Using Logic Programming for Compiling APL

Howard Derby

Computer Science Department
California Institute of Technology

5134:TR:84

Using Logic Programming for Compiling APL

5134:TR:84

Thesis by
Howard Derby

In Partial Fulfilment of the Requirements for the Degree of
Master of Science

March 1984

Abstract

APL is a dynamically typed language which deals with arrays whose type, number of dimensions (rank) and size are not fixed at compile time, but are instead determined at run time. This makes APL more difficult to compile than static languages like Pascal or FORTRAN. This thesis describes a prototype implementation of the core of an APL compiler. The intention thus far has been to demonstrate techniques for dealing with some of the issues that arise when trying to implement APL efficiently, rather than to produce a working implementation. The present program does not do any of the initial lexical processing required, and only compiles into intermediate code. Object code is never produced. The prototype has many APL features missing and is undoubtedly full of bugs.

APL[Iver62] is an old language but there are still no efficient implementations. "Efficient" in this case means that an APL implementation of a particular algorithm is roughly as fast as an implementation of the same algorithm in another language. Other languages are generally more efficient than APL because compilers are available for them, whereas all production APL implementations are interpreters. APL's array primitives, which operate on arrays of arbitrary type and rank, are what make it difficult to compile. Compiling APL by threading together code for the individual primitives gives poor results because general implementations of the primitives contain code for dealing with all the rank and type combinations that can possibly occur, which makes them inefficient. Generating good code requires having information about which ranks and types the programmer uses, so code is produced for only those cases that actually occur. To do this, the compiler needs to know the types and ranks of arrays at compile time. The method used to get this information determines how the compiler will appear to the APL programmer. If no restrictions are placed on the language, it will be impossible in most cases to get the information that is needed, and inefficient code will result. A compiler that required declarations for all types and ranks would be very restrictive on the programmer but would be able to produce efficient code.

Reasonable approaches lie between these two extremes. The compiler can get some of the information it needs by exploiting the relationships between an APL primitive's arguments and its results. For example, it is known that $\rho \phi X$ is ρX. The process of using these relationships is called type inference. Effective use of type inference requires having fairly complete information about the state at each point in the program, because missing information propagates through an expression just as known information does. This makes it essential that inference be done not just within a statement but between statements as well. Doing inference between separate functions APL is also very helpful. My work differs from other compiler efforts [Weid83,Budd83] in that it stresses complete interprocedural inference in an attempt to generate efficient code in spite of complete lack of declarations.

APL is a highly interactive, interpretive language, and this frustrates attempts to do type inference between statements. Users can interrupt and edit functions at run time. Language features like $\underline{\phi}$, which executes an arbitrary character vector as an APL expression, can change any variable, ruining any dataflow analysis the compiler does to support type inference between variables. In all but the simplest cases it would be impossible to determine at compile time which, if any, variables were modified. Not only would the executed expression have to be interpreted, but in many cases much of the code that followed would have to be fully general, making it inefficient. As a result, the presence of a single execute can negate most

of the benefits of compiling. APL also includes a large number of so–called "system functions", which refer functions and variables using character strings. These can also ruin the compiler's dataflow analysis. In addition, ⎕FX and ⎕EX can change a function's parse tree. Consider the statement A←F–X. This can parse as either A←F(–X) or as A←(F)–(X), depending on whether F is a monadic function or a variable. Since system functions can change a name from a function to a variable or vise versa, the execution of system functions can change the parse trees of defined functions. This means the use of system functions in a function can have even more extreme effects than the presence of ⍎.

This can be handled in a variety of ways. The approach taken in this compiler is to remove ⍎ and system functions from the language. Fortunately, many programs don't use these features. Eliminating them makes APL essentially static, but the most useful of the dynamic properties of APL, the variable array types and ranks, are retained. APL programs that are heavily compute bound, such as signal processing applications, would benefit immensely from a compiler of this sort. APL could be successfully applied to many new areas if a working compiler of this type existed.

To make it possible for the compiler to extract the information it needs without any declarations, several other restrictions have been imposed. The compiler works on an entire workspace at once, allowing it to see the "big picture". Execution always starts in the same function, which takes no arguments, and has no result. No global variables exist when execution starts. Input (⎕ ⍞) must be done in such a way that the result always has the same rank and type. The program must be such that ranks and types can be determined at compile time. For example, a function that works recursively over the rank of an array cannot be compiled.

