NASA Contractor Report 178196

ICASE REPORT NO. 86-67

ICASE

PS: A NONPROCEDURAL LANGUAGE WITH DATA TYPES AND MODULES

Maya B. Gokhale

Contract No. NAS1-18107

October 1986

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

PS: A NONPROCEDURAL LANGUAGE WITH DATA TYPES AND MODULES

Maya B. Gokhale
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716

ABSTRACT

The Problem Specification (PS) nonprocedural language is a very high level language for algorithm specification. PS is suitable for nonprogrammers, who can specify a problem using mathematically-oriented equations; for expert programmers, who can prototype different versions of a software system for evaluation; and for those who wish to use specifications for portions (if not all) of a program. PS has data types and modules similar to Modula-2. The compiler generates C code.

In this paper, we first show PS by example, and then discuss efficiency issues in scheduling and code generation.

This research was supported in part by the National Aeronautics and Space Administration under NASA Contract Number NAS1-18107 while the author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665, and in part by UDRF Grant LTR860114.

PS: A Nonprocedural Language with Data Types and Modules

Maya B. Gokhale*

Keywords and Concepts: very high level language, equational specification, automatic program generation

1 Introduction

In this paper we introduce a very high level language for algorithm specification. In the Problem Specification (PS) language, the user expresses the algorithm as a set of equations. The PS compiler analyses the specification to determine an execution ordering, and generates a procedural program in the C language (under Berkeley Unix).

PS is intended for a wide user community, ranging from domain expert to expert programmer. For the domain expert, PS offers a mathematically oriented language in which to express the problem. For example, notation used to describe algorithms in numerical analysis can be transcribed with very slight syntactic modification directly into PS. In this paper, we show a PS module to do Gaussian Elimination and compare the PS equations with the algorithm description in a standard numerical analysis text.

At the other extreme, PS is a useful tool for the expert programmer in that it facilitates rapid prototyping. "Exploratory programming" is a technique recognized as essential to gaining an understanding of a new application area. In hardware this is referred to as breadboarding. PS is useful in breadboarding alternate approaches to a problem. Experience may be gained from using different versions, each generated from a different specification in PS. Use of procedural languages to actually code alternate approaches would be prohibitive in time. Use of AI languages would necessitate a complete rethinking of the problem for the production version.

Since data structures in PS are almost identical to Pascal or Modula data structures, it is easier to shift from the PS specification to an equivalent production program than from Lisp or Prolog.

In the continuum between nonprogrammer and expert, PS can be used to generate component modules of a system. We recognize that one language cannot solve all problems for all users. Certain classes of problems can be solved concisely and easily in PS. Others, which perhaps depend on idiosyncracies of the operating environment, are better expressed in other languages. For example, a stream of characters can be converted to a stream of tokens by the *Lex* tool (or even *scanf*). Each token can then be used by a PS module in a computation.

The novel features of PS in comparison to similar nonprocedural languages [2, 4, 7-10, 12] are as follows:

- The language is strongly typed. Declarations follow most syntactic conventions of Pascal or Modula. The compiler enforces type checking and reports inconsistencies or incompleteness in type usage.
- The language is modular. We feel this feature is essential to the stepwise refinement of a problem solution. The basic program unit is the module, which is semantically equivalent to a side-effect-free function.
- Our system is compatible with the external environment. It is easy to compose a program of PS modules intermixed with modules in a procedural language. In our current implementation, the PS compiler generates a C function from each module description. A module may invoke other modules written in PS *or* C. Conversely, a C function may invoke a PS module.

In this paper we hope to impart a feel for PS through example. The next section gives examples ranging from simple program fragments to complete PS modules. Finally we discuss the problem of efficiency, especially as related to storage.

