Logic programming

Lecture 23 Tuesday, April 22, 2014

1 Logic programming

Logic programming has its roots in automated theorem proving. Logic programming differs from theorem proving in that logic programming uses the framework of a logic to specify and perform computation. Essentially, a logic program computes values, using mechanisms that are also useful for deduction. Logic programming typically restricts itself to well-behaved fragments of logic.

We can think of logic programs as having two interpretations. In the *declarative interpretation*, a logic program declares *what* is being computed. In the *procedural interpretation*, a logic program program describes *how* a computation takes place. In the declarative interpretation, one can reason about the correctness of a program without needing to think about underlying computation mechanisms; this makes declarative programs easier to understand, and to develop. A lot of the time, once we have developed a declarative program using a logic programming language, we also have an executable specification, that is, a procedural interpretation that tells us *how* to compute what we described. This is one of the appealing features of logic programming. (In practice, executable specifications obtained this way are often inefficient; an understanding of the underlying computational mechanism is needed to make the execution of the declarative program efficient.)

We'll consider some of the concepts of logic programming by considering the programming language Prolog, which was developed in the early 70s, initially as a programming language for natural language processing.

2 Prolog

We start by introducing some terminology and syntax.

Variables	$X, Y, Z \in \mathbf{Var}$
Function symbols	$f, g, h, a, b, c \in \mathbf{FSym}$
Terms	$s, t ::= X \mid A$
Atoms	$A, H ::= f(t_1, \ldots, t_n)$

We assume that each function symbol has a fixed *arity*, that is, the number of arguments associated with it. A *constant* (ranged over by a, b, c) is a function symbol with arity 0. For example, alice and bob are constants. Suppose function symbol child has arity 2; then child(alice, bob) is a term, but child(alice) is not.

Variables are meant to be place-holders for terms. We call a variable-free term a *ground term*. We call a term of the form $f(t_1, \ldots, t_n)$ an *atom*.

We don't assume any specific set of function symbols. Indeed, the choice of function symbols and their arity is part of the logic program. There may be some common function symbols that are useful in many programs, such as a function symbol for integer addition. Function symbols do not have any implicit meaning, even function symbols for things like addition. We'll be defining the "meaning" of function symbols shortly.

2.1 Unification

Variables are place-holders for terms. A *substitution* is a finite map from variables to terms. Much like the type substitutions we saw in Lecture 13, we can define the application of a substitution σ to a term, and define composition of substitutions.

Importantly, the concept of *unification* also applies to term substitutions. To recap: a substitution σ *unifies* terms s and t if $\sigma(s) = \sigma(t)$. Substitution σ is the *most general unifier* of terms s and t if for any substitution σ' that unifies s and t, we have $\sigma' = \sigma \circ \sigma''$ for some σ''. That is, a substitution is the most general unifier of s and t if we can extend it to obtain any other unifier of s and t.

It may be the case that there is no unifier for some pairs of terms. For example, there is no substitution that unifies $\mathsf{f}(X, Y)$ and $\mathsf{g}(\mathsf{alice})$. In that case, we say that unification fails.

We can extend the notion of unification and most general unifiers to sets of term equations $\{s_1 = t_1, \ldots, s_n = t_n\}$.

Substitutions are the result of logic program computations. We'll see this once we start defining logic programs, which we now address.

2.2 Clauses

A *clause* has the following form:

$$f(t_1, \ldots, t_n) \text{ :- } A_1, \ldots, A_m.$$

where f is a function symbol, $t_1, \ldots, t_n$ are terms, and $A_1, \ldots, A_m$ is a sequence of atoms. We call $f(t_1, \ldots, t_n)$ the *head* of the clause, and $A_1, \ldots, A_m$ the *body*. If the body is an empty sequence of atoms, then we write $f(t_1, \ldots, t_n)$.

Intuitively, we can think of a clause as meaning "$f(t_1, \ldots, t_n)$ is true if $A_1, \ldots, A_m$ are all true." We can think of a clause as being the logical formula

$$\forall X_1, \ldots, X_k.\ A_1 \wedge \cdots \wedge A_m \Rightarrow f(t_1, \ldots, t_n)$$

where $X_1, \ldots, X_k$ are the variables that appear in $t_1, \ldots, t_n$ and $A_1, \ldots, A_m$.

A *logic program* is a set of clauses. A program is activated by providing an initial *query*, which is a sequence of atoms. A solution to a query is a substitution σ such that $\sigma(f(t_1, \ldots, t_n))$ is true.

For example, consider the following logic program, a set of simple clauses that define what it means for one list to be the concatenation of two other lists. We use [] as a 0-arity function symbol standing in for the empty list, and $[t_1|t_2]$ as syntactic sugar for $\mathsf{cons}(t_1, t_2)$, where function symbol cons is intended to be a list constructor: t_1 is the head of the list, and t_2 is the rest of the list. Similarly, we use $[t_1, \ldots, t_n]$ as sugar for the list of the n elements $t_1, \ldots, t_n$.

$$\mathsf{append}([], Y, Y).$$
$$\mathsf{append}([H|T], Y, [H|U]) \text{ :- } \mathsf{append}(T, Y, U).$$

The query $\mathsf{append}([\mathsf{alice}], [\mathsf{bob}], [\mathsf{alice}, \mathsf{bob}])$ is true, since, based on our intuition of clauses, we have:

> $\mathsf{append}([\mathsf{alice}], [\mathsf{bob}], [\mathsf{alice}, \mathsf{bob}])$ is true if $\mathsf{append}([], [\mathsf{bob}], [\mathsf{bob}])$ is true.
> $\mathsf{append}([], [\mathsf{bob}], [\mathsf{bob}])$ is true.

