ISSN 0381-9019

A TRANSFORMATION APPROACH TO IMPLEMENTING AGGREGATE OPERATIONS

BY

Tony Middleton

TR79-017, June 1979

A TRANSFORMATION APPROACH TO IMPLEMENTING AGGREGATE OPERATIONS

by

Tony Middleton

School of Computer Science
University of New Brunswick
Fredericton, N.B.

TR79-017, June 1979

A TRANSFORMATION APPROACH TO IMPLEMENTING AGGREGATE OPERATIONS

Tony Middleton

School of Computer Science
University of New Brunswick
Fredericton, N.B.

ABSTRACT

Some of the expressive power of certain programming languages depends on the provision of convenient means of operating on collections, or "aggregates", of data values. This paper argues that certain aggregate opertaions could be implemented by applying transformations to fragments of program which access the elements of an aggregate. The proposals are considered to be a viable means of supporting "data abstractions" - i.e. for allowing certain operations in a language to be insensitive to the detailed implementation of a data structure.

1. Introduction

The techniques presented here depend on the notion of a "data path". Typical data paths are

(i) all the elements in a vector

(ii) all the elements in a list

(iii) all the elements in a row of an array

(iv) all the elements in a diagonal of an array,

and so on.

Certain aggregate operators can be applied to a data path. E.g.:

(i) Find the maximum value

(ii) Count the number of items

(iii) Sum the items

(iv) Test for membership of some value amongst the set of items.

(v) Sort the items

and so on. As pointed out in [6], such aggregate operations are an effective means of enhancing the expressive power of a language (also, see [3]).

This paper proposes a new approach to the implementation of aggregate operators. The method depends on applying transformations to programs (rather like [2,5,8]). In this case the transformations used take the form of making substitutions for particular "events" in a program structure. An "event" corresponds to the passage of control through a particular point in a program structure, as in [10].

Here, the sort of events used are

(i) The start of processing of an aggregate

(ii) The availability of a single item from the aggregate

(iii) The end of processing of an aggregate.

Aggregate operations can be implemented by making suitable substitutions for such events.

Once some elementary examples have been presented, the paper goes on to deal with "two-level" structures - i.e. aggregates of aggregates (or, in the terminology used below "groups of blocks").

It is also shown how the techniques proposed can be used to implement "abstract data structures" (see [4,7]). In the approach used here, the implementation of abstract data structures reduces to making aggregate operations insensitive to the nature of the data path to which they are applied.

2. Types of Programming Language

Before proceeding, it is worth discussing the broad class of languages to which the techniques are most easily applied. Strachey (see [9]) outlined three broad classes of programming language, which I have taken the liberty of re-naming. The classes are as follows

(a) Dynamic-Type Languages

The type of a variable can vary at run-time. POP2 (see [1]), LISP and SNOBOL fall into this broad class. Each data item carries type information along with its data value. Such features generally

imply two things

(i) A storage overhead, involved in storing type information

(ii) Slower execution, since even primitive operations such as addition can require the run-time testing of data type.

(b) Fixed-Type Languages

In these languages, the type of a variable is fixed throughout execution. FORTRAN, PASCAL and ALGOL68 fall into this class (if we overlook subtleties involving the use of unions in ALGOL68). These languages tend to have the following features

(i) There is a more static treatment of storage and data values do not carry type information with them

(ii) Type-checking is done by the compiler

(iii) Since type information is known at translation time, the translator can generate more efficient code than is the case for dynamic-type languages.

(c) Type-less Languages

These languages use a "bare-hands" approach. It is up to the user to see that the operations he performs are compatible with the (implicit) "type" of the information he manipulates. Assembler languages fall into this class. All data is essentially a "bunch of bits" to be manipulated as the user pleases. Many systems programming languages fall into this class in that, even if they do have some formal treatment of types, they include a "do-anything" type which is essentially a machine word to be operated on in any manner that the

machine operations will allow.

