Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Perle di Laziness I: Hamming & Ulam Numbers

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **11**, 12 aprile 2021

Lezione 11a

Riscaldamento: Piccoli miracoli di Laziness

Miracoli Lazy: minimo con mergesort

Per cominciare, un esempio molto famoso di "sfruttamento" della laziness a fini di efficienza (tratto dall'Haskell wiki: `https://wiki.haskell.org/Performance/Laziness`)

Vediamo la definizione di mergesort.

`myMin` è lineare nella lunghezza della lista!

Attenzione che la cosa ha senso perché `dividi` **è una funzione che `estrae' subito dell'informazione**. Se fosse scritta nello stile di `foldl` (e ci sono implementazioni naturali siffatte) il giochino non funziona!

```
 -- divide in due una lista
dividi []  = ([],[])
dividi [x] = ([x],[])
dividi (x:y:xs) = (x:ls, y:rs) where
   (ls,rs) = dividi xs
 -- mergesort sulle liste:
mergesort xs = merge (mergesort ls)(mergesort rs)
   where (ls, rs) = dividi xs
 -- minimo sulle liste:
myMin = head . mergesort
```

Problemi con liste infinite II

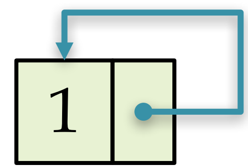
Attenzione che alcuni funzionali giustamente non sanno che ad esempio la lista è crescente! Il processo di generazione **deve consumare meno input di quanto output produce**!

Attenti anche a **list-comprehension** su liste infinite! o **filter**!

```
-- potremo essere tentati di definire:
factorsNT n = [x | x<-[1..], n `mod` x == 0 ]
-- factors n = [x | x<-[1..n], n `mod` x == 0 ]
-- non termina perché cerca in tutti i naturali...
> factorsNT 24
[1,2,3,4,6,8,12,24^CInterrupted.
-- utile usare takeWhile che si ferma quando la
-- condizione è falsa
factors n = [x | x<-takeWhile (<n)[1..],
        n `mod` x == 0 ]
> factors 24
[1,2,3,4,6,8,12,24]
-- lo stream dei primi si definisce facilmente...
primes = [p | p<-[2..], factors p = [1,p]]
```

Definizioni Circolari

In una definizione come `ones` o `nats'`, il compilatore Haskell capisce che deve generare una lista infinita e durante la generazione è sufficiente **spostare un puntatore**.



Cosa che non gli è possibile per `powers` o `natGren` che dipendono da un parametro… tuttavia…

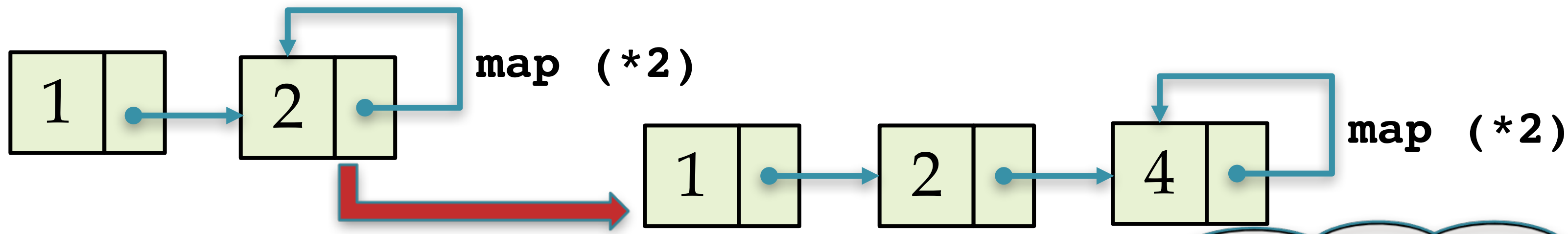
```
 -- Quindi è molto meglio:
powers' n = pws where pws = 1 : map (n*) pws

 -- Generalizziamo: non circolare
iter f x = x : map f (iter f x)
 -- oppure: circolare:
 -- il trucco è liberarsi dei parametri necessari
 -- in una definizione locale in cui sono `globali`
iterCirc f x = fs where fs = x : map f fs
```

Valutazione Lazy di Espressioni

Vediamo la differenza nella valutazione di `iter` e `iterCirc`.

