Tipi di Dato Ricorsivi

Luca Abeni

April 19, 2017

1 Tipi di Dato

Vari linguaggi di programmazione permettono all'utente di definire nuovi tipi di dato definendo per ogni nuovo tipo l'insieme dei suoi valori e le operazoni che si possono compiere su tali valori.

Un esempio di tipi di dato definiti dall'utente (forse il caso più semplice di tipo definito dall'utente) è costituito dai tipi di dato enumerativi. Per esempio, si può definire un tipo `colore` con valori `rosso`, `blu` e `verde` definendo poi le varie operazioni che si possono compiere su tali valori. Questo modo di definire nuovi tipi di dato rende chiaramente complicata la definizione di un tipo quando il numero di valori è molto alto e rende impossibile definire nuovi tipi di dato quando il numero di valori non è finito. Si può quindi generalizzare il concetto di tipo enumerativo, usando dei *costruttori di dati* invece che semplici valori costanti come `rosso`, `blu` e `verde`. Il nuovo tipo di dati che si va a definire ha quindi valori generati da uno o più costruttori, che operano su uno o più argomenti. E' chiaro che un costruttore che ha un argomento di tipo intero può generare un grande numero di valori (potenzialmente infiniti), risolvendo il problema. Si noti inoltre che i valori `rosso`, `blu` e `verde` di cui sopra altro non sono che casi particolari di costruttori di dati[1].

I valori del nuovo tipo di dato sono quindi partizionati in vari sottoinsiemi, chiamati *varianti*; in altre parole, l'insieme dei possibili valori (insieme di definizione del tipo di dato) è l'unione disgiunta delle varianti del tipo di dato. Ogni variante è quindi associata ad un costruttore, che genera i valori per tale variante; in altre parole, una variante è l'insieme dei valori generati da un singolo costruttore di dato. Notare che se si assume che due costruttori diversi non possano mai generare lo stesso valore, si può dire che l'insieme di definizione del tipo di dato è l'unione (o somma fra insiemi) delle varianti (invece che "unione disgiunta"). Più tardi diventerà chiaro come mai il concetto di "somma" è interessante in questo contesto.

Usando la sintassi di ML, si può quindi definire un nuovo tipo di dato in modo estremamente generico con

```
datatype <name> = <cons1> [arg1] | <cons2> [arg2] | ... | <consn> [argn];
```

dove i vari costruttori `<cons1>`... `<consn>` rappresentano le varianti del tipo. Notare che i costruttori sono vere e proprie funzioni che ricevono un argomento (in ML un costruttore ha un solo argomento; se servono più argomenti, si può usare un argomento di tipo tupla) mappandolo in un valore del nuovo tipo di dato.

Il seguente esempio (basato su ML) mostra come definire un tipo `numero` che può avere valori interi o floating point:

```
datatype numero = intero of int | reale of real;
val sommanumeri = fn (intero a, intero b) => intero (a + b)
                   | (intero a, reale  b) => reale  ((real a) + b)
                   | (reale  a, intero b) => reale  (a + (real b))
                   | (reale  a, reale  b) => reale  (a + b);
val sottrainumeri = fn (intero a, intero b) => intero (a - b)
                     | (intero a, reale  b) => reale  ((real a) - b)
                     | (reale  a, intero b) => reale  (a - (real b))
                     | (reale  a, reale  b) => reale  (a - b);
 ...
```

[1] In particolare, sono costruttori che non ricevono argomenti in ingresso - o hanno 0 argomenti. Quindi, ogni costruttore genera un unico valore, che viene identificato con il nome del costruttore.

2 Ricorsione

La tecnica della *ricorsione* (strettamente legata al concetto matematico di *induzione*) è usata in informatica) per definire un qualche genere di "entità"[2] basata su se stessa. L'esempio tipico riguarda le funzioni ricorsive: si può definire una funzione $f()$ esprimendo il valore di $f(n)$ come funzione di altri valori calcolati da $f()$ (tipicamente, $f(n-1)$). Ma si possono usare tecniche simili anche per definire un insieme descrivendone gli elementi in base ad altri elementi contenuti nell'insieme (esempio: se l'elemento n appartiene all'insieme, anche $f(n)$ appartiene all'insieme).

