Permutazione degli elementi di una lista

Luca Padovani
padovani@sti.uniurb.it

Sommario

Prendiamo spunto da un esercizio non banale per fare alcune riflessioni su un approccio strutturato alla risoluzione di problemi complessi.

1 Introduzione

È frequente, programmando in un linguaggio funzionale come OCaml [2], dover risolvere problemi che hanno una connotazione "induttiva". Intendiamo, con questo termine, indicare quei problemi la cui soluzione può essere ricondotta alla soluzione dello stesso problema su un input di "dimensione" o "complessità" inferiore.

I termini "dimensione" e "complessità" sono da intendersi in un senso molto ampio, e in effetti la dimensione o la complessità di un input dipendono dal problema stesso. Per esempio, lavorando con numeri interi, la soluzione del problema per un certo numero n può dipendere dalla soluzione per il numero $n-1$. Lavorando con liste, la soluzione del problema per una certa lista $l = [a_1; a_2; \ldots; a_n]$ può dipendere da una sotto-lista di l, quale può essere $[a_2; \ldots; a_n]$, la coda di l.

Esempio (*Fattoriale*) Un semplice esempio di problema su numeri interi che ha una connotazione induttiva è il calcolo del *fattoriale*, che matematicamente è definito come segue:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{se } n > 0 \end{cases}$$

In questo caso, la soluzione del problema per un generico numero n si riconduce alla soluzione del problema per il numero $n-1$. Una volta risolto il problema per $n-1$ (cioè, una volta nota la quantità $(n-1)!$), con una semplice operazione extra (la moltiplicazione di n per $(n-1)!$) si riesce ad ottenere la soluzione del problema per l'input n. □

Esempio (*Lunghezza di una lista*) Un semplice esempio di problema su liste che ha una connotazione induttiva è il calcolo della lunghezza di una lista, che, in notazione matematica, possiamo definire come segue:

$$|l| = \begin{cases} 0 & \text{se } l = [] \\ 1 + |l'| & \text{se } l = x :: l' \end{cases}$$

In questo caso, la soluzione del problema per una generica lista l si riconduce alla soluzione dello stesso problema per la "coda" di l, qui indicata con l', ovvero la sotto-lista ottenuta eliminando il primo elemento di l. Una volta risolto il problema per l', con una semplice operazione extra (la somma di 1 a $|l'|$) si riesce ad ottenere la soluzione del problema per l. □

2 Un problema complesso

Veniamo ora all'esercizio seguente: si vuole scrivere una funzione OCaml, che chiameremo `perm`, che prende in input una lista l di elementi e produce in output una lista con tutte le permutazioni degli elementi di l. Per esempio, la funzione `perm` applicata alla lista $[a_1; a_2; a_3]$ (a_1, a_2 e a_3 indicano tre valori qualsiasi, purché tutti dello stesso tipo) deve ritornare la lista

$$[[a_1; a_2; a_3]; [a_1; a_3; a_2]; [a_2; a_1; a_3]; \\ [a_2; a_3; a_1]; [a_3; a_1; a_2]; [a_3; a_2; a_1]]$$

La prima cosa da capire è se questo particolare problema ha una soluzione induttiva elegante. Qui l'"eleganza" non è una questione di stile: ci possono essere problemi che ammettono una soluzione induttiva, ma tale soluzione è significativamente più complessa di una soluzione *ad-hoc*. In tal caso diremo che la soluzione induttiva non è elegante, e magari preferiremo la soluzione *ad-hoc*.

Per capire se un problema ha una soluzione induttiva elegante, possiamo partire da una istanza specifica del problema, supporre che un istanza più "semplice" derivata da tale istanza specifica sia facilmente risolvibile, e capire come, a

partire dalla soluzione dell'istanza più semplice, sia possibile ottenere (con qualche operazione extra), la soluzione per l'istanza di partenza.

Consideriamo dunque una istanza specifica del problema delle permutazioni: supponiamo di dover calcolare tutte le permutazioni degli elementi nella lista $[1;2;3]$. Prendiamo come istanza più semplice la lista $[2;3]$, che è la coda della lista di partenza, e diamo per scontato di riuscire a risolvere tale sotto-problema.

Ci ritroviamo con la seguente lista:

$$[[2;3];[3;2]]$$

La domanda da porsi è: avendo $[[2;3];[3;2]]$ come soluzione del problema del calcolo delle permutazioni di $[2;3]$, quali sono le operazioni extra che devo fare per ottenere le permutazioni di $[1;2;3]$?

