PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

TWO-LEVEL PROLOG

Antonio Porto

Departamento de Informatica
Universidade Nova de Lisboa
2825 Monte da Caparica
Portugal

ABSTRACT

Two-level Prolog is introduced as an alternative to Prolog.

It features clauses with head, body and conditions, that can be read at two levels - at the object level, as an implication from the conjunction of conditions and body to the head; at the meta-level, as an implication from the conditions to the meta-predicate "execution step", whose arguments are head and body. Both implications may have two forms, normal and exclusive. Exclusive implications achieve the effect of Prolog's cut in a more restrained and principled way.

Two-level Prolog is easily implemented on top of Prolog, as each of its clauses can be translated into two Prolog clauses. A summary of the different types of clauses is presented along with their translations.

It is argued that Two-level Prolog promotes a more reliable programming style than Prolog, and is better suited for writing interpreters. Some supporting examples are given.

1 INTRODUCTION

This paper is about an attempt at the design of a basic logic programming language for replacing Prolog. Some ideas trace back to (Pereira and Porto 80) and could later be found in (Porto 82); views put forward in (Pereira 82) were also influential. A full-scale presentation and discussion of the language can be found in the author's doctorate thesis (Porto 84).

With regard to Prolog, two main concerns can be found in Two-level Prolog: a more restricted use of the effects of cuts, aimed at program reliability, and a less restricted notion of execution step, aimed at simpler and more expressive writing of interpreters in the language.

2 THE TWO LEVELS

The writing of Prolog interpreters in Prolog can be achieved through a system predicate that accesses the program clauses ("clause" in Edinburgh Prolog). In the famous three-line interpreter for Prolog,

```
i( true ).

i( (G1,Gn) ) :- i( G1 ), i( Gn ).

i( G ) :- clause( G, B ), i( B ).
```

"clause" becomes associated with the notion of execution step, in this case the unification of a goal with a clause head causing the goal to be replaced by the clause body.

Thus clauses themselves can be seen as defining execution steps. But the choice of a symbol like "clause" to denote the meta-predicate corresponding to the execution step is not the best. Much better is simply using the symbol that stands for the implication in the clauses.

The idea is that a clause can be read at two levels. Given a clause written in the form

```
H <- B.
```

the reading at the object level is that the literal H is implied by B. At the meta-level, the clause is read as an assertion for the meta-predicate "<-", with arguments H and B, and a suitable interpretation is that an execution step on a goal H results in B.

Thus each program statement can be understood to denote in fact two clauses for different predicates - a particular predicate in the head, and the general "execution step" meta-predicate. For the sake of tradition in logic programming, a program statement will continue to be called a clause.

Given the double reading, "<-" can be used instead of "clause", and the previous interpreter will now look like

```
i( true ).

i( (G1,Gn) )  <-  i( G1 ), i( Gn ).

i( G )  <-  ( G <- B ), i( B ).
```

3 THE DOUBLE IMPLICATION

If a clause H<-B can be read as a unit clause for the meta-predicate "<-", one wonders why couldn't there be non-unit clauses for it. They can be introduced by the use of another implication symbol between a meta-head of the form H<-B and a conjunction of meta-literals.

It is preferable to use an implication symbol that is different from "<-", so "<=" will be used. The more general form of clause will then be

H <- B <= C.

where H will be called the head, B the body, and C the conditions. "<-" binds tighter than "<=", so that the clause is equivalent to (H <- B) <= C.

The meta-level interpretation is that an execution step on a goal matching the head will result in the body if the conditions are true. This reading is equivalent to the Prolog clause

(H <- B) :- C.

for the predicate "<-". Thus the notion of an execution step is extended to include both the unification and the evaluation of the conditions, which can then be viewed as an extension of the unification, further restraining the admissible patterns of arguments in the goal.

At the object level, we can read the clause as a double implication

(H <- B) <- C

that is logically equivalent to

H <- C ^ B

and so the reading is that a literal H is implied by the conjunction of literals C and B. The equivalent Prolog clause is

H :- C, B.

for the predicate of H.

Notice that literals in the conditions can be seen both as meta-level and object level literals. There is in fact a collapsing of the two levels, for a program is still seen to correspond to a single set of axioms with respect to which all predicates are to be evaluated. Accordingly, there is no distinction between meta-level and object level variables. The two-level distinction is just a good way to interpret the relation between the two readings of a clause, that can be used for control purposes.

Going back to the interpreter, we can rewrite its third clause as

i(G) <- i(B) <= G <- B.

This form is rather nice, for it defines the execution step for the interpreter "i" as being equivalent to the execution step for the underlying interpreter.

4 THE EXCLUSIVE IMPLICATION

The main limitation of Horn clause logic as a practical programming language is the lack of an IF-THEN-ELSE construct, for providing both concise and efficient definitions. It was introduced in the Prolog language under the disguised form of the cut operator, which is in fact more powerful in a dangerous way.

