THE DYNAMIC INCREMENTAL COMPILER OF APL\3000

Ronald L. Johnston
Computer Research Laboratory
Hewlett-Packard Company
3500 Deer Creek Road
Palo Alto, California, USA 94304
(415) 494-1444

ABSTRACT

Most APL implementations to date have been interpretive because of the dynamic nature of the language. APL\3000 employs a Dynamic Incremental Compiler to allow all the flexibility of change afforded by interpretation, but giving the added bonus of faster execution for programs run more than once. APL\3000 compiles code on a statement-by-statement basis as needed, saving the code and reusing it where possible. A statement is recompiled only when made necessary by changes in syntax or changes in variable bindings. The compiler produces optimized code by employing the Abrams techniques of Drag-along and Beating.

INTERPRETIVE PROCESSING

For years now, APL has been considered nearly impossible to compile because of its late binding of names. Since there are no static declarations of name attributes, name bindings can be determined only at run-time, at which point a name takes on the attributes of whatever is assigned to it. Further, names can have different meanings in different contexts, and can even have different meanings on different executions of the very same statement. The language allows so much to change dynamically that any given compilation of a statement could prove inapplicable to a subsequent invocation of the statement.

This dynamic, potentially unstable nature has prompted most APL implementations to choose an interpretive approach. While straightforward, it carries with it some inherent performance limitations. There is a relatively high overhead for interpretation when compared to executing compiled code because an interpreter performs syntax analysis and operand-conformability and domain checking on every execution of a statement. Often the interpretive overhead is more costly than the execution itself, particularly when the statement deals with operands of few elements.

NAIVE PROCESSING

Most APL systems have also been handicapped by combining interpretation with "naive" operation, in which each subexpression is evaluated immediately after its operands have been evaluated. For instance, the statement

```
A←5↑B×C*D
```

where B, C, and D are numeric vectors of length 100, would be evaluated naively as shown by the following stylized code:

```
for i←1 until 100 do  { t1←C*D }
  t1[i] ← C[i] * D[i];
for i←1 until 100 do  { t2←B×t1 }
  t2[i] ← B[i] × t1[i];
for i←1 until 5 do    { t3←5↑t2 }
  t3[i] ← t2[i];
A ← t3;
```

This approach is noticeably wasteful, pointing out out one major source of inefficiency in a naive APL system: its rigidly literal execution of the statement, one function at a time, forces unnecessary array-shaped temporary results (and their associated accessing loops) to be produced in order to generate the end result. In this particular example, only the first 5 elements of the results of B×C*D are actually needed, a naive system fully evaluates all of its elements (100) and then takes the first 5.

APL\3000: AN INCREMENTAL COMPILER

APL\3000 is a complete APL system that runs on the HP 3000 minicomputer. It is not an interpreter, nor is it "naive", being instead a dynamic incremental compiler which uses the techniques of Drag-along and Beating to produce optimized code. These techniques will be more fully described later in the paper. The flexible, interactive nature of the language has not been changed to accommodate compilation; there have been no restrictions placed on the language. APL\3000 appears to be a "standard" APL interpreter, with few hints that it is really a compiler. Calculator mode input is immediately compiled and executed, as are Quad-input and the argument to the Execute function. User-defined functions are dynamically compiled statement-by-statement. The code produced is then saved for subsequent invocations.

By not compiling a statement until its execution is demanded, it is possible to overcome APL's lack of name declarations. Code can be generated that matches the specific run-time attributes of each name. This technique has the beneficial consequence of also leaving some statements uncompiled: those that have never been executed.

The idea of dynamically compiling code for APL is not entirely new. It has been employed by at least one interpretive APL system as a technique for faster processing of some primitive functions. However, its application was in a naive setting, and the object code was simply discarded after its single execution because it could not be guaranteed valid later.

SIGNATURE CODE AND BINDING ERRORS

Recognizing the potential variability of name bindings, the APL\3000 compiler hedges against any attribute changes that would invalidate the compiled code. In addition to the working code, it produces a preamble block of "SIGNATURE CODE" which specifies and checks the assumptions bound into the working code. Though signature and working code will be discussed as if they were physically separate entities, this is really only a conceptual distinction. They are actually just two components of a single code block, with the block's signature always preceding the block's working code as below.

```
+-------------+-----------------------+
| Signature   |        Working        |
|   Code      |         Code          |
+-------------+-----------------------+
```

On re-execution of a statement's code block, the signature code is executed first to test the validity of the working code which follows. If the signature assertions are satisfied, then the working code is executed with no further interpretive overhead. However, if the signature code finds that the working code is no longer valid, the code "breaks", causing a "BINDING ERROR." The dynamic compiler is then automatically invoked to produce code for the new situation. Figure 1 illustrates this process.

