Giornate di studio su:

"Programmazione Strutturata: Esperienze e Orientamenti"

L 76-3

Milano
23-24-25 giugno 1976

Strutture di dati: livelli di astrazione e metodi di definizione come strumento di progetto dei programmi

Pierluigi Della Vigna, Carlo Ghezzi
Istituto di Elettrotecnica ed Elettronica - Politecnico di Milano e Centro del CNR per l'Ingegneria dei Sistemi per l'Elaborazione delle Informazioni

Alberto Martelli, Franco Sirovich
Istituto di Elaborazione dell'Informazione del CNR - Pisa

Honeywell
Honeywell Information Systems Italia

STRUTTURE DI DATI : LIVELLI DI ASTRAZIONE E METODI DI DEFINIZIONE COME STRUMENTO DI PROGETTO DEI PROGRAMMI

Pierluigi Della Vigna (°)
Carlo Ghezzi (°)
Alberto Martelli (°°)
Franco Sirovich (°°)

(°) Istituto di Elettrotecnica ed Elettronica - Politecnico di Milano e Centro del CNR per l'Ingegneria dei Sistemi per l'Elaborazione delle Informazioni

(°°) Istituto di Elaborazione dell'Informazione del CNR - Pisa

Introduzione

La definizione da parte del progettista e del programmatore di tipi e strutture di dati gioca un ruolo essenziale nel processo di produzione di software. E' infatti molto stretto il legame intercorrente fra sviluppo di programmi per livelli di astrazione e sviluppo di tipi di dato per livelli di astrazione. Benchè tale legame risultasse di primaria importanza anche nei primi articoli che proponevano la programmazione strutturata come metodologia per lo sviluppo di software (vedi ad esempio [Dijkstra 1972]) solo recentemente le ricerche sul problema dello sviluppo dei tipi di dato per livelli di astrazione hanno ricevuto la necessaria attenzione.

In queste note discuteremo in dettaglio il legame esistente fra programmazione strutturata e sviluppo di tipi di dato astratti e quindi tenteremo una valutazione di alcuni linguaggi di programmazione dal punto di vista del supporto da essi offerto per lo sviluppo di tipi di dato astratti.

Concluderemo con un breve esame dei metodi di specifica (più o meno formale) dei tipi di dato astratti.

1. Programmazione strutturata e definizione di tipi di dato astratti.

La programmazione strutturata è una metodologia per il progetto di programmi ([Djikstra 1972] , [Wirth 1974] , [Knuth 1974] e per un'ampia rassegna [Maiocchi 1973]) che impone una disciplina nella programmazione allo scopo di ottenere programmi più affidabili. Secondo questa metodologia, un problema deve essere risolto attraverso un procedimento di decomposizioni successive. Il primo passo è dunque quello di scrivere un programma che risolve il problema dato e che fa uso non solo dei costrutti del linguaggio di implementazione prescelto ma anche di tipi di dato ed operazioni che, benchè non fornite dal linguaggio di implementazione, siano quelle più naturali per formulare la soluzione al problema. Tale programma è dunque scritto per una macchina astratta che costituisce il primo livello di astrazione nello sviluppo del programma. Ovviamente, i tipi di dato e le operazioni astratte (astrazioni) andranno programmate in termini di un ulteriore livello di astrazione fino a che risulterà facile e naturale realizzare le astrazioni date direttamente in termini del linguaggio di implementazione.

Seguendo questa metodologia, il progettista o il programmatore deve risolvere il problema in termini di una macchina astratta, senza preoccuparsi del modo in cui le astrazioni che usa saranno poi realizzate. Una volta assicuratosi che il programma scritto è una corretta soluzione al problema dato, può passare ad affrontare la realizzazione delle

astrazioni utilizzate precedentemente, senza la preoccupazione che la soluzione che sceglierà possa invalidare la correttezza dei programmi che usano tali astrazioni.

Questo modo di considerare la programmazione strutturata ruota attorno alla nozione di astrazione come meccanismo per esprimere i dettagli che ad un certo livello sono rilevanti e nascondere quelli che sono irrilevanti. I linguaggi di programmazione convenzionali forniscono uno strumento adeguato, la procedura, per un particolare tipo di astrazione. Oltre a costituire uno strumento per decomporre il compito affidato ad un programma, la procedura permette di separare chiaramente la funzione (di un componente di un programma) dal modo in cui viene realizzata. L'uso delle procedure e una disciplina nell'uso delle strutture di controllo constituiscono uno strumento largamente soddisfacente per realizzare astrazioni funzionali. Tuttavia le astrazioni riguardanti tipi di dato non sono realizzabili in modo soddisfacente mediante l'uso di procedure. Per giustificare tale affermazione è necessario precisare meglio il concetto di tipo di dato astratto.

Quando un programmatore fa uso di un oggetto astratto deve potersi limitare a tenere in considerazione soltanto il comportamento caratteristico di tale oggetto e non deve avere bisogno di conoscere i meccanismi utilizzati per realizzare tale comportamento. Il comportamento di un oggetto, per quanto riguarda il suo uso, è descritto dall'insieme di

operazioni che su di esso possono essere compiute e che quindi lo caratterizzano. Un <u>tipo di dato astratto</u> può dunque essere definito come una classe di oggetti astratti che è completamente caratterizzata dalle operazioni disponibili su tali oggetti. Quindi introdurre un tipo di dato astratto vuol dire definire l'insieme delle operazioni che lo caratterizzano. Qualunque informazione riguardante l'implementazione, per esempio il modo di rappresentare l'oggetto in memoria, non deve essere nè fornita nè richiesta all'utilizzatore dell'oggetto, ed è necessaria solo per implementare le operazioni caratterizzanti il tipo di dato astratto.

