AN EVALUATION OF PROLOG AS A PROTOTYPING SYSTEM

Ute Leibrandt and Peter Schnupp
InterFace Computer GmbH
Oberföhringer Str. 24a

D - 8000 München 81
F.R. Germany

Summary

Some experience has been gathered with Prolog for specification and prototyping of critical parts of interactive information systems. It is felt that this language is the first viable prototype for a really BROAD-BAND formal specification tool allowing immediate testing of the specified system. The main reason for this conclusion is the total abstraction from control flow in Prolog which frees one from the traditional program paradigms (e.g. STRUCTURED, FUNCTIONAL or OBJECT-ORIENTED) and design strategies (TOP down or BOTTOM up). However, although already quite usable for most specification and prototyping problems there is still room for further development of the language and its implementation. Most notably, there is a need for better tracing and debugging features, a more adaptable CONSULTING mechanism, and a virtual terminal model supporting the definition of FORM TERMINALS needed in many business programming tasks.

1. The Language Prolog

Prolog, although originally conceived for artificial intelligence applications, can be used as a formal specification language for software systems because its underlying semantics provide for a very high degree of abstraction. Its computational model is that of a RESOLUTION THEOREM PROVER [KOWALSKI] working on a DATA BASE. This data base consists of two parts. The first one is a set of CLAUSES, i.e. logical facts and rules. It corresponds roughly to the program text of a conventional programming language and may be read in (CONSULTED) from files, built up at runtime (ASSERTED) by the running program, or created interactively by the user. The second data base component is the intermediate data maintained by the Prolog interpreter trying to resolve a current GOAL. This data obeys essentially a stack regime leading to a remarkably simple ABSTRACT STACK MACHINE [KOMOROWSKI].

The current GOAL is a logical statement entered by the user during the man-system-dialogue. The Prolog interpreter tries to prove this goal statement by searching for clauses in the data base which match the goal statement. If the goal contains variables to be proven they are instantiated to constants or logical functions that the variables stand for. This process is called UNIFICATION. If Prolog succeeds with this UNIFICATION AND RESOLUTION it answers YES and prints the variable unifications, if any. Otherwise it answers NO. This does not imply the falseness of the logical statement goal entered but rather its unprovability given the facts and rules present in the data basa. If Prolog is considered as a specification language for commercial applications, this LOGICAL computational model may be seen under another aspect: it implements a highly adaptable query language over an essentially rational data base containing not only facts but also logical rules to derive new facts from those already known.

2. The Prolog Procedure Model

Considering Prolog a LOGICAL AND RELATIONAL DATA BASE LANGUAGE, does not explain its remarkable usefulness for specification and prototyping of application programs to be later realized in a conventional procedural language. This is accomplished by a quite natural, procedural concept introduced into this (essentially nonprocedural) language. A procedure is composed of clauses with the same name and ARITY (i.e. number of arguments). To understand the flow of control in Prolog, the procedure can be visualized as a box with four ports (cf. Fig.1).

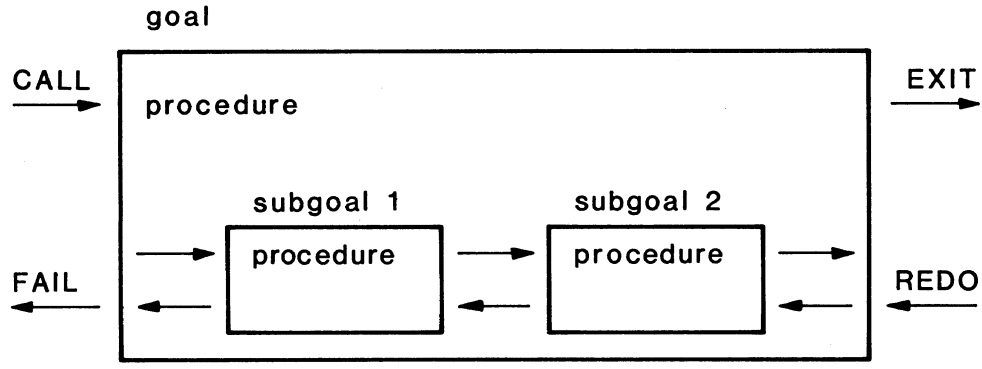

Fig.1: BOX MODEL of Prolog procedures

Entering the box (PROCEDURE) from the CALL port implies that the Prolog interpreter is called to resolve a component clause by satisfying subgoals in the body of that clause. It implies nothing about the result of the call. Subgoals (sub-"procedures") are to be understood as embedded boxes.

