First-Order Logic

Zdravko Markov

March 8, 2005

1 Syntax

Fisrtly, we shall define briefly the language of First-Order Logic (FOL) (or Predicate calculus). The alphabet of this language consists of the following types of symbols: *variables, constants, functions, predicates, logical connectives, quantifiers and punctuation symbols.* Let us denote variables with alphanumerical strings beginning with capitals, constants – with alphanumerical strings beginning with lower case letter (or just numbers). The functions are usually denotes as f, g and h (also indexed), and the predicates – as p, q, r or just simple words as $father$, $mother$, $likes$ etc. As these types of symbols may overlap, the type of a paricular symbol depends on the context where it appears. The logical connectives are: $\wedge$ (*conjunction*), $\vee$ (*disjunction*), $\neg$ (*negation*), $\leftarrow$ or $\rightarrow$ (*implication*) and $\leftrightarrow$ (*equivalence*). The quantifiers are: $\forall$ (*universal*) and $\exists$ +*existential*). The punctuation symbols are: "(", ")" and ",".

A basic element of FOL is called *term*, and is defined as follows:

- a variable is a term;
- a constant is a term;
- if f is a n-argument function ($n \geq 0$) and $t_1, t_2, ..., t_n$ are terms, then $f(t_1, t_2, ..., t_n)$ is a term.

The terms are used to construct *formulas* in the following way:

- if p is an n-argument predicate ($n \geq 0$) and $t_1, t_2, ..., t_n$ are terms, then $p(t_1, t_2, ..., t_n)$ is a formula (called *atomic formula* or just *atom*;)
- if F and G are formulas, then $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$, $F \leftrightarrow G$ are formulas too;
- if F is a formula and X – a variable, then $\forall X F$ and $\exists X F$ are also formulas.

Given the alphabet, the language of FOL consists of all formulas obtained by applying the above rules.

One of the purpose of FOL is to describe the meaning of natural language sentences. For example, having the sentence "For every man there exists a woman that he loves", we may construct the following FOL formula:

$$\forall X \exists Y man(X) \rightarrow woman(Y) \wedge loves(X, Y)$$

Or, "John loves Mary" can be written as a formula (in fact, an atom) without variables (here we use lower case letters for John and Mary, because they are constants):

$$loves(john, mary)$$

Terms/formulas without variables are called *ground* terms/formulas.

If a formula has only universaly quantified variables we may skip the quantifiers. For example, "Every student likes every professor" can be written as:

$$\forall X \forall Y is(X, student) \wedge is(Y, professor) \rightarrow likes(X, Y)$$

and also as:

$$is(X, student) \wedge is(Y, professor) \rightarrow likes(X, Y)$$

Note that the formulas do not have to be always true (as the sentences they represent).

Hereafter we define a subset of FOL that is used in logic programming.

- An atom or its negation is called *literal.*
- If A is an atom, then the literals A and $\neg A$ are called *complementary.*
- A disjunction of literals is called *clause.*
- A clause with no more than one positive literal (atom without negation) is called *Horn clause.*
- A clause with no literals is called empty clause ($\square$) and denotes the logical constant "false".

There is another notation for Horn clauses that is used in *Prolog* (a programming language that uses the syntax and implement the semantics of logic programs). Consider a Horn clause of the following type:

$$A \vee \neg B_1 \vee \neg B_2 \vee ... \vee \neg B_m,$$

where $A, B_1, ..., B_m$ ($m \geq 0$) are atoms. Then using the simple transformation $p \leftarrow q = p \vee \neg q$ we can write down the above clause as an implication:

$$A \leftarrow B_1, B_2, ..., B_m$$

.

In Prolog, instead of $\leftarrow$ we use $:-$. So, the Prolog syntax for this clause is:

$$A : -B_1, B_2, ..., B_m$$

.

Such a clause is called *program clause* (or *rule*), where A is the clause *head*, and $B_1, B_2, ..., B_m$ – the clause *body.* According to the definition of Horn clauses we may have a clause with no positive literals, i.e.

$$: -B_1, B_2, ..., B_m,$$

.

that may be written also as

$$? - B_1, B_2, ..., B_m,$$

.

