Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Programmazione su Liste II
Efficienza

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **4**, 11 marzo 2021

Lezione 4a:

Moltiplicazione tra matrici

...andiamo oltre: prodotto tra matrici

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

Dovendo fare il prodotto riga x colonna, e volendo usare la funzione `prodottoScalare`, cominciamo a scrivere una funzione che genera la trasposta.

Dopodichè 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 funziona 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 4b:

Un po' di zucchero sintattico [e non solo]

Con un poco di zucchero (sintattico)

Molto usata dai programmatori Haskell la clausola **where** che permette di scrivere programmi simili a usuali notazioni matematiche.

Si usa prevalentemente in due situazioni:

1. definizioni **locali** di funzioni/dati;
2. **evitare di ricalcolare** un valore che serve più di una volta.

Vediamo l'esempio di powerset (++ è l'append tra liste).

```
 -- curiosamente questo termine tipa... perché?
j = x x where x = \y -> y
 -- l'auto-applicazione è limitata a un tipo noto

 -- vediamo il powerset: distinguiamo gli "insiemi"
 -- che contengono x e quelli che non la contengono
powerset (x:xs) = pws ++ map (x:) pws where
    pws = powerset xs
 -- qui where ci evita di ricalcolare lo stesso set
powerset [] = [[]]
```

Efficienza I

Nelle liste, l'inserzione in testa si fa in tempo $\theta(1)$, mentre l'inserzione in coda costa $\theta(n)$, dove n è la lunghezza della lista.

Ne consegue che la naturale definizione del rovesciamento di una lista è $\theta(n^2)$ invece che lineare come uno si aspetterebbe.

Si può migliorare l'efficienza, **accumulando i risultati parziali nei parametri**!

Osservate che `reverseAux` **non ha senso come funzione standalone** (top-level). Viene definita come **locale**.

```
 -- definizione di append (uguale a ++)
append (x:xs) ys = x:append xs ys
append    []  ys = ys
 -- funzione reverse, versione 1
reverse (x:xs) = reverse xs ++ [x]
reverse   []   = []
 -- funzione reverse lineare
reverseE (x:xs) = reverseAux xs [] where
   reverseAux (x:xs) ys = reverseAux xs (x:ys)
   reverseAux   []   ys = ys
```

Un ricordo del prof. Corrado Böhm

"La programmazione funzionale è stateless, ma i parametri delle funzioni sono stateful"

si possono usare i parametri per accumulare i risultati parziali di una computazione

inaugurazione del primo calcolatore "italiano" allo IAC (~1955)

foldl e foldr

Il funzionale `myFold` definito la lezione scorsa è predefinito in Prelude col nome `foldr`. Ha una sorellina: `foldl`, utile proprio quando una **ricorsione trae vantaggio nello "spingere in avanti"** dei risultati durante una ricorsione come `reverseEff`.

Il parametro `v` accumula i risultati calcolati durante la discesa ricorsiva.

Anche se per funzioni associative (come +) non fa differenza, la sommatoria con `foldl` risulta più efficiente perché quella definita con `foldr` necessita di aspettare il ritorno dalle chiamate ricorsive per eseguire i conti e quindi **lascia molte espressioni non valutate in memoria**.

Prosaicamente:

$$foldl\ (\#)\ v\ [x_0, x_1, \ldots, x_n] = (\ldots((v\ \#\ x_0)\ \#\ x_1)\ldots)\ \#\ x_n$$

$$foldr\ (\#)\ v\ [x_0, x_1, \ldots, x_n] = x_0 \# (x_1 \# (\ldots (x_n \#\ v)\ \ldots))$$

```
myFoldl :: (a -> b -> a) -> a -> [b] -> a
myFoldl f v    []    = v
myFoldl f v (x:xs) = myFoldl f (f v x) xs
```

back to reverse

Dovrebbe essere chiaro che `reverse` è il *foldr* di `(++)` (circa perché chiaramente occorre invertire i parametri e rendere lista il primo parametro).