In sostanza, un tipo di dato astratto deve essere inteso come un tipo di dato primitivo di un linguaggio di programmazione. Un programmatore può dichiarare oggetti di tipo primitivo (peresempio, <u>integer</u>), eseguire su di essi le usuali operazioni aritmetiche ma non è in generale interessato alla rappresentazione degli interi nella macchina. E' invece opportuno che esista una protezione contro un uso errato o non significativo degli oggetti (ad esempio, moltiplicare un intero per un carattere) attraverso una conversione di tipo (se significativo) o una segnalazione di errore. Nel caso di tipi di dato primitivi il programmatore fa uso di una astrazione che è già implementata (ad un livello di dettaglio inferiore) dal linguaggio di programmazione e dal suo compilatore.

In modo del tutto analogo, un tipo di dato astratto verrà usato ad un certo livello di astrazione ed implementato ad un

livello più basso. Una adeguata implementazione di un tipo di dato astratto deve soddisfare i seguenti requisiti:

1) Deve garantire che l'utilizzatore di un tipo di dato astratto possa manipolare gli oggetti di quel tipo solo attraverso le operazioni del tipo e non attraverso la manipolazione diretta della rappresentazione in memoria.

2) Così come le procedure costituiscono uno strumento per "estendere" il linguaggio di implementazione per quanto riguarda le operazioni disponibili, i tipi di dato astratti devono costituire un'estensione del linguaggio mediante appunto nuovi tipi di dato. Non ci devono essere differenze fra i tipi di dato primitivi e quelli definiti dal programmatore, in particolare per quanto riguarda i meccanismi di controllo del tipo. Si noti come un severo controllo di tipo possa contribuire a raggiungere l'obiettivo di impedire le manipolazioni dirette della rappresentazione di un oggetto astratto.

3) L'implementazione di un tipo di dato astratto dovrebbe costituire un componente chiaramente separato dal resto del programma, compilabile separatamente e con interfacce ben definite e limitate alle operazioni caratterizzanti il tipo di dato astratto (quindi in particolare non deve avere dati in comune con altri componenti).

Il soddisfacimento di questi requisiti è necessario per far sì che la chiara distinzione concettuale utilizzata nella programmazione per livelli di astrazione, fra utilizzo di

una astrazione e definizione di tale astrazione sia mantenuta dalla implementazione ([Flon 1975]). In questo modo si può modificare la implementazione di unaastrazione senza timore di influenzare in qualche modo la correttezza dei componenti che utilizzano tale astrazione. Il collegamento fra il concetto di astrazione (tipo di dato e operazione) e quelli di "procedural eucapsulation" ([Zilles 1973]), "information hiding" ([Parnas 1971]), "programmazione modulare" ([Parnas 1972a] e 1972b]) è dunque molto stretto ed è stata infatti proposta una metodologia di programmazione che combina la programmazione strutturata con quella modulare ([Liskov 1972 , 1974]) e che appunto si fonda sul concetto di astrazione.

Se si considerano i requisiti per l'implementazione dei tipi di dato astratti, si può notare che si traducono immediatamente in requisiti del linguaggio di implementazione. Esamineremo ora da tale punto di vista alcuni linguaggi di programmazione, tenendo altresì presente che, se un linguaggio è adeguato per implementare astrazioni, una semplice definizione "concettuale" di una astrazione deve essere rispecchiata da una implementazione altrettanto semplice.

2. Utilizzo dei tipi di dati in alcuni linguaggi di programmazione.

In questo paragrafo verranno studiate le diverse possibilità offerte da alcuni linguaggi di programmazione per definire e utilizzare astrazioni sui dati.

I linguaggi che verranno considerati sono il PL-1, il PASCAL e il SIMULA 67. Altri linguaggi, anche più diffusi, quali COBOL, FORTRAN e ALGOL non verranno presi in considerazione perchè ritenuti a prima vista incapaci di esprimere alcun tipo di astrazione sui dati. Anzichè elencare le diverse strutture di dati offerte dai linguaggi, verrà sviluppato un esempio tendente a illustrare le possibilità di utilizzare le strutture di dati già presenti e di introdurre nuovi tipi di dati. Lo esempio riguarderà la definizione del tipo stack, con le operazioni associate di creazione di un oggetto del tipo stack, inserimento di un elemento in cima ad uno stack (operazione push), espulsione di un elemento dalla cima di uno stack (operazione pop), lettura dell'elemento di cima e interrogazione (empty) per controllare se lo stack è vuoto (*). Per tale esempio ci si è basati su [Aiello 1974].

(*) In tutti gli esempi che verranno illustrati si è cercato di definire il tipo stack nel modo più semplice e compatto. In particolare, si è per brevità tralasciato di definire le condizioni di errore nell'uso del tipo anche se si ritiene che la definizione di una diagnostica connessa con l'uso di tipi astratti sia un aspetto di estrema importanza nello sviluppo di grossi programmi.

