Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages

COMPILATION AND DELAYED EVALUATION IN APL

by

Leo J. Guibas and Douglas K. Wyatt

Xerox Palo Alto Research Center
3333 *Coyote Hill Road*
Palo Alto, Cal. 94304

0. Introduction

Most existing APL implementations are interpretive in nature, that is, each time an APL statement is encountered it is executed by a body of code that is perfectly general, i.e. capable of evaluating any APL expression, and is in no way tailored to the statement on hand. This costly generality is said to be justified because APL variables are typeless and thus can vary arbitrarily in type, shape, and size during the execution of a program. What this argument overlooks is that the operational semantics of an APL statement are not modified by the varying storage requirements of its variables.

The first proposal for a non fully interpretive implementation was the thesis of P. Abrams [1], in which a high level interpreter can defer performing certain operations by compiling code which a low level interpreter must later be called upon to execute. The benefit thus gained is that intelligence gathered from a wider context can be brought to bear on the evaluation of a subexpression. Thus on evaluating $(A+B)[I]$, only the addition $A[I]+B[I]$ will be performed. More recently, A. Perlis and several of his students at Yale [9,10] have presented a scheme by which a full-fledged APL compiler can be written. The compiled code generated can then be very efficiently executed on a specialized hardware processor. A similar scheme is used in the newly released HP/3000 APL [12].

This paper builds on and extends the above ideas in several directions. We start by studying in some depth the two key notions all this work has in common, namely *compilation* and *delayed evaluation* in the context of APL. By delayed evaluation we mean the strategy of deferring the computation of intermediate results until the moment they are needed. Thus large intermediate expressions are not built in storage; instead their elements are "streamed" in time. Delayed evaluation for APL was probably first proposed by Barton (see [8]).

Many APL operators do not correspond to any real data operations. Instead their effect is to rename the elements of the array they act upon. A wide class of such operators, which we will call the *grid selectors*, can be handled by essentially pushing them down the expression tree and incorporating their effect into the leaf accessors. Semantically this is equivalent to the *drag-along* transformations described by Abrams. Performing this optimization will be shown to be an integral part of delayed evaluation.

In order to focus our attention on the above issues, we make a number of simplifying assumptions. We confine our attention to code compilation for single APL expressions, such as might occur in an "APL Calculator", where user defined functions are not allowed. Of course we will be critically concerned with the re-usability of the compiled code for future evaluations. We also ignore the distinctions among the various APL primitive types and assume that all our arrays are of one uniform numeric type. We have studied the situation without these simplifying assumptions, but plan to report on this elsewhere.

The following is a list of the main contributions of this paper.

- We present an algorithm for incorporating the selector operators into the accessors for the leaves of the expression tree. The algorithm runs in time proportional to the size of the tree, as opposed to its path length (which is the case for the algorithms of [10] and [12]).

Although arbitrary reshapes cannot be handled by the above algorithm, an especially important case can: that of a *conforming* reshape. The reshape $A\rho B$ is called conforming if ρB is a suffix of A.

- By using conforming reshapes we can eliminate inner and outer products from the expression tree and replace them with scalar operators and reductions along the last dimension. We do this by introducing appropriate selectors on the product arguments, then eventually absorbing these selectors into the leaf accessors. The same mechanism handles *scalar extension*, the convention of making scalar operands of scalar operators conform to arbitrary arrays.

- Once products, scalar extensions, and selectors have been eliminated, what is left is an expression tree consisting entirely of scalar operators and reductions along the last dimension. As a consequence, during execution, the dimension currently being worked on obeys a strict stack-like discipline. This implies that we can generate extremely efficient code that is *independent of the ranks of the arguments.*

Several APL operators use the elements of their operands several times. A pure delayed evaluation strategy would require multiple reevaluations.

- We introduce a general buffering mechanism, called *slicing*, which allows portions of a subexpression that will be repeatedly needed to be saved, to avoid future recomputation. Slicing is well integrated with the evaluation on demand mechanism. For example, when operators that break the streaming are encountered, slicing is used to determine the minimum size buffer required between the order in which a subexpression can deliver its result, and the order in which the full expression needs it.

- The compiled code is very efficient. A minimal number of loop variables is maintained and accessors are shared among as many expression atoms as possible. Finally, the code generated is well suited for execution by an ordinary minicomputer, such as a PDP-11, or a Data General Nova. We have implemented this compiler on the Alto computer at Xerox PARC.

The plan of the paper is this: We start with a general discussion of compilation and delayed evaluation. Then we motivate the structures and algorithms we need to introduce by showing how to handle a wider and wider class of the primitive APL operators. We discuss various ways of tailoring an evaluator for a particular expression. Some of this tailoring is possible based only on the expression itself, while other optimizations require knowledge of the (sizes of) the atom bindings in the expression. The reader should always be alert to the kind of knowledge being used, for this affects the validity of the compiled code across reexecutions of a statement.

1. The Intentional Representation of Expressions

APL, like many other very high level languages, is characterized by its ability to manipulate "large" objects. Thus APL deals with multi-dimensional arrays, SETL [11] deals with sets, LISP deals with lists, etc. The word "large" refers to a comparison between the size of the primitive objects of the language and the complexity of its primitive operations on them, contrasted with the size and complexity of the objects manipulated inside the processor of a present-day computer. Thus an array typically occupies several storage locations and the evaluation of an array sum $A+B$ in APL requires the execution of a number of machine instructions proportional to the size of the arrays A or B.