Osservate che fs viene `prodotta` un elemento alla volta e il nostro processo **è sempre allineato all'ultimo elemento prodotto** quindi è sufficiente moltiplicare per 2 l'ultimo elemento generato!




```
iter (*2) 1
→ 1:map (*2) (iter (*2) 1)
→ 1:2:map (*2) (map (*2) (iter (*2) 1))
→ 1:2:4:map (*2) (map (*2)(map (*2) (iter (*2) 1)))

iterCirc (*2) 1
→ 1 : map (*2) fs
→ 1 : 2 : map (*2) fs
→ 1 : 2 : 4 : map (*2) fs
```

produrre n elementi è quadratico in n

produrre n elementi è lineare in n

Computazione di Fibonacci

Dovrebbe esservi naturale ragionare in termini di equazioni ricorsive, e trovare quella la **cui unica soluzione è la successione di Fibonacci**. Prendiamo un approccio più "artigianale".

Vediamo come procede la computazione: all'inizio `fibs` ha solo i primi due numeri, quindi è pronto anche il primo di `tail fibs`

```
fibs       = 0 | 1 | 1  2  3  5  8 13 21 34 55
tail fibs  = 1 | 1 | 2  3  5  8 13 21 34 55 89
```

input pronto — input non disponibile

Attenzione alle definizioni circolari! Il processo di generazione **deve consumare meno input di quanto output produce**!

L'equazione ricorsiva sotto **è soddisfatta da ogni stream** e infatti **non definisce niente**! È inconsistente!

```
 -- per esempio:
incstnts = (head incstnts):(tail incstnts)
```

Una piccola chicca per finire…

Definiamo lo stream dei numeri di **Fibonacci** usando il trucco visto prima di considerare due stream di fibonacci un un passo avanti all'altro e sommo i numeri corrispondenti.

Visto che la definizione è circolare, in realtà esiste un'unica copia della lista `fibs`.

```
 -- Uso di un generatore (parametrico) per
 -- definire i naturali:
fibs = 0:1:zipWith (+) fibs (tail fibs)

 -- È più carino usare definizioni guardate,
 -- in cui non posso chiedere il tail e produco
 -- sempre almeno un elemento (ho mutua ricorsione)
fibs' = 0:fibs''
fibs''= 1:zipWith (+) fibs' fibs''

 -- da cui, sostituendo in fibs''
 --(anche da fibs, spingendo dentro l'uno...)
fibs'''= 0:zipWith (+) fibs''' (1:fibs''')
```

Lezione 11b

Hamming Numbers

Hamming Numbers (1)

Problema: generare tutti i composti di 2, 3 e 5 in ordine crescente.

Formalmente: $H = \{ 2^i \cdot 3^j \cdot 5^k \mid i, j, k \geq 0\}$

Cominciamo con una soluzione che filtra tutti i numeri che non sono composti di 2, 3 e 5 (in verità **generalizziamo a un qualsiasi insieme di generatori**).

Questa soluzione è estremamente inefficiente, perché gli Hamming numbers hanno **densità 0**! ($\lim_{n\to\infty} H_n/n = 0$, dove H_n è il numero di elementi di una successione minori di n).

```
 -- funzione che controlla se un numero n è composto
 -- solo dei numeri in una lista gs
compositeOf   gs   1    = True
compositeOf   []   n    = False
compositeOf gs@(g:tgs) n
   | n `mod` g == 0   = compositeOf gs (n `div` g)
   | otherwise        = compositeOf tgs n

hamming gs = filter (compositOf gs) [1..]
```

Hamming Numbers (2)

Cominciamo col caso semplice di generare i composti di soli due numeri p e q e vediamo una soluzione “artigianale”.

Supponiamo di aver generato un certo numero di composti $h_1 \ldots h_n$ Chi è il prossimo? Supponendo che gli ultimi inseriti siano $p \cdot h_i$ e $q \cdot h_j$ ($i, j<n$), il prossimo sarà il minore tra $p \cdot h_{i+1}$ e $q \cdot h_{j+1}$

Scriviamo il generatore `nextH`. Ovviamente non abbiamo bisogno degli indici ma solo di sapere dove sono arrivato…

Scriviamo ora la funzione hamming **in stile circolare**.

```
nextH p q (hp:hps) (hq:hqs)
   | nextp < nextq = nextp:nextH p q hps (hq:hqs)
   | nextp > nextq = nextq:nextH p q (hp:hps) hqs
   | otherwise     = nextp:nextH p q  hps hqs
  where nextp=p*hp
        nextq=q*hq

hamming p q = hs where hs = 1:nextH p q hs hs
```

Hamming Numbers (3)

Facciamo un po' di pulizia: effettivamente moltiplico sempre gli elementi del primo stream per p e quelli del secondo stream per q: posso farlo subito con un `map (p*)` e un `map (q*)`.

