INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

PREDICATE LOGIC AS PROGRAMMING LANGUAGE*

Robert KOWALSKI

University of Edinburgh, Department of Computational Logic
Edinburgh, Scotland

The interpretation of predicate logic as a programming language is based upon the interpretation of implications

B if A_1 and ... and A_n

as procedure declarations, where B is the prccedure name and $A_1,...,A_n$ is the set of procedure calls A_i constituting the procedure body. An axiomatisation of a problem domain is a program for solving problems in that domain. Individual problems are posed as theorems to be proved. Proofs are computations generated by the theorem-prover which executes the program incorporated in the axioms. Our thesis is that predicate logic is a useful and practical, high-level, non-deterministic programming language with sound theoretical foundations.

1. INTRODUCTION

The purpose of programming languages is to enable the communication from man to machine of problems and their general means of solution.

The first programming languages were machine languages. To communicate, the programmer had to learn the psychology of the machine and to express his problems in machine-oriented terms. Higher-level languages developed from machine languages through the provision of facilities for the expression of problems in terms closer to their original conceptualisation.

Concerned with the other end of the man-to-machine communication problem, predicate logic derives from efforts to formalise the properties of rational human thought. Until recently, it was studied with little interest in its potential as a language for man-machine communication. This potential has been realised by recent discoveries in computational logic which have made possible the interpretation of sentences in predicate logic as programs, of derivations as computations and of proof procedures as feasible executors of predicate logic programs.

As a programming language, predicate logic is the only language which is entirely user-oriented. It differs from existing high-level languages in that it possesses no features which are meaningful only in machine-level terms. It differs from functional languages like LISP, based on the λ-calculus, in that it derives from the normative study of human logic, rather than from investigations into the mathematical logic of functions.

This paper deals only in a preliminary way with some of the issues raised by the consideration of predicate logic as a programming language. The semantics of predicate logic as a programming language is investigated in another paper with Maarten van Emden {5}. A more comprehensive investigation of the use of predicate logic for the representation of knowledge is in preparation. Hayes { 8 } and Sandewall {23 } have also concerned themselves with topics related to the programming language interpretation of predicate logic. An earlier investigation with similar objectives was carried out by Cordell Green { 7 }.

2. SYNTAX

All questions concerning logical implication in first order logic can be replaced by questions concerning unsatisfiability of sentences in clausal form.

*This research was sponsored by a grant from the Science Research Council.

Such sentences have an especially simple syntax and lack none of the expressive power of the full predicate calculus. A sentence in clausal form is a set of clauses. A clause is a pair of sets of atomic formulas, written

$$B_1,...,B_m \leftarrow A_1,...,A_n$$

An atomic formula has the form $P(t_1,...,t_k)$ where P is a k-ary predicate symbol and the t_i are terms. A term is either a variable x,y,z,... or an expression $f(t_1,...,t_k)$, where f is a k-ary function symbol and the t_i are terms. The sets of all predicate symbols, function symbols and variables are any three sets of mutually disjoint symbols. Constants are 0-ary function symbols.

3. SEMANTICS

The semantics of sentences in clausal form is as simple as their syntax. Interpret a set of clauses $\{C_1,...,C_n\}$ as a conjunction,

C_1 and C_2 and ... and C_n .

Interpret a clause $B_1,...,B_m \leftarrow A_1,...,A_n$, containing variables $x_1,...,x_k$ as a universally quantified implication,

for all $x_1,...,x_k$, B_1 or ... or B_m
is implied by A_1 and ... and A_n.

The special cases where m = 0 or n = 0 deserve special readings.

If n = 0, read

for all $x_1,...,x_k$, B_1 or ... or B_m .

If m = 0

for no $x_1,...,x_k$, A_1 and ... and A_n.

If both m = 0 and n = 0, write the null clause, □

interpreted as denoting falsity (or contradiction).

Methods for transforming arbitrary first-order sentences into clausal form are described in Nilsson's book {19}. It is our thesis, however, that clausal form defines a natural and useful language in its own right, that thoughts can conveniently be expressed directly in clausal form, and that literal translation from another language, such as full predicate logic, often distorts the original thought.