A) PL-1

Il metodo più compatto per definire un tipo e associare al tipo le operazioni ammesse è quello che utilizza procedure con più punti di ingresso, come illustrato in fig. 2.1

```
create : PROCEDURE;
         DECLARE stack (100) CHAR (1) CONTROLLED;
         DECLARE top FIXED BIN (15) STATIC;
         /* lo stack contiene al massimo 100 elementi e
         top è l'elemento di cima. Stack è dichiarato
         controlled, lasciando così al programmatore la
         possibilità di allocarlo e disallocarlo mediante
         l'uso delle istruzioni ALLOCATE e FREE. Top è
         dichiarato STATIC e pertanto viene allocato in
         compilazione*/
         ALLOCATE stack;
         stack (1) = '!'; top = 1;
         /* il carattere! denota il fondo dello stack*/


      push : ENTRY (elem);/*inserisce elem in cima allo stack*/
             DECLARE elem CHAR (1);
             top = top+1
             stack (top)= elem, /* il controllo sul superamento
             delle dimensioni di stack è lasciato alla procedu-
             ra chiamante */
             RETURN;
      pop :  ENTRY;/*elimina l'elemento in cima allo stack*/
             IF top = 1 THEN
             top = top -1; /* ciò garantisce di rimanere al-
             l'interno dello stack*/
             RETURN;
top-element: ENTRY RETURNS (CHAR(1))/* fornisce l'elemento
             in cima allo stack */
             RETURN (stack(top)
     empty:  ENTRY RETURNS CHAR(1);
             DECLARE void CHAR (1);
             IF top=1 THEN  void='0'B ELSE void = '1'B;
             /* void indica se lo stack è vuoto oppure no
             a seconda che valga '0'B oppure '1'B  */
             RETURN (void);
             END create;
```

In questo modo è possibile manipolare un dato del tipo stack soltanto utilizzando le operazioni create , push, pop, top-element che sono ENTRY Point della procedura; pertanto non è direttamente manipolabile la struttura dei dati che rappresenta in memoria lo stack. D'altra parte questa soluzione non permette di creare più dati del tipo stack contemporaneamente, col che è azzardato affermare che l'utente è in grado di definire un proprio tipo stack.

Un altro inconveniente della soluzione adottata deriva dalla scelta di rappresentare lo stack come array controllato, poichè ciò implica che per esso venga allocata memoria per la massima dimensione prevista per lo stack. Più favorevole è la soluzione seguente :

```
DECLARE  1  stack BASED (p),
            2 next PTR ,
            2 value CHAR (1)
```

che realizza lo stack mediante una lista di strutture.

Un'altra soluzione è quella che utilizza l'attributo LIKE, la cui funzione è quella di fare una copia della dichiarazione di una struttura già dichiarata. "DECLARE a LIKE b" richiede che b venga dichiarato allo stesso o più alto livello di a.

```
DECLARE x FIXED BIN (15);
DECLARE 1 stack BASED (p),
          2 m FIXED BIN (15),
          2 body CHAR (x REFER (m))
          2 top FIXED BIN (15);
```

```
/* stack è una struttura a cui ci si riferisce tramite il
puntatore p,composta da un vettore body, la cui dimensione
m viene fissata tramite il valore di una variabileesterna
x, e da un elemento top, che dà la posizione in body della
cima dello stack*/
create:PROCEDURE (stack-ref);
      DECLARE 1 name BASED (stack-ref) LIKE stack;
      /*dichiara una struttura di nome "name", accessibile
      mediante il puntatore "stack-ref", copia di stack */
      name. body (1) = '!'; /*! è il fondo dello stack crea
      to e name. body(1) rappresenta il primo elemento del
      sottocampo body della struttura name*/
      name. top=1;
      RETURN;
push: ENTRY(stack-ref,elem); /* stack-ref è il puntatore
      allo stack nel quale va inserito elem */
      DECLARE stack-ref PTR, elem CHAR(1);
      DECLARE 1 name BASED (stack-ref) LIKE stack;
      name.top=name.top+1;
      name.body (name.top) = elem;
      RETURN;
pop:  ENTRY (stack-ref);
      DECLARE stack-ref PTR;
      DECLARE 1 name BASED (stack-ref) LIKE stack;
      name.top = name.top-1;
      RETURN;
  op-element :
      ENTRY (stack-ref) RETURNS (CHAR(1));
      DECLARE stack-ref PTR;
      DECLARE 1name BASED (stack-ref) LIKE stack;
```

```
        RETURN (name.body (name.top));
empty:ENTRY (stack-ref) RETURNS (CHAR(1));
        DECLARE stack-ref PTR;
        DECLARE 1 name BASED (stack-ref) LIKE stack;
        DECLARE void CHAR (1);
        IF name.top=1 THEN void='0'B ELSE
        void = '1'B;
        RETURN (void);
        END create;
        .
        .
        .
        .
        DECLARE 1 stack-1 BASED (p1)LIKE stack;
        DECLARE 1 stack-2 BASED (p2)LIKE stack;
            .
            .
            .
            .
        DECLARE 1 stack-10 BASED (10) LIKE stack;
        /*vengono create 10 istanze del tipo stack */
        x = 100;
        ALLOCATE stack-1;
            .
            .
            .
        x = 10;
        ALLOCATE stack-10;
        CALL create       (p1);
            .
            .
        CALL create       (p10);
            .
            .
            .
```

```
push (p3, 'b')
    .
    .
    .
```

fig. 2.2

Con la soluzione illustrata, pur potendo creare più variabili del tipo stack, il programmatore non risulterebbe vincolato ad accedere ad una variabile del tipo stack esclusivamente colle operazioni create push, pop, top-element e empty, ma è possibile manipolare direttamente la struttura dei dat che realizza in memoria il tipo.