2 PS by Example

A PS program consists of a sequence of module descriptions. Within the module, a programmer can describe the structure of the data items and the relationship among data items. The PS compiler then analyzes the data dependencies; synthesizes a *schedule*; an ordering to the generation of data

```
type I = 1 .. n;
var A: array [I] of int; { declare a variable }

define A[I] = I; { define a value for each element A }
```

Figure 1: A Simple Example

items; and produces a C program, complete with type and variable declarations and control structure. Since the PS language is nonprocedural, it contains no control constructs such as "goto" or "for" or "while". There is not even the implicit control from one statement to the next in sequence. This implies that a PS module is unordered. The lexical sequence of equations is unrelated to computation sequence of the generated program. Since the programmer does not control the sequence of program execution, the language is, of necessity, single assignment. The value of a data item may be defined exactly once. For this reason, we refer to PS as a dataflow language. In a PS equation of the form *left hand side* = *expression*, the *left hand side* can be considered the name of an arc (value) on a dataflow graph and the *expression* is a node (computation) producing the value.

The examples below show how aggregate data items can be given values without control structures such as loops or recursion; how recurrences are defined; and how to create structured data types.

2.1 Index Sets

As in Modula-2 or Pascal, the type statement describes a data structure which is then used in variable declarations and in equations. Figure 1 illustrates a simple use of a subrange type. Each element of A is given a value by the "define" statement because the type identifier I is used to index the array. Use of a subrange type identifier as a subscript denotes universal quantification over the subrange. Thus, the subrange type declaration is used to establish an *index set*. Use of the index set to subscript an array indicates that the equation is true for each element of A indexed by I. This can be contrasted to the use of iteration control constructs to define multiple occurrences of an action in procedural languages. The equivalent procedural code reads

```
type I,J = 1 .. n;
var A,B : array [I, J] of real;

define A[I,J] = B[J, I];
```

Figure 2: Transpose of a Matrix

```
type I = 1 .. N;
var Fib: array [I] of int;
define
  Fib[I] = if I = 1 then 1
           else if I = 2 then 1
           else Fib[I-1] + Fib[I-2];
```

Figure 3: Fibonacci Sequence

```
for I = 1 to n do
  A[I] = [I]
```

Another example is shown in Figure 2, where A is defined as the transpose of B. (B must be defined elsewhere, either by another equation or as an input parameter). The equation is equivalent to the first order logic equation

$$\forall I, J(A[I, J] = B[J, I])$$

Use of the same subscript in A and B denotes the same instance of that subscript, so that the first dimension of A corresponds to the second dimension of B and vice versa.

2.2 Recurrences

Recurrence relations may be expressed in a PS equation. For example, computation of the first N Fibonacci numbers is shown in Figure 3.

Here again we use the subrange type I to establish an index set. The conditional equation defines each element of an array Fib. Notice that we use an array because of the single assignment rule: a variable can receive a

```
type I = 0 .. N;
var X, Y: array [I] of int;
define
  X[I] = if I=0 then 2 else X[I-1]**2;
  Y[I] = if I=N then X[0] else Y[I+1] + X[N-I];
```

Figure 4: A Reverse Ordering

value exactly once. Rather than reassign to the same variable, we give values to successive elements of the vector `Fib`. The vector records the history of the computation of the `N`'th Fibonacci number as in Lucid [2]. Adherence to the single assignment rule allows us to define Fibonacci as a specification. We shall see in the next section that the compiler converts the specification into an iteration, and reduces the vector in size. In this case only three elements are needed regardless of `N`.

This example illustrates a common pattern of recurrences- definition of one or more base cases followed by the recurrence relation for the general case. Use of the -1 and -2 seem to indicate "previous" elements of the array Fib. However, a "previous" element need not be lexicographically lower in index than a successor. Figure 4 illustrates this point.

In this example, the array X is defined through a recurrence as in Figure 3. However, in the equation for Y, the previous element of the sequence is of higher index than the successor. The scheduling component of the PS compiler can generate iterations of either increasing or decreasing index values[1].