At the heart of this strategy is the assumption that most APL statements are "well-behaved" - that is, they do not really exercise the dynamic nature of the language. If this is the case, then the code that is generated will remain valid over several, perhaps all, executions of a statement. However, if this proves not to be the case, it is certainly not desirable to be continually recompiling a given statement;

```
                      +-------------+
                      | Compile     |
New                   | "HARD"      |     +---------+     +-------+
Expression ---------->| Signature   |---->| Execute |---->| Save  |-->
                      | and Working |     | Working |     | Code  |
                      | Code        |     | Code    |     | Block |
                      +-------------+     +---------+     +-------+

                      +-----------+       +---------+
Previously            | Test      |       | Execute |
Compiled   ---------->| Signature |------>| Working |-->
Expression            | Code      |       | Code    |
                      +-----------+       +---------+
                            |
                            |
                       Code Breaks
                            |
                            |
                            v
                      +-------------+                     +----------+
                      | Compile     |     +---------+     | Replace  |
                      | "SOFT"      |     | Execute |     | Old Code |
                      | Signature   |---->| Working |---->| With     |-->
                      | and Working |     | Code    |     | New      |
                      | Code        |     +---------+     | Version  |
                      +-------------+                     +----------+
```

FIGURE 1: Execution of an expression

the cost of compiling is too high, and this would reduce APL\3000 to the equivalent of an interpreter. To avoid this, the dynamic compiler takes the defensive measure of changing the type of code it emits for that statement.

HARD AND SOFT CODE

Signature code is actually a collection of assertions about the identifiers involved in an expression. These assertions specify, for each identifier, the attributes which were current at compile time and which must remain unchanged for the working code to be valid. Two general types of identifier bindings are made in compiling expressions; these are reflected by two types of working code and hence two types of signature code: "Hard" and "Soft."

HARD code is the type of code initially emitted for an expression. Here, to gain the fastest possible execution, certain attributes of identifiers and their data descriptors are bound at compile-time as constants which are not expected to change: 1) rank, 2) rho, 3) machine representation, and 4) data storage layout. This prevents the expression's working code from having to make run-time computations of such things as subscripting polynomials, loop induction steps and limits, subscript range limits, and so on. If re-execution of this code fails later because an identifier changed attributes, then that identifier is assumed to be unstable (in the current expression, anyway) and the second type of code - soft code - is generated by the incremental compiler for any of the expression's computations involving that identifier.

SOFT code is less specifically tailored to the current attributes of the offending identifier. Rank determines the maximum number of nested loops needed to access a variable's data, so it continues to be bound into the code. Representation dictates the type of machine instructions needed to deal with the data, so it too is bound into the code. The soft signature for the identifier will assert that both 1) rank and 2) machine representation are unchanged since the code was compiled. All other attributes of the identifier are allowed to change from one invocation of the code to the next without causing the code to break. Soft code is less efficient than hard code because it requires more housekeeping computation and more indirection through the Attribute Table, but the lost efficiency rarely approaches the cost of repeated recompilation.

DRAG-ALONG AND BEATING

APL\3000 produces non-naive code by employing two optimization strategies conceived by Philip Abrams in 1970. In his doctoral thesis [1], he described two processes, "Drag-along" and "Beating", which might be used to dramatically improve APL performance for large array operands.

DRAG-ALONG is the process of deferring operations on array expressions as long as possible. As more global context is recognized, it can often lead to simplification or optimization of the original expression. The most common optimization results from detecting a sequence of operations that can share the same evaluation loops, reducing both the number of loops and the number of array-shaped temporaries created.

BEATING substitutes data-descriptor manipulation for brute force data copying in many selection functions - a selection function being one which rearranges or selects data without changing its values. Abrams elaborated the concept of using data descriptors to indicate how a block of linear storage is to be accessed in order to exhibit a particular rank, shape, and ordering. He further showed that if the data descriptors were separated from the storage they apply to, then the functions Take, Drop, Reverse, Transpose, and Subscripting by a scalar or Arithmetic Progression Vector (APV, also called "J-vectors") could be implemented with no data movement. By applying a set of transformations to the original data's descriptor in these cases, a new descriptor can be calculated which properly indicates the selected data, and which shares the data block with its original owner.

