PROGRAMACIÓN FUNCIONAL

Modelo Funcional: Reducción

Reducción

- Computación por reducción
- Propiedades de la reducción
- Representando errores: bottom
  - Funciones parciales y totales
- Órdenes de reducción
  - Funciones estrictas y no-estrictas

Computación

- ¿Cómo calcular el valor de una expresión?

  1) Reemplazar una subexpresión que coincida con una instancia del lado izquierdo de una ecuación por la correspondiente instancia del lado derecho.

  2) Repetir el paso 1) hasta que no haya ninguna subexpresión que cumpla la condición.

- Mecanismo de reducción

Reducción - Definiciones

- Redex (*reducible expression*)
  - subexpresión que coincide con una instancia del lado izquierdo de una ecuación
- Forma normal
  - expresión que no contiene redexes
- Mecanismo de reducción

  1) Localizar un redex

  2) Reemplazarlo

  3) Repetir hasta que la expresión esté en forma normal

Reducción

- ¿Qué propiedades tiene la reducción?
  - ¿Siempre termina?
    (¿la forma normal existe?)
    - Normalización
  - Cuando termina, ¿da un único valor?
    (¿la forma normal es única?)
    - Confluencia
  - ¿Hay más de una forma de reducir?
    Si es así, ¿son todas equivalentes?
    - Órdenes de reducción

Normalización

- Considere el siguiente script:

```
infinito :: Int
infinito = infinito + 1

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

- ¿Cómo se reduce la expresión infinito?
- ¿Y (recip 0)?

Normalización

- Forma normal
  - Expresión que no se puede reducir
  - Por abuso de lenguaje, *valor*
- La reducción pretende obtener la forma normal
- No toda expresión tiene forma normal
- ¿Puede ser que una expresión tenga forma normal, pero la reducción no la encuentre?

Normalización

- ¿Cuál es el valor de infinito? ¿Existe?
- Visión operacional:
  - expresiones cuyas computaciones no terminan
  - expresiones que no están definidas
- Visión denotacional:
  - BOTTOM (⊥): valor teórico que representa a un error o a una computación que no termina.
  - ¡NO SE PUEDE manejar de manera operacional!

Bottom

- Bottom es un valor especial
  - denotado ⊥
  - representa computaciones erróneas o que no terminan
  - no se puede preguntar si algo es ⊥ sin obtener ⊥
- Todos los tipos deben contener este valor de error, pues todos deben poder devolver error.
- ¿Cuál es, entonces el tipo de ⊥?

Bottom

- En ecuaciones

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

  - La expresión bottom, si se evalúa, nunca termina

- En Haskell

```
error :: String -> a
-- error es predefinida, ¡pero sin ecuaciones!
```

  - La expresión (error "Mensaje de error") aborta y devuelve el mensaje "Mensaje de error"

Bottom

- Una expresión vale ⊥ si
  - no está definida
  - su computación no termina
- Considerar

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

- ¿Cuánto vale (f 2)?
- ¿Y (f bottom)?

Bottom y Funciones

- ¿Puede una función devolver ⊥ al recibir un valor definido?
    - Funciones parciales y totales
- ¿Y qué pasa cuando recibe ⊥?
  - Si lo precisa...
  - Si no lo precisa...
    - Funciones estrictas y no-estrictas

Funciones Parciales

- Considerar las siguientes definiciones

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

- Para todo `n` definido, la expresión `(succ n)` está definida
  - La función `succ` es TOTAL
- La expresión `(recip n)` no está definida para `n=0`
  - La función `recip` es PARCIAL

Funciones Parciales

- Función Parcial
  - función que no está definida (vale ⊥) al ser aplicada a un valor definido (≠⊥)
- ¿Por qué no cambiar el tipo para que exprese exactamente el dominio?
  - Porque en ese caso no existe un programa que implemente inferencia de tipos
    - (Es un resultado de Teoría de la Computación)

Funciones Parciales

- ¿Qué diferencia hay entre estas dos funciones?

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

recipE :: Float -> Float
recipE x | x > 0 = 1/x
recipE x | x == 0 = error ``No puedo calcular 1/0``
```

- (recip x = recipE x) para todo x.
- Si x=0, ambas valen ⊥ PERO
  ¡el mensaje de error que ofrecen es diferente!

Funciones No Estrictas

- Considerar la siguiente definición

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

- ¿Cuál es el valor de (const infinito 2)?
- ¿Y el de (const 2 infinito)?
- ¿Es necesario el primer argumento para calcular el resultado de const?
- ¿Y el segundo?

Funciones No Estrictas

- Función estricta
  - cuando recibe ⊥, retorna ⊥
- Función no estricta
  - cuando recibe ⊥, puede retornar un valor definido (≠⊥)
- Si la función precisa el valor para dar el resultado, entonces es estricta.
- Pero si no lo precisa, podemos elegir...

Orden de Evaluación

- ¿Cómo saber cuál redex elegir si hay más de uno? ¿Hace diferencia?

const 2 infinito

2

const 2 (infinito + 1)

2

const 2 ((infinito + 1) + 1)

2

...

Orden de Evaluación

- Orden de evaluación
  - Algoritmo para la elección del redex a reducir
- Orden APLICATIVO
  - Primero los redexes internos

    (≡ primero los argumentos y luego la aplicación)
- Orden NORMAL
  - Primero los redexes externos

    (≡ primero la aplicación, y si aún están, los argumentos)
- En ambos casos, si hay más de uno al mismo nivel, elige el de más a la izquierda

Orden de Evaluación

- ¿Cuál orden encuentra la forma normal siempre que exista?
- ¿Cuál es el resultado de (const 2 infinito) si reducimos con orden aplicativo?
- ¿Y si reducimos con orden normal?
- ¿Observa relación entre el orden de evaluación y el hecho de que una función sea estricta o no?

Orden de Evaluación

- La elección del orden de evaluación implica realizar la elección de si las funciones serán estrictas o no (cuando no precisen su argumento)
  - Orden Aplicativo
    $\Rightarrow$ TODAS las funciones son estrictas
  - Orden Normal
    $\Rightarrow$ hay funciones estrictas y no estrictas
    (no estrictas las que no precisen su argumento)

Orden de evaluación

- En un lenguaje con efectos laterales, es NECESARIO que las funciones sean estrictas
  - ¿Por qué?
- En un lenguaje puro las funciones PUEDEN ser no estrictas
  - DECISIÓN DE DISEÑO
- Los diseñadores de Haskell eligieron que tenga funciones no estrictas

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

- ¿Cuál orden usar?
  - ¿Orden aplicativo?
  - ¿Orden normal?
- Solución para mejorar la eficiencia:
  - 'recordar' que las copias de x son el mismo valor, para no replicar trabajo
  - Da origen a la evaluación *lazy*

Evaluación *Lazy*

- Evaluación perezosa (o *lazy*)
  - Evaluación en orden normal, con las siguientes características adicionales:
    - 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 y computaciones infinitas
- Desventajas
  - es difícil calcular el costo de ejecución (pues depende del contexto en el que se usa...)

Resumen

- Reducción. Normalización.
- Bottom. Funciones parciales y totales.
- Funciones estrictas y no estrictas.
- Órdenes de reducción.
- *Lazy evaluation*.