

Introducción a Haskell

Programación Avanzada

UNRC

Pablo Castro

El Lenguaje Haskell

Haskell fue introducido en 1987 con el objetivo de introducir un lenguaje funcional moderno

- Hugs, es un interprete de Haskell muy usado, se puede obtener en www.haskell.org/hugs
- Glasgow Haskell, es un compilador para Haskell se puede obtener en <http://www.haskell.org/ghc/>.

ghc es más completo y también tiene un interprete ghci

Tipos Básicos de Haskell

Haskell tiene un conjunto rico de tipos básicos de datos

- **Booleanos**, tipos de booleanos con las operaciones lógicas,
- **Int**: Enteros de precisión fija,
- **Char**: 'a', 'b', 'c', etc
- **Integer**: Enteros de precisión variable,
- **Float**: Números reales.

Escribimos:

$E :: T$

Cuando E es de tipo T

El Tipo Bool

El tipo `Bool` tiene dos valores `true` y `false`, y las siguientes operaciones:

- `&& :: Bool -> Bool -> Bool`
- `|| :: Bool -> Bool -> Bool`
- `not :: Bool -> Bool`

Cualquier función: `f :: A -> Bool` es llamado predicado, y puede utilizarse con estas operaciones.

`== :: A -> A -> Bool`

La igualdad es la más conocida

El Tipo Char

El tipo `char` contiene los valores `'a'`, `'b'`, `'c'`, ... etc, y las siguientes funciones:

- `ord :: Char -> Int` convierte caracteres a enteros.
- `chr :: Int -> Char` convierte enteros a caracteres.

Los Strings se modelan como una lista de chars.

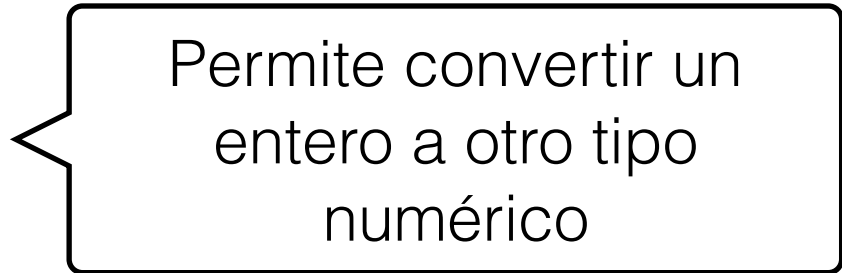
Sistema de Números

Haskell tiene varios tipos numéricos:

- `Int`, enteros con precisión limitada $[-2^{29}, 2^{29})$.
- `Integer`, enteros con precision variable,
- `Float`, reales 3.14159
- `Double`, reales con doble precisión.

Tenemos funciones de conversión entre ellos, por ejemplo:

`fromInteger`



Permite convertir un entero a otro tipo numérico

Tuplas

Usando los tipos básicos podemos construir tuplas y listas.
Dados tipos A y B :

(A, B)

El tipo de pares de A y B

Por ejemplo: $(\text{True}, 1) :: (\text{Bool}, \text{Int})$

Operaciones:

$\text{fst} :: (a, b) \rightarrow a$

Devuelve el primer
componente

$\text{snd} :: (a, b) \rightarrow b$

Devuelve el segundo
componente

Listas

Dado un cualquier tipo a

$[a]$

Es el tipo a las listas de
tipo a

Donde:

- $[]$ es la lista vacía
- $x : xs$ es la lista con x a la cabeza y luego xs a la cola

Todos los elementos de la lista son del mismo tipo.

Funciones sobre Listas

Algunas funciones útiles sobre listas:

- `head :: [a] -> a`, devuelve la cabeza de la lista.
- `last :: [a] -> a`, devuelve el último elemento.
- `tail :: [a] -> [a]`, devuelve la cola de la lista.
- `++ :: [a] -> [a] -> [a]`, concatena dos listas.