In generale, le definizioni ricorsive sono date "per casi", vale a dire sono composte da varie clausole. Una di queste è la cosiddetta *base* (detta anche *base induttiva*); esistono poi una o più clausole o *passi induttivi* che permettono di generare/calcolare nuovi elementi a partire da elementi esistenti. La base è una clausola della definizione ricorsiva che non fa riferimento all'"entità" che si sta definendo (per esempio: il fattoriale di 0 è 1, o 0 è un numero naturale, o 1 è un numero primo, etc...) ed ha il compito di porre fine alla ricorsione: senza una base induttiva, si da' origine ad una ricorsione infinita (che non risulta essere troppo utile dal punto di vista pratico).

E' noto come sia possibile usare la ricorsione per definire funzioni (e come la ricorsione sia l'unico modo per ripetere ciclicamente un qualche tipo di operazione in linguaggi di programmazione funzionali - in altre parole, la ricorsione può essere considerata come una sorta di "equivalente funzionale" dell'iterazione). In sostanza, una funzione $f : \mathcal{N} \to \mathcal{X}$ è definibile definendo una funzione $g : \mathcal{N} \times \mathcal{X} \to \mathcal{X}$, un valore $f(0) = a$ ed imponendo che $f(n+1) = g(n, f(n))$.

Più nei dettagli, una funzione è definibile per ricorsione quando ha come dominio l'insieme dei naturali (o un insieme comunque numerabile); il codominio può essere invece un generico insieme $\mathcal{X}$. Come base induttiva, si definisce il valore della funzione per il più piccolo valore facente parte del dominio (per esempio, $f(0) = a$, con $a \in \mathcal{X}$) e come passo induttivo si definisce il valore di $f(n+1)$ in base al valore di $f(n)$; come detto sopra, questo si può fare definendo $f(n+1) = g(n, f(n)))$. Notare che il dominio di $g()$ è l'insieme delle coppie di elementi presi dal dominio e dal codominio di $f()$, mentre il codominio di $g()$ è uguale al codominio di $f()$.

E' importante notare che il concetto matematico di induzione può essere usato per definire non solo funzioni ma anche insiemi, proprietà dei numeri o altro. Per esempio, un insieme può essere definito per induzione indicando uno o più elementi che ne fanno parte (base induttiva) e definendo nuovi elementi dell'insieme a partire da elementi ad esso appartenenti (passo induttivo). Usando questa tecnica l'insieme dei numeri naturali può essere definito in base agli *assiomi di Peano*:

- Base induttiva: 0 è un naturale ($0 \in \mathcal{N}$)
- $n \in \mathcal{N} \Rightarrow n+1 \in \mathcal{N}$ (in altre parole, esiste una funzione $s : \mathcal{N} \to \mathcal{N} - \{0\}$)

3 Tipi di Dati Ricorsivi

Come noto, per definire un tipo di dato occorre definire il suo insieme di definizione (insieme dei valori appartenenti al dato), più le operazioni che si possono applicare ai valori del tipo. Poiché un insieme può essere definito per induzione (come appena visto), si può pensare di applicare la ricorsione per definire nuovi tipi di dato.

Se si ricorda che i valori di un tipo possono essere generati da costruttori che ricevono in ingresso uno o più argomenti, diventa chiaro come applicare la ricorsione ai tipi di dato: un costruttore di un nuovo tipo di dato `T` può avere un argomento di tipo `T`. Chiaramente, per evitare ricorsione infinita occorre una base induttiva; questo vuol dire che una delle varianti del tipo deve avere costruttore che non ha argomenti di tipo `T`. Come passo induttivo, possono invece esistere altre varianti con costruttori che hanno argomenti di tipo `T`.

Per esempio, è possibile definire il tipo `naturale` basato sulla definizione ricorsiva dell'insieme dei naturali fornita da Peano: il tipo sarà costituito da due varianti, una delle quali (la base induttiva) avrà costruttore costante `zero` (che rappresenta il numero 0), mentre l'altra avrà costruttore `successivo`, che genera un naturale a partire da un naturale. Usando il costrutto `datatype` di ML, questo si scriverà

```
datatype naturale = zero | successivo of naturale;
```

[2] Il termine "entità" è qui usato informalmente per indicare genericamente funzioni, insiemi, valori... Ma, come vedremo a breve, anche tipi di dato!

