ISSN 0381-9019

ON THE USE OF "FIXED" DATA PATHS FOR ASSIGNMENT

BY

TONY MIDDLETON

TR79-019, AUGUST 1979

ON THE USE OF "FIXED" DATA PATHS

FOR ASSIGNMENT

by

Tony Middleton

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

TR79-019, August 1979

ON THE USE OF "FIXED" DATA PATHS FOR ASSIGNMENT

Tony Middleton

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

Key Words: Data paths, Assignment, Data Abstraction

General: In earlier work [6,8] it was shown how the notion of "data paths" could be used to achieve some degree of "data abstraction" [5]. In [6] it was shown how certain aggregate operators (e.g. find the sum of a set of data values, find the maximum etc.) could be made insensitive to the details of the data path from which data values were extracted. This was achieved by representing the data path by a single body of code (the "access method") which would access all values in the data path (this body of code would correspond to the notion of an "RHS-FIX" used later in this paper).

In [8], it was shown how further operations could be performed on data paths if these data paths were characterized by a set of primitive operations upon them (e.g. initialize data path, get current item etc.). This is the usual approach to abstract data structures: characterizing the abstract data structure by a set of primitive operations which can be supplied for any particular implementation of the abstract data structure.

In [8], there were two basic types of data path:

(i) An LHS data path, which can appear on the left-hand side of an assignment statement.

(ii) An RHS data path, which can appear on the right-hand side of an assignment statement.

This paper will concern itself with two basic strategies for translating an assignment of the form

$$P_L \leftarrow P_R;$$

where P_L is an LHS data path and P_R is an RHS data path. The two basic strategies can be denoted as

$$\text{LHS\_FIX } (P_L) \leftarrow \text{UNFIX}(P_R);$$

and

$$\text{UNFIX}(P_L) \leftarrow \text{RHS\_FIX}(P_R);$$

where the following concepts are used

(i) LHS_FIX constructs a single ("fixed") program component which will access "slots" in the LHS data path.

(ii) RHS_FIX constructs a single ("fixed") program component which will access all data values in the RHS data path.

(iii) An "unfixed" data path is represented by a collection of components; one component for each primitive operation on the data path.

Thus, in an assignment, either the left-hand side or the right-hand side is "fixed" as a single component.

The other, "unfixed" component can be incorporated in the fixed component, by making a sequence of substitutions (as will be seen), to

produce a piece of program which will perform the entire assignment.

If the LHS data path is fixed, then the program structure can be tailored to efficient techniques for processing the LHS data path. On the other hand, if the RHS data path is fixed, the program structure can be tailored to efficient access for processing the RHS data path.

BASIC OPERATIONS

The following basic operations are used

INIT Initialize a data path.

GET Get the current value from a data path (only relevant to RHS data paths).

PUT Store a value at the current location in a data path (only relevant to LHS data paths).

MOVE Advance to next item in the data path

CLEAN-UP Perform any tidying-up operations necessary after all usage of a data path.

EOP Test for end of data path.

A TRIVIAL EXAMPLE

The mechanics of the two approaches are best demonstrated through a simple, trivial example in which two strategies result in the same code. Once the basic mechanics have been presented, non-trivial examples will be given in which the two strategies produce different results.

The examples will be developed informally in a language which is

essentially POP2 [2] (A variety of syntactic liberties are taken and a more restrictive usage of the stack is presumed. For details, see [8], which also gives reasons for considering this type of language as suited to this particular line of research).

Let P_L be an LHS data path (with basic components $INIT_L$, PUT_L, etc.). Then

$$LHS\_FIX(P_L)$$

is the component (in the case of "simple" data paths).

$INIT_L$;
<start_block>
while not (EOP_L v <cond>) do
    PUT_L; $MOVE_L$; <move>
endwhile
$CLEAN\_UP_L$;
<end_block>

the symbols "<" and ">" are used to denote "events" for which substitutions can be made to achieve the effects of some higher-level operation.