4. EXAMPLE: A PROGRAM FOR COMPUTING FACTORIAL

(F1) Fact(0,s(0)) ←
(F2) Fact(s(x),u) ← Fact(x,v),Times(s(x),v,u)

Regard the terms 0, s(0), s(s(0)),... as the numerals 0,1,2,... . Read Fact(x,y) as stating that the

factorial of x is y and Times(x,y,z) as stating that x times y is z. Read s(x) as referring to the successor of x. Given a program (or set of clauses) for computing the Times relation, (F1) and (F2) constitute a program for computing the factorial relation. To compute the factorial of the number 2, we add to the program the clause

(F3) ← Fact(s(s0)),x)

which states that no x is the factorial of s(s(0)). This contradicts (F1) and (F2) which logically imply that the factorial of 2 is 2. There exist proof procedures which detect the contradiction by finding the counter-instance s(s(0)) of x which is the factorial of s(s(0)). These proof procedures compute the factorial of 2 without deriving any logical consequences of (F1)-(F4) which do not belong to the computation.

5. EXAMPLE: A PROGRAM FOR APPENDING LIST STRUCTURES.

(A1) Append(nil,z,z) ←
(A2) Append(cons(x,y),z,cons(x,u)) ← Append(y,z,u)

Interpret a term such as cons(x,cons(y,cons(z,nil))) as a list [x,y,z], as is done in such list processing languages as LISP. The constant term nil represents the empty list. Read Append(x,y,z) as stating that z results from appending the list y to the list x. The first clause asserts that the result of appending any list z to the empty list nil is just z itself. The second clause asserts that the result of appending z to the non-empty list cons(x,y) is cons(x,u) where u is the result of appending z to y. To append the list [c] to the list [a,b], we add to the program the clause

(A3) ← Append(cons(a,cons(b,nil)),cons(c,nil),x)

which states that no list x results from appending [c] to [a,b]. This statement contradicts (A1) and (A2) which logically imply that [a,b,c] results from appending [c] to [a,b]. Any correct and complete proof-procedure will prove the unsatisfiability of the set of clauses [(A1),(A2),(A3)]. Some proof-procedures will do so by constructing the counter-instance cons(a,cons(b,cons(c,nil))) of x, without deriving logical consequences of (A1)-(A3) which do not play an essential rôle in the construction.

6. HORN CLAUSES

The preceding two examples used only the Horn clause subset of predicate logic. Robert Hill has shown that, in general, Horn clauses are adequate for defining all relations computable over the domains of Herbrand universes. A Horn clause is a clause

$$B_1,\ldots,B_m \leftarrow A_1,\ldots,A_n$$

containing at most one disjunct in the conclusion, i.e. $m \leq 1$. In order to convert existing programs into Horn clause programs (or, better, to reformulate them) it is useful to bear in mind the procedural interpretation of Horn clauses. There are four kinds of Horn clauses.

(1) $B \leftarrow A_1,\ldots,A_n$ (when neither n=0 nor m=0) is interpreted as a procedure declaration. The conclusion B is interpreted as the procedure name. The antecedent $\{A_1,\ldots,A_n\}$ is interpreted as the procedure body. It consists of a set of procedure calls A_i.

(2) $B \leftarrow$ (when n=0) is interpreted as an assertion of fact. It can be regarded as a special kind of procedure which has an empty body.

(3) $\leftarrow A_1,\ldots,A_n$ (when m=0) is interpreted as a goal statement which asserts the goal of successfully executing all of the procedure calls A_i. A goal statement can be regarded as procedure which has no name.

(4) $\square$ (when n=0 and m=0), the null clause is interpreted as a halt statement. It can be regarded as a satisfied goal statement, i.e. as a nameless procedure with an empty body.

In the rest of this paper we will generally use the term procedure in the wide sense which includes assertions, goal statements and the halt statement as special cases.

7. PROCEDURE INVOCATION

Useful inference systems for demonstrating the unsatisfiability of sentences in clausal form can be formulated without logical axioms and with just a single inference rule called resolution {21}. In the procedural interpretation, resolution is interpreted as procedure invocation. For example, from the goal statement

← Fact(s(s(0)),x)

and from the procedure

Fact(s(x),u) ← Fact(x,v),Times(s(x),v,u)

resolution derives the new goal statement

← Fact(s(0),v),Times(s(s(0)),v,x).

