

Introducción

Juan Manuel Rabasedas

23/09/2011

Problemas del Software

- Sistemas de software gigantes, cores múltiples, sistemas distribuidos, etc, presentan enormes desafíos.
- ¿Cómo lidiamos con ellos?
- ¿Cómo reducimos el costo y tiempo de desarrollo?
- ¿Cómo incrementamos nuestra confianza en nuestros programas?

Lenguajes de Programación

- Una forma de encarar estos problemas es mediante el diseño de nuevos lenguajes de programación.
- Los programas deben ser escritos en forma clara, concisa, y a un alto nivel de abstracción.
- Los componentes de software desarrollados deben poder ser reusados.
- La semántica debe ser clara para facilitar la verificación formal.
- El lenguaje debe permitir el prototipado rápido y proveer herramientas potentes.

Programación funcional

- La programación funcional cumple con todos estos requerimientos.
- Es cada vez más usada en la industria.
- Su simpleza y versatilidad lo hacen ideal para aprender conceptos de:
 - programación,
 - lenguajes de programación,
 - computabilidad,
 - semántica,
 - verificación, derivación, testing.

¿Qué es la programación funcional

Hay diferentes opiniones:

- Es un estilo de programación en el cual el método básico de computar es aplicar funciones a argumentos.
- Un lenguaje de programación funcional es un lenguaje que permite y alienta el uso de un estilo funcional.

- Programas como funciones $f : \text{Input} \rightarrow \text{Output}$
 - sin variables, sin estado, sin efectos laterales
 - todas sus dependencias explicitas
 - la salida Output solo depende de la entrada Input, no del entorno
- Abstracciones
 - Abstracción de datos
 - Abstracción de funciones
 - Modularización y decomposición
- Especificación y Verificación
 - Tipado
 - Clara semantica denotacional y operacional.

- Programas Concisos
- Sistemas de Tipos Poderoso
- Funciones Recursivas
- Listas por Comprensión
- Fácilidad para probar propiedades de programas.
- Funciones de Alto Orden
- Evaluación Perezosa
- Facilidad para definir DSLs
- Efectos Monádicos

Preludio

- Muchas funciones de uso común están definidas en el Preludio (Prelude.hs)
- Además de las operaciones aritméticas usuales se definen muchas funciones que operan sobre listas: *head*, *tail*, *(!!)*, *take*, *drop*, *length*, *sum*, *product*, *(++)*, *reverse*.

GHC

- Usaremos GHC, un compilador (ghc) e intérprete (ghci) de Haskell 98 y que también soporta varias extensiones del mismo.
- <http://www.haskell.org/ghc>

Aplicación de funciones

- En Haskell la aplicación se denota con un espacio y asocia a la izquierda.

Matemática	Haskell
$f(x)$	$f\ x$
$f(x, x)$	$f\ x\ y$
$f(g(x))$	$f\ (g\ x)$
$f(x, g(y))$	$f\ x\ (g\ y)$
$f(x)g(y)$	$f\ x * g\ y$

- La aplicación tiene mayor precedencia que cualquier otro operador: $f\ x + y = (f\ x) + y$

Nombres y comentarios

- Las funciones y sus argumentos deben empezar con minúscula, y pueden ser seguidos por cero o más letras (mayúsculas o minúsculas), dígitos, guiones bajos, y apóstrofes.
- Las palabras reservadas son: **case class data default deriving do else if import in infix infixl infixr instance let module newtype of then type where**
- Los comentarios se escriben:

```
-- comentario que finaliza junto con la l\'ineal'  
{- bloque de comentarios para comentarios m\'as extens
```

Offside Rule

- En una serie de definiciones, cada definición debe empezar en la misma columna.

$$a = b + c$$

where

$$b = 1$$
$$c = 2$$
$$d = a + 2$$

- Gracias a esta regla no hace falta un sintaxis explícita para agrupar definiciones.

Operadores infijos

- Los operadores infijos son funciones como cualquier otra.
- Una función se puede hacer infija con backquotes:

$$10 \text{ 'div' } 4 = \text{div } 10 \ 4$$

- Se pueden definir operadores infijos usando alguno de los símbolos disponibles:

$$a ** b = (a * b) + (a + 1) * (b - 1)$$

- La asociatividad y precedencia se indica usando **infixr** (asociativad der.), **infixl** (asociatividad izq.), o **infix** (si los paréntesis deben ser obligatorios)

$$\text{infixr } 6 \ (**)$$

tipos

- Un tipo es un nombre para una colección de valores
 - Ej: Bool contiene los valores True y False.
 - Escribimos `True :: Bool` y `False :: Bool`.
- En general, si una expresión e tiene tipo t escribimos

$$e :: t$$

- **En Haskell, toda expresión válida tiene un tipo**
- El tipo de cada expresión es calculado previo a su evaluación mediante la inferencia de tipos.
- Si no es posible encontrar un tipo (por ejemplo `(True + 4)`) el compilador/intérprete protestará con un error de tipo.

