ISSN 0381-9019

ON ASSIGNMENT BETWEEN DATA PATHS

BY

TONY MIDDLETON

TR79-018, JULY 1979

ON ASSIGNMENT BETWEEN DATA PATHS

by

Tony Middleton

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

Abstract

This paper discusses an approach to the manipulation of "data paths". In particular, assignment between data paths is discussed. The notion of a "data path" is somewhat similar to that of an "abstract data structure" except that there is greater emphasis on the fact that elements are processed sequentially. The work is part of a broader effort to allow higher-level operations to be applicable to a wider range of data paths. The aim is to specify higher-level operations in such a manner that the nature of the data paths being manipulated is conveyed explicitly, and naturally, along with the specification of the operations being performed. Only simple cases are dealt with, but that is considered to reflect "work done so far" rather than inherent limitations of the approach.

Earlier Work

As pointed out by Leavenworth and Sammet (1974), "aggregate operations" are a very effective means of enhancing the expressive power of a language. These are operations which can be applied to a whole set of data values at once. This allows the programmer to avoid the "blow by blow" account of operations at the atomic level (an undesirable situation, as argued by Backus, 1978). Earley's paper (1974) is a good example of the expressive power which such operations can provide.

In an earlier paper (Middleton, 1979) it was shown that data paths could be represented in such a manner that certain aggregate operators could be applied to a variety of data paths thus providing some degree of "data abstraction" (see Liskov and Zilles, 1974).

As an example, the data path

```
<start_block>
 ℓ ← L;
while ℓ ≠ nil do
      <item:head(ℓ)>
      ℓ ←  tail(ℓ);
endwhile
<end_block>
```

can be used to process all the items in the list L. This data path contains the following three "events":

(i) <start_block>.

The start of processing of an aggregate

(ii) <item:head(ℓ)>

This indicates the availability of a data item. The expression used to access that item ("head(ℓ)") is conveyed by this event.

(iii) <end_block>

The end of processing of the aggregate.

Certain aggregate operators can be implemented by making substitutions for events in the data path. E.g. the SUM operator can be implemented by the substitutions

```
<start_block>  ==>  s ← 0;
     <item:e>  ==>  s ← s + e;
  <end_block>  ==>  <item:s>
```

The application of the SUM operator to the above data path would yield

```
s ← 0;
ℓ ← L;
while  ℓ ≠ nil  do
        s ← s + head(ℓ);
        ℓ ← tail(ℓ);
endwhile;
<item:s>
```

The earlier paper discussed both one-level and two-level paths (a two-level path is an "aggregate of aggregates", such as a list of sub-lists). This paper will confine itself to the discussion of one-level paths.

Types of Programming Language

Before proceeding, it is worth discussing the two broad classes of programming language to which one might consider applying the techniques discussed here.

(i) Dynamic-Type Languages

The type of a variable can vary at run-time. POP2, SNOBOL and LISP fall into this class.

(ii) Fixed-Type Languages

In these languages, the type of a variable is fixed throughout execution. FORTRAN, PASCAL and Algol68 fall into this class (overlooking sublteties about the use of unions in Algol68 and the use of variants of a record in PASCAL).

Supplying an exact, formal distinction between the two classes would be difficult, but the distinction is a very useful basis for certain general arguments.

The language used here corresponds most closely to POP2 (see Burstall et al, 1972). The differences are primarily a question of syntax: those which are semantic take the form of avoiding certain semantic features present in POP2. Thus, the paper presupposes no essential semantic features which are outside the capabilities of POP2 (the same applies to SNOBOL). The differences between the language used in the paper and POP2 are

(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

would have to be scanned to extract this information (and even that would not be enough in the case of partially used structures).

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

Representation of Data Paths

A data path will be represented by a collection of operations that can be performed on it. This is different to the approach used in earlier work in which a data path was represented by a single textual component (the "access method") which would access all data elements sequentially.

There are two broad classes of data paths.

(i) Left-hand side (LHS) data paths which appear on the left-hand side of an assignment.

(ii) Right-hand side (RHS) data paths, which appear on the right-hand side of an assignment (earlier work concerned itself solely with RHS data paths).

The basic operations on data paths are as follows:

INIT - This operation initialises the data path, before any elements are processed.

GET - This operation accesses the current value in a data path. The operation only applies to RHS data paths.

PUT - This operation stores a data value in the current position in a data path. The operation only applies to LHS data paths.

MOVE - This operation causes the data path to be advanced to the next data item.

EOP - This is a test for the end of the data path.

