Translating Pascal for Execution on a Prolog-based System

M. H. WILLIAMS* AND G. CHEN
Department of Computer Science, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ

One of the objectives of the Japanese Fifth Generation Computer Project is to develop computer systems whose kernel languages are based on logic programming rather than on the conventional imperative languages which have been in general use until now. This has led to conjecture about the problem of the large base of existing software which is implemented in imperative languages. To this end a study has been conducted into the possibility of translating programs written in Pascal-S into Prolog. Because of the radically different control and data structures in these two languages, the translation process is not straightforward. The problems associated with this process are discussed and its performance assessed.

Received October 1984

1. INTRODUCTION

In recent years Prolog,[1,2] a programming language based on symbolic logic, has been used to develop programs in a variety of application areas (e.g. natural language processing,[3] expert systems,[4] database query languages,[5] CAD modellers,[6] etc.). In the process logic programming languages have been found to have many advantages over conventional imperative languages.

The Japanese have recognised the potential of logic programming and have selected it as one of the foundation stones on which to establish their Fifth Generation Computer Project.[7] One of their aims is to develop several different computer systems whose underlying kernel languages will be variants of logic programming.

Since the announcement of the details of the Japanese plan, research on logic programming and its applications has intensified. At the same time concern has been voiced over the problem of the large base of existing software which is implemented in imperative languages, and what might happen to it if computer systems with these radically different architectures were to replace existing systems.

In order to assess the seriousness of this problem a study has been conducted to investigate the possibility of implementing conventional languages on a Prolog machine. For this purpose Pascal was chosen as representative of the class of imperative programming languages. Besides its virtues in the area of well-structured program constructs and rich data structures, Pascal has the advantage of a reasonably precise formal specification,[8–10] which is generally accepted and adhered to.

The radically different semantics (involving both data structures and program constructs) between Pascal and Prolog gives rise to several major problems in the translation process. These include the translation of **goto** statements, the evaluation of expressions, the execution of assignments, the translation of procedure or function bodies and the translation of control statements.

As a first step in preparing a Pascal program for execution on a Prolog machine, all **goto** statements can be removed from it by mapping it into an equivalent Pascal program without **goto** statements. The process for accomplishing this is described in Ref. 11.

To simplify the problem it was decided to concentrate on a subset of Pascal, called Pascal-S.[12] This does not include variant fields within record types, pointer types or packed variables. Set and file types will be discussed briefly but their implementation is not dealt with in full.

The next section discusses the main features of the two languages and the general principles of translation between them. Subsequent sections concentrate on different aspects of the problem.

2. GENERAL PRINCIPLES

In order to understand the problems of translation from a conventional imperative language like Pascal to a declarative language like Prolog, consider briefly the main features of the two languages and the differences between them.

2.1. Pascal

In an imperative programming language such as Pascal, the fundamental mode of operation is based on changing the state of variables through assignments or other similar language constructs. These variables are used to imitate the storage of the underlying machine. This basic dependence upon variables and the association of values with variables is characteristic of a von Neumann architecture.[13]

The execution of a Pascal program which realises some algorithm may be regarded as a sequence of state transformations in which the state of the store or the current point of control or both may change. This is reflected in the three main constructs of the language, namely,

(*a*) declarations which define the attributes of storage locations,

(*b*) assignment statements which effect transformations from one state to another by updating storage locations, and

(*c*) control statements which determine the point of control at any instant and hence the order of state transformations.

2.2. Prolog

By contrast Prolog is a high-level programming language based on a subset of first-order predicate calculus formulae known as Horn clauses. A well-formed

* To whom correspondence should be addressed.

first-order predicate calculus formula in clausal form has the form

$$P_1 \vee P_2 \vee \ldots \vee P_n \vee \neg Q_1 \vee \neg Q_2 \vee \ldots \vee \neg Q_m$$

or $P_1 \vee P_2 \vee \ldots \vee P_n \leftarrow Q_1 \wedge Q_2 \ldots \wedge Q_m$

or $P_1 \vee P_2 \vee \ldots P_n$ if $Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$

A Horn clause is a restricted case in which $0 \leqslant n \leqslant 1$ and has one of the following three forms:

$$P_1 \leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$$

$$P_1$$

$$\leftarrow Q_1 \wedge Q_2 \wedge \ldots \wedge Q_m$$

where the symbol '$\leftarrow$' is read as 'if' or 'is implied by'. The first clause is an implication or rule, the second is an assertion or fact and the third a query.