Exiting the box through the EXIT port indicates a succesful resolution of the goal procedure.

Entering the box through the REDO port indicates that a later goal has failed and that the system is backtracking in an attempt to find alternatives to previous solutions.

The procedure FAILS, i.e. leaves through the FAIL port, if no resolution succeeds or after all possible resolutions have been found by backtracking.

For any invocation there is always one CALL and one FAIL although there may be arbitrarily many passes through EXITs and REDOs.

For example, if there is the data base:

```
reports_to(white,jones).          % 1
reports_to(mcdonald,jones).       % 2
reports_to(nixon,smith).          % 3
```

the question

```
? reports_to( X, smith).
```

is resolved step by step:

```
CALL  : reports_to( X, smith).      % compare with fact 1
REDO  : reports_to( X, smith).      % compare with fact 2
REDO  : reports_to( X, smith).      % compare with fact 3
EXIT  : reports_to( nixon, smith). % the comparison with
                                    % fact 3 was successful
```

This procedure box may be readily interpreted as a function in a functional programming model.

Therefore, it should not be too surprising that Prolog can be used not only for specification but also for prototyping of dialog applications working on a data base. The usefulness of Prolog surpasses the limits of this particular class of applications. The reason is that the Prolog language model and interpreter provide for two very important and quite unique abstractions.

3. Abstractions in the Prolog Computational Model

The first important abstraction in a Prolog specification (or, we might say a PROGRAM PROTOTYPE, in view of the executability of this specification) is its abstraction from flow of control. As the algorithm to be used for interpretation is always confined to the Prolog interpreter only (the resolution and backtracking strategy of the theorem prover) there is no provision for the usual control constructs, e.g. if-then-else, while or go-to, in the language. The only mechanisms to be employed for influencing the course of computation are

- the sequence of notation of clauses or of the consultation of files, essentially determining the order of alternative choices for backtracking;

- a language element called CUT, signifying the fixation of the unification and resolution choices in the Prolog PROCEDURE currently under evaluation and, hence, CUTTING SHORT the Prolog backtracking;

- recursion as the standard looping mechanism.

Nonetheless, it should be noted that the Prolog backtracking strategy allows a quite natural notation for iteration too: backtracking a sequence of predicates located between the always successful (pseudo-)predicate REPEAT and the always unsuccessful FAIL causes a ping-pong movement of evaluation and backtracking between them.

The second important abstraction in Prolog is that of data flow. A Prolog procedure is always a logical predicate. As the semantics of a predicate knows nothing about INPUT or OUTPUT arguments, the same is true for the formal parameters of a Prolog procedure. Depending on the actual unification of the arguments X and Y when it is called (resolved) a Prolog predicate P(X,Y) may be used to test for the existence of the logical relation P between the constant terms X and Y. It may be a rule for the computation of X from Y or vice-versa, or even a generator of all possible X-Y-pairs.

For example, the below defined predicate APPEND

```
append( [], L , L ).
append( [X|L1] , L2 , [X|L3] ):
        append( L1 , L2 , L3 ).  e:lnct<3
```

may be used to COMPUTE the list Lx which results from appending list [a,b] and list [c,d]

```
?- append( [a,b] , [c,d] , Lx ).
```

```
Lx = [a,b,c,d]
```

or APPEND may be used to generate all possible lists which may have generated the list [a,b,c,d].

```
?- append( Lx , Ly , [a,b,c,d] ).
Lx = []          Ly = [a,b,c,d] ;
Lx = [a]         Ly = [b,c,d] ;
Lx = [a,b]       Ly = [c,d] ;
Lx = [a,b,c]     Ly = [d] ;
Lx = [a,b,c,d]   Ly = [] ;
```

If the user wants to avoid such a sometimes IRRITATING sometimes WANTED ambiguity (as shown in this predicate APPEND example), he will define his tasks more carefully.

This is an extremely useful property for prototyping because it leads to a great conciseness in the logical building blocks to be defined for a given modeling task.