Per definire completamente il tipo `naturale`, dovremo poi definire alcune semplici operazioni sui suoi valori. Per esempio, conversione da `int` a `naturale` (funzione `i2n`), conversione da `naturale` a `int` (funzione `n2i`) e somma di due numeri naturali (funzione `nsum`):

```
val rec i2n = fn 0 => zero
              | x => successivo (i2n (x - 1));

val rec n2i = fn zero          => 0
              | successivo n => 1 + n2i n;

val rec nsum =
     fn zero          => (fn n => n)
     | successivo a => (fn n => successivo (nsum a n));
```

Sebbene interessante dal punto di vista matematico, la definizione del tipo `naturale` non è troppo utile... Esistono però una serie di strutture dati che possono essere definite come tipi ricorsivi e sono molto utili: sitratta di liste, alberi e strutture dati dinamiche di questo genere. Per esempio, una lista di interi è definibile ricorsivamente come segue: una lista è vuota oppure è la concatenazione di un intero ed una lista. Il costruttore "lista vuota" costituisce la base induttiva, mentre il costruttore "concatenazione di intero e lista" costituisce il passo induttivo. Notare che una lista è una struttura dati ricorsiva perché il costruttore "concatenazione di intero e lista" riceve come secondo argomento un dato di tipo lista. Usando la sintassi di Standard ML, una lista di interi è quindi definibile come

```
datatype lista = vuota | cons of (int * lista);
```

dove `vuota` è il costruttore di lista vuota, mentre `cons` genera la variante contente tutte le liste non vuote. Si noti che il costruttore `cons` ha un argomento di tipo `int * lista` (coppia formata da un intero ed una lista), perché una funzione ML non può avere più di un argomento (quindi, si ricorre ad una coppia per raggruppare due argomenti in uno).

4 Liste Immutabili e Non

Il tipo di dato ricorsivo "`lista`" introdotto nella sezione precedente costituisce un ben particolare tipo di lista, detto "lista immutabile". Il perché di questo nome può essere capito considerando le operazioni implementate su questo tipo. Poiché le liste immutabili (e le strutture dati immutabili in genere) sono prevalentemente utilizzate nei linguaggi funzionali, il linguaggio Standard ML verrà usato nei prossimi esempi.

Le prime due operazioni sul tipo `lista` sono i due costruttori `vuota` e `cons`, che generano rispettivamente liste vuote e liste non vuote (vale a dire, concatenazioni di interi e liste). Per poter operare sulle liste immutabili in modo generico, servono altre due operazioni dette `car` e `cdr`. Data una lista non vuota, `car` ritorna il primo intero della lista (la cosiddetta "testa della lista"), mentre `cdr` ritorna la lista ottenuta eliminando il primo elemento. Sia `car` che `cdr` non sono definite per liste vuote. Una semplice implementazione di queste due funzioni è data da:

```
val car = fn cons (v,_) => v;
val cdr = fn cons (_,l) => l;
```

Si noti che nel processare queste definizioni un interprete od un compilatore Standar ML genererà un warning dicendo chi pattern nn sono esaustivi: sia `car` che `cdr` sono infatti definite usando pattern matching su un valore di tipo `lista`, ma l'unico pattern presente nella definizione matcha con liste non vuote (costruttore `cons`)... Non è presente alcun pattern che matchi con liste vuote (costruttore `vuota`). Questo comporta che se `car` o `cdr` è applicata ad una lista vuota si genera un'eccezione (cosa non proprio "puramente funzionale"). Tutto questo riflette il fatto che `car` e `cdr` non sono definite per liste vuote.