Funciones

Dados dos tipos a y b :

$a \rightarrow b$

Es el tipo de las
funciones de a en b

Por ejemplo:

$\text{Not} : \text{Bool} \rightarrow \text{Bool}$

Función de Bool en
 Bool

Las funciones de alto orden son aquellas que toman como parámetros funciones o retornan funciones

$. : (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

Definir la composición de
funciones

Definición de Funciones

Una función se puede definir por casos:

```
sign : Int->Int
sign x | x>=0 = 1
      | x<0  = -1
```

También por pattern matching:

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

Toma los primeros n
elementos de una lista

Se pueden usar los constructores de los tipos

Funciones Estrictas

La expresión:

`undefined :: a`

representa un “error”
de tipo `a`

Una función $f :: a_0 \rightarrow a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n$ se dice estricta en el parámetro i si:

$$f\ x_0\ x_1\ \dots\ x_{i-1}\ \text{undef}\ x_{i+1}\ \dots\ x_n = \text{undef}$$

Por ejemplo:

`True && b = b`
`False && _ = False`

Es estricta en su primer
parámetro

Polimorfismo

Consideremos la siguiente función:

```
drop :: Int -> [a] -> [a]
```

```
drop n [] = []
```

```
drop 0 xs = xs
```

```
drop n (x:xs) = drop (n-1) xs
```

tira los primeros n
elementos de la lista

En `[a]`, `a` puede ser cualquier tipo, se llama variable de tipos, y `drop` se dice que es polimorfica.

Patrones de Recursión

Consideremos la función:

```
sum [] = 0
sum (x:xs) = x+sum xs
```

Si ejecutamos esta función para $[1,..,n]$ obtenemos:

$1 + (2 + (3 + (... + n))$

Asocia a la derecha

De la misma forma podríamos reemplazar el $+$ por $*$

$1 * (2 * (3 * (... * n))$

Patrones de Recursión

Podemos generalizar esto para una operación @

Para $[X_0, X_1, \dots, X_n]$ calculamos

$$X_0 @ (X_1 @ (\dots @ X_n))$$

Esto se define por medio de la función `foldr`:

z es el elemento que
devolvemos en el
caso []

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Ejemplo: `foldr @ e [X0, X1, X2] = X0 @ (X1 @ (X2 @ e))`

Ejemplos

- `sum xs = foldr (+) 0 xs`
- `mult xs = foldr (*) 1 xs`
- `conj xs = foldr (&) true xs`
- `disj xs = foldr (||) false xs`

Todas estas funciones son asociativas, es decir, no importa como asociemos, pero:

`foldr (-) 0 xs`



No es asociativa

Foldl, Asociando a la Izquierda

También podríamos haber asociado a la izquierda:

```
sum n [] = n
```

```
sum n (x:xs) = sum (n+x) xs
```

Si ejecutamos `sum 0 [1,2,3]` obtenemos:

`((0+1)+2)+3`

Es decir, asocia a la izquierda

Este patrón se computa con `foldl`:

```
foldl :: (a->b->a)->a->[b]->a
```

```
foldl f z [] = z
```

```
foldl f z x:xs = foldl f (f z x) xs
```

Foldl

Por ejemplo:

$$\text{foldl } (+) \ 0 \ [X_0, X_1, X_2] = ((0 + X_0) + X_1) + X_2$$

Foldl vs Foldr:

- Evalúan igual si \mathfrak{f} es asociativa y su dos parámetros tienen el mismo tipo sobre listas finitas
- Si la función \mathfrak{f} no es **estricta** en su segundo parámetro, entonces foldr puede funcionar para listas infinitas

Ejemplo:

```
foldr (&&) false [false, false, ...] = false
foldl (&&) false [false, false, ...] stack overflow
```

