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 **14**, 8 aprile 2022

Lezione 14a

Ricorsione e coricorsione
Induzione e coinduzione

Insiemi (Algebre) Induttivi

Un insieme induttivo *A* viene definito ”partendo **dal basso**” (tipicamente dall’insieme **vuoto**) definendo le regole che permettono di **costruire valori** di tipo *A*.

A sarà il **minimo insieme chiuso** rispetto alle operazioni di costruzione: questo viene espresso usualmente da un ”assioma di minimalità” (ad es., Peano: “Nient’altro è un numero naturale”).

L’insieme A è il **minimo punto fisso** di un’operazione di costruzione.

Gli elementi di *A* saranno il risultato di **un’applicazione finita** dei costruttori, a partire da **costruttori costanti**.

Associato a un insieme induttivo, c’è naturalmente associato un **principio di induzione** che permette di provare proposizioni su A.

Da un punto di vista “informatico”, viene associata una **minima algebra** (che in Teoria delle Categorie è un **oggetto iniziale**) e un principio di **ricorsione** e/o iterazione che permette di definire funzioni $f : A \rightarrow B$.

Insiemi (co-Algebre) co-Induttive

Un insieme co-induttivo A viene definito "partendo **dall'alto**" (ad esempio da un opportuno **universo U**) definendo le regole che permettono di riconoscere **valori** che **non appartengono** ad A.

A sarà il **massimo insieme chiuso** rispetto alle operazioni di "eliminazione" (qui non è garantito si riesca a fare in massimo ω passi – servirebbe una nozione di co-continuità – cfr. Thm. Kleene)

L'insieme A è il **massimo punto fisso** di un'operazione di eliminazione. **Computazionalmente**, gli elementi di $U \setminus A$ saranno riconosciuti dopo un **numero finito** di **applicazioni** dei **distruttori che "scartano" i costruttori di A.**

Associato a un insieme co-induttivo, c'è naturalmente associato un **principio di co-induzione** che permette usualmente di **provare equivalenze** su A.

Da un punto di vista "informatico", viene associata una **massima algebra** (che in Teoria delle Categorie è un **oggetto finale**): come tale, il principio di **co-ricorsione** e/o **co-iterazione** che permette di definire funzioni f : B→A, cioè costruire oggetti di tipo A.

Esempi

Il numeri **naturali** sono il più semplice insieme induttivo (infinito).

Infatti anche tipi come **booleani**, **coppie**, **somme** etc. sono tipi induttivi, ma non hanno costruttori ricorsivi.

Chi sono i principi di induzione, iterazione e ricorsione per questi tipi? Cominciamo con i tipi **non ricorsivi**, per cui di solito "questa storia non viene raccontata":

Per i **booleani** (e le somme, *aka* `Either`) il principi di ricorsione è l'**if-then-else**, e il principio di induzione è l'analisi per casi.

Come faccio a definire una funzione sul tipo **somma**? $f :$ `Either` $A\ B \to C$ viene definita a partire da $g : A \to C$ e $h : B \to C$. A seconda che $x \in$ `Either` $A\ B$ sia nella forma `left`(a) con $a \in A$ oppure della forma `right`(b) con $b \in B$ applico g oppure h.

Per le **coppie** $f : (A, B) \to C$ viene costruita a partire da una funzione $g : A \to B \to C$ che "accede" agli elementi della coppia.

Naturali

Per i numeri naturali il principio di ricorsione è (in Haskell):

```
iter f b 0 = b
iter f b n = f (iter f b (n-1))
```

Ma chi è **iter**? A ben vedere, è il comportamento dei **Numerali di Church**! **iter f b n ≡ n f b**.

Più tipicamente, si considera come principio primo di ricorsione sui naturali uno schema leggermente più generale, chiamato **ricorsione primitiva** (che può essere costruito a partire da **iter** armeggiando con le coppie… ➧ **Esercizio**)

```
rec f b 0 = b
rec f b n = f n (rec f b (n-1))
```

Anche questo definibile in **λ-calcolo tipato semplice**: tuttavia alcune funzioni (come il predecessore) non hanno i tipi che uno si aspetta e di conseguenza non possono essere a loro volta "iterate". La **sottrazione non è computabile nei tipi semplici**!

Per codificare in modo sistematico tutto ciò occorre assumerli (Godel system T) o teorie di tipi più generali, come **Sistema F**.

Liste & Induzione

Vediamo chi è l'iterazione per le liste (finite)

```
iter f b [] = b
iter f b (x:xs) = f x (iter f b xs)
```

Ma chi è **`iter`**? A occhio, direi che è **`foldr`**! Che infatti è il principio di ricorsione generalizzato sulle liste. Sulle liste non sono necessari principi più generali, magari altre forme di iterazione (*aka* **`foldl`**).

So far, so good… tutte cose note… magari note con altro linguaggio.