Tipos básicos

Algunos tipos básicos de haskell son:

- Bool , booleanos
- Char , caracteres
- Int, enteros de precisión fija.
- Integer, enteros de precisión arbitraria.
- Float, números de punto flotante de precisión simple.

Listas

- Una lista es una secuencia de valores del mismo tipo
 - `[True, True, False, True] :: [Bool]`
 - `['h', 'o', 'l', 'a', ''] :: [Char]`
- en general, `[t]` es una lista con elementos de tipo `t`
- `t`, puede ser cualquier tipo válido.
- No hay restricción con respecto a la longitud de las listas.

Tuplas

- Una secuencia es una secuencia finita de valores de tipos (posiblemente) distintos.
 - $(\text{True}, \text{True}) :: (\text{Bool}, \text{Bool})$
 - $(\text{True}, 'a', 'b') :: (\text{Bool}, \text{Char}, \text{Char})$
- En general, $(t_1, t_2, \dots, t_n)$ es el tipo de una n -tupla cuyas componente i tiene tipo t_i , para i en $1 \dots n$.
- A diferencia de las listas, las tuplas tienen explicitado en su tipo la cantidad de elementos que almacenan.
- Los tipos de las tuplas no tiene restricciones.
 $('a'; (\text{True}; 'c')) :: (\text{Char}; (\text{Bool}; \text{Char}))$

Funciones

- Una función mapea valores de un tipo en valores de otro:
 - $\neg :: \text{Bool} \rightarrow \text{Bool}$
 - $\text{isDigit} :: \text{Char} \rightarrow \text{Bool}$
- En general, Un tipo $t_1 \rightarrow t_2$ mapea valores de t_1 en valores de t_2 .
- Se pueden escribir funciones con múltiples argumentos o resultados usando tuplas y listas.

$\text{add} :: (\text{Int}; \text{Int}) \rightarrow \text{Int}$

$\text{add } (x; y) = x + y$

$\text{deceroa} :: \text{Int} \rightarrow [\text{Int}]$

$\text{deceroa } n = [0..n]$

Currificación y aplicación parcial

- Otra manera de tomar múltiples argumentos es devolver una función como resultado

$$add' :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$$
$$add' \ x \ y = x + y$$

- A diferencia de *add*, *add'* toma los argumentos de a uno por vez. Se dice que *add'* está currificada.

Currificación y aplicación parcial

- Otra manera de tomar múltiples argumentos es devolver una función como resultado

$$\begin{aligned} add' &:: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\ add' \ x \ y &= x + y \end{aligned}$$

- A diferencia de add , add' toma los argumentos de a uno por vez. Se dice que add' está currificada.
- La ventaja de la versión currificada es que permite la aplicación parcial:

$$\begin{aligned} suma3 &:: \text{Int} \rightarrow \text{Int} \\ suma3 &= add' \ 3 \end{aligned}$$

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$$
$$mult\ x\ y\ z = x * y * z$$

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: Int \rightarrow (Int \rightarrow (Int \rightarrow Int))$$
$$mult\ x\ y\ z = x * y * z$$

- Para evitar escribir muchos paréntesis, por convención asumimos que el constructor de tipos $\rightarrow$ asocia a la derecha.

$$mult :: Int \rightarrow Int \rightarrow Int \rightarrow Int$$

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: Int \rightarrow (Int \rightarrow (Int \rightarrow Int))$$
$$mult\ x\ y\ z = x * y * z$$

- Para evitar escribir muchos paréntesis, por convención asumimos que el constructor de tipos $\rightarrow$ asocia a la derecha.

$$mult :: Int \rightarrow Int \rightarrow Int \rightarrow Int$$

- Notar que esta convención es consistente con la aplicación asociando a la izquierda.

Currificación

- Si queremos expresar una función que tome 3 argumentos devolvemos una función que devuelve una función:

$$mult :: Int \rightarrow (Int \rightarrow (Int \rightarrow Int))$$
$$mult\ x\ y\ z = x * y * z$$

- Para evitar escribir muchos paréntesis, por convención asumimos que el constructor de tipos $\rightarrow$ asocia a la derecha.
 $mult :: Int \rightarrow Int \rightarrow Int \rightarrow Int$
- Notar que esta convención es consistente con la aplicación asociando a la izquierda.
- En Haskell por omisión todas las funciones están currificadas.

Nombres de los tipos

- A excepción de listas, tuplas y funciones, los nombres de los tipos concretos comienzan con mayúsculas.

Nombres de los tipos

- A excepción de listas, tuplas y funciones, los nombres de los tipos concretos comienzan con mayúsculas.
- El espacio de nombres de los tipos está completamente separado del espacio de nombres de las expresiones.