These three forms of Horn clauses form the basic constructs in Prolog. The first form is that of a Prolog rule in which the predicate on the left-hand side is defined to be the conjunction of goals on the right-hand side (the rule body), e.g.

```
parent(X, Y):- father(X, Y).
```

which may be read as 'X is a parent of Y if X is a father of Y'. The second form reflects a Prolog fact; an example of this might be

```
father(john, mary).
```

which may be read as 'john is the father of mary'. The third form reflects a question to be answered. For example

```
?- father(john, X).
```

which may be read as 'of whom is john the father?'.

The basic data structures used in Prolog are numeric and string constants, variables and compound terms. The concept of a variable in Prolog is totally different from that in Pascal. A variable in Prolog stands for a single object; it will be undefined before it is instantiated (its value is determined), but after it has been instantiated its value may not be altered except when backtracking returns to the point where the variable was instantiated and undoes this instantiation.

2.3. Translating from Pascal to Prolog

Following the state transformation approach a block in Pascal can be regarded as an action which transforms some initial state S_0 into some final state S_n. Each statement ST_i ($1 \leqq i \leqq n$) of the block will transform state S_{i-1} into state S_i, so that the execution of the block as a whole may be represented by the sequence

$$\{S_0\}ST_1\{S_1\}ST_2\ldots\{S_{n-1}\}ST_n\{S_n\}$$

in which ST_i is the ith statement of the block and S_{i-1} is the pre-state vector of ST_i and S_i its post-state vector. Alternatively this may be written as

$$ST_1(S_0, S_1), ST_2(S_1, S_2), \ldots, ST_n(S_{n-1}, S_n)$$

where each ST_i represents a function which maps state S_{i-1} into state S_i.

If each statement ST_i is thought of as being represented by a Prolog rule, a block consisting of statements ST_1, ST_2, ..., ST_n may be translated as

$$block(S_0, S_n) \leftarrow st_1(S_0, S_1), st_2(S_1, S_2), \ldots, st_n(S_{n-1}, S_n)$$

where the subgoals st_1, st_2, ..., st_n are the names of predicates defining the effects of statements ST_1, ST_2, ..., ST_n. These are similar to procedure calls in conventional languages.[14]

Control statements are treated in a similar manner except that some control component is required to test the condition part of the control statement and transfer control accordingly. For example, a **while** statement

```
while c = 1 do begin ST1; ST2; ...; STn end
```

will be translated as the goal

$\ldots statement_i(S_{i-1}, S_i)\ldots$

in the sequence of statements in which it appears and will result in a definition of the form

$$statement_i(S_0, S_{n+1}) \leftarrow C = 1, st_1(S_0, S_1), st_2(S_1, S_2), \ldots, st_n(S_{n-1}, S_n), statement_i(S_n, S_{n+1}).$$

$$statement_i(S, S).$$

Simple statements such as assignment statements, input–output statements and procedure calls do not necessarily require separate rules to define them; they may be incorporated directly in the parent rule.

In practice, it is not necessary to pass the complete state as a parameter from statement to statement; usually a subset of the state, a substate vector, will suffice. This will be discussed in detail in later sections.

3. DECLARATIONS AND STATEMENTS

In order to translate a goto-less Pascal program into Prolog, the program must first be parsed and converted into a suitable internal form.

In the approach adopted the declaration part and the statement part are treated separately. The translation for each part makes use of the ability in Prolog to define a wide range of operators, and to specify for each its precedence and associativity. If all Pascal reserved words are defined as operators, a section of a Pascal program may be treated as a Prolog term and parsed accordingly. The resulting program will have the form of a Prolog term. For example, a **while** statement of form **while** C **do** X may be represented as

```
while(C, X),
```

a sequence of statements as

```
;(S1, S2)
```

where S_1 is the first statement of the sequence and S_2 the remainder of the sequence, and so on.