Una soluzione che non presenta questo inconveniente consiste nell'associare un nome ad ogni nuovo stack che viene creato. In questo caso, all'interno della procedura che implementa il tipo stack, vengono collegati in una lista i nomi delle stack create con i relativi puntatori ai valori dello stack stesso (fig. 2.3). Pertanto il semplice nome di uno stack non consente di manipolarne il contenuto se non chiamando le procedure del tipo stack le quali hanno accesso alla lista delle stack create.

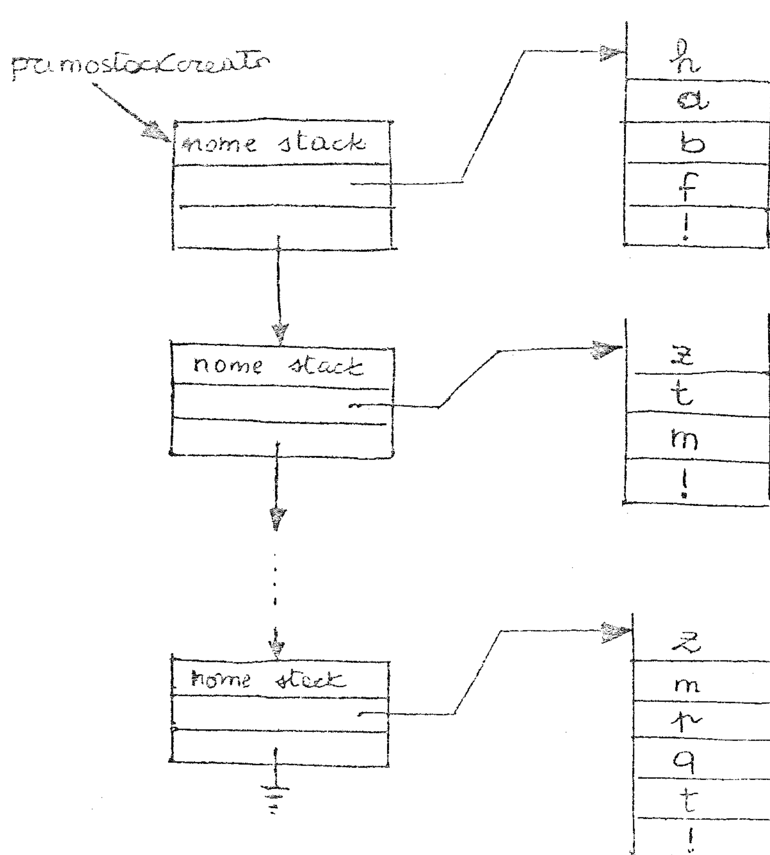

fig. 23.

Questa soluzione, peraltro, è estremamente onerosa specialmente se confrontata con la relativa semplicità del problema.

B) Pascal

In Pascal, oltre ai tipi semplici, strutturati e al tipo puntatore, è possibile definire nuovi tipi mediante il costruttore <u>type</u>, in funzione dei tipi già esistenti.

Nel caso dello stack, si veda il programma di fig. 24.

```
type stack = record body: array 1..100 of char;
                         top : integer ;
{i sottocampi body e top del record indicano rispettivamente
```

il vettore di caratteri in cui è memorizzato lo stack e l'indice della cima dello stack}

```
procedure create (var x : stack);
begin x . body [1]: = "!";
      x . top : = 1
end;
procedure push (elem: char; var x : stack); {elem viene
trasmessa per valore, stack per reference} :
begin x . top : = x . top  + 1;
      x . body [x . top]: = elem
end;
procedure pop (var x : stack);
begin x . top : = x . top - 1
end;
function top-element (var x : stack) : char;
begin top-element : = x . body [x . top]
end;
function empty (var x : stack) : boolean;
begin
     if x . top = 1 then empty : = true else empty : = false
end;
```

fig. 2.4

E' bene notare che, al di là della estrema semplicità e chiarezza con le quali è stato possibile definire il tipo stack, non esistendo nessuna possibilità di collegare rigidamente un tipo di dato definito dal programmatore alle procedure che realizzano i relativi operatori, risulta possibile manipolare un dato di un certo tipo direttamente nella sua rappresentazione in memoria e non attraverso le operazioni descritte.

Si noti infine che il Pascal non ammette arrays dinamici e pertanto, in questo caso, esiste soltanto il tipo stack di lunghezza prefissata a 100, mentre potrebbe essere utile disporre di diverse istanze di stack di dimensioni massime diverse.

C) SIMULA - 67

Il SIMULA-67 consente di realizzare nuovi tipi di dati mediante le classi. Nel caso del tipo stack la definizione di classe è riportata in fig. 2.5

```
class stack (n); integer n;
comment n definisce la dimensione dello stack e viene
specificato ogni volta che si crea un nuovo stack;
begin char array body [1.. n]; integer top;
      comment viene specificata la rappresentazione in memo-
              ria dello stack mediante un array. Nel segui-
              to vengono precisate le operazioni lecite sul
              tipo;

      procedure create;
                begin;
                      top : = 1;
                      body [1] : = '!'
                      end create;
      Procedure  push (elem); char elem;
                 begin;
                       top : = top + 1;
                       body [top] : = elem;
                 end push;
```

```
        procedure pop;
                  begin;
                        top = top - 1;
                  end pop;
char procedure top - element,
                begin;
                       top-element : = body [top];
                end, top-element
boolean procedure empty;
                  begin
                       if top = 1 then empty : = true
                       else empty : = false
                  end empty;


end stack;
```

fig. 2.5

Il programmatore può generare più oggetti di una classe mediante la primitiva new, tenendo presente che l'accesso ad un oggetto avviene soltanto tramite puntatori, nel seguente mddo :

```
ref (stack) stack 1, stack 2;
COMMENT stack 1 e stack 2 sono puntatori a 2 oggetti della
        classe stack;
stack 1 : - new stack (100);
stack 2 : - new stack (200);
```

