Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

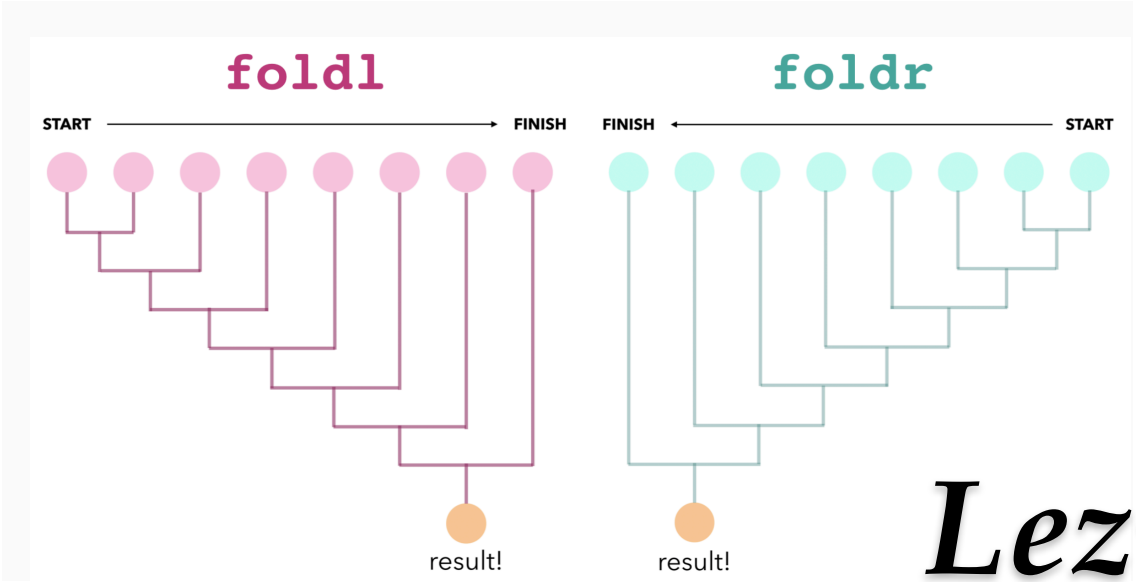
Verso un ordine superiore

Applicazioni a piccole applicazioni

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **4**, 4 marzo 2022



Lezione 4a:

Funzion(al)i di Ordine Superiore

Program Calculation: append

Come detto fin dall'inizio, i programmi Haskell riducono come le espressioni delle scuole medie.

Oggi vedremo alcuni esempi partendo da **++** (append).

```
[1, 2] ++ [3, 4, 5]
   = { notazione }
(1:(2:[])) ++ (3:(4:(5:[]))) =
   = { clausola (2) }
1:((2:[]) ++ (3:(4:(5:[])))) =
   = { clausola (2) }
1:(2:([] ++ (3:(4:(5:[]))))) =
   = { clausola (1) }
1:(2:(3:(4:(5:[]))))
   = { notazione }
[1, 2, 3, 4, 5]
```

```
  []   ++ ys = ys              -- (1)
(x:xs) ++ ys = x:(xs ++ xs)    -- (2)
```

Zucchero sintattico: enumerazioni

Riscaldamento: vediamo un po' di **utili notazioni** sulle liste, chiamate enumerazioni:

`[m..n]` è la lista **finita** [m, m+1, m+2, …, n]

`[m..]` è la lista **infinita** [m, m+1, m+2, …]

`[m,n..p]` è la lista **finita** [m, m+n-m=n, m+2(m - n), …, p]

`[m,n..]` è la lista **infinita** [m, m+n-m=n, m+2(m - n), …]

Esempio: `[1, 3..11]` è la lista `[1,3,5,7,9,11]`.

`[1, 3..]` è la lista dei **dispari** (**tutti!**)

Ma `[1, 1, 2, 3, 5..]` non è la lista di Fibonacci☺

Questa notazione **non è limitata agli interi**, ma coinvolge tutti i tipi di enumerati…

Esempio:

```
> ['a'..'z']
"abcdefghijklmnopqrstuvwzyz"
```

Curryficazione e Ordine Superiore

La **curryficazione** si fonda sul fatto che esiste un **isomorfismo**, anche da un punto di vista degli insiemi (attenti alle **parentesi**):

$$A \times B \to C \cong A \to (B \to C)$$

Il **vantaggio** delle versioni curryficate è poter passare solo un argomento e questo favorisce la composizionalità.

