ISSN 0381-9019

REPRESENTING DATA PATHS BY PROGRAM STRUCTURES

BY

TONY MIDDLETON

TR79-020, AUGUST 1979

REPRESENTING DATA PATHS
BY PROGRAM STRUCTURES

Tony Middleton

School of Computer Science
University of New Brunswick
P.O. Box 4400
Fredericton, N.B.
Canada

Summary

This paper discusses an approach to using program fragments to characterise data aggregates (or "data paths"). The main topic of the paper is the use of this representation to specify assignments between data paths. The approach allows certain assignments between paths to be represented in a conceptually simpler manner than would be the case for conventional techniques.

1. The Use of Program Fragments to Characterise Data Aggregates

In Algol68, a program fragment such as

```
begin read(x) ; x end;
```

can be used to denote a data value in certain contexts, as serial clauses are allowed to deliver values much like conventional expressions.

Why not extend the idea to data aggregates? I.e. why not allow a program fragment to represent an aggregate (or "stream", or "data path") of values?

Thus, some notation such as

```
read(x);
while x ≥ 0 do
      use x
      read(x)
endwhile;
```

might be used to represent the set of values which is obtained by using all those values in the input stream up until the occurence of a negative value.

Now other researchers (e.g. [4,8]) have indicated that the expressive power of programming languages can be greatly enhanced by the use of "aggregate opertions", i.e. operations which can be applied to whole collections of data items. This approach allows the programmer to avoid much of the tedious, error-prone, blow-by-blow description of processes of which Backus has complained [1].

One aggregate operation which must be present in any higher-level language (implicitly or explicitly) is assignment. The paper studies the problem of assignment between data paths when the data paths are represented by program fragments.

2. Language Notation Used

The language notation used for developing examples essentially corresponds to POP2 [3], except that

(i) The right-hand side and left-hand side of assignments are reversed to the more usual order.

(ii) Square brackets are used to denote array accessing.

(iii) More convenient control structures are assumed (it is not difficult to write POP2 macros to implement such control structures)

(iv) Minor liberties are taken with certain standard identifiers (e.g. HEAD and TAIL are used instead of HD and TL)

The language notation used was chosen partly for other work by the same author (e.g. [10,11]).

POP2 is a "dynamic-type" language - i.e. the type of a variable can vary at run-time (likewise SNOBOL, LISP and APL).

An alternative would have been to use a "fixed-type" language, in which the type of a variable is fixed at translation-time (e.g. FORTRAN, Algol68 (ignoring subtleties about the use of unions) or PASCAL (ignoring subtleties about record variants)).

The general reasons for basing the notation on a variable-type language were as follows

(i) There is no need to worry about the need for the translator to supply correct declarations for any system-generated variables

(ii) Information needed to control iterations can easily be acquired through the use of suitable run-time routines (e.g. to find out the number of elements in a vector). In a fixed-type language, declarations might have to be scanned to extract this information (though Algol68 provides some relief in this respect).

(iii) In a variable-type language, there is an isolation between the form of a data structure and its contents. It is convenient to isolate some functions (e.e. count the number of elements in a list) from the nature of the contents of the data structures manipulated.

The choice of language notation is more relevant to other work by the same author (e.g. [10,11]), than it is to the present paper, but it will be used here for consistency. The only deviation from previous notation will be that, for emphasis, "⇐ " will be used to denote assignment between aggregates.

3. Some Motivating Examples

Before proceeding to details, it is worth presenting some examples which indicate the motivation behind this work.

Each example involves an assignment of the form

$$P_B \Leftarrow P_A$$

where P_B is a left-hand side (LHS) data path and P_A is a right-hand side (RHS) data path, and both P_A and P_B are described by program fragments.