La manipolazione di oggettidella classe stack può essere fatto nel modo seguente :

```
stack 1 . create    ; stack 2. create
COMMENT esegue la procedura create   sull'istanza stack 1
        e nell'istanza stack 2 della classe stack;
stack 1 .push ('a');
stack 2 .push ('b');
COMMENT esegue la procedura push rispettivamente con para-
        metro 'a' e 'b' sulle istanze stack 1 e stack della classe
        stack;
x:=stack 1. top-element;
COMMENT x riceve il valore 'a';
   .
   .
   .
   .
```

Risulta evidente che il SIMULA-67 permette di costruire un tipo raggruppando sia la sua rappresentazione in termini di altri tipi che le operazioni colle quali può essere manipolato.

E' bene notare però che il linguaggio non è in grado di vincolare il programmatore ad usare un oggetto di una classe soltanto attraverso le procedure della classe, ma può essere direttamente isolata la implementazione in memoria del tipo.
Si può infatti scrivere

stack 1. body [3] : = 'x',

accedendo dunque direttamente all'array body che contiene i dati dell'oggetto stack 1.

Un altro inconveniente sorge quando una classe x contiene una sottoclasse y. In questo caso la dichiarazione <u>ref</u> (x) alfa permette ad alfa di puntare non solo a oggetti del tipo x

ma anche ad oggetti del tipo y, saltando quindi un numero illimitato di livelli di astrazione.

D) Il linguaggio CLU

Il linguaggio CLU [Liskov 1976, Schaffert 1975] è stato definito come supporto per il progetto dei programmi e delle associate strutture di dati per livelli di astrazione. Il linguaggio permette di definire tipi astratti di dati, colle seguenti caratteristiche :

1. la definizione di un tipo astratto deve contenere la definizione di tutte le operazioni ad esso applicabili,
2. l'uso di un oggetto di tipo astratto non richiede la conoscenza della implementazione nel linguaggio del tipo astratto stesso;
3. un oggetto di tipo astratto può essere manipolato soltanto mediante le operazioni per esso definite e non mediante manipolazione diretta della sua rappresentazione nel linguaggio.

Alcune di queste caratteristiche sono presenti nei linguaggi analizzati nel precedente paragrafo, ma mai esse sono presenti contemporaneamente.

Queste tre caratteristiche consentono di estendere anche ai tipi definiti dal programmatore un importante strumento che facilita il progetto di programmi corretti, quale la verifica di compatibilità fra operazioni e operandi (type checking)
Viceversa, la mancanza di una rigida connessione fra operatori e

operandi implica, come avviene in massimo grado per l'ALGOL-68, una conversione automatica dei tipi degli operandi, non controllata esplicitamente dal programmatore. E' ben noto che questo dà luogo a programmi difficilmente comprensibili nel loro effettivo significato e quindi poco affidabili.

La possibilità di definire tipi astratti di dati, grazie soprattutto al punto 2, consente di modificare la rappresentazione di un tipo senza dover mutare il programma che utilizza oggetti di quel tipo.
Questo consente, ad esempio, di scegliere nella fase di iniziale messa a punto del programma una rappresentazione facile da implementare anche se poco efficiente, che permette di provare, anche sperimentalmente, la globale correttezza del programma. La scelta di una diversa e più efficiente realizzazione, e la sua prova, può essere dilazionata alla fase di effettivo utilizzo del programma.
Alla luce di queste considerazioni acquistano interesse anche ricerche volte alla scelta delle strutture dei dati e alla misura di efficienza della loro prestazione [Gotlieb 1974].

Nel linguaggio CLU svolgono un ruolo fondamentale due tipi di moduli : le procedure, per rappresentare astrazioni sulle operazioni e i clusters, per rappresentare tipi astratti di dati. In particolare, un cluster è costituito dalla descrizione del modo come il tipo è realizzato e da una collezione di procedure (operations) che operano su di esso. Nel caso del tipo stack, il cluster è riportato in fig. 2.6.

```
stack = cluster is create, push, pop, top-element, empty;
% sul tipo stack specificato dal cluster si può operare con
% le operazioni
% sopra specificate. Segue la specificazione della rappresenta-
% zione interna del tipo

rep = array [ char] ;
create = oper (       ) returns (cvt);
        % l'operazione non ha parametri di ingresso e restitui-
        % sce un singolo valore.

        % un argomento del tipo cvt ha tipo astratto
        % all'esterno dell'operazione e tipo rep, cioè array [ char]
        % all'interno.
        r : rep : = rep $ create (1);
        % con rep $ create (1) si crea un oggetto del tipo rep,
        % cioè array [ char] , di lunghezza nulla e estremo
        % inferiore 1. Il linguaggio ammette infatti arrays di
        % dimensione non fissata a priori e operazioni di crea-
        % zione e manipolazione.
        % L'oggetto creato è denotato da r.
        store (r, 1, '!');
        % memorizza il carattere! nella posizione di indice 1 di
        % r.
        return (r);
        % Le 3 operazioni precedenti potevano essere così compatta
        % te:
        % return (rep $ store (rep $ create (1), 1, '!');
        end create:
```

```
push = oper (pila: cvt, elem: char);
       % push riceve parametri pila ed elem.
       array [char] $ extendh (pila, elem),
       % extendh (pila, elem) è una operazione pirimitiva
       % del linguaggio che estende di un elemento la dimen-
       % sione superiore dell'array pila
       % e vi inserisce l'elemento elem
       return;
       end push;
pop =  oper (pila : cvt);
       rep $ retracth (pila);
       % retracth (pila) diminuisce di uno il valore dell'e-
       % stremo superiore, e quindi della dimensione della
       % pila
       return;
       end pop;
top-element= oper (pila: cvt) returns (char);
       % restituisce il carattere in cima alla pila
       return (rep$ fetch (pila, rep $ high (pila)));
       % rep $ high (pila) fornisce il valore dell'estremo
       % superiore di pila
       % rep $ fetch (pila, x ) fornisce l'elemento
       % di indice x di pila
       end top-element;
end stack
```

