Logics for Artificial Intelligence- AI Master

Sessions 5 and 6 – Logic Programming

On the first part of the course we studied the use of First-Order Logic as a mechanism for representing and validating arguments. The language of first-order logic was quite expressive, including the use of constants, variables, functions, predicates, logical operators and quantifiers. *Resolution* is a method that may be used to validate an argument (if we can reach the empty clause from the premises and the negation of the conclusion) or to show that an argument is invalid (if we check all the possibilities of applying resolution to the premises and the negation of the conclusion and it is impossible to obtain the empty clause). However, in some cases we might get into an infinite loop and not be able to show if an argument is valid or invalid. Finally, *model theory* may be used to show that an argument is invalid, by providing a counterexample (a logical interpretation in which the premises are true but the conclusion is false); however, it may not show that an argument is valid. In fact, First-Order Logic is *undecidable*: there isn't any algorithmic procedure that can say, in a finite time, for any argument, whether it is valid or not.

In these sessions on *Logic Programming* we are going to focus on a subset of First-Order Logic. *Logic programs* will be composed of elements, called *clauses*, which can have two forms:

- *Facts*: atomic predicates applied to constants or variables[1].
    - Example: *Father (john, peter). Father (peter, sue).*
- *Rules*: conditional formulas of the form (p1 ^ p2 ^ … ^ pm → q), where p1, p2, …, pm and q are atomic predicates applied to constants or variables.[2]
    - Example*: Father(x,y) ^ Father (y,z) → Grandfather (x,z)*.

A logic program will be considered as the set of premises of an argument. After loading a logic program in a Prolog interpreter we can make *queries*, asking if a certain formula (an atomic predicate applied to constants or variables) is a logical consequence of these premises. For instance, with the two facts and the rule given above, the query *Grandfather(john,sue)* would return a positive result, whereas the query *Grandfather(john,peter)* would return a negative result.

This kind of reasoning is particularly important in the construction of intelligent systems because the *Facts* would represent the information that we have about a particular problem (e.g. the personal and clinical data of a new patient) and the *Rules* would represent the knowledge base with general knowledge about the domain (e.g. medical knowledge that allows to infer the clinical state of a patient from his/her clinical data). The kind of reasoning needed to answer queries by a Prolog interpreter is exactly the same that would be needed by the *Inference Engine* of a Knowledge-Based System to obtain the answer of a particular problem (e.g. the disease of the new patient).

---

[1] We are not going to consider the use of functions. Variables are implictly taken to be universally quantified. Facts can't have negations, so they can be seen as *positive* data.

[2] A rule of this kind is usually represented in Prolog as q:-p1,p2, …,pm, where q is the *head* of the rule and p1,p2, …, pm are its *body*.

It is important to notice that both facts and rules are, in fact, *Horn clauses* (disjunctions of positive/negative predicates applied to constants or variables, in which at most one predicate is positive). *Facts* are disjunctions with a single positive predicate. *Rules* are clauses of the form (¬p1 ˅ ¬p2 ˅ … ˅ ¬pm ˅ q), in which the only positive predicate is the last one (the conclusion of the rule). Thus, a logic program (the set of premises) is a set of Horn clauses. As a query is a single positive predicate, its negation is also a Horn clause (e.g. ¬*Grandfather(john,sue)*). The mechanism used by Prolog interpreters to try to show the validity of the argument (i.e. to try to show that the query is a logical consequence of the set of facts and rules) is called *backward chaining*:

```
Procedure BackwardChaining [q1, q2, …, qn]
    If n=0 then Return True
    For each clause c of the logic program do
      If c=[¬p1, ¬p2, …, ¬pm, q1] and BackwardChaining [p1, p2, …, pm, q2, q3, …, qn]
      then Return True
    EndFor
    Return False
```

In this pseudo-code q1, q2 … , qn are the queries to be solved (objectives). They are considered sequentially, starting with q1. For each one, we check all the clauses that permit to obtain the objective *directly* (fact, which is a clause of the form [q1], like a rule with 0 conditions) or *indirectly* (rule, which is a clause of the form [¬p1, ¬p2, …, ¬pm, q1], with m conditions). Each objective that appears as the conclusion of a rule is replaced by a set of subobjectives, which are the premises of the rule (if the objective matches a fact, it is simply dropped). For instance, the query *Grandfather(john, sue)* would be replaced by the subobjectives *Father(john, y)* and *Father(y,sue)*, using the rule shown above. A query is not considered until all the subobjectives derived from the previous query are finished; therefore, this procedure is implementing a *depth-first search*.