Note that the semantics of APL, although they completely determine the meaning of an expression in the language, do not fully specify how that expression is to be computed. For example, the semantics of the language leave us free, in evaluating $A+B$, to add the corresponding elements of A and B in whichever order we please. Thus we can regard APL expressions more as a *specification* of the result we desire to compute, rather than as a *detailed algorithm* for evaluation on a serial computer. For the majority of APL operators the cleavage between what the semantics of the language require and what the evaluator is free to choose falls along the following lines. The semantics specify what *data operations* are to be performed, i.e. how each element of the result array depends on some of the elements of the operand arrays. The order in which the result elements are to be evaluated, however, that is the *control* of the computation, is usually left unspecified.

We can often use this freedom in sequencing to advantage, by matching the order in which the result may be required (e.g. for display, according to standard APL conventions) with the orders in which the operands may most conveniently be traversed. In the traditional APL evaluators an operation is executed only after its operands have been fully evaluated. This has the advantage (assuming the usual convention of storing arrays in row major format) that at any moment there is a very efficient way of traversing the arguments of an operation (i.e. the row major order). However, this is not the only possibility. Suppose, for example, that we wish to display the result of $⍉A+B$, where A and B are evaluated matrices. Then, if we traverse A and B in column major order, we can display the result without ever having to generate the intermediate array $A+B$. At the expense of slightly more cumbersome traversal, we have avoided generating a possibly large intermediate array. Furthermore we can optimize our freedom in sequencing over the entire expression we wish to evaluate. There is a simple way of sequencing through $A++/B\circ.\times C$ so that elements of the result can begin to be displayed *before any of the implied subexpressions have been fully evaluated.* Thus we come to the other extreme, that of evaluation on demand, or *delayed evaluation.* Such evaluation strategies have been discussed previously in the context of very high level languages. See, for example, [3,4].

In the above description of equivalent evaluation techniques we have not dealt with the issue of side effects. The equivalence is valid only as long as all operations return proper values. This unfortunately is not always so in APL, because of undefined forms such as `1÷0`, or `¯1*.5`. The traditional evaluation strategy would report an error in computing `2↑6 6 6÷2 1 0`, because of the division by 0. However, delayed evaluation would return `3 6`, since the division by 0 was never required, so it never occurred. This raises numerous issues which we will not discuss in this paper.

2. The Stylized Access Modes

One way to accomplish evaluation on demand is to regard each APL expression as an *object* capable of responding to certain questions. Some of the questions we may want to ask are

1) how many dimensions do you have?
2) what is your I-th dimension?
3) what is your $[I;J;\ldots;K]$-th element?

This brings us to an object oriented view of expressions analogous to that of SIMULA [2] or SMALLTALK [5] classes, ALPHARD [13] forms, or CLU [7] clusters. Naturally we can arrange that the ability to respond to the above messages is nicely obtained through recursion. Assuming that fully evaluated arrays (such as the atoms of an expression) can respond in the obvious way, our task is simply to associate with each APL operator procedures for responding to the above questions, given that the operator can ask these same questions of its operands. For example, in $2↑(A+B)$ the subexpression $(A+B)$ can respond to the request for an element by having "+" issue requests for the appropriate elements to A and B, and then use its "local expertise" to perform the addition.

In the above scheme we have essentially regarded each APL expression as a random access storage device. It is clear that keeping each subexpression in a state of readiness to provide an arbitrary element will involve very substantial overhead. Furthermore, this ability to access elements in random order is not frequently needed in the evaluation of APL expressions. Much more common is the situation in which we need all elements of an expression, one at a time, in the order in which they would occur if the expression had been evaluated and the corresponding array stored in row major form (*ravel* order). We will name this important way of accessing an expression *ravel mode.* In this mode we wish to regard an expression as a coroutine, which upon successive calls will deliver successive elements of the array it represents. By restricting ourselves to highly stylized access modes, such as ravel access, we have a much better prospect of an efficient implementation.

In order to understand what access modes are useful, we have to understand in detail how the various APL operators use the elements of their operands to produce the elements of the result. For example, for the compression operator / it will certainly be advantageous to have its argument be able to respond to the message "skip" as well as to the message "next". The argument may in fact generate the element being skipped and just throw it away, or it may be able to propagate the "skip" message further down the expression to attain a real saving in the computation. As another example, we may wish to break the message "next" of ravel mode into two distinct messages: "advance" and "fetch". The reason for this is that several evaluated atoms (e.g. in $A+B\times C$) may be able to share the same accessor (further explained later) and thus we can get by with a single "advance" message for all three atoms.

The perspective offered by the above discussion is that of associating with each node in the expression tree an *access mode*. The access modes are determined from the top down. An operator is told that the subtree it heads needs to be accessed in a certain way. Then by knowing how the elements of its result depend on the operand elements, it decides in which modes its arguments must be accessed. Thus access modes correspond to *inherited attributes*, in the sense of Knuth [6].