`myCurry` e `myUncurry` si scrivono "**guardando i tipi**" e sono funzioni che prendono **altre funzioni come argomento**.

È possibile invertire i parametri di una funzione, usando `flip`.

```
-- Esistono ovviamente due funzionali predefiniti
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
myCurry f a b = f(a, b)
-- il pattern matching decompone la
myUncurry f (a,b) = f a b
> uncurry (+) (23,19)
42
> curry (\(x,y)->x+y) (16,26)
42
```

```
-- piccoli funzionali che
-- aiutano a vivere meglio
flip :: (a -> b -> c) ->
                  b -> a -> c
myFlip f a b = f b a

> flip (:) [2,3] 1
[1,2,3]
```

Schemi di Programmi: map & filter

Abbiamo già visto **map** che applica una funzione a tutti gli elementi di una lista. Vediamo il tipo e una definizione.

Ricordiamo le proprietà principali di map:

map i = i **map (f . g) = map f . map g**

Un tipico funzionale è **filter** che **seleziona** i **valori** di una lista che **soddisfano** un certo **predicato**.

Un'interessante definizione di **filter**, usando **map** e **concat**.

```
-- applichiamo una funzione a tutti
-- i valori di una lista
map :: (a -> b) -> [a] -> [b]
myMap f (x:xs) = f x : myMap f xs
myMap _   []   = [] -- anche myMap _ _ = []

filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) =
   if p x then x:ts else ts
      where ts = filter p xs
```

```
test p x = if p x
              then [x]
              else []
filter p xs =
   concat . map (test p) xs
```

Program Calculation: filter

Vediamo come riduce la seconda versione di **`filter`** (definita in termini di **`concat`** e **`map`**). Consideriamo il predicato **`odd`** che vale True se x è dispari.

Stavolta facciamo **grandi passi**… (e non passi elementari)

abbiamo ridotto la composizione di funzioni

```
filter odd [15, 44, 60, 79, 81, 83]
   = { def. di filter }
concat(map (test p) [15, 44, 60, 79, 81, 83])
   = { map applica test odd a tutti gli elementi }
concat [test odd 15, test odd 44, test odd 60,
 test odd 79, test odd 81, test odd 83]
   = { applichiamo test odd ovunque}
concat [[15], [], [], [79], [81], [83]]
   = { applichiamo concat }
[15] ++ [] ++ [] ++ [79] ++ [81] ++ [83]
   = { applichiamo ++ ovunque (da sinistra)}
[15, 79, 81, 83]
```

```
filter p = concat . map (test p)
test p x = if p x then [x] else []
```

List comprehension

Un modo molto amato dagli Haskellioti per trattare le liste è **list comprehension**, reminescente di **set comprehension**, il **tipico modo di definire gli insiemi** in matematica. Dato un insieme A e un predicato P su A, si definisce l'insieme $\{ x \in A \mid P(x) \}$.

Si possono usare dei predicati (stile **`filter`**).

In effetti assomiglia molto a **`filter`** e **`map`**.

In realtà, in Haskell sono le definizioni date per list comprehension ad essere **tradotte in termini di `map` e `filter`**.

```
> [x^2 | x<-[1..5]]
[1,4,9,16,25]

firsts :: [(a,b)] -> [a]
firsts ps = [x | (x, _) <- ps]

-- lista dei fattori di un numero:
-- posso usare predicati, detti guards:
factors n = [x | x<-[1..n], n `mod` x == 0]

-- map & filter con list comprehension:
myMapLC f xs = [f x | x<-xs]
myFilterLC p xs = [x | x<-xs, p x]
```

More on List comprehension

Usando list comprehension si riescono a dare anche definizioni molto complesse.

Ad esempio, date due liste, possiamo calcolare **la lista di tutte le possibili coppie**, ossia il ”prodotto cartesiano” tra due liste (osservate l’ordine).