Intuitivamente, quello che devo fare è: considero $[2;3]$; considero l'elemento che non ho considerato nella soluzione del sotto-problema, ovvero 1. Se "inserisco" l'1 in tutte le posizioni possibili di $[2;3]$, ottengo una lista di liste così fatta:

$$[[1;2;3];[2;1;3];[2;3;1]]$$

Queste liste sono caratterizzate dal fatto che gli elementi 2 e 3 compaiono sempre nell'ordine in cui comparivano nella lista $[2;3]$. L'elemento 1 ha occupato in ciascun caso una posizione diversa.

Non ho ancora finito. Il sotto-problema delle permutazioni di $[2;3]$ ha generato un'altra soluzione, ovvero la lista $[3;2]$. Anche in questo caso devo inserire l'1 in tutte le posizioni possibili, ottenendo una lista di liste così fatta:

$$[[1;3;2];[3;1;2];[3;2;1]]$$

Notare che nessuna di queste soluzioni era già stata trovata in precedenza, perché queste liste sono caratterizzate dall'avere il 3 sempre prima del 2.

Ricapitolando:

1. parto da una generica lista $l = x :: m$, fatta da una testa x e una coda m;
2. calcolo le permutazioni di m. Questo calcolo mi produrrà una lista di liste $[n_1; n_2; \ldots; n_k]$ corrispondenti a tutte le permutazioni degli elementi di m;
3. considero *ogni* soluzione del sotto-problema, cioè ogni lista n_i, e inserisco x in *ogni* posizione di n_i. Questa operazione darà origine, per ogni n_i, a una lista di liste che indichiamo con L_i;
4. unisco tutte le soluzioni così trovate per ottenere tutte le permutazioni di l.

Occorre prestare attenzione all'operazione di "unione" a cui si fa riferimento nell'ultimo passaggio: da *ogni* permutazione n_i scaturisce una lista di liste L_i. Se creassi semplicemente una lista contenente queste liste come elementi, otterrei $[L_1; L_2; \ldots; L_k]$ che è una lista di liste di liste! Devo quindi eliminare un livello appiattendo tutte le liste L_i. Questa operazione può essere fatta dalla funzione di libreria `flatten`.

Per completare la soluzione del problema delle permutazioni, non resta altro che definire il caso base del problema, cioè le permutazioni della lista vuota: c'è un solo modo di permutare gli elementi della lista vuota, non permutare nulla! In altri termini, la lista di permutazioni di [] è [[]].

In codice OCaml:

```
let rec perm =
  function
    [] -> [[]]
  | hd::tl ->
      flatten (map (insert_all hd)
                   (perm tl))
```

3 Un problema diverso

A parte `flatten` e `map`, che sono funzioni definite nella libreria standard di OCaml (si veda l'appendice A), non abbiamo però detto come è fatta la funzione `insert_all`. Solo se riusciamo a scrivere la funzione `insert_all` abbiamo una soluzione effettiva del problema di partenza (e una versione funzionante di `perm`!). In questo senso, stiamo procedendo in modo *top-down*, riducendo il problema più complesso (`perm`) in termini di un problema diverso e intuitivamente più semplice (`insert_all`).

Vediamo allora se anche la `insert_all` è risolvibile induttivamente. Ricapitoliamo il problema: dati un elemento x ed una lista $l = [a_1; a_2; \ldots; a_n]$, la funzione `insert_all` deve produrre la lista di liste

$$\begin{array}{l}[[x; a_1; \ldots; a_n]; [a_1; x; a_2; \ldots; a_n];\\ \ldots\\ [a_1; \ldots; x; a_n]; [a_1; \ldots; a_n; x]]\end{array}$$

Ripetiamo lo stesso schema di ragionamento usato prima. Supponiamo di riuscire ad ottenere una soluzione del problema per una sotto-lista di l, diciamo $[a_2; \ldots; a_n]$, ovvero

$$[[x; a_2; \ldots; a_n]; \ldots; [a_2; \ldots; a_n; x]]$$

Quali sono le operazioni extra che devo fare per ottenere una soluzione per l? Tanto per cominciare devo aggiungere l'elemento a_1 in testa a tutte le liste, ottenendo

$$[[a_1; x; a_2; \ldots; a_n]; \ldots; [a_1; a_2; \ldots; a_n; x]]$$

Supponiamo quindi che esista una funzione `prepend_all` che prende un elemento x ed una lista di liste e che fa esattamente ciò. Poi, devo aggiungere la lista in cui x compare come primo elemento, ovvero $[x; a_1; a_2; \ldots; a_n]$.