The result of the query $\mathsf{append}([\mathsf{alice}], [\mathsf{bob}], [\mathsf{alice}, \mathsf{bob}])$ is the empty substitution.

The query $\mathsf{append}([\mathsf{alice}], [\mathsf{bob}], Z)$ has the (unique) solution of the substitution $[\ Z \mapsto [\mathsf{alice}, \mathsf{bob}]\ ]$.

The query $\mathsf{append}([\mathsf{alice}], Y, [\mathsf{bob}])$ has no solutions.

2.3 Computation

How do we go about finding a solution to a query? That is, what is the procedural interpretation of a logic program?

We start with the empty substitution. Given a query, we look to see what clauses have heads that can be unified with that query. If we find a clause with a head that has the same function symbol, then we try to unify the query with the head of the clause by extending our substitution; if they can be unified, then we regard each atom in the body of the clause as a query that we are trying to satisfy. We repeat, recursively, until either unification fails, or we have satisfied all queries, and thus have a solution (which is the substitution we have built up by successive unifications).

Now, for a given query, there may be many clauses that have a head with the same function symbol. How do we choose which clause to consider? In logic programming, the choice is nondeterministic: you just magically pick one that works.

However, for implementation purposes, we need some way to decide which clause to consider. In Prolog, the order that clauses are declared in is important: clauses are considered in the order that they are declared. Thus, in Prolog, a program is not a *set* of clauses, but rather a *list* of clauses. A depth-first strategy is used for evaluation in Prolog: when unification fails, we *backtrack* to the last choice point (a point in the computation at which there were more than one applicable clauses), and try a different choice; if there are no more choices, then we backtrack to previous choice point. If no choice point is left, then we fail. Backtracking to a choice point requires us to throw away the extensions to the substitution that we made after that choice point.

2.4 Termination

A Prolog program may fail to terminate. The termination of a recursive Prolog program depends on the order of the clauses, and on the order of atoms in clause bodies. For example, consider the following program, that describes edges in a graph, and allows us to find which nodes are reachable from other nodes (i.e., the reflexive, transitive closure of the edge relation).

$$\begin{aligned}&\mathsf{edge}(\mathsf{alice}, \mathsf{bob}).\\ &\mathsf{reach}(X, X).\\ &\mathsf{reach}(X, Z) \text{ :– } \mathsf{edge}(X, Y), \mathsf{reach}(Y, Z).\end{aligned}$$

When executing the query $\mathsf{reach}(\mathsf{alice}, Y)$, Prolog finds 2 solutions, Y mapping to alice and to bob. But if we modify the order of the clauses, as shown below, Prolog fails to terminate (or, depending on the implementation, may halt with a fatal error). (Why is that? Try executing the logic program according to the rules for computation we defined above.)

$$\begin{aligned}&\mathsf{edge}(\mathsf{alice}, \mathsf{bob}).\\ &\mathsf{reach}(X, Z) \text{ :– } \mathsf{reach}(Y, Z), \mathsf{edge}(X, Y).\\ &\mathsf{reach}(X, X).\end{aligned}$$

Thus, the procedural interpretation of a Prolog program can be quite different from its logical interpretation: a Prolog programmer must be aware of *how* the program will execute in order to guarantee termination and efficiency.

3 Datalog

Datalog, like Prolog, is a logic programming language. However, the semantics of Datalog differ from the semantics of Prolog. Syntactically, Datalog is a subset of Prolog.

In a Datalog program, the order of clauses are not important: a Datalog program can be thought of as a set of clauses, rather than a list. Moreover, unlike Prolog queries, Datalog queries (on finite sets) are guaranteed to terminate. Indeed, evaluation of Datalog programs is polynomial in the size of the program. More specifically, it is possible to enumerate all the ground terms implied by a Datalog program (i.e., a set of clauses) in time polynomial in the number of clauses in the program. (The degree of the polynomial is essentially the maximum number of variables that appear in any rule.)

In order to achieve decidability, Datalog is less expressive than Prolog. In particular, Datalog makes the following restrictions.

- Datalog does not allow any terms other than constants as arguments of functions. E.g., whereas $\mathsf{append}(\mathsf{cons}(\mathsf{alice}, []), [])$ is a syntactically valid Prolog term, it is not syntactically valid Datalog.
- Datalog requires that any variable that appears in the head of a clause also appears in the body of the clause. For example, the clause $\mathsf{foo}(X, Y)$:– $\mathsf{bar}(X)$. is not allowed in Datalog, since the variable Y appears in the head of the clause, but not the body.

(There are also some additional restrictions when we allow negation. That is, Datalog restricts how terms can be negated in order to ensure decidability.)