Qualsiasi operazione sulle liste è implementabile basandosi solamente su `vuota`, `cons`, `car` e `cdr`. Per esempio, le seguenti funzioni mostrano come controllare se una lista è vuota, calcolarne la lunghezza o concatenare due liste usando solo le primitive appena citate:

```
val isempty = fn vuota => true
              | _      => false;

val rec lunghezza = fn vuota        => 0
                     | cons (_, l) => 1 + lunghezza l;
```

```
val rec concatena = fn vuota        => (fn b => b)
                  | cons (e, l) => (fn b => cons (e, concatena l b));
```

Si noti come molte delle funzioni che agiscono sulle liste sono ricorsive (essendo le liste definite come un tipo di dato ricorsivo, questo non dovrebbe stupire più di troppo).

La funzione `isempty` è molto semplice e ritorna un valore booleano basandosi su pattern matching: se la lista passata come argomento matcha una lista generata da dal costruttore `vuota` (vale a dire, se è una lista vuota), ritorna `true` altrimenti (wildcard pattern) ritorna `false`.

La funzione `lunghezza` lavora in modo ricorsivo, usando il pattern `vuota` come base induttiva (la lunghezza di una lista vuota è 0) e dicendo che se la lunghezza di una lista `l` è n, allora la lunghezza della lista ottenuta inserendo un qualsiasi intero in testa ad `l` è $n + 1$ (passo induttivo).

La funzione `concatena`, infine, usa la concatenazione di una lista vuota con una generica lista `b` come base induttiva (concatenando una lista vuota con una lista `b`, si ottiene `b`). Il passo induttivo è costituito dal fatto che concatenare una lista composta da un intero `n` ed una lista `l` con una lista `b` equivale a creare una lista composta dall'intero `n` seguito dalla concatenazione di `l` e `b`.

Per finire, una funzione per inserire un numero intero in una lista ordinata può essere implementata come segue:

```
val rec inserisci = fn n => fn l =>
    if (l = vuota) orelse (n < car l)
      then
        cons (n, l)
      else
        cons (car l, inserisci n (cdr l))
```

Quando si vuole inserire un nuovo elemento in una lista vuota (`l = vuota`), la lista risultante è creata (tramite `cons`) aggiungendo il nuovo elemento in testa alla lista. Questo viene fatto anche se l'elemento che si vuole inserire è più piccolo del primo elemento della lista (`n < car l`). Altrimenti, viene creata (sempre tramite `cons`) una nuova lista formata dalla testa di `l` (`car l`) seguita dalla lista creata inserendo il numero nel resto di `l` (`inserisci n (cdr l)`). In ogni caso, una lista contenete `n` inserito nella giusta posizione viene creata tramite una o più invocazioni di `cons`, invece di modificare la lista `l`.

Per esempio, si consideri cosa accade quando si invoca inserisci 5 cons(2, cons (4, cons(7, null))): poiché $5 > 2$, `inserisci` invoca cons (2, inserisci 5 cons (4, cons(7, null))), quindi cons (2, cons (4, inserisci 5 cons che finalmente ritorna cons (2, cons (4, cons (5, cons(7, null)))). Sono quindi stati creati ben 3 nuovi valori di tipo `lista`.

Per capire meglio il comportamento di una lista immutabile ed il perché del suo nome, è utile vedere come essa possa essere definita in un linguaggio che non supporta direttamente tipi di dati ricorsivi, come il linguaggio C. In questo caso, si usano dei puntatori per collegare i vari elementi della lista: in particolare, ogni elemento della lista è rappresentato da una struttura composta da un campo intero (il valore di tale elemento) ed un puntatore al prossimo elemento della lista. Quindi, invece di avere un tipo di dato `lista` definito basandosi su se stesso si ha un tipo di dato `lista` definito usando un puntatore a `lista`. I due costruttori `vuota` e `cons` sono implementati come funzioni che ritornano un puntatore a `lista` (queste funzioni allocano quindi dinamicamente la memoria necessaria a contenere una stuttura `lista`) e le funzioni `car` e `cdr` semplicemente ritornano i valori dei campi della struttura `lista`. Una lista è terminata da un elemento che ha puntatore al prossimo elemento uguale a `NULL` (questo si può considerare l'equivalente della base induttiva). Riassumendo, una semplice implementazione in C può apparire in questo modo:

```
struct lista {
  int val;
  struct lista *next;
};

struct lista *vuota(void)
{
  return NULL;
}

struct lista *cons(int v, struct lista *l)
{
```

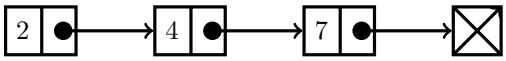

Figure 1: Lista di esempio contenente gli interi 2, 4 e 7.