CLEAN_UP - This is relevant when certain tidying-up operations are needed after all usage of a data path. In most cases it is not relevant and would be represented by the null string. In such cases, no explicit indication of its usage will be given (the correct point in a program structure at which a clean-up operation should be inserted being obvious in most cases).

The Basic Assignment Strategy

An assignment will have the form

$$P_L \leftarrow P_R$$

where P_L is an LHS data path and P_R is an RHS data path. In more detail, the form will be

$$I_L:F_L \leftarrow I_R:F_R$$

where

I_L is an identifier from which the entire LHS data path can be accessed (the "root" of P_L).

F_L is the form of the LHS data path

I_R, F_R have a similar meaning for the RHS data path.

As an example, the assignment

L:NEW_LIST ← V:VECTOR

would assign all the data values in the vector V to the list L, creating new list cells as individual elements are assigned.

The LHS data path L:NEW_LIST can be represented by the components

INIT L ← nil;

PUT L ← L <> cellof (<slot>, nil);

MOVE ∅ (i.e., null string)

EOP false

CLEAN_UP ∅

The "event" <slot> indicates that point in the text which would receive the value supplied by the RHS data path. (<> denotes list concatenation)

The RHS data path V:VECTOR can be represented by the components

INIT i ← 1;

GET <item:V[i]>

MOVE i ← i + 1;

EOP i > size(V)

CLEAN_UP ∅

where i is a system-generated identifier, which is unique for each invocation, and "size" is a run-time routine which interrogates the size of the vector.

The general structure for the assignment

$$P_L \leftarrow P_R$$

is

$INIT_L$; $INIT_R$;

while not ($EOP_L \vee EOP_R$) do

    ASSIGN(PUT_L, GET_R);

    $MOVE_L$; $MOVE_R$;

endwhile

$CLEAN_UP_L$; $CLEAN_UP_R$;

Where the ASSIGN (compile-time) operation will search PUT_L for an occurrence of a slot-event and replace it by an expression taken from the item-event in GET_R;

In the case of

L:NEW_LIST ← V:VECTOR

this yields

```
L ← nil; i ← 1;
while not (false ∨ (i > size(V))) do
     L ← L <> cell of (V[i], nil);
     i ← i + 1;
endwhile;
```

If the translator has any facility for manipulating logical expressions, e.g. for exploiting equivalences such as

$$\underline{false} \vee x \equiv x$$

$$x \wedge \underline{true} \equiv x$$

$$\underline{not}\ (x \vee y) \equiv (\underline{not}\ x) \wedge (\underline{not}\ y)$$

and so on, then the above can be reduced to;

```
L ← nil; i ← 1;
while not (i > size(V)) do
    L ← L <> cellof (V[i], nil);
    i ← i + 1;
endwhile;
```

This works, but it may not be the best way to do things as the <> operator may well be implement in such a way that the whole of L is scanned each time an item is added to the list.

This problem can be overcome by representing the LHS data path L:NEW_LIST by the components

```
INIT      L ← cellof (nil, nil); ℓ ← L;
PUT       tail(ℓ) ← cellof (<slot>, nil);
MOVE      ℓ ← tail(ℓ);
EOP       false
CLEAN_UP  L ← tail(L);
```

This would result in the code

```
L ← cellof (nil;nil); ℓ ← L; i ← 1;
while not (i > size(V)) do
    tail(ℓ) ← cellof (V[i], nil);
    ℓ ← tail(ℓ); i ← i + 1;
endwhile
    L ← tail(L);
```

Here ℓ is a system-generated identifier used for pointing to the last cell in the list L as it is being constructed. L would originally have one redundant cell at the front, but this would be removed at the end of all assignments by the clean-up operation.

Here, two techniques have been given for building up a linked list: one good, one bad. In experienced POP2 programmers often use the concatenation operator in such a way that a list is (unnecessarily) rescanned every time an item is added to it. One advantage of higher-level operators is that such aspects of "good practice" can be implemented by the translator. (A further example would be the insertion of early exits from a loop when translating $\exists$ -operators and $\forall$-operators).

Even between lists and vectors, we have obviously not yet discussed all the problems. Remaining problems will be discussed once a few background concepts have been introduced.

Unary and Binary Data Paths

The data paths L:NEW_LIST and V:VECTOR above are examples of unary data paths. They can be considered as sequences of single items:

$$<x_1, x_2, \ldots x_n>$$

A binary data path can be considered as a sequence of pairs, so:

$$<(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)>$$

In a unary data path, value-producing events will have the form

<item:e>

whereas, in a binary data path, value-producing events will have the form

$$<\text{item: } e_1, e_2>$$

One can easily imagine more general, n-ary paths, but they will not be covered in this paper.

