

Introducció a Haskell

Albert Rubio

Llenguatges de Programació, FIB, UPC

Primavera 2016

Continguts

- 1 Introducció
- 2 Primers programes
- 3 Tipus estructurats
- 4 Pattern matching

Continguts

- 1 Introducció
- 2 Primers programes
- 3 Tipus estructurats
- 4 Pattern matching

Presentació de Haskell

- Llenguatge funcional pur. No assignacions.
No gestió memòria explícita.
- Lazy evaluation: tractar estructures molt grans o infinites.
- Sistemes de tipus potents.
Tipus polimòrfics. Inferència de tipus automàtica.
- Funcions d'ordre superior. Funcions com a paràmetres.
Les funcions són “first class objects”
- Al 1987 degut a la proliferació de FPLs és decideix definir un Standard: HASKELL
- Al 1998 es crea una versió estable Haskell98.

Presentació de Haskell

- Usarem `ghc`. *Glasgow Haskell Compiler*.
Compilació separada: obtenir codi objecte
Muntar executable.
- Usarem `ghci`. Com a interpret. Carregant el programa.
Executant per línia de comandes.

Inicialment usarem l'interpret

Presentació de Haskell

Comandes ghci,

“:comanda” d’entre les que hi ha per exemple:

- `:?` ,per invocar el Help.
- `:l <nomarxiu>` , per carregar un arxiu amb codi.
- `:r` ,per repetir l’ultima acció.
- `:t <expr>` ,per consultar el tipus d’una expressió.
- `:quit` ,per sortir.
- ...

Continguts

- 1 Introducció
- 2 **Primers programes**
- 3 Tipus estructurats
- 4 Pattern matching

Definicions de funcions

- Les definicions de funcions comencen amb minúscula.
- Format: totes les definicions del mateix àmbit han de tenir la mateixa indentació.
- Com alternativa al format es pot usar “;” i “{” (veure exemples).
- Les funcions es podran definir amb guardes i “pattern matching”.
- Els paràmetres no es passen, s'apliquen! No calen tots!
- Les funcions es podran definir: composant funcions que poden incloure crides recursives.
- No existeix cap instrucció per iterar tipus **while**, **for**, etc.
- L'avaluació d'una expressió es fa usant les definicions.
- Es pot declarar el tipus de les noves funcions
nomf:: t1->t2->...->tn
o deixar que Haskell infereixi el tipus.

if-then-else i case

El `if-then-else` és una funció de tres paràmetres

- Un booleà i dues expressions del mateix tipus
- retorna el resultat d'una de les dues expressions

```
prod n m =  
    if n == 0 then 0  
    else (prod (n-1) m) + m
```

Similarment, el `case` també és una funció.

```
prod n m =  
    case n of  
    0 -> 0  
    n -> (prod (n-1) m) + m
```

Guardes

Afegeixen condicions al patró d'entrada (veure més endavant)

```
prod n m  
  | n == 0      = 0  
  | otherwise = (prod (n-1) m) + m
```

Entra per la primera satisfactible.

Es pot assumir que els anteriors han fallat

Noteu que la igualtat va després de la guarda!

Ús de funcions: crides

Els paràmetres no es passen s'apliquen.

NO cal passar tots els parametres.

Podem considerar que totes les funcions tenen un únic paràmetre i el seu resultat pot ser una nova funció.

Exemple:

`prod 3 5` és en realitat `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> Int -> Int
```

Ús de funcions: crides

Els paràmetres no es passen s'apliquen.

NO cal passar tots els parametres.

Podem considerar que totes les funcions tenen un únic paràmetre i el seu resultat pot ser una nova funció.

Exemple:

`prod 3 5` és en realitat `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> (Int -> Int)
```

Ús de funcions: crides

Els paràmetres no es passen s'apliquen.

NO cal passar tots els parametres.

Podem considerar que totes les funcions tenen un únic paràmetre i el seu resultat pot ser una nova funció.

Exemple:

`prod 3 5` és en realitat `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> (Int -> Int)
```

```
(prod 3) :: Int -> Int
```

Ús de funcions: crides

Els paràmetres no es passen s'apliquen.

NO cal passar tots els parametres.

Podem considerar que totes les funcions tenen un únic paràmetre i el seu resultat pot ser una nova funció.

Exemple:

`prod 3 5` és en realitat `(prod 3) 5`

Primer apliquem 3 i el resultat és un funció que espera un altre enter.

```
prod :: Int -> (Int -> Int)
```

```
(prod 3) :: Int -> Int
```

```
((prod 3) 5) :: Int
```

Tipus bàsics

Tipus predefinitos bàsics: Bool, Char, Int, Integer, Float

- Bool: True, False, &&, ||, not

Per exemple, podem definir xor com

```
xor :: Bool -> Bool -> Bool
```

```
xor x y = ( x || y ) && not ( x && y )
```

- Char: entre cometes simples ' '

Alguns caràcters especials: '\t', '\n', ...

Funcions de conversió (import Data.Char):

- ord :: char -> Int
- chr :: Int -> char.

Definicions senzilles

Tipus predefinitos bàsics: Bool, Char, Int, Integer, Float

- Int: $+$, $-$, $*$, $^$ (infix)
div, mod, abs i negate (prefix). 'div' (infix)
 - Acotats per 2147483647. Precisió arbitrària: Integer.
 - Podem passar a Float amb: fromIntegral.
- Float: reals però amb precisió limitada.

Operacions estàndard més

$^ :: \text{Float} \rightarrow \text{Int} \rightarrow \text{Float}$ i $** :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float}$.

Conversió: ceiling, floor i round $:: \text{Float} \rightarrow \text{Int}$.

- Double: per a tenir doble precisió.
- Rational: (racional) fraccions d'Integers amb precisió arbitrària.

Definicions locals: *let*

Per definir variables locals usarem *let*:

$\text{let } v_1 = E_1 ; \dots ; v_n = E_n \text{ in } E$

És una expressió i té el tipus d' E .