More generally, given a goal statement

$$\leftarrow A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_n$$

and a procedure

$$B \leftarrow B_1,\ldots,B_m$$

whose name B matches the selected procedure call A_i (in the sense that some most general substitution θ of terms for variables makes A_i and B identical, resolution derives the new goal statement

$$\leftarrow (A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n)\,\theta\,.$$

Notice that treating variables as universally quantified within the clause in which they occur means that all variable occurrences are interpreted as local to the procedure in which they occur.

Procedure invocation, in the form of resolution, can also be used to derive new assertions from old assertions using procedures as antecedent theorems in PLANNER {9}. More generally, procedure invocation can be applied to derive new procedures from old procedures. In its general form, given a selected procedure call A_i in a procedure

$$A \leftarrow A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_n$$

and given a procedure

$$B \leftarrow B_1,\ldots,B_m$$

whose name matches (with substitution θ) the selected procedure call, resolution derives the new procedure

$$(A \leftarrow A_1,\ldots,A_{i-1},B_1,\ldots,B_m,A_{i+1},\ldots,A_n)\,\theta\,.$$

In this paper we concern ourselves primarily with the use of resolution to derive new goal statements from old ones.

8. COMPUTATION

The standard notion of computation, applied to Horn clause programs, concerns the repeated use of procedure invocation in order to derive new goal statements from old ones with the ultimate objective of deriving the halt statement. More precisely, given a set S of Horn clauses and an initial goal statement $C_1 \in S$, a computation is a sequence of goal statements $C_1,\ldots,C_n$ such that C_{i+1} is derived by procedure invocation from C_i using a procedure in S whose name matches some selected procedure call in C_i. A computation is successful if it ends with the halt statement, i.e. if $C_n = \square$. A computation terminates without success if the selected procedure call in the end goal statement C_n matches the name of no procedure in S.

Fig. 1 illustrates the only successful computation determined by the program (F1),(F2), activated by the initial goal statement (F3), and employing the criterion of selecting procedure calls of the form Fact (s,t) in preference to calls of the form Times(s',t',u').

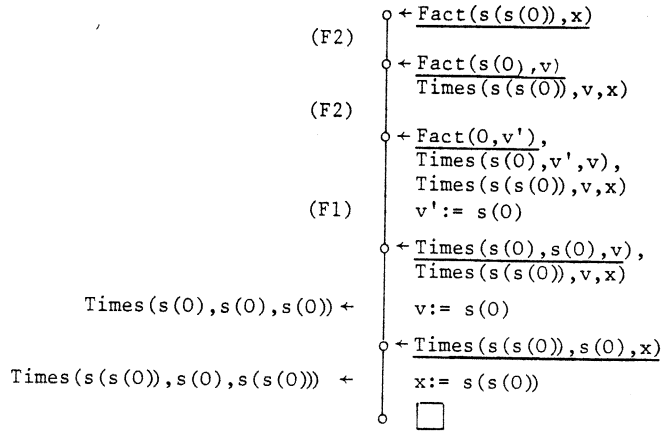

Fig. 1. A computation of the factorial of 2. In each goal statement, the selected procedure call is underlined. The arc, connecting C_i with C_{i+1} is labelled by the procedure used to derive C_{i+1} from C_i. The same arc is labelled by the assignment of terms to variables which is that part of the matching substitution which can be interpreted as passing output from the procedure name to the procedure call.

In the logic interpretation, computations are resolution derivations. The end goal statement of a computation is a logical consequence of the original set of sentences S. In particular, if the computation is successful, then it is a refutation of S, i.e. a demonstration of the unsatisfiability of S. Among existing theorem-proving systems, Loveland's model elimination {14}, Reiter's ordered resolution {20} and our SL-resolution {10} are general purpose systems which, given a set of Horn clauses and an initial goal statement, admit the generation of no derivation which cannot be interpreted as a computation in the sense defined above. Kuehner's system {12} is special-purpose in that it is designed to deal only with sets of Horn clauses. However, his system has a bi-directional facility which can supplement the capability for generating new goal statements from old goal statements with a complimentary capability for generating new assertions from old ones. Our connection graph system {11} is a general purpose system which also provides bi-directional capabilities as well as providing facilities for deriving new procedures from old ones, as in macro-processing.