<u>Function Application</u>

Let P be a unary (RHS) data path, and let GETp be the GET component for P. If f is a unary function (one input, one output) then f can be applied to P by making the following substitution in GETp

<item:e> $\Longrightarrow$ <item:f(e)>

If P is a binary (RHS) data path, and if f is a binary function (two inputs, one output), then the substitution used is

<item:e_1,e_2> $\Longrightarrow$ <item:f(e_1,e_2)>

Application of a unary function to a unary data path produces another unary data path. Application of binary function to a binary data path produces a unary data path. Other cases can be imagined, but will not be dealt with here.

It is implicit in all the above that the number of input/output parameters of a function can be known and fixed. Where variadic functions could be allowed, some special notation should be used to convey the fact that the number of input/output arguments is fixed. E.g. in POP2 a function to find the larger of two items would be defined so

```
FIXED FUNCTION MAX2 A B ⟹ C;
IF A > B THEN A ELSE B CLOSE → C;
END;
```

(POP2 uses "close" where I use "endif").

The "fixed" would convey the fact that the function has a fixed number of input/output parameters. For certain functions, this imposes no significant restriction on the use of the function (e.g. those functions which correspond

to scalar functions in APL).

Functions (essentially record selectors) can be applied to LHS paths. This will be covered later.

Parallel Combination of Data Paths

Parallel combination applies only to RHS data paths. Let X and Y be two unary data paths, where

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

then

$$X \,||\, Y$$

will be used to denote the "parallel combination" (Earley, 1974) of the two data paths X and Y, and represents the sequence

$$\langle (x_1,y_1),(x_2,y_2),\ \ldots\ (x_k,y_k) \rangle$$

where k is the lesser of m and n.

Let P be the data path $P_A||P_B$, where, of course P_A and P_B are both unary (RHS) data paths. The components of P are as follows

INT	$INIT_A$; $INIT_B$;
GET	<item:e_a,e_b> where GET_A and GET_B are assumed to have the forms <item:e_A> and <item:e_B>, respectively.
MOVE	$MOVE_A$; $MOVE_B$;
EOP	$EOP_A \vee EOP_B$
CLEAN_UP	$CLEAN_UP_A$; $CLEAN_UP_B$;

An Example
(Involving the RHS_FIX of a Data Path)

In earlier work, the use of a single body of text which would access all elements in a data path corresponded to the use of, what will be termed here, the "RHS_FIX" of a data path. Lets consider the evaluation of the expression

MAX(MAX2(L:LIST||FRONT(V:VECTOR)))

where

(i) Here, L:LIST is an RHS data path, slightly different to the list data paths use previously.

(ii) V is assumed to be a vector of pairs. Pairs are made up of the two components "front" and "back" and are not regarded as being the same as list cells (which place special restrictions on the nature of their second component).

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

INIT ℓ ← L;

GET <item:head(ℓ)>

MOVE ℓ ← tail(ℓ);

EOP ℓ = <u>nil</u>

CLEAN-UP ∅

Now, in fact, the expression to be evaluated is best expressed more explicitly as

MAX(RHS_FIX(MAX2(L:LIST || FRONT(V:VECTOR))))

where RHS-FIX(P) is the component

```
<start_block>
INIT_p;
while not EOP_p do
      GET_p;
      MOVE_p;
endwhile;
CLEAN_UP_p;
<end-block>
```

where GET_p will contain an item-producing event (or "events" - but no such case is treated here).

As shown in earlier work, certain aggregate operations can be implemented by making suitable substitutions in such a component. In the case of the MAX operator, which is applied to a unary data path (not to be confused with the scalar function MAX2), the substitutions are:

```
<start:block>  ==>  m ← -∞;
   <item:e>    ==>  if e > m then
                       m ← e
                    endif;
  <end:block>  ==>  <item:m>
```

Now RHS_FIX(MAX2(L:LIST || FRONT(V:VECTOR)))

is the component

```
<start-block>
ℓ ← L; i ← 1;
while not ((ℓ=nil) v (i> size(V))) do
     <item:max2(head(ℓ), front(V[i]))>
      ℓ ← tail(ℓ); i ← i + 1;
endwhile;
<end-block>
```

If the rules for the MAX operator are now applied, the result is

```
M ← - ∞
ℓ ← L; i ← 1;
while not ((ℓ=nil) ∨ (i > size(V))) do
    if max2(head(ℓ), front(V[i])) > m then
        m ← max2( head(ℓ), front(V[i]))
    endif
    ℓ ← tail(ℓ); i ← i + 1;
endwhile;
<item:m>
```

All has proceeded well except for the redundant occurence of the expression "max2(.....)". There are three possible approaches to this

(i) Leave such problems to be resolved by some optimising translator

(ii) In the MAX operator, use the substitution

```
<item:e>  ⟹   t ← e;
               if t > m then
                  m ← t
               endif;
```

where t is a system-generated identifier. (This substituion would be used where "e" involves non-trivial calculations).