2.3 Record Structures

We have shown vectors and matrices being declared and then used in equations. PS also has a rich facility for user-defined types. Figure 5 shows the declaration of a record `xint` which is used to store an arbitrary precision integer. The field `len` holds the size of the integer. `val` holds each seqment of the integer. As the example demonstrates, PS supports dynamically sized arrays. In this record, the size of the array `val` depends on the value of the field `len`.

[1] It cannot however, generate arbitrary orderings.

```
type
  xint = record
         len: int;
         val: array[len-1] of int;
         end;
```

Figure 5: A Record Structure

2.4 The xadd Module

With this introduction, let us now compose a module in PS. This module uses the xint data type shown above. The module xadd, shown in Figure 6 adds two arbitrary precision positive integers. Each item of interest has been numbered on the left. These parenthesized numbers are not part of the input. On line (1) is shown the module header. The module name xadd appears first, followed by the keyword module, the module's input parameters (in parentheses), and the output results (in square brackets). This module has just one output, the item c. The parameters a, b, and c are of type xint, which is defined in the module. It is not necessary in PS to declare a type before it can be used. The parameter BETA is related to the wordsize. It is $2^{wordsize-2}$, so that a.val[i]+b.val[i] does not overflow a word.

Line (2) declares three subrange types, i, t and subi , which are used in the equations. Line (3) shows an alternate way to declaring a two dimensional array from Figure 2. The two methods are interchangeable (as in some Pascals).

Line 4 begins the equations. First N, the upper bound for i, t, and the two arrays sum and carry is defined in terms of the input parameters. The function max must be defined by the user as another module. Line 5 shows the definition of carry, which is a recurrence. The base case is for carry[0], which gives the initial carry in a value of 0. The recursive case is given in the second arm of the conditional. It is defined to be the previous sum plus the previous carry divided by the wordsize parameter BETA. Thus if an overflow is going to occur, the carry will be set to 1, otherwise to 0. Next is the definition of sum. The initial value sum[0] is the result of adding the two input values a and b. This sum cannot be expressed simply as a.val[i] + b.val[i] because the lengths of the two arrays might be different and also, the range of i is one greater than the larger of a and b. Use of a function

```
(1) xadd : module ( a,b : xint; BETA: int ) : [ c : xint ] ;
    type
      xint = record
         len : int ;
         val : array [ 0 .. len-1 ] of int ;
        end ;  (* record *)
(2)    i,t = 1 .. N ; subi = 1 .. c.len;
     var
(3)   sum : array [ 0 .. N ] of array [ 0 .. N ] of int ;
      carry : array [ 0 .. N ] of int ;
      N: int;

     define
(4)    N = 1 + max( a.len , b.len ) ;

(5)   carry[t] = if t=0 then 0
                 else (sum[t-1, t-1] + carry[t-1]) div BETA;

(6)   sum[ t, i ] = if t = 0 then add(a, b)
                    else if ( i = t ) then
                      (sum[ t-1, i ] + carry[t-1] ) mod BETA)
                    else
                       sum[ t-1 , i ] ;

(7)   c.val[subi] = sum[N,i]    ;

(8)   c.len     = if sum[N,N] <> 0  then N
                  else N - 1 ;
 end xadd  ;
```

Figure 6: The xadd Module

a	1	7	3	7	a= 7371		
b	2	6	5	5	b= 5562		
sum[0]	3	13	8	12	0	carry[0]	0
sum[1]	3	13	8	12	0	carry[1]	0
sum[2]	3	5	8	12	0	carry[2]	1
sum[3]	3	5	1	12	0	carry[3]	1
sum[4]	3	5	1	5	0	carry[4]	1
sum[5]	3	5	1	5	1	carry[5]	0

Figure 7: An Example Run of xadd

add aids in modularity. Rather than expressing the sum inline as a large conditional expression , we can postpone definition of the add function to a later stage. In fact, we do not show add here for the sake of brevity. Each digit of the sum may exceed BETA. If a digit is too large, we must do modulo arithmetic and generate a carry to the next digit.