A questo punto però, nextH **non dipende più da** p **e** q e quindi posso togliere due parametri.

```
nextH (hp:hps) (hq:hqs)
   | hp < hq    = hp:nextH hps (hq:hqs)
   | hp > hq    = hq:nextH (hp:hps) hqs
   | otherwise  = hp:nextH hps hqs

hamming p q = hs
    where hs = 1:nextH (map (p*) hs) (map (q*) hs)
```

Hamming Numbers (4)

Questa idea si può tuttavia generalizzare: se ho *n* generatori:

1. li posso mettere su una lista `gs` di generatori
2. generare una lista di streams (di prodotti con tutti gli elementi di `gs`)
3. ogni volta scegliere il minimo tra le teste
4. eliminare tutte le occorrenze del minimo scelto dagli stream

```
hamming gs = hs where
   hs = 1:nextH map (\x->map (x*) hs) gs

   nextH xs = m:nextH ys where
      m  = foldr1 min (map head xs)
      ys = map (\zs@(z:tzs)->
                 if z==m then tzs else zs) xs
```

generiamo le |gs| liste dalla lista risultato hs

calcoliamo il minimo delle teste

rimuoviamo i minimi

Hamming Numbers (5)

Immaginando di avere la lista risultato, il *prossimo numero sarà il minimo delle teste delle liste ottenute mappando (p*) nella lista risultato.*

Immaginiamo che 6 sia l’ultimo elemento inserito in `hs`:

```
hs          = 1  2  3  4  5  6    8    9 10 12 15
map (2*) hs = 2  4  6  8 10 12   16   18 20 24 30
map (3*) hs = 3  6  9 12 15 18   24   27 30 36 45
map (5*) hs = 5 10 15 20 25 30   40   45 50 60 75
```

input pronto **input non pronto**

Tuttavia, un vero programmatore Haskell ragiona piuttosto in termini di equazioni ricorsive…

Hamming Numbers (6)

Pensiamo a una definizione induttiva dell'insieme degli Hamming H. Vediamo il caso di `soli' 2 numeri p ed q:

$$H = \{ 1 \} \cup \{ p \cdot h \mid h \in H \} \cup \{ q \cdot h \mid h \in H \}$$

che è una **proprietà di chiusura** e praticamente si può scrivere in Haskell, ricordando che lavoriamo con liste ordinate, e vogliamo **evitare duplicati** e **produrre una lista** ordinata.

Osservate che, come nel caso artigianale, si consuma al più un elemento per ogni lista per produrre un nuovo elemento della lista risultato!

```
 -- definiamo prima union (merge senza duplicati)
union xs@(x:txs) ys@(y:tys)
      | x < y        = x:union txs ys
      | x > y        = y:union xs tys
      | otherwise    = x:union txs tys

hamming p q = hs where
   hs = 1:union (map (p*) hs)(map (q*) hs)
```

Hamming Numbers (7)

Problema: Proviamo a generare tutti i composti di un insieme G di generatori. Come prima avremo:

$$H = \{1\} \cup \bigcup_{g \in G} \{ g \cdot n \mid n \in H \}$$

che non è diverso da prima, ricordando che possiamo \`foldare' la funzione union su una lista di liste e mappare l'operazione `(g*)` dentro la stessa lista di liste…

Nota: `foldr1` evita di dover dare un valore iniziale.

Attenzione invece che `foldl` non termina mai su stream!

```
union xs@(x:txs) ys@(y:tys)
      | x < y       = x:union txs ys
      | x > y       = y:union xs tys
      | otherwise   = x:union txs tys

hamming gs = hs where
   hs = 1:foldr1 union (map (\x->map (x*) hs) gs)
-- hs = 1:foldr union [] (map (\x->map (x*) hs) gs)
```

Lezione 11c

Ulam numbers

Ulam Numbers: Defin. e CodeGolf

Problema: I **numeri di Ulam** $\{u_n\}_{n \in \mathbb{N}}$ sono definiti come segue:

u_1=1, u_2=2, mentre u_{n+1} è il minimo numero che si scrive in **modo unico** come somma di due elementi **distinti** in $\{u_1, \ldots, u_n\}$.