APL\3000 implements this DESCRIPTOR CALCULUS, attaching a reference count to each data block. In this way, operations on one variable (which must not alter others by side effect) can detect sharing and acquire a private copy of the data block if necessary. As an illustration, if A is the 5-element numeric vector 1.1 2.2 3.3 4.4 5.5, and B is assigned the Reversal of A, they would share A's data block:

```
 REF  A[1]  A[2]  A[3]  A[4]  A[5]
+-----+-----+-----+-----+-----+-----+
|  2  | 1.1 | 2.2 | 3.3 | 4.4 | 5.5 |
+-----+-----+-----+-----+-----+-----+
COUNT B[5]  B[4]  B[3]  B[2]  B[1]
```

Beating may interact with drag-along by restricting the scope of a deferred operation to produce only the pertinent elements of the result. Applying these strategies to the example given earlier,

```
A←5↑B×C*D
```

APL\3000 would generate the following (stylized) code:

```
for i←1 until 5 do    { t1←5↑B×C*D }
  t1[i] ← B[i] × C[i] * D[i];
A ← t1;
```

Note that Drag-along enabled the naive approach's 3 loops with 3 temporaries to be merged into 1 loop with 1 temporary, and Beating allowed the loop to be limited to 5 elements rather than 100.

THE COMPILATION PROCESS

Compilation is a 3-step process which flows roughly as follows:

```
SOURCE --> TREE --> FOLIATED TREE --> E-CODE
       (1)      (2)               (3)
```

The first step is to perform a SYNTAX ANALYSIS of the statement, scanning the tokenized source, referring to the Attribute Table to find the current name bindings. The result of this step is an ordered set of expression trees. The statement may need to be split into more than one expression tree in order to avoid side effects which would give unpredictable results. Function calls, assignment statements, and shared-variable accesses all have the ability to change the current attributes of identifiers. The compiler must isolate any of these from the processing of the rest of the statement in order to generate code which cannot invalidate itself. Figure 2 shows the expression tree for a statement which requires only 1 tree.

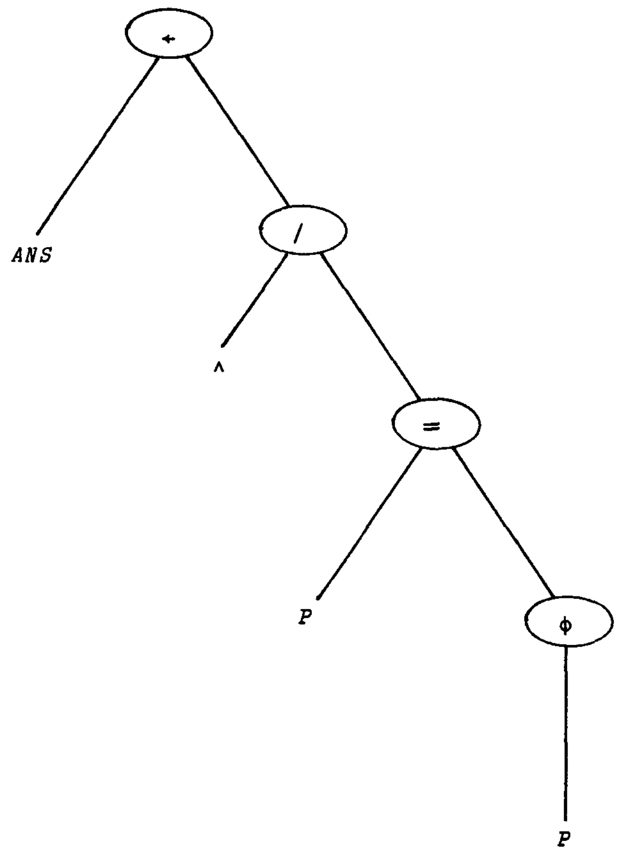

FIGURE 2: Expression Tree for *ANS* ← ∧/*P*=⌽*P* (*P* a variable; value immaterial)

With the syntax of the statement determined, the next compilation step is called FOLIATION. This is the context-gathering process of tree traversal during which the drag-along and beating strategies are applied. The result is a much fuller tree in which each node has attached to it information describing the shape, representation, and the accessing methods of its result.

Two types of auxiliary description nodes are used to represent this information. One, the "RRR" node, describes the general structure of an item: 1) Rank (number of dimensions), 2) Rho (size of each dimension), and 3) Representation (machine data type). The other, the "DELOFF" node, indicates the data-access information for an item: 1) DEL (steps for each coordinate), and 2) OFFset (location of its logical first element).