Recursion over ranks

```
∇Z←F X
[1] Z←X
[2] →(0=ρρX)/0
[3] Z←F +/X
∇
```

There are also a large number of restrictions that are not inherent in the design but are caused by omissions from the compiler. Most of the scalar functions are missing, and ⍋⍒⌸⌽⍉? are all not implemented. No input or output facilities

have been provided, because input raises system questions and APL output is complicated. Indexed assignment is missing, but could easily be added. None of these omissions represent fundamental problems but would simply require additional programming effort to implement them. A problem that is more fundamental is the inability of the compiled code to correctly handle integer overflow. The code generated by the present version will trap on integer overflow. In addition, the compiler doesn't do any of the lexical preprocessing required and only generates unoptimized intermediate code.

The compiler is based on logic programming. It uses an axiomatic description of the APL primitives for both type inference and code generation. The primitives are defined using a Prolog–like language that includes rewrite rules for defining functions. The remainder of the compiler in written in Prolog.

PROLOG

A Prolog [Rou75, Clock81, Per78] program is a set of logical assertions about abstract objects called terms. A term is either a variable, a constant, or a function applied to one or more terms. Constants are either numbers, like 4, or atoms, like foo. Variables are alphanumeric strings which must start with a capital letter, which distinguishes them from constants, which must start with a lower case letter. Example terms are:

```
fred sue 4656 Variable f(X,why) f(g(w(X)))
```

Functions may also be represented by prefix, postfix, or infix operators instead of the usual mathematical notation. Functions to be used in such a way must be declared to the parser. Examples of infix usage:

```
34+17 Z is 4*Y 3-2>4
```

The term "Z is 4*Y" is equivalent to "is(Z,*(4,Y))". Some common arithmetic operations are already declared with the standard precedences.

The functions about which logical assertions are made are called predicates. The assertions are organized as a list of Horn clauses. A typical clause is

```
f(X) :- g(X) , h(X).
```

This means f(X) is true if g(X) is true and h(X) is true. The infix operator :- has the meaning "if", and the infix comma has the meaning "and". Thus to prove f(4), it would suffice to prove both g(4) and h(4). The variable X represents any term. Variables are local to the clause in which they appear. A clause may also be a simple assertion without any antecedents, ie

```
g1(X,X).
```

The period at the end is necessary to indicate the end of the clause.

It is important to realize that none of the functions have any semantics associated with them; they represent symbolic, not computational connectives.

This Prolog program implements part of Peano arithmetic:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

le(X,X).
le(X,s(Y)) :- le(X,Y).
```

The predicate add holds if the sum of the first two arguments equals the third. The first clause asserts the common identity 0+X=X. The second clause says that the sum of the successor of X, and Y is the the successor of the sum of X and Y. Note that the successor function s is not defined anywhere. It has no operational behavior. The successor of 0 is not 1, but merely s(0). The results computed by add are in terms of successor functions. The le predicate is true is its first argument is less than or equal to its second.

A Prolog program is "run" by giving Prolog a theorem to prove. The theorem to be proved is in the form of a list of terms to be proved, in order. Each term in the list is called a goal. The goals are proved by searching the list of clauses for one whose head unifies with the goal, and then proving its antecedents, treating them as subgoals. This process is called resolution[Rob65]. Two terms are said to unify if there exists a set of variable substitutions that will make the two terms

identical. For example, f(X,4) and f(ref,Y) unify with X=ref in the first term and Y=4 in the second. Unification is the operation humans intuitively perform when doing algebraic pattern matching.

If a subgoal is encountered which will not unify with the head of any clause, it cannot be proven and is said to fail. If this occurs, Prolog backtracks in an attempt to find a proof that does not depend on the failed subgoal. It rejects the proof of the previous subgoal and attempts to re–prove it by resolving with a different clause. If there are no untried clauses remaining, it too is failed and the backtracking continues. If no alternative paths can be found, then the whole process fails. If Prolog can prove all of a goal's subgoals, then the goal is said to succeed.

There are two other Prolog features that need to be described. One is the ";" connective. A semicolon can be used in a subgoal list to indicate "or" just as comma is used to indicate "and". Thus

```
f(X) :- a(X) ; b(X).
```

is equivalent to

```
f(X) :- a(X).
f(X) :- b(X).
```

Prolog also has a predicate called cut. It always succeeds but has the unusual effect of preventing backtracking. If backtracking reaches a cut, the entire goal is failed. The not predicate succeeds if its argument goal fails and visa versa. It may be defined using cut by

```
not(P) :- P ,!, fail.
not(P).
```

not works by first attempting to solve the goal P. If P succeeds, then a cut is performed and then failed, which causes not(P) to fail without trying further ways of proving P. If P fails, then the second clause for not succeeds.

REWRITE RULES

While Prolog contains good facilities for defining predicates, it has no way of assigning semantics to functions. This can be done by adding rewrite rules. A rewrite rule of the form X := Y states that a term of the form X may be replaced by Y. For example, addition may now be defined by