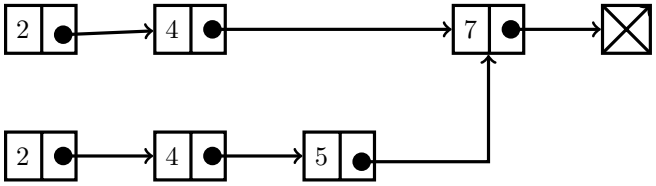

Figure 2: Lista della Figura 1 dopo aver eseguito inserisci (5, l);.

```
  struct lista *res;

  res = malloc(sizeof(struct lista));
  res->val = v;
  res->next = l;

  return res;
}

int car(struct lista *l)
{
  return l->val;
}

struct lista *cdr(struct lista *l)
{
  return l->next;
}
```

Si noti che per semplificare l'implementazione il risultato di `malloc()` non viene controllato (un'implementazione più "robusta" dovrebbe invece controllare che `malloc()` non ritorni `NULL`).

La funzione `inserisci` può quindi essere definita analogamente a quanto fatto in ML, usando solo le funzioni `vuota()`, `cons()`, `car()` e `cdr()` definite qui sopra:

```
struct lista *inserisci(int n, struct lista *l)
{
  return ((l == vuota()) || (n < car(l))) ?
                  cons(n, l) : cons(car(l), inserisci (n, cdr(l)));
}
```

Questa definizione usa il costrutto di *if aritmetico* del linguaggio C, che è equivalente all'espressione `if...then...else...end` di ML ma è forse meno leggibile rispetto ad un costrutto di selezione "tradizionale". La seguente implementazione è equivalente (anche se non strutturata e "meno funzionale"):

```
struct lista *inserisci(int n, struct lista *l)
{
  if ((l == vuota()) || (n < car(l))) {
    return cons(n, l);
  } else {
    return cons(car(l), inserisci (n, cdr(l)));
  }
}
```

Sia ora `l` un puntatore alla lista di Figura 1, contenente i valori 2, 4, e 7 e si consideri cosa avviene quando si invoca l = inserisci (5, l);. Poiché $5 > 2$, `inserisci` invoca ricorsivamente inserisci (5, cdr(l));, con `cdr(l)` che è un puntatore al secondo elemento della lista `l` (è quindi un puntatore ad una lista contenente i valori 4 e 7). Il risultato di questa invocazione ricorsiva di `inserisci()` sarà poi passato a `cons()`

(che allocherà dinamicamente una nuova struttura di tipo `lista`). Poiché `car(cdr(l))` vale 4 e $5 > 4$, inserisci (5, cdr(l)); invocherà ancora ricorsivamente `inserici()`, con parametri 5 e `cdr(cdr(l))`. A questo punto, poiché `car(cdr(cdr(l)))` vale 7 e $5 > 7$, inserisci (5, cdr(cdr(l))); ritornerà `cons(5, cdr(cdr(l))`, allocando dinamincamente una struttura di tipo `lista`. `cons()` sarà poi invocata altre 2 volte, e come risultato `inserisci(5, l);` ritornerà una lista costituita da 3 nuovi elementi dinamicamente allocati (`cons()` è stata invocata 3 volte), senza modificare alcun elemento della lista di Figura 1. Questo è il motivo per cui questo tipo di lista viene detto "immutabile": i campi `val` e `next` della struttura `lista` vengono settati quando la struttura è allocata (da `cons()`) e non vengono mai più variati. Il risultato finale è visibile in Figura 2, che mostra in basso i 3 nuovi elementi allocati dinamicamente (si noti che ora il puntatore `l` punta all'elemento di valore 2 più in basso).

E' anche importante notare che nell'esempio precedente il puntatore al primo elemento di `l` può essere stato "perso", lasciando delle zone di memoria allocate dinamicamente ma non più raggiungibili: se un programma invoca

```
l = vuota();
l = inserisci(4, l);
l = inserisci(2, l);
l = inserisci(7, l);
l = inserisci(5, l);
```

le strutture dinamicamente allocate dalla seconda invocazione di `inserisci()` (vale a dire, `inserisci(2, l);`) non hanno più alcun puntatore che "permtta di raggiungerle".