Except for the connection graph system whose completeness has not yet been demonstrated, all of these systems are complete and correct in the sense that a set of Horn clauses S is unsatisfiable if and only if the inference system admits a refutation of S. All of these systems avoid redundancy by selecting, for the application of procedure invocation, only a single procedure call in every goal statement. Other systems {14,15} which allow only the generation of new goal statements from old ones differ from these by admitting all the n! redundant sequences possible for selecting in turn n procedure calls from a goal statement $\leftarrow A_1,\ldots,A_n$.

In the sequel, we refer to proof procedures which derive new goal statements from old ones, using a selection criterion to avoid redundancy, as top-down procedures, to distinguish them from bottom-up procedures which derive new assertions from old assertions.

9. NON-DETERMINISM

Predicate logic is an essentially non-deterministic programming language. Non-determinism is due to the fact that a given program and activating goal statement may admit more than a single legitimate computation. Consider the following program for selecting an element from a list

(M1) Member(x,cons(x,y)) ←
(M2) Member(z,cons(x,y)) ← Member(z,y)

Fig. 2 illustrates the space of all computations determined by the program (M1),(M2) activated by the goal statement

(M3) ← Member(x,cons(a,cons(b,nil)))

which asserts the goal of finding an x which is a member of the list [a,b].

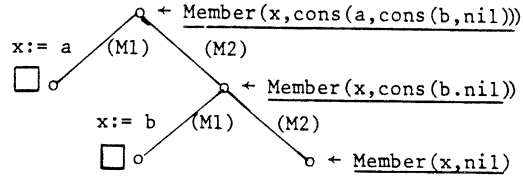

Fig. 2. The space of all computations determined by (M1)-(M3). The space contains two successful computations and one unsuccessfully terminating computation.

The non-determinism of predicate logic programs does not arise in the manner foreseen by McCarthy {16} and Floyd {6}, through the addition to a deterministic language of an explicit amb or choice primitive. Predicate logic is essentially non-deterministic since it provides for the computation of relations, treating functions as a special kind of relation. As in PLANNER non-determinism is implemented by means of pattern-directed procedure invocation. It is the possibility that more than one procedure can have a name which matches a selected procedure call which gives rise to non-determinism.

The implementation of procedure call by pattern-matching has other consequences besides providing a tool for the implementation of non-determinism. In particular, the use of pattern-matching makes it unnecessary to use selector functions for accessing the components of data structures. Thus, for example, when using the cons function symbol for list processing it is unnecessary to use car and cdr functions for accessing the first and second components of pairs cons(s,t). A related use of pattern-matching is for the implementation of conditional tests on the form of data structures. This is illustrated, for instance, in the factorial example where pattern-matching implements a conditional test on the structure of the first argument t of a procedure call Fact(t,u). If t is 0 then the assertion (F1) responds. If t is s(x), for some x, then the recursive procedure (F2) responds. If t is a variable, then both (F1) and (F2) respond non-deterministically.

Predicate logic programs exhibit a second kind of non-determinism due to the fact that procedure bodies consist of a set of procedure calls which can be executed in any sequence. This kind of non-determinism is investigated in the section after next.

10. INPUT-OUTPUT

The generation and application, during procedure invocation, of the substitution θ which matches the selected procedure call A_i in a goal statement

$$\leftarrow A_1,\ldots,A_{i-1},A_i,A_{i+1},\ldots,A_n$$

with the name B of a procedure

$$B \leftarrow B_1,\ldots,B_m$$

has to do with the transfer of input and output. Instantiation of variables occurring in the procedure name B by terms occurring in the procedure call A_i corresponds to passing input from A_i to the body $B_1,\ldots,B_m$ of the procedure through the procedure name. The instantiated procedure body $(B_1,\ldots,B_m)\theta$ is the result of the input transfer. Instantiation of variables occurring in the procedure call A_i by terms occurring in the procedure name B corresponds to passing output (or, rather, partial output) back to the procedure call A_i which distributes it to the remaining procedure calls $A_1,\ldots,A_{i-1},A_{i+1},\ldots,A_n$. The instantiated residue $(A_1,\ldots,A_{i-1},A_{i+1},\ldots,A_n)\theta$

of the original goal statement is the result of this output transfer.