The "language" used in this paper is an informal notation which corresponds fairly closely to POP2 (except that a few liberties have been taken with syntax. E.g. The right-hand side and left-hand side of assignments are reversed to the more usual ordering, square brackets are used to indicate array accessing, and slightly more convenient control structures are presumed. These changes are essentially syntactic. No facilities are assumed which are outside the essential semantics of POP2). (see[1]). This is a dynamic type language in which many operations on a data structure can be divorced from the type of element contained in the structure, and only depend on the "form" of the structure (e.g. one can define a function to count the number of elements in a linked list which is insensitive to the type of information contained in the list - this is not so in a fixed-type language such as FORTRAN, PASCAL, or ALGOL68).

The isolation of form from content in variable-type languages makes it somewhat easier to demonstrate the methods being proposed here. It is felt that the methods can be extended to be useful for fixed-type languages. However, this extension has been deferrred for later study.

3. Access Methods

Data paths can be characterised by their "access methods". I.e. by a fragment of program which will access all elements in the data path. For example, if v is a vector containing n elements, then the access method for porcessing all elements in v is

```
<start_block>
 for i to n do
      <item: v[i]>
 endfor
<end_block>
```

The symbols "<" and ">" are used to delimit events in the program fragment. Here, the three types of event are

(i) <start_block>: the start of processing of an aggregate

(ii) <end_block>: the end of processing of an aggregate

(iii) <item: v[i]>: the availability of a data value. Here, the event name is followed by an expression ("v[i]") which denotes the available item.

These event-names will be used as markers when substitutions are made to implement aggregate operations. Where an expression accompanies the event name, this expression may be used in the transformation.

Small letters will be used for system-generated identifiers or constants. Here "i" and "n" denote symbols which would be provided by the translator. The user, when applying an aggregate operator, would use some notation such as

SUM(v)

or

MAX(v)

and leave the translator to provide symbols for the array index and the size of the array. Here, details can depend on the language. For

example

(i) In ALGOL68 the expression

upb v - lwb v + 1

would be used for "n" if v was a parameter and the lower bound of v was not known.

(ii) In FORTRAN, "n" could be obtained by inspecting array declarations, in the case where v was referenced in the main array, and where v was fully occupied.

(iii) In FORTRAN, in cases where declarations would give a misleading indication of the number of (useful) data items in an array (e.g. for partially-used arrays), it is recommended that declarations be written in a special form. E.g. the notation

INTEGER v(20/N)

could be used to declare a one-dimensional array of 20 integers in which only the first N elements were actually used. Such information could be used by a pre-processor to determine limits for iteration for implementing aggregate operations.

In this paper, I will simply extract symbols such as "i" and "n" from mid-air and not discuss how the symbols are actually generated in any particular implementation. Such details are not relevant to the general nature of the arguments being presented here. (In a dynamic-type language, such parameters as "n" can be easily obtained as all data objects are "self-describing" and carry sufficient information with them that limits for iteration can be easily obtained by inserting a

call to a suitable run-time routine.)

A suitable access method for processing all elements of a list L would be

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

where ℓ is a system-generated variable used to scan through the list L.

4. Use of Transformations

In this section, it will be shown how transformations can be used to implement certain aggregate operators. ("transformations" here being simple substitutions, rather the more general "tree-for-tree" substitution of [5]).

4.1 The Maximum Operator (MAX)

The transformation used here is

(i) Replace start_block by

$$m \leftarrow -\infty;$$

where "m" is a system-generated identifier and "$-\infty$" denotes a "lower-than-anything" value.

(ii) Replace <item:e> by

```
if e > m then
    m ← e
endif
```

(iii) Replace <end_block> by

<item:m>

indicating the availability of the result m (this could be an input to a higher-level operator in the case of two-level data paths, as will be seen below).

Thus, for the above vector V, the expression

MAX(V)

expands to

```
m ← - ∞;
for i to n do
    if V[i] > m then
        m ← V[i]
    endif
endfor ;
<item:m>
```

In the case of the above list L, the expression

MAX(L)

expands to

```
m ← - ∞;
ℓ ← L ;
while ℓ ≠ nil do
  if head(ℓ) > m then
       m ← v[i]
  endif
endwhile ;
<item:m>
```

The transformation to implement MAX can be written more formally as

```
<start_block>  ⟹   m ← - ∞
<item:e>       ⟹   if e > m then
                       m ← e
                   endif
<end_block>    ⟹   <item:m>
```

where each of the above three rules is of the form

text to be searched for ⟹ text to be used to replace it by

This notation will be used to explain the implementation of other operators.