(iii) Take a somewhat more general approach than (ii) and allow item-producing events in general to be affected by substitutions of the form

```
<item:e> ⟹ t ← e; <item:t>
```

(with obvious variations in the case of binary data paths). This means that we can no longer assume that GET-components have the form <item:e> (or $<item:e_1,e_2>$ in the case of binary paths). Such features would no doubt complicate the rules for manipulation of data paths (but probably not in such a way as to seriously hinder progress).

Without wishing to trivialise such problems, they are regarded as outside the scope of the current paper.

Type Information

The forgoeing has suggested that the proposed constructs can be translated without regard to "type" information. In a sense, this is true inasmuch as when processing atoms we don't care if the atoms are reals, integers, strings etc. In fact "atoms" can be anything which is "atomic" with respect to the process being performed. However, a certain amount of type information is essential in translating expressions such as

MAX(MAX2(L:LIST || FRONT(V:VECTOR)))

E.g. we need to be aware of such "types" as

unary (scalar) function	(USF)
binary (scalar) function	(BSF)
unary data path (fixed)	(UDPF)
binary data path (fixed)	(BDPF)
unary data path (unfixed)	(UDPU)
binary data path (unfixed)	(BDPU)

and the translator would have certain "type rules" of the form.

RHS_FIX (UDPU)	produces	UDPF
RHS_FIX (BDPU)	produces	BDPF
UDPU ‖ UDPU	produces	BDPU
UDPF ‖ UDPU	produces	foul!
USF (UDPF)	produces	UDPF
USF (UDPU)	produces	UDPU
BSF (UDPF)	produces	foul!
BSF (BDPF)	produces	UDPF
BSF (BDPU)	produces	UDPU

and so on. Also, there would be rules for the application of "reducers" (operations which reduce a one level data path to an atom (or a two-level data path to a one-level data path; but two-level data paths are not discussed here)). An expression of the form

REDUCER (UDPU)

would be converted to

REDUCER (RHS_FIX (UDPU))

Such rules will not be formalised in this paper, but clearly a translator would need to manipulate type information of this sort.

In the case of the expression

MAX(MAX2(L:LIST ‖ FRONT(V:VECTOR)))

the "type form" is

REDUCER(BSF(UDPU ‖ USF(UDPU)))

which produces the form

REDUCER(BSF(UDPU ‖ UDPU))

which produces the form

REDUCER(BSF(BDPU))

which procudes the form

REDUCER(UDPU)

which is regarded as

REDUCER(RHS_FIX(UDPU))

which produces the form

REDUCER(UDPF)

which produces "ATOM". Clearly, such rules would need to be extended if two-level paths are to be processed.

General Comment

Let the notation

$$V \sim \text{VECTOR(RECORD}(t_1{:}f_1,\ t_2{:}f_2,\ t_3{:}f_3))$$

mean that "V is a vector of records whose components are f_1, f_2 and f_3 (of types t_1, t_2 and t_3, respectively)"

Declarations of this general nature can be made in fixed-type languages such as PASCAL and Algol68, where data structures can be nested.

In such languages, one often wishes that one could write expressions such as

$$f_2(V)$$

to denote an object which could be treated as a "vector of values of type t_2". However, type checking would not allow this expression.

In a language of the sort proposed here, an expression of the form

$$f_2(V{:}\text{VECTOR})$$

can be thought of as

$$f_2(f_p(V))$$

where f_p is a (notional) function which, when applied to a vector produces a unary data path whose elements are of the same type as the elements of V. "Applying" f_2 to this data path would denote applying f_2 to each element of the data path.

As an example, if t_1 and t_2 were both "real" then one would like to be able to write an expression such as

$$f_1(V) + f_2(V)$$

where '+' is an operator already defined to operate on vectors of reals.

It is felt that the proposals in this paper go some way towards this sort of flexibility (but they do it at the expense of expanding all such higher-level operators at the point of call - a serious problem which is discussed below). It seems to the author that there is much to be gained by incorporating the notion of "sequence" in a programming language and into its type structure. Sequences are given some prominence by Low (1974). Also, the notion of "sequence" is somewhat implied in the Lucid language (Ashcroft, 1977). This should perhaps be contrasted to SETL (Schwartz, 1973) where the set is the primary data structure and sequences ("tuples") are somewhat secondary.