During foliation, the expression tree is traversed leaf-to-root, with all subtrees of a node being visited before the node itself. This guarantees that a full description of each of the node's operands is available to the compiler as it foliates that node. If the node being foliated is a leaf (constant or identifier), the required descriptive information is immediately available from either the Attribute Table or the Constant Block. As an identifier node is foliated, its signature code is placed into the object code block to indicate the attributes which are being bound into its RRR and DELOFF nodes. A given identifier has a signature emitted for it only once during the tree traversal, regardless of how many times it appears in the tree.

As the tree traversal continues, the leaf descriptions are pushed upwards towards the root of the tree, each function node causing the descriptions they inherit to be changed according to their defined effect on their operands. Note that the foliation process is not concerned with actual values of variables, simply with their structure and accessing information.

If the node to be foliated is a function for which beating can be performed, its RRR and DELOFF nodes are derived by modifying those of its operands according to the subscript calculus defined by Abrams.

Foliation continues until either the root of the tree has been reached or a function node has been reached for which there is no operand value-independent way of predicting the structure of its result. In either case the foliated tree (or subtree) is ready for the next compilation step, "Code Generation." Figure 3 shows Figure 2's tree after foliation.

The CODE-GENERATION process is performed by traversing the fully-foliated tree again, this time working from the root of the tree towards its leaves. By utilizing the

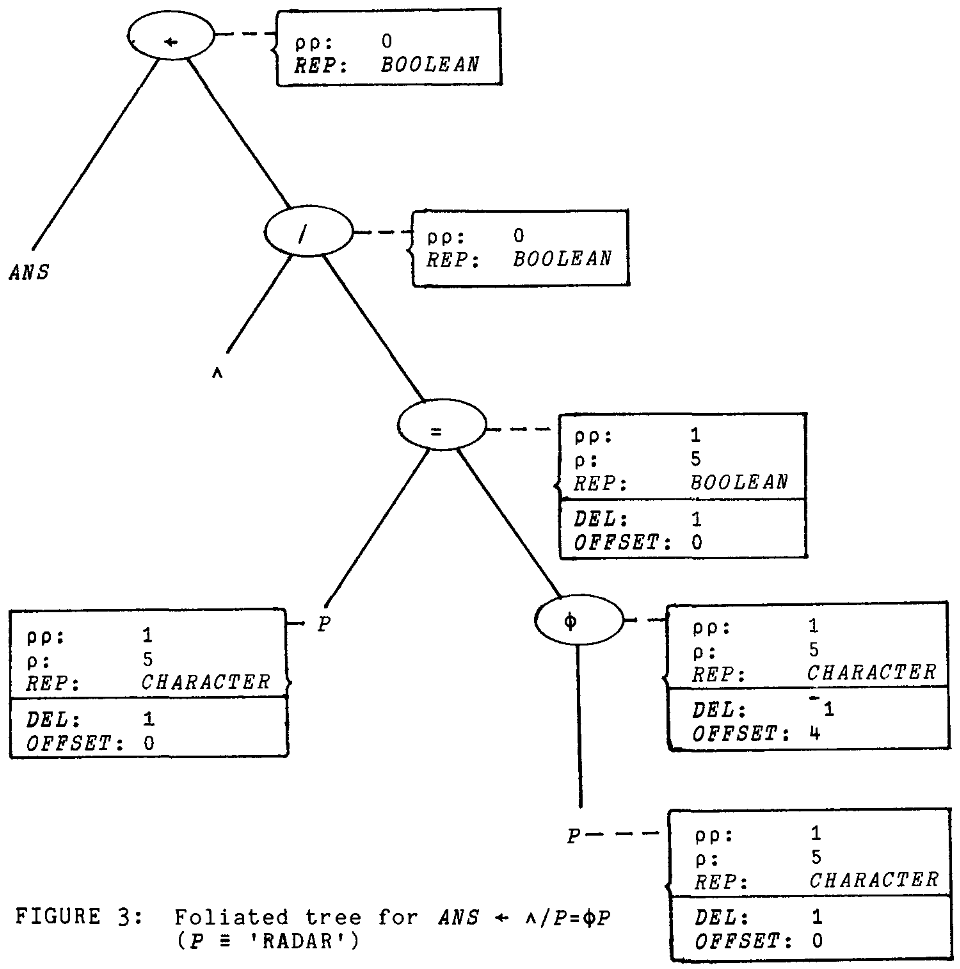

FIGURE 3: Foliated tree for *ANS* ← ∧/*P*=⌽*P* (*P* ≡ 'RADAR')

context information that has been attached to the tree in the form of RRR and DELOFF nodes, non-naive code is generated. This means that all the optimizations described earlier are applied: loop merging, reduction of temporaries, evaluation of only the required results, and producing the results of selection functions by descriptor calculus alone.