It is proposed here to replace the use of cuts by the use of a more restrained construct, meant just to provide IF-THEN-ELSE definitions for predicates. This construct is the exclusive implication, represented at the object level by "<->" (binding tighter than "<=") and at the meta-level by "<=>" (binding looser than "<-").

An exclusive clause like

H <-> B <= C.

corresponds, on a first approximation, to an if-and-only-if definition for the predicate of H, justifying the choice of symbol for the implication. The exact meaning is more complex and can only be formulated in the context of the sequence of clauses for the predicate: At the object level, IF a literal matches H and C is true, THEN the literal is equivalent to the resulting instance of B, ELSE its value is defined only by the following clauses for the predicate; at the meta-level, IF a goal matches H and C is true, THEN an execution step on that goal results in B, ELSE it is defined by the following clauses. The two readings of an exclusive clause correspond to two IF-THEN-ELSE definitions with the same IF part.

A logical declarative interpretation of the IF-THEN-ELSE exists, and corresponds to a disjunction between a conjunction of the IF part with the THEN part and a conjunction of the ELSE part with the negation of the IF part. Implementation of this interpretation would be ideal, but there are obvious difficulties in dealing with negation. The simplest approximation is the cut-like approach of evaluating the first solution of the IF part and then committing either to the THEN or the ELSE part, according to whether there is such a solution or not. This will be the approach we assume here. It is sound only in those cases where the IF part is just a

test in which no variables are bound, its evaluation resulting then in either TRUE or FALSE.

In case the exclusive implication is the meta-level one we have a clause

```
H <- B <=> C.
```

whose definitions' IF part is just the unification of a goal with the head H.

A formal definition of the semantics of Two-level Prolog clauses can be found in (Porto 84).

5 THE LANGUAGE AND ITS IMPLEMENTATION

Two-level Prolog can be easily implemented on top of a Prolog system, by writing a translator from each statement into the two Prolog clauses that correspond to its two readings.

For syntactic sugaring we define the symbols "!" and "?" as standing for an exclusive implication when the body is, respectively, equivalent to TRUE or FALSE.

A summary will now be presented of all possible forms of clauses accepted in the language, along with their two translations into Prolog. For appropriate translation of exclusive clauses at the meta-level, the unification of the second argument of a goal for the predicate "<-" with the body of the clause must be performed only after the commitment. Notice also that the execution step for clauses with "?" always fails, instead of succeding with "fail".

Two-level Prolog	Prolog	
	Object level	Meta-level
`H.`	`H.`	`H <- true.`
`H <- B.`	`H :- B.`	`H <- B.`
`H <= C.`	`H :- C.`	`H <- true :- C.`
`H <- B <= C.`	`H :- C, B.`	`H <- B :- C.`
`H <-> B.`	`H :- !, B.`	`H <- X :- !, X=B.`
`H <-> B <= C.`	`H :- C, !, B.`	`H <- X :- C, !, X=B.`
`H ! .`	`H :- !.`	`H <- X :- !, X=true.`
`H ? .`	`H :- !, fail.`	`H <- _ :- !, fail.`
`H ! <= C.`	`H :- C, !.`	`H <- X :- C, !, X=true.`
`H ? <= C.`	`H :- C, !, fail.`	`H <- _ :- C, !, fail.`
`H <=> C.`	`H :- !, C.`	`H <- X :- !, X=true, C.`
`H <- B <=> C.`	`H :- !, C, B.`	`H <- X :- !, X=B, C.`

6 PROGRAMMING METHODOLOGY

Typically, Prolog programmers will use some cuts as add-ons that are placed in certain locations in the program after a first version has been written without them, so that backtracking into certain parts can be avoided; a common practice is to add as few cuts as possible. This usually leads to cuts being placed in positions that should be considered wrong, in the sense that some definitions of predicates only work properly because they are used in certain contexts, and should they be lifted from there they wouldn't anymore. A typical case is putting a cut at the end of each body of a clause for a certain predicate, in order to make it deterministic. This relieves the programmer from worrying about making deterministic every definition of a predicate used in those bodies, as they should, thus making them incorrect and not reusable in other contexts. In instances where one would really like to produce just the first solution to a non-deterministic goal, this should be made explicit by wrapping the goal with a meta-predicate for doing so, instead of relying on a conjoined cut that has a broader effect.

Two-level Prolog is meant to encourage a better logic programming methodology, mainly by virtue of its more restricted use of

commitment. In writing a clause one should decide whether it is defining a deterministic step or not. In case it is, an exclusive clause should always be used, checking whether the corresponding IF just amounts to a unification pattern or needs the expressing of conditions.

Since there is at most one commitment per clause, programs become more readable. In the definition of a predicate, exclusive clauses define an IF-THEN-ELSE chain, and the other clauses define non-determinism.