Fig. 3 illustrates the only successful computation determined by the program (A1),(A2) activated by the initial goal statement (A3). The assignments labelling the arc which connects consecutive goal statements C_i and C_{i+1} are the output components of the substitution generated in deriving C_{i+1} from C_i. Notice how the final output x:= cons(a,cons(b,cons(c,nil))) is the composition of the intermediate partial outputs x:= cons(a,x'), x':= cons(b,x''), x'':= cons(c,nil). Computation of output from input is computation by successive approximation. In this example the successive approximations to the final output are x:= cons(a,x'), x:= cons(a,cons(b,x'')), x:= cons(a,cons(b,cons(c,nil))). Notice how the predicate logic notion of procedure differs from the usual notion of a procedure which initially accepts input and eventually returns output only upon successful termination.

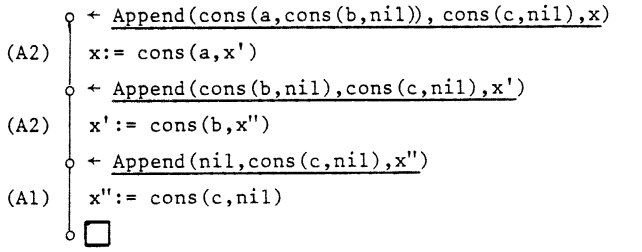

Fig. 3. Computation of output from input by successive approximation.

In fact, predicate logic programs do not explicitly distinguish between input and output. For this reason the rôle of input and output arguments of a procedure name can change from one procedure call to another. For example, in the goal statement

(F4) ← Fact(x,s(0))

the second argument of Fact behaves as an input position whereas the first argument behaves as output position. In the goal statement (F3) the input and output positions are reversed. Fig. 4 illustrates the space of all computations determined by the program (F1),(F2) activated by (F4). Notice how changing the input-output positions of a procedure can turn a deterministic program which computes a function into a non-deterministic program which computes the function's inverse.

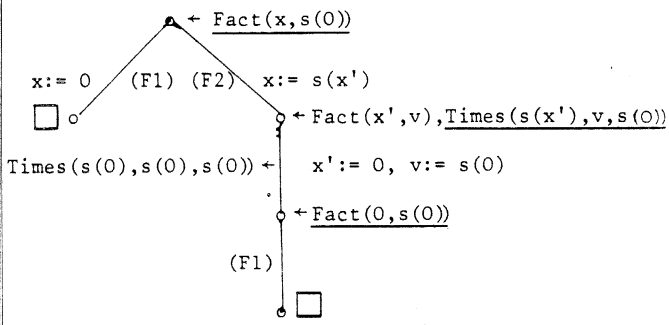

Fig. 4. The transformation of a 'deterministic' program into a non-deterministic one by changing the rôle of input and output arguments.

The ability to exploit the lack of explicit distinction between input and output is available also in the assertional programming languages ABSYS and ABSET [4], which in other ways resemble predicate logic as a programming language.

11. SEQUENCING OF PROCEDURE CALLS

A procedure body consists of a set of procedure calls. Although a top-down proof procedure selects and executes procedure calls in some sequence, the specification of this sequence is not determined by the predicate logic program itself. The sequencing of procedure calls has no syntactic representation. Neither does it have a semantics, in the sense that sequencing does not affect the input-output behaviour of programs. This does not mean that sequencing is not important. Intelligent sequencing of procedure calls is a necessity for practical programming.

Consider the following program for sorting lists. This same program was also investigated for a similar purpose in [11].

