Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

Perle di Laziness II

Primi, primo amore

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

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **12**, 15 aprile 2021

Lezione 12a

Riscaldamento su Liste Infinite

Stream di Stream Infiniti

Definiamo la lista di tutte le potenze (la lista dei quadrati, dei cubi, etc.)

Ovviamente, **quanto scritto sotto non è l'output che si ottiene**, in quanto l'interprete non finirà mai di stampare la prima lista, cioè quella dei quadrati!

Analogamente, avrei che `concat powertable = powertable !! 1` perché foldando `(++)` non finirei mai l'append della prima lista (che fa ricorsione sul primo parametro.

Ma sarebbe possibile stampare e/o produrre una lista di tutte le potenze?

Il problema è un po' più generale concerne la fairness del mescolamento di stream di streams

```
powerTable = [[m^n | m<-[1..]] | n<-[2..]]
> powertable
[[1, 4, 9 , 16, 25, …],
 [1, 8, 27, 64, 125, …],
 [1,16, 81, 256,625, …]
  …]
> take 11 (concat powertable)
[1, 4, 9 , 16, 25, 36, 49, 64, 81, 100, 121]
```

How to mingle streams

[Richard Bird, JFP **25**, 2015]

Come "mescolare" uno stream di streams in un unico stream in modo che tutti gli elementi appaiano nello stream risultato dopo un numero finito di elementi?

Possiamo considerare il funzionale `interleave`, definito nella libreria Haskell sugli stream.

Ci sono metodi più fair? Ad esempio in modo che il primo elemento dell'n-esimo stream compaia prima della posizione 2^n?

```
interleave :: [a] -> [a] -> [a]
interleave (x:xs) ys = x : interleave ys xs

>take 15 (foldr1 interleave (map tail powerTable))
[4,8,9,16,16,27,25,32,36,64,49,81,64,125,81]
-- osservate: metà (8) sono quadrati, un quarto (4) sono
-- cubi, un ottavo (2) sono alla quarta, uno alla quinta
```

Diagonali (1)

Un'idea è linearizzare lo stream di stream lungo le diagonali.

Quest'idea è reminescente di come **Cantor ha "numerato" i razionali** e trova molte applicazioni in Teoria della Calcolabilità.

In questo modo, il primo elemento dell'n-esimo stream apparirà dopo solo $n^2/2$ elementi.

Ma come definire le diagonali? **Idea1**: partire col primo stream, prendere il primo elemento, aggiugere uno stream, prendere i 2 primi elementi, aggiungere uno stream…

Osservate che `diags powertable` è uno **stream di liste finite**!

```
crop :: [[a]]->[[a]]->[[a]]
crop xss (ys:yss) =
   map head xss:crop (ys:map tail xss) yss

diags :: [[a]] -> [[a]]
diags (xs:xss) = crop [xs] xss
-- diags: Stream (Stream a) -> Stream [a]
```

Diagonali (2)

Possiamo seguire un'altra idea: scriviamo le diagonali:

$$x_{1,1}$$
$$x_{1,2}\ \ x_{2,1}$$
$$x_{1,3}\ \ x_{2,2}\ \ x_{3,1}$$

Se aggiungo una nuova riga, diciamo $x_{0,1}\ \ x_{0,2}\ \ x_{0,3}$ come cambiano le diagonali?

$$\mathbf{x_{0,1}}$$
$$\mathbf{x_{0,2}}\ \ x_{1,1}$$
$$\mathbf{x_{0,3}}\ \ x_{1,2}\ \ x_{2,1}$$
$$\mathbf{x_{0,4}}\ \ x_{1,3}\ \ x_{2,2}\ \ x_{3,1}$$

La prima diagonale è il primo elemento della prima riga, e poi le altre diagonali sono come prima, ma con in testa un elemento della prima riga. Quanto basta per un'equazione ricorsiva produttiva!

```
diags :: [[a]] -> [[a]]
diags ((x:xs):xss) = [x]:zipWith (:) xs (diags xss)
-- oppure
diags (xs:xss) = zipWith (:) xs ([]:diags xss)
```

Diagonali (3)

Possiamo seguire una terza idea: prendiamo il primo elemento:

$$\begin{array}{llll} x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \cdots \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \cdots \\ x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \cdots \\ x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} \cdots \end{array}$$

Spalmiamo il resto della prima riga in testa alle righe seguenti e proseguiamo in questo modo:

$$\begin{array}{llll} x_{1,2} & x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \cdots \\ x_{1,3} & x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} \cdots \\ x_{1,4} & x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} \cdots \end{array}$$

Per la verità, questo procedimento risulta più costoso. Quante operazioni vengono eseguite per estrarre l'n-esimo elemento?

