Estratto da

R. Ferro e A. Zanardo (a cura di), *Atti degli incontri di logica matematica* Volume 3, Siena 8-11 gennaio 1985, Padova 24-27 ottobre 1985, Siena 2-5 aprile 1986.

Disponibile in rete su http://www.ailalogica.it

COSTRUZIONI DI TIPI NEL LINGUAGGIO DI PEANO

STEFANO STEFANI
Siena

INTRODUZIONE

Il grande progresso compiuto dalla microelettronica consente oggi di realizzare sistemi logicamente assai complessi a costi spesso irrisori. Il compito di definire nei minimi dettagli il comportamento di tali sistemi, cosi' da poterli realizzare, e' di conseguenza assai arduo e si presenta frequentemente. L'espediente usato di prodittori di elettronica per superare questo ostacolo, che impedirebbe loro di mettere a frutto la propria abililita' tecnologica, e' stato di produrre, in un numero relativamente piccolo di modelli, dispositivi programmabili, e quindi capaci di essere successivamente piegati a usi diversi. In questo modo la patata bollente e' passata ai produttori di software.
Il genere di applicazioni a cui ci stiamo riferendo, note come "embedded", considera quindi il software al servizio della macchina, e non viceversa. Questo peculiare punto di vista rende opportuna la definizione di linguaggi di programmazione particolarmente adatti alla programmazione embedded. Essi dovranno avere un potere espressivo molto elevato, che consenta di descrivere nel dettaglio il comportamento della macchina, pur vivendo in ambienti implementativi estremamente poveri e tra loro scarsamente omogenei. Sorge in particolare il problema di stabilire una base per la semantica dei dati che, partendo da un numero minimo di assunzioni, consenta la costruzione di tipi in modo adeguatamente flessibile e potente, fornendo cosi' una via per sintetizzare comportamenti anche molto complessi della macchina fisica.
L'ipotesi che assumiamo e' che l'impiego di capaci meccanismi di astrazione formalmente robusti, sia particolarmente utile proprio nel programmare dispositivi che si trovano ad interagire in modo stretto e concreto con l'ambiente esterno.

E' in ogni caso generalmente riconosciuto che la possibilita' di definire un ricco repertorio di tipi, come tutto cio' che contribuisce a rafforzare l'aspetto statico della semantica, e' particolarmente utile per applicazioni di questo genere, in cui la convalida a posteriori e' sovente problematica, ed e' richiesta una notevole efficienza del codice prodotto. Queste preoccupazioni sono riconoscibili anche nella definizione del linguaggio **ADA** , che tuttavia sommerge questo aspetto in un mare di complessita' che, per il supporto che richiedono, lo rendono inadatto quantomeno alla programmazione di piccoli sistemi.
Il linguaggio sperimentale **PEANO** e' stato definito e realizzato dall'autore con lo scopo principale di indagare la validita' di questa assunzione circa la validita' delle tecniche di astrazione. In esso il problema di specificare una semantica per i tipi di dato che sia generale, ma sufficientemente vicina alla natura fisica della macchina di cui si vuole determinare il funzionamento senza pero' dipendere da questa, e' affrontato individuando un unico tipo fondamentale di dati (**urtype**) che possiede simultaneamente la natura di anello aritmetico e di anello booleano. Esso e' inoltre fornito dell'usuale ordinamento e dell'uguaglianza, visti pero' anche questi come operazioni. Ogni altro tipo verra' quindi costruito o derivato, direttamente od indirettamente, da **urtype** (tranne pochi particolari casi riguardanti piu' che altro il controllo delle eccezioni). Una delle coseguenze di operare in modo algebricamente rigoroso e' la rinuncia all'uso dei tipi **access** (o **pointer**). I seguenti paragrafi saranno dedicati ad esporre in qualche dettaglio i meccanismi scelti per le costruzioni dei tipi, cercando in particolare, di vedere come costruire di tipi astratti e ricorsivi senza usare il costruttore **pointer**, e mantenendo un completo controllo sulla strategia di gestione dello spazio.