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow \text{Int}$
Para cualquier tipo a la función $length$ es la misma.

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow \text{Int}$
Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow \text{Int}$
Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.
- Las variables de tipo pueden ser instanciadas a otros tipos

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow Int$
Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.
- Las variables de tipo pueden ser instanciadas a otros tipos

$length [False; True] \leftarrow a = Bool$

$length ['a'; 'b'] \leftarrow a = Char$

Polimorfismo

- Una función es polimórfica si su tipo contiene variables de tipo. ej: $length :: [a] \rightarrow Int$
Para cualquier tipo a la función $length$ es la misma.
- Las variables de tipo se escriben con minúscula.
- Las variables de tipo pueden ser instanciadas a otros tipos

$length [False; True] \leftarrow a = Bool$

$length ['a'; 'b'] \leftarrow a = Char$

- A veces se llama polimorfismo paramétrico a este tipo de polimorfismo.

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas.

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas.
- Esta sobrecarga se realiza mediante clases de tipo.

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas.
- Esta sobrecarga se realiza mediante clases de tipo.
- El operador $(+)$ tiene tipo:

$$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas.
- Esta sobrecarga se realiza mediante clases de tipo.
- El operador $(+)$ tiene tipo:

$$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

- $(+)$ está definido para cualquier tipo que sea una instancia de la clase Num.

Sobrecarga de funciones

- ¿Cuál es el tipo de $3 + 2$?
- En Haskell, los números y las operaciones aritméticas están sobrecargadas.
- Esta sobrecarga se realiza mediante clases de tipo.
- El operador $(+)$ tiene tipo:

$$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$

- $(+)$ está definido para cualquier tipo que sea una instancia de la clase Num.
- A diferencia del polimorfismo paramétrico, hay una definición distinta de $(+)$ para cada instancia.

Clases de tipos

- Haskell provee una serie de clases básicas:
- `Eq` son los tipos cuyos valores pueden ser comparados para ver si son iguales o no.

$(\equiv) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$

$(\neq) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$

- ¿Qué tipo no puede ser instancia de esta clase?

Clases de tipos

- Haskell provee una serie de clases básicas:
- `Eq` son los tipos cuyos valores pueden ser comparados para ver si son iguales o no.

$$(\equiv) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$(\neq) :: Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$$

- ¿Qué tipo no puede ser instancia de esta clase?
- `Ord` son los tipos que además de ser instancias de `Eq` poseen un orden total.

$$(<), (\leq), (>), (\geq) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$$
$$min, max :: Ord\ a \Rightarrow a \rightarrow a \rightarrow a$$

Clases de tipos

- Show son los tipos cuyos valores pueden ser convertidos en una cadena de caracteres.

show :: Show *a* $\Rightarrow$ *a* $\rightarrow$ String

- Read es la clase dual. Son los tipos que se pueden obtener de una cadena de caracteres.

read :: Read *a* $\Rightarrow$ String $\rightarrow$ *a*

- ¿A qué tipo corresponde la instancia que leerá
 - $\neg$ (*read* "False")?
 - "3"?
 - "3" :: Int?

Clases de tipos

- Num son los tipos numéricos
- Sus instancias deben ser instancias de Show y Eq e implementar:

$(+), (.), (*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$
 $\text{negate}, \text{abs}, \text{signum} :: \text{Num } a \Rightarrow a \rightarrow a$

Clases de tipos

- Num son los tipos numéricos
- Sus instancias deben ser instancias de Show y Eq e implementar:

$(+), (.), (*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$
 $\text{negate}, \text{abs}, \text{signum} :: \text{Num } a \Rightarrow a \rightarrow a$

¿Y la división?

Clases de tipos

- Num son los tipos numéricos
- Sus instancias deben ser instancias de Show y Eq e implementar:

$$(+) , (..) , (*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$
$$\text{negate}, \text{abs}, \text{signum} :: \text{Num } a \Rightarrow a \rightarrow a$$

¿Y la división?

- Integral son los tipos que son Num e implementan

$$\text{div}, \text{mod} :: \text{Integral} \Rightarrow a \rightarrow a \rightarrow a$$

Clases de tipos

- Num son los tipos numéricos
- Sus instancias deben ser instancias de Show y Eq e implementar:

$$(+), (\cdot), (*) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$$
$$\text{negate}, \text{abs}, \text{signum} :: \text{Num } a \Rightarrow a \rightarrow a$$

¿Y la división?

- Integral son los tipos que son Num e implementan

$$\text{div}, \text{mod} :: \text{Integral } a \Rightarrow a \rightarrow a \rightarrow a$$

- Fractional son los tipos que son Num e implementan

$$(/) :: \text{Fractional } a \Rightarrow a \rightarrow a \rightarrow a$$
$$\text{recip} :: \text{Fractional } a \Rightarrow a \rightarrow a$$