Currificación y órdenes de evaluación

Taller de Álgebra I

Segundo cuatrimestre de 2013

Currificación

- Consideremos las siguientes funciones:
- ```
  promedioA ::  (Float, Float) -> Float
  promedioA (x,y) = (x+y)/2
  ```
- ```
  promedioB ::  Float -> Float -> Float
  promedioB x y = (x+y)/2
  ```

Currificación

- Consideremos las siguientes funciones:
- ```
  promedioA ::  (Float, Float) -> Float
  promedioA (x,y) = (x+y)/2
  ```
- ```
  promedioB ::  Float -> Float -> Float
  promedioB x y = (x+y)/2
  ```
- La notación utilizada en la segunda función se llama currificación.
- En primera instancia, evita el uso de varios signos de puntuación (comas y paréntesis), pero obliga a un espacio entre el nombre de la función y sus parámetros.

Currificación

- Sin embargo, la currificación tiene un sentido más profundo:

Currificación

- Sin embargo, la currificación tiene un sentido más profundo:
  1. `promedioB 3 4` $\rightsquigarrow$ `3.5 :: Float`

Currificación

- Sin embargo, la currificación tiene un sentido más profundo:
  1. `promedioB 3 4` $\rightsquigarrow$ `3.5 :: Float`
  2. `promedioB 3`

Currificación

- Sin embargo, la currificación tiene un sentido más profundo:
  1. `promedioB 3 4` $\rightsquigarrow$ `3.5 :: Float`
  2. `promedioB 3 :: Float -> Float`

Currificación

- Sin embargo, la currificación tiene un sentido más profundo:
  1. `promedioB 3 4` $\rightsquigarrow$ `3.5 :: Float`
  2. `promedioB 3 :: Float -> Float`
- Esta última expresión es una nueva función que toma un `Float` y retorna otro `Float`!

Currificación

- Sin embargo, la currificación tiene un sentido más profundo:
  1. `promedioB 3 4` $\rightsquigarrow$ `3.5 :: Float`
  2. `promedioB 3 :: Float -> Float`
- Esta última expresión es una nueva función que toma un `Float` y retorna otro `Float`!
- Por ejemplo,
- `f = promedioB 3`
  `f 4` $\rightsquigarrow$ `3.5 :: Float`

Currificación

- Por ejemplo, a partir de una función para sumar dos enteros, podemos obtener una función para incrementar en 1 a un entero:
- ```
  suma ::  Int -> Int -> Int
  suma x y = x + y
  ```
- ```
  inc ::  Int -> Int
  inc = suma 1
  ```

Alto orden

- Más aún, podemos pasar como parámetro una función!
- ```
  operar ::  Int -> (Int -> Int) -> Int
  operar x f = f (x+1)
  ```
  ```

Alto orden

- Más aún, podemos pasar como parámetro una función!
- ```
  operar ::  Int -> (Int -> Int) -> Int
  operar x f = f (x+1)
  ```
- ¿Qué hace esta función?
  ```

Alto orden

- Más aún, podemos pasar como parámetro una función!
- ```
  operar ::  Int -> (Int -> Int) -> Int
  operar x f = f (x+1)
  ```
- ¿Qué hace esta función?
  1. `operar 3 inc`
  ```

Alto orden

- Más aún, podemos pasar como parámetro una función!
- ```
  operar ::  Int -> (Int -> Int) -> Int
  operar x f = f (x+1)
  ```
- ¿Qué hace esta función?
  1. `operar 3 inc` $\rightsquigarrow$ `5`
  ```

Alto orden

- Más aún, podemos pasar como parámetro una función!
- ```
  operar ::  Int -> (Int -> Int) -> Int
  operar x f = f (x+1)
  ```
- ¿Qué hace esta función?
  1. `operar 3 inc` $\rightsquigarrow$ `5`
  2. `operar 3 (suma 4)`
  ```

Alto orden

- Más aún, podemos pasar como parámetro una función!
- ```
  operar ::  Int -> (Int -> Int) -> Int
  operar x f = f (x+1)
  ```
- ¿Qué hace esta función?
  1. `operar 3 inc` $\rightsquigarrow$ `5`
  2. `operar 3 (suma 4)` $\rightsquigarrow$ `8`
  ```