COSTRUZIONE E DERIVAZIONE DI TIPI

Come si e' detto, alla base di tutte le costruzioni di tipo vi e' in ogni caso **urtype.** Da esso si formano nuovi tipi con l'uso delle primitive di costruzione e con il procedimento della derivazione di sottotipi. Ai costruttori corisponderanno altrettanti selettori, mentre alle derivazioni corrisponderanno l'operazione di inclusione (^) e quelle di restrizione o cast. Una semantica per **urtype** si trasferira' quindi agevolmente su tutti gli altri oggetti. Se un tipo e' definito ricorrendo al solo meccanismo di derivazione (o e' un tipo astratto) e' detto "tipo semplice", altrimenti e'

"tipo composto". I tipi semplici sono quindi direttamente od indirettamente sottotipi di **urtype**. La derivazione di un sottotipo avviene specificando quali operazioni vengono ereditate (restrizione del tipo algebrico) e fornendo un predicato che ne restringe eventualmente il dominio. In questo modo sono predefiniti ad esempio il tipo **integer, boolean e byte.**
I costruttori sono piu' o meno quelli tradizionali. Essi consentono di esprimere il prodotto diretto (**record**), la somma diretta (**sum**), la potenza (**array**), la successione (**stream**), l'enumerazione (**[** ... **]**), le applicazioni (**function** e **form**); e' inoltre presente il meccanismo di astrazione che cercheremo ora di illustrare rapidamente.
Per astrazione sui tipi di dati si intende, per cosi' dire, la possibilita' di separare l'utilizzazione di un tipo di dati dai dettagli e sottigliezze della sua realizzazione. In un certo senso cio' consiste nel separare l'aspetto categoriale di un tipo, ossia le sue proprieta' sotto i morfismi, dalla costruzione di una particolare struttura algebrica che renda sicuramente non vuota tale categoria (e che abbia desiderabili proprieta' effettive). La possibilita' di effettuare una tale separazione diviene oltretutto uno fonte preziosa di modularita'.
In **peano** i tipi astratti sono distinti in finiti o meno in dipendenza del fatto che il numero dei loro elementi (valori) possieda o meno un confine superiore statico, cioe' indipendente dal contesto di esecuzione del programma.
Definire un tipo astratto, dal punto di vista dell'utente, consiste semplicemente nel dichiarare che esso e' astratto e nell'elencare i morfismi di cui e' dotato. Dal punto di vista dell'implementatore si tratta invece di specificare una opportuna strategia implementativa determinando:

i) Un tipo di dati, di solito non astratto, che consenta di "realizzare" il tipo astratto in questione.
ii) I morfismi di cui il tipo astratto e' fornito, descritti tramite applicazioni che agiscono sulla realizzazione che si e' scelta.
iii) Nel caso in cui il tipo astratto non sia finito le procedure **initialize, split, dispose** che determinano la gestione della memoria.

Nel modulo che implementa un tipo astratto gli oggetti che si stanno definendo possono quindi essere considerati sotto un dupplice punto di vista. Il simbolo **@** anteposto ad un nome astratto specifica che ci si vuole riferire alla sua relizzazione.
Conviene notare che per i tipi astratti che non siano finiti, l'assegnazione (**:=**) e l'uguaglianza (**=**) non sono considerate in nessun caso definite implicitamente,

ma sono invece cosiderate rispettivamente una forma ed una funzione che necessitano di una definizione esplicita. Occorrera' ovviamente preoccuparsi di rispettare le regole di prova che le riguardano.
E' facile intuire che rifiutando il costruttore **pointer** sara' necessario trovare comunque una qualche risorsa che consenta di descrivere i tipi ricorsivi. La soluzione proposta consiste nell'ammettere per ogni tipo di dati la possibilita' di tre sorte per denotare gli oggetti: le costanti, le variabili usuali, e gli indicatori (il punto fondamentale e' che gli indicatori non formano un nuovo tipo). Le variabili e gli indicatori sono se si vuole i nomi propri ed i nomi comuni degli oggetti: ogni oggetto ha un unico nome proprio, ma puo' eventualmente avere piu' modi alternativi di essere denotato. Come vedremo, e' necessario ammettere che in un linguaggio imperativo vi sia una controllata quantita' di aliasing , cosi' come e' necessario che la valutazione dipenda dal contesto. La valutazione di una variabile e di un indicatore che denotano un medesimo oggetto sara' identica. In altri termini non esiste la necessita' di un operatore di "dereferenza". La rappresentazione (**@**) di un oggetto astratto e' un'indicatore.
Sugli indicatori sono definite la forma universale (cioe' predefinita in ogni tipo) **let ... == ...** , e la funzione predicato **is ... == ...** . La prima lega un indicatore a sinistra ad un oggetto a destra, dove puo' comparire tanto una variabile quanto un'altro indicatore. La seconda decide se due nomi denotano un identico oggetto. Si tratta quindi di un predicato di identita' che prescinde percio' dalla valutazione. Riformulando la tradizionale distinzione, i parametri delle forme e funzioni sono distinti in posti per espressioni (o posti-valore), e posti per indicatori (o posti-oggetto); mentre le espressioni sono valutate prima del passaggio, sott'intendendo un legame di assegnezione, gli indicatori sono passati non valutati, ed il legame tra variabile formale e parametro sottintende invece la forma **let ... == ...** . E' quindi possibile sostituire una variabile in un posto per indicatore. Ad esempio, le procedure di assegnazione **:=** (il plurale e' dovuto al polimorfismo) avranno come primo parametro un posto oggetto e come secondo un posto valore. Si noti come il processo di valutazione sia in ogni caso attivato dalla necessita' di formare un legame per valore. Le funzioni **is ... == ...** avranno invece due posti-oggetto.
Ovviamente se **is** x == y valuta **true** allora anche x = y valuta **true** .
Nel dominio di un indicatore vi e' sempre l'oggetto vuoto **null ;** se **is** x == **null** valuta **true** allora la valutazione non e' definita su x .

La forma standard **split**, ad un posto per oggetto, deve essere specificata nella costruzione di un tipo astratto non finito, ed ha il compito di risolvere l'aliasing in modo tale da "trasformare" il legame **let ... == ...** nell'assegnazione.
Per esemplificare i punti ora esposti, e per mostrare almeno un poco l' "aroma" del linguaggio **Peano**, riportiamo ora come esempio di tipi astratti non finiti un possibile modo di costruire liste, sperando che l'ampia cultura generale, la fervida fantasia ed il robusto buon senso del lettore siano sufficienti a sopperire a cio' che non si e' detto.

```
Type: intlist is abstract.

---------- implementation part --------------------

Constant: nknots  =1000 ;
          maxcount=nknots .

Types :
  refcnt is 0..maxcount inherits succ,pred ;

  knot  is record
             content is integer;
             next    is intlist;
             cn      is refcnt
           end;

  index    is 0..nknots-1 ;

  intlist  is realized as knot.

Variables :
 storage   is array[index] of knot;
 freeknots is intlist.

The form initstore is:
  i,old_i are index;
  first_time is boolean.
 Begin first_time:=true;
   for i in index
    loop
      storage[i].cn:=0;
      if first_time then first_time:=false
       else let @(storage[i].next) == storage[old_i]
      end
     old_i:=i
    end;
    let @freeknots == storage[i]
  end.
```

```
The form dispose(list),
                where list indicates intlist, is:
 begin
  if  not is @list == null  then
   @list.cn := pred @list.cn
   if @list.cn=0 then
     dispose(@list.next);
     let @(@list.next) == @freeknots
     let @freeknots == @list
   end
  end
 end.

The function newknot() -> intlist is:
 begin
  let @newknot == @freeknots
  let @freeknots == @(@freeknots.next);
  @newknot.cn := 1
 end.

The form split(list),
          where list indicates intlist, is:
 begin
  if  not is @list == null then @list.cn := succ @list.cn
  end
 end.

The form &:=(t,q) ,
 where t indicates intlist
          and q is intlist, is:
 begin
  dispose(t);
  let @t == @q
  split(t)
 end.

The function equal(t,q) -> boolean,
               where t and q indicate intlist, is:
 begin
  equal :=
     if is @t == @q then true
     else if is @t == null then false
     else if @t.content <> @q.content then false
      else equal(@t.next, @q.next)
 end.

The function &=(t,q) -> boolean,
               where t and q are intlist, is:
 begin
  &= := equal(t,q)
 end.
```



```
The function putf(n,list) -> intlist,
     where n is integer and list is intlist, is :
 begin
  bind putf to newknot()
  @putf.content := n
  @putf.next    := list
 end.

The function first(list) -> integer,
               where list is intlist, is:
 begin
  first := @list.content
 end.

The function rest(list) -> intlist,
               where list is intlist, is:
 begin
  rest := @list.next
 end.

The function emptylist -> intlist is:
 begin
  let @emptylist == null
 end.

The function isempty(list) -> boolean,
                    where list is intlist, is:
  begin
   isempty :=   is @list == null
  end.

--------  end of implementation part  --------
```