One set of rules is used to analyse the declarations in a program and will parse each of the definitions for constants, types, variables, etc. Information such as the type and scope of each variable is stored for use in parsing the statement part. The second set of rules is used to parse the statements in the body of a block. The statements are parsed in top-down fashion with information being retrieved and stored as required.

As intimated in the previous section, when a control statement is translated into Prolog, it is not always necessary to pass the complete state vector as a parameter

to it. In order to determine for each statement ST_i the smallest subset of the state vector which will suffice, two sets of variables, M_i and A_i, are defined.

The set M_i consists of those legally accessible variables in the scope of statement ST_i whose values may be modified when ST_i is executed, e.g. destinations of assignment statements, parameters for input statements and actual parameters associated with formal variable parameters whose values are modified in procedures or functions called within ST_i.

The set A_i consists of those legally accessible variables in the scope of statement ST_i whose values may be accessed in any expressions in ST_i, excluding any variables whose values are set within ST_i before being used. For example, in the following **repeat** statement (ST_i)

```
repeat v: = a; w: = w+1; x: = v+w;...until c;
```

the set A_i will contain the variable w but not the variable v.

During the parse a syntax tree is constructed for each body of a block. Each leaf node of this tree represents a simple statement (assignment statement, input–output statement or procedure call); each branch node corresponds to a compound or structured statement in the block. Each branch node has associated with it the two variable sets A and M.

By the end of the parse the source program will be decomposed into a constant table, a variable definition table (which maps each variable to its type), a procedure table (in which each procedure or function name is associated with a parameter list and a syntax tree) and the syntax tree for the main body of the program.

4. VARIABLES AND ASSIGNMENTS

The basic structure of the Prolog output code produced by the translation process is described in Section 2. The body of each Pascal block will be translated into a Prolog rule which calls the subgoals corresponding to the statements in this block. All Pascal variables required in this block must be accessible in this rule, and the proper subsets of the variables will be passed to and updated by the subgoals corresponding to each statement in the block. Thus the compound and structured statements will call the statements in their bodies until a simple statement such as an assignment, procedure call or input–output statement is encountered.

The simple variables in a source program can be handled by being passed as parameters between rules. Each time a variable is updated by an assignment in Pascal, a new variable will be created to substitute for the old one in Prolog. To accomplish this the compiler keeps a record of each Pascal variable name paired with its corresponding Prolog token (a Prolog variable) at any point in the compilation process. The current token for a Pascal variable will be changed whenever the latter is updated by an assignment or input statement or substituted by an actual parameter in a procedure or function call. For example,

```
a: = b+c; b: = b+1;
...a...b...;
```

will be translated as

```
A is B+C,  B1 is B+1,
...A...B1...
```

where '**is**' is a built-in predicate in Prolog which sets the first argument equal to the result of the expression in the second argument.

The structured type array is more difficult to represent since Prolog does not support arrays as such. In the approach adopted a predicate **access** is used to select an element from an array and return this element in a Prolog variable, while a predicate **update** is used to update a single specified element in an array.

Initially Prolog lists were used to represent arrays. The predicates **access** and **update** operated on lists, finding the appropriate element and performing the operation required. However, this proved to be very inefficient. Each access to (or update of) an array required time proportional to the number of elements in the array to perform.

As an alternative to simple lists, height-balanced binary trees were tried (see Section 7). These reduced the access time for an array of n elements from o(n) to $o(\log_2 n)$ which produced a considerable improvement in the execution time of the resulting programs.

It is natural to represent a variable of type **record** as a Prolog term of arity n corresponding to n fields. Such a representation can be nested if the fields of a record type are in turn structured data types such as arrays or other records. A variable of type **set** can be handled as a list if special operations are defined for performing the test of membership of a set, union, intersection and so on. These can be provided as built-in functions to be used in the Prolog program. The file type can also be handled in Prolog without difficulty.

Simple statements such as assignments, input–output statements and procedure calls will not be defined as separate rules, instead they will be translated directly in the context of their parent rules. In translating an expression the code for calling the functions in the expression and the code for evaluating the elements of structured variables will be generated first, the results being stored in temporary variables. Then the predicate **is** is used to evaluate the expression and to instantiate a variable with this value. This variable may be a new version of a simple variable, or the value with which a tree is updated or simply an intermediate variable used, for instance, in testing a condition. When a new version of a variable is created, the compiler will update the present token corresponding to the variable and this token will be used in the following context until the variable is again updated.