In *How to Mingle Streams*, Richard Bird dimostra che queste 3 definizioni coincidono per **coinduzione** (un principio di dimostrazione tipico per oggetti infiniti)

```
mingle :: [[a]]->[a]
mingle ((x:xs):xss) = x : mingle (zipWith (:) xs xss)
```

E lo stream delle potenze ordinate?

Per produrre lo stream di tutte le potenze ordinate (maggiori di 1), con (o senza) ripetizioni, parrebbe sufficiente foldare dentro `powerTable` la funzione `merge` (o la funzione `union`).

Sfortunatamente questo processo **non è produttivo**: la sequenza di `merge` o di `union` non riuscirebbe **mai a stabilire chi sia il primo elemento**.

D'altra parte noi sappiamo che ognuno degli stream successivi **comincia con un numero più grande degli stream precedenti**. Possiamo innescare la produzione del risultato utilizzando questa conoscenza con queste due varianti di `merge` e `union`.

```
> take 20 (foldr1 union (map tail powertable))
^CInterrupted.

unionP (x:xs) ys = x : union xs ys
mergeP (x:xs) ys = x : merge xs ys

> take 20 (foldr1 mergeP (map tail powertable))
[4,8,9,16,16,25,27,32,36,49,64,64,64,81,81,100,121,12
5,128,144]
> take 20 (foldr1 unionP (map tail powertable))
[4,8,9,16,25,27,32,36,49,64,81,100,121,125,128,144,16
9,196,216,225]
```

Lezione 12b

Primi, primo amore

La notizia della settimana

CORRIERE DELLA SERA / SCIENZE

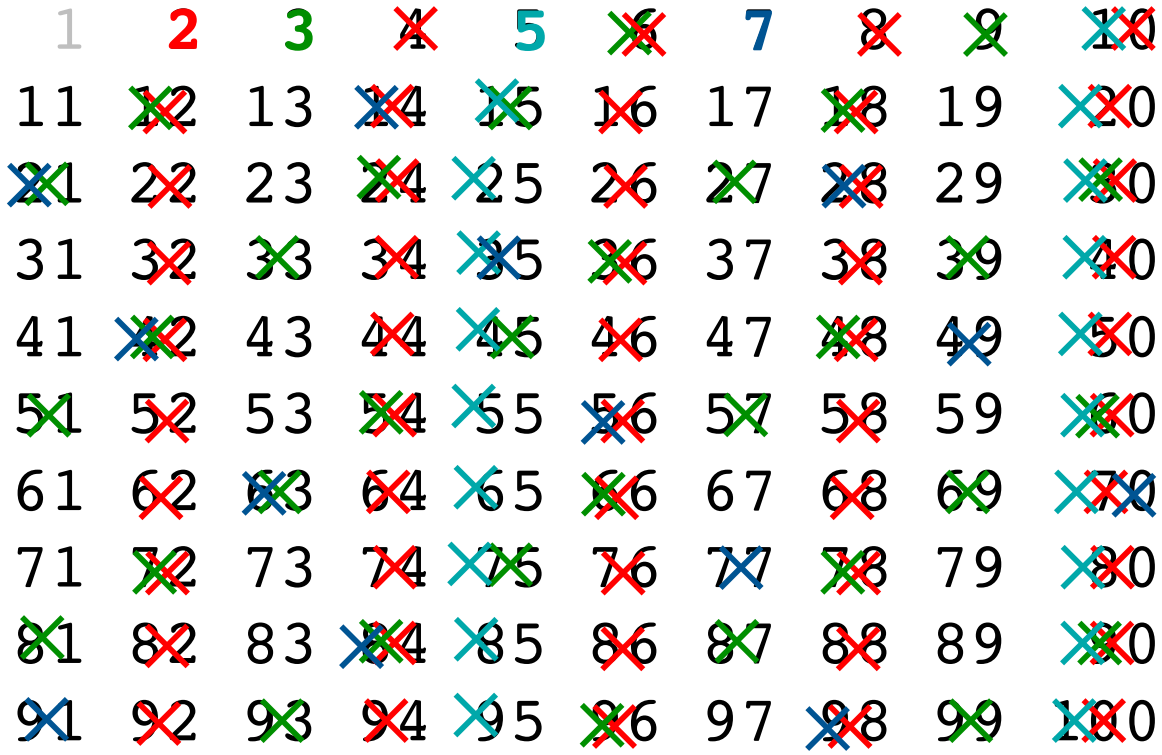
MATEMATICA

Numero primo record: il più grande ha oltre 23 milioni di cifre

Superato di un milione di cifre il più alto precedente, che risaliva a due anni fa. Per scriverlo occorrerebbe un libro di 9 mila pagine