fig. 2.6

Infine, si osservi che l'esempio descritto definisce il tipo stack nel caso particolare in cui i dati elementari siano del tipo carattere. Spesso, peraltro, ha senso caratterizzare in modo sostanzialmente identico un tipo astratto in modo parametrico rispetto ai tipi dei dati elementari che compongono oggetti del tipo astratto. Nel caso in esame, infatti, il comportamento di uno stack, rispetto alle operazioni che si sono definite, è lo stesso indipendentemente dall'essere caratteri o di altro tipo gli elementi costitutivi.

Il linguaggio CLU consente di realizzare queste importanti astrazioni definendo i generatori di tipo (type generators), che differiscono dai consueti tipi in quanto definiscono una classe di tipi piuttosto che un singolo tipo.
Il cluster che definisce il generatore di tipo stack sarebbe definito come in fig. 2.7.

```
stack = cluster [elementtype:type]  is create, push,
        pop, top-element, empty;
rep = array [elementtype];
•
•
•
•
•
•
•
end stack
```

fig. 2.7

L'utilizzo del linguaggio CLU richiede un sistema che, a grandi linee, può essere così tratteggiato. Quando viene definita una nuova astrazione, ad essa viene associato un descrittore che viene inserito nella base di dati del sistema.

Il descrittore è composto da :

1) l'interfaccia con altri moduli, e cioè la specifica dei tipi di tutti i parametri di ingresso e uscita per quanto riguarda le astrazioni funzionali e dei tipi di tutti i parametri di ingresso e uscita di tutte le operazioni per quanto riguarda i tipi di dato astratti.

2) un insieme di diverse implementazioni per l'astrazione. Tale insieme può essere vuoto, quando ci si accontenta di verificare il corretto uso dell'astrazione, deve contenere almeno una implementazione in fase di esecuzione e in generale può contenere diverse realizzazioni, più o meno efficienti, da scegliersi a seconda delle particolari esigenze.

Da quanto detto risulta chiaro che una caratteristica del CLU è quella di essere un linguaggio fortemente tipizzato, estendendo la verifica di compatibilità fra i tipi anche alle interfaccie fra i moduli, verifica effettuata al momento della compilazione.

3. Metodi di specifica di tipi di dato astratti

La specifica del comportamento di una astrazione costituisce in realtà un problema centrale nella programmazione strutturati. Infatti, è chiaro che, in mancanza di una specifica precisa e completa delle astrazioni, il programmatore o il progettista che la usa sarà portato, sulla base della propria intuizione, a fare delle ipotesi sul modo in cui l'astrazione verrà implementata, ipotesi che non necessariamente saranno soddisfatte poi dall'implementazione. D'altra parte, le specifiche delle astrazioni costituiscono le definizioni del problema di programmazione ad un successivo livello di astrazione ed è quindi indispensabile che esse siano il più possibile precise e complete. La specifica delle astrazioni introdotte costituisce la parte più importante della documentazione che accompagna sia le varie fasi di sviluppo di un progetto che il programma finale.

Molto spesso le specifiche di una astrazione vengono espresse in linguaggio naturale o addirittura affidate semplicemente al nome che si sceglie per l'astrazione. E' evidente l'inadeguatezza intrinseca in tale metodo, in primo luogo per l'ambiguità propria del linguaggio naturale. Se si tiene presente che lo scopo principale delle specifiche è quello di costituire uno strumento essenziale per la verifica, informale o formale, della correttezza di programmi (dei programmi che usano le astrazioni come di quelli che le implementano), risulta evidente la necessità di individuare un linguaggio formale in cui esprimere tali specifiche.

Si possono fare alcune osservazioni sui criteri da seguire nella scelta del linguaggio formale di specifica, tenendo presente in primo luogo considerazioni di ordine pratico. Infatti deve essere possibile, senza eccessive difficoltà, costruire le specifiche di una astrazione. D'altra parte, una persona esperta della notazione impiegata deve essere in grado di ricostruire, a partire dalle specifiche il concetto (astrazione) che esse intendono descrivere. E' chiaro che è qui presente un elemento di giudizio soggettivo ma a nostro avviso di notevole rilevanza per le applicazioni in ambienti di produzione industriale di software. In conclusione, la "fatica" necessaria per stendere o per comprendere le specifiche deve essere di gran lunga inferiore a quella necessaria per stendere o comprendere il programma che le realizza.

Infine, vorremmo sottolineare l'esigenza di una neutralità del linguaggio di specifica rispetto alle sue applicazioni. Deve essere possibile specificare precisamente quelle proprietà dell'astrazione che sono rilevanti e niente di più. In sostanza, una specifica deve dire quale è il comportamento di una astrazione, ma nulla deve essere anche specificato riguardo al modo in cui tale comportamento è ottenuto. Infine, risulta ovvio che è desiderabile un ampio campo di applicabilità del linguaggio di specifica in modo che non sia necessario modificarlo per diverse applicazioni.