In a logic program there may be different rules that have the same conclusion. In this case, the Prolog interpreter considers the rules sequentially, in the same order in which they are written in the program. Notice that the conditions of a rule are introduced in the list of objectives also in the same order in which they appear in the rule. Thus, both the order of the conditions of the rules and the order of the facts and rules in the program are very important in the inference process. In fact, it is easy to see that the Prolog interpreter may enter into infinite loops, even in cases in which the query is actually a logic consequence of the set of facts and rules.

The most important fact is to notice that this inference procedure to answer queries in Logic Programming is exactly the use of resolution (more concretely, **linear resolution restricted to the predicate on the right, starting with the clause coming from the support set** –the negation of the query). For instance, in the example shown above, the move from the objective *Grandfather(john,sue)* to the subobjectives *Father(john,y)* and *Father(y,sue)* corresponds to the resolution between the clause *¬Grandfather(john,sue)* –the support set- with the clause *(¬Father(x,y) ˅ ¬Father(y,z) ˅ Grandfather(x,z))*, with the unifier *{john/x}* and *{sue/z}*. In summary, linear resolution restricted to the predicate on the right, starting with the support set, may be used as an efficient inference mechanism within knowledge-based systems.

The programmer may change the way in which the Prolog interpreter works by using the *cut* operator (represented with the symbol "!"). It can be used within a rule as follows:

q :- p1, p2, ! , p3, p4, p5.

If we want to satisfy the goal q using this rule, we need to prove the five subobjectives p1, p2, p3, p4 and p5, in this order. If p1 or p2 fail, we have to search for other rules that permit to deduce q. However, if p1 and p2 are true and we reach the cut operator, that means that the *only* way to prove q will be to prove p3, p4 and p5. If these three objectives succeed, q will be proven; however, if one of them fails, we will not consider any other way of proving q, and q will fail.

This operator may be useful to implement conditional inference procedures of the form

T(x,y) = If P(x) then Q(x,y) else R(x,y) as follows:

T(x,y) :- P(x), !, Q(x,y).

T(x,y) :- R(x,y).

Prolog also includes a negation operator (*not*), which may be used to check if a certain condition fails. For example, we can write a rule to deduce that someone does not have any child as follows:

NoChildren(x) :- not(Father(x,y)).

Notice that this operator does not correspond to the classical operator of negation (¬). In order to prove *NoChildren(john)* we do not have to prove *¬Father(john,y)* for all possible *y*s. We only have to check that there isn't any value *y* for which *Father(john,y)* succeeds (i.e., we have to check if the query *Father(john,y)* gives any result or fails). For instance, if we have the fact *Father(john,peter)*, then *NoChildren(john)* will fail. This use of negation is called *Negation As Failure*.

Negation leads us to the topic of *non-monotonic reasoning*. Classical reasoning is *monotonic* (if B is a logical consequence of a set of premises A, it will also be a logical consequence of any set of premises that contains A). However, human reasoning is mostly non-monotonic (given further evidence –more premises- we may retract a previous conclusion).

For example, we may represent easily in First-Order Logic the general assertion that birds fly. However, we may think of many situations in which this assertion fails: the bird is too young, or wet, or sick, or wounded, or it has a broken wing, or it is a penguin, etc. In First-Order Logic we could define a predicate *Abnormal(x)* to represent all those situations explicitly, and then we could say that birds that are not abnormal fly. However, in this case, in order to prove that any bird flies, we would have to prove that it is not abnormal, so we would have to explicitly prove that it is not wet, it is not sick, it is not a penguin, etc.

Logic programming provides a much nicer way of dealing with this kind of exceptions, using the negation operator. The idea would be to use rules such as

Flies(x) :- Bird(x), not(Abnormal(x)).

Abnormal(x) :- Penguin(x).

Abnormal(x) :- Wounded (x) .

… #One rule for each kind of "abnormality" to be considered

In this way, in order to apply the general rule, we would only have to show that, with the available data, it is not possible to prove that the bird is "abnormal" in any way.

Complementary material you should study on weeks 6 and 7:

- Textbook by Brachman and Levesque: chapter 5 (especially 5.3.1 Backward Chaining) and chapter 6 (except 6.8).
- Course *Logic, Language and Information 1* (J.Davoren, G.Restall): Chapter 6 - Computer Science: Propositional Prolog (you can skip 6.3). This introduction to Logic Programming only covers the simple propositional case (0-ary predicates, without constants or variables), but it explains nicely how resolution is the underlying inference mechanism.
- Course *Logic, Language and Information 2* (J.Davoren, G.Restall): Chapter 6 – Predicate Logic Programming. Continuation of the previous material that covers the first-order case.
- In the web page http://lpn.swi-prolog.org you can find a good introduction to Prolog. Each chapter contains many examples and exercises, which are directly embedded in a web-based Prolog interpreter (click the wheel on the top right corner of the examples to access the interpreter). If you prefer, you can also download the SWI Prolog interpreter from http://www.swi-prolog.org.