(S1) Sort(x,y)← Perm(x,y),Ord(y)
(S2) Perm(nil,nil) ←
(S3) Perm(z,cons(x,y)) ← Perm(z',y),Del(x,z,z')
(S4) Del(x,cons(x,y),y) ←
(S5) Del(x,cons(y,z),cons(y,z')) ← Del(x,z,z')
(S6) Ord(nil) ←
(S7) Ord(cons(x,nil)) ←
(S8) Ord(cons(x,cons(y,z))) ← LE(x,y),Ord(cons(y,z))
(S9) LE(1,2) ← (S10) LE(1,3) ←
(S11) LE(2,3) ← (S12) LE(x,x) ←

Here read Sort(x,y) as stating that y is a sorted version of the list x; Perm(x,y), that y is a permutation of x; Ord(y), that y is ordered; Del(x,y,z), that z results by deleting one occurrence of x from y; and LE(x,y), that x is less than or equal to y.

(S1) states that y is a sorted version of x if y is a permutation of x and y is ordered. If (S1) is interpreted by a top-down proof procedure which selects and completes the execution of the procedure call Perm(x,y) before activating Ord(y), and if in addition the first arguments of Sort, Perm and Ord are considered as input positions, then (S1) can be read as stating that

(S1.1) in order to sort the list x, first generate a permutation y of x, then test that y is ordered; if it is, then y is a sorted version of x.

The meaning of the program does not change, however, if it is interpreted by a top-down proof procedure which selects and completes the execution of Ord(y) before selecting Perm(x,y). In such a case, still reading x as input variable, (S1) can be read as stating that

(S1.2) in order to sort the list x, first generate an ordered list y, then test that y is a permutation of x; if so then y is a sorted version of x.

Clearly the difference in efficiency can be enormous, but the meaning, as determined by the input-output relation Sort(x,y), computed by the program, is the same. It is in this sense that the sequencing of procedure calls can be said to have no semantics.

The use of parallel processes and co-routines is a particular way of sequencing procedure calls. The possibility of independent parallel processing arises when, for example, different procedure calls in the same body share no variables. In such a case, the independent procedure calls can be activated simultaneously and, given a single processor, their execution sequences can be interleaved arbitrarily. On the other hand, the procedure (S1) in the sorting example illustrates a situation where two procedure calls can be executed semi-independently as co-routines. That the use of co-routines is possible in this example is due, in the first place, to the fact that partial output from the procedure call Perm(x,y) is transmitted to the latent call Ord(y) and secondly that such partially specified input can initiate computation as efficiently as totally specified input. Fig. 5 illustrates an unsuccessfully terminating computation determined by selecting for activation an instance of the procedure call Ord(y) before completing the execution of Perm(x,y). If (S1) is interpreted by a top-down proof procedure which selects the procedure call Perm(x,y) before

Ord(y) but interrupts the execution of Perm(x,y) activating Ord(y) in order to monitor the partial output of Perm(x,y), then reading x as input variable, (S1) states that

(S1.3) in order to sort the list x, beginning with the empty sublist nil, first generate an initial sublist of a permutation of x, then test that the sublist is ordered. If it is not ordered, generate another sublist if there is any which has not been generated. If it is ordered but is not a complete permutation, then add another element to the sublist and test that the new sublist is ordered. If it is ordered and is a complete permutation of x, then it is the desired sorted version of x.

The equivalence of (S1.1),(S1.2) and (S1.3) can be demonstrated by noting that they differ only with respect to the different sequencing of procedure calls which they impose on the same program (S1).

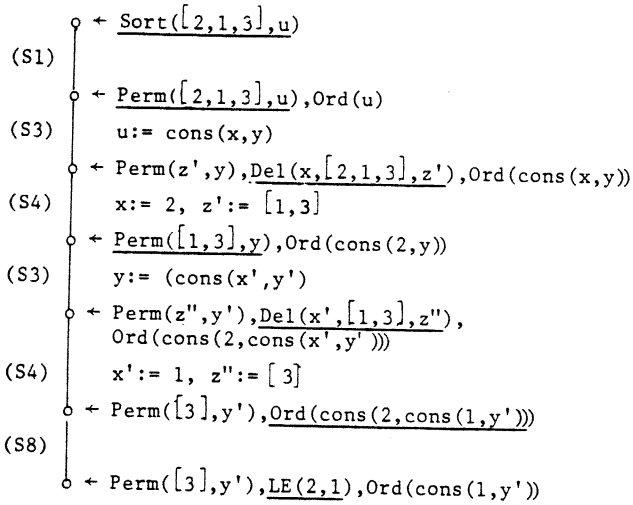

Fig. 5. An unsucessfully terminating computation determined by the program (S1)-(S12) activated by the goal of sorting the list [2,1,3] incorporated in the goal statement ← Sort([2,1,3],u). Here the notation [2,1,3] is an abbreviation for cons(2,cons(1,cons (3,nil))). The computation terminates because no procedure name matches the call LE(2,1).