As an example, consider

```
for i to m                              (Example 1)
do for j to n
   do use x[i,j]
   od
od    ⇐     read(z);
            while z ≥ 0 do
                use z;
                read(z);
            endwhile;
```

This will assign values to an array x by rows. The values will be taken from the input stream up until the first occurence of a negative number. The next example involves moving information from one cyclic queue to another:

```
while $(n_1$ back $last_1) \neq first_1$                (Example 2)
do
    $last_1 \leftarrow n_1$ back $last_1$;
    use $queue_1$ $[last_1]$;
od
endwhile  ⟸    while $first_2 \neq last_2$
                do
                    use $queue_2[first_2]$;
                    $first_2 \leftarrow n_2$ back $first_2$;
                od
                endwhile;
```

The queues are stored "end-around" in vectors. "First" and "last" are indices to indicate the least recent and most recent entries in a queue respectively. The back operator moves one position backwards in the queue structure, and is defined so

```
n back i ≡ if i = 1 then n
           else i-1
           endif;
```

he queues are of lengths n_1 and n_2 and the condition "first=last" denotes the empty queue.

The LHS (left-hand side) data path stores items in $queue_1$, working backwards from just behind the last item. The RHS (right-hand side) data path extracts items from queue$_2$, working backwards from the first item.

At this point, it is worth stating an advantage which is being claimed for this approach. In the case of assignment, the idea is to have a "conceptual separation" between the LHS data path and the RHS data path. The two paths can be thought of a separate entities and disposed of separately in a "divide and conquer" fashion somewhat akin to that employed in other program design strategies, such as "stepwise refinement" [12].

For example, having defined a "conceptual unit" which will store items in queue$_1$ this unit can be used in a different assignment:

```
while ($n_1$ back $last_1$) ≠ first                 (Example 3)
do
     $last_1$ ← $n_1$ back $last_1$;
     use $queue_1$ [$last_1$];
od
endwhile       ⟸        read(x);
                         while x ≥ 0 do
                               use x;
                               read(x);
                         endwhile;
```

Here, the RHS data path consists of a set of data values on the input stream, terminated by the first occurence of a negative number.

The next example involves conversion from one representation of text to another. Let the two representations be as follows

(a) The words of text are stored, end-to-end in a vector VEC. The index of the first character of each word is stored in FIRST. The variable NWORDS indicates the number of words in the text. The vector SIZE contains the size of each word.

(b) The text is stored in a chained list of "blocks". Each block is a record structure containing the following components

CHARS - a vector for storing up to NBLOCK characters

COUNT - a count of the number of character slots actually used in a block

NEXT - a pointer to the next block

In this representation, a block is inserted at the end of each word.

If "old" is a string of form (a) which is to be converted to form (b) and stored in "new", then the conversion can be specified as follows:

```
new ← new_block;                              (Example 4)
last ← new;
repeat
     for i to nblock
     do
          count(new) ← i;
          use chars(new)[i];
     od
     next(last) ← new_block;
     last ← next(last);
```

```
        forever     <==
                                  for k to nwords
                                  do n ← size[k]
                                     f ← first[k]
                                     for j from f to f+n-1
                                     do use vec[j]
                                     od;
                                     use blank;
                                  od;
```

where

new_block   generates a new block

last   points to the most recent block generated

Now, an example which highlights a weakness of this approach:

```
        for i to n                                           (Example 5)
        do
           use p[i]
        od          <==            for j to n
                                   do
                                      use q[j]
                                   od;
```

This doesn't seem to offer any advantage over

```
                              for i to n
                              do p [i] ← q[i]
                              od;
```

In fact, one might well argue that the approach suggested is significantly worse than conventional techniques for such a straightforward example.

4. Mechanics

The assignment

$$P_B \Longleftarrow P_A$$

can be regarded as the "parallel" (or, more correctly, "pseudo-parallel") usage of the two data paths P_A and P_B.

It is well known ([2,5]) that a pseudo-parallel effect can be achieved by using a co-routine mechanism. Such a mechanism will now be outlined.

The approach presented here provides a co-routine-like mechanism by using source-level program transfomrations (rather like [6] and [9]). The mechanism is, in fact, very much like the use of "open-coded co-routines". Instead of manipulating program addresses, as in the case of true co-routines, "addresses" are conveyed by integer variables which are used to control suitable computed-goto statements.