Map

Map, dada una función y una lista aplica esa función a cada elemento de la lista

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : (map f xs)
```

Es decir:

$$\text{map } f \ [X_0, X_1, X_2, \dots, X_n] = [f \ X_0, f \ X_1, \dots, f \ X_n]$$

Por ejemplo:

$$\text{map } (*2) \ [1, 2, 3, 4, 5] = [2, 4, 6, 8, 10]$$

Filter

Dada una lista filtra los elementos de la lista usando un predicado:

```
filter :: (a->Bool) -> [a] -> [a]
filter f [] = []
filter f (x:xs)
    | f x = x:(filter f xs)
    | Otherwise = filter f xs
```

Por ejemplo:

```
filter (isEven) [1,2,3,4,5] = [2,4]
```

Puede funcionar para listas infinitas

Listas por Comprensión

En conjuntos podemos hacer:

$$\{2 * x \mid x \in \{0, 1, 2, 3, 4\}\}$$

En Haskell tenemos listas por comprensión, por ejemplo:

$[2 * x \mid x \leftarrow [0, 1, 2, 3, 4]]$



Generador

Retorna la lista:

$[0, 2, 4, 6, 8]$

Listas por Comprensión

Podemos tener muchos generadores:

```
[ (x, y) | x <- [0, 1, 2], y <- [2, 3, 4] ]
```

Devuelve:

```
[ (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 2), (2, 3), (2, 4) ]
```

Podemos utilizar guardas para filtrar elementos:

```
[ x | x <- [0..], isEven x ]
```

Devuelve:

```
[ 0, 2, 4, 6, 8, 10, ... ]
```

Declarando Tipos Nuevos

Podemos definir nuevos tipos con el constructor `type`:

```
type String = [Char]
```

Es la definición de String
en Haskell

También por ejemplo:

```
type Pos = (Int, Int)
```

Posiciones en un tablero

Y podemos usar este tipo nuevo:

```
type Board = [Pos]
```

Un tablero es una lista de
posiciones

Tipos Nuevos

Podemos definir tipos con nuevos valores:

```
data Bool = False | True
```

Y se pueden definir tipos inductivos

```
data Nat = Zero | Succ Nat
```

Definen los naturales por medio de dos constructores:

```
Zero :: Nat          y          Succ :: Nat -> Nat
```

Cuyos valores son:

```
Zero, Succ Zero, Succ (Succ Zero), ...
```


Árboles Binarios

Los árboles binarios son un tipo de datos muy útil:

```
Data Tree a = Nil | Node (Tree a) a (Tree a)
```



Hijo izquierdo



Hijo derecho

Un ejemplo de función sobre árboles:

```
size :: Tree a -> Int
```

```
size Nil = 0
```

```
size (Node hi r hd) = 1+size hi+size hd
```

Clases en Haskell

Una operación se dice sobrecargada si puede utilizarse para varios tipos.

```
elem x [] = False
elem x (y:ys) | x==y = True
               | otherwise = elem x ys
```

Solo está bien definida si
el tipo tiene la igualdad
definida

Una **clase** en Haskell define una colección de tipos que tienen una operación en común

Clases en Haskell

La clase que tiene la igualdad se define:

Todos los tipos que **instancien** esta clase deben tener la igualdad definida

```
Class Eq a where  
    (==) :: a -> a -> Bool
```

Por ejemplo, para decir que Nat pertenecen a Eq:

```
instance (Eq a) => Eq Nat where  
    Zero == Zero = True  
    Zero == Succ n = False  
    Succ n == Succ m = n == m
```

Ejemplo Clases

En el ejemplo anterior el perfil de la función sería:

```
elem :: Eq a => a -> [a] -> Bool
```

Dado que el tipo `a`
pertenece a la clase `Eq`

La función tiene este
perfil

Otras clases importantes:

- `Show`: provee una función para mostrar por pantalla,
- `Ord`: provee relaciones de orden,

Quicksort

Quicksort en Pascal:

```
Procedure QSort(numbers : Array of Integer; left : Integer; right : Integer);
Var
    pivot, l_ptr, r_ptr : Integer;

Begin
    l_ptr := left;
    r_ptr := right;
    pivot := numbers[left];

    While (left < right) do
        Begin
            While ((numbers[right] >= pivot) AND (left < right)) do
                right := right - 1;

            If (left <> right) Then
                Begin
                    numbers[left] := numbers[right];
                    left := left + 1;
                End;

            While ((numbers[left] <= pivot) AND (left < right)) do
                left := left + 1;

            If (left <> right) Then
                Begin
                    numbers[right] := numbers[left];
                    right := right - 1;
                End;
        End;

    numbers[left] := pivot;
    pivot := left;
    left := l_ptr;
    right := r_ptr;

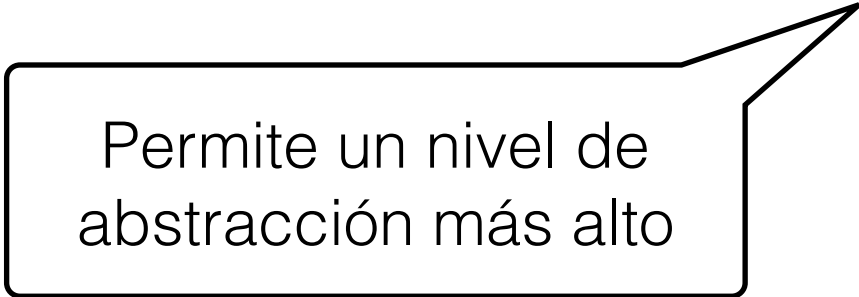
    If (left < pivot) Then
        QSort(numbers, left, pivot-1);

    If (right > pivot) Then
        QSort(numbers, pivot+1, right);

End;
```

QuickSort en Haskell

```
qsort [] = []  
qsort (x:xs) = qsort left ++ [x] ++ qsort  
right  
    where  
    left = [a | a <- xs, a <= x]  
    right = [b | b <- xs, b > x]
```



Permite un nivel de
abstracción más alto

Cambiando el Orden de Evaluación

Muchas veces cambiar el orden de ejecución puede mejorar la eficiencia de las funciones

```
sum n [] = n
```

```
sum n (x:xs) = sum (n+x) xs
```

Acumula en el
parámetro

Podemos mejorar el uso del espacio mediante \$!