Anche altri funzioni come **concat** possono essere definite per list comprehension.

```
 -- prendendo da due liste è facile costruire
 -- la lista `prodotto cartesiano’
> [(x,y) | x<-[1,2,3], y<-[4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
 -- attenti all’ordine!
> [(x,y) |  y<-[4,5], x<-[1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]

myConcat xss = [x | x<-xs, xs<-xss]
> myConcat [[1,2,3],[4,5],[6,7],[8]]
[1,2,3,4,5,6,7,8]
```

Una famosa applicazione di map

John Backus (progettista di Fortran & Algol) nel **1978** scrisse un celeberrimo articolo che rilanciò la programmazione funzionale ponendo l'accento sulle sue **virtù composizionali**.

Fece l'esempio del **prodotto scalare**. Vediamo una prima versione.

Rispetto ai programmi iterativi **non ci sono** accumulatori e indici.

Notate che `[0..]` è la lista di **tutti i naturali**. Haskell è lazy, e `zip` prende solo le prime 6 potenze…

Ovviamente dobbiamo scrivere il **numero rovesciato**, ma ho scelto un esempio palindromo, per non sbagliare ☺

```
 -- idea: costruiamo la lista di coppie
 -- mappiamo la funzione uncurry (*)
 -- sommiamo i prodotti ottenuti
ps1 xs ys = sum (map (uncurry (*)) (zip xs ys))

-- vediamo che numero è 101101 in binario
>ps1 [1,0,1,1,0,1] [2^x | x<-[0..]]
45
```

Program Calculation: ps1

Ecco il prodotto scalare (sempre a grandi passi)

```
ps1 [5, 4, 9] [10^x | x <-[0..]]
```
= { def. di **ps1** }
```
sum(map (uncurry (*)) (zip [5, 4, 9] [10^x | x <-[0..]]))
```
= { **zip** srotola le liste, notazione list-comprehension) }
```
sum(map (uncurry (*)) (zip [5,4,9] [1,10, 100, 1000, …]))
```
= { applichiamo **zip**, che **taglia la coda infinita**}
```
sum(map (uncurry (*)) [(5,1),(4,10),(9,100)]))
```
= { **map** applica **uncurry (*)** a tutti le coppie }
```
sum ([(5*1),(4*10),(9*100)]))
```
= { facciamo un passo di **sum** per vedere quando
```
(5*1) + sum [(4*10),(9*100)]
```
= { … e così via e poi prodotti e somme…}
```
[945]
```

attenzione: non è detto che l'ordine di valutazione sia proprio questo, causa laziness!

```
ps1 xs ys = sum (map (uncurry (*)) (zip xs ys))
```

Ma senza currying e zip?

Per capire a fondo i meccanismi dell'**ideologia funzionale**, scriviamo **un'altra versione** del prodotto scalare.

Nessuno mi vieta di avere liste di funzioni…

A questo punto, occorre un funzionale che data una lista di funzioni e di argomenti, applica **ordinatamente** una lista di funzioni a una lista di argomenti. Io lo chiamo **`applyL`**.

```
-- applicazione parziale di una funzione binaria
>t map (*)
(Num a) => [a] -> [a -> a]

applyL :: [a -> b] -> [a] -> [b]
applyL (f:fs)(x:xs)= f x : applyL fs xs
applyL    _    _   = []
-- notare come ho risolto i casi in cui almeno
-- una delle due liste è vuota (vedi zip)

-- versione alternativa di prodotto scalare
ps2 xs ys = sum (applyL (map (*) xs) ys)
```

Schemi di programmi: zipWith

Nel mondo Haskell, il mio pet-functional **`applyL`** non è molto popolare… Decisamente più popolare **`zipWith`**.

Se guardate i tipi, è una **variazione di `map`** che mappa una **funzione binaria** su due liste.

Volendo c'è pure quella *n*-aria. Vedremo come generalizzare.

`zipWith` **è più 'generale' di `zip`**: infatti `zip` si può ottenere da `zipWith` passando come funzione l'operatore **`(,)`** che genera le coppie, che potete definire come: **`(,) = \x y -> (x, y)`**

⏵ **Esercizio**: Scrivere `zipWith` senza decomporre liste con ricorsione, usando `applyL` (e se lo ritenete utile, `map`)

```
 -- spesso è utile una versione 'binaria' di map
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
myZipWith f (x:xs)(y:ys)= f x y: myZipWith f xs ys
myZipWith f   _     _   = []

zip = zipWith (,)

-- ancora il prodotto scalare:
ps3 xs ys = sum (zipWith (*) xs ys)
```

Facezie con map, filter & co.