It should be made clear right now that the mechanism can be *hideously inefficient* and is not being recommended as a practical implementation technique.

I am only trying to say "it can be done this way", not "it should be done this way". Later, it is hoped to progress towards a more practical implementation technique.

The following notation is used

i_A An integer variable used to keep track of progress thru P_A.

i_B Similar, for P_B.

ℓa_k A label, in a transformed version of P_A, which is associated with the k'th "control point" in P_A.

ℓb_k Similar, for P_B.

ℓa_0 A special control point associated with "entry" to the path P_A.

ℓb_0 Similar, for P_B.

begin A label associated with the start of processing P_A and P_B.

finish A label to which control passes after either P_A or P_B is exhausted.

use (A_k) The k'th occurrence of "*use*" in P_A. (The order of numbering of the occurrences of "*use*" in a path has no effect on the mechanisms given here.)

use (B_k) Similar, for P_B.

The techniques discussed involve source-to-source program transformations, rather like [6] and [9].

The general structure used to realize

$$P_B \Longleftarrow P_A$$

is:

```
begin:      $i_A \leftarrow 0$;  $i_B \leftarrow 0$;

choose$_A$:   goto ($\ell a_0$, $\ell a_1$, $\ell a_2$, ..., $\ell a_M$), $i_A + 1$;

$\ell a_0$:        +------------------------+
            | Access                 |
            |   Mechanism            |
            |     for $P_A$              |
            +------------------------+

            goto finish:

$\ell b_0$          +------------------------+
            | Access                 |
            |   Mechanism            |
            |     for $P_B$              |
            +------------------------+

finish:
```

The "access mechanism" for P_A is simply the code structure used to represent P_A with the following transformation

```
use ($A_k$) $\Longrightarrow$ $i_A \leftarrow k$;
              use ($A_k$)
              goto ($\ell b_0$, $\ell b_1$, ..., $\ell b_N$), $i_B + 1$;
          $\ell a_k$;
```

applied to each of the M "use-events" in P_A. The assignment "$i_A \leftarrow k$" effectively "remembers" that control must return to control point ℓa_k when an item is next needed from P_A. When control reaches the end of the access mechanism for P_A, the statement "goto finish" is executed. (Similarly for P_B. Hence exhaustion of either data path will cause processing to terminate.)

A similar transformation is used in P_B:

```
use $(B_k) \Longrightarrow$ $i_B \leftarrow k$;
               use $(B_k)$
               goto $\text{choose}_A$;
               $\ell b_k$:
```

After this, the following is done

(a) Let V_A be a variable which receives the current data value from P_A.

(b) use (A_k) will have the form

```
use $e_A$;
```

make the substitution

```
use $e_A$;  $\Longrightarrow$  $V_A \leftarrow e_A$;
```

for all values of k.

(c) use (B_k) will have the form

```
use $e_B$;
```

make the substitution

```
use $e_B$;  $\Longrightarrow$  $e_B \leftarrow V_A$;
```

for all values of k.

5. An Example

As an example, consider the assignment

```
for i to n                                        (Example 6)
do
    use x[i]
    use x[2*n-i+1];
```

```
od    ⟸
              read(y);
              while y ≥ 0   do
                   use y;
                   read(y);
              endwhile;
              read(z);
              while z ≥ 0   do
                   use z;
                   read(z);
              endwhile;
```

Let the occurrences of use in each path be numbered in the order in which they occur textually.