Such a clause is called *goal.* Also, if $m = 0$, then we get just A, which is another specific form of a Horn clause called *fact.*

A conjunction (or set) of program clauses (rules), facts, or goals is called *logic program.*

2 Substitutions and unification

A set of the type $\theta = \{V_1/t_1, V_2/t_2, ..., V_n/t_n\}$, where V_i are all different variables ($V_i \neq V_j \forall i \neq j$) and t_i – terms ($t_i \neq V_i$, $i = 1, ..., n$), is called *substitution*.

Let t is a term or a clause. Substitution θ is applied to t by replacing each variable V_i that appears in t with t_i. The result of this application is denoted by $t\theta$. $t\theta$ is also called an *instance* of t. The transformation that replaces terms with variables is called *inverse substitution*, denoted by θ^{-1}. For example, let $t_1 = f(a, b, g(a, b))$, $t_2 = f(A, B, g(C, D))$ and $\theta = \{A/a, B/b, C/a, D/b\}$. Then $t_1\theta = t_2$ and $t_2\theta^{-1} = t_1$.

Let t_1 and t_2 be terms. t_1 is *more general* than t_2, denoted $t_1 \geq t_2$ (t_2 is *more specific* than t_1), if there is a substitution θ (inverse substitution θ^{-1}), such that $t_1\theta = t_2$ ($t_2\theta^{-1} = t_1$).

The term generalization relation induces a *lattice* for every term, where the lowemost element is the term itself and the uppermost element is a variable.

A substitution, such that, when applied to two different terms make them identical, is called *unifier*. The process of finding such a substitution is called *unification*. For example, let $t_1 = f(X, b, U)$ and $t_2 = f(a, Y, Z)$. Then $\theta_1 = \{X/a, Y/b, Z/c, U/c\}$ and $\theta_2 = \{X/a, Y/b, Z/U\}$ and both unifiers of t_1 and t_2, because $t_1\theta_1 = t_2\theta_1 = f(a, b, c)$ and $t_1\theta_2 = t_2\theta_2 = f(a, b, U)$. Two thers may have more than one unifier as well as no unifiers at all. If they have at least one unifier, they also must have a *most general unifier* (*mgu*). In the above example t_1 and t_2 have many unifiers, but θ_2 is the most general one, because $f(a, b, U)$ is more general than $f(a, b, c)$ and all terms obtained by applying other unifiers to t_1 and t_2.

An inverse substitution, such that, when applied to two different terms makes them identical, is called *anti-unifier*. In contrast to the unifiers, two terms have always an anti-unifier. In fact, any two terms t_1 and t_2 can be made identical by applying the inverse substitution $\{t_1/X, t_2/X\}$. Consequently, for any two terms, there exists a least general anti-unifier, which in the ML terminology we usually call *least general generalization* (*lgg*).

For example, $f(X, g(a, X), Y, Z) = lgg(f(a, g(a, a), b, c), f(b, g(a, b), a, a)$ and all the other anti-unifiers of these terms are more general than $f(X, g(a, X), Y, Z)$, including the most general one – a variable.

Graphically, all term operations defined above can be shown in a lattice (note that the lower part of this lattice does not always exist).

```
               V
              ...
    anti-unifiers of t1 and t2
              ...
          lgg(t1,t2)
              /\
             /  \
            /    \
           /      \
         t1        t2
           \      /
            \    /
             \  /
              \/
          mgu(t1,t2)
              ...
      unifiers of t1 and t2
              ...
```

3 Semanics of logic programs and Prolog

Let P be a logic program. The set of all ground atoms that can be built by using predicates from P with arguments – functions and constants also from P, is called *Herbrand base* of P, denoted B_P.

Let M is a subset of B_P, and $C = A$ `:-` $B_1, ..., B_n$ $(n \geq 0)$ – a clause from P. M is a *model* of C, if for all ground instances $C\theta$ of C, either $A\theta \in M$ or $\exists B_j, B_j\theta \notin M$. Obviously the empty clause $\Box$ has no model. That is way we usually use the symbol $\Box$ to represent the logic constant "false".