5. HANDLING PASCAL PARAMETERS

The approach used to treat variables discussed in the previous section applies equally to variables in the main body of the program as it does to local variables or formal value parameters within a procedure or function. However, the treatment of formal variable parameters and global variables referred to within a procedure or function body poses rather different problems.

The major problem is caused by the fact that two distinct variable names may refer to the same location in a Pascal program, i.e. variable aliasing. This may arise in Pascal-S when the same variable is used as actual parameter for two different formal variable parameters, e.g.

```
var i:integer;
procedure ex1(var x,y:integer);
  begin
    ...
  end;
...
ex1(i,i);
...
```

or when a variable used as an actual parameter in a procedure call is also referred to directly as a global variable within the procedure or function body, e.g.

```
var i:integer;
procedure ex2(var x:integer);
  begin
    ...
    i:=i+1;
    x:=x+i;
    ...
  end;
...
ex2(i);
...
```

The problem may be further complicated by several levels of indirection, e.g.

```
var i:integer;
procedure ex3a(var x:integer);
  procedure ex3b(var y:integer);
    procedure ex3c(var u, v:integer);
      begin      (* body of ex3c *)
        ...
      end;
    begin      (* body of ex3b *)
      ...
      ex3c(y,i);
      ...
    end;
  begin      (* body of ex3a *)
    ...
    ex3b(x);
    ...
  end;
begin      (* main program *)
  ...
  ex3a(i);
  ...
end.
```

or may be disguised as in the case of a subscripted variable, e.g.

```
procedure ex4(var x, y:integer);
  begin
    ...
    x:=...;
    y:=...;
    ...
  end;
ex4(a[i],a[j]);
```

In the approach described in the previous section each Pascal variable x is translated as a sequence of Prolog variables $\{X_1, X_2, \ldots, X_n\}$ in the compilation process. However, when variable aliasing may arise, it is necessary to split the translation into two mappings, one from the domain of variables to that of locations and the other from locations to values.

For each procedure or function P whose set of formal parameters contains at least one variable parameter, a set S can be defined which consists of all variable parameters from the formal parameter set and any global variables referred to in the procedure or function body. If the set S contains more than one member then:

(*a*) When a call to P is compiled, code is generated for each element x in S defining a map @: Int → Value from a location N in the Prolog database to the present value of x (@(N, X)), and a reference to N will be the initial value of x when P is activated. This is followed by the call to P and finally code to collect the value of x from the data base and destroy this map.

(*b*) In compiling the body of P whenever a reference to an element x in S is encountered, a reference is made to the data base (@(N)) using special predicates ac_fp and up_fp defined as:

```
ac_fp(@(N), V) ← @(N, V).
up_fp(@(N), V) ← retract(@(N,_)), assert(@(N, V)).
```

For the sake of efficiency, this process can be improved in two ways:

(*a*) if no call to P can give rise to variable aliasing, the elements of S can be treated in the same way as variables dealt with in the previous section;

(*b*) if calls to P can give rise to variable aliasing, it may still be possible to define for each procedure call of P a subset of S which is relevant.

To handle the latter case where some calls to P may cause variable aliasing, the predicates ac_fp and up_fp are redefined as follows:

```
ac_fp(@(N), V) ← @(N, V).
ac_fp(V, V).
up_fp(@(N), V, @(N)) ← retract(@(N, _)), assert
                       (@(N, V)).
up_fp(_, V, V).
```

In each case the first rule involves a double mapping while the second uses a single map.

When a call to P is compiled, if an element x in S is an alias for at least one other element of S, the map for x and its initial value will be defined and inserted before the call, and code generated to collect the value of x and destroy this map after the call. If x is not an alias for at least one other element of S, the present value of x will be passed to P directly.

The process of detecting variable aliasing can be described briefly as follows.