Mentre `reverseEff` si può ottenere con il *foldl* di `(:)` (circa perché anche qui occorre invertire il ruolo dei parametri).

Più elegantemente lo possiamo fare con un funzionale.

```
reverse    = foldr (\x xs -> xs ++ [x]) []
reverseEff = foldl (\xs x -> x:xs) []
-- ma definendo un funzionale invertipar simile
-- a curry e uncurry
Invertipar :: (a -> b -> c) -> b -> a -> c
invertipar f x y = f y x
-- abbiamo:
reverseEff' = foldl (invertipar (:)) []
```

Efficienza: fibonacci

Vediamo un esempio famoso: la funzione di fibonacci.

Dovreste sapere che applicare la regola induttiva per calcolare i numeri di fibonacci porta a un programma di complessità esponenziale (per la precisione $\theta(\varphi^n)$, dove $\varphi=\frac{1+\sqrt{5}}{2}$).

Il programma iterativo "efficiente" (si può fare anche meglio a dire il vero) esegue $\theta(n)$ somme mantenendo memorizzati gli ultimi 2 numeri di fibonacci calcolati.

```
 -- fib inefficiente che ricalcola molte volte
 -- gli stessi valori (a meno di memoization)
fib n = if n<2 then n else fib (n-1) + fib (n-2)

 -- funzione efficiente che mantiene gli ultimi
 -- due numeri nei parametri della funz. ausiliaria
fibEff n = if n<2 then n else fibAux n 1 0 where
   fibAux 1 f1 f2 = f1
   fibAux n f1 f2 = fibAux (n-1) (f1+f2) f1
```

Composizionalità…

Problema: generare tutti i sottoinsiemi di k elementi di una lista di n elementi. Questi sono $\binom{n}{k}$.

I programmatori Haskell amano scrivere funzioni componendone altre. Usiamo il `powerset` e `filter`.

Provare a generare le cinquine del lotto con questo metodo!

```
 -- filter seleziona la sottolista degli elementi
 -- che soddisfano il predicato p
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p (x:xs)
  |     p x     = x : myFilter p xs
  | otherwise = myFilter p xs
filter p [] = []
 -- è facile ora definire le combinazioni
cmbs k xs = filter (\x->length x == k) (powerset xs)
 -- oppure ricorndando il compositore…
cmbs k xs = filter ((== k).length) (powerset xs)
 -- esagerando…
cmbs k = filter ((== k).length) . powerset
```

…ed Efficienza

A volte risulta necessario "entrare nelle funzioni" per specificarne il comportamento…

Scriviamo una funzione che genera solo le $\binom{n}{k}$ combinazioni invece delle 2^n sottosequenze per poi filtrarle.

Osservate che:

- segue lo stesso schema ricorsivo del calcolo ricorsivo dei coefficienti binomiali.
- nella funzione ausiliaria cmbs' n è sempre la lunghezza di xs.

```
cmbs k xs = cmbs' k xs (length xs) where
  cmbs' xs@(x:txs) n k
    | n==k       = [xs]
    | k==0       = [[]]
    | otherwise = map (x:) (cmbs' txs (n-1) (k-1))
                    ++ (cmbs' txs (n-1) k)
```

Lezione da apprendere

Si può usare la **versione inefficiente** (ma evidentemete facile da capire) **come la specifica**, dopodiché con ragionamenti puramente **algebrici e/o induttivi** si dimostra **l'equivalenza con la funzione efficiente**.

Faremo questo giochetto più volte nel seguito del corso.

Lezione 4c:

Un altro cucchiaino di zucchero

List comprehension

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

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

```
-- [1..5] è zucchero sintattico per [1,2,3,4,5]
> [x^2 | x<-[1..5]]
[1,4,9,16,25]
firsts :: [(a,b)] -> [a]
firsts ps = [x | (x, _) <- ps]
-- Più in generale:
myMapLC f xs = [f x | x<-xs]
-- posso usare predicati, detti guards:
myFilterLC p xs = [x | x<-xs, p x]
-- lista dei fattori di un numero:
factors n = [x | x<-[1..n], n `mod` x == 0]
```

List comprehension ed efficienza

Vediamo il "prodotto cartesiano" tra due liste (osservate l'ordine).