Un po' meno noto (nella cultura media di uno studente di Informatica del XXI secolo) è che l'**induzione** è la **controparte logica** di **iterazione** e **ricorsione**…

Ma a ben vedere, una **dimostrazione per induzione**, altri non è che una **funzione** che **trasforma una prova** di $P\ 0$ e una prova di $P\ n \Rightarrow P\ n{+}1$ in una prova di $\forall\, n.\ P\ n$.

È una funzione a valori logici: se la **definizione ricorsiva definisce una funzione totale**! Altrimenti è una prova **inconsistente**.

Insiemi coinduttivi

L'insieme coinduttivo che origina dai costruttori dei numeri naturali sono i naturali, con in più un **naturale infinito non-standard** che usualmente si chiama $\omega = S^{\infty}$, cioè che ha un numero infinito di applicazioni del costruttore S (successore).

Sarei tentato di scrivere S^{∞} Z… ma nessuno può andare all'infinito a vedere cosa c'è dopo infinite applicazioni di S 😉

Se tolgo il costruttore Z, ottengo un insieme con un unico elemento, ω appunto, abbastanza poco interessante (potete poi chiedervi chi sono co-iterazione, co-ricorsione e induzione…).

Più interessanti sono le liste (che d'ora in poi chiamerò **Stream**). Ha senso **definire funzioni per ricorsione su Stream**? **NO**, non sarebbero computabili… a meno di non generare altri stream!

Quello che posso fare è definire funzioni che creano stream, di cui in tempo finito, noi potremo vedere una porzione (iniziale) finita.

Non a caso, i tipi induttivi posso vederli come massimi punti fissi, ma anche come **coalgebre finali** (detto volgarmente, le frecce (cioè le funzioni) entrano).

co-iterazione

Ma chi è il principio di computazione che mi permette di generare stream?

Sarà una funzione duale all'iterazione che mi permette di definire una funzione *f*: $A \rightarrow$ `Stream` B. L'abbiamo già vista in Haskell e si chiama **`iterate`**.

Presa una funzione da `f :: a → b` e un valore `x :: b` permette di costruire uno stream come segue:

```
iterate f x = x : iterate f (f x)
```

cioè crea lo stream: **`x : f x : f (f x) : … : f`^{n} **`x : …`**

A volte, per motivi di espressività, è meglio avere un principio un po' più generale di ricorsione, che è chiamato in letteratura **`unfold`** e che sarebbe **la co-ricorsione** (su cui però non c'è altrettanto accordo sulla definizione…):

```
unfold :: (b -> (a, b)) -> b -> Stream a
unfold f y = x : unfold f y' where (x, y') = f y
```

Liste & Strem in Haskell

Se però ho le liste classiche con anche lista vuota, la massima co-algebra **contiene anche le liste finite**. Conviene in tal caso tenerne conto, e magari scrivere una funzione **`unfoldLS`** più generale che costruisce anche liste finite.

```
unfold :: (b -> Bool) -> (b -> (a, b)) -> b -> [a]
unfold p f y = if p y then []
               else x : unfold p f y' where (x, y') = f y
```

Con queste definizioni, possiamo definire funzionali che funzionano/creano sia liste finite che infinite. Facciamo i grandi classici. Osservare che **`map`** e **`zip`** sono equivalenti a quelle note, sia su liste finite che infinite.

```
zip xs ys = unfold p f (xs, ys)  where
        p xs ys = null xs or null ys
        f ((x:xs),(y:ys)) = ((x, y), (xs, ys))

iterate f b = unfold false (\x-> (x, f x)) b

map f xs = unfold null (f . head) tail
```

Co-induzione

Ma qual è la contro-parte logica? È un sistema per derivare equivalenze sull'insieme co-induttivo.

Questa tecnica di prova è stata introdotta con il nome di **bisimulazione** per studiare proprietà dei processi (che spesso sono "computazioni" non-terminanti).

Viene definita così (ancora caso "misto") sulle liste (finite/infinite) di Haskell:

Si definisce una bisimulazione R tra coppie di liste come segue:

$R\ xs\ ys \Rightarrow xs = ys = \bot$ oppure $xs = ys = []$

oppure $\exists v, vs, ws.\ xs = v : vs\ \&\ ys = v : ws\ \&\ vs\ R\ ws$

Dopodichè, definiamo $xs \approx ys$ se **esiste** una bisimulazione R tale che $R\ xs\ ys$.

Questo equivale a scegliere la **massima equivalenza**, definita come un massimo punto fisso.

Esempio di prova per Co-induzione

Vediamo una proprietà ben nota (se ricordate le definizioni alternative dello stream dei numeri naturali o lo stream delle potenze di un numero) che si dimostra "facilmente" per coinduzione:

```
map f (iterate f x) = iterate f (f x)
```

È sufficiente dimostrare che la relazione R che contiene la coppia (anzi le infinite coppie per `f` e `x` di tipi opportuni):

{ `map f (iterate f x), iterate f (f x) | f, x` opportuni}

è una bisimulazione.