Subsequent versions (sum[i] for i > 0) ripple the carry across the integer. New values of sum are defined only along the diagonal (for the arm of the conditional $i = t$). The other values are copied to the t'th row of sum from the $t-1$st row. For the t'th row and t'th column, the result of the t'th carry is factored into the result.

Lines (7) and (8) define a value for the output parameter c. The val array is simply the last row of sum. The index subi is used rather than i because, the c vector may be one digit smaller than sum. The length is either N, if there was carry, or N −1 otherwise.

Let us trace the behavior of the xadd module when called with the parameters a.val = (1,7,3,7), b.val = (2,6,5,5) and BETA = 8, so that each val[i] can hold a number in the range -8 .. 7 if two's complement representation is used for integers. The numbers are stored with least significant digit first, so that with a= 7371, a.val[0]= 1. Figure 7 shows a tabular representation of carry and sum for each value of t and i in the range.

2.5 The Gauss Module

From arbitrary precision addition, let us turn to a problem in linear algebra. Gauss elimination is a popular technique for solving a set of n equations in n unknowns. The non-pivoting version of the algorithm is adapted from [3] as follows:

Given the $n \times (n+1)$ matrix A containing a square matrix of order n in its first n columns and the n-vector of righthand sides in its last column, we perform the elimination in $(n-1)$ steps, $k = 1, 2, \ldots, n-1$. "In step k, the elements $a_{ij}^{(k)}$ with $i, j > k$ are transformed according to

$$m_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \quad a_{ij}^{(k+1)} = a_{ij}^{(k)} - m_{ik} a_{kj}^{(k)},$$

$$i = k+1, k+2, \ldots, n, \quad j = k+1, \ldots, n, n+1.$$

»

We transcribe these formulas into PS. Figure 8 shows the Gauss module. Notice that in this module, we have put the definitions first and the declarations after. PS allows statements to be given in arbitrary order. The `define` section shows the PS form of the two equations. `m` is the vector of successive multipliers. `aOut` holds intermediate forms of the matrix `a`.

In the module, there are three equations, one for each local variable and one for the output matrix `G`. The definition of `m` uses a conditional expression. For each `i` and `k` such that `i > k`, a multiplier element is defined in terms of the current iteration of `aOut`. All other multiplier elements are 0.

The first version of the `aOut` matrix takes its value from the input matrix `a`. Subsequent versions are defined in terms of previous `aOuts` and previous `ms`. Thus the two arrays `m` and `aOut` are defined by a mutual recurrence. The output from the module is the last iteration of aOut. The module has inputs `a` and `n`. which are respectively the original matrix of simultaneous equations and the number of rows or columns. `a` contains an additional column for the right hand sides of the equations. Output is the array `G` in upper triangular form. Values for the unknowns may be derived from `G` by back substitution.

The declarations consist of type and local variable declarations. The type section shows that three index sets are used. `k` is an iteration index. `i` and `j` are used to index the matrix `a`. `i` is also used to index the array of multipliers.

We have tried to show with the `Gauss` and `xadd` modules representative problems and their solutions in PS. In the next section we address a question which readily comes to mind when examining the PS modules: what is the relationship between data structures declared in PS and those generated in the C program. If the relationship is one-to-one, the generated program is so storage inefficient as to be unusable.

```
Gauss: module(a: array [i,j] of real; N: int):
              [G: array [i,j] of real];


define
m[k,i] = if (i>k) then  aOut[k,i,k] / aOut[k,k,k]
           else  0;

  aOut[k,i,j] = if (k=1) then a[i,j]
                else
                   if ((i>k-1)  and (j>k-1)) then
                      aOut[k-1,i,j] - m[k-1,i]*aOut[k-1,k-1,j]
                  else if (i<=k-1) (* and all j  *) then
                      aOut[k-1,i,j]
                   else  0;

  G = aOut[N];

type
  k, i = 1 .. N;
  j    = 1 .. N+1;

var
  m: array [k, i] of real;     (* multipliers *)
  aOut: array [k,i,j] of real;  (* each successive generation
                                   of a[i,j] is represented
                                   by k'th dimension        *)

end Gauss;
```

