Algoritmos y Estructuras de Datos I

Primer cuatrimestre de 2013

Departamento de Computación - FCEyN - UBA

Programación funcional - clase 5

Órdenes de evaluación - Alto orden

Reducción

Modelo de cómputo: Especificación que define cómo se calcula el valor de una expresión.

Reducción: Mecanismo de evaluación en Haskell

- Reemplazar una subexpresión por otra
  - Subexpresión reemplazada:
    - Instancia del lado izquierdo de una ecuación orientada
    - Se llama redex (*reducible expression*) o radical
  - Subexpresión reemplazante:
    - Lado derecho, instanciado de manera acorde
    - El resto de la expresión no cambia

Instanciación: asignación de expresiones a variables de un pattern

Ejemplo

- Tenemos la expresión:
  `suma (restar 2 (amigos Juan)) 4`
- ... y tenemos la ecuación:
  `restar x y = x - y`
- Reducción:
  1. Busco un redex y una asignación
     `suma (restar 2 (amigos Juan)) 4`
     (`restar 2 (amigos Juan)`: redex)
     - Asignación:
       `x` ← `2`
       `y` ← `(amigos Juan)`
  2. Reemplazo el redex con esa asignación

`suma (restar 2 (amigos Juan)) 4` ⇝ `suma (2 - (amigos Juan)) 4`

Formas normales

- Las expresiones se reducen hasta que no haya más redexes
- Como resultado se obtiene una forma normal (expresión que involucra solamente constantes y constructores)
- **Mecanismo de reducción:**
  1. Si la expresión está en forma normal, terminamos
  2. Si no, buscar un redex
  3. Reemplazarlo y volver a empezar

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**
  - `f x = f (f x)` ¿cuánto vale `f 3`?
  - `infinito = infinito + 1` ¿cuánto vale `infinito`?
  - `inverso x | x /= 0 = 1 / x` ¿cuánto vale `inverso 0`?
- Si se consigue, ¿toda estrategia encuentra la misma? **Sí**
- Esta propiedad se llama confluencia

Bottom

- Las expresiones que no tienen forma normal se llaman indefinidas
  - Se puede decir que su valor es $\bot$
- ¿Podemos definir en Haskell una función que siempre se indefine?

```
bottom :: a
bottom = bottom
```

- Cualquier intento de evaluar bottom se indefine

```
g :: Int -> Int
g x = if x == bottom then 1 else 0
```

`g 2` $\rightsquigarrow \bot$

Indefinición

- Le pasamos un valor definido a una función ...
  - Funciones parciales: a veces devuelven $\bot$
  - Funciones totales: nunca devuelven $\bot$
- Le pasamos $\bot$ a una función, ¿devuelve $\bot$?
  - No siempre, depende de sus ecuaciones y del orden de reducción
  - Funciones estrictas: $f \bot \rightsquigarrow \bot$
  - Funciones no estrictas: $f \bot \rightsquigarrow$ valor

Funciones totales vs. parciales

- Ejemplo: función total:

```
suc :: Integer -> Integer
suc x = x + 1
```

- Ejemplo: función parcial:

```
recip :: Float -> Float
recip x | x /= 0 = 1/x
```

- Las dos son funciones estrictas: si les pasamos $\bot$, devuelven $\bot$

Funciones estrictas vs. no estrictas

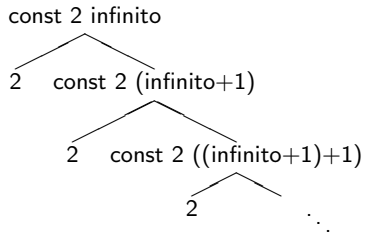
- ```
  const :: a -> b -> a
  const x y = x
  ```
- ¿Cuánto vale `const 2 bottom`?
- Depende del **diseño** del lenguaje!
- El secreto está en el orden de evaluación.

Orden de evaluación

- Forma de elegir el próximo redex
- Recordar confluencia!
  - Si por dos órdenes llegamos a valores definidos, es el mismo valor
  - ... pero puede ser que un orden llegue a $\perp$ y otro no

```
infinito :: Integer
infinito = infinito + 1
```

```
const :: a -> b -> a
const x y = x
```



Órdenes de evaluación

- Orden aplicativo
  - Primero redexes internos
  - Primero los argumentos, después la aplicación
- Orden normal
  - El redex más externo para el que pueda hacer pattern matching
  - Primero la aplicación, después los argumentos (si se necesitan)
- Los dos empiezan a izquierda en caso de más de un redex del mismo nivel
- El orden normal siempre encuentra la forma normal (si la hay)

Ejemplos de evaluación (1/3)

Primero redexes internos: `f x = 0`

f (1/0) se indefine

Primero redexes externos: `f x = 0`

f (1/0) $\rightsquigarrow$ 0
```

Ejemplos de evaluación (2/3)

Primero redexes internos:

```
infinito :: Integer
infinito = infinito + 1
```

```
const :: a -> b -> a
const x y = x
```

const 2 infinito $\rightsquigarrow$ const 2 (infinito+1) $\rightsquigarrow$ const 2 ((infinito+1)+1) $\rightsquigarrow$
const 2 (((infinito+1)+1)+1) $\rightsquigarrow$ ...

Primero redexes externos:

```
infinito :: Integer -> Integer
infinito = infinito + 1
```

```
const :: a -> b -> a
const x y = x
```

const 2 infinito $\rightsquigarrow$ 2

Ejemplos de evaluación (3/3)

Primero redexes internos:

```
head :: [a] -> a
head (x:xs) = x
```

```
tail :: [a] -> [a]
tail (x:xs) = xs
```

```
inc :: [a] -> [a]
inc [] = []
inc (x:xs) = (x+1):inc xs
```

head (tail (inc [1,2,3,4])) $\rightsquigarrow$ head (tail (2:inc [2,3,4])) $\rightsquigarrow$
head (tail (2:3:inc [3,4])) $\rightsquigarrow$ head (tail (2:3:4:inc [4])) $\rightsquigarrow$
head (tail (2:3:4:5:inc [])) $\rightsquigarrow$ head (tail (2:3:4:5:[])) $\rightsquigarrow$ head (3:4:5:[]) $\rightsquigarrow$ 3

Primero redexes externos:

```
head :: [a] -> a
head (x:xs) = x
```

```
tail :: [a] -> a
tail (x:xs) = xs
```

```
inc :: [a] -> a
inc [] = []
inc (x:xs) = (x+1):inc xs
```

head (tail (inc [1,2,3,4])) $\rightsquigarrow$ head (tail (2:inc [2,3,4])) $\rightsquigarrow$ head (inc [2,3,4]) $\rightsquigarrow$
head (3:inc [3,4]) $\rightsquigarrow$ 3

Propiedades de los órdenes de evaluación

- En el orden aplicativo ...
  - `const` es estricta
  - Todas las funciones son estrictas!
- En el orden normal ...
  - `const` es no estricta
  - Hay funciones estrictas y no estrictas, depende del hecho de que necesiten evaluar todos sus argumentos

Órdenes de evaluación y performance

```
fib 1 = 1
fib 2 = 1
fib n | n > 2 = fib (n-1) + fib (n-2)
```

¿Cuántas reducciones necesito?

- Supongamos que `fib 20` hace 200 000 reducciones
- ¿Cuántas reducciones hace `const 3 (fib 20)` ?
  - Aplicativo: 200 001
  - Normal: 1
- Sea `quin x = x + x + x + x + x`
- ¿Cuántas reducciones hace `quin (fib 20)` ?
  - Aplicativo: 200 005
  - Normal: 1 000 005

Evaluación *lazy*

- Es el orden de evaluación que utiliza Haskell
- Orden normal (primero redexes externos), pero aprovechando la transparencia referencial.
  - Si una expresión vuelve a aparecer, recuerda el valor anterior
  - En el ejemplo anterior, `quin (fib 20)` hace 200 005 reducciones

Estructuras infinitas

- Es una ventaja de la evaluación lazy
- Ejemplos:
  - ```
    naturales, pares, impares :: [Integer]
    naturales = sucesion 0 1
    impares = sucesion 1 2
    pares = sucesion 0 2
    ```
  - ```
    sucesion :: Integer -> Integer -> [Integer]
    sucesion n p = n:sucesion (n+p) p
    ```
- Evaluación de estructuras infinitas
  - `head naturales` $\rightsquigarrow$ `head (sucesion 0 1)` $\rightsquigarrow$ `head (0:sucesion 1 1)` $\rightsquigarrow$ `0`
  - `take 10 impares` $\rightsquigarrow \cdots \rightsquigarrow$ `[1,3,5,7,9,11,13,15,17,19]`

Alto orden

- Las funciones de alto orden son aquellas que **reciben o devuelven funciones**.
- Iteran una función arbitraria (pasada como argumento) sobre una estructura de datos, en un cierto orden.
- Construyen un valor de retorno.
- Las tres más usadas:
  - `map`
  - `filter`
  - `fold`

map

- Aplica la función a cada uno de los elementos de la lista, y construye la lista de resultado.
- Hace en Haskell exactamente lo mismo que nuestras listas por comprensión en el lenguaje de especificación, sin condiciones.
- ```
  map :: (a -> a) - [a] -> [a]
  map f [] = []
  map f (x:xs) = f x : map f xs
  ```
- Ejemplos:
  - `map (+1) [1..5]` $\rightsquigarrow$ `[2, 3, 4, 5, 6]`
  - `map (toLower) ''abcDEFG12+*#''` $\rightsquigarrow$ `''abcdefg12+*#''`
  - `map ('mod'3) [1..10]` $\rightsquigarrow$ `[1, 2, 0, 1, 2, 0, 1, 2, 0, 1]`
  ```

filter

- Filtra los elementos de una lista que cumplen con una condición, dada por una función que retorna Bool.
- ```
  filter: (Int -> Bool) -> [Int] -> [Int]
  filter f [] = []
  filter f (x:xs) | f x = x:filter f xs
                  | otherwise = filter f xs
  ```
- Filtrar los elementos mayores a dos de una lista:
  ```
  filtroMayoresADos xs = filter mayorADos xs
                       where mayorADos x = x>2
  ```
- Filtrar los elementos pares de una lista:
  ```
  filtroPares xs = filter par xs
                 where par x = (x `mod` 2 == 0)
  ```


fold

- sumar los elementos de una lista
  ```
  suma [] = 0
  suma (n:ns) = n + suma ns
  ```
- verificar que sean todos verdaderos
  ```
  all [] = True
  all (b:bs) = b && all bs
  ```
- aplanar una lista de listas
  ```
  concat [] = []
  concat (xs:xss) = xs ++ concat xss
  ```


fold

- Definición de `foldr`:
  ```
  foldr f a [] = a
  foldr f a (x:xs) = x `f` (foldr f a xs)
  ```
  - `suma = foldr (+) 0`
  - `all = foldr (&&) True`
  - `concat = foldr (++) []`
- ¿Cuál es el tipo de `foldr`?
  - De los ejemplos se deduce `(a -> a -> a) -> a -> [a] -> a`
  - Pero puede ser un poco más general:
    ```
    (a -> b -> b) -> b -> [a] -> b
    ```
- Notar que la currificación es muy adecuada para usar en combinación con fold
    ```