```
sum n [] = n
```

```
sum n (x:xs) = (sum $! (n+x)) xs
```

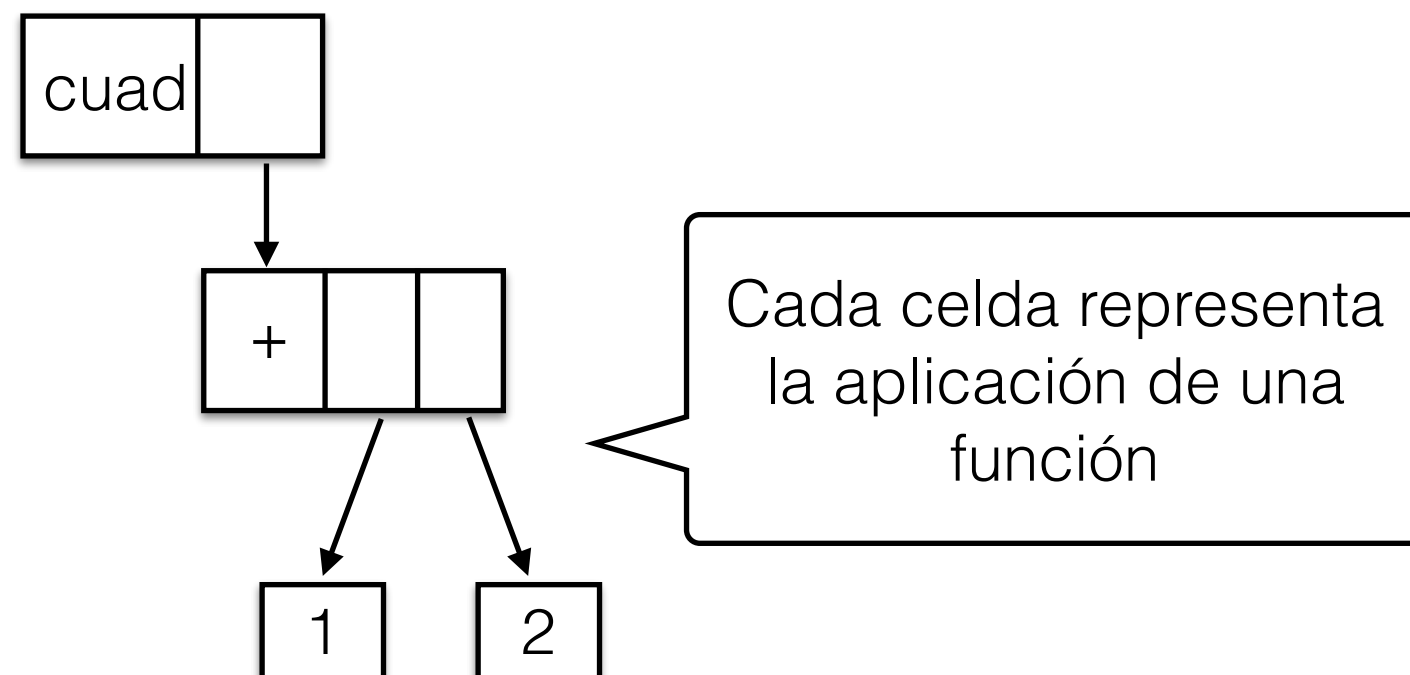
Este parametro se evalúa
antes de la función

Evaluación en Haskell

Haskell usa **evaluación lazy**.

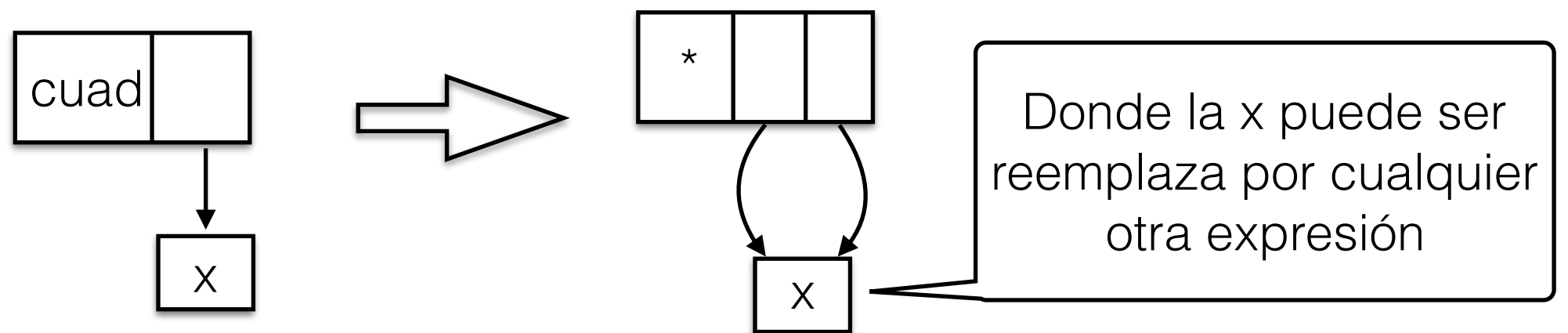
Cada expresión es representa en memoria con punteros.

Por ejemplo: `cuad (1+2)` se representa:



Evaluación en Haskell

Las definiciones son representadas como reglas:



Ejemplo de evaluación:

