32002: AI (Logic programming and Prolog.) **Spring 2015**

Lecture 8: Feb. 02, 2015

Lecturer: K.R. Chowdhary *: Professor of CS (VF)*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

8.1 Introduction

Prolog is a logic programming language, implemented in two parts:

1. *logic*, which describes the problem, and
2. *control*, provides the solution method.

This is in contrast to other programming languages, where description and solution go together, and they are hardly distinguishable. This, feature of prolog helps in separate developments for each part, one by the programmer and other by implementer.

Prolog is useful in problem areas, such as artificial intelligence, natural language processing, databases, etc., but pretty useless in others, such as graphics or numerical algorithms. The main applications of the language can be found in the area of Artificial Intelligence; but PROLOG is being used in other areas in which symbol manipulation is of prime importance.

8.2 Logic Programming

In more conventional languages such as C, C++, and Java, a program is a specification of a sequence of instructions to be executed one after the other by a target machine to solve the problem concerned. The *description* of the problem is incorporated *implicitly* in this specification, and usually it is not possible to clearly distinguish between the *description* (logic) of the problem, and the *method* (control) used for its solution. In logic programming, the description of the problem and the method for its solution are explicitly separated from each other.

$$\boxed{algorithm = logic + control} \tag{8.1}$$

The term 'logic'in equation (8.1) indicates the descriptive component of the algorithm, that is, the description of the problem; the term 'control'indicates the component that tries to find a solution, taking the description of the problem as a point of departure. So, the logic component defines what the algorithm is supposed to do; the control component indicates how it should be done.

A specific problem is described in terms of relevant objects and relations between objects, which are then represented in the *clausal* form of logic - a restricted form of first-order predicate logic. The logic component for a specific problem is generally called a *logic program.* The control component employs logical deduction or

reasoning for deriving new facts from the logic program, thus solving the given problem (using the deduction method). The splitting of an algorithm into a logic component and a control component has a number of advantages:

- The two components may be developed independent of each other. For example, when describing the problem we do not have to be familiar with how the control component operates on the resulting description; thus knowledge of the declarative reading of the problem specification suffices.
- A logic component may be developed using a method of stepwise refinement; we have only to watch over the correctness of the specification.
- Changes to the control component affect (under certain conditions) only the efficiency of the algorithm; they do not influence the solutions produced.

The implementation of Logic programming is explained in figure 8.1.

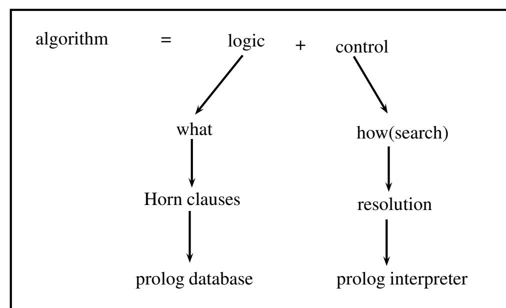

Figure 8.1: Logic Programming.

When all facts and rules have been identified, then a specific problem may be looked upon as a query concerning the objects and their interrelationships.

8.3 Logic vs. Control

Different control strategies for the same logical representation generate different behaviors. Also, the information about a problem-domain can be represented using logic in different ways. Alternative representations can have a more significant effect on the efficiency of an algorithm compared to alternative control strategies for the same representation.

Consider the problem of sorting a list. In one representation, we can have a definition with one assertion consisting of two arguments.

sorting x *gives* $y \leftarrow y$ *is a permutation of* x, y *is ordered.*

Here $\leftarrow$ is read as 'if'and ','is logical *and* operator. The first argument generates permutations of x and then it is tested whether they are ordered. Executing procedure calls as coroutines the procedure generates permutations, one element at a time. Whenever a new element is generated, the generation of other elements is suspended while it is determined whether the new element preserves the orderedness of the existing partial permutation.

Thus the efficiency of an algorithm can be improved through two different approaches, either by improving the logic component or by leaving the logic component unchanged and improving the control over its use.

In a logic programming system, specification of the control component is subordinate to specification of the logic component. The control component can either be expressed by the programmer in a separate control-specifying language, or it can be determined by the system itself. When logic is used, as in the relational calculus, for example, to specify queries for a knowledge-base, the control component is determined entirely by the system.

In "quicksort", like the predicates "permutation" and "ordered" before, the predicates "empty," "first," "rest," "partitioning," and "appending" can be defined independently from the definition of "sorting." Consider that, partitioning x_2 by x_1 is intended to give the list u of the elements of x_2 which are smaller than or equal to x_1 and the list v of the elements of x_2 which are greater than x_1.

$$\begin{aligned}
&sorting\ x\ gives\ y \leftarrow x\ is\ empty,\ y\ is\ empty \\
&sorting\ x\ gives\ y \leftarrow first\ element\ of\ x\ is\ x_1,\ rest\ is\ x_2, \\
&\qquad partitioning\ x_2\ by\ x_1\ gives\ u\ and\ v, \\
&\qquad sorting\ u\ gives\ u', \\
&\qquad sorting\ v\ gives\ v', \\
&\qquad appending\ w\ to\ u' gives\ y, \\
&\qquad first\ element\ of\ w\ is\ x_1, \\
&\qquad rest\ of\ w\ is\ v'.
\end{aligned}$$

8.3.1 Procedure-calls Execution

In the simplest backward reasoning based execution strategy, procedure calls are executed one at a time in the sequence in which they are written. Typically, an algorithm can be improved by executing the same procedure calls either as coroutines or as communicating parallel processes. A new algorithm (A_2),

$$A_2 = L + C_2$$

is obtained from the original algorithm (A_1),

$$A_1 = L + C_1$$

by replacing one control strategy (C_1) by another (C_2) leaving the logic (L) of the algorithm unchanged. For example, executing procedure calls in sequence, the procedure,

$$sorting\ x\ gives\ y \leftarrow y\ is\ a\ permutation\ of\ x,\ y\ is\ ordered$$

first generates permutations of x and then tests whether they are ordered. By executing procedure calls as coroutines, the procedure generates permutations, one element at a time. Whenever a new element is generated, the generation of other elements is suspended while it is determined whether the new element preserves the orderedness of the existing partial permutation.

8.3.2 Backward vs. Forward Reasoning

Recursive definitions are common in mathematics, where they are more likely to be understood as forward-chaining rather than backward-chaining. Consider, for example, the definition of factorial:

$$\begin{aligned} &factorial\ of\ 0\ is\ 1 \leftarrow \\ &factorial\ of\ x\ is\ u \leftarrow y\ plus\ 1\ is\ x, \\ &\qquad factorial\ of\ y\ is\ v, \\ &\qquad x\ times\ v\ is\ u. \end{aligned}$$

The mathematician is likely to understand such a definition forward-chaining, generating the sequence of assertions as follows:

$$\begin{aligned} &factorial\ of\ 0\ is\ 1 \leftarrow \\ &factorial\ of\ 1\ is\ 1 \leftarrow \\ &factorial\ of\ 2\ is\ 2 \leftarrow \\ &factorial\ of\ 3\ is\ 6 \leftarrow \\ &\qquad and\ so\ on. \end{aligned}$$

Conventional programming language implementations understand recursions as backward-chaining. Programmers, accordingly, tend to identify recursive definitions with backward-chaining execution. However, there is one exception, and that is Fibonacci series, which is efficient when interpreted as forward reasoning. It is left as an exercises for the students to verify the same.

8.4 Prolog Program Structure

Prolog is a declarative (i.e., descriptive) language. Programming in Prolog means describing the *world*. Using such programs means asking Prolog questions about the previously described world. The simplest way of describing the world is by stating facts, like "elephant is bigger than horse", as,

```
bigger(elephant, horse).
```

Let's add a few more facts to our this program to appear as:

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

This is a syntactically correct program, and after having compiled it, we can ask the Prolog system questions (or *queries*) about it.

```
?- bigger(donkey, dog). <enter>
Yes
```

The query 'bigger(donkey, dog)'(i.e. the question "Is a donkey bigger than a dog?") succeeds, because the fact 'bigger(donkey, dog)'was previously communicated to the Prolog system. Our next query is, "is a monkey bigger than an elephant?"

```
?- bigger(elephant, monkey).
No
```

The reply by prolog is "No". The reason for this is that, the program has said nothing about the relationship between elephants and monkeys. Still, we note that, the program says - 'elephants are bigger than horses', and 'horses are bigger than donkeys', which in turn are bigger than monkeys. Thus, elephants are also to be bigger than monkeys. In mathematical terms: the bigger-relation is *transitive*. But this has also not been defined in our program. The correct interpretation of the negative answer Prolog has given in above is the following: "from the information communicated to the program it cannot be proved that an elephant is bigger than a monkey". As an exercise, we can try the proof by resolution refutation and would find that it cannot be proved.

Solution would be to define a new relation, which we will call *isbigger*, as the *transitive closure*. Animal X is bigger than animal Y either if this has been stated as a *fact*. Otherwise, there is an animal Z, for which it has been stated as a fact that animal X is bigger than animal Z, and it can be shown that animal Z is bigger than animal Y. In Prolog such statements are called *rules* and are implemented as follows:

```
isbigger(X, Y) :- bigger(X, Y).                 %rule1
isbigger(X, Y) :- bigger(X, Z), isbigger(Z, Y). %rule2
```

where ':-'stands for 'if'and comma (,) between 'bigger(X, Z)'and 'isbigger(Z,Y)'stands for 'AND', and a semicolon (;) for 'OR'. If from now on we use 'isbigger'instead of 'bigger'in our queries, the program will work as intended, as shown in query below:

```
?- isbigger(elephant, monkey).
Yes
```

In the rule1 above, the predicate 'isbigger(X, Y)'is called *goal*, and 'bigger(X, Y)'is called *sub-goal*. In the rule2 'isbigger(X,Y)'is goal and the expressions after the sign ':-'are called sub-goals. The goal is also called *head* of the rule, and the expressions after sign ':-' is called *body* of the rule statement.

In fact, the rule1 above corresponds to the predicate,

$$if\ bigger(X, Y)\ then\ isbigger(X, Y),$$

or

$$bigger(X, Y) \rightarrow isbigger(X, Y).$$

Similarly, predicate expression for rule2 is

$$bigger(X, Z) \wedge isbigger(Z, Y) \rightarrow isbigger(X, Y).$$

The expressions which are not conditionals, i.e., like,

```
bigger(elephant, horse).
bigger(horse, donkey).
bigger(donkey, dog).
bigger(donkey, monkey).
```

are called *facts*(or assertions). The facts and rules, together, make the knowledge-base (KB) in a program.

For the query 'isbigger(elephant, monkey)'the Prolog still cannot find the fact 'bigger(elephant, monkey)'in its database, so it tries to use the second rule instead. This is done by matching the query with the head of the rule, which is 'isbigger(X, Y)'. When doing so, the two variables get bound: $X =$ elephant, and $Y =$ monkey. The rule says that in order to prove the goal 'isbigger(X,Y)'(with the variable bindings that's equivalent to isbigger(elephant, monkey)), Prolog needs to prove the two subgoals 'bigger(X, Z)'and 'isbigger(Z, Y)', with the same variable bindings. Hence, the rule2 gets transformed to:

$$\begin{aligned} isbigger(elephant, monkey) : -bigger(elephant, Z), \\ isbigger(Z, monkey). \end{aligned}$$