<u>Acceptors</u>

In an assignment such as

L:NEW_LIST ← V:VECTOR

the elements of L are "atomic" with respect to the assignment process.

However, one can be a little more general. The elements of L can be (homogeneous) trees, composed of record structures.

In such cases, there are two general types of assignments

(i) A new (LHS) structure is being created.

(ii) An already existing (LHS) structure is being partially or wholly updated.

In the first case, an "acceptor" expression is used. In the second case, function application is used in a manner somewhat analagous to function application for RHS data paths, (but the functions will be restricted to record selector functions, or combinations thereof).

As an example of an acceptor expression, consider the assignment

L:NEW_LIST(PAIR($1,$2)) ← KEYS:LIST || VALUES:LIST

where

KEYS is a list of keys

VALUES is a list of values to be associated with the keys

L is to be an association list. I.e. a list of pairs whose front item is a key and whose back item is to be its associated value (for any pair in L, the front and back items come from corresponding positions in KEYS and VALUES respectively).

The terms $1 and $2 are short-hand for <slot:1> and <slot:2> respectively.

These are simply parameterised slot-events of the form <slot:i>, where i is the "slot number". For an n-ary path, the slot-numbers indicate which parts of a structure will receive which values from item-producing events in the RHS data path.

In the above assignment, the RHS data path would have an event of the form

$$<item:e_1,e_2>$$

The expression e_1 would be assigned to "slot 1" of the LHS data path, and e_2 would be assigned to "slot 2".

The LHS data path would have the components

```
INIT        L ← cellof (nil, nil); ℓ ← L;
PUT         tail(ℓ) ← cellof (pairof (<slot:1>,<slot:2>), nil);
MOVE        ℓ ← tail(ℓ);
EOP         false
CLEAN_UP    L ← tail(L);
```

This is the same as the (more efficient) LHS data path L:NEW_LIST used above, except that the PUT-component differs from the previous version

```
tail(ℓ) ← cellof (<slot>, nil);
```

in an obvious way; namely, that "<slot>" has been replaced by "pairof (<slot:1>,<slot:2>)". The second expression is a straightforward derivation from the expression "pair($1,$2)" in the original assignment (it is assumed that the constructor function for a record class "rec" is "recof". If this systematic relationship does not exist between names of record classes and their constructor functions, then the translator can maintain suitable association tables to achieve the same effect).

The RHS data path KEYS:LIST || VALUES:LIST would have the components

```
INIT        ℓ' ← KEYS; ℓ" ← VALUES;
GET         <item:head(ℓ'), head(ℓ")>
MOVE        ℓ' ← tail(ℓ'); ℓ" ← tail(ℓ");
EOP         (ℓ' = nil) ∨ (ℓ" = nil)
CLEAN_UP    ∅
```

The expression e_1 would be assigned to "slot 1" of the LHS data path, and e_2 would be assigned to "slot 2".

The LHS data path would have the components

INIT	L ← cell of (nil, nil); ℓ ← L;
PUT	tail(ℓ) ← cell of (pair of (<slot:1>,<slot:2>), nil);
MOVE	ℓ ← tail(ℓ);
EOP	false
CLEAN_UP	L ← tail(L);

This is the same as the (more efficient) LHS data path L:NEW_LIST used above, except that the PUT-component differs from the previous version

tail(ℓ) ← cell of (<slot>, nil);

in an obvious way; namely, that "<slot>" has been replaced by "pair of (<slot:1>,<slot:2>)". The second expression is a straightforward derivation from the expression "pair($1,$2)" in the original assignment (it is assumed that the constructor function for a record class "rec" is "recof". If this systematic relationship does not exist between names of record classes and their constructor functions, then the translator can maintain suitable association tables to achieve the same effect).

The RHS data path KEYS:LIST || VALUES:LIST would have the components

INIT	ℓ' ← KEYS; ℓ'' ← VALUES;
GET	<item:head(ℓ'), head(ℓ'')>
MOVE	ℓ' ← tail(ℓ'); ℓ'' ← tail(ℓ'');
EOP	(ℓ' = nil) $\vee$ (ℓ'' = nil)
CLEAN_UP	Ø

(In a situation like this, the programmer would probably know that the EOP - condition can be simplified, as the lists KEYS and VALUES would most likely have the same length. However, in this limited context, no such deduction can be made by the translator).