di **Paolo Virtuani**

Crivelli di Eratostene e di Eulero



Si cancellano tutti i multipli del primo numero cancellato.
Ci si ferma a $\sqrt{n}$.
Si fa molto lavoro inutile: per esempio il 30 si cancella 3 volte, come multiplo di 2, 3 e 5.
Il Crivello di Eulero cerca di evitare questo lavoro inutile.

Crivello di Eratostene in Haskell

La definizione di crivello di Eratostene si può tradurre in modo immediato in Haskell, senza occuparci di particolari codifiche.

Calcola una lista di liste, la cui prima è [2..], la seconda sono i dispari [3, 5..], la terza [5,7,11,13,15,…] i non divisibili per 2 e 3…

Osserviamo subito un beneficio di avere una lista infinita: possiamo interrogarla in molti modi senza modificare il programma.

Ad esempio possiamo prendere i primi 100 primi con:

```
take 100 primes
```

oppure i primi minori di 1000 con

```
takeWhile (<1000) primes
```

```
primes = map head (iterate sieve [2..]) where
  sieve (p:ps) = [x | x<-ps, x `mod` p /= 0]
```

Eratostene: un po' di manipolazioni 1

La definizione precedente coinvolge una **lista infinita di liste infinite**. Benché la lazy evaluation assicuri di non dover calcolare nulla di più della testa di ciascuna di quelle liste, la cosa può sembrare un po' innaturale.

Facciamo un po' di manipolazioni algebriche. Riscriviamo in:

```
primes    = rsieves [2..]
rsieve xs = map head (iterate sieve xs)
```

Istanziando `xs` con `p:ps` otteniamo dalla seconda:

```
rsieve (p:ps) = map head (iterate sieve (p:ps))
```

Usando la definizione di `iterate` nella parte destra:

```
map head ((p:ps):iterate sieve (sieve (p:ps)))
```

semplificando `map` e `head`:

```
p:map head (iterate sieve (sieve (p:ps)))
```

usando la definizione di `sieve`, otteniamo:

```
p:map head (iterate sieve [x|x<-ps, x mod p/=0])
```

Eratostene: un po' di manipolazioni 2

Dall'ultima espressione della slide precedente:

```
p:map head (iterate sieve [x|x<-ps, x mod p/=0])
```

abbiamo che abbiamo un'altra istanza di `rsieve`. Per cui:

```
p: rsieve [x | x<-ps, x mod p/=0]
```

Morale: partendo dalla definizione di rsieve, ne abbiamo trovato un'altra, motivo per cui possiamo raccogliere tutto in un'unica equazione ricorsiva:

```
rsieve (p:ps)=p:rsieve [x | x<-xs, x mod p/=0]
```

che può essere una nuova definizione di rsieve. Questa definizione è per ricorsione esplicita e non usa liste di liste infinite.

Abbiamo riscoperto un celebre programma:

```
primes = filterP [2..] where
  filterP (p:ps) = p:filterP [x | x<-xs, x `mod` p /= 0]
```

The `unfaithful' sieve of Eratosthenes

Trattasi di un programma molto famoso. Tutt'oggi sulla Home page ufficiale di Haskell (`https://www.haskell.org`).

È dovuto a **Turner** (**1975**) ed ha influenzato la progettazione stessa di Haskell (laziness, list comprehension etc.).

Personalmente, fatico a considerare `crivello di Eratostene' qualcosa che usa il `mod`. Cmq, lo **stream risultato passa successivi filtri per ogni nuovo primo trovato**, come nell'originale greco.

In principio preferirei il seguente:

```
primes = filterP' [2..] where
  filterP (p:ps) = p:filterP' ps \\ iterate (+p) (p*p)
```

Trial Division

È sicuramente l'algoritmo funzionale **più efficiente** tra quelli elementari basati sul **filtraggio**.

L'idea consiste nel **testare la divisibilità di ogni naturale per i primi già trovati**. In Haskell, tutto ciò è estremamente semplice e non richiede particolari progettazioni di strutture dati, oppure uso di indici per scorrere array etc.