Abbiamo definito `insert_all`:

```
let rec insert_all x =
  function
    [] -> [[x]]
  | hd::tl ->
      (x::hd::tl)::(prepend_all
                      hd
                      (insert_all x tl))
```

4 Ultimo passaggio

La funzione `insert_all` aveva ancora un "buco", la funzione `prepend_all` di cui, nel corso del nostro ragionamento *top-down*, abbiamo assunto l'esistenza. La libreria standard non fornisce una siffatta funzione, ma codificarla è molto semplice: dato un elemento x e una lista di liste $l = [m_1; \ldots; m_n]$, la funzione `prepend_all` deve ritornare una lista composta dalle m_i, ciascuna aumentata dell'elemento x aggiunto in testa. In codice OCaml:

```
let prepend_all x l =
  map (function m -> x::m) l
```

5 Conclusioni

In ultima analisi, abbiamo definito tre funzioni:

$$\begin{array}{rcl} \texttt{perm} & : & \alpha\ \mathit{list} \to \alpha\ \mathit{list}\ \mathit{list} \\ \texttt{insert\_all} & : & \alpha \to \alpha\ \mathit{list} \to \alpha\ \mathit{list}\ \mathit{list} \\ \texttt{prepend\_all} & : & \alpha \to \alpha\ \mathit{list} \to \alpha\ \mathit{list} \end{array}$$

Operativamente, abbiamo scomposto il problema originario in due modi diversi:

1. abbiamo formulato la soluzione del problema in termini di una soluzione dello stesso problema su un input "più piccolo" (ragionamento *induttivo*);
2. abbiamo definito `perm` usando un'altra funzione, `insert_all`, che risolve un problema diverso, ma più semplice (approccio *top-down*).

Su scala diversa, abbiamo applicato la stessa scomposizione su `insert_all`. Alla fine, ci siamo ricondotti alla scrittura della funzione `prepend_all`, che è sufficientemente semplice da poter essere implementata in modo non ricorsivo, e facendo uso esclusivamente di funzioni della libreria standard.

Sull'approccio *top-down*. Devo risolvere un problema P su un input x. Riesco a ottenere una soluzione $P(x)$ dalla soluzione $Q(x)$ di un problema Q diverso da P, ma che è un po' più semplice di P. L'input y di Q dipende in genere da x.

L'approccio *top-down* non è sempre l'approccio migliore (quello che produce la soluzione più efficiente), ma spesso è quello che produce la soluzione più semplice. In retrospettiva, non era affatto evidente, dal testo del problema originario, che sarebbe servita una funzione come la `prepend_all`. In generale, un eventuale approccio *bottom-up*, che parte da funzioni semplici per costruirne via via di più complesse, deve essere preceduto da un primo prototipo di soluzione ottenuto con l'approccio *top-down*.

Sull'approccio *induttivo*. Devo risolvere un problema P su un input x. Se l'input x è sufficientemente "semplice", so calcolare $P(x)$ immediatamente. Se x non è sufficientemente semplice, riesco a calcolare $P(x)$ a partire dalla soluzione dello stesso problema P su un input più semplice, derivato in qualche modo da x.

Un caso? In tutti gli esempi esaminati in questo documento, il ragionamento induttivo su una funzione che operava su una lista l ci portava sempre a considerare la coda di l come input per la chiamata ricorsiva. Questo non è sempre vero.

Esempio (*Fibonacci*) Tornando ad un esempio con numeri interi, la funzione di Fibonacci è

matematicamente definita in questo modo:

$$\mathit{fibo}(n) = \begin{cases} 1 & \text{se } n \leq 1 \\ \mathit{fibo}(n-1) + \mathit{fibo}(n-2) & \text{se } n > 1 \end{cases}$$

Qui la funzione *fibo* è definita in termini di se stessa sia sull'input $n-1$ che sull'input $n-2$, e $n-2$ non è l'immediato predecessore di n. □

Prendendo spunto dalla funzione di Fibonacci (notare le *due* chiamate ricorsive!) si provi a pensare ad un problema sulle liste risolvibile induttivamente, ma tale per cui il sotto-problema da risolvere non concerne la coda della lista di partenza, ma più sotto-liste della lista di partenza che sono disgiunte tra loro. Suggerimento: pensate ad un algoritmo di ordinamento.

A Funzioni di libreria

Al fine di rendere auto-contenuto questo documento, sono definite in questa appendice le funzioni di libreria utilizzate nella soluzione dei problemi trattati:

```
let rec map f =
  function
    [] -> []
  | hd::tl -> (f hd)::(map f tl)

let rec flatten =
  function
    [] -> []
  | hd::tl -> hd @ flatten tl
```

Non è detto che le omonime funzioni nella libreria standard di OCaml siano definite *esattamente* nello stesso modo. Operativamente, però, queste si possono considerare equivalenti.

Riferimenti bibliografici

[1] "The Caml language", `http://caml.inria.fr/`

[2] X. Leroy et. al., "The Objective Caml system: Documentation and user's manual", `http://caml.inria.fr/ocaml/html man/index.html`

[3] C. Reade, "Elements of Functional Programming", Addison-Wesley, 1989.