The entire assignment would become

```
ℓ' ← KEYS; ℓ" ← VALUES;
L ← cellof  (nil;nil); ℓ ← L;
while not ((ℓ' = nil) ∨ (ℓ" = nil)) do
  tail(ℓ) ← cellof  ( pairof  (head(ℓ'), head(ℓ")), nil);
  ℓ' ← tail(ℓ)'; ℓ" ← tail(ℓ");
  ℓ ← tail(ℓ);
endwhile;
  L ← tail(L);
```

It is considered that this is far less straightforward than

```
L:NEW_LIST(PAIR($1,$2)) ← KEYS:LIST || VALUES:LIST
```

(Incidentally, if the user is suspicious about there always being a list on the left-hand side, he is quite right: certain problems are being deferred for later discussion).

Next, the case of partially updating a structure. This is easy when a unary data path is being assigned. Consider the assignment

```
FRONT(L:LIST) ← L1:LIST;
```

Here, L is a list of pairs, and the contents of L1 are to be assigned to the front components of the pairs in L.

Note, that

```
FRONT(L:NEW_LIST)
```

was not used on the left hand side (it wouldn't make sense). For the above assignment, the LHS data path used is

```
INIT    ℓ ← L;
PUT     front(head(ℓ)) ← <slot>;
MOVE    ℓ ← tail(ℓ);
EOP     ℓ = nil
```

It is worth noting that, had we performed the assignment

```
L:LIST ← L1:LIST;
```

then the LHS data path would be exactly the same, except that the PUT component would be

```
head(ℓ) ← <slot>
```

This latter path corresponds to the (very unusual?) case in which we wish to <u>re</u>-assign, at the top level, to a list of already existing cells. The LHS data path

```
FRONT(L:LIST)
```

can be obtained from the (LHS) data path

```
L:LIST
```

by "applying" FRONT to the left-hand side of the PUT component.

The remaining development of this assignment follows in an obvious way.

The case of partially updating a structure when a binary data path is involved is not so easy.

Suppose L is an already-existing list of triples. Let the three fields

of a triple be "one", "two" and "three".

The following two assignments

$$ONE(L:LIST) \leftarrow V1:VECTOR;$$

$$THREE(L:LIST) \leftarrow V3:VECTOR;$$

would update the one-fields and three-fields of L, using the contents of vectors V1 and V3 respectively. This will work (but is inefficient). However, suppose one wanted to write the assignment in the form

$$\text{whatever} \leftarrow V1:VECTOR \,||\, V3:VECTOR$$

What would "whatever" be?

"Whatever" needs to somehow convey the fact that the updating of the one-fields and the updating of the two-fields have something in common: iteration along the list L (or, at least, they should convey this fact to the translator if the optimisation (of using the same loop to iterate along both the one-fields and the three-fields) is to be easily detected by the translator).

I don't at present have a solution to this problem, and will leave it as "further work". (There is probably a more general problem lurking behind this example. Consider

$$n \leftarrow \#P;$$

$$m \leftarrow MAX(P);$$

where P is some unary data path. Two higher-level operations must be performed on the same data path. How must data paths be represented so that it is easy to "hang several operators on the same data path"?)

Tables

Let's imagine that our language has tables, as in SNOBOL. (square

brackets will be used to denote table accessing, as well as array accessing). The assignment

T:NEW_TABLE ← KEYS:LIST || VALUES:LIST

would have the affect of generating a new table and using it to associate values in VALUES with corresponding keys in KEYS. The LHS data path T:NEW_TABLE would have the components

INIT	T ← TABLE();
PUT	T[<slot:1>] ← <slot:2>;
MOVE	∅
EOP	<u>false</u>
CLEAN_UP	∅

Clearly, this is a binary data path and a binary data path is expected on the right-hand side of the assignment.

A similar LHS data path, T:TABLE, could be used when the table is only being updated. This would be the same as the above data path, except that the INIT component would be null.

Before leaving tables, it is worth suggesting the usefulness of the following three RHS data paths

KEYS(T)	A unary data path supplying the keys of a table
VALUES(T)	A unary data path supplying the values associated with the keys.
CONTENTS(T)	The same as KEYS(T) || VALUES(T), but merging of the two associated loops would follow more easily.

In SNOBOL, these data paths would most probably involve conversion of T to an array, followed by iteration over the array.

All-at-once Allocation

So far, this paper has avoided assignments such as

V:NEW_VECTOR ← L:LIST;

Here, the LHS data path is a "preallocated" data path. Its size must first be determined so that allocation can be performed all-at-once before assigning individual items. This size-value must be extracted from the RHS data path.

In such a situation, one technique which always works (but is often inefficient, as we will see), is the following

$$s \leftarrow \#P_R;$$
$$I_L \leftarrow \text{generate } (P_L, S);$$
$$P_L' \leftarrow P_R;$$

this being for the general case

$$P_L \leftarrow P_R;$$

where P_L is pre-allocated.