3. The Compilation of Streams

In this section we limit ourselves to APL expressions containing only scalar operators. As the reader may suspect, handling such expressions is relatively trivial. However, confinement to a domain where the task is well understood will allow us to focus our attention on setting the context for the following developments.

We will further limit the present discussion by disallowing as non-conformable scalar expressions where all atoms do not have identical shapes. We will deal with the very important special case of scalar atoms (which conform to any array according to the APL rules) in section 8. Consider how to evaluate $A+B\times C$. A clean way of obtaining the delayed evaluation effect is by implementing each scalar operator such as + or × as a reentrant coroutine. A different instance of the coroutine is used for each occurrence of the operator in the expression. Naturally all subexpressions (including the atoms) are accessed in ravel mode. Unfortunately, interpreting via reentrant coroutines is attractive only as long as the cost of a coroutine call and return is small compared to the processing performed between successive invocations of the coroutine. Assuming costs for machine operations such as are common today, then in $A+B\times C$ for example, each element of the result generated requires one addition, one mutiplication, and fourteen coroutine control transfer instructions. There are other hidden costs as well. Each instance of the atom accessing coroutine, if implemented in the obvious way, will be maintaining its own local copy of a counter and an offset into the atom array, when clearly these variables can be shared (and thus updated only once).

We have here the classical argument for compilation. Before we can discuss compilation in detail, however, we need to say a few more words about the machine model we have in mind. We assume a stack machine with all APL scalar operators as primitives. In addition, the execution environment contains certain data structures specifying how arrays are to be traversed, called *accessors*. The notion of an accessor was first introduced by Perlis in [10] (where it is called a *ladder*). A detailed discussion of these structures will be given in the next section. In the context of the current section an accessor can be thought of simply as the index of the array element we are currently accessing. Thus in the evaluation of $A+B\times C$ all three atoms can clearly share the same accessor. Our instruction repertoire will include the instructions *advance*(l), which advances accessor l to the next array position, and *fetch*(l,a), which pushes on the stack the element referenced by accessor l in atom a. It will become clear in the next section that the above two operations can be implemented with a few machine instructions on most computers.

Compilation is now straightforward. Assume that we have formed the expression tree during the lexical analysis of the expression. In a first pass, the *dimensions* pass, the conformability of the atoms is checked (and storage for the result can be allocated if we are executing an assignment, e.g. $Z\leftarrow A+B\times C$). Next, in the *push* pass, an accessor is created to be shared by all atoms, and initialized to point to the first element. In the *code generation* pass a traversal of the expression tree in endorder suffices to generate a codestream performing the scalar computations. For the example $Z\leftarrow A+B\times C$ the code would be:

fetch(l,B)
fetch(l,C)
multiply
fetch(l,A)
add
store(l,Z)
advance(l).

In the above l denotes the shared accessor of all atoms; the last instruction advances this accessor in preparation for the next iteration. Note that this code is correct irrespective of the dimensionality and size of the atoms (albeit not of their type). This information has been confined within l. Note also that we have obtained the effect of Abrams' *beating* optimization with no extra work. Finally the above code needs to be encapsulated by an appropriate loop, and we are ready to execute.

4. Operators that Break the Streaming

In the previous section we saw how simple it is to stream the evaluation of an expression composed solely of scalar operators. We now take a brief look at the other end of the spectrum, namely operators that cause any reasonable streaming mechanism to break down. Such operators include ⌽ (rotation), ⍋ , ⍒ , and arbitrary subscripting [ ]. The evaluation of these operators requires either a *random access* mode, or partial evaluation in temporary storage. There other cases where evaluation is necessary. For example the argument of (monadic) ⍳ and the left argument of / (compression) must be fully evaluated before even the conformability can be checked. Finally, subevaluations may be useful even when they are not necessary. Such storage time trade offs will be taken up in section 8. Thus the compiler generates a number of code streams corresponding to broken subexpressions. At run time these codestreams are invoked to replace a broken subexpression by (possibly portions of) an evaluated atom.

5. The Universal Selector

In this section we discuss compilation of expressions involving a subset of the selection operators of APL, as well as scalar operators. The selection operators we will handle are ↑ , ↓ , ⍉ , ⌽ (reversal), and [*E1*;*E2*;...;*EN*] (subscripting, where the *EI* are arithmetic progressions, i.e., expressions equivalent to $A+B\times\iota C$ for some integer scalars A, B, and C). We will name the above selection operators the *grid selectors*, for reasons that will become clear shortly. The grid selectors operate on an array argument (the right argument, except for subscripting), by extracting and/or renaming a portion of it. This is done according to a second argument, the *control argument* (one may consider monadic ⍉ and ⌽ to have default control arguments). The control argument must always be fully evaluated in order to check conformability with operations higher up in the tree. Thus it will be convenient to think of the control argument as being part of the selector, and not an object to which delayed evaluation is applicable.

We can think of the elements of an array A as occupying lattice points in a space of $\rho\rho A$ dimensions, within a bounding box of size ρA. Note that each of the grid selectors, when applied to A, results in an object that occupies a sublattice of the original lattice. In other words, moving across any coordinate of the result array can be viewed as moving along some set of coordinates of A, using equal size steps. Let us invent a generalized selector, called the *universal selector*, that can represent any such selection operation. *Then multiple selectors applied to the same array can be composed into a single instance of the universal selector.* Thus, if we start with an APL expression involving only scalar operators and grid selectors, we can think of a (null) universal selector starting at each atom and traversing the path to the root of the expression tree.

In this process the universal selector absorbs into itself any grid selectors it encounters. We will speak of the stage when this processing is done as the *push* pass. When this process is complete, our expression will have only scalar operators left. An instance of the universal selector will be associated with each leaf, indicating the composition of all selectors that must be applied to that evaluated array.

Although the composition of selectors can be most naturally thought of as happening from the bottom up, we will in fact carry out this process top down. The reasons for this are twofold. Firstly the cost of the push pass now becomes proportional to the size of the expression tree, as opposed to the path length of the tree. Secondly we will be able to leave with each node an instance of the universal selector which represents the composition of all selectors above that node. As we will see in section 8, this will provide us with essential information needed in storage/time trade off decisions at that node. The end result is quite similar to the normal form for select expressions first described by Abrams. However, the implementation of the transformations is much less cumbersome than Abrams' solution.

The data structure that represents a universal selector is called a *stepper*. A stepper U will be associated with every node of the expression tree. U will represent the state of the universal selector *before* the selector represented by the node (if any) has been absorbed. A node N incorporates itself into the stepper if it is a selector node, and passes this stepper on to its offspring. The newly formed stepper U is characterized by n, the rank of N's offspring, and four arrays q, s, d, and l, defined over the interval $[1,n]$. These arrays encode the way in which elements of the current node partake in the formation of the final result:

- $q[i]$ is an integer in $[1,r]$ and denotes which coordinate of the result array the i-th coordinate of the current node corresponds to; the set of all $q[i]$ with $q[i] = j$ is the set of coordinates of the current node which have been collapsed into the j-th coordinate of the result (recall that a transpose or subscripting may reduce the number of dimensions)
- $s[i]$ denotes the index, along the i-th coordinate, of the element of the current node which contributes to the first element of the result (e.g. the `[0;0;...;0]` element of the result in 0-origin)
- $d[i]$ indicates by how much to move along the i-th coordinate of the current node in order to arrive at the next element of the result along coordinate $q[i]$; (it can be negative)
- $l[i]$ indicates the size of the result along the $q[i]$-th coordinate; note that $l[i] = l[j]$ if $q[i] = q[j]$

Initially, for a null accessor at the root R we have $n = \rho\rho R$, $q[i] = i$, $s[i] = 0$, $d[i] = 1$, and $l[i] = \rho R[i]$, for all i. Let us now see how to incorporate various grid selectors using the stepper structure. When stepper U is about to absorb selector S, which in turn is applied to a node N, we will let p denote $\rho\rho N$, and the array r denote ρN. Primed quantities indicate the new values.

1. *Monadic Transpose.* A monadic transpose Θ can be absorbed in U by the following simple program:

```
n' ← n;
FOR i IN [1,n'] DO
    BEGIN
    q'[i] ← q[n+1-i];
    s'[i] ← s[n+1-i];
    d'[i] ← d[n+1-i];
    l'[i] ← l[n+1-i];
    END;
```

2. *Dyadic Transpose.* Let $c[i]$, i in $[1,p]$, denote the control argument. The following program shows how to absorb this transpose into U:

```
n' ← p;
FOR i IN [1,n'] DO
    BEGIN
    q'[i] ← q[c[i]];
    s'[i] ← s[c[i]];
    d'[i] ← d[c[i]];
    l'[i] ← l[c[i]];
    END;
```

3. *Take.* As above, let $c[i]$ denote the control argument. We have:

```
n' ← n;
FOR i IN [1,n'] DO
    BEGIN
    q'[i] ← q[i];
    s'[i] ← IF c[i] < 0
        THEN s[i] + d[i]*(r[i]+c[i])
        ELSE s[i];
    d'[i] ← d[i];
    l'[i] ← ABS(c[i]);
    END;
```

4. *Reversal.* Reversal along the k-th coordinate can be implemented as follows:

```
n' ← n;
FOR i IN [1,n'] DO
    BEGIN
    q'[i] ← q[i];
    s'[i] ← IF i = k THEN r[i]-s[i]+1 ELSE s[i];
    d'[i] ← IF i = k THEN -d[i] ELSE d[i];
    l'[i] ← l[i];
    END;
```

The above examples should be sufficient to illustrate how the stepper U can absorb the various grid selectors into itself.

Once all the steppers have stopped propagating by reaching the leaves of the tree, the grid selectors can be completely removed from the expression. We know from section 3 how to compile code for a tree of scalar operators, so the remaining issue is how to use the steppers to compile code for accessing the evaluated atoms of the expression. Note that each element of an atom is used at most once in computing some element of the final result. For each atom A there is an associated stepper U. We will use U to compute a new data structure, called an *accessor*, which will allow us to step through the elements of A in the proper order. A itself is assumed to be stored in ravel order. The accessor T obtained from U consists of π, the current position into the (stored representation of the) array A; α, the starting value of π; and two arrays $\gamma[i]$ and $\delta[i]$, defined for i in [1, max $q[j]$]. Intuitively, $\gamma[i]$ denotes the distance by which we have to increment π to obtain the next element of A needed for computing the next element of the result along the i-th dimension. The related quantity $\delta[i]$ denotes the distance by which π has to be incremented to attain the same goal as above, but now assuming that we have completely cycled through all dimensions higher than i in the result.