Alcuni problemi interessanti sorgono quando è necessario specificare un tipo di dato astratto (si veda [Liskov 1975] per una discussione sul tema). In tale caso, è necessario speci-

ficare sia il dominio, cioè l'insieme di oggetti che appartengono al tipo di dato, sia le operazioni su tale dominio.
Un primo aspetto importante della specifica delle operazioni è la loro funzionalità, cioè i dominii dei loro argomenti e dei valori restituiti.

Nel caso del tipo di dato stack di caratteri, la funzionalità degli operatori può essere specificata come in fig. 3.1

CREATE	:	⟶	STACK
PUSH	: STACK xCHAR	⟶	STACK
POP	: STACK	⟶	STACK U STACKERROR
TOP-ELEMENT:	STACK	⟶	CHAR U CHAERROR
EMPTY?	: STACK	⟶	BOOLEAN

Fig. 3.1 - Funzionalità di operazioni associate al tipo stack di caratteri.

Una prima osservazione è che nella specifica delle funzionalità è presente il dominio che si sta specificando (STACK) insieme ad altri dominii che si suppongono già specificati (CHAR e BOOLEAN). Conseguentemente, si possono distinguere tre generi di operazione. I costruttori primitivi sono quelle operazioni che non hanno operandi appartenenti al tipo di dato che si sta specificando e invece producono risultati appartenenti a tale tipo (in questo caso, CREATE).

I costruttori di combinazione sono invece operazioni che producono risultati appartenenti al tipo di dato, operando però anche su dati appartenenti al tipo (ad esempio, PUSH e POP), Infine, vi sono in generale operazioni che potremmo chiamare proiettori le quali producono risultati non appartenenti al tipo di dato in esame (TOP-ELEMENT e EMPTY?). Mentre la specifica delle funzionalità pone in evidenza il genere delle singole operazioni, nulla ancora viene detto nè riguardo al modo in cui tali operazioni saranno programmate (il che è un aspetto positivo), nè riguardo alle proprietà di cui tali operazioni godranno (e che devono costituire per l'utilizzatore una descrizione completa del tipo di dato astratto).

Per specificare le proprietà delle singole operazioni è necessario un formalismo per descrivere gli oggetti appartenenti al tipo di dato in esame. Basta infatti osservare che le proprietà di tali operazioni sono appunto proprietà di applicazioni delle operazioni ad oggetti del tipo.

Questo problema viene risolto in due modi: un primo modo consiste nel fornire un modello astratto degli oggetti del tipo e nel definire le operazioni in termini del modello. Il modello deve ovviamente godere di proprietà ben note e disporre di operatori propri che vengono utilizzati per la specifica delle operazioni del tipo di dato. Ad esempio, come modelli astratti sono stati usati grafi [Earley 1971, Della Vigna-Ghezzi 1976] o insiemi [Earley 1973]. Questo modo di definire il dominio presenta alcuni svantaggi. In primo luogo, può accadere che la specifica del tipo sia sovrabbondante, contenendo cioè dettagli che sono inerenti all'uso del particolare modello

astratto impiegato e che non necessariamente dovrebbero essere preservati da un'implementazione corretta dell'astrazione.
Per la stessa ragione l'uso di un certo modello astratto può condurre a specifiche farraginose per certi tipi di dato astratto per i quali quindi il modello non è adatto. Infine, è difficile che il modello stesso non suggerisca anche un particolare tipo di implementazione, "nascondendo" all'implementatore una implementazione differente.

Il secondo modo di descrivere gli oggetti appartenenti al tipo di dato astratto consiste nel definirli induttivamente mediante le operazioni disponibili sugli oggetti stessi. L'insieme degli oggetti è cioè definito come la chiusura rispetto ai costruttori primitivi e di combinazione. Per esempio, l'insieme degli stack è ottenibile mediante l'unico costruttore primitivo CREATE, che produce lo stack vuoto, e da tutti gli stack che si ottengono applicandovi una qualunque sequenza di operazioni PUSH e POP. Gli oggetti appartenenti al tipo di dato possono essere visti come gli stati (non necessariamente finiti) di una macchina astratta [Parnas 1972a]. Vi saranno allora operazioni che modificano lo stato della macchina (costruttori), ed operazioni che permettono di osservare aspetti di uno stato (proiettori). Le specifiche delle operazioni saranno date indicando l'effetto delle operazioni del primo tipo sul risultato prodotto da quelle del secondo tipo.

Un'alternativa all'uso degli stati di una macchina per specifi-

care un tipo di dato astratto consiste neldare una lista di proprietà possedute dagli oggetti e dalle operazioni su di essi. Questo approccio può essere formalizzato esprimendo le proprietà come assiomi di una teoria logica del primo ordine con uguaglianza e induzione [Hoare 1973]. Da più parti sono stati avanzati dei dubbi sulla comprensibilità delle specifiche che in tal modo si ottengono e sulla difficoltà nello stendere tale specifiche.

