Llenguatges Funcionals: Haskell

Albert Rubio

Llenguatges de Programació, FIB, UPC

Primavera 2014

Sistema de tipus en Haskell

1. Tipus predefinits
2. Polimorfisme paramètric
3. Nous tipus de dades
4. Classes de tipus

Continguts

1. Tipus predefinits
2. Polimorfisme paramètric
3. Nous tipus de dades
4. Classes de tipus

Tipus predefinits

Com ja hem vist existeixen una sèrie de tipus predefinits:

- Int, Integer,...
- Char
- Bool
- Funcions: a -> ... -> a
- Llistes, Tuples

```
   5  :: Integer
  'a' :: Char
  inc :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

Tipus predefinits

Tenim altres tipus predefinits, per exemple per manegar errors:

`Maybe`

Tenim dos constructors: `Just` (que té un parametre) i `Nothing`
Tenim opercions com ara

```
isJust :: Maybe a -> Bool
isNothing :: Maybe a -> Bool
fromJust :: Maybe a -> a
fromMaybe :: a -> Maybe a -> a
```

Per exemple podem definir

```
mitjana:: [a] -> Maybe a
mitjana [] = Nothing
mitjana l  = Just (sum l `div` length l)
```

Continguts

1 Tipus predefinits

2 Polimorfisme paramètric

3 Nous tipus de dades

4 Classes de tipus

Tipus polimòrfics

Utilització de variables de tipus que poden pendre qualsevol valor.

```
mida :: [a] -> Integer
```

Les variables de tipus estan implicitament quantificades universalment. Noteu que ens permet escriure llistes de qualsevol tipus, però no llistes heterogènies.

Aquest tipus de polimorfisme s'anomena *polimorfisme paramètric*.

Altres exemples:

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

És un mecanisme molt senzill per tenir definicions genèriques.

Tipus polimòrfics

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

Per a utilitzar funcions amb tipus polimòrfics cal que hi hagi una substitució de les variables de tipus que s'adeqüi a l'aplicació que estem fent.

`map even [3,6,1]` te tipus [Bool] ja que

- la variable te tipus *a* es pot substituir per `Integer`
- la variable te tipus *b* es pot substituir per `Bool`

Tipus polimòrfics

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

Per a utilitzar funcions amb tipus polimòrfics cal que hi hagi una substitució de les variables de tipus que s'adeqüi a l'aplicació que estem fent.

`map even [3,6,1]` te tipus [Bool] ja que

- la variable te tipus *a* es pot substituir per `Integer`
- la variable te tipus *b* es pot substituir per `Bool`

Una expressió dona error de tipus si no existeix una substitució per a les varaibles de tipus. Per exemple

`map not ['b','c']` dóna error de tipus ja que

- per un banda la variable de tipus *a* hauria de ser `Bool`
- per l'altre la variable de tipus *a* hauria de ser `Char`

Continguts

1 Tipus predefinits

2 Polimorfisme paramètric

3 Nous tipus de dades

4 Classes de tipus

Creació de tipus de dades

```
data Color = Vermell | Verd | Blau | Violat
```

Alternativa

```
newtype Dolars = Dolars Int
```

```
llistaDolars :: [Dolars]
llistaDolars = [Dolars 3, Dolars 27]
```

És com data, però quan hi ha un únic constructor de tipus.

Tot newtype es pot escriure com data, però no el contrari

```
data Dolars = Dolars Int
```

Definicions amb Data

El tipus predefinit `Maybe` està definit amb un data:

```
data Maybe a = Just a | Nothing
```

Definicions amb Data

El tipus predefinit `Maybe` està definit amb un data:

```
data Maybe a = Just a | Nothing
```

El tipus predefinit `Either` està definit amb un data:

```
data Either a b = Left a | Right b
```

Admet funcions com ara:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
lefts :: [Either a b] -> [a]
rights :: [Either a b] -> [b]
partitionEithers :: [Either a b] -> ([a], [b])
```

Creació de tipus de dades

Podem definir tipus recursius:

```
data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Abuit
```

Creació de tipus de dades

Podem definir tipus recursius:

```
data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Abuit

data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Fulla a
```

Creació de tipus de dades

Podem definir tipus recursius:

```
data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Abuit

data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Fulla a

data ArbGen a = Node a [ArbGen a]
              | Abuit
```

Els constructors `Node`, `Abuit` o `Fulla` són operadors.

Creació de tipus de dades

No s'ha de confondre amb els Tipus sinònims:

```
type String  = [Char]
type Person  = (Name,Address)
type Name    = String
type AssocList a b  = [(a,b)]
```

Són com els typedef de C++, però admeten polimorfisme.

És a dir, els dos tipus són intercanviables.

Continguts

1. Tipus predefinits
2. Polimorfisme paramètric
3. Nous tipus de dades
4. Classes de tipus

Classes de tipus

Les classes de Haskell no són tipus, sino categories de tipus.

(Type) classes predefinides

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

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

És un mecanisme similar als Interfaces de Java.