(*a*) For each call to a procedure or function an entry is created which records the formal variable parameters and the corresponding actual parameters used in the call. In addition, for each procedure or function definition an entry is set up to record each global variable used in the body of the definition.

(*b*) Once the whole program has been scanned and all calls analysed, the closure is constructed, yielding the chain of possible actual parameters corresponding to each formal variable parameter.

(*c*) For each procedure or function P the chains corresponding to the formal variable parameters of P are searched for any elements which occur in more than one chain, or for any global variables which are used in the body of P. If any such element is found, variable aliasing may occur.

(*d*) If two array elements with variable subscripts a[i], a[j], are used as actual parameters, it is, in general, far more difficult to determine at compile time whether aliasing occurs (i.e. . i = j) or not. In this instance it is simply assumed that aliasing may take place.

6. CONTROL STATEMENTS

Before the Prolog code for each compound or control statement is generated, two state vectors, IN and OUT, are constructed for the statement. The variable sets A_i and M_i are used for this purpose as follows.

(1) For the main body of the program the two vectors will both be empty and the corresponding Prolog rule generated is:

$$\text{program} \leftarrow st_1(IN_1, OUT_1), st_2(IN_2, OUT_2) \ldots, st_n(IN_n, OUT_n)$$

where n is the number of control statements in the main body which are not nested within other control statements.

(2) For each procedure or function body, the variables in IN will be those in the formal parameter list together with any global variables accessed in the body. The variables in OUT will be those formal variable parameters and global variables modified in the body. In the case of a function, the function name will also belong to OUT. The Prolog code generated is similar to that for the main body except that two parameter vectors are used in this case.

(3) For each **if** or **case** statement the variables in set OUT are those in M_i whose values are used in subsequent statements at the same level (ST_{i+1}, ..., ST_n), i.e.

$$\{x \mid x \in M_i \wedge x \in (A_{i+1} \cup A_{i+2} \cup \ldots \cup A_n)\},$$

together with any variables which will be used in its parent's successor statements (belonging to its parent's OUT set) but which will not be modified by its own successor statements, i.e.

$$\{x \mid x \in M_i \wedge x \in (OUT(parent(ST_i)) \backslash (M_{i+1} \cup \ldots \cup M_n))\}$$

where OUT(parent(ST_i)), the OUT set of ST_i's parent statement, has already been constructed. Thus the OUT set for conditional statement ST_i will be:

$$OUT = M_i \cap (A_{i+1} \cup \ldots \cup A_n \cup (OUT(parent(ST_i)) \backslash (M_{i+1} \cup \ldots \cup M_n)))$$

The set IN is defined as:

$$IN = A_i \cup OUT$$

i.e. all variables used in ST_i(in set A_i) together with any variables in OUT which return values if the variables are not updated.

The Prolog rule generated for an **if** statement is:

st_i(IN, OUT) ← condition, ...goals for then clause...
st_i(IN, OUT)← ...goals for else clause...

If the condition is satisfied and found to be true then the goals for the **then** clause will be executed. Otherwise the second rule will be attempted and the goals for the **else** clause executed. If the **else** clause is absent, the following rules will be generated:

st_i(IN, OUT) ← condition, ...code for then clause...
st_i(IN, IN) ←

In this case if the condition fails, the post-state vector will be the same as the pre-state vector.

The code generated for a **case** statement ST_i is similar, namely,

$st_i(Label_1, IN, OUT)$ ← ...code for option 1...
$st_i(Label_2, IN, OUT)$ ← ...code for option 2...
...
$st_i(Label_n, IN, OUT)$ ← ...code for option n...
$st_i(\_, \_, \_)$ ← ...error occurs...

(4) For an iterative statement (**for**, **while** or **repeat**) the sets IN and OUT will be defined in the same way as those for conditional statements.

The code generated for a **for** statement will be:

st_i(IN, OUT) ← ...bound condition,
...goals for loop body...,
...modify bound condition...,
st_i(IN′, OUT).
st_i(IN, IN) ←

which states that to execute a **for** statement it is necessary to satisfy the bound condition first. If it holds, the goals for the body will be called one by one, the bound condition will be modified and the goal itself will be called to perform further iterations. If the condition fails, the current state will be returned to the post-state vector of this rule by the second rule.