More formally, let the shape ρA be $[k_1,k_2,...,k_n]$ and define $h[i] = k_{i+1}k_{i+2}\cdots k_n$ ($h[n] = 1$). Then for $1 \leq i \leq \max q[j]$ we have

$$\gamma[i] = \sum_{q[j]=i} d[j]h[j], \text{ and}$$

$$\delta[i] = \gamma[i] - \sum_{i<j\leq n} \gamma[j]h[j], \text{ and}$$

$$\alpha = \sum_{1\leq j\leq n} s[j]h[j].$$

In order to understand the meaning of accessors, we now describe how they are used in the code compilation. Observe that, once the grid selectors have been removed from the expression tree, the shapes resulting from applying each universal selector to its atom must be identical. (This follows from the requirement on the conformability of scalar operator arguments - again disallowing scalar extension.) Let this common shape, which is also the shape of the expression result, be $[c_1,c_2,...,c_m]$. This shape is used to form a data structure global to the expression, called the *coordinate ladder*. The coordinate ladder is described by two arrays, *count*[i] and *limit*[i], for i in [0,m-1]. During execution, the array *count*[i] indicates the coordinates of the result element currently being produced. The array *limit*[i] is initialized by *limit*[i] ← c_i, i in [0,m-1], and is constant throughout execution. Also included is a global variable *coord*, indicating the coordinate currently being worked on. Using the coordinate ladder, an accessor T then implements the following operations:

init(T): $T.\pi \leftarrow T.\alpha$ (using the PASCAL notation for field extraction)

fetch(T,A): push on the stack the contents of [(base address of A) + $T.\pi$]; it may seem redundant to specify both T and A - we are anticipating the sharing of accessors discussed below

advance(T): $T.\pi \leftarrow T.\pi + T.\delta$[*coord*]

skip(T): $T.\pi \leftarrow T.\pi + T.\gamma$[*coord*]; this operation arises in the implementation of compression and will not be treated further in the current section

We are now ready to describe the compiled code for our expression. Let T_1, T_2, ..., T_s denote the list of accessors generated during the elimination of grid selectors. The compiled code has the form:

```
        coord ← -1;
        <Initializations>;
        init(T1);
        .....
        init(Ts);
Loop:   WHILE coord < last_coord DO
            BEGIN
            coord ← coord+1;
            count[coord] ← 0;
            END:
        <code for scalar operations,
         as described in section 3>;
        .....
        .....
Advance: advance(T1);
        .....
        advance(Ts);

        count[coord] ← count[coord] + 1;
        IF count[coord] < limit[coord]
            THEN GO TO Loop
        ELSE IF coord = 0
            THEN DONE
        ELSE
            BEGIN
            coord ← coord-1;
            GO TO Advance;
            END;
```

We will call the code following all the *Advance* instructions the *Universal Looper*.

There are two important optimizations we perform on the above code. They are called *Accessor Sharing* and *Coordinate Compression*; we deal with them in turn. By *Accessor Sharing* we refer to the fact that the same accessor can often be shared by several atoms in the expression. We can accomplish this sharing as follows. An atom A which is a descendant of some selector S in the tree will be called *visible* from S, if there is no other selector on the path from S to A. Add a dummy grid selector to the top of the expression tree. An accessor is generated not by each leaf, but rather *by each selector that has a non-empty set of atoms visible below it*. When a stepper reaches this selector, it can be used to generate an accessor that will be shared by the set of atoms in question.

The next optimization, *Coordinate Compression*, is important because it frequently happens that the applied grid selectors affect only a few of the coordinates of the atoms involved. Thus the ravel order in which these atoms are stored in memory corresponds to a large extent with the order in which they need to be accessed so as to produce the result. Specifically, if for some coordinate c it is true that all accessors generated have $\delta[c] = 0$ (and $\gamma[c]$ is not needed, i.e. there is no compression "/" along that coordinate), then coordinate c can be merged into coordinate c+1.

For practical APL expressions the above optimizations are very important. Consider $A+B\times C$, for example. All three leaves A, B, and C are visible from the dummy selector at the root, thus they can all share the same accessor T. Coordinate compression will then collapse the $\rho\rho A$ loops implicit in T and the coordinate ladder into just one huge loop that goes around $\times/\rho A$ times. This code is certainly the best we can hope to generate for the above expression. And in general, these optimizations allow us to get by with the smallest number of accessors and loops possible.

Note that the push pass must happen every time the shapes of the expression atoms change. However, the compiled code previously generated will still be valid. The same code can be reused with the newly generated accessors, as described above.

6. Reduction

Reduction has two novel aspects. Firstly it must generate its own looping code, which is not part of the universal looper. Secondly it has a number of nasty special cases, which will be briefly mentioned at the end of the section. What happens when a stepper goes through a reduction node in the expression tree? Assume the reduction is along the k-th dimension of offspring node N. The newly formed stepper will have another dimension added to it. The semantics of APL require that this new dimension be traversed in the reverse direction. An additional variable m, the depth of the coordinate ladder at the current point in the tree, must also be maintained. (In the previous section m was constant; it was always the rank of the final result). A stepper, in going through a reduction, in effect also ensures that the reduced coordinate k has become the last coordinate of the reduction's argument. Here is the reduction absorption code.

```
n' ← n+1; m' ← m+1
FOR i IN [1,k) DO
    BEGIN
    q'[i] ← q[i]; s'[i] ← s[i];
    d'[i] ← d[i]; l'[i] ← l[i];
    END;
COMMENT add a new coordinate - recall
    that the semantics of APL require that
    it be traversed in the reverse direction;
q'[k] ← m'; s'[k] ← ρN[k] - 1;
d'[k] ← -1; l'[k] ← ρN[k];
```

```
FOR i IN [k+1,n'] DO
    BEGIN
    q'[i] ← q[i-1]; s'[i] ← s[i-1];
    d'[i] ← d[i-1]; l'[i] ← l[i-1];
    END;
```

A reduction node must compile a loop that applies the appropriate binary operation between all elements along the reduced coordinate. The length of the reduced coordinate is saved in the expression frame. At run-time the compiled code pushes that length on the coordinate ladder and initiates a loop starting with the appropriate identity element and repeatedly fetches, advances, and operates on the next element of the argument subexpression. When the coordinate is exhausted the coordinate ladder is popped (i.e. *coord* is decremented), and the result returned. The details of these operations are straightforward and will not be described. Note that the coordinate ladder gets used as a stack. The compiled code always manipulates the global rung pointer *coord*, and thus is never aware of the dimension number of the coordinate being worked on. As a consequence, the compiled code remains valid for reexecution of the same expression, as long as the types of the expression atoms do not change. *The ranks and dimension vectors can change without invalidating the code.*

Figure 6.1 illustrates the various transformations described in this and the previous section. The reader is advised to study this example in detail. As a final note, boundary conditions for reduction give rise to many complications. Consider the expressions =/'' (↔ 1), =/'A' (↔ 'A'), =/'AA' (↔ 1), =/'AAA' (↔ 0). Due to lack of space we do not discuss techniques for handling these complications.

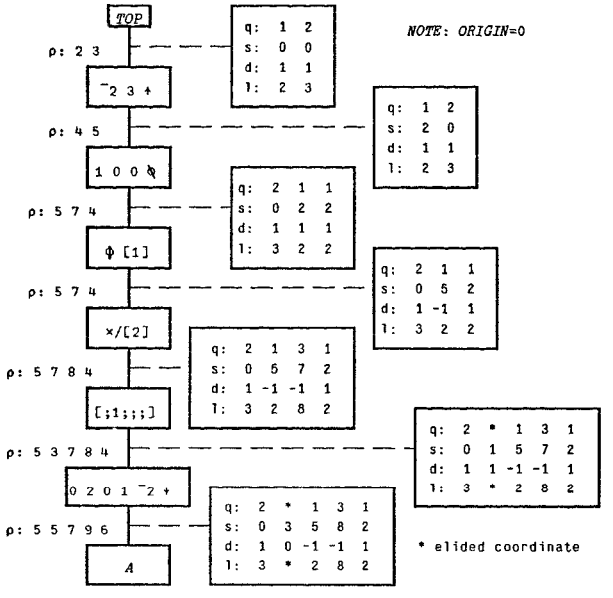

Fig. 6.1.
Propagation of steppers

7. A Specialized Reshape, with Application to Inner and Outer Product, and Scalar Extension

In this section we illustrate the power of the universal selector mechanism introduced in section 5. We show how this mechanism can handle a special case of dyadic reshape, which we will call *conforming reshape*. Using this as our tool we can then transform expressions containing inner or outer products into equivalent expressions containing only scalar operators, grid selectors, reductions, and conforming reshapes. These are expressions we already know how to compile. The same can be done with the scalar extension problem we have postponed until this section, that is the problem of scalar operators with one scalar and one non-scalar argument.

A dyadic reshape $A\rho B$ will be called conforming if ρB is a "suffix" of A. (Equivalently, $(\rho B) = (-\rho\rho B)\uparrow A$). Note that if B is scalar, this is always the case. Such a reshape preserves the structure of B; it only adds "dummy" copies along the new dimensions. A conforming reshape can be incorporated into a stepper by marking the coordinates introduced by the reshape as dummy (setting their d's to 0).

It turns out that by introducing appropriate conforming reshapes and transposes on the arguments, we can transform an outer product into a scalar operator, and an inner product into a scalar operator followed by a reduction. How this is done is in fact most succinctly expressed in APL itself.

Let $\oplus$ and $\otimes$ be any dyadic scalar operators; "↔" stands for "equivalent to".