If the strategy

$$LHS\_FIX(P_L) \leftarrow UNFIX(P_R);$$

is adopted, then $LHS\_FIX(P_L)$ is generated and, within it, the following substitutions are made.

<start_block> $\Longrightarrow$ $INIT_R$;

<move> $\Longrightarrow$ $MOVE_R$;

<end_block> $\Longrightarrow$ $CLEAN\_UP_R$;

<slot> $\Longrightarrow$ e_R, where GET_R has the form <item: e_R>

<cond> $\Longrightarrow$ EOP_R

The meaning of this will be more clear if an explicit example is given. Consider the assignment

V:VECTOR ← L:LIST

where V is a vector whose storage has been already allocated. The LHS data path V:VECTOR would be represented by the components (see [8].

$INIT_L$ ι ← 1;

PUT_L [ι] ← <slot>;

MOVE ι ← ι + 1;

EOP_L ι> size(V)

CLEAN_UP ϕ (ie. null string).

where ι is a system-generated symbol, unique for each application of this rule (likewise for any other lower-case identifiers which are extracted from mid-air).

The RHS data path L:LIST is represented by the components

$INIT_R$ ℓ ← L;

GET_R <item:head(ℓ)>

$MOVE_R$ ℓ ← tail(ℓ);

EOP_R $\quad \ell$ = nil

$CLEAN\_UP_R$ $\quad \phi$

If we now perform the above substitutions, the result is:

```
ι ← 1;  ℓ ← L;
while not ((ι> size (V)) ∨ (ℓ = nil)) do
      V[ι] ← head(ℓ);
      ι ← ι + 1; ℓ ← tail(ℓ)
endwhile;
```

The second strategy, is to generate $RHS\_FIX(P_R)$, which, in the case of a "simple" data path, is the component

```
<start_block>
INIT_R:
while not  (<cond> ∨ EOP_R) do
      GET_R; <move>; MOVE_R;
endwhile;
<end_block>
CLEAN_UP_R;
```

and to make the following substitutions within this component:

<start_block> $\Longrightarrow$ $INIT_L$;

<move> $\Longrightarrow$ $MOVE_L$;

<end_block> $\Longrightarrow$ $CLEAN\_UP_L$;

<item:e_R> causes the substitution

<slot> $\Longrightarrow$ e_R within PUT_L

(which then replaces <item:e_R>

<cond> $\Longrightarrow$ EOP_L

If these substitutions are made in RHS_FIX (L:LIST), which is the component

```
<start_block>
ℓ ← L;
while not (<cond> ∨ (ℓ=nil)) do
      <item:head(ℓ)>; <move>;
      ℓ ← tail(ℓ);
endwhile;
<end_block>
```

Then we get exactly the same result as for the first assignment strategy.

SERIAL COMPOSITION OF PATHS

Let

$$X = \langle x_1, x_2, \ldots, x_n \rangle$$

denote that X is a data path whose elements are $x_1, x_2, \ldots, x_n$.

Let $Y = \langle y_1, y_2, \ldots y_m \rangle$

Then

$$X \oplus Y$$

will be used to denote

$$\langle x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \rangle$$

which is the "serial composition" of X and Y (as in [3]). This sequence will also be referred to as the "catenation" of X and Y.

Now consider an assignment such as

V1:VECTOR ⊕ V2:VECTOR ← V3:VECTOR ⊕ V4:VECTOR

where it is assumed that storage has already been allocated for all these vectors (some treatment of the storage allocation problem is given in [8]).

This could be handled as follows

V:VECTOR ← V3:VECTOR ⊕ V4:VECTOR;
V1:VECTOR ⊕ V3:VECTOR ← V:VECTOR;

where V is a vector of suitable size. However, this approach would involve extra storage for the vector V.

Suppose we wished to avoid use of this extra (data) storage, and, to this end, were prepared to put up with the following penalties:

(i) More complicated translation mechanisms.

(ii) Extra program storage, needed for a more complicated assignment technique (this storage being less than the data storage which would be otherwise incurred).