Per dimostrare ciò, di nuovo occorre fare ricorso all'algebra dei programmi e "svolgere" le due espressioni.

```
map f (iterate f x) = map f (x: iterate f (f x))
= f x: map f (iterate f (f x))
```

e anche:

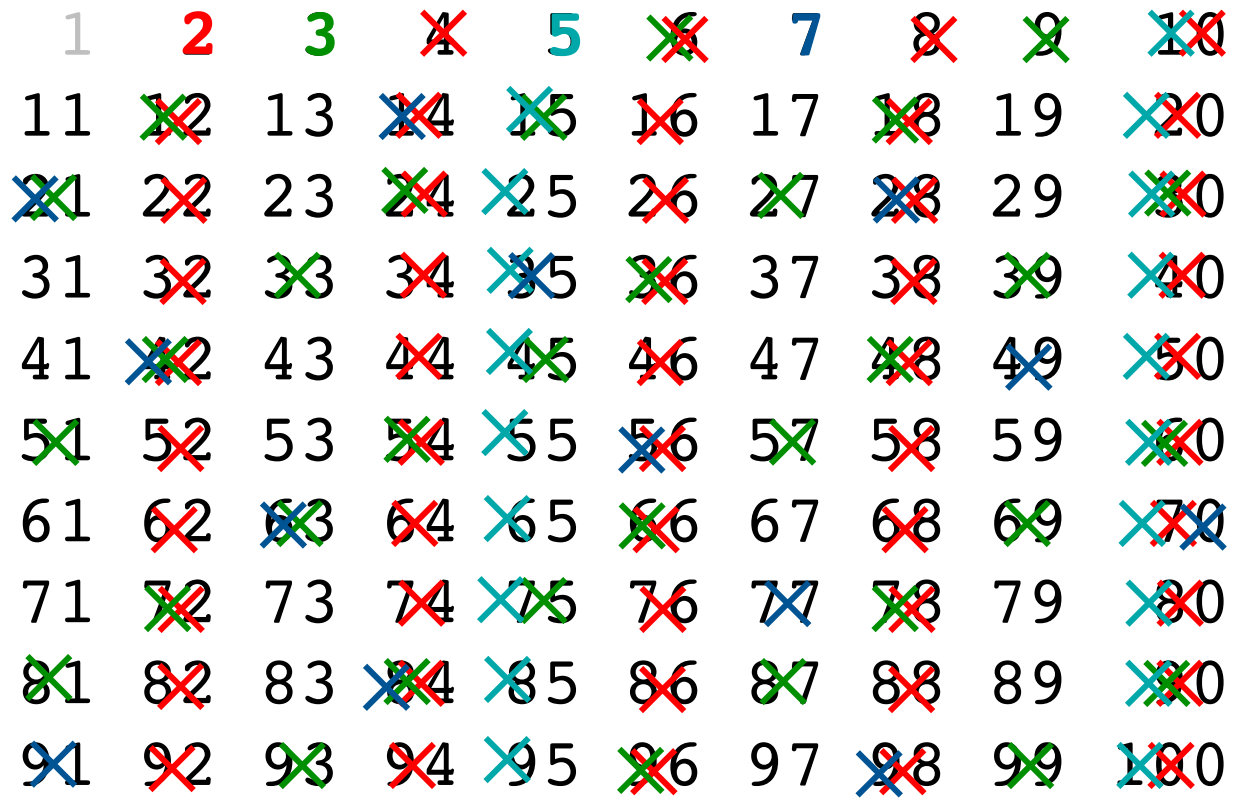
```
iterate f (f x) = f x : iterate f  (f (f x))
```

Le due espressioni hanno la stessa testa (`f x`) e le code stanno in *R* (dove il ruolo di `f` è giocato da `f . f`)

Lezione 14a

Primi: variazioni sul Tema

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** (necessarie in un linguaggio imperativo)

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 numeri 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 laziness 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]
```

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.

Ovviamente non serve cercare divisori di x oltre $\sqrt{x}$.

```
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

Rivediamo 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
```

Lezione 14b

Crivello di Eulero

Crivello di Eulero naïve

È un programma **estremamente inefficiente** (a causa di un esagerato nesting di chiamate ricorsive come l'Eratostene con **`minus`**).

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**).

```
-- minus assumendo ys 'contenuta' in xs:
-- risparmio controlli
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) = --ss non può mai essere deall.
     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
```

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 il solito trucco per rendere produttiva union
unionP (x:xs) ys = x:union xs ys
```

Hamming Numbers (8)

Ci sono modi migliori per generare gli Hamming numbers generalizzati senza produrre duplicati da rimuovere con `union`?

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**, perché genera gli Hamming dei primi una sola volta.

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

Lezione 14

That's all Folks…

Grazie per l'attenzione…

…Domande?