Mecanismo de Reducción

Taller de Álgebra I

Segundo cuatrimestre 2019

Reducción

¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell si escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

Reducción

¿Cómo ejecuta Haskell?

¿Qué sucede en Haskell si escribo una expresión? ¿Cómo se transforma esa expresión en un resultado?

- Dado el siguiente programa:

```
resta :: Integer -> Integer -> Integer
resta x y = x - y

suma :: Integer -> Integer -> Integer
suma x y = x + y

negar :: Integer -> Integer
negar x = -x
```

- ¿Qué sucede al evaluar la expresión `suma (resta 2 (negar 42)) 4`

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:

Reducción

```
suma (resta 2 (negar 42)) 4
```

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.

Reducción

`suma (resta 2 (negar 42)) 4`

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: $\texttt{suma}\ (\underbrace{\texttt{resta 2 (negar 42)}}_{\text{redex}})\ \texttt{4}$

Reducción

`suma (resta 2 (negar 42)) 4`

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: $\texttt{suma}\ (\underbrace{\texttt{resta 2 (negar 42)}}_{\text{redex}})\ \texttt{4}$
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - $\texttt{x} \leftarrow \texttt{2}$
     - $\texttt{y} \leftarrow \texttt{(negar 42)}$

Reducción

`suma (resta 2 (negar 42)) 4`

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: $\texttt{suma (}\underbrace{\texttt{resta 2 (negar 42)}}_{\text{redex}}\texttt{) 4}$
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - `x` ← `2`
     - `y` ← `(negar 42)`
  4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
     - `suma (resta 2 (negar 42)) 4` ⇝ `suma (2 - (negar 42)) 4`

Reducción

`suma (resta 2 (negar 42)) 4`

- El mecanismo de evaluación en un Lenguaje Funcional es la reducción:
  1. Elegimos una subexpresión. Vamos a reemplazar esta subexpresión por otra.
  2. La subexpresión a reemplazar es alguna instancia del lado izquierdo de alguna ecuación orientada del programa, y se la llama radical o redex (*reducible expression*).
     - Buscamos un redex: $\texttt{suma}\ (\underbrace{\texttt{resta 2 (negar 42)}}_{\text{redex}})\ \texttt{4}$
  3. La reemplazaremos por el lado derecho de esa misma ecuación, ligando los parámetros.
     - `resta x y = x - y`
     - $\texttt{x} \leftarrow \texttt{2}$
     - $\texttt{y} \leftarrow \texttt{(negar 42)}$
  4. Reemplazamos el redex con lo anterior y el resto de la expresión no cambia.
     - $\texttt{suma (resta 2 (negar 42)) 4} \rightsquigarrow \texttt{suma (2 - (negar 42)) 4}$
  5. Si la expresión resultante aún puede reducirse, volvemos al paso 1.

Órdenes de evaluación en Haskell

Orden normal o lazy ("perezoso"):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

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

Órdenes de evaluación en Haskell

Orden normal o lazy ("perezoso"):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
   suma (3+4) (suc (2*3))
⇝ (3+4) + (suc (2*3))
```

Órdenes de evaluación en Haskell

Orden normal o lazy ("perezoso"):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
   suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
⇝  7 + (suc (2*3))
```

Orden normal o lazy ("perezoso"):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
   suma (3+4) (suc (2*3))
⇝ (3+4) + (suc (2*3))
⇝ 7 + (suc (2*3))
⇝ 7 + ((2*3) + 1)
```

Órdenes de evaluación en Haskell

Orden normal o lazy ("perezoso"):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
   suma (3+4) (suc (2*3))
⇝  (3+4) + (suc (2*3))
⇝  7 + (suc (2*3))
⇝  7 + ((2*3) + 1)
⇝  7 + (6 + 1)
```

Órdenes de evaluación en Haskell

Orden normal o lazy ("perezoso"):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
   suma (3+4) (suc (2*3))
⇝ (3+4) + (suc (2*3))
⇝ 7 + (suc (2*3))
⇝ 7 + ((2*3) + 1)
⇝ 7 + (6 + 1)
⇝ 7 + 7
```

Órdenes de evaluación en Haskell

Orden normal o lazy (“perezoso”):

Reduce el redex más externo para el cual se sepa qué ecuación del programa se debe aplicar; es decir que primero evalúa la función y después los argumentos (si se necesitan).

Ejemplo:

```
   suma (3+4) (suc (2*3))
⇝ (3+4) + (suc (2*3))
⇝ 7 + (suc (2*3))
⇝ 7 + ((2*3) + 1)
⇝ 7 + (6 + 1)
⇝ 7 + 7
⇝ 14
```

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.

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

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.
    ```
    suc ::  Integer -> Integer
    suc x = x + 1
    ```
  - Funciones parciales: hay argumentos para los cuales se indefinen.

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.
    ```
    suc :: Integer -> Integer
    suc x = x + 1
    ```
  - Funciones parciales: hay argumentos para los cuales se indefinen.
    ```
    inv :: Float -> Float
    inv x | x /= 0 = 1/x
    ```

Indefinición

- Las expresiones para las cuales Haskell no encuentra un resultado se dicen que están indefinidas ($\perp$).
- ¿Cómo podemos clasificar las funciones?
  - Funciones totales: nunca se indefinen.
    ```
    suc :: Integer -> Integer
    suc x = x + 1
    ```
  - Funciones parciales: hay argumentos para los cuales se indefinen.
    ```
    inv :: Float -> Float
    inv x | x /= 0 = 1/x
    ```

Ejercicio: reducir las siguientes expresiones

- `(inv 1 == 0) && (inv 0 == 1)`
- `(inv 1 == 1) && (inv 0 == 1)`
- `(inv 0 == 1) && (inv 1 == 1)`

Reutilizando código en Haskell

Reutilizando código en Haskell

- Para poder reutilizar código necesitamos que nuestros archivos donde definimos las funciones se constituyan como *módulos*.

Reutilizando código en Haskell

- Para poder reutilizar código necesitamos que nuestros archivos donde definimos las funciones se constituyan como *módulos*.
- Un *módulo* en Haskell es una colección de definiciones de funciones, (y eventualmente también de tipos y/o clases de tipos). Todas las funciones, tipos y clases de tipos con las que trabajamos hasta ahora son parte del módulo `Prelude`, que es importado por defecto.

Reutilizando código en Haskell

- Para poder reutilizar código necesitamos que nuestros archivos donde definimos las funciones se constituyan como *módulos*.
- Un *módulo* en Haskell es una colección de definiciones de funciones, (y eventualmente también de tipos y/o clases de tipos). Todas las funciones, tipos y clases de tipos con las que trabajamos hasta ahora son parte del módulo `Prelude`, que es importado por defecto.
- Para definir un nuevo modulo debemos crear el archivo con el mismo nombre que el módulo más la extensión ".hs"

Reutilizando código en Haskell

- Para poder reutilizar código necesitamos que nuestros archivos donde definimos las funciones se constituyan como *módulos*.
- Un *módulo* en Haskell es una colección de definiciones de funciones, (y eventualmente también de tipos y/o clases de tipos). Todas las funciones, tipos y clases de tipos con las que trabajamos hasta ahora son parte del módulo `Prelude`, que es importado por defecto.
- Para definir un nuevo modulo debemos crear el archivo con el mismo nombre que el módulo más la extensión ".hs"
- En el archivo aparece primero la palabra reservada `module` antes del nombre (debe empezar con mayúscula), despues las funciones que queremos exportar y luego `where`

```haskell
module FuncionesSimples
(suma, doble)
where

suma :: Num a => a -> a -> a
suma x y =  x + y

doble :: Num a => a -> a
doble x = 2 * x

triple :: Num a => a -> a
triple x = 3 * x
```

Usando funciones anteriores

Ahora, para importar las funciones que exporta el módulo `FuncionesSimples` y usarlas, en nuestro archivo nuevo escribimos:

```
module FuncionesComplejas
where
import FuncionesSimples

cuadruple :: Num a => a -> a
cuadruple x = doble (doble x)

sumaTupla :: Num a => (a,a) -> a
sumaTupla t = suma (fst t) (snd t)
```

- Las funciones `doble` y `suma` usadas en este nuevo módulo son las definidas en `FuncionesSimples.hs`.
- Si no especificamos cuáles funciones exportamos en un módulo, se exporta todo por defecto.

En GHCi

Al cargarlo en GHCi podemos ahora usar TODAS las funciones que definimos en FuncionesComplejas.hs más las que importamos de FuncionesSimples.hs:

```
Prelude> :l FuncionesComplejas
[1 of 2] Compiling FuncionesSimples ( FuncionesSimples.hs,
    interpreted )
[2 of 2] Compiling FuncionesComplejas ( FuncionesComplejas.hs,
    interpreted )
Ok, modules loaded: FuncionesComplejas, FuncionesSimples.
*FuncionesComplejas> doble 4
8
*FuncionesComplejas> sumaTupla (2,3)
5
```

Ejercicios de números enteros

Dar el tipo e implementar las siguientes funciones:

1. `unidades`: dado un entero, devuelve el dígito de las unidades del número (el dígito menos significativo).
2. `sumaUnidades3`: dados 3 enteros, devuelve la suma de los dígitos de las unidades de los 3 números.
3. `todosImpares`: dados 3 números enteros determina si son todos impares.
4. `alMenosUnImpar`: dados 3 números enteros determina si al menos uno de ellos es impar.
5. `alMenosDosImpares`: dados 3 números enteros determina si al menos dos de ellos son impares.
6. `alMenosDosPares`: dados 3 números enteros determina si al menos dos de ellos son pares.

Ejercicios de relaciones

Dar el tipo e implementar las siguientes funciones:

7. Dados dos enteros a, b implementar tres funciones: $r1$, $r2$ y $r3$ que determinen si $a \sim b$ para cada uno de los siguientes casos:
   1. $a \sim b$ sii tienen la misma paridad
   2. $a \sim b$ sii $2a + 3b$ es divisible por 5
   3. $a \sim b$ sii los dígitos de las unidades de a, b y ab son todos distintos
8. Se define en $\mathbb{R}$ la relación de equivalencia asociada a la partición
$$\mathbb{R} = (-\infty, 3) \cup [3, +\infty)$$
   Implementar una función que dados dos números $x, y \in \mathbb{R}$ determine si $x \sim y$.
9. Repetir el ejercicio anterior para la partición
$$\mathbb{R} = (-\infty, 3) \cup [3, 7) \cup [7, +\infty).$$
10. Dados (a, b) y (p, q) en $\mathbb{Z} \times \mathbb{Z} - \{(0, 0)\}$, implementar una función que determine si $(a, b) \sim (p, q)$, considerando que $(a, b) \sim (p, q)$ sii existe $k \in \mathbb{R}$ tal que $(a, b) = k(p, q)$