PROGRAMACIÓN FUNCIONAL

Evaluación *Lazy*

Evaluación *Lazy*

- Primera Parte:
  - Eficiencia en funcional
  - Evaluación *lazy*
- Segunda Parte:
  - Estructuras de datos infinitas
  - Ejemplos

Eficiencia

- ¿Cómo se mide la eficiencia de un programa?
  - En unidades abstractas que dependen del modelo de cómputo (e.g. asignaciones, reducciones, etc.)
  - Mediante estimaciones para TODAS las posibles computaciones de un programa
- Modelos de eficiencia
  - Peor caso
  - Caso promedio
  - Costo amortizado
  - Modelos probabilísticos

Eficiencia

- Nos interesa la eficiencia
  - en tiempo (¿cuánto tarda el programa?)
  - en espacio (¿cuánta 'memoria' es necesaria?)
- En un lenguaje funcional
  - tiempo = número de reducciones
  - espacio = tamaño de las expresiones intermedias
- Pero...
  - ¡el orden de reducción afecta la eficiencia!

Eficiencia

- Ejemplo 1: considere la función

  quin x = x + x + x + x + x

- ¿Cuánto cuesta reducir (quin (fib 22))?
  (sabemos que (fib 22) cuesta ~1.000.000 reducciones)
  - Con orden aplicativo: ~1.000.000 reds.
  - Con orden normal: ~ 5.000.000 reds.
    - ¡se copia (fib 22) cinco veces, y cada copia se reduce en forma separada!

Eficiencia

- Ejemplo 2: considere además la función

  const x y = x

- ¿Cuánto cuesta reducir (const 3 (quin (fib 22)))?
  - Con orden aplicativo: ~1.000.000 reds.
  - Con orden normal: ¡1 reducción!
    - ¡el argumento que no se precisa, NO SE REDUCE!
- Entonces, ¿cuál orden usar?

Eficiencia

- Ejemplo 3: considere las funciones

```
fd :: (Int, Int) -> Int
fd (a,b) = a+a
test :: (Int, Int)
test = (3, quin (fib 22))
```

- ¿Cuántas reducciones lleva (fd test)?
  - Con orden aplicativo: ~1.000.000 reds.
  - Con orden normal: 3 reducciones
- Entonces, ¿cuál orden usar?

Eficiencia

- Para medir eficiencia debemos determinar con mayor precisión el orden de reducción
- Sabemos que el orden normal tiene ventajas sobre el aplicativo
- ¿Podríamos conseguir eliminar la desventaja del costo de duplicación?
  - Sí, si agregamos ciertas características al orden normal...

Evaluación *Lazy*

- Evaluación perezosa (o *lazy*)
  - Evaluación en orden normal, con las siguientes características adicionales:
    - El argumento de una función sólo se evalúa cuando es necesario para el cómputo
    - Un argumento no es necesariamente evaluado por completo; sólo se evalúan aquellas partes que contribuyen efectivamente al cómputo
    - Si un argumento se evalúa, tal evaluación se realiza sólo una vez

Evaluación *Lazy*

- Ventajas
  - usualmente mayor eficiencia
  - mejores condiciones de terminación
  - no hay necesidad de estructuras intermedias al componer funciones
  - manipulación de estructuras de datos infinitas
  - manipulación de computaciones infinitas
- Desventajas
  - es difícil calcular el costo de ejecución (pues depende del contexto en el que se usa...)

Evaluación *Lazy*

- Eficiencia:
  - ¿Cuánto cuesta reducir (quin (fib 22))?
    - Con orden aplicativo: ~1.000.000 reds.
    - Con evaluación lazy: ~1.000.000 reds.
- Mejores condiciones de terminación:
  - ¿A qué reduce (const 3 (1/0))?
    - Con orden aplicativo: error
    - Con evaluación lazy: 3

Evaluación *Lazy*

- No se necesitan estructuras intermedias:
  - ¿Cómo reduce (map f (map g [1..30]))?

    map f (map g [1..30]) = (por fromTo.2)

    map f (map g (1 : [2..30])) = (por map.2)

    map f (g 1 : map g [2..30]) = (por map.2)

    f (g 1) : map f (map g [2..30]) = ... =

    f (g 1) : f (g 2) : map f (map g [3..30]) = ... =

    [f (g 1), f (g 2), ..., f (g 30)]

Evaluación *Lazy*

- Calcular el costo de ejecución es difícil:

```
isort = foldr insert [ ]
insert x [ ] = [ x ]
insert x (y:ys) = if (x>y) then y : insert x ys
                           else x : y : ys
lej = [8,6,1,7,5,9,3,2,4]++[1000,999..10]
```

  - ¿Cuánto cuesta calcular (isort lej)?
    - Aproximadamente 3.500.000 reds.
  - ¿Y (head (isort lej))?
    - Aproximadamente 8.000 reds.

Estructuras infinitas

- Un estructura infinita
  - no tiene formal normal
  - en cada paso de reducción aumenta la cantidad de información que hay en ella
- Ejemplo:

```
nats = from 0

from n = n : from (n+1)
```

- ¿Cómo reduce la expresión nats?

Estructuras infinitas

- Considere las siguientes definiciones

```
iterate :: ??
iterate f x = x : iterate f (f x)

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

- ¿Cómo sería la reducción de (iterate (+1) 0)?
- ¿Y la de (take 4 (iterate (+1) 0))?

Estructuras infinitas

- Todo tipo algebraico recursivo
  - tiene los elementos inductivos ya vistos,

  y además
  - tiene elementos infinitos y parcialmente definidos
- Una demostración por inducción estructural
  - sólo prueba la propiedad para los elementos inductivos (finitos y definidos)
  - NO lo prueba ni para $\perp$ ni para los elementos infinitos

Elementos parciales

- Un elemento parcialmente definido
  - es distinto de $\perp$
  - contiene a $\perp$ en alguna de sus partes,
- Ejemplo:

```
rara :: Int -> [ Int ]
rara 0 = [ ]
rara n = error "Soy un mal elemento" : rara (n-1)
```

  - ¿Qué valor tiene (length (rara 10))?
  - ¿Y (head (rara 10))?

Elementos parciales

- Otro ejemplo:

```
otraRara :: Int -> [ Int ]
otraRara 0 = error "Soy una mala lista"
otraRara n = n : otraRara (n-1)
```

  - ¿Qué valor tiene (head (otraRara 3))?
  - ¿Y (length (otraRara 3))?

- Abusando de la notación, podríamos decir que:

```
rara 10 = [ ⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥, ⊥ ]
otraRara 3 = 3 : 2 : 1 : ⊥
```

Estructuras infinitas

- Ventajas:
  - modularidad de programas
  - claridad
- Modularidad:
  - la estructura es una fuente de recursos
  - el programa que la usa determina cuántos precisa
  - no se calculan más de los necesarios
- Claridad:
  - cada parte de un programa resulta más clara

Estructuras infinitas

- **Ejemplo:** generación de números primos

```
primos = criba (iterate (+1) 2)
criba (x:xs) = x : criba (filter (noMultiplo x) xs)
noMultiplo x y = y `mod` x /= 0
```

- ¿Cuál es el valor de (take 20 primos)?

```
take 20 primos =
  [  2,  3,  5,  7, 11, 13, 17, 19, 23, 29,
    31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
```

  - Este algoritmo no es eficiente; hay otros mejores

Resumen

- Evaluación lazy
  - permite mayor modularidad
  - evita estructuras intermedias
  - permite estructuras infinitas y parciales
  - dificulta el cálculo de eficiencia