```
0+X := X.
s(X)+Y := s(X+Y).
```

Any term in which a + operator appears will be rewritten as the sum, if the arguments are numbers in successor notation. A term like s(0)+foo will be rewritten, first as s(0+foo) and then as s(foo). There are some rather messy issues regarding the precise semantics of rewrite rules, which in turn relate to how rewrites are implemented. In particular, there are questions about which terms may be rewritten and when during execution they may be rewritten. If the desired semantics of rewrite rules is that of multi-valued functions, then care must be taken to assure that no term is rewritten using two different rules. This does not apply to different instances of the same term. The exact semantics the compiler uses will be discussed later.

Rewrite rules may be used to define a programming language, such as APL. APL data types are represented by terms, and functions are defined that access the properties of those data types, much like defining abstract data types in object oriented languages. APL's data type is an array of elements, which may be either characters or numbers. The main properties of an APL array are its rank, shape, and the elements it contains. For example, the constant vector 1 2 3 may be represented in Prolog as vec([1,2,3]). (In Prolog, [a,b,c] is a three long list containing a, b, and c.) As another example, consider the term iota(N). It represents the vector returned by ιN, but does not compute its elements. Functions are needed to access its properties. The rank function returns the rank of any term that represents an APL array. Since ιN returns a vector, rank iota(N) = 1, and the rule

```
rank iota(N) := 1
```

is used to specify that fact. Likewise, a method for accessing the elements of an array must be provided. The function ⌾ represents subscripting with a list of integer indices. Since iota(N) has rank one, it requires a list of length one as a subscript. iota(N) has i in the i'th place, so the rule

```
iota(N)⌾[I] := I
```

defines the elements of ιN. Defining the shape is more difficult. The APL function ρ returns the shape of an array as a vector. It is easiest to define ρ directly on each term that represents an array. For example, the shape of ιN is given by the rule

```
(rho iota(N))⌾[1] := N.
```

The collection of rules of this form for each array defining term defines the element properties of ρ, but it does not specify its rank or shape, which must be given explicitly by

```
rank rho(X) := 1.
(rho rho X)⌾[1] := rank X.
```

Note that (rho rho rho X)⌾[1] = rank(rho X) = 1, and rank rho rho X = 1, ie $\rho\rho$X always returns a vector of length 1.

It is important to keep in mind that three very distinct types of objects are being used. One type is the class of symbolic objects that have no significance to the APL programmer and are simply intermediate data structures in the compiler's implementation. The other two are APL arrays and APL array elements. The distinction between these two is important because they often appear to have the same form. This difference is imposed by the target machine, which can manipulate numbers but knows nothing about the properties of the APL scalars. The indexing operation X⌾I returns the value of an APL array X at position I not as an APL scalar but as a number, which is suitable for manipulation by the arithmetic instructions. This differs from the APL indexing function, which returns APL arrays. The scalar constant 1 as used by the APL programmer can be represented by scalar(1). It the usual array properties, like rank scalar(1)=0. It is not a number, so it cannot be used in arithmetic operations– add(scalar(1),scalar(2)) is illegal. Scalars must

be indexed with an empty list of subscripts (since they have rank 0) to get their numeric value. Thus scalar(1)@[]=1 , and add(scalar(1)@[],scalar(2)@[])=3. But just as scalar(1) isn't a number, 1 isn't a scalar: rank(1) is undefined, as is 1@[].

These techniques can be applied with equal success to more complex APL functions. The monadic scalar functions are readily defined using these techniques. Because all of the monadic scalar functions have the same dataflow, they are first grouped by representing all applications of them by msc(F,X), where F is the function to be applied (+−×÷ etc) and X is the argument array. mscop(F,X) is defined for each monadic scalar function F, and it returns F applied to the number X. The rules

```
rank msc(F,X) := rank X.
rho msc(F,X) := rho X.
msc(F,X)@I := mscop(F,X@I).
```

define the element by element application of F to the array X. The result array is defined to have the same rank and shape as the argument, and each element of the result is the function F applied to the corresponding element of the argument.

APL operators can be defined similarly. Outer products o.F are first translated to outer(X,F,Y), where X and Y are the left and right arguments. The defining rules are

```
outer(X,F,Y) := outer(X,rank X,F,Y,rank Y).
rank outer(X,RX,F,Y,RY) := RX+RY.
(rho outer(X,RX,F,Y,RY))@[I] := (rho X)@[I] :- I=<RX.
(rho outer(X,RX,F,Y,RY))@[I] := (rho Y)@[I-RX].
outer(X,RX,F,Y,RY)@I := dscop(F,X@J,Y@K) :- split(I,RX,J,K).
```

The function dscop(F,X,Y) is defined analogously to mscop; it applies the dyadic scalar function F to the values (not arrays) X and Y. The predicate split(I,S,L,R) splits the list S into a left part L of length I and the remainder R. Formally, append(L,R,S) and length(L,I). This corresponds to outer product's effect of concatenating the axes of its left and right arguments. All of the APL (non–system) functions and operators can be defined in this way, although most not as easily as monadic scalar functions or outer product. All of this is done in terms of scalar equations, so enough information exists to execute each APL operation on a scalar machine.

There is no reason when creating a term for any rewrites to be done immediately. This feature makes it easy for a rewrite system to use lazy evaluation, only rewriting those terms that affect the progress of the computation. In practical terms, it means that it is possible to compute, say, the rank of an array without computing it's elements. It is also possible to do algebra on the definitions. For example, the definitions of monadic – and ι can be combined. The rules

```
iota(N)@[I] := I.
msc(F,X)@I := mscop(F,X@I).
```

can be combined to form the net rewrite

```
msc(F,iota(N))@[I] := mscop(F,I).
```

For F = –, we have that the i'th element of $-\iota$N is –i. The ability of the rules to combine in this way allows "beating and dragging" to occur. The formation of all of the intermediate arrays is not needed. There are cases when it is actually desirable to create an intermediate array for reasons of efficiency. This occurs in operations like compression or grade up where the data flow is highly variable. In these cases the operation can be defined in terms of an intermediate array. For example compression uses an vector which contains the index of all the ones in the left argument. Creation of intermediate arrays may also be beneficial when each array element is referenced many times, such as in an inner or outer product. At present, this possibility is ignored. Any array can be explicitly instantiated by assigning it to an APL variable in the source program.

COMPILING CODE FROM REWRITE RULES

Compiling requires generating intermediate code based on the rewrite description of the APL primitives. A goal can be compiled by a process very similar to interpreting it. Subgoals that can be put off until run time generate intermediate code. Those which cannot be done at run time are executed interpretively. The rewrite system is augmented to include declarations of predicates and functions that can be done at run time. The intermediate code operations are defined by these rules. Also included in the rules are interpretive methods of execution, so that the intermediate code can be executed at compile time. For example

```
code cmpe(X,Y) :- X=:=Y.
```

declares cmpe to be an integer equality predicate that can be executed at run time. The predicate =:= compares two Prolog integers, failing if they differ. The statement

```
code iadd(X,Y) := Z :- Z is X+Y.
```

defines iadd as a run–time integer add. Consider the program

```
code print(X) :- fail.  time
code iadd(X,Y) := Z :- Z is X+Y.
code cmpe(X,Y) :- X=:=Y.

inc(X) := iadd(X,1).
count(I,N) :- cmpe(I,N).
count(I,N) :- print(I) , count(inc(I),N).
```

The predicate count(I,N) prints the integers from I to N. The following is an oversimplified description of the compilation process, but it retains the essential features. Suppose we compile the goal count(1,100). The process starts by attempting to resolve count. It finds two possibilities, and attempts to generate code for both. The first possibility has one subgoal, cmpe(I,N), which is emitted as code. The second alternative first does print(I), which is simply emitted. It then recurses. Starting on the expressions, it must rewrite inc(I) since it cannot be done at run time. It is rewritten as iadd(I,1), which can be. A temporary is needed to hold the results of run time functions, so the code T<–iadd(I,1) is generated, with the subgoal rewritten as count(T,N). Since this is an example of tail recursion, a loop is generated.

The actual compilation process is much less elegant than the idealistic one described above. First, to simplify the analytic burden, run time alternatives and tail recursion are spelled out explicitly rather then leaving them for the compiler to discover. The operator ":" is used to express alternatives that may be executed at run time. It is very similar to the Prolog ";". Recursive run–time code is never needed in the description of APL primitives, so no attempt was made to represent loops recursively. Instead, the arcane syntax

```
F <: K :> G >> L
```

is used to describe a loop. F is executed first, then the condition K is checked. If it succeeds, then the loop terminates. The "condition" K may contain additional code after the test to do final calculations. If K fails, then G is executed. L is a tail recursive–looking call to the predicate that contains this construct. It specifies the correspondence of variable values when looping. The loop primitive can be defined formally by

```
F <: K :> G >> L :- F , (K ; G , L).
```

Second, something besides raw Prolog variables are need to represent run–time variables. A term of the form temp(N,Type,Value,Code) is used to represent them. This corresponds to a register or a memory location. If an explicit assignment is made, as in a goal like

```
Q<–iadd(W,5)
```

a temporary will automatically be allocated. This means that in addition to emitting the iadd operation, Q will be bound to a term representing a run–time variable. When temporaries are assigned in more than one place, such as in a conditional, it is necessary to pre–allocate the temporary. This necessitates declaring those variables that will be used in that way. This is done by prefixing the subgoals with a list of variables and the word "by".