As an example, take this Prolog version of "quicksort" :

```
quicksort(L,S) :- qs(L,S,[]), !.

qs(X1.Xn,S,Sn) :- partition(Xn,X1,L,R), qs(L,S,Si), qs(R,Si,Sn).
qs([],S,S).

partition(X1.Xn,X,X1.L,R) :- X1 =< X, partition(Xn,X,L,R).
partition(X1.Xn,X,L,X1.R) :- partition(Xn,X,L,R).
partition([],_,[],[]).
```

Although it works, "partition" is ill-defined and should not be used outside "quicksort". The correct Two-level Prolog version should look like this:

```
quicksort(L,S) <-> qs(L,S,[]).

qs(X1.Xn,S,Sn) <-> partition(Xn,X1,L,R), qs(L,S,Si), qs(R,Si,Sn).
qs([],S,S) ! .

partition(X1.Xn,X,X1.L,R) <-> partition(Xn,X,L,R)  <=  X1 =< X.
partition(X1.Xn,X,L,X1.R) <-> partition(Xn,X,L,R).
partition([],_,[],[]) ! .
```

One can see at a glance that all execution steps defined by the clauses are deterministic.

The definition for "partition" is now declaratively correct in terms of logical IF-THEN-ELSE semantics. With a cut-like implementation of exclusive clauses it is only operationally correct when used with its first two arguments ground, as it is expected to be; this is not, however, expressed in the language, and with a correct implementation of the alluded semantics the program would behave correctly.

The arithmetic test in "partition" is now in a more pleasing situation as a condition for a particular type of recursion step.

The exclusive implications used in the last clauses for each predicate should not be regarded as superflous, but rather as constituting a declarative closing of the definitions for the patterns of arguments that were used. This is good practice in view of possible later extensions of the definitions.

As another example of the exclusive implication methodology, the final version of the three-line interpreter for Two-level Prolog should be

```
i( true ) ! .

i( (G1,Gn) ) <-> i( G1 ), i( Gn ).

i( G ) <- i( B )  <=>  G <- B.
```

7 THE WRITING OF INTERPRETERS

The importance has long been recognized of the ability to write logic programming interpreters in the logic programming language itself. We will briefly try to show how Two-level Prolog is more suitable than Prolog for writing interpreters. Two main reasons stand out.

First. If we look back at the two

three-line interpreters for Prolog and Two-level Prolog, we can see there is a fundamental difference between them. Whereas the Prolog interpreter only works for programs without cuts, the Two-level Prolog interpreter works for any program written in the language. This is because exclusive definitions of predicates translate into exclusive definitions for the corresponding execution steps. In Prolog, cuts have to be dealt with by a non-trivial extension of the interpreter.

Second. The Two-level Prolog clauses

```
H <- B  <=  C.
```

and

```
H <- C, B.
```

are declaratively equivalent at the object level definitions for H, but differ operationally at the corresponding meta-level definitions for an execution step on H. Thus conditions provide control for defining the amount of computation on particular execution steps, and this can be put to use with very simple interpreters.

An illustrating case is that of co-routining. In its simplest form it is an alternation of steps between co-routined goals. Using the connective "/" as a meta-predicate to join goals in co-routining, the interpreter reduces to this:

```
G1/Gn <- NG  <=>  ( G1 <- NG1 ),
                  ( Gn <- NGn ),
                  ( NG1/NGn >> NG ).

( true/G >> G ) ! .

( G/true >> G ) ! .

( G >> G ) ! .
```

The interpreter defines a step on coroutined goals as doing a step on each one, and then joining the results in coroutining under "/", taking care of termination. As an example of its use, take the following definition of a program to find a path with no loops between two points of a graph:

```
loop_free_path(A,B,P) <-> path(A,B,P) /
                          no_loop(P).

path(A,B,A.B.[])  <=  link(A,B).

path(A,B,A.X) <- path(C,B,X)  <=>  link(A,C).

no_loop([]) ! .

no_loop(X1.Xn) <-> test(Xn,X1) / no_loop(Xn).

test([],_) ! .

test(X._,X) ? .

test(_.Xn,X) <-> test(Xn,X).
```

A recursive step on "path" produces one node for the path, and a recursive step on each of "noloop" and "test" consumes one such node, so the system is well synchronized.

Another interesting application is an interpreter with an explanation facility based on the trace of execution steps. In programs to be interpreted, goals corresponding to unwanted detail in the explanations can usually be stated in conditions rather than bodies of clauses, and so will not be traced.

ACKNOWLEDGEMENTS

Luis M. Pereira first suggested to me that program statements be stored as clauses for a meta-predicate corresponding to the execution step. I also thank him for all the useful discussions that helped me shape the ideas expressed in this paper.

REFERENCES

Pereira, L.M. Logic Control with Logic. in Implementations of Prolog, ed. J. A. Campbell, Ellis Horwood, Chicester, 1984

Pereira, L.M. and Porto, A. Intelligent backtracking and sidetracking in Horn clause programs - the theory. Internal report, Dep. de Informatica, Universidade Nova de Lisboa, 1980

Porto, A. Epilog: a language for extended programming in logic. in Implementations of Prolog, ed. J. A. Campbell, Ellis Horwood, Chichester, 1984

Porto, A. Controlo sequencial de programas em logica. Doctorate thesis (in portuguese), Dep. de Informatica, Universidade Nova de Lisboa, 1984