Outer Product: $A\circ.\otimes B$

$$A\circ.\otimes B \leftrightarrow (((\rho\rho B)\phi\iota(\rho\rho B)+\rho\rho A)\lozenge((\rho B),\rho A)\rho A)\otimes((\rho A),\rho B)\rho B$$

Inner Product: $A\oplus.\otimes B$ (note: $^{-}1\uparrow\rho A = 1\uparrow\rho B$)

```
WA ← (1↑ρB),ρA
KA ← (ρρA)-1
VA ← ιρWA
ZA ← (KAϕ¯1↓VA),¯1↑VA
TA ← ZA⌽WAρA

WB ← (¯1↑ρA),ρB
KB ← ρρA
VB ← ιρWB
ZB ← ((KB≠VB)/VB),VB[KB]
TB ← ZB⌽WBρB

A⊕.⊗B ↔ ⊕/TA⊗TB
```

We have broken the inner product transformation up into a series of subexpressions for the sake of clarity. The reader can verify that each argument of the product is operated on by a conforming reshape and (possibly) a transpose. Note that if we were thinking of evaluating an APL expression in the straightforward way, the above transformations would be extremely expensive, as we are in effect creating may copies of the arguments of each (inner or outer) product. Since the above transformations show that these operators are redundant, one suspects that they were introduced into the language in order to provide efficient implementations of certain common operations. With our delayed evaluation strategy the multiple copies will of course never be generated and they introduce absolutely no overhead at run-time.

Scalar extension can be handled in an entirely analogous way. In the conformabilty pass scalar operators with one scalar and one non-scalar argument can make a note of this fact. Later, during the push pass, these operators can just in effect introduce a conforming reshape on to the scalar argument that will make it conform to the non-scalar one. (Using the same principle of "dummy expansions" we can easily implement more general kinds of conformability than APL allows).

8. Slicing

In this section we introduce a general technique for buffering portions of an array as its elements are computed, which we will call *slicing*. This technique is an integral part of our compilation with delayed evaluation strategy. Slicing gets used to store subexpressions whose value will be required many times, thus saving recomputation. It also gets used to moderate the effects of operators that break the streaming. The results of such operators are often not needed in their entirety, but only in certain "slices". An appropriate buffering scheme between the full expression and the subexpression headed by the breaking operator can then save space.

A *k-slice* of array (or *subexpression*) A is defined as $A[i_1;i_2;\ldots i_{n-k};;\ldots;;]$, where i_1, i_2, ...,i_{n-k} are valid indices for array A, with $n = \rho\rho A$. In other words, a k-slice is a k-dimensional array obtained from A by arbitrarily fixing a value for all but the last k coordinates, then letting these k coordinates vary through all their allowed values. We will call inner coordinates *higher*. Note that for each k, as we traverse A in ravel order, we will generate a complete set of k-slices of A. Our buffering scheme will work by always computing and saving a slice of appropriate size for a given subexpression.

There are numerous situations in evaluating APL expressions in which a subexpression of modest size should be saved in order to avoid wasted recomputation. Consider as examples $A+\iota 10000$, where A is a very complex scalar, or $A\circ.\times B$, where again A is complex and B is large. Note that we already have the tools to discover when these situations arise. In both of the above cases a conforming reshape was introduced during the processing of the expression. This conforming reshape leads to steppers with d's equal to 0 along certain coordinates (to be called the *dummy* coordinates), thus signalling the re-use of certain elements.

Such a conforming reshape indicates the need to save a slice of its *selected result*. By "selected" we mean that only that portion of the true slice need be generated which will eventually partake in the production of the final result. The slice size can be determined once the conforming reshape has been absorbed into the stepper. Let s be the coordinate just lower than the outermost dummy coordinate. (Take s = -1, if the outermost dummy coordinate is coordinate 0). Storage will be allocated for all non-dummy coordinates of the stepper which are higher that s. Coordinate s itself will be called the *slicing* coordinate.

The slice naturally acts as a buffer between the full expression and the subexpression below the conforming reshape. The code for the subexpression is placed in a separate codestream. The main and subexpression codestreams communicate data via the slice. Control is accomplished via a consuming accessor (in the main code) and a producing accessor (in the subexpression code). The consuming accessor is built from the stepper in the usual way, except that advancing along dimension s resets to the origin (and advancing along any dimension lower that s is a no-op). The stepper which the subexpression receives has all dummy coordinates removed. This modified stepper is then propagated down the subexpression in the usual way.

Finally the producing accessor is built from a trivial stepper for the subexpression's selected result, except again that advancing along dimension s resets to the origin.

How does control pass back and forth between the two codestreams? Let us first note that each codestream will be responsible for its own accessors. Yet we want all codestreams to share the global coordinate ladder, for obvious efficiency reasons. It turns out that the following simple policy solves the coordination problem. Every time a slice (producing or consuming) accessor is advanced, control passes to the partner codestream, if *coord* (the coordinate being advanced) is lower or equal to the slicing dimension. This elegant rule also subsumes initialization difficulties. At the beginning we set *coord* = -1 and start by advancing the main codestream along that dimension.

Of course slicing may recursively happen within the subexpression, and so on. In general there will be several separate codestreams, one for each piece of the entire expression that was introduced by slicing. (This may be smaller than the number of conforming reshapes in the expression, but this is a further optimization we do not discuss here.) The above coordination rule works in the general case as well. For instance, each scalar which is needed many times will be computed exactly once, no matter where it appears in the entire expression. This happens because the stepper for a scalar always consists entirely of dummy dimensions, and thus the scalar becomes available through a slice with slicing coordinate equal to -1. Therefore the scalar will be computed exactly once, namely when *coord* = -1 and the various accessors are advanced at the beginning of time.