The complete code structure that would be produced by the proposed mechanisms is:

```
          begin:  $i_A \leftarrow 0$;  $i_B \leftarrow 0$;
          $\text{choose}_A$:   goto ($\ell a_0$, $\ell a_1$, $\ell a_2$), $i_A+1$;
          $\ell a_0$:          ⎧ read(y);
                               ⎪ while y ≥ 0   do
                               ⎪        $i_A \leftarrow 1$;
     Access mechanism          ⎨        $V_A \leftarrow y$;
     for $P_A$                 ⎪        goto ($\ell b_0$, $\ell b_1$, $\ell b_2$), $i_B+1$
     (RHS data path)           ⎪        $\ell a_1$:
                               ⎪        read(y);
                               ⎪ endwhile;
                               ⎩ read(z);
```

```
                 while z ≥ 0  do
                      $i_A$ ← 2;
                      $V_A$ ← z;
                      goto ($\ell b_0$,$\ell b_1$,$\ell b_2$); $i_B$+1;
                      $\ell a_2$;
                      read(z);
                 endwhile;
                 goto finish;

        $\ell b_0$:        for i to n
                       do
                            $i_B$ ← 1;
                            x[i] ← $V_A$;
Access mechanism            goto choose$_A$;
for $P_B$                     $\ell b_1$:
(LHS data path)             $i_B$ ← 2;
                            x[2*n-i+1] ← $V_A$;
                            goto choose$_Z$;
                            $\ell b_2$:
                       od;
        finish:
```

This is a very ugly piece of program, but

(i) The ugliness can be hidden from the user.

(ii) I'm only trying to show that assignments such as those in examples 1 to 4 can be translated mechanically. Later on it is hoped to progress towards more sophisticated translation techniques (at least for simple cases).

6. Relevance to the "Structured Programming" Debate

As we all know, there has been (justifiably) much debate in recent years about how to structure programs properly. Knuth [7] has related that, for certain program fragments, the "structured programming" methodology can sometimes seem like an ill-fitting straight jacket.

It is felt that the sort of assignments presented in Examples 1 to 4 produce similar problems.

As an example, let's look at some alternatives for Example 1, using conventional techniques.

One approach would be

```
read(x);                                    (Example 7(a))
for i to m while z ≥ 0
do
    for j to n while z ≥ 0
    do
        x[i,j] ← z;
        read(z);
    od
od;
```

There are two occurrences of the test "z ≥ 0". This is not very elegant. (Furthermore, the programmer might forget one of them!)

Another approach might be

```
read(z); i ← 1; j ← 1;                      (Example 7(b))
while (z ≥ 0) ∧ (i ≤ m)
do
    x[i,j] ← z;
    read(z);
    j ← j+1;
    if j > n then
        j ← 1; i ← i+1
    endif
od
endwhile
```

This only uses one occurrence of the test "$z \geq 0$", but the counted loop structure for accessing x is less obvious than its equivalent using nested for-loops.

We are getting into the sort of problems discussed by Knuth [ ]. In such a situation, particularly if we are concerned about efficiency, the programmer may be strongly tempted to resort to the use of the goto:

```
     read(z);
     for i to m
     do
          for j to n
          do
               if z ≥ 0 then
                  x[i,j] ← z;
               else
                  goto done;
               endif;
               read(z);
          od
     od;
done:
```

And so on.

What is the basic problem? Well, one can regard the assignment problem as one of "co-ordinating" two processes: one of which supplies items from P_A and the other of which stores the items in P_B. Problems arise when there is a "structural mismatch" between the two processes, ie. when the control structures for accessing P_A and for accessing P_B are quite different and cannot be easily "jammed" together (as they can in Example 5).

Thus, it is felt that the need to "co-ordinate" two pseudo-parallel processes, as in assignment between data aggregates, can give rise to substantial problems of program structuring.

It is being proposed here that such problems can be substantially alleviated by achieving the sort of "conceptual separation" between the supplier and consumer processes which is demonstrated in Examples 1 to 4.

7. Application of Other Aggregate Operators

In [10], it was shown how other aggregate operators can be applied to aggregates which are characterized by program fragments.

As an example, the expression

```
sumx   SUM(read(x);
               while x ≥ 0 do
                     use x;
                     read(x)
               endwhile);
```

can be translated using the techniques given in [10], by first rephrasing the fragment "read(x) ..... endwhile" as

```
<start:block>
read(x);
while x ≥ 0  do
      <item:x>
      read(x)
endwhile;
<end:block>
```

and then applying the substitution mechanisms given in [10].