The format of the code generated for a **while** or a **repeat** statement will be similar to that for a **for** statement.

Example

A simple example is given to demonstrate the translation process. The procedure 'mean' reads a group of integers terminated by a zero and finds the maximum, minimum and average values. The main body simply calls procedure 'mean' and prints the results.

```
program exam(input, output);
  var a,b,c:integer;
  procedure mean(var min,max,ave:integer);
    var a, t, n:integer;
    begin
      read(a); max: = a; min: = a; n: = 0; t: = 0;
      while a > 0 do
        begin
          if a > max
            then max: = a
            else if a < min
              then min: = a;
          n: =n+1; t: = t+a; read(a)
        end;
      ave: = t div n
    end;
  begin
    mean(a,b,c);
    writeln(a); writeln(b); writeln(c)
  end.
```

In the topmost level of procedure 'mean' there are six simple statements and one structured statement (the **while** statement). In the **while** body there are three simple

statements and one structured statement, an **if** statement (called if1), within which another **if** statement (called if2) is nested. For the **while** statement, A = {a, max, min, t, n}, M = {max, min, t, n}. Because the variables max, min, t, n will all be used after the **while** statement, OUT = M, and IN = (A ∪ OUT) = A. For if1, A = {a, max, min}, M = {max, min}. Since variables max and min occur in the OUT set of the parent rule (OUT for while), OUT = M and IN = A. The same applies to if2, where A = {a, min}, M = {min}, OUT = M, IN = A. Thus the Prolog program will be:

```
exam:- mean (_, _, _, A, B, C),
     write(A), nl, write (B), nl,
     write(C), nl.
mean(_, _, _, Min, Max, Ave):- read(A),
     while(A, A, A, 0, 0, Max, Min, T, N),
     Ave is T/N.
while (A, Max, Min, T, N, Max1, Min1, T1, N1):-
     A > 0, if1(A, Max, Min, Max0, Min0),
     N0 is N+1, T0 is T+A, read(A1),
     while(A1, Max0, Min0, T0, N0, Max1, Min1, T1,
     N1).
while(_, Max, Min, T, N, Max, Min, T, N).
if1(A, Max, Min, A, Min):- A > Max.
if1(A, Max, Min, Max, Min1):- if2(A, Min, Min1).
if2(A, Min, A):- A < Min.
if2(_, Min, Min).
```

7. DISCUSSION

The chief differences between Pascal and Prolog from the point of view of translation are the following.

(*a*) Iterative control structures. Whereas most algorithms in conventional languages employ iterative constructs, these have to be achieved by recursive constructs in Prolog.

(*b*) Procedure calls. In general Prolog programs make much heavier use of procedure calls than do Pascal programs. Furthermore, each procedure call in Prolog requires more time than a Pascal procedure call, partly because of the pattern matching which is performed in seeking a matching rule and partly because of the additional control stack management which is required for backtracking.

(*c*) Data representations. In view of the fundamental differences between variables and parameters in Pascal and those in Prolog, each Pascal variable or parameter may be translated as a pair of entries, (name,loc) and (loc,value), in the Prolog database. However, this is very inefficient. A more efficient solution which uses Prolog variables and parameters requires a much greater degree of processing at compile time.

(*d*) Data structures. A fundamental structured data type available in virtually all conventional programming languages is the array. However, arrays are not provided in Prolog and their effect must be achieved by other means. Initially lists were used, but these proved to be too inefficient. Several different alternative approaches involving Prolog terms were tried but problems were encountered due to limitations on the size of such a term and methods of updating it. The best solution tried was a binary tree representation. This requires $o(\log_2 n)$ time to access, which proved more acceptable, although even with this representation the execution time is large for programs which make heavy use of large arrays. Several simple examples have been tested on the C-Prolog system to assess this bottleneck problem. Of six examples tested, three use an array as the main data structure. The sizes of the arrays range from 5 to 400 elements. The time efficiency of compiled code compared with the Unix Pascal interpreter EM1 and with Prolog written by hand is shown in Table 1. CODE/PASC is the ratio of the execution time of code generated to that of the original program running under Pascal EM1, CODE/PROL is that of execution time of generated code to that of an equivalent Prolog program written by hand. Each of the test programs was executed with a number of different sets of data, and the range of ratios presented in the table reflects the range of ratios observed. The results show clearly the bottleneck which arises due to array variables. Although this is not the only source of inefficiency (e.g. heavy recursion is also inefficient), it did appear to be the most important single factor.