Osservate come lo stream primes viene usato per andare a prendere i divisori da testare…

```
-- definizione del funzionale predefinito all
myAll f (x:xs) = if f x then myAll f xs else False
myAll f []     = True
--
primes = 2:[x | x<-[3..], isPrime x] where
   isPrime x = all (\p-> x `mod` p /= 0) (factorsToTry x)
   factorsToTry x = takeWhile (\p->p*p <= x) primes
```

Crivello di Eratostene di Richard Bird

Ritroviamo l'idea del generatore di primi basato sul vedere i **primi come sfondo dei composti**: qui però i composti, in accordo con l'algoritmo di Eratostene, sono visti come:

$$composites = \bigcup_{p \in primes} p \cdot \mathbb{N}_{\geq p}$$

In questo caso, ovviamente, faccio **l'unione di insiemi con intersezione non vuota** in accordo col crivello di Eratostene, dove molti numeri vengono cancellati più volte (ad esempio il 30, come multiplo di 2, 3 e 5).

È un programma estremamente efficiente rispetto a quelli visti finora, anche se, lavorando con streams, memoria e tempo dedicato alla fusione di stream sono molto onerosi rispetto alle versioni imperative.

```
primes = 2:([3..] `minus` cmps where
  cmps = foldr1 unionP [multiples p | p<-primes]
  multiples p = map (p*) [p..]
  unionP (x:xs) ys = x : union xs ys
```

Crivello di Eulero naïve

È un programma **estremamente inefficiente** (a causa di un esagerato nesting di chiamate ricorsive come l'Eratostene con \\). Tuttavia esprime in modo estremamente naturale l'idea di eliminare **una sola volta** ciascun composto, a causa del **suo minimo divisore primo**.

La funzione `eulerSieve` si applica all'insieme dei naturali **ancora potenziali primi** a cui vengono `tolti' tutti i multipli del nuovo primo trovato (non multipli di primi precedenti).

```
-- differenza di insiemi come liste ordinate
-- si assume ys 'contenuta' in xs
minus xs@(x:txs) ys@(y:tys)
  | x==y      = minus txs tys
  | otherwise = x:minus txs ys
-- ss è la lista dei sopravvissuti...
primes = eulerSieve [2..] where
  eulerSieve ss@(p:tss) =
     p : eulerSieve tss `minus` (map (p*) ss)
```

Crivello di Eulero

Questo programma ricalca la struttura del crivello di Eratostene di Richard Bird, ma **caratterizza i composti come l'unione di insiemi disgiunti**: i multipli di 2 (e di primi maggiori di 2), i multipli di 3 (e primi maggiori di 3) etc.

Si basa sulla definizione induttiva dei numeri S_k **sopravvissuti** e dei numeri E_k **cancellati** alla k-esima passata del crivello di Eulero.

$$S_0 = \mathbb{N}_{\geq 2} \qquad E_0 = \varnothing \qquad S_{k+1} = S_k \setminus E_{k+1} \qquad E_{k+1} = p_{k+1} \cdot S_k \,|^{k+1}$$

Ad esempio, E_1 sono i pari maggiori di 2 e S_1 i dispari maggiori uguali di 3, E_2 sono i dispari moltiplicati per 3 e quindi S_2 non contiene numeri divisibili per 2 e 3. Alla fine:

$$primes = \mathbb{N}_{\geq 2} \setminus \bigcup_{k \in \mathbb{N}} E_k$$

Osserviamo che `eulerSieve` **calcolava**: $primes = \bigcap_{k \in \mathbb{N}} S_k$

```
primes = 2:([3..] `sMinus` (cmps primes [2..])) where
 cmps (p:ps) ss@(s:tss) = es `unionP` cmps ps ss' where
       es = map (p*) ss     -- i nuovi cancellati
       ss'= tss `minus` es -- i nuovi sopavvissuti
```

Lezione 12c

Numeri Primi e Numeri di Hamming

Hamming Numbers (8)

Gli eleganti programmi per gli Hamming **generano più volte gli stessi numeri** (rimossi dalla `union`). È possibile fare meglio?

Ripensiamo le equazioni ricorsive degli Hamming numbers, **assumendo che i generatori *G* siano primi tra loro.**

Sia $G=\{g\} \cup G'$, l'insieme $H(G)$ contiene tutte le potenze di g moltiplicate per tutti gli $H(G')$. Questo **genera tutti gli elementi una sola volta**, perché gli $H(G')$ non contengono g come fattore.