The assignment $s \leftarrow \#P_R$ calculates the size of P_R and assigns this value to a system-generated variable s. "generate(P_L,s)" means "allocate storage for the type of structure associated with P_L, use s to determine the size, and assign the resulting structure to the variable used as the 'root' of P_L"

P_L' is "that version of P_L which is relevant once the associated structure has been generated".

The #-operator can be implemented by first forming the RHS_FIX of P_R; in this case

```
<start-block>
ℓ ← L;
 while not (ℓ=nil) do
      <item:head(ℓ)>
      ℓ ← tail(ℓ)
 endwhile;
 <end-block>
```

and making the #-operator substitutions

```
<start_block>  ==>   n ← 0;
    <item:e>   ==>   n ← n + 1;
  <end_block>  ==>   <item:n>
```

(the assignment of n to s follows in an obvious way).

In this case, the general assignment

$$s \leftarrow \#P_R;$$
$$I_L \leftarrow \text{generate}\ (P_L, s);$$
$$P_L' \leftarrow P_R;$$

takes the form:

```
s ← #(L:LIST);
I_L ← generate-vector(s);
V:VECTOR ← L:LIST;
```

The (already allocated) LHS data path V:VECTOR is represented by the components

```
INIT      i ← 1;
PUT       V[i] ← <slot>;
MOVE      i ← i + 1;
EOP       i > size(V)
```

This is the general case. Of course, in this context we know that EOP could be made false: this would cause terminated to be controlled by the RHS data path in a situation where the RHS and LHS data paths are known to be the same size.

Assuming that EOP, above, is taken as "false" the complete assignment

```
V:VECTOR ← L:LIST
```

would be expanded as

```
n ← 0;
ℓ ← L;
while not (ℓ=nil) do
      n ← n + 1;
      ℓ ← tail(ℓ);
endwhile;
s ← n;
V ← generate_vector(s);
i ← 1; ℓ ← L;
while not (ℓ=nil) do
      V[i] ← head(ℓ);
      i ← i + 1; ℓ ← tail(ℓ);
endwhile;
```

which is clearly somewhat more verbose than

```
V:NEW_VECTOR ← L:LIST;
```

Next, consider the assignment

```
VL:NEW_VECTOR ← VR:VECTOR;
```

If this was treated in the same way as the assignment

V:NEW_VECTOR ← L:LIST;

there would be an obvious inefficiency involved in iterating over the whole of VR in order to determine its size!

Such a situation can be overcome if RHS data paths such as VR:VECTOR have their INIT component modified to deliver a size-value, wherever this can be done easily:

INIT $\quad i_R \leftarrow 1;$ <block-size:size(VR)>

The assignment rules could then be modified to treat

$$P_L \leftarrow P_R;$$

as

$$P_L' \leftarrow P_R;$$

(where P_L' differs from P_L in assuming that all allocation is already performed) and, whilst doing this, using the size-event for storage generation. Thus, in this case, the substitution

<block_size:size(VR)> $\Longrightarrow$

VL ← generate_vector(size(VR));

would be made. (In the case where the size-event were not needed, a "throw-away" process would get rid of it after translation of the entire statement).

Doubly-Linked Lists

There are quite a few other data paths which can be dreamed up (also, see Earley, 1974).

As a further example, consider doubly-linked lists. An assignment such as

```
DL:NEW_DLIST ← L:LIST;
```

Let doubly-linked lists be made up of records of type "dcell" having the following components (in the order given)

INFO	the datum stored at the node
BACKWARD	backward pointer
FORWARD	forward pointer

The assignment can be translated by using the following LHS data path (analagous to the more efficient technique for singly-linked lists):

```
INIT      DL ← dcellof (nil, nil, nil); ℓ ← DL;
PUT       forward(ℓ) ← dcellof (<slot>, ℓ,nil);
MOVE      ∅
EOP       false
CLEAN_UP  DL ← forward (DL);
          backward (DL) ← nil;
```

Two points are worth mentioning here

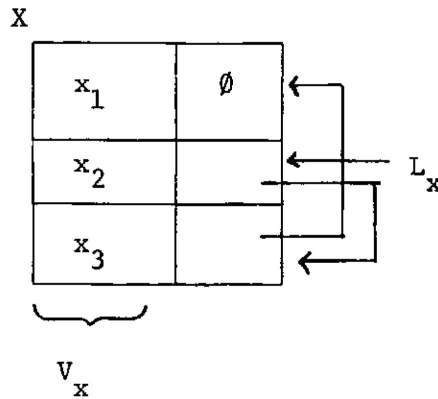
(i) Examples like this involve the manipulation of pointers (explicitly or implicitly). It is well known that pointer manipulations can cause nasty bugs. The delegation of pointer manipulations to higher-level operators might ease this problem.