Tornando all'esempio precedente, quando si invoca `l = inserisci(5, l)` le strutture contenenti i primi due elementi della "vecchia lista" possono quindi essere non più raggiungibili, ma la memoria nelle quali sono memorizzate non è stata rilasciata da alcuna chiamata a `free()`. Questo indica chiaramente che un'implementazione delle liste organizzata in questo modo rischia di generare dei *memory leak*, quindi un qualche tipo di meccanismo di garbage collection si rende necessario.

La precedente implementazione di liste immutabili in C può essere confrontata con un'implementazione "più tradizionale", non basata su strutture dati immutabili ma su variabili modificabili:

```
struct lista {
  int val;
  struct lista *next;
};

struct lista *inserisci(int n, struct lista *l)
{
  struct lista *res, *prev, *new;

  new = malloc(sizeof(struct lista));
  new->val = n;

  res = l;
  prev = NULL;
  while ((l != NULL) && (l->val < n)) {
    prev = l;
    l = l->next;
  }
  new->next = l;
  if (prev) {
    prev->next = new;
  } else {
    res = new;
  }

  return res;
}
```

Si noti che questa implementazione di `inserisci()` modifica il campo `next` dell'elemento precedente a quello che viene inserito, quindi la lista non è immutabile. D'altra parte, non soffre dei problemi di memory leak evidenziati per l'implementazione precedente (in sostanza, questo ci dice che tecniche di garbage

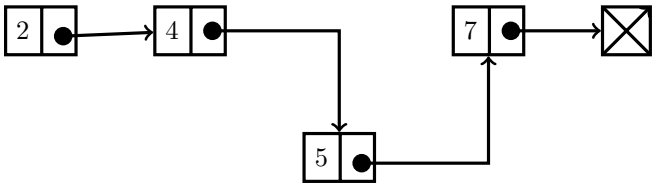

Figure 3: Lista della Figura 1 dopo aver eseguito inserisci (5, l); usando le liste "tradizionali" (non immutabili).

collection sono necessarie solo se si utilizzano strutture dati immutabili). Il risultato dell'operazione l = inserisci (5, l); usando questa implementazione delle liste è visibile in Figura 3 (si noti che la freccia uscente dall'elemento "4" è cambiata).

Per finire, a titolo di confronto si fornisce anche un'implementazione di una lista immutabile in Java (si ricordi che una VM Java implementa un garbage collector di default, quindi i memory leak dovuti all'utilizzo di strutture dati immutabili non sono un problema):

```
public class List {
    private final Integer val;
    private final List next;

    public List(){
        val=null;
        next=null;
    }
    public List(int v, List l) {
        val = v;
        next = l;
    }
    public int car() {
        return val;
    }
    public List cdr() {
        return next;
    }


    public void stampaLista() {
        if (next!=null) {
            System.out.println(val);
            next.stampaLista();
        }
    }

    public List inserisci(int n){
        if (next==null || n < val)
        return new List(n,this);
        else return new List(car(),cdr().inserisci(n));
    }

    public static void main(String a[]){ // main di test
        List l = new List();
        l = l.inserisci(1);
        l = l.inserisci(5);
        l = l.inserisci(3);
        l = l.inserisci(9);
        l = l.inserisci(7);
        l.stampaLista();
    }
```

}

E' interessante notare come in Java il qualificatore `final` permetta di specificare esplicitamente che i campi della classe non verranno mai modificati (la struttura dati è immutabile). Inoltre, si noti che Java (come molti altri linguaggi orientati agli oggetti) obbliga ad usare lo stesso nome per i costruttori e per la classe. Non ci saranno quindi due costruttori `vuota` e `cons` per le due varianti, ma due costruttori chiamati entrambi `List` che si distinguono per il numero ed il tipo dei propri argomenti (un costruttore - quello di lista vuota - non ha argomenti, mentre l'altro - quello corrispondente a `cons` - ha un argomento di tipo `int` ed uno di tipo `List`).