4.2 Minimum Operator (MIN)

This is an obvious analogy to the MAX operator.

4.3 Membership Test (IN)

Let x IN p

be true if the value x is found in the data path p, false otherwise. The IN operator can be implemented by making the following substitutions in the access method for p

```
<start_block> ⟹ v ← false ;
<item:e>      ⟹ if e = x then
                    v ← true
                    goto lab
                endif
<end_block>   ⟹ lab: <item:v>
```

where v is a system-generated variable and lab is a system-generated label used for early exit from the search loop.

4.4 The Counting Operator (#)

This operator counts the number of items in a data path. It is implemented as follows

```
<start_block> ⟹ v ← 0;
<item:e>      ⟹ v ← v + 1;
<end_block>   ⟹ <item:v>
```

4.5 The Selection Operator (WITH)

This selects all items of a data path which satisfy some logical condition. Thus, it produces one data path from another.

The general form is

$$x \in p \text{ WITH } c(x)$$

where

p - is some data path

x - denotes a typical item from p

c(x) - is some logical condition containing references to x.

The transformation used is

<start_block> ⟹ <start_block>

(ie no change)

<item:e> ⟹ if c(e) then

<item:e>

endif

<end_block> ⟹ <end_block>

Here "c(e)" means "rewrite c(x), substituting e for x".

Thus, using the data path for V above, the expression

X ∈ V WITH X>0

is implemented by making substitutions in the data path

<start_block>

for i to n do

<item:v[i]>

endfor

<end;block>

the substitutions yield the following

```
<start:block>
for i to n do
   if v[i] > 0 then
        <item:v[i]>
   endif
endfor;
<end:block>
```

This block (or data path) can now be operated on in the same manner as any other block.

4.5 The Sum Operator (SUM)

This can be implemented by the substitutions

<start_block> $\Longrightarrow$ s ← 0;

<item_block> $\Longrightarrow$ s ← s + e;

<end_block> $\Longrightarrow$ <item:s>

It can now be shown that a data path defined by a WITH-operator can be treated like any other. Consider the expression

SUM(X ∈ V WITH X>0)

The translation of

X ∈ V WITH X>0

has just been presented. The translation can be completed by making the substitutions for the SUM operator. This yields

```
S ← 0;
for i to n do
   if V[i] > 0 then
      S ← S+V[i]
   endif
endfor
<item:S>
```

4.6 Universal Quantification (ALL)

The expression

(ALL x∈p)(c(x))

is to have the value true if all items in p satisfy the condition c. Otherwise, the value is false. This operator can be implemented by the substitutions

```
<start:block> ⟹ v ←true
<item:e>      ⟹ if not c(e) then
                    v ← false
                    goto lab
                 endif
<end:block>   ⟹ lab: <item:v>
```

where, again v and lab are unique system-generated symbols and do not conflict with similar symbols generated in other translations of aggregate operators.

4.7 Existential Operator (ANY)

The expression

(ANY x∈p)(c(x))

has the value true if any item in p satisfies the condition c. Otherwise, the expression has the value false.

The operator can be implemented by the substitutions

```
<start:block> ⟹ v ← false;

<item:e>  ⟹  if c(e) then
                 v ← true
                 goto lab
             endif;

<end:block> ⟹ <item:v>
```

No doubt, other operators can be invented, but we have enough here to illustrate the mechanisms being proposed.