It is interesting that a sequencing of procedure calls which may be useful for one specification of input and output positions may be unusable for a different specification. Fig. 4 illustrates how a different sequencing of procedure calls is appropriate in (F2) when the second argument of Fact(x,y) is used for input rather than the first. For another example, suppose that the predicate symbols P and Q denote relations which are one-one functions. Consider the procedure declaration

R(x,z) ← P(x,y),Q(y,z).

Given a procedure call of the form R(t,z), where t contains no variables, the first argument position of the call acts as the input position and the second argument acts as output position. The selection of P(t,y) in preference to Q(y,z) in the instantiated procedure body leads to a deterministic computation. The unique output y:= t' of the procedure call P(t,y) is obtained and passed as input to the latent procedure call Q(y,z). The call Q(t',z) then succeeds with unique output z:= t". The alternative selection of Q(y,z) in preference to P(t,y) determines the much less efficient, non-deterministic algorithm which first generates pairs of output (t',t") for the procedure call Q(y,z) and then checks that P(t,t'). However, if the original procedure call has the form R(x,s), where s contains no variables, the first argument acts as output position and the second acts as input position. Efficient sequencing of procedure calls in the instantiated body ←P(x,y),Q(y,s) requires the activation of Q(y,s) in preference to P(x,y).

The viability of predicate logic as a programming language depends upon the eventual provision of an auxiliary control language which would provide a programmer with the ability to specify appropriate sequencing instructions to the interpreting proof procedure. Such a control language ought to be incapable of affecting the meaning of programs, influencing only their efficiency. Some day it may be possible to devise autonomous proof procedures which are able to determine for themselves efficient ways of sequencing procedure calls and of sequencing the application of procedures when more than one responds to a selected procedure call. In the meanwhile, it will not be possible to program effectively without the aid of an auxiliary control language. The importance and utility of such a control language has been argued by Pat Hayes {8}.

ACKNOWLEDGMENT

That sets of axioms are like programs, in the way that different formulations can have equivalent meanings but very different influences on efficiency, is a point of view which runs counter to the prevailing moods in symbolic logic and in artificial intelligence. In particular, the attacks by Anderson and Hayes {1} and by Minsky and Papert {17} against the utility of the theorem-proving paradigm depend upon the assumption that axioms convey meaning but not pragmatic information. Our contrary point of view was reinforced by joint research with Alain Colmerauer (reported in {11}) on axiomatisations of grammars, regarded as programs for syntactic analysis. Further reinforcement was provided by the work of Philippe Roussel {22}, who showed that many uses of the equality relation could be replaced by the more efficiently mechanisable identity relation. Roussel's experience encouraged us to abandon the equality relation altogether, replacing equalities, not interpretable as identities, by implications, as in the procedural interpretation of Horn clauses. The work of Colmerauer and Roussel has since resulted in the elaboration, at the University of Aix-Marseille, of the PROLOG language {3} based on predicate logic. The work of Hayes, arguing that control structures are needed to provide pragmatic information which cannot usefully be expressed by axioms, has now been reported {8}.

Another common impression about theorem-proving is that deduction is completely consequence-oriented and therefore unsuitable for goal-oriented problem-solving. Our contrary attitude was substantiated by our studies of Loveland's model elimination {13} and by our interpretation of model elimination as a goal-oriented resolution system {10}. Later, the discovery by Bob Boyer and J Moore {2} that certain ways of efficiently implementing theorem-provers resemble ways of implementing programming language interpreters, helped to suggest that theorem-provers can be regarded as interpreters for programs written in predicate logic. The work of Boyer and Moore led to the implementation of BAROQUE {18}, an experimental language with a LISP-like interpreter written in predicate logic and interpreted in turn by a resolution theorem-proving program written in POP-2.