- És la forma de tenir sobrecàrrega en Haskell.
- És una altra forma de polimorfisme.

Classes de tipus

La definició d'una class és una definició parcial:

```
class  Eq a  where
  (==), (/=)              :: a -> a -> Bool

  x /= y                  =  not (x == y)
  x == y                  =  not (x /= y)
```

El mínim que cal per completar la definició és definir el == o el /=

Quan diem que un tipus pertany a una classe cal completar la definició.

Classes de tipus

```
class  (Eq a) => Ord a  where
  compare              :: a -> a -> Ordering
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min             :: a -> a -> a

  compare x y
       | x == y    =  EQ
       | x <= y    =  LT
       | otherwise =  GT
  x <= y           =  compare x y /= GT
  x <  y           =  compare x y == LT
  x >= y           =  compare x y /= LT
  x >  y           =  compare x y == GT

data  Ordering  =  LT | EQ | GT
```

El mínim que cal per completar la definició és definir el `(<=)` o el `compare`

Altres classes: `Show`, `Read`, `Enum`, `Num`, ...

Classes de tipus

```
data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Abuit
```

Instanciació:

```
instance Eq a => Eq (ArbBin a) where
  x == y                  =  x `iguals` y
```

Classes de tipus

```
data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Abuit
```

Instanciació:

```
instance Eq a => Eq (ArbBin a) where
  (==)  =  iguals
```

Classes de tipus

```
data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Abuit
```

Instanciació:

```
instance Eq a => Eq (Arbre a) where
  Abuit == Abuit =  True
  _     == Abuit =  False
  Abuit == _     =  False
  (Node x a1 a2) == (Node y b1 b2) = x==y && a1==b1 && a2==b2
```

Classes de tipus

Instanciació predefinida: *deriving*

```
data Carta = Two | Three | Four
     | Five | Six | Seven | Eight | Nine | Ten
     | Jack | Queen | King | Ace
   deriving (Read,  Show, Enum, Eq, Ord)
```

Es basa en la definició per crear les operacions.
El `read` i el `show` usen els constructors com strings.

```
data ArbBin a = Node a (ArbBin a) (ArbBin a)
              | Fulla a
   deriving (Read,  Show, Eq, Ord)
```

No permet derivar `Enum`.
La (==) derivada és la mateixa que hem definit abans.

Classes de tipus

Ús de les classes en les declaracions de tipus.

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

Quin tipus té `suma`?

Classes de tipus

Ús de les classes en les declaracions de tipus.

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

Quin tipus té `suma`?

```
suma :: [a] -> a
```

Classes de tipus

Ús de les classes en les declaracions de tipus.

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

Quin tipus té `suma`?

```
suma :: [a] -> a
```

Incorrecte! el tipus `a` no pot ser qualsevol cosa.

Classes de tipus

Ús de les classes en les declaracions de tipus.

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

Quin tipus té `suma`?

```
suma :: [a] -> a
```

Incorrecte! el tipus `a` ha de tenir l'operció `(+)`.

Classes de tipus

Ús de les classes en les declaracions de tipus.

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

Quin tipus té `suma`?

```
suma :: [a] -> a
```

Incorrecte! el tipus `a` ha de ser de la classe `Num`.

Classes de tipus

Ús de les classes en les declaracions de tipus.

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

Quin tipus té `suma`?

```
suma :: [a] -> a
```

Incorrecte! el tipus `a` ha de ser de la classe `Num`.
Tipus correcte:

```
suma :: Num a => [a] -> a
```

El que es posa abans de => són condicions sobre les variables de tipus.

Classes de tipus

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)     :: a -> a -> a
    negate            :: a -> a
    abs, signum       :: a -> a
    fromInteger       :: Integer -> a

    x - y             =  x + negate y
    negate x          =  0 - x
```

Obliga a que els tipus de la classe `Num`, siguin també de la classe `Eq` i la classe `Show`.

Per fer un `instance`, cal definir totes le operacions menys `negate` o `(-)`

Els tipus `Int`, `Integer`, `Float` y `Double` són "instances" de la classe `Num`.

Creació de noves classes

```
class Pred a where
 sat :: a -> Bool
 unsat :: a -> Bool

 unsat = not . sat
```

Creació de noves classes

```
class Pred a where
 sat :: a -> Bool
 unsat :: a -> Bool

 unsat = not . sat
```

Podem instanciar de la manera usual:

```
instance Pred a => Pred (ArbBin a) where
 sat Abuit = True
 sat (Node x b c) = (sat x) && (sat b) && (sat c)
```

Creació de noves classes

```
class Pred a where
 sat :: a -> Bool
 unsat :: a -> Bool

 unsat = not . sat
```

Podem instanciar de la manera usual:

```
instance Pred a => Pred (ArbBin a) where
 sat Abuit = True
 sat (Node x b c) = (sat x) && (sat b) && (sat c)

instance (Pred Int) where
 sat 0 = True
 sat _ = False
```