La sequenza generata è: 1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26, 28, 36, …

Ad esempio, 5 **non è un Ulam number** perché 5 = 1+4 = 2+3 e i numeri 1, 2, 3, 4 sono tutti Ulam numbers.

Cominciamo con un virtuosismo: esiste un programma Haskell di soli 67 caratteri, dalla **codeGolf community**:

Haskell, ~~70~~ 67 characters

```
u n=take n$1:2:[x|x<-[1..],[_]<-[[y|y<-u$n-1,z<-u$n-1,y<z,y+z==x]]]
```

Questo programma seleziona gli Ulam numbers tra tutti i naturali, scegliendo quegli x tali che la lista delle somme x=y+z ha lunghezza 1, con y e z presi dagli Ulam già calcolati.

Ulam Numbers: from OEIS

Andando sulla pagina dell'OEIS (On-line Encyclopedia of Integer Sequences) troviamo un altro oscuro programma per generare gli Ulam numbers:

```
(Haskell)
a002858 n = a002858_list !! (n-1)
a002858_list = 1 : 2 : ulam 2 2 a002858_list
ulam :: Int -> Integer -> [Integer] -> [Integer]
ulam n u us = u' : ulam (n + 1) u' us where
   u' = f 0 (u+1) us'
   f 2 z _                           = f 0 (z + 1) us'
   f e z (v:vs) | z - v <= v         = if e == 1 then z else f 0 (z + 1) us'
                | z - v `elem` us'   = f (e + 1) z vs
                | otherwise          = f e z vs
   us' = take n us
-- Reinhard Zumkeller, Nov 03 2011
```

che filtra i naturali, contando per ogni n il numero di tutte le somme di Ulam number precedenti che danno n come risultato, e eleggendo n come prossimo Ulam number se e solo se questo numero è 1.

Cerchiamo soluzioni **più eleganti** e **più efficienti**.

Generazione efficiente del next Ulam

Avendo i primi n Ulam numbers in ordine crescente (come è naturale avere) e un candidato $m > u_n$, possiamo contare in **tempo lineare** in n quante coppie $u_i + u_j = m$.

Infatti, sapendo che $u_i + u_j < m$ **possiamo escludere** tutte le somme del tipo $u_i + u_k$ con $k < j$, perché essendo gli elementi ordinati in ordine crescente, $u_i + u_k < u_i + u_j < m$.

Analogamente, sapendo che $u_i + u_j > m$ **possiamo escludere** tutte le somme del tipo $u_k + u_j$ con $k > i$, perché essendo gli elementi ordinati in ordine crescente, $u_k + u_i > u_i + u_j > m$.

Ma come implementare questa idea in Haskell dove **non abbiamo né vettori**, **né liste doppiamente concatenate** che si possono scorrere sia da sinistra a destra che da destra a sinistra?

Idea: rovesciare la lista degli Ulam

Avendo lo **stream `us` degli Ulam in costruzione** e la **lista rovesciata degli Ulam generati finora `rus`**, possiamo simulare il precedete procedimento: **avanzare su `us` significa andare avanti**, mentre **avanzare su `rus` significa andare indietro**.

Definiamo una funzione `isUlam` che segue questo principio avalendosi di una funzione ausiliaria `countSums m us rs`, sotto la precondizione che `rs` sia il reverse di `us`.

```
isUlam m us = countSums m us (reverse us)==1 where
   countSums m us@(u:tus) rs@(r:trs)
  -- us contiene i primi n<m ulam numbers
  -- rs = reverse us
      | u >= r     = 0 -- poi genero somme simmetriche
      | m == s     = 1 + countSums m tus trs
      | m  < s     = countSums m us trs
             -- avanzo su rs
      | otherwise = countSums m tus rs where
             -- avanzo su us
   s = u + r
```

Tentazioni & soluzioni

A questo punto, uno sarebbe tentato di scrivere:

```
ulams = 1:2:[x | x<-[3..], isUlam x ulams]
```

Purtroppo, **`isUlam` vuole una lista finita** e si potrebbe provare:

```
ulams = 1:2:[x | x<-[3..],
             isUlam x (takeWhile (<x) ulams)]
```

ma purtroppo `takeWhile` **si arresta quando trova un numero maggiore o uguale a x** e questo non succede perché x è maggiore di tutti gli Ulam generati finora.