Figure 8: Gauss Module

3 Storage Reuse in the Generated Code

The single assignment property of variables in PS makes it possible for the compiler to schedule the equations using only dataflow analysis. However, it also results in a plethora of variables in the PS program, and therefore, if a simpleminded storage allocation is implemented, in the generated program. Indeed, excessive use of storage has long been a criticism of applicative languages in general. Our goal is to have significantly fewer storage locations allocated in the generated program than are declared in the PS program. In this chapter we discuss some techniques to minimize the storage requirements of the generated program. Some of the issues discussed below occur in the reuse of temporaries and in register allocation in compiler optimization [1, 11].

We would like to have the structure of the storage allocated in the generated program resemble the PS data structures as much as possible. Thus we reject a Lisp-like heap in which to store data as linked lists. If the PS structure is an array, we would like the C structure to bear some resemblance to an array; if the PS structure is a record, we would like to generate a C structure declaration. This constraint is imposed to make the interface between PS modules and C modules as simple as possible.

In PS, every variable is local to exactly one module (since there are no global variables and since modules may not be nested). A variable can be an input parameter, an output result, or a local variable. We will first consider storage reuse of local variables and then the problem of efficient parameter passing.

3.1 Virtual Dimensions

Because we are concerned with the large-scale reuse of storage, we will not attempt to reuse scalars within a module. Instead we will concentrate on arrays, in particular, on locating *virtual* array dimensions. (Structures containing arrays are also amenable, with some additional analysis, to these techniques). If an array dimension is physical, that dimension will have the same number of elements in the generated program as in the PS module. If a dimension is virtual, there will be fewer elements in the generated program than in the PS program. The number of representative elements required in the generated program is called the *window* of that dimension. Analysis of the expressions used to subscript an array on the right hand side of an assertion help us locate virtual dimensions and determine the size of the

window of a virtual dimension.

3.2 Virtual Dimensions in Recursive Equations

Example:
Consider the specification of factorial:

```
factorial: module(n: int):[facout: int];
type i = 0 .. n;
var fac: array [i] of int;
define
  fac[i] = if i = 0 then 1 else
           fac[i-1]*i;
  facout = fac[n];
end factorial;
```

fac must be declared as a one dimensional array. However, only one element of fac is needed, fac[n], and to compute any fac[i], at most one other element of fac is needed, fac[i-1]. This suggests that we need only reserve two storage locations for fac, one for the current element and one for the new element being computed. Thus the i dimension of fac is virtual with window size two. The generated program need only have a vector of two elements, regardless of the n. Notice that we have taken a formal specification of factorial, and constructed an iterative program which reuses storage. Although this is a relatively simple transformation from tail recursion to iteration, the same optimization can be applied to non-tail recursive equations (such as the specification of xadd in Figure 6). [5] gives a full discussion of locating virtual dimensions in a set of recursive equations. In his technique, the scheduling component of the compiler looks in recursive array definitions for the pattern $i - k$ on the the i'th dimension of the array reference on the right hand side of the definition. Here, k stands for a manifest constant positive integer.