The initiation of the work reported in this paper owes much to the profitable interactions we have had with Hayes, Colmerauer, Roussel, Boyer and Moore. In particular, the general thesis that computation and deduction are very nearly the same is due to Pat Hayes. This paper would not have been written, however, without the encouragement and enthusiasm for predicate logic programming of my colleagues David Warren and Maarten van Emden. We owe a special debt to Michael Gordon for his continuing interest and helpful criticsms, and to Aaron Sloman for his detailed and useful comments on an earlier draft of this paper.

This research was initiated during a visit to the University of Aix-Marseille, supported by C.N.R.S. It was continued with the aid of a Science Research Council grant to Bernard Meltzer.

REFERENCES

{1} D.B. Anderson and P.J. Hayes, The logician's folly, D.C.L. Memo No 54, University of Edinburgh, 1972.

{2} R.S. Boyer and J S. Moore, The sharing of structure in theorem-proving programs, Machine Intelligence 7, Edinburgh University Press, Edinburgh, 1972, 101-116.

{3} A. Colmerauer, H. Kanoui, R. Pasero and P.Roussel, Un système de communication homme-machine en français, Rapport preliminaire, Groupe de Researche en Intelligence Artificielle, Université d'Aix-Marseille, Luminy, 1972.

{4} E.W. Elcock, J.M. Foster,P.M.D. Gray,J.J.McGregor and A.M. Murray, ABSET, a programming language based on sets: motivation and examples, Machine Intelligence 6, Edinburgh University Press, Edinburgh, 1971, 467-492.

{5} M. van Emden and R. Kowalski, The semantics of predicate logic as programming language, D.C.L. Memo No 73, University of Edinburgh, 1974.

{6} R.W. Floyd, Non-deterministic algorithms, J.A.C.M. vol. 14, No. 4, 1967, 636-644.

{7} C. Green, Application of theorem proving to problem solving, Proceedings of IJCAI, Washington D.C., 1969, 219-239.

{8} P.J. Hayes, Computation and deduction, Proceedings MFCS Conf., Czechoslovakian Academy of Sciences, 1973.

{9} C. Hewitt, PLANNER: a language for proving theorems in robots, Proceedings of IJCAI, Washington D.C., 1969, 295-301.

{10} R. Kowalski and D. Kuehner, Linear resolution with selection function, Artificial Intelligence 2, 1971, 227-260.

{11} R. Kowalski, A proof procedure using connection graphs, D.C.L. Memo No. 74, University of Edinburgh, 1973.

{12} D. Kuehner, Some special purpose resolution systems, Machine Intelligence 7, Edinburgh University Press, Edinburgh, 1972, 117-128.

{13} D.W. Loveland, A simplified format for the model-elimination theorem-proving procedure, J.A.C.M. vol. 16, 1969, 349-363.

{14} D. Loveland, A linear format for resolution, Proceedings IRIA Symposium on Automatic Demonstration, Springer-Verlag, 1970, 147-162.

{15} D. Luckham, Refinement theorems in resolution theory, Proceedings IRIA Symposium on Automatic Demonstration, Springer-Verlag, 1970, 162-190.

{16} J. McCarthy, A basis for a mathematical theory of computation, Computer Programming and Formal Systems, North Holland, Amsterdam, 1963.

{17} M. Minsky and S. Papert, Progress Report, Artificial Intelligence memo no. 252, M.I.T., January 1972.

{18} J S. Moore, Computational logic: structure sharing and proof of program properties, Part I, D.C.L. Memo No, 67, University of Edinburgh, 1973.

{19} N. Nilsson, Problem solving methods in artificial intelligence, McGraw-Hill, New York, 1971.

{20} R. Reiter, Two results on ordering for resolution with merging and linear format, J.A.C.M. vol 15, no. 4, 1971, 630-646.

{21} J.A. Robinson, A machine-oriented logic based on the resolution principle, J.A.C.M., vol 12, 1965, 23-41.

{22} P. Roussel, Définition et traitement de l'égalite formelle en demonstration automatique, Thèse, U.E.R. de Luminy, 1972.

{23} E. Sandewall, Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs, Proceedings of IJCAI-3, August, 1973, 230-234.