(iii) Slower execution (should this be the case).

Now, in the general case, we cannot assume that V1 and V3 are the same size and that V2 and V4 are the same size (which would lead to an obvious simplification). This being so, we are left with a choice between

LHS_FIX(P_L) ← UNFIX(P_R);

and UNFIX(P_L) ← RHS_FIX(P_R);

where P_L = V1:VECTOR ⊕ V2:VECTOR and P_R = V3:VECTOR ⊕ V4:VECTOR

(ignoring a completely different stragegy, mentioned below, which is considered to be outside the scope of this paper).

THE FIXED VERSION OF $P_1 \oplus P_2$

The fixed version of a catenated LHS path will now be given (the analogy for RHS paths is obvious).

Let P_1 and P_2 be simple, LHS data paths (without being too formal, a "simple" data path is a one-level data path which does not involve "path constructor" operations, such as catenation, parallel composition [3], etc. e.g. V:VECTOR, L:LIST, etc.).

Then

$$\text{LHS\_FIX } (P_1 \oplus P_2)$$

can be translated as

$$\text{START}(\text{LHS\_FIX}(P_1));$$
$$\text{STOP } (\text{LHS\_FIX}(P_2));$$

where

(i) START(P) means "generate P,
then throw away the <end_block> event".

(ii) STOP(P) means "generate P, then throw away the
<start_block> event".

Thus

LHS_FIX(V1:VECTOR ⊕ V2:VECTOR)

would be

```
ι1  1;
<start_block>
while not ((ι1 > size(V1)) v <cond>) do
      V1[ι1] ← <slot>; ι1 ← ι1 + 1;
      <move>
endwhile;
ι2 ← 1;
while not ((ι2 > size(V2)) v <cond>) do
      V2[ι2] ← <slot>; ι2 ← ι2 + 1;
      <move>
endwhile;
<end_block>
```

The assignment

```
V1:VECTOR ⊕ V2:VECTOR ← V:VECTOR
```

can now be translated by the procedure given above (for the case where the LHS data path is fixed). This yeilds

```
ι1 ← 1; ι ← 1;
while not ((ι1 > size(V1)) v (ι > size(V))) do
      V1[ι1] ← V[ι]; ι1 ← ι1 + 1;
      ι ← ι + 1;
endwhile;
ι2 ← 1;
while not ((ι2 > size(V2)) v (ι > size(V))) do
      V2[ι2] ← V[ι]; ι2 ← ι2 + 1;
      ι ← ι + 1;
endwhile;
```

Before proceeding, it is worth assessing the code generated here. On the positive side:

(i) It works.

(ii) Such techniques, when applied to the full assignment

V1:VECTOR ⊕ V2:VECTOR ← V3:VECTOR ⊕ V4:VECTOR

will enable us to avoid generation of an intermediate result (as outlined below).

(iii) The source statement is at high-level and the programmer is relieved of inessential (and "bug-creating"?) detail.

On the negative side, the test "ι > size(V)" could be omitted without changing the effect of the code (in the case where #V = #V1+#V2, where # is the size-operator [6]).

At present, I have no methodology for incorporating such optimizations within the translation techniques (but expect some progress in this respect).

The assignment

V:VECTOR ← V1:VECTOR ⊕ V2:VECTOR

can be translated by the alternative stragegy using the fixed version of the RHS data path.

THE UNFIXED VERSION OF $P_1 \oplus P_2$

Let P_1 and P_2 be two LHS data paths (treatment of RHS data paths follows by analogy).

The components of the unfixed version of $P_1 \oplus P_2$ can be constructed as follows

```
INIT    INIT1; INIT2; first ← true;
MOVE    if first then
           if EOP1 then
              first ← false;
                 MOVE2;
           else
                 MOVE1;
           endif
        else
           MOVE2
        endif;
EOP     if first then
           if EOP1 then
              first ← false;
              EOP2
           else
              false
           endif
        else
           EOP2
        endif;
```

```
PUT         if first then
               if EOP_1 then
                  first ← false
                     PUT_2;
               else
                     PUT_1;
               endif
            else
               PUT_2;
            endif;
CLEAN_UP    CLEAN_UP_1; CLEAN_UP_2;
```

This, of course, is a "severely unfixed" form of $P_1 \oplus P_2$ based on the following philosophy:

(i) This is part of a larger piece of work in which it is intended to base several abstractions on the notion of a data path (with objectives similar to [3]). This is expected to produce hierarchies of abstractions based upon abstractions (rather like [1]).