Alto orden

- Más aún, podemos pasar como parámetro una función!
- ```
  operar ::  Int -> (Int -> Int) -> Int
  operar x f = f (x+1)
  ```
- ¿Qué hace esta función?
  1. `operar 3 inc` $\rightsquigarrow$ `5`
  2. `operar 3 (suma 4)` $\rightsquigarrow$ `8`
- ¿Cómo funciona todo esto?
  ```

Reducción

- En el contexto de los lenguajes funcionales, llamamos modelo de cómputo a la especificación que define cómo se calcula el valor de una expresión.

Reducción

- En el contexto de los lenguajes funcionales, llamamos modelo de cómputo a la especificación que define cómo se calcula el valor de una expresión.
- El mecanismo de evaluación en Haskell es la reducción:
  1. Reemplazamos una subexpresión por otra.
  2. La subexpresión reemplazada es una instancia del lado izquierdo de alguna ecuación orientada del programa, y se llama redex (*reducible expression*) o radical.
  3. La subexpresión reemplazante es el lado derecho de la ecuación, instanciado de manera acorde.
  4. El resto de la expresin no cambia.

Reducción

- `suma (restar 2 (amigos Juan)) 4`
- `restar x y = x - y`

Reducción

- `suma (restar 2 (amigos Juan)) 4`
- `restar x y = x - y`
- Buscamos un redex y una asignación:
  `suma (restar 2 (amigos Juan)) 4`

redex

Reducción

- `suma (restar 2 (amigos Juan)) 4`
- `restar x y = x - y`
- Buscamos un redex y una asignación:
  `suma (restar 2 (amigos Juan)) 4`
  (redex)
  1. `x` $\leftarrow$ `2`
  2. `y` $\leftarrow$ `(amigos Juan)`

Reducción

- `suma (restar 2 (amigos Juan)) 4`
- `restar x y = x - y`
- Buscamos un redex y una asignación:
  `suma (restar 2 (amigos Juan)) 4`
  (redex)
  1. `x` $\leftarrow$ `2`
  2. `y` $\leftarrow$ `(amigos Juan)`
- Reemplazamos el redex con esa asignación:
  `suma (restar 2 (amigos Juan)) 4` $\rightsquigarrow$ `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)

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, reemplazarlo y volver a empezar.

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal?

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**
  1. `f x = f (f x)`

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**
  1. `f x = f (f x)` – ¿cuánto vale `f 3`?

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**
  1. `f x = f (f x)` – ¿cuánto vale `f 3`?
  2. `infinito = infinito + 1`

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**
  1. `f x = f (f x)` – ¿cuánto vale `f 3`?
  2. `infinito = infinito + 1` – ¿cuánto vale `infinito`?

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**
  1. `f x = f (f x)` – ¿cuánto vale `f 3`?
  2. `infinito = infinito + 1` – ¿cuánto vale `infinito`?
  3. `inverso x | x /= 0 = 1 / x`

Confluencia de estrategias de reducción

- ¿Toda expresión tiene forma normal? **No!**
  1. `f x = f (f x)` – ¿cuánto vale `f 3`?
  2. `infinito = infinito + 1` – ¿cuánto vale `infinito`?
  3. `inverso x | x /= 0 = 1 / x` – ¿cuánto vale `inverso 0`?

Confluencia de estrategias de reducción

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

Confluencia de estrategias de reducción

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

Confluencia de estrategias de reducción

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

Bottom

- Decimos que las expresiones que no tienen forma normal están indefinidas, y representamos su valor con el signo $\perp$.

Bottom

- Decimos que las expresiones que no tienen forma normal están indefinidas, y representamos su valor con el signo $\bot$.
- ¿Podemos definir en Haskell una función que siempre se indefine?

Bottom

- Decimos que las expresiones que no tienen forma normal están indefinidas, y representamos su valor con el signo $\perp$.
- ¿Podemos definir en Haskell una función que siempre se indefine?

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

Bottom

- Decimos que las expresiones que no tienen forma normal están indefinidas, y representamos su valor con el signo $\bot$.
- ¿Podemos definir en Haskell una función que siempre se indefine?

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

- Otra opción (del preludio de Haskell):

```
undefined ::  a
```

Bottom

- Decimos que las expresiones que no tienen forma normal están indefinidas, y representamos su valor con el signo $\bot$.
- ¿Podemos definir en Haskell una función que siempre se indefine?

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

- Otra opción (del preludio de Haskell):