5. Two-Level Structures

A wider range of situations can be dealt with if "two-level" data paths can be handled. These are data paths which involve "aggregates of aggregates", or "groups of blocks" in the terminology used here. Examples of two-level data paths are

(i) processing rows of an array

(ii) Processing columns of an array

(iii) Processing a list of sub-lists

and so on. As an example, consider processing an array A by rows (each row being a "block"). The associated data path would be

```
<start:group>
 for i to m do
<start:block>
     for j to n do
      <item:A[i,j]>
     endfor
<end:block>
 endfor
<end:group>
```

Such a data path would be indicated by an expression such as

A[i,*]

This meaning "A, divided into rows". Similarly

A[*,j]

would denote the grouping of A into columns. Thus, an expression such as

MAX(SUM(A[i,*]))

would denote finding the maximum row-sum for A. The application of the SUM operator to the group A[i,*] would "reduce" the group to a single block (a data path which produced the row sums).

Only one extension is needed to the rules produced so far: when a "reducing" operator is applied to a group, the substitutions

<start:group> $\Longrightarrow$ <start:block>

<end:group> $\Longrightarrow$ <end:block>

must be made. Bearing this in mind, the expression

SUM(A[i,*])

becomes

```
<start:block>
 for i to m do
     s ← 0
     for j to n do
     s ← s + A[i,j]
     endfor
     <item:s>
 endfor;
<end:block>
```

We now have a block. I.e. A one-level data path supplying items, each of which is a sum.

Now, applying the MAX operator substitutions, we obtain

```
m' ← - ∞
for i to m do
     s ← 0
     for j to n do
         s ← s + A[i,j]
     endfor
     if s > m' then
     m' ← s
     endif
endfor;
<item:m'>
```

(where m' is an identifier generated by the MAX operator).

As another example, the expression

MAX(#(x∈A[i,*] WITH x > 0))

would find the maximum number of positive values in any row of X. This would be translated in the following manner: first, apply the substitutions for WITH

```
<start:group>
 for i to m do
     <start:block>
     for j to n do
       if A[i,j] > 0 then
         <item: A[i,j]>
       endif
     endfor
     <end:block>
 endfor
 <end:group>
```

Now apply the substitutions for the #-operator(counting), bearing in mind that a group is being reduced to a block:

```
<start:block>
for i to m do
    v ← 0
    for j to n do
      if A[i,j] > 0 then
         v ← v + 1
      endif
    endfor
    <item:v>
endfor
<end:block>
```

Finally, apply the substitutions for the MAX operator

```
m' ← - ∞
for i to m do
    v ← 0
    for j to n do
      if A[i,j] > 0 then
         v ← v + 1
      endif
    endfor
    if v > m' then
       m' ← v
    endif
endfor
<item:m'>
```

(where m' is generated by the MAX operator)

It is fairly obvious that the techniques can be extended to arrays of higher dimensionality, and this point will not be discussed.

Only one minor point needs to be cleared up. In the above, the term

<item:m'>

was left dangling in mid air. What would happen, in fact, is that the above translation of

MAX(SUM(A[i,*]))

would take place in some context such as

ROWMAX ← MAX(SUM(A[i,*]))

which would be converted to

ROWMAX ← t

where t is a system-generated identifier. Then, prior to this, the system would insert the above translation of MAX(SUM(A[i,*])), making the substitution

<item:e>⟹t ← e

This would result in the following code

```
m' ← - ∞
for i to m do
    s ← 0
    for j to n do
        s ← s + A[i,j]
    endfor
    if s > m' then
       m' ← s
    endif
endfor;
t ← m';
ROWMAX ← t
```

No doubt with a little bit of effort, the translator could be trained to replace the last two assignments by

ROWMAX ← m'

6. Dealing with Special Data Structures

This approach allows aggregate operators to be applied to specialised data structures, provided that access methods are specified for the data structure. As an example, let's consider "jagged" arrays, in which the number of items in each row can vary. Let the ith row contain the items

$$A_{i1} \quad A_{i2} \quad A_{i3} \quad \cdots \quad A_{iN_i}$$

where N_i is the number of items in the ith row. Assume that such an array is mapped into a vector in the following manner

(i) The first element indicates the number of rows

(ii) This is followed by a pair of elements for each row. The first element specifies the number of elements in the row (N_i). The second element indicates the index of the first element of the row within the vector used to store the jagged array.