(ii) This being the case, when the basic operators are defined for a data path, we do not necessarily know how they are to be used.

(iii) In particular, no assumptions (or very few) can be made about the relative execution of the basic operators.

For example, no assumption is made to the effect that some single operator is responsible for setting "first" correctly after the path P_1 is exhausted. This is because the operator definitions, as given, embody very few assumptions about the relative sequence of execution of different operators.

In the context of the single abstraction "assignment", more is known about the relative sequencing of the basic operators, e.g.

(i) EOP is always false prior to the execution of a PUT or MOVE operation.

(ii) PUT and MOVE are always used in the sequence

PUT; MOVE;

(so there is no need for MOVE to do anything which would inevitably be performed by PUT).

(iii) EOP can be made solely responsible for setting "first" correctly.

Thus, in this context, the operators can be simplified. Typically, MOVE can be defined as

if first then $MOVE_1$;
else $MOVE_2$; endif;

In general, basic operators (low level abstractions) can be made more efficient if we know some constraints on their usage in higher-level abstractions (in particular, constraints on relative execution). There are some interesting general questions here, but they will not be pursued. They are beyond the scope of this paper (and, at present, beyond the author's detailed comprehension!).

NON-TRIVIAL CASES

The exact nature of the basic operators chosen to represent an unfixed data path depends upon assumptions about their usage in higher-level abstractions, as indicated above.

Regardless of how these choices are made, one is still left with the basic problems of assignment strategy.

If an assignment such as

V1:VECTOR ⊕ V2:VECTOR ← V3:VECTOR ⊕ V4:VECTOR

is to be translated without the use of intermediate storage, then either the LHS data path or the RHS data path must be unfixed (here, the choice does not matter: in other cases it does), and substitutions made as before using the other, unfixed data path. Enough detail has been given here to show how the above assignment can be mechanically translated (fixing either the LHS data path or the RHS data path).

The derivations will not be given here (they are laborious, but obvious).

Another example will be given based on the notion of a "merged" data path. The notation

$$\text{MERGE}(P_1, P_2)$$

where P_1 and P_2 must both be RHS data paths, and are assumed to be sorted, will merge P_1 and P_2 into a single, sorted data path.

The unfixed version of $\text{MERGE}(P_1, P_2)$ can be constructed as follows:

INIT	INIT_1; INIT_2;
EOP	$\text{EOP}_1 \wedge \text{EOP}_2$

```
GET         if EOP1 then GET2
            else
               if EOP2 then GET1
               else
                  if VALOF(GET1) < VALOF(GET2) then
                     GET1
                  else
                     GET2
                  endif
               endif
            endif
MOVE        if EOP1 then MOVE2
            else
               if EOP2 then MOVE1
               else
                  if VALOF(GET1) < VALOF(GET2) then
                     MOVE1
                  else
                     MOVE2
                  endif
               endif
            endif
CLEAN_UP    CLEAN_UP1; CLEAN_UP2;

where       VALOF(<item:e>) = e.
```

This is, again, a "severely unfixed" form of the data path, and remarks similar to those for unfixed catenated paths are applicable.