In the PS compiler, a different technique is used. To locate a virtual dimension, we form the *dependency vector* for each recursive occurrence in an array definition. For each dimension j, the j'th component of the dependency vector is defined as the difference

$$Lhs_j - Rhs_j$$

where *Lhs* and *Rhs* are the subscript expressions used to index the j'th dimension. For factorial, the dependency vector is

$$(i - (i - 1)) = (1)$$

from which we can derive the window size. Notice that by using the dependency vector, we are freed from looking for a specific pattern such as $i - k$. Exactly the same dependency vector is obtained if the recursive equation reads

```
type i = 0 .. n-1;
define
  fac[i+1] = if i = 0 then 1 else
             fac[i]*(i+1);
```

[6] discusses the use of dependency vectors to locate virtual dimensions in the context of scheduling for parallel execution.
The C program generated from the original formulation of the factorial module reads as follows:

```
int factorial(n)
int n;
  {
  int fac[2];
  int facout;
  int i;
  for (i=0; i <= n; i++)
    {
    fac[1] = (i==0) ? 1 :
                fac[0]*i;
    fac[0] = fac[1];
    }
  facout = fac[1];
  return facout;
  }
```

Additional optimizations are possible (for example, eliminating `facout`). Since these can be done by fairly standard optimizing compilers, we will not consider them further here.

3.3 Virtual Dimensions in Nonrecursive Equations

The dependency graph technique is used in the presence of recursive equations. However, it is also possible to have a virtual dimension for an array used in a nonrecursive example.

Example:

```
x[i] = a[i] + b[i];
c[i] = x[i] * x[i];
```

The two equations are not directly or mutually recursive. Depending on the loop structure of the generated code, however, the local variable x can either be a scalar or a vector. If a separate loop is generated for each equation, x must be represented by a vector:

```
for (i=start_i;i<=stop_i; i++)
 x[i] = a[i] + b[i];

for (i=start_i;i<=stop_i; i++)
 c[i] = x[i] + x[i];
```

This schedule can be improved by putting the two equations into a single loop. Not only is there reduced loop set up and evaluate overhead, but the dimensionality of x can be reduced:

```
for (i=start_i;i<=stop_i; i++)
 {
 x = a[i] + b[i];
 c[i] = x + x;
 }
```

Thus a second form of memory optimization in local variables involves maximizing the scope of loops, so that interim variables can have one or more dimension become virtual.

3.4 Parameter Passing

As a single assignment language, PS has copy-in copy-out semantics on parameters to modules. If this mode of parameter transmission is used in the

generated program, serious inefficiency is incurred in the amount of storage used. In fact, other nonprocedural languages [9] have not allowed modules because of these inefficiencies. We deem the module to be indispensible, and therefore make special effort to reduce the overhead of parameter passing.

We again limit our discussion to arrays. Although it also applies to arrays within structures, the latter case is more complicated in details, and adds little to the basic criteria for parameter space reuse.

We would like to replace whenever possible a strict pass-by-value of an array by a pass by reference. However, the parameter passing mode must be consistent. We cannot with one call to module M pass an array A by reference and with another call, pass by value. Therefore, we opt always to pass by reference. For each array A passed to a module in a reference such as

```
(* Assertion q *) X = M(A);
```

where M is a module which returns one result whose type is compatible with X,

- Is A used in an assertion which is scheduled after Assertion q?
- If so, generate code to copy A to a new temporary variable T, and pass to module M a reference to T.
- If not, pass a reference to A itself. In this case, we have avoided the overhead of copying the array.

Now, consider the role of A in M. Let us call the formal parameter A' to distinguish it from the actual parameter A. We are guaranteed that A is passed "by value" in the sense that any change to A' in M does not affect the value(s) of the output parameters of the caller of M. This was guaranteed by the parameter passing analysis above. At the PS level, A' cannot appear on the left hand side of an assertion. This restriction does not of course apply in the generated C code. If there are local variables or output parameters of the same type as A', it might be desirable to reuse A' storage for another variable.

Example:

```
(* C and B are local variables or output
   parameters  of the same type as APrime   *)
```

```
C[i] = APrime[i] * 2;
B[i] = APrime[i] + C[i];
```

Here, after virtual dimension analysis, C is reduced to a scalar. Then, since B has the same type as APrime, we can alias the former to the latter, giving the following output C code:

```
C = APrime[i] * 2;
APrime[i] += C;
```

In this case, we have avoided having the variable B appear in the generated program. Before such a transformation can be done, we must of course insure that all uses of APrime have been completed prior to the reassignment. If such a schedule cannot be effected, (for example, if APrime and B are needed in the same equation), then B must be allocated its own storage.