Generalizzando **`and`** e **`or`** lungo la direzione di **`filter`**, otteniamo una specie di "esiste" e "per ogni" sugli elementi di una lista, che sono anch'essi predefiniti: **`any`** ed **`all`**.

Sono facilmente definibili usando `map`.

Oppure usando `filter`.

Vediamo due esempi di uso.

```
-- applichiamo una funzione a tutti
-- i valori di una lista
any :: (a -> Bool) -> [a] -> Bool
all :: (a -> Bool) -> [a] -> Bool

-- usando map e i già visti and/or
myAny p = or . map p
myAll p = and . map p

-- oppure usando filter:
anotherAny p = null . filter p
anotherAll p = not any
```

```
> all even [2, 4, 6, 8]
True
> any odd [2, 4, 6, 8]
False
```

Ricerca su lista

Anche qui, vedere se un valore v sta in una lista si può fare scorrendo banalmente una lista (**ricorsivamente**, s'intende).

Immaginiamo si voglia ritornare la **posizione** dell'elemento (se presente) **-1 altrimenti**. Occorre passare la posiz. sui parametri.

```
find :: (Eq a) => a -> [a] -> Int
find v xs = findAux v xs 0
 -- ricerca sequenziale su lista non ordinata
findAux v (x:xs) n =
   if v==x then n
      else findAux x ys (n+1)
findAux x []     = -1

 -- stile funzionale
findVPH xs =
 -- seleziona il primo elemento (posiz.) dal primo elem.
 (fst . head) (
   -- seleziona l'eventuale occorrenza di v
    filter (\(x,y)->y==v)
    -- accoppia posizioni/valori mettendo un v posticcio
       (zip [0..] xs)++[(-1,v)])
```

Program Calculation: findVPH

Vediamo un'esecuzione di find. Chiamo **`t10`** il termine **`(\(x, y) -> 10==y)`** per ragioni di spazio.

facciamo le riduzioni lazy stavolta

```
findVPH 10 [5, 10, 4, 9, 10]
   = { def. di findVPH }
fst (head (filter t10 (zip [0..] [5, 10, 4, 9, 10])))
   = { zip srotola le liste e le accoppia – versione lazy}
fst(head(filter t10 (0,5):zip [1..] [10, 4, 9, 10])))
   = { applichiamo filter t10, che scarta il primo)}
fst(head(filter t10 (zip [1..] [10, 4, 9, 10])))
  = { applichiamo zip, che offre una nuova coppia)}
fst(head(filter t10 (1,10):(zip [1..] [4,9,10])))
   = { filter passa questa coppia }
fst(head ((1,10):((filter( t10 (zip [1..] [4,9,10])))
 = { head può ridurre, scartando la coda }
fst (1,10) = 1
```

```
findVPH v xs =
  (fst . head) (filter (\(x, y) -> v==y) (zip [0..] xs)
```

Schemi di programmi: foldr

Guardate le definizioni di tutte queste funzioni sulle liste viste la lezione precedente: hanno tutte la stessa struttura!

Si può generalizzare il loro **pattern di ricorsione** con un un unico funzionale: **`foldr`**.

L'idea è che `foldr (#) v [x0,x1, …,xn] = x0#(x1#(… (xn # v)…))`

dove `(#)` è un generico operatore o funziona binaria e `v` un valore.

Ad esempio:

```
foldr (+) 0 [1,2,3] = 1+(2+(3+0))) = 6 =sum [1, 2, 3]
```

```
sum :: (Num a) => [a] -> a
mySum   []   = 0
mySum (x:xs) = x + mySum xs

length :: [a] -> Int
myLength [] = 0
myLength (x:xs) = 1 + myLength xs

and :: [Bool] -> Bool
myAnd [] = True
myAnd (x:xs) = x && myAnd xs
```

Schemi di programmi: foldr

In generale, **`foldr`** generalizza tutte quelle ricorsioni in cui si 'raccolgono' e calcolano i risultati al rientro dalla ricorsione.