```
undefined ::  a
```

- Cualquier intento de evaluar `bottom` o `undefined` se indefine.

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

`g 2` $\rightsquigarrow \bot$

Indefinicin

- Si pasamos una valor definido a una función, puede devolver $\perp$?
  1. Funciones parciales: a veces devuelven $\perp$.
  2. Funciones totales: nunca devuelven $\perp$.

Indefinicin

- Si pasamos una valor definido a una función, puede devolver $\perp$?
  1. Funciones parciales: a veces devuelven $\perp$.
  2. Funciones totales: nunca devuelven $\perp$.
- ¿Qué sucede si le pasamos a una función como parámetro una expresión que se indefine?
  1. Funciones estrictas: $f \perp \rightsquigarrow \perp$.
  2. Funciones no estrictas: $f \perp \rightsquigarrow$ valor.

Funciones totales vs. parciales

- Ejemplo de una función total: `suc :: Integer -> Integer`

```
suc x = x + 1
```

Funciones totales vs. parciales

- Ejemplo de una función total: `suc :: Integer -> Integer`

```
suc x = x + 1
```

- Ejemplo de una función parcial: `recip :: Float -> Float`

```
recip x | x /= 0 = 1/x
```

Funciones totales vs. parciales

- Ejemplo de una función total: `suc :: Integer -> Integer`

```
suc x = x + 1
```

- Ejemplo de una función parcial: `recip :: Float -> Float`

```
recip x | x /= 0 = 1/x
```

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

Funciones estrictas vs. no estrictas

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

Funciones estrictas vs. no estrictas

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

- ¿A qué expresión reduce `const 2 bottom`?
  ```

Funciones estrictas vs. no estrictas

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

- ¿A qué expresión reduce `const 2 bottom`?

- Depende del **diseño** del lenguaje! El secreto está en el orden de evaluación.
  ```

Órdenes de evaluación

- Orden aplicativo:
  1. Primero redexes internos.
  2. Primero los argumentos, después la función.
- Orden normal:
  1. El redex más externo para el que pueda saber qué ecuación del programa se debe aplicar.
  2. Primero la función, 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 existe.

Órdenes de evaluación

- Orden aplicativo:

```
suma (3+4) (inc (2*3))
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
⇝ suma 7 7
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
⇝ suma 7 7
⇝ 14
```

Órdenes de evaluación

- Orden aplicativo:

```
suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
⇝ suma 7 7
⇝ 14
```

- Orden normal:

```
suma (3+4) (inc (2*3))
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
⇝ suma 7 7
⇝ 14
```

- Orden normal:

```
 suma (3+4) (inc (2*3))
⇝ (3+4) + (inc 6)
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
⇝ suma 7 7
⇝ 14
```

- Orden normal:

```
 suma (3+4) (inc (2*3))
⇝ (3+4) + (inc 6)
⇝ 7 + (inc 6)
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
⇝ suma 7 7
⇝ 14
```

- Orden normal:

```
 suma (3+4) (inc (2*3))
⇝ (3+4) + (inc 6)
⇝ 7 + (inc 6)
⇝ 7 + 7
```

Órdenes de evaluación

- Orden aplicativo:

```
 suma (3+4) (inc (2*3))
⇝ suma (3+4) (inc 6)
⇝ suma 7 (inc 6)
⇝ suma 7 7
⇝ 14
```

- Orden normal:

```
 suma (3+4) (inc (2*3))
⇝ (3+4) + (inc 6)
⇝ 7 + (inc 6)
⇝ 7 + 7
⇝ 14
```

Evaluación *lazy*

Evaluación *lazy*

Evaluación *lazy*

- Se trata del orden de evaluación que utiliza Haskell.
- Consiste en aplicar el orden normal (primero redexes externos), pero guardando una lista de las subexpresiones ya evaluadas, para evitar que una misma expresión se evalúe dos veces.

Ejercicios

1. Dar dos funciones f y g en Haskell tales que $(f \circ g)$ esté definida siempre, pero $(g \circ f)$ esté siempre indefinida.
2. Consideremos la función $f : \mathbb{N} \to \mathbb{R}$ definida por $f(n) = \sqrt{n}$ si $n > 0$ y $f(0) = 1$. Programarla en Haskell y verificar que $f(n)$ se indefine si $n < 0$.
3. Programar una función f que reciba como parámetros dos funciones g y h, de modo tal que $f = (g \circ h)$. ¿Qué signatura tiene f?
4. Programar una función tal que, dado un número a, devuelva una función que a su vez tome como parámetro un número b y esta segunda función retorne a/b.
5. Dado que programamos en Haskell, ¿era necesario especificar el ejercicio anterior de manera tan complicada?