```
dscop(=,X,Y) := Z :- [Z] by X=:=Y , Z<–1 : Z<–0.
```

This statement is part of the compiler. Note how ugly it is. The "[Z] by" part declares that Z is to be set conditionally. The ":" acts just like Prolog's ";" except that it generates a run–time conditional. The "<–" 's are just assignments. The other messy thing is that when values are passed to loops to initialize counters and such, they must be passed not as constants but as temporaries. This means that the simple counting program above would have to look like

```
inc(X) := iadd(X,1).
count(I,N) :- <:cmpe(I,N):> print(I) , T<–inc(I) >> count(T,N).
```

and be called with

```
count(copy(1),100).
```

The copy function simply copies its argument to a temporary. It is defined with

```
copy(X) := Y :- Y<-X.
```

The third issue that has been ignored in the rewrite system is backtracking. The run-time rules are not allowed to backtrack. Alternatives on predicate failure are restricted to those explicitly provided by the alternative and looping constructs. All unification must be done at compile time; comparison of run-time values must be done using comparison predicates. This is somewhat confusing because the intermediate code expresses alternatives in terms of failure and backtracking. This does not mean that backtracking can occur at runtime. Consider a piece of intermediate code(which looks much like the rewrite system code)

```
T<-iadd(4,X) , cmpe(T,W) , Z<-1 : Z<-0
```

In Prolog, the value of T would be unbound if the cmpe were to fail. This is not true in the intermediate code. Intermediate code rules leave T undefined if the cmpe fails. Once a predicate succeeds, meaning all rewrites have been completed and all subgoals have been solved, it cannot backtrack. It is as if a Prolog cut had been placed at the end of each clause. No further alternatives for either subgoals or for that predicate or function will be tried. New alternatives will have to be created by the parent goal. This has the effect of making each procedure closed- after it returns, no further activations are possible, so it's frame may be discarded.

It is now appropriate to discuss the precise rewrite semantics used by the compiler. The rewrite rules are hashed by function. If a function has no rewrite rules, then it is never given an opportunity to be rewritten and is treated as an ordinary term. If there are rewrite rules present, then an attempt is made to rewrite each use of that function when it is first encountered. If it cannot be rewritten, it is left alone. Once this is done, no further rewrites may take place. First encountered means when the term is needed. If it appears in a predicate, it is needed just before execution of the predicate takes place. The rewrite system assembles subterms for the predicate by rewriting all the arguments. If rewritable terms appear in the arguments to a rewritable function, then they themselves are first rewritten. If a rewritable term appears in the replacement field of a rewrite rule, it is first rewritten before the substitution is made. Rewrites do not create fail points; once a rewrite

has been accepted it cannot be backtracked. If a rewrite should fail, meaning one of its subgoals could not be solved, then the next matching rewrite is tried. If no rewrite succeeds, then by default the term is not rewritten. Any intermediate code generated by the rewrite process is inserted into the code stream in the order in which the rewrites are encountered.

COMPILING APL

In order to generate good code for APL operations, the compiler must know the ranks and types of the operands. Type in APL is not only whether an array contains characters or numbers, but also the way in which a numeric array is represented internally. APL represents numbers in three different internal forms: bit–packed arrays for boolean (0,1) arrays, integers for arrays comprised solely of integers, and floating point arrays when needed. However, aside from accuracy considerations this is never the programmer's concern. Rules defining the type properties of APL operations are put into the the rewrite system much like rules defining the other properties of APL. Unfortunately, this presents somewhat of a problem because operations on integers tend to produce integers but generate floating point results when they overflow. As a result, the type rules are only approximate. Many solutions to this problem are available, but none seem very good. The approach taken here, which is clearly inconsistent with the definition of APL, is to generate an error on integer overflow and require that the programmer float numbers himself if floating point is required. This does not affect the automatic conversion when integers are added to floating point, but only inhibits conversion on overflow. This behavior is similar to that of static languages.

Interpreters spend much of their time checking type and rank cases, and if these are known at compile time, great efficiency improvements are possible[Weid83]. Operations like transpose can disappear completely, and the formation of many intermediates becomes unnecessary. Unfortunately, it is impossible in all but the simplest cases to get type and rank information for an isolated function. Some information about how it is to be used is needed. There are several ways to get this information. This compiler uses global interprocedural analysis, without declarations of any sort. This means that an entire workspace must be compiled at one time. It assumes that execution begins in the same niladic function each time. There are no variables when execution starts, so all globals must be set up by APL code. The compiler assumes that all ranks and types can be determined at compile time. To aid in this process, the compiler will create a separate instance of an APL

statement for each type and rank case that can be shown to exist at compile time. In (hopefully) most cases, it will be possible to exhaust the cases which occur and to determine the control flow conditions that govern them. If this can be done, then a new APL program will have been produced in which the types and ranks occuring within each APL statement are constant. The compiler cannot handle a program which cannot be put in this form.

Type and rank information is accumulated by the first phase of the compiler. To simplify the analysis, the compiler assumes that the input has been transformed into an equivalent program where each statement is either a simple assignment statement which assigns only one variable, a branch to an expression, a function call where all arguments are variables and which assigns the result to a variable, or a null statement which is the last statement in the function and simply indicates return. The compiler follows the possible control flow paths from start to finish. Execution begins in the main function, and initially there are no variables. Thus all rank and type information is known. The compiler then examines the first statement. It is one of the four basic statement types: assignment, branch, call, or return. If it is an assignment, then it has the form

Var ← Expression

and the type and rank of var can be determined by evaluating type(Expression) and rank(Expression). If Expression contains no variables, then its value is constant and the type and rank are clearly calculable. If they do contain variables, (which cannot happen in the first statement but may occur later) then the compiler will attempt to determine what it needs to know about the variables by looking backward along the control paths that lead to the statement being processed. This process is called interstatement inference. In many cases all that is required is the type and rank of the variables, which will be known so no further work is required. If the expression is such that the type and rank of the result do not depend solely on the type and ranks of the variables it contains, then the compiler makes an attempt to determine the values of these parameters. For example, consider the following program:

```
∇FOO
[1] X←⍳4
[2] Y←X⍴3
∇
```

When statement [2] is processed, the rank of Y will be equal to the shape of X,

which is 4. So when processing rank(Xρ3), the compiler will derive rank(Xρ3) = rho(X)⌷[1]. The rewrite system has a rule for evaluating shapes of APL variables. This rule couples the expression handling rewrite system to the interstatement inference code. The compiler will discover that X was last specified by the first statement, and that rho(X)⌷[1] = rho(ι4)⌷[1]. This new expression is passed back to the rewrite system, which calculates that rho(ι4)⌷[1] = 4. Thus the rank of Y is 4.

We have now described the basic workings of interstatement inference. Not all APL control flow is as simple as that of the example, and the proper allowances must be made. The simplest complication is that of branching. The APL $\rightarrow$ operation is a computed goto, which specifies the statement number where execution will continue. The target of a goto may be any statement, or a $\rightarrow$0 which terminates function execution and causes a return. Because of the adverse affects of considering control paths that are never taken, the compiler assumes that all branches are done using labels, and that the target of a branch is either the next statement ($\rightarrow\iota$0) or a statement whose label appears in the branch. The major restriction that this imposes is that 0 cannot be used as a target to stop function execution. The code that branches generate is just a computed goto, with checking for $\rightarrow\iota$0.

The compiler makes no attempt to determine which of the branch choices will be taken. It considers all of the possible targets as potential next statements and generates instances for those paths. This could result in an exponential explosion in statement instances, since each instance of a branch is considered separately. To help combat this growth, the compiler attempts to merge instances. Two instances can be merged if all successors of the instance merge and the assumptions made about the states of variables before the execution of each instance are the same. In other words two instances can be merged if they and all their successors do the same work. To aid in the merging process, the compiler stores the set of constraints on variable values that must be satisfied if the code generated for each instance is to work properly. Merger is checked by verifying each constraint in the environment at the proposed merge point. If two instances are merged and the compiler later discovers that they must be left separate, they are split.

Unmergeable Instances

```
∇F
[1] →X/L
[2] Y←⌊/Y
[3] L:Z←Y÷4
∇
```

In the above example, statement [3] can be reached by two control paths; directly from statement [2] or by branch from statement [1] Unfortunately, the rank of Y will be different if the branch in statement [1] is taken than if it isn't. As a result, the compiler will generate two instances for statement [3], one for each path.

Instance Merging

```
∇FOO
[1] →X/L
[2] Y←Y+1
[3] L:Z←Y÷4
∇
```

This example differs from the last in that statement [2] doesn't change the rank of Y. As a result, Y will have the same rank in statement [3] independent of the path taken. Statement merging takes place and only one instance is produced. There are cases where the same efficient code can handle arrays of various ranks. For example, the code for dividing by a constant (as in F[3]) doesn't depend on the rank of the array. Matching ranks are sufficient but not necessary conditions for statement merging. Unfortunately, the compiler doesn't know what the true conditions are.

Loops in APL are implemented using branches. The same code that handles instance merging also handles loops. Consider the set of instances that would be formed if the control flow paths in a function were followed for an arbitrary number of steps. This could result in an infinite set of instances, but only if the types and ranks kept changing. If closure occurs and a finite set is produced, then because of instance merging the compiler will stop generating new instances once it has completed the set. If closure does not occur, the compiler will get stuck and infinite loop. In practice, this can only occur if the rank of an array is being increased in a loop. This is fairly unusual, and represents a known limitation that a programmer

would have to avoid. In general, a compiler should recognize closure failures in loops and give up, or better yet use more general code to handle it.

Closure Failure