Attenzione ai tipi: **`foldr`** inietta una funzione binaria in una lista. Il primo argomento ha il tipo dei valori della lista (**a**), il secondo è uguale al tipo di ritorno (**b**) perché si applica sui risultati ottenuti ricorsivamente.

Esercizio: scrivere **`map f`** usando **`foldr`** (e senza decomporre liste!)

```
foldr ::  (a -> b -> b) -> b -> [a] -> b
myFoldr f v (x:xs) = f x (myFoldr f v xs)
myFoldr f v   []   = v
-- ad esempio:
mySum = foldr (+) 0
myLength = foldr (\x->(+1)) 0
myAnd = foldr (&&) True
```

La sorellina di foldr: foldr1

Alcune funzioni (ad esempio il minimo di una lista) non hanno un evidente valore "iniziale" che si può dare su lista vuota che non influenzi la computazione globale (di solito è l'elemento neutro dell'operazione).

Nel caso del minimo (risp. massimo) sarebbe +∞ (risp. -∞). Per questo c'è **`foldr1`**.

```
foldr1 ::  (a -> a -> a) -> [a] -> a
myFoldr1 f (x:xs) = f x (myFoldr1 f v xs)
myFoldr1 f   [x]  = x

-- ad esempio:
myMinimum = foldr1 min
myMaximum = foldr1 max

-- se vi piace:
mySum = foldr1 (+)
-- ma dà errore su lista vuota!
```

Verificare se una lista è ordinata

Possiamo ovviamente scrivere un programma per pattern matching, che ricalca gli **analoghi programmi imperativi,** che semplicemente confronta ogni elemento col successivo…

Il vero programmatore Haskell preferisce comporre funzioni, vedendo le **strutture dati** nella loro **globalità**.

```
 -- stile imperativo: controlla ogni el col success.
ordinata []  = True
ordinata [x] = True
ordinata (x:y:xs) = x <= y && ordinata (y:xs)

 -- stile funzionale
ordinataVPH xs =
   foldr (&&) True (zipWith (<=) xs (tail xs))

  -- c'è l'and predefinito sulle liste
and = foldr (&&) True  -- io lo chiamerei forall
ordinataVPH' xs = and (zipWith (<=) xs (tail xs))
```

Lezione 4a:

Moltiplicazione tra matrici

Oltre ps: prodotto tra matrici

Rappresentiamo una **matrice** come una **lista di liste**, memorizzata per righe.

Dovendo fare il prodotto riga per colonna, e volendo usare la funzione **ps** che fa il prodotto scalare, cominciamo a scrivere una funzione che genera la **trasposta**.

Dopodiché dovremmo "mappare" per ogni riga r della prima, la funzione `(prodottoScalare r)` nella seconda. Come fare?

```
 -- trasposta di una matrice
 -- caso degenere: la matrice è vuota
trasposta []    = []
 -- vero caso base: la matrice ha una sola riga
trasposta [xs]  = map (\x->[x]) xs
 -- se ho trasposto n-1 righe, mi basterà mettere
 -- tutti gli elementi della prima riga in testa
 -- alle colonne (=righe)…
trasposta (xs:xss) = zipWith (:) xs (traposta xss)
```

Miracoli dell'ordine superiore

Ovviamente avendo una riga `r` e una lista di colonne `b`, è facile scrivere il prodotto di `r` per `b`.

Dopo aver astratto su `r` otteniamo una funzione che può essere mappata dentro la prima matrice da moltiplicare.

A ben vedere (essendo prodottoScalare scritto con una map) le tre map annidate corrispondono a 3 cicli for annidati, ma fa tutt'un altro effetto 😉

```
 -- riga i-esima del prodotto (r è i-esima riga)
 -- b la matrice da moltiplicare
map (prodottoScalare r) b
 -- per fare il prodotto occore mappare questa
 -- funzione dentro la matrice a
 -- avendo cura di astrarre su r
prodMat a b =
  map (\x-> map (prodottoScalare x)
                (trasposta b)) a
 -- miracoli dell'ordine superiore
 -- notare l'utilità di astrarre e applicare 1 arg.
```

Lezione 4

That's all Folks…

Grazie per l'attenzione…

…Domande?