Definiamo la funzione `concat` 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]

-- ma anche:
myConcat' = foldl (++) []
```

Un grande classico della propaganda

Questo è un programma da libro di scuola, che mostra quanto sia facile programmare **quicksort** in Haskell.

Ma si tratterà di verità o propaganda?

Qual è il punto di forza di Quicksort (rispetto a Mergesort)? Non allocare memoria e l'efficienza della procedura partiziona (che poi permette di `ricombinare' i risultati in $\theta(1)$). E la complessità di ++?

Ma qui? quanta memoria viene allocata?

```
 -- in Haskell quicksort si scrive in 3 righe…
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a<-xs, a<=x ]
      larger  = [b | b<-xs, b>x ]
```

Efficienza II: Tupling

Possiamo migliorare l'efficienza di QuickSort facendo l'operazione di partizionamento in **un'unica passata** della lista, rinunciando purtroppo a list comprehension.

Secondo Richard Bird, il **tupling è il duale dell'uso dei parametri**: metto **più informazione nei risultati** calcolati dalla funzione.

```
-- possiamo tuttavia evitare le due passate
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
   where
      (smaller, larger) = partiziona xs x
      partiziona (x:xs) p =
         if x<=p then (x:s,l) else (s, x:l)
               where (l,s) = partiziona xs
      partiziona [] p = ([], [])
```

... e MergeSort?

Cominciamo con lo scrivere in Haskell una funzione `merge` che fonde liste ordinate.

Ricordiamo le funzioni predefinite:

`take :: Int->[a]->[a]` e `drop :: Int->[a]->[a]`

che rispettivamente **prendono** e **scartano** i primi *n* elementi da una lista e voilà (**esercizio**: dare le definizioni di `take` e `drop`).

Esercizio: scrivere MergeSort in modo che scomponga la lista in un'unica passata (attenzione che `length` scorre la lista)

```
merge [] ys = ys
merge xs [] = xs
merge xs@(x:txs) ys@(y:tys)
  |    x < y    = x:merge txs ys
  | otherwise = y:merge xs tys
mergeSort xs =
    merge (mergeSort (take n xs))
          (mergeSort (drop n xs))
   where n = length xs `div` 2
```

Fibonacci Efficiente con Tupling

Vediamo come scrivere la funzione efficiente di Fibonacci usando le coppie piuttosto che i parametri per memorizzare gli ultimi due numeri di Fibonacci.

Un approccio "più generale" definendo un funzionale `for`.

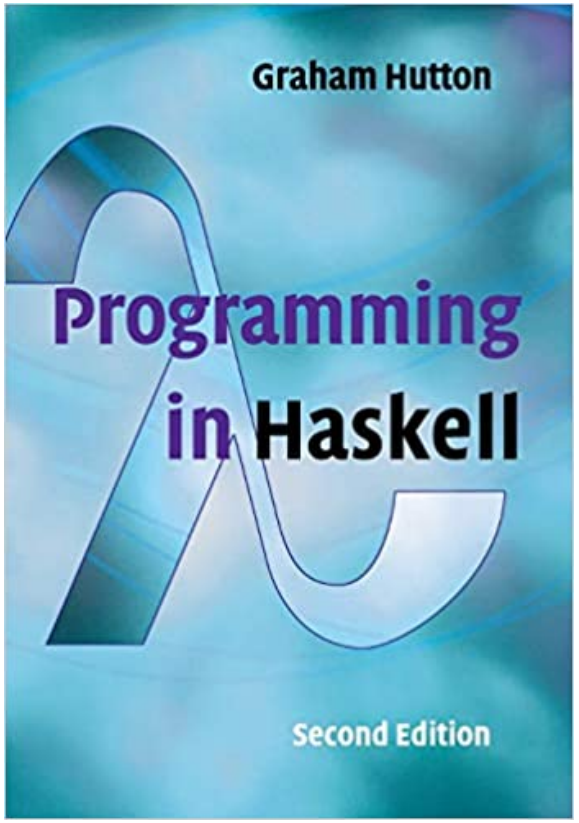
```
-- l'idea è iterare la trasformazione di fibonacci
 fib = fst . fibAux where
   fibAux 0 = (0,1)
   fibAux n = (f1+f2, f1)
      where (f1, f2) = fibAux (n-1)