Functions which will require loops to be generated in order to step through non-scalar operands are examined to see if their loops can be shared with those of other functions. If they have identical DELOFF descriptions, then their loops can be merged into one set. Multi-dimensional operands could potentially require several nested loops - one for each dimension. However, it is often the case that such loops can be collapsed into fewer loops because the operand's data access is very regular (row-major order, for instance).

After the looping structures have been decided upon, the code within the loops is generated by traversing the tree. Starting at the tree's root and working towards its leaves, each function node is visited and its corresponding machine-instruction sequence is emitted into the Code Block. This traversal continues until all nodes of the expression tree have been visited and their corresponding instructions emitted. Every instruction which has the possibility of failing (e.g., divide could fail by dividing by 0), has associated with it a source pointer by which the source token in error might be identified. When the code-generation process has been completed, the resultant code block is passed to the E-machine for execution. Figure 4 shows the Hard code generated from the foliated tree of Figure 3.

EXECUTION: THE E-MACHINE

Our original intent was that the compiler produce HP 3000 code. However, it soon became clear that the machine architecture made this an impractical approach. The HP 3000 is a "pure-code" machine which strictly enforces the distinction between "code" and "data". It does not allow one to generate code (which is data to the compiler) and immediately view the result as executable code. Instead, it is necessary to invoke the operating system's Linker/Loader in order to put the code into a form the

```
{ SIGNATURE CODE }
   ASSERT: (1=ρρP) ∧ (5=ρP) ∧ (CHAR=REP(P))
            (0=OFFSET(P)) ∧ (1=DEL(P));
   ASSERT: WRITEABLE(ANS);

{ WORKING CODE }
   spad1 ← 0;   { initialize forward register to OFFSET(P) }
   spad2 ← 4;   { initialize reverse register to OFFSET( P) }
   spad3 ← 5;   { initialize loop-limit register to ϕP }
   temp  ← 1;
   WHILE spad1 ≠ spad3 DO
      BEGIN
      temp ← temp ∧ P[spad1] = P[spad2];
      IF temp = 0 THEN GOTO finish;    { early-out }
      spad1 +← 1;
      spad2 +← -1;
      END;
   finish:  ANS ← temp;
```

FIGURE 4: Code Block for *ANS ← ∧/P=ϕP*
(Hard code, P←'RADAR')

machine will execute. This is too slow a process to be practical for the dynamic compiler, which seeks to provide immediate response to an execution request, whether it be for first execution or recompilation of a function's statement, or for processing of calculator mode, the execute function, or Quad input.

The compiler's target machine is, instead, a hypothetical "E-machine" which is simulated on the HP 3000 by a combination of firmware and software. This machine has a fairly traditional scalar-oriented architecture which has, for every APL primitive function, a set of corresponding machine instructions. It has 256 Scratch-Pad (SPAD) Registers, which are used mainly for loop controls, counters, and indexing of variables. All computation is done on its Stack, which is also used to hold intermediate scalar results. The machine shares the Attribute Table with the compiler itself, calling upon it to give the current name bindings as needed by either Hard or Soft code. A 32-bit address space is supported, though the HP 3000 itself is a 16-bit minicomputer. The larger address space is provided by a paged virtual-memory scheme which has microcoded support. This allows APL\3000 to handle very large workspaces - the practical limit being the amount of on-line disk storage available.

When the E-machine is invoked, it is given the address of a code block to execute, and an initial Program Counter offset from the beginning of that block. This makes it possible to bypass the signature code when checking it is unnecessary - as is the case when the code has just been compiled and the signature is guaranteed correct. The E-machine executes until the code block either terminates normally or causes an error which prevents its completion. The E-machine indicates its success or failure and passes control back to the process which invoked it. If it terminates on an error, the type and source location of the error are also passed back so that the appropriate error-handling mechanism may be invoked.

REFERENCES

[1] Abrams, Philip s., "An APL Machine", PhD dissertation, SLAC Report No 114, February 1970

[2] Hewlett-Packard Journal, July 1977