Table 1. Ratio of the execution time of the Prolog code generated by this system to that of the original Pascal program running under the EM1 interpreter (CODE/PASC), and of the Prolog code generated by this system to that of an equivalent Prolog program written by hand (CODE/PROL) for a number of sample programs.

Example no.	Uses arrays	CODE/PASC	CODE/PROL
1	Yes	10–25	11–19
2	Yes	10–23	6–10
3	Yes	8–18	3–5.5
4	No	0.9–1.6	1.5–1.9
5	No	1.4–2.4	1.3
6	No	1.3	1.4

The method discussed above has been applied to the translation from Pascal-S into Prolog. In Pascal-S, the data structures are the basic data types integer, real, char and boolean and the structured data types **array** and **record** (without variants). Omitted are the **set** and **file** structures (apart from the standard textfiles input and output), the pointer type and packing options. For the dynamic data type, one solution might be to use a tree to hold all the objects and to pass it as a parameter to the rules.

The 'real' data type was omitted from this particular study since real numbers are not available in some Prolog implementations. However, this does not affect the basic method and algorithm.

Another simplification which Pascal-S makes is the omission of procedure parameters. Although these may be handled by the predicate '= . .' ('univ'), their use complicates the analysis of variables and parameters outlined in the paper and may necessitate the use of the simpler and less efficient approach mentioned in (*c*) above.

Acknowledgement

The authors would like to thank Stuart Anderson for helpful discussions and suggestions and Greg Michaelson for reading and providing comments on this paper. Mr G. Chen is currently supported by a scholarship from the British Council.

REFERENCES

1. W. F. Clocksin and C S. Mellish, *Programming in Prolog*. Springer-Verlag, Heidelberg (1981).
2. P. Roussel, *Prolog: Manual de Référence et d'Utilisation*, Groupe d'Intelligence artificielle, Université d'Aix-Marseille, Luminy (1975).
3. F. C. N. Pereira and D. H. D. Warren, Definite clause grammars for language analysis – a survey of the formalism and a comparison with argumented transition networks. *Artificial Intelligence* **13** (3), pp. 231–278 (May 1980).
4. K. L. Clark and F. G. McCabe, *Prolog: A Language for Implementing Expert Systems*. Department of Computing, Imperial College, London (November 1980).
5. J. C. Neves, S. O. Anderson and M. H. Williams, A Prolog implementation of Query-by-Example. *Proceedings of the 7th International Computing Symposium*, edited H. J. Schneider, pp. 318–332. B. G. Teubner, Stuttgart (1983).
6. J. Camacho Gonzalez, M. H. Williams and I. E. Aitchison, Evaluation of the effectiveness of Prolog for a CAD application. *IEEE Computer Graphics and Applications* **4** (3), 67–75 (March 1984).
7. T. Moto-Oka, Challenge for knowledge information processing systems. In *Fifth Generation Computer Systems*, pp. 3–89. North-Holland, Amsterdam (1982).
8. N. Wirth, The programming language Pascal. *Acta Informatica* **1** (1) 35–63, (1971).
9. *Specification for Computer Programming Language Pascal*. British Standards Institute, BS 6192:1982 (1982).
10. K. Jensen and N. Wirth, *Pascal User's Manual and Report*. Springer-Verlag, New York (1975).
11. M. H. Williams and G. Chen, Restructuring Pascal programs containing GOTO statements, *The Computer Journal*, **28** (2) 134–137 (1985)
12. N. Wirth, Pascal-S: a subset and its implementation. In *PASCAL: the Language and its Implementation*. Wiley-Interscience, New York (1981).
13. J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* **21** (8) 613–641 (August 1978).
14. H. Gallaire, A study of Prolog. In *Computer Program Synthesis Methodologies*, edited A. W. Bierman and G. Guiho, pp. 173–212. Reidel, Dordrecht (1983).