Now there is a well-known strategy for using merged data paths [4] which, in the terminology of this paper, is equivalent to defining

$$\text{RHS\_FIX}(\text{MERGE}(P_1, P_2))$$

as follows

```
INIT1; INIT2;
<start-block>
while not (EOP1 ∨ EOP2) do
      if VALOF(GET1) < VALOF(GET2) then
         GET1
      else
         GET2
      endif
      <move>
endwhile;
if EOP1, then
   while not EOP2 do
         GET2; <move>
   endwhile;
else
   while not EOP1 do
         GET1; <move>
   endwhile;
CLEAN_UP1; CLEAN_UP2;
<end_block>
```

Now consider the assignment

$$P_1 \oplus P_2 \leftarrow \text{MERGE}(P_3, P_4);$$

How is this to be translated? Ideally, the translator would perform some analysis of time/storage trade-offs and make the decision on this basis. I have studied this possibility [7] without too much success: my own view being that exact analysis of such situations is probably too laborious to justify the savings made in most cases.

My own anticipation is that, in such a situation, the translator should use approximate rules such as "unfixed merged paths are worse than unfixed catenated paths" or "if in doubt, fix the left hand side", or whatever. These rules could be varied to "tune" the system.

Certain cases will be obvious. E.g. for the assignment

$$P_1 \leftarrow P_2 \oplus P_3$$

the RHS data path would be fixed.

For

$$P_1 \oplus P_2 \leftarrow P_3$$

the LHS data path would be fixed. For other cases, the translator will have to make a "best guess".

This paper does not concern itself with how to make that "best guess".

The sole intent of this paper has simply been to delineate the

mechanics of two alternative stragegies for performing assignments between data paths.

These are not the only strategies possible, as will be briefly indicated below.

OTHER STRATEGIES

There are other strategies for assignment which are considered outside the scope of the present paper.

Consider the assignment

$$P_1 \oplus P_2 \leftarrow P_3 \oplus P_4;$$

This could be dealth with by using four loops, one for each of the following purposes:

(i) Assignment between P_1 and P_3

(ii) Assignment between P_1 and P_4

(iii) Assignment between P_2 and P_3

(iv) Assignment between P_2 and P_4

These four loops would be connected by an obvious conditional structure.

The assignment

$$P_1 \oplus P_2 \leftarrow \text{MERGE}(P_3, P_4);$$

suggests yet another strategy. This could be treated as

$$P_1 \leftarrow \text{OPEN}(\text{RHS\_FIX}(\text{MERGE}(P_3, P_4)));$$

$$P_2 \leftarrow \text{CLOSE}(\text{RHS\_FIX}(\text{MERGE}(P_3, P_4)));$$

where

$$\text{OPEN}(\text{RHS\_FIX}(\text{MERGE}(P_3, P_4)))$$

is a fixed, efficient path which will commence processing of items from $\text{MERGE}(P_3, P_4)$

$$\text{CLOSE}(\text{RHS\_FIX}(\text{MERGE}(P_3, P_4)))$$

is a fixed, efficient path which will continue processing of items from $\text{MERGE}(P_3, P_4)$

CONCLUSION

As programming languages proceed to higher levels, translators will have more and more responsibility for debating alternative programming strategies (at present, this is primarily the responsibility of the programmer).

Assignment between data aggregates is one feature which must be present in any higher level language (implicitly or explicitly).

It is felt that this paper has demonstrated the relevance of the notions of fixed and unfixed data paths in debating strategies for such higher level assignments.

REFERENCES

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

[2] R.M. Burstall, J.S. Collins and R.J. Popplestone — Programming POP2, Edinburgh Press, 1972.

[3] J. Earley — High Level Operations in Automatic Programming. Proc. of Symp. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 34-42.

[4] M. Elson — Data Structures, Science Research Associates Inc., 1975.

[5] B. Liskov and L. Zilles — Programming with Abstract Data Types, Proc. of Symp. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 50-59.

[6] A.G. Middleton — A Transformation Approach to the Implementation of Aggregate Operations, Technical Report TR79-017, School of Computer Science, Univ. of New Brunswick.

[7] A.G. Middleton — Initial Work on a Program Analyser, Dept. of Mathematics, Univ. of Southampton, 1975.

[8] A.G. Middleton — On Assignment Between Data Paths, Technical Report TR79-018, School of Computer Science, Univ. of New Brunswick.