(iii) These pairs are followed by the contents of each row of the array.

A suitable access method for such an array would be

```
m ← A[1];          (number of rows)
<start:group>
for i to m do
     n ← A[2*i] ;  (size of row)
     k ← A[2*i+1]; (initial index for row)
     <start:block>
     for j to n do
          <item:  A[k+j-1]>
     endfor
     <end:block>
endfor
<end:group>
```

One could indicate such a data path by syntax such as

JAGGED(A)

or simply by reference to A, if the nature of A were specified in some declaration. Note that such a data path can easily be converted to a one-level data path by the substitutions.

<start:block> ⇒ null string

<end:block> ⇒ null string

<start:group> ⇒ <start:block>

<end:group> ⇒ <end:block>

(The order of applying the substitutions being important!) Such a data path could be denoted by syntax such as

BLOCK(JAGGED(A))

or BLOCK(A)

again, depending on whether the nature of A was conveyed through some declaration.

Clearly an operation such as

MAX(SUM(JAGGED(A)))

could be translated just as effectively for a jagged array as for a regular array, by using the substitution mechanisms already proposed.

At this point, it is worth mentioning a problem area. The #-operator is sometimes translated inefficiently. For example, the expression

MAX(#(JAGGED(A)))

would be translated to inefficient code. The substitution mechanism would result in a loop of the form

```
v ← 0;
for j to n do
    v ← v+1
endfor
```

where, clearly

```
v ← n
```

achieves the same effect. One way to overcome this would be to indicate in the data path the availability of the block size. Some notation such as

```
m ← A[1];
<start:group>
for i to m do
<start:block>
   n ← A[2*i];
   <block size:n>
   k ← A[2*i+1];
   <start:block>
      for j to n do
         <item:  A[k+j-1]
      endfor
   <end:block>
   endfor
   <end:group>
```

The #-operator could replace

```
<block size:n>
```

by

```
<item:n>
```

However, we are now left with a "do-nothing" loop

```
for j to n do
      <item: A[k+j-1]>
endfor
```

If a clean-up operation removes the unused

```
<item: A[k+j-1]>
```

then it is probably not too much to expect the translator to remove a redundant loop of the form

for j to n do

endfor

However, having said these few optimistic words, it must be admitted that the problem is not trivial and places an extra burden on the translator if such inefficiences are to be avoided.

7. Conclusion

This paper has outlined proposals for waht is believed to be a novel approach for implementing aggregate operators in such a way that they can be applied to a variety of data structures (provided "access methods" are supplied). It is hoped to extend the methods to deal with serial and parallel paths, as mentioned in Easley's paper ([3]).

A variety of minor technical problems remain to be solved. However, it is hoped that the informal presentation given here is precise enough to convince the reader (as the author believes) that the methods provide a viable means of extending the variety of data structures to which common aggregate operators can be applied (and therefore, to that extent, provide some means for data abstraction).

ACKNOWLEDGEMENT

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

References

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

[2] R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs". JACM, Vol. 24, Jan. 1977, pp. 44-67.

[3] J. Earley, "High Level Operations in Automatic Programming". Proc. of SIGPLAN Symposium on Very High Level Languages, March 1974, pp. 34-42.

[4] J.V. Guttag and J.J. Horning, "The Algebraic Specification of Abstract Data Types". Acta Informatica, Vol. 10, No.1, 1978, pp. 27-52.

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

[6] B.M. Leavenworth and J.E. Sammet, "An Overview of Non-procedural Languages", Proc. of SIGPLAN Symposium on Very High Level Languages, March 1974, pp. 1-12.

[7] B.H. Liskov, A. Snyder, R. Atkinson and C. Shaffert, "Abstraction Mechanisms in CLU", C.ACM, Vol. 20, No. 8, Aug. 1977, pp. 564-576.

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

[9] C. Strachey, "Varieties of Programming Language", In Infotech State of the Art Report No. 7 on High Level Languages.

[10] C.T. Zahn, "A Control Statement for Natural top-down Structured Programming". Symp. on Programming Languages, Paris, 1974.