(ii) If such data structures are used in a "read-only" manner, then a "parameterised" view of them can be taken (see Middleton, 1977) E.g. (what is actually) a doubly-linked list can be viewed as a singly-linked list, if one states the two fields to be viewed as "head" and "tail".

Multiple Structures

Sometimes, two or more (notional) structures are imposed over the same set of data objects.

Consider the structure



This can be considered as a collection of objects (X) over which two notional structures are imposed:

L_x a list which puts the objects of X in the order $\langle x_2, x_3, x_1 \rangle$

V_x a vector which puts the objects of X in the order $\langle x_1, x_2, x_3 \rangle$

An earlier paper (Middleton, 1977) suggests ways in which one might try to provide a certain degree of convenience in handling such structures (in a "read-only" manner) E.g. by extending such ideas, one might be able to handle an expression such as

$$SUM(MAX2(L_x \| V_x))$$

which would allow one to think of notional structures (such as L_x and V_x) in their own right.

However, it is not at all clear how to deal with operations which involve storage-generation/updating, as such operations raise the problem of "interference" between notional (and conceptually separate?) data structures.

Use of Routines

The term "routine" will be used generically for "subroutine", "function", "procedure" etc. without being formal about its exact meaning. Consider the following sequence

```
M1        MAX(V1:VECTOR);
M2        MAX(V2:VECTOR);
M3        MAX(V3:VECTOR);
ML        MAX(L:LIST);
```

Now all of the forgoeing has been based on a macro-processor/text-manipulator treatment of operators. I.e. operators are expanded at the point of call much as in the use of macro-processors.

But, clearly, the above suggests that an expression of the form

MAX(whatever:VECTOR)

might best be treated as a call to a routine, rather than have three expansions of the MAX operator. This (depending on the exact overhead for routines) would normally save on storage at the expense of execution time. But what about the call

ML ← MAX(L:LIST);

Three choices come to mind

(i) Expand the operator at the point of call, using the techniques given in this paper.

(ii) (Assuming a routine is available to compute MAX for vectors). First, convert L to a vector, and then use a routine to compute MAX.

(iii) Use another version of the MAX routine which can operate on lists (this being sensible if MAX is applied to lists elsewhere in the program)

It isn't obvious what to do in such situations, as can be appreciated from Low's work (1978).

Conclusions

This work is in its early stages of development, so its not appropriate to draw very strong conclusions. However, the following remarks seem in order:

(i) The approach allows certain common operations on data structures to be expressed in a concise (and natural?) manner.

(ii) The notation is at a higher level, and suppresses a lot of "administrative" detail (incrementing counters, advancing pointers, initialisations, clean_up operations, etc.).

(iii) The techniques are most naturally mapped into a variable-type language (e.g. POP2, SNOBOL) rather than a fixed-type language (e.g. PASCAL, Algol68). (It is anticipated that some work-in-progress will reinforce this view).

(iv) The last two sections, "Multiple Structures" and "Use of Routines" indicate two serious future problem areas in this research (and for similar research by other people?)

(v) Two-level paths will no doubt present further problems.

It is felt that enough arguments have been presented to suggest that the "data path" approach to implementing aggregate operators is useful. Work is in progress to extend the ideas.

References

E.A. Ashcroft	Lucid Programming, Research Report CS-77-03, Dept. of Computer Science, Univ. of Waterloo, 1977.
J. Backus	Can Programming be Liberated from the von Neumann Style?, CACM, Vol. 21, No. 8, August 1978, pp. 613-641.
R.M. Burstall, J.S. Collins and R.J. Popplestone	Programming in POP2, Edinburgh Press, 1972.
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.
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.
B.M. Leavenworth and J.E. Sammett	An Overview of Non-Procedural Languages, Proc. of Symp. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp 1-12.
B. Liskov and S. 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.
J.R. Low	Automatic Data Structure Selection: An Example and Overview, CACM, Vol. 21, May 1978, pp. 376-385.
A.G. Middleton	A Macro Approach to Abstractions of Data Structures, SIGPLAN Notices, April 1977, pp 75-79.
A.G. Middleton	A Transformation Approach to Implementing Aggregate Operations, Technical Report TR79-017, School of Computer Science, Univ. of New Brunswick.
J.T. Schwartz	The SETL Language and Examples of its Use, Courant Inst. of Math. Sciences, 1973.