-- oppure possiamo definire un funzionale for che
-- compone n volte una funzione…
for f 0 = \x -> x
for f n = f . (for f (n-1))
>:t for
for :: (Num a, Eq a) => (b -> b) -> a -> b -> b
-- e lo applichiamo alla trasf. di fibonacci
fib n = (fst . (for g n)) (0,1)
        where g (x, y) = (x+y, x)
```

Lezione 4d:

Una piccola applicazione

tratta da:



Voting System - 1

Problema: Abbiamo una lista che raccoglie le preferenze. Vogliamo determinare quale candidato ha avuto più voti.

Scriviamo intanto una funzione che conta i voti. Privilegiamo la programmazione "composizionale"

Tornerà infine utile rimuovere i duplicati. Vediamo ancora una soluzione composizionale.

```
> votes :: [String]
votes = [“Rossi”, “Bianchi”, “Verdi”, “Bianchi”,
“Bianchi”, “Rossi”]

>count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)
> count “Rossi” votes
2

rmvDups :: Eq a => [a] -> [a]
rmvDups   []   = []
rmvDups (x:xs) = x : filter (/=x) xs
>rmvDups votes
[“Rossi”, “Bianchi”, “Verdi”]
```

Voting System - 2

Scriviamo una funzione `result` che presenta i risultati ordinati in ordine di preferenza. Con le definizioni date finora, posso:

1. calcolare la lista dei candidati (senza ripetizioni)
2. per ciascun candidato, contare i voti
3. accoppiare in una lista voti/candidati e ordinarla.

Infine, il vincitore sarà l'ultimo di questa lista (e proietto il secondo elemento della coppia)

Esercizio: definire `last`.

```
results :: Ord a => [a] -> [(Int, a)]
results vs = sort [(count v vs, v) | v <- rmvDups vs]

>results votes
[(1, "Verdi"), (2, "Rossi"), (3,"Bianchi)]

-- and the winner is…

winner :: Ord a => [a] -> a
winner = snd . last . results
```

Voting System - 3

Problema: Consideriamo un caso più complesso: la lista di voti è una sequenza di liste di preferenze.

Il vincitore si ottiene **eliminando iterativamente quelli che ricevono meno prime scelte**.

Nel nostro esempio, il primo a essere eliminato è `"Rossi"` e poi viene rimosso… finché non resta un solo candidato.

```
> ballots :: [[String]]
votes = [["Rossi", "Verdi"],
         ["Bianchi"],
         ["Verdi", "Rossi", "Bianchi"],
         ["Bianchi", "Verdi", "Rossi"],
         ["Verdi"]]
votes' = [["Verdi"],
         ["Bianchi"],
         ["Verdi", "Bianchi"],
         ["Bianchi", "Verdi"],
         ["Verdi"]]
votes''=[["Verdi"],[],["Verdi"], ["Verdi"],["Verdi"]]
```

Voting System - 4

Possiamo sfruttare parte del lavoro precedente per scrivere una funzione `rank` che ordina i candidati.

Poi dobbiamo eliminare il candidato eliminato ed eventuali ballot vuoti.

E infine comporre i pezzi.

```
rank :: Ord a => [[a]] -> [a]
rank = map snd . result . map head

elim :: Eq a => [[a]] -> [[a]]
elim x = map (filter (/=x))

rmvEmpty = filter (/=[])

winnerB :: Ord a => [[a]] -> a
winnerB  = let (c:cs) = rank (rmvEmpty bs)
   in if cs =[] then c
                else winnerB (elim c bs)
```

Lezione 4

That's all Folks…

Grazie per l'attenzione…

…Domande?