Occorre conoscere il numero degli ulam numbers già generati.

Definiamo una funzione nextUlams che **tiene abbastanza informazione nei parametri**.

Numero di Ulam numbers generati finora

```
nextUlams us n m =
  if isUlam m (take n us)
     then m:nextUlams us (n+1) (m+1)
     else nextUlams us n (m+1)

ulams = 1:2:nextUlams ulams 2 3
```

Prossimo candidato

Raffinamenti

La soluzione precedente ha numerose inefficienze:

- deve sempre **estrarre i primi *n* elementi** dallo stream `ulams`,
- ad ogni iterazione **deve rovesciarli,**
- la funzione `countSums` conta il numero di somme, ma **arrivati a 2 potrebbe arrestarsi** decretando che il numero in questione non è un numero di Ulam.

Possiamo evitare tutto questo **aggiungendo informazioni sui parametri**!

- Trasmettiamo **`in avanti' il numero di somme già trovate**,
- **manteniamo una lista rovesciata** degli Ulam già trovati,
- Non occorre estrarre i primi n elementi dallo stream degli ulam perché la condizione `r<=u` **verrà soddisfatta prima di andare oltre l'ultimo Ulam già calcolato**.

Raffinamenti

```
nextUlams n us rs
   | isUlam n 0 us rs = n: nextUlams (n+1) us (n:rs)
   | otherwise          = nextUlams (n+1) us rs
 where
   isUlam n 2 _ _ = False
   isUlam n k us@(u:tus) rs@(r:trs) =
      | r<=u      = k == 1
      | s==n      = isUlam n (k+1) tus trs
      | s <n      = isUlam n   k   tus rs
      | otherwise = isUlam n   k   us trs
    where s = u + r

ulams = 1:2:nextUlams 3 ulams [2,1]
```

Trasmettiamo in avanti il numero di somme già trovate

manteniamo gli Ulam noti rovesciati

Generatori di Ulam Numbers

Abbiamo seguito l'idea tipica dei crivelli per generare i numeri primi: filtriamo i numeri di Ulam partendo dai naturali.

Ma i numeri di Ulam sono necessariamente somme di numeri di Ulam precedenti, mentre **molti numeri non sono somma di alcuna coppia di Ulam numbers**, essi formano la successione:

$$v = 23, 25, 33, 35, 43, 45, 67, 92, 94, 96, \ldots$$

(asintoticamente sono **molti di più degli Ulam numbers** $u_n \approx 13.5n$ mentre $v_n \approx 2.5n$).

Avendo uno stream infinito, possiamo facilmente generare lo stream di streams contenente le somma del primo elemento con i successivi, le somme del secondo con tutti i successivi etc.

Ma come usarlo?

```
-- allSums :: Num a => [a]->[[a]]
allSums (x:xs) = map (x+) xs:allSums xs
```

Generatori circolari?

```
ulam    1  2  3  4  6  8 11 13 16 18 ...
( 1+)   3  4  5  7  9 12 14 17 19 27 ...
( 2+)   5  6  8 10 13 15 18 20 28 30 ...
( 3+)   7  9 11 14 16 19 21 29 31 39 ...
( 4+)  10 12 15 17 20 22 30 32 40 42 ...
( 6+)  14 17 19 22 24 32 34 42 44 53 ...
( 8+)  19 21 24 26 34 36 44 46 55 56 ...
(11+)  24 27 29 37 39 47 49 58 59 64 ...
(13+)  29 31 39 41 49 51 60 61 66 70 ...
(16+)  34 42 44 52 54 63 64 69 73 78 ...
       ...
```

Se avessimo gli Ulam numbers già pronti, questo sarebbe lo stream di stream. Un'idea potrebbe essere quella di **fondere questi stream** (ad es. qualcosa del tipo **`foldr1 merge`**) e scegliere solo i numeri che appaiono 1 volta solta.

Purtroppo, se ho appena generato l'**8**, scarto facilmente il 9 e il 10 (ho già due occorrenze), ma non sono mai nella condizione di dire che l'**11** sia il prossimo perché la merge vorrebbe andare a \`vedere' cosa c'è dopo il **9** e il **10**… **ma non è ancora disponibile.**

Lezione 11

That's all Folks…

Grazie per l'attenzione…

…Domande?