The same idea can be used to save space when encountering operators that break the streaming. Such operators stop the propagation of a stepper S coming down from the root. However, rather than evaluating the entire subexpression, we can often proceed by only having the subexpression a slice at a time. Thus, for example, $\phi 3\phi MATRIX$ can easily be evaluated a row at a time, etc. The smallest required slice is a k-slice, with k the smallest integer such that all but the last k coordinates of the subexpression correspond to ravel order traversal.

This addition of "memory" to our delayed evaluation strategy is not entirely without cost at run-time. If the run-time bindings of the atoms are such that the slice is needed only once, then we are clearly doing unnecessary memory references. This, however, is a somewhat rare event, and furthermore tends to come into effect only when expressions are small, in which case we can afford the slowdown. The benefits of generality and overall efficiency for the compiled code seem well worth the price.

Figure 8.1 shows the run-time environment for the execution of the expression $T+(S1+S2)\times A$, with $S1$, $S2$ scalars. The subexpression $S1+S2$ has been sliced using a one-element buffer.

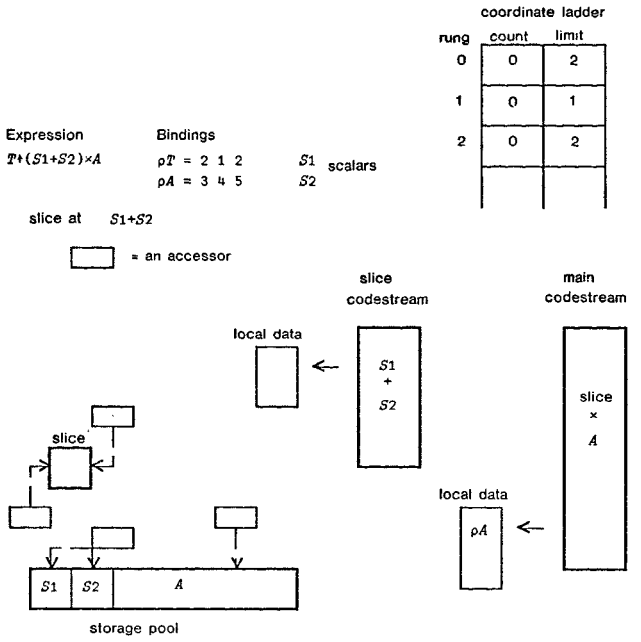

Fig. 8.1.
The Run-Time Environment

9. Conclusion

We have seen how to compile good code for a dynamic language. The generated code must be preceded by a preamble stating the assumptions for its validity. In our case these assumptions consist mostly of assertions about the expression's atom types. In ordinary APL usage, it is extremely unlikely that these assumptions will be violated during multiple executions of the expression. If that should happen, then the compiler must be re-invoked on the expression. Note that if our machine were able to interpret bytecodes relative to a type specification, even that step would not be necessary.

Acknowledgements: The authors would like to thank Alan J. Perlis, Ronald L. Rivest, Alan Kay, and Peter Deutsch for valuable comments on the paper.

10. References

[1] Philip Abrams, *"An APL Machine"*, SLAC Report #114, Stanford University, February 1970

[2] Birtwistle, Dahl, Myhrhaug, and Nygaard, SIMULA BEGIN, Auerbach, 1973

[3] A. P. Ershov, *"On the Essence of Compilation"*, Proceedings of IFIP Conference on Formal Description of Programming Concepts, August 1977, pp. 1.1-1.28

[4] Peter Henderson and James H. Morris, Jr., *"A Lazy Evaluator"*, Proceedings of the 3rd ACM Symposium on Principles of Programming Languages, January 1976, pp. 95-103

[5] Alan Kay et. al., SMALLTALK-72 INSTRUCTION MANUAL, Xerox PARC Technical Report, SSL 76-6, 1976

[6] Donald E. Knuth, *"Semantics of Context Free Languages"*, MAth. Sys. Th. 2, 127, 1968

[7] Liskov, Snyder, Atkinson, and Schaffert, *"Abstraction Mechanisms in* CLU", Proceedings of ACM Conference on Language Design for Reliable Software, March 1977, pp. 166-178

[8] W. M. McKeeman, *"An Approach to Computer Language Design"*, Ph.D. Dissertation, Stanford University, 1966

[9] Terry Miller, *"Compiling a Dynamic Language"*, Ph.D. Thesis, Yale University, 1977

[10] Alan J. Perlis, *"Steps Toward an APL Compiler - Updated"*, Research Report #24, Computer Science Department, Yale University, March 1975

[11] Jacob T. Schwartz, *"On Programming: An Interim Report on the* SETL *Project; Part I: Generalities"*, Computer Science Department, Courant Institute of New York University, February 1973

[12] Eric J. van Dyke, *"A Dynamic Incremental Compiler for an Interpretive Language"* Hewlett-Packard Journal, July 1977, pp. 17-23

[13] Wulf, London, and Shaw, *"Abstraction and Verification in Alphard: Introduction and Methodology"*, Carnegie-Mellon University and USC Information Sciences Institute Technical Report, 1976