E' stato recentemente proposto un approccio algebrico alla specifica dei tipi di dato astratti, che, benchè ancora carente, conduce a specifiche di più facile stesura e comprensibilità [Guttag 1975, Zilles 1976]. Seguendo tale approccio la specifica della funzionalità è integrata da assiomi detti <u>relazioni</u>. Gli oggetti appartenenti al tipo di dato (<u>parole</u> dell'algebra) sono descritti dalle sequenze di applicazioni di operazioni che li producono. Le relazioni specificano (assieme alle funzionalità) le proprietà delle operazioni del tipo di dato fornendo un modo per decidere se una data sequenza di operazioni è una sequenza legale. Gli assiomi che specificano lo stack di carattere sono mostrati in figura 3.2. Si può osservare che gli assiomi 3 e 4 esprimono

---

1. TOP-ELEMENT (PUSH(S,C))=C
2. TOP-ELEMENT(CREATE)=CHARERROR
3. POP(PUSH(S,C))=S
4. POP(CREATE)=STACKERROR
5. EMPTY?(PUSH(S,C))=FALSE
6. EMPTY?(CREATE)=TRUE

Fig. 3.2 - Relazioni di una specifica algebrica del tipo di dato stack di caratteri

proprietà dei vari costruttori ed introducono una relazione di equivalenza fra gli oggetti del tipo che fra l'altro rende agevole la definizione delle proprietà dei proiettori. La presenza di operazioni di proiezione rende necessario il ricorso ad algebre eterogenee per trattare diversi tipi di dato contemporaneamente.

Per concludere, vorremmo discutere almeno brevemente il problema degli effetti laterali. Nella maggior parte dei linguaggi di programmazione gli oggetti appartenenti a un tipo di dato possono essere modificati in vario modo.
Per le operazioni che provocano tali effetti laterali la specifica della funzionalità può dunque essere priva di senso o solamente parziale. Per esempio l'operazione PUSH potrebbe essere implementata (come negli esempi mostrati nel capitolo precedente) in modo di non restituire alcun valore.

Poichè d'altra parte il ricorso a specifiche matematiche, di tipo funzionale, è particolarmente comodo e naturale, un modo convenzionale di aggirare questa difficoltà consiste nel descrivere un oggetto modificabile mediante una coppia costituita dal suo nome di identità (unico per ogni oggetto distinto) e dal suo stato. Ogni operazione che ha un effetto laterale è dunque definita come una funzione che restituisce una nuova coppia rappresentante lo stesso oggetto ma in uno stato modificato.

BIBLIOGRAFIA

[Aiello 1974] J.M. Aiello, An investigation of current language support for the data requirements of structured programming. MIT-Project MAC Tech. Memorandum 51, CSG Memo 105, Settembre 1974.

[Della Vigna 1976] P. Della Vigna, C. Ghezzi Data structures and graph grammars, da presentare alla Conferenza ECI 76, Amsterdam, Agosto 1976.

[Dijkstra 1972] E. W. Dijkstra, Notes on structured programming, in O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming, Academic Press, pp. 1-81, 1972.

[Earley 1971] J. Earley, Toward an understanding of data struαtures, Communications of the ACM, Vol. 14, pp.617-627, Ottobre 1971.

[Earley 1973] J. Earley, Relational Level data structures for programming languages, Acta Informatica, Vol. 2, pp. 293-309, 1973.

[Flon 1975] L. Flon, Program design with abstract data types, Dept. of computer science, Carnegie-Mellon Univ., Giugno 1975.

[Gotlieb 1974] C.C. Gotlieb, F.W. Tompa, Choosing a storage schema, Acta Informatica vol. 3, pp.297-319, 1974.

[Guttag 1975] J.V. Guttag, The specification and application to programming of abstract data types, Technical Report CSRG-59, University of Toronto, Settembre 1975

[Hoare 1973] C.A.R. Hoare, N. Wirth, An axiomatic definition of the programming language PASCAL, Acta Informatica, Vol.2, pp. 335-355, 1973.

[Knuth 1974] D.E. Knuth, Structured programming with go to statements, ACM Computing Surveys, Vol. 6, N. 4, pp. 261-301, Dicembre 1974.

[Liskov 1972] B.H. Liskov, A. design methodology for reliable software systems, Proc. 1972 Fall Joint Comp. Conf., Vol. 41, AFIPS Press, pp. 191-159, 1972.

[Liskov 1974] B.H. Liskov, S. Zilles, Programming with abstract data types, Proc. ACM Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, pp. 50-59, Aprile 1974.

[Liskov 1975] B.H. Liskov, S.N. Zilles, Specification techniques for data abstractions, IEEE Trans. on Software Engineering, Vol. SE-1, N.1, pp.7-19, Marzo 1975

[Liskov 1976] B.H. Liskov, An Introduction to CLU, MIT- Laboratory for Computer Science, Computation Structures Group Memo 136, Febbraio 1976.

[Maiocchi 1973] M. Maiocchi, R. Polillo, Techniche di program mazione strutturata, Informatica, Vol. IV, N. 3/4, pp. 262-285, Settembre/dicembre 1973.

[Parnas 1971] D.L. Parnas, Information distribution aspects of design methodology, Proc. IFIP Congress 1971, TA-3 pp. 26-30.

[Parnas 1972 a] D.L. Parnas, A technique for software modules specification with examples, Communications of the ACM, Vol. 15, pp. 330-336, Maggio 1972.

[Parnas 1972 b] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Vol. 15, pp. 1053-1068, Dicembre 1972.

[Schaffert 1975] C. Shaffert, A. Snyder, R. Atkinson, The CLU reference Manual, Rev. 1, MIT-Project MAC, Settembre 1975.

[Wirth 1974] N. Wirth, On the composition of well-structured programs, ACM Computing Surveys, Vol. 6, N. 4, pp.247-259, Dicembre 1974.

[Zilles 1973] S.N. Zilles, Procedural encapsulation: a linguistic protection technique, SIGPLAN notices, 8, pp. 142-146, Settembre 1973.