaE [1..1000000]
1784293664
*Main> :set +s
*Main> sumaE [1..1000000]
1784293664
(2.16 secs, 145435772 bytes)
```

Plegado estricto

- Versión estricta de `foldl` en el `Data.List`

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []     = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

- Comparación de plegado y plegado estricto:s

```
*Main> foldl (+) 0 [2,3,5]
10
*Main> foldl' (+) 0 [2,3,5]
10
*Main> foldl (+) 0 [1..1000000]
*** Exception: stack overflow
*Main> foldl' (+) 0 [1..1000000]
500000500000
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. Cap. 7: Eficiencia.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 12: Lazy evaluation.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 2: Introducción a Haskell.
   - Cap. 8: Evaluación perezosa. Redes de procesos.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 17: Lazy programming.