The analysis just outlined can be used in the Gauss to give the input and output matrices the same locations.

4 Conclusion

In this paper, we have introduced a new nonprocedural language PS in which equations define the relationships between data. The language provides user-defined data types in a Pascal or Modula-2 framework, and accepts a specification which a sequence of module descriptions. The current implementation of the PS compiler is written in Berkeley Pascal using the *llama* parser generator and generates C code.

We have discussed several optimizations to minimize storage requirements in the generated program. We attempt to locate virtual array dimensions, so that the virtual dimension may be replaced in the generated program by a window of elements. We have seen how loop merging can uncover virtual dimensions, so that local variables can be reduced in dimensionality. Parameter passing is another area in which storage reuse is important. We have discussed conditions under which it is possible to pass by reference rather than by value, thus saving space and avoiding unnecessary copying.

5 Bibliography

[1] Aho, A., et. al., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] Ashcroft, E. and Wadge, W., "Lucid, A Nonprocedural Language with Iteration," CACM July 1977.

[3] Dahlquist, G. and Bjork, A., *Numerical Methods*, Prentice-Hall, 1974.

[4] Gokhale, M., "Generating Parallel Programs from Nonprocedural Specifications," 4th JCIT, 1984.

[5] Lu, K.-S., "MODEL Program Generator: System and Programming Documentation," TR 1982, U. of Pennsylvania.

[6] Myers, T. and Gokhale, M., "Parallel Scheduling of Recursively Defined Arrays," submitted to the *Journal of Symbolic Computation*, 1986.

[7] Pneuli, A. and Prywes, N., "Scheduling Equational Specifications and Nonprocedural Programs," Chapter 13 in *Automatic Program Construction Techniques*, MacMillan 1984.

[8] Prywes, N., et. al., "Automatic Program Generation in Distributed Cooperative Computation," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-14, No. 2, 1984.

[9] Prywes, N. et. al., "Programming Supercomputers in an Equational Language," First International Conference on Supercomputing Systems, 1985.

[10] Tinaztepe, C., et. al., "Generation of Software for Computer Controlled Test Equipment for Testing Analog Circuits," *IEEE Transactions on Circuits and Systems*, 1979.

[11] Raoult, J., and Sethi, R., "The Global Storage Needs of a Subcomputation," ACM *Symposium on Principles of Programming Languages*, 1984.

[12] Shi, Y., " Very-High Level Concurrent Programming," Ph.D dissertation in CS, U. of Pennsylvania, 1984.

Standard Bibliographic Page

1. Report No. NASA CR-178196 ICASE Report No. 86-67	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle PS: A NONPROCEDURAL LANGUAGE WITH DATA TYPES AND MODULES		5. Report Date October 1986 6. Performing Organization Code
7. Author(s) Maya B. Gokhale		8. Performing Organization Report No. 86-67
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report 14. Sponsoring Agency Code 505-90-21-01

15. Supplementary Notes

Langley Technical Monitor:
J. C. South

Int. Conf. on Software Engineering

Final Report

16. Abstract

The Problem Specification (PS) nonprocedural language is a very high level language for algorithm specification. PS is suitable for nonprogrammers, who can specify a problem using mathematically-oriented equatins; for expert programmers, who can prototype different versions of a software system for evaluation; and for those who wish to use specifications for portions (if not all) of a program. PS has data types and modules similar to Modula-2. The compiler generates C code.

In this paper, we first show PS by example, and then discuss efficiency issues in scheduling and code generation.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
very high level language, equational specification, automatic program generation	61 - Computer Programming and Software 62 - Computer Systems Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	19	A02

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA Langley Form 63 (June 1985)