4. Practical Experiences

Prolog was used in several specification and prototyping tasks, most notably for the development of a personal data base system called LEPORELLO and a dedicated data management system to be used for the CONCEPT MODELING method introduced by Ortner and Wedekind [ORTNER]. The most difficult specification problem and hence the most important system aspects to be modeled with a prototype was the user dialog interface for LEPORELLO and the basic data structures and their manipulation in the case of the data management system. Therefore, there was a marked difference in the specification and prototyping strategy, a top-down approach for LEPORELLO versus a bottom-up one for the concept constructor.

Prolog proved itself most suitable in both cases, despite differences in the difficulties encountered. The bottom-up specification could be constructed in a very smooth way. Especially helpful was the fast implementation of test frames for each component, made possible by Prolog. A test driver to generate, in sequence, the test cases considered sufficient requires typically, twenty to thirty lines in Prolog. It can be written routinely in about fifteen minutes. Futhermore these test cases can be quite usable later on as a specification of the test drivers for the final implementation. The testing of one or a few procedures at a time is supported in a very convenient way by the built-in debug features. These provide, essentially, a selective or total tracing of the flow of control through the ports of the procedure box model.

The top-down specification and modeling of LEPORELLO seemed less smooth. Partly, this may be ascribed to the fact that LEPORELLO was our first attempt at using Prolog and suffered from a lack of experience in the use and a less than optimal exploitation of all its features. On the other hand, it is felt that the problems encountered were due to the top-down approach as well.

Firstly, specifying and modeling the deeper levels of the system it got increasingly more difficult to keep the specification crisp and consistent and avoid an ever growing collection of ad-hoc procedures usable at one and only one place. This experience certainly shows not a fault of Prolog but rather a virtue. It makes evident that an exact specification is needed even for the lowest system level to get the prototype running. These problems are usually glanced over as trivial in a conventional specification. Yet they are encountered later during the detail design or even the realization, leading either to a system difficult to modularize and maintain or to expensive specification changes at a late time in systems development.

The second difficulty encountered with the top-down approach, however, seems to be less a problem of Prolog and more one of its debugging facilities as currently defined and implemented. For a specification constructed top-down typically much larger CHUNKS of the system have to be tried in one test run making it difficult to analyze and understand from the trace protocol the flow of control through the various procedures and the unification history. Nevertheless, Prolog can be judged a quite useful prototyping tool for top-down system specifications as well.

5. Consequences of Practical Experiences

The debugging package we used first made it difficult to limit the trace output significantly for error diagnosis. The debugger performed a single step walkthrough. It stopped at the named port (i.e. CALL, REDO, EXIT, FAIL) but nevertheless delivered all steps of flow of control. There existed no practical means to extract specific information regarding particular goals, variables, etc., other than awaiting patiently the arrival of the relevant information.

Therefore we installed a SCREEN-ORIENTED debugger. The screen-oriented debugger divides the screen in four windows. There is the debugger status line, the debugger display area, the debugger command line and the user i/o area.

The debugger performs the following tasks [LEIBRANDT].

help — Print information about all available commands. Enter <return> to leave this command and to reenter the debugger.

abort — Return to the interpreter level. Switch off debugging.

break	Cause the current execution to be suspended and a new copy of the IF/Prolog interpreter to be made available to you. The current database is kept. When you exit from the NEW-COPY interpreter by typing the end-of-file character, your previous program will be resumed [CLOCKSIN].
cont / <return>	Continue execution.
call	Call current subgoal.
back	Backtrack.
repeat	Repeat the last subgoal.
skip	Skip the next subgoal.
leap	Displays the calls and results of the executed goals at the current level. Deeper debugging is LEAPT OVER.
creep	Displays every command that is executed.
spyonly	Watch spy points only.
spyon / spyoff	Enable / disable watching of spy points.
spy	Call the spy point editor.
parent	Show parent goal, i.e. the 1st predecessor goal.
nodebug	Exit debugging mode and continue execution.

At the next step we provided out Prolog interpreter with an exception handler. The exception handling mechanism is used by all built-in predicates and it can be used of by the user's predicates in the exactly same way.