Thus, the representation of data aggregates by program fragments allows for the implementation of other aggregate operators besides the assignment operator. (Hence the desire to keep the language notation used here reasonably consistent with that used in other work by the author.)

However, such other aggregate operators are not strictly relevant to the current paper, and no more will be said about them.

8. Efficiency

It is only the intention of this paper to show that certain assignment can be phrased as in Examples 1 to 4 and that these assignments can be translated by the co-routine-like methods given above.

The author has studied other techniques [11] for translating aggregate assignments. These other techniques can give rise to more efficient code.

It is hoped to produce methodologies which take an assignment as given in this paper and reformulate it so as to derive the more efficient program structures which result from the sort of approach given in [11].

However, at present, ideas for doing this are only vaguely formulated and the problem is left as "future research".

Conclusions

Assignments between aggregates can be difficult to formulate using conventional program structuring techniques. The problem is essentially one of "co-ordinating" two processes which are utilized in a pseudo-parallel manner. The problem is especially severe when each of the processes are complicated in their own right and there is a "structural mismatch" between them.

Examples 1 to 4 show how certain aggregate assignments can be formulated in a manner which achieves a "conceptual separation" between the supplier and consumer processes. This conceptual separation is considered to give rise to formulations of assignments which are easier for the programmer to understand than those derived by conventional techniques.

The mechanism given here for translating such assignments is essentially analagous to the use of co-routines, a mechanism which is already known to be well-suited to the pseudo-parallel processing of data paths [2,5].

What is new about the proposals given here is that such mechanisms be invoked *implicitly* by the programmer by expressing aggregate assignments in the form given in Examples 1 to 4.

Certain straight forward assignments (such as Example 5) gain nothing from being formulated in the manner proposed in this paper.

The problem of producing more efficient code remains to be solved. If this further problem can be solved, then it is felt that the facilities discussed here would provide a very useful extension to the expressive power of programming languages.

ACKNOWLEDGEMENT

This work has been partially supported by the National Science and Engineering Research Council of Canada.

[1] J. Backus "Can Programming be Liberated from the van Neumann Style?", CACM, Vol. 21, No. 8, August 1978, pp. 613-641.

[2] J. Bezevin, J.L. Nebut and R. Rannou "Another View of Co-routines", SIGPLAN Notices, Vol. 13, No. 5, May 1978, pp. 23-35.

[3] R.M. Burstall, J.S. Collins and R.J. Popplestone "Programming in POP2", Edinburgh Press, 1972.

[4] J. Earley "High Level Operations in Automatic Programming", Proc. of SIGPLAN Symposium on Very High Level Programming Languages, Aug. 1977, pp. 34-42.

[5] D. Grune "A View of Coroutines", SIGPLAN Notices, Vol. 12, No. 7, July 1977, pp. 75-81.

[6] D.F. Kibler, J.M. Neighbours and T.A. Standish "Program Manipulation Via an Efficient Production System", Proc. of SIGPLAN/ SIGART Symp. on Artificial Intelligence and Programming Languages, Aug. 1977, pp. 163-173.

[7] D.E. Knuth "Structured Programming with go to Statements", Computing Surveys, Vol. 6, 1974, pp. 261-301.

[8] B.M. Leavenworth and J.E. Sammett "An Overview of Non-Procedural Languages", Proc. of SIGPLAN Symposium on Very High Level Programming Languages, Aug. 1977, pp. 1-12.

[9] D.B. Loveman "Program Improvement by Source-to-Source Transformations", JACM, Vol. 24, Jan. 1977, pp. 121-145.

[10] A.G. Middleton "A Transformation Approach to Implementing Aggregate Operations", Technical Report TR79-017, School of Computer Science, University of New Brunswick, 1979.

[11] A.G. Middleton "On the use of "Fixed" Data Paths for Assignment", Technical Report TR79-019, School of Computer Science, University of New Brunswick, 1979.

[12] N. Wirth "Program Development by Stepwise Refinement", CACM, Vol. 14, No. 4, April 1971, pp. 221-227.