```
prod n m =  
  if n == 0 then 0  
  else let x = div n 2 ; y = mod n 2  
        in if y == 0 then 2* (prod x m)  
            else (prod (n-1) m) + m
```

Definicions locals: `let`

Per definir variables locals usarem *let*:

$\text{let } v_1 = E_1 ; \dots ; v_n = E_n \text{ in } E$

És una expressió i té el tipus d' E .

```
prod n m =  
    if n == 0 then 0  
    else let x = div n 2  
          y = mod n 2  
          in if y == 0 then 2* (prod x m)  
              else (prod (n-1) m) + m
```

Definicions locals: where

El *where* permet definir constants o funcions locals

No és una expressió!

Es posa al final per afegir les definicions que falten.

where {*Def1*;...*Defn*}

```
prod n m =  
    if n == 0 then 0  
    else if y == 0 then 2* (prod x m)  
        else (prod (n-1) m) + m  
where {x = div n 2;  
      y = mod n 2}
```

Definicions locals: where

El *where* permet definir constants o funcions locals

No és una expressió!

Es posa al final per afegir les definicions que falten.

`where {Def1; ... Defn}`

```
prod n m =  
  if n == 0 then 0  
  else if y == 0 then 2* (prod x m)  
       else (prod (n-1) m) + m  
  where x = div n 2  
        y = mod n 2
```

Definicions locals: where

El *where* permet definir constants o funcions locals

No és una expressió!

Es posa al final per afegir les definicions que falten.

`where {Def1; ... Defn}`

```
prod n m =  
    if n == 0 then 0  
    else if y == 0 then 2* (prod x m)  
        else (prod (n-1) m) + m  
    where x = div n 2  
          y = mod n 2
```

Es poden definir també funcions (locals) noves que s'han usat.

Definicions locals: where

El *where* permet definir constants o funcions locals

No és una expressió!

Es posa al final per afegir les definicions que falten.

`where {Def1; ... Defn}`

```
prod n m
  | n==0      = 0
  | y == 0    = 2* (prod x m)
  | otherwise = (prod (n-1) m) + m
where x = div n 2
      y = mod n 2
```

Es poden definir també funcions (locals) noves que s'han usat.

Les variables locals a les guardes han de ser definides amb **where**

Continguts

- 1 Introducció
- 2 Primers programes
- 3 Tipus estructurats**
- 4 Pattern matching

Tuples

Tupla: $(camp_1, \dots, camp_n)$

Exemples: $(3, \text{True}, 2)$ o $((3, 2), 'a')$

Els camps són de tipus heterogenis.

Les tuples de dos elements (parells) tenen un tracte especial. Proveu
`:t (,)`

Accés per tuples de dos elements: `fst` i `snd`

`snd (2,3)` és `3`

NO tenim operacions per accedir a les posicions per tuples generals

Ens les podem definir.

Millor amb pattern matching (veure després)

Llistes

Tots els elements són del mateix tipus.

Llista:

- llista buida: `[]`
- afegir un element: `(cap:cua)`. Proveu `:t (:)`
- concatenació: `l1++l2`.

Exemples: `(2:(3:[]))++(1:[])` és la llista `[2,3,1]`

Accés: `head` i `tail`

Hi ha moltíssimes operacions predefinides sobre llistes.

Sintaxi de les llistes:

`[2]` és el mateix que `(2:[])`
`[2,3,4,5]` és el mateix que `2:(3:(4:(5:[])))`

Continguts

- 1 Introducció
- 2 Primers programes
- 3 Tipus estructurats
- 4 Pattern matching

Pattern matching

Definició de funcions (simbòlica):

$\text{sumar } [] = 0$

$\text{sumar } (x:xs) = x + (\text{sumar } xs)$

$\text{expr1 matches expr2}$ si existeix una substitució per les variables de expr1 que la fan igual que expr2 .

Exemple:

$x : xs \text{ matches } [2, 5, 8]$, perquè és $2 : (5 : (8 : []))$

Substitució: $x = 2$; $xs = [5, 8]$

Entra pel primer matching que troba.

Es pot assumir que els anteriors han fallat

NO confondre amb *unificació* (Prolog).

Pattern matching

Els patrons també es poden usar en

- el **case**:

```
sumar l = case l of
    [] -> 0
    (x:xs) -> x+(sumar xs)
```

- el **where**

```
divisio n m
    | n < m = (0,n)
    | otherwise = (q+1,r)
    where (q,r) = divisio (n-m) m
```