```
∇FOO
[1] L:X←X,[.5]X
[2] →(0<I←I-1)/L
∇
```

This function increases the rank of X by one for each iteration. As a result, the compiler will be unable to find a sufficient set of instances for statement [2]. Each time the compiler reaches statement [2], the compiler will look for an appropriate instance of statement [1]. Since the rank of X has increased, none of the previously generated instances will be satisfactory, and the compiler will generate a new instance of statement [1]. This will require an instance of statement [2], which the compiler will attempt to merge with the existing instances. The catch is that each instance of statement [2] can lead to an instance of statement [1], and the constraints on merging an instance of statement [2] include satisfying the constraints of all nodes that follow it. This cannot be satisfied because X has increased rank, so a new instance of statement [2] is also needed. This is an infinite loop, from which the compiler will not escape until it runs out of memory.

Calls to defined functions present more of a challenge. The present compiler handles each function call as a separate statement. This isn't a restriction because an APL statement containing a function call can be broken down into an equivalent set of statements such that this holds. The compilation of user functions is governed by function use, not function definition. Because functions can be called with different arguments and with different states of the global variables they use, functions create instances much the same as statements do. When an new instance of a function call is needed, the compiler looks for an existing function instance that is compatible with the arguments and the global variables at that point. If no such function instance can be found, a new function instance is created by creating a new instance for the first statement of the function and letting things progress from there. If interstatement inference shows that types and ranks depend upon values passed from outside, which can be either arguments or global variables, the compiler requests the needed information from the caller. It also stores the information obtained from outside as a constraint on the use of this function instance, just as for statement instances. This allows the new function instance to be used by other callers.

Since the control flow through a function is not guaranteed to be the same each time it is executed, there are cases where the state after a function call depends on control flow within the function in such a way that separate statement instances are needed to handle the different cases after return. Because of this, a function must have not one return but several, depending on what control path its execution followed. The statement following a function call has one instance for each different return path the function can take. In most cases these instances will merge since like most APL functions, user functions tend to produce results whose types and ranks are not control dependent.

Different Return States

```
∇Z←F X
[1] Z←X
[2] →K/L
[3] Z←⌊/X
∇

∇Z←G X
[1] T←F X
[2] Z←⍟T
∇
```

In this example, the function F can return its argument unchanged or with rank reduced by one, depending on the value of K. As a result, the function G will need two different instances of G[2], one for each case. Which instance is used will depend on which return F takes.

Recursive functions complicate this because they refer to incomplete function instances whose input and output properties are unknown. By taking care to update the data structures in the proper order no problems occur unless closure of function instances fails. This is analogous to failure of loop closure that can occur in ordinary function processing. The key to successful treatment of recursive functions is to not consider return paths until it is known that they can be reached. If when processing a function a recursive call is encountered, the compiler may choose to implement the call using an incomplete function instance (most likely the one that is currently being built). The only return possibilities that need be considered immediately are those that have already been shown to exist. The return conditions for all those

paths are known since the paths have been worked out. The catch is that if a new return path is discovered, then all of the callers will need to generate a new return case. Consider the following function:

```
∇Z←FACT N;T
[1] Z←1
[2] →(N=0)/END
[3] T←FACT N-1
[4] Z←N×T
[5] END:
∇
```

This function recursively defines factorial. When the compiler processes this function, it will first process statement 1. The branch in statement [2] has two possible targets, and it will try the sequential possibility first. That brings it to statement [3], where it encounters a recursive call. Checking the constraints for the existing instance of FACT, (which is the one it is working on now) it finds that the only constraint is that the argument be an integer scalar. That holds if N is an integer scalar, which is the precondition for this instance of FACT, so this instance can in fact be used. So statement [3] can be implemented with a recursive call to the instance being created. Statement [4] comes next, and an instance is needed for each return possibility– but there are none. So the compiler assumes for the time being that FACT cannot return and continues evaluating branch alternatives. The compiler has avoided needing the properties of the result of FACT, which is crucial since they are not yet available. Statement [2] can also execute a branch to END, which is statement [5], so an instance of statement [5] is created for this flow path. After statement [5] the function returns, so the compiler has now discovered a return path. It must now go back and patch up all calls to FACT to account for this possibility. The only call was in statement [3], so this new return possibility causes an instance of statement [4] to be created for return from statement [3]. This path has T an integer scalar, so statement [4] is compiled for those conditions. Next comes statement [5], which gets merged with the existing instance of statement [5], and the processing of FACT is complete.

Given the set of statement instances and the flow between them, it is easy for the second phase of the compiler to generate intermediate code for each APL statement using the techniques described for compiling code from rewrite rules. Assignment statements are implemented by compiling a predicate which creates an instance of a variable on the run–time heap. It does this by working through the elements of the array in row major order, storing each as it is computed.

Indexed assignment has not been implemented but is a simple extension of normal assignment. Gotos evaluate the shape of the target expression, and compute its first element if the target is non–empty. Calls and returns are compiled in the obvious way. Code generation produces intermediate code only; the machine code generator was never implemented.

The storage manager was never implemented, but a brief description of its intended function is in order. APL variables are always fully instantiated in memory. All but scalars are stored on a heap. Garbage collection is not needed since it is always clear what values are active. Compaction is necessary to prevent memory fragmentation. New entries are simply allocated off the top of the heap. Each heap entry is a vector of elements, which is pointed to by the variable that references it. Array elements are stored in row major order. The shape of an array is also stored on the heap as a vector of integers. Every APL variable is associated with a two word storage area. If it is a scalar, the first word holds its value and the second is unused. If it is not, the first word points to the shape and the second points to the data. Each heap entry has a back pointer so that if the garbage compactor has to move an entry the forward pointer can be changed. It also has a word which contains the data length so the storage compactor knows how much data to move. Thus there is a two word overhead for each heap entry, giving a total of 6 words of overhead for each non–scalar APL array. Because this storage management scheme has never been tried, it is not clear how efficient it would be.

CONCLUSIONS

Because no working system was produced, it would be presumptuous to assert that these techniques work. However, I find the results encouraging. First, the use of rewrite rules to describe the APL primitives works very well. They provide a single description of APL for both inference and code generation purposes, avoiding having two separate descriptions of APL. In addition, the rules look much like a specification for the operations, even though they are suitable for computational purposes. This allows fewer opportunities for error and helps to produce bug–free descriptions of the primitives. The use of Prolog for doing the type inference code was a mixed blessing. Many things were more difficult to do or do right in Prolog than in other languages, and some things that would be required in a production implementation are simply impossible. On the other hand, the use of Prolog is what made writing this prototype possible at all. The amount of time spent coding the prototype was about 4 weeks in 1982 and early 1983. One of the reasons I

believe that APL compilers are just starting to appear is the massive effort required to program them in conventional languages. The prototype was developed using Edinburg Prolog running under TOPS–20[Per78]. I found Edinburg Prolog pleasant to use and fairly efficient. To my knowledge it is still the Prolog of choice.

The intermediate code generated by the compiler is very poor. This was expected, but the code is worse than I had imagined. It contains a large number of unnecessary move instructions, a lot of dead code and unpropagated constants, multiplications for subscript calculation that need to be removed by strength reduction, and a lot of generally inefficient ways of doing things. Even so, it captures the basic dataflow characteristics of the APL operations much better than simply stringing together implementations of the primitives would. Optimization and machine code generation is left as a textbook exercise in compiler construction. Given a good optimizer, the compiler would generate code that I believe would compete favorably with the code produced by current FORTRAN compilers in many cases.

Even though it would impose many restrictions on the APL programmer, I believe that an APL compiler could be produced using these principles that would place the efficiency of the language near that of more conventional languages, such as FORTRAN, and still retain APL's great advantage in ease of programming and programmer productivity. The speed improvement over interpreted APL would depend not only on the quality of the interpreter but also on the nature of the programs being compiled. The compiler could be compatible with an interpreter so that programs could be developed interactively and then compiled without much additional effort.

REFERENCES

[Budd83] Budd, Timothy A., "An APL Compiler for the UNIX Timesharing System" APL83 Conference Proceedings, April, 1983

[Clock81] Clocksin, W. F., and Mellish, C. S. "Programming in Prolog" Springer-Verlag: Berlin, New York, 1981

[Iver62] Iverson, Kenneth E., "A Programming Language" John Wiley and Sons, inc.: New York, London, Sidney, 1962

[Per78] Pereira, Luis Moniz; Pereira, Fernando C. N.; and Warren, David H. D. "USER'S GUIDE to DECsystem-10 PROLOG"

[Rob65] Robinson, J. A., "A Machine-Oriented Logic Based on the Resolution Principle" JACM vol. 12, pp. 23-29, 1965

[Rou75] Roussel, P. "Prolog Manuel ele Reference et d'Utilisation" Groupe d'Intelligence Artificielle, Marseille-Luminy, 1975

[Weid83] Weidmann, Clark, "A Performance Comparason Between an APL Interpeter and Compiler" APL83 Conference Proceedings, April, 1983