M is a *model of a logic program* P, if M is a model of any clause from P. The intersection of all models of P is called *least Herbrand model*, denoted M_P. The intuition behind the notion of model is to show *when a clause or a logic program is true.* This, of course depends on the context where the clause appears, and this context is represented by its model (a set of ground atoms, i.e. facts).

Let P_1 and P_2 are logic programs (sets of clauses). P_2 is a *logical consequence* of P_1, denoted $P_1 \models P_2$, if every model of P_1 is also a model of P_2.

A logic program P is called *satisfiable* (intuitively, consistent or true), if P has a model. Otherwise P is unsatisfiable (intuitively, inconsistent or false). Obviously, P is unsatisfiable, when $P \models \Box$. Further, the *deduction theorem* says that $P_1 \models P_2$ is equivalent to $P_1 \wedge \neg P_2 \models \Box$.

An important result in logic programming is that the least Herbrand model of a program P is unique and consists of all ground atoms that are logical consequences of P, i.e.

$$M_P = \{A | A \ is \ a \ ground \ atom, P \models A\}$$

.

In particular, this applies to clauses too. We say that a clause C *covers* a ground atom A, if $C \models A$, i.e. A belongs to all models of C.

It is interesting to find out the logical consequences of a logic program P, i.e. *what follows from a logic program.* However, according to the above definition this requires an exhaustive search through all possible models of P, which is computationally very expensive. Fortunately, there is another approach, called *inference rules*, that may be used for this purpose.

An *inference rule* is a procedure I for transforming one formula (program, clause) P into another one Q, denoted $P \vdash_I Q$. A rule I is *correct and complete*, if $P \vdash_I P$ only when $P_1 \models P_2$.

Hereafter we briefly discuss a correct and complete inference rule, called *resolution.* Let C_1 and C_2 be clauses, such that there exist a pair of literals $L_1 \in C_1$ and $L_2 \in C_2$ that can be made complementary by applying a most general unifier μ, i.e. $L_1\mu = \neg L_2\mu$. Then the clause $C = (C_1 \backslash \{L_1\} \cup C_2 \backslash \{L_2\})\mu$ is called *resolvent* of C_1 and C_2. Most importantly, $C_1 \wedge C_2 \models C$.

For example, consider the following two clauses:

$C_1 = grandfather(X, Y) : -parent(X, Z), father(Z, Y).$
$C_2 = parent(A, B) : -father(A, B).$

The resolvent of C_1 and C_2 is:

$C_1 = grandfather(X, Y) : -father(X, Z), father(Z, Y),$

where the literals $\neg parent(X, Z)$ in C_1 and $parent(A, B)$ in C_2 have been made complementary by the substitution $\mu = \{A/X, B/Z\}$.

By using the resolution rule we can check, if an atom A or a conjunction of atoms $A_1, A_2, ..., A_n$ logically follows from a logic program P. This can be done by applying a specific type of the resolution rule, that is implemented in Prolog. After loading the logic program P

in the Prolog database, we can execute queries in the form of $? - A.$ or $? - A_1, A_2, ..., A_n.$ (in fact, goals in the language of logic programming). The Prolog system answers these queries by printing "yes" or "no" along with the substitutions for the variables in the atoms (in case of yes). For example, assume that the following program has been loaded in the database:

```
grandfather(X,Y) :- parent(X,Z), father(Z,Y).
parent(A,B) :- father(A,B).
father(john,bill).
father(bill,ann).
father(bill,mary).
```

Then we may ask Prolog, if $grandfather(john, ann)$ is true:

```
?- grandfather(jihn,ann).
yes
?-
```

Another query may be "Who are the grandchildren of John?", specified by the following goal (by typing ; after the Prolog answer we ask for alternative solutions):

```
?- grandfather(john,X).
X=ann;
X=mary;
no
?-
```