Attenzione! `allProduct xs ys = [x*y | x<-xs, y<-ys]` qui non funzionerebbe! Non li genera in ordine!

L'ordine è fondamentale, vedi trucco per `unionP`.

```
hamming gs = 1:hamming' gs where
   hamming' (g:gs) = ps `unionP` hs `unionP` allProducts ps hs
   where   ps = g : map (g*) ps -- potenze di g
           hs = hamming' gs -- chiamata ricorsiva
allProducts (x:xs) zs = map (x*) zs `unionP` allProducts xs zs
 -- occorre un trucco per rendere produttiva union
 -- so che il minore è il primo della prima lista… gs ordinata
unionP (x:xs) ys = x:union xs ys
```

Hamming Numbers (8)

Gli eleganti programmi precedenti **generano più volte gli stessi numeri** (rimossi dalla `union`). È possibile fare meglio?

Ripensiamo le equazioni ricorsive degli Hamming numbers, **assumendo che i generatori *G* siano primi tra loro.**

$$H(G) = g \cdot H(G) \cup H(G \setminus \{g\})$$

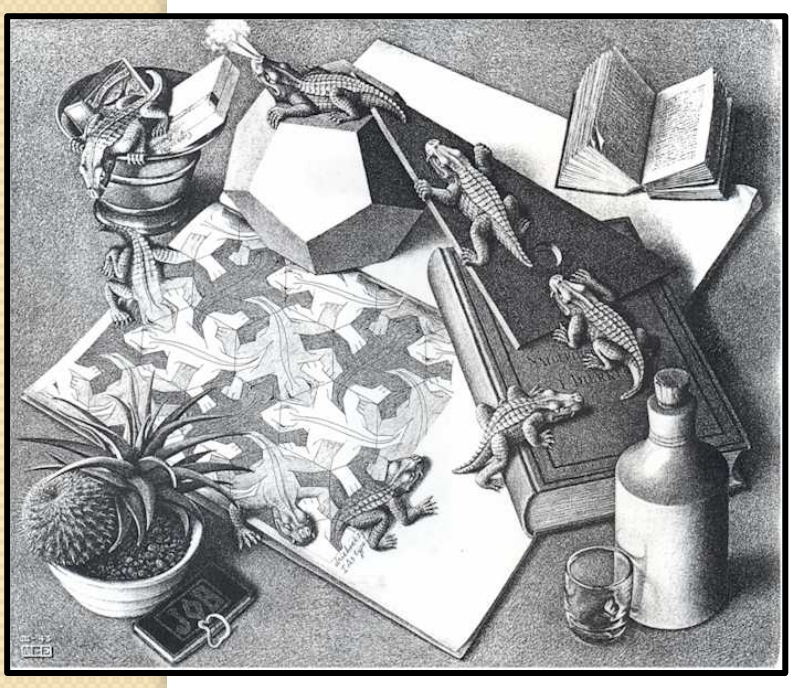
da cui, considerando $H'(G) = H(G) \setminus \{1\}$ ho:

$$H'(G) = \{g\} \cup g \cdot H'(G) \cup H'(G \setminus \{g\})$$

che **sono tutti disgiunti**… (dimostrare!).

```
-- possiamo semplificare union visto che gli stream
-- sono disgiunti: migliora un po' l'efficienza
dUnion xs@(x:txs) ys@(y:tys) =
   if x<y then x:dUnion txs ys
          else y:dUnion xs tys
--
hamming gs = 1:hamming' gs where
   hamming' [] = []
   hamming' (g:gs) = hs where
      hs = [g] dUnion map (g*) hs `dUnion` hamming gs
```

Figura Sfondo: primi e composti



Gli Hammings funzionano **anche per una lista infinita di generatori**.

Idea: Calcolare i **primi** come i Naturali meno gli **hamming dei primi** (che sono ovviamente i composti).

Devo **evitare di produrre tra gli Hamming i generatori stessi**: comincio da `(p*p)`: tuttavia questi mi servono per generare altri composti e li **devo reintrodurre con** `ps `dUnion cmpts`

È una sorta di **Crivello di Eulero**.

```
primes = 2:([3..] `minus` composites primes) where
   composites (p:ps) = cmpts where
      cmpts = (p*p):map (p*) (ps `dUnion` cmpts)
                  `union` (composites ps)
```

Lezione 12

That's all Folks…

Grazie per l'attenzione…

…Domande?