Programación Funcional

Curso 2019-20

DEFINICIÓN DE FUNCIONES EN HASKELL

Más ajuste con patrones constantes: booleanos

```
not True  = False
not False = True
```

```
infixr 3 && – conjunción
True && y = y
False && _ = False
```

```
– También podría ser
True && True = True
_ && _ = False
```

```
infixr 2 || -- disyunción
False || y = y
True || _ = True
```

```
(||) False y = y
(||) True _ = True
```

Cosas que aprendemos

- Patrones para distinguir casos
- Definición de operador infijo

  `infixr`
  `infixl` *prioridad operador*
  `infix`

  `x && y && z` ≡ `x && (y && z)`
  `x && y || z` ≡ `(x && y) || z`
- Variable anónima `_`: cada aparición es una variable nueva
- Uso prefijo de operadores `(||)` , `(&&)`: a veces es necesario o conveniente

Ajuste de patrones no constantes: tuplas

Con ajuste de patrones

```
fst  (x,_) = x
snd  (_,y) = y
swap (x,y) = (y,x)
```

Sin ajuste de patrones

```
fst xy = ???
snd xy = ???
swap xy = (snd xy , fst xy)
```

Suma de complejos

```
infixl 6 +.
(a,b) +.(c,d) = (a+b,c+d)
```

Sin ajuste

```
z +.z' = (fst z+fst z',snd z+snd z')
```

Cosas que aprendemos

- Ajuste de patrones da acceso a componentes
- Sin ajuste de patrones
  - Necesitamos más primitivas
  - Programas más complejos

```
soluciones' (a,b,c) =
  let d = b^2-4*a*c
      e = -b/2*a
      r = sqrt d/2*a
  in  if d>0  then [e+r,e-r] else
      if d==0 then [e]
              else []
```

```
soluciones (a,b,c)
   | d>0  = [e+r,e-r]
   | d==0 = [e]
   | d<0  = []
   where d = b^2-4*a*c
         e = -b/2*a
         r = sqrt d/2*a
```

Cosas que aprendemos

- Patrones, guardas, defs. locales, condicionales pueden coexistir
- Patrón *(a,b,c)*: determina la forma del argumento y da acceso a sus componentes
- *let* no combina bien con ecuaciones guardadas
- *where*: definiciones locales cuyo ámbito se extiende a un grupo de ecuaciones previo con el mismo lado izquierdo
- *where* no forma parte de las sintaxis de las expresiones (como las guardas)
- *where* puede usarse para una sola ecuación

```
soluciones'' (a,b,c) =
  if d>0  then [e+r,e-r] else
  if d==0 then [e]
          else []
  where {d = b^2-4*a*c ; e = -b/2*a ; r = sqrt d/2*a}
```

Ajuste de patrones con listas

```
-- cabeza y resto de una lista
head :: [a] -> a
head (x:xs)  = x
tail :: [a] -> [a]
tail (x:xs) = xs

-- test de lista vacia
null :: [a] -> Bool
null []    = True
null (_:_) = False

-- longitud de una lista
length :: [a] -> Int
length [] = 0
length (x:xs) = 1+length xs

-- concatenacion de listas
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[ ]    ++ ys = ys
(x:xs) ++ ys = x:(xs++ys)
```

Cosas que aprendemos

- `(x:xs)` $\rightsquigarrow$ patrón genérico de lista no vacía
- Patrones distinguen casos y dan acceso a componentes
- Recursión como mecanismo de control
- Típicamente, la recursión implica recorridos izda-dcha de las listas

Recordemos: declaración de tipos es opcional, pero altamente recomendada

Otros ejemplos de programación con listas

```
-- pertenencia a una lista
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem x (x':xs) = x==x' || elem x xs

-- n-esimo elemento de una lista
infixl 9 !!
(!!) :: [a] -> Int -> a
(x:_)  !! 0 =  x
(_:xs) !! n =  xs !! (n-1)

-- suma de los elementos
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

```
-- Tomar n elementos en cabeza
take :: Int -> [a] -> [a]
take _ [] = []
take n (x:xs)
  | n <= 0 = []
  | otherwise = x:take (n-1) xs

-- Emparejar dos listas
zip :: [a] -> [b] -> [(a, b)]
zip (x:xs) (y:ys) = (x,y):zip xs ys
zip _ _ = []

-- Ultimo elemento de una lista
last :: [a] -> a
last [x] = x
last (x:x':xs) = last (x':xs)
```

Otro ejemplo: inversa de una lista

```
-- reverse [x1,...,xn] = [xn,...,x1]
reverse ::  [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Evaluación de `reverse [1,2,3,4]`

```
01 r [1,2,3,4] ≡(equivalencia sintáctica, no es un paso real)
02 r (1:2:3:4:[]) =
03 r (2:3:4:[])++[1] =
04 (r (3:4:[])++[2])++[1] =
05 ((r (4:[])++[3])++[2])++[1] =
06 (((r []++[4])++[3])++[2])++[1] =
07 ((([]++[4])++[3])++[2])++[1] =
08 (([4]++[3])++[2])++[1] =
09 ((4:([]++[3]))++[2])++[1] =
10 (4:(([]++[3])++[2]))++[1] =
11 4:((([]++[3])++[2])++[1]) =
12 4:(([3]++[2])++[1]) ≡
13 4:(((3:[])++[2])++[1]) =
14 4:((3:([]++[2]))++[1]) =
15 4:(3:(([]++[2])++[1])) =
16 4:(3:([2]++[1])) ≡
17 4:(3:((2:[])++[1])) =
18 4:(3:(2:([]++[1]))) =
19 4:(3:(2:[1])) ≡
20 [4,3,2,1]
```

15 pasos de reducción $\rightsquigarrow$ complejidad cuadrática

`reverse` con complejidad lineal

Usamos una función auxiliar con un argumento adicional que juega el papel de *acumulador*, en el que se va construyendo la inversa de `xs` según se va recorriendo `xs`

```
reverse xs  = revAux xs []

revAux :: [a] -> [a] -> [a]
revAux []     acc = acc
revAux (x:xs) acc = revAux xs (x:acc)
```

O bien, usando `where` para definir `revAux` localmente:

```
reverse xs = revAux xs []
  where
  revAux :: [a] -> [a] -> [a]
  revAux []     acc = acc
  revAux (x:xs) acc = revAux xs (x:acc)
```

Evaluación de `reverse` lineal

```
01 r [1,2,3,4] ≡
02 r (1:2:3:4:[]) =
03 raux (1:2:3:4:[]) [] =
04 raux (2:3:4:[]) (1:[]) =
05 raux (3:4:[]) (2:1:[]) =
06 raux (4:[]) (3:2:1:[]) =
07 raux [] (4:3:2:1:[]) =
08 4:3:2:1:[] ≡
09 [4,3,2,1]
```

6 pasos de reducción $\rightsquigarrow$ complejidad lineal

`raux` usa recursión final (*tail recursion*)