Similar to a debugger who watches the running program and stops execution at specified spypoints to give the control to the user, the exception handler watches the interpretation of the Prolog program and causes an exception ot the specified EXCEPTION GOAL to give the execution to the exception handling program.

The exception handling program may be written by the user. If there is no exception handling program the specified EXCEPTION GOAL simply causes fail.

For example:

The wrong use of the built-in predicate ARG

```
?- arg(4,date(year,month,day),Arg).
```

causes an exception message:

```
EXCEPTION: arg(4,date(year,month,day)) : out_of_range
```

The message may be trapped by the programmer - and even the result NO may be trapped - with the exception handling program, e.g.:

```
exception(out_of_range,arg(_,_,_)) :- true.
```

6. Deficiencies of Current Prolog Implementations

If we consider our Prolog system as typical (it implements approximately the language as described in the QUASI-STANDARD of Clocksin and Mellish's textbook [CLOCKSIN]) it is a workable prototyping tool but not yet a perfect one. Especially worthwhile for this field of application would be a further development of language and system in the following directions.

A shortcoming is the very indiscriminate CONSULTING and RECONSULTING mechanism. Essentially, CONSULTING a file appends all facts and rules in it to the current data base, while RECONSULTING first erases all procedures with the same name already in the data base before it consults the new procedures. CONSULTING in Prolog is a feature which bears strong resemblance to modularization in conventional programming languages. This resemblance has the unfortunate side-effect of misleading the novice user to treat the consulted files as modules. Prolog, however does not maintain the strict modularity, because it treats the data base as a heap. As a consequence if used for prototyping, a file is often RECONSULTED after editing to correct some error found during the debugging session. This is impossible if the definition clauses of some procedure are distributed over more than one consult file. RECONSULTING Prolog would then change all clauses corresponding to the one selected, without regard to the MODULE in which it appears to be located. Clearly, a means to limit the scope of the changes would be desirable, particularly if one intends to adopt the object-oriented paradigm with its object and methods modularization.

Lastly, the character stream oriented i/o model of Prolog makes it difficult to model virtual terminals which are more sophisticated than a simple teletype or line printer. To model the behaviour of the user interface of typical commercial applications a virtual form terminal would be a great help. It should allow one to bind one or more formats to a record describing its appearance on a video screen. The pseudo predicate representing this form terminal would then display the already unified record terms according to the format selected and unify its variable terms with the values input by the user.

7. Conclusion

Despite the shortcomings discussed in the last section Prolog is even in its present form a very satisfying tool for prototyping because it allows to execute and test a formal specification in the framework of a rigid semantic model, a RESOLUTION THEOREM PROVER.

The user of Prolog can formulate his model in the same RELATIONAL manner in which he perceives it, because the underlying model of Prolog allows the problem to be expressed at a higher level of abstraction, independent of the machine oriented, procedural aspects which abound in conventional languages. He describes the relationships of objects (Prolog facts) and the means of making interferences (Prolog rules) from the given set of described relationships. Fortunately, it reveals to the user virtually instantly fundamental logic errors in the user's model. It turns out that in most cases there are more different kinds of errors than the user who tests his specification for the first time with a computer could imagine. Thus Prolog appears to provide a better alternative to the TRANSFORMATIONAL school of software development which starts with untested formal specifications and later on attempts to prove that the programs are CORRECT with respect to this formal (but nevertheless probably faulty) specification. Therefore, from our experience we conclude that formal specification without concurrent prototyping should be considered dangerous.

References

CLOCKSIN, W. F., AND C. S. MELLISH:
Programming in Prolog.
Berlin - Heidelberg - New York: Springer-Verlag 1981.

KOMOROWSKI, H. J.:
An Abstract PROLOG Machine.
Proc. European Conf. on Integrated Interactive Computing Systems (ECICS 82), Stresa September 1982, 149 p. (1982).

KOWALSKI, R.:

Algorithm = logic + control.

CAM. 22 424 p. (July 1979).

LEIBRANDT, U., L. BERNHARD, P. FOLKJAER, AND W. GELDMACHER:

IF/Prolog User's Manual.

München: InterFace Computer 1983.

ORTNER, E.:

Aspekte einer Konstruktionssprache für den Datenbankentwurf.

Darmstadt: Töche-Mittler 1983.