Foundations of Logic Programming

Many of the materials discussed here can be found in a textbook on logic or discrete mathematics. We will run through these materials rather quickly, but will slow down on some of the deeper or conceptually difficult issues.

1. Syntax of Logic Systems

Logic is a language. Like any language, it has syntax, which defines legal logical expressions (well-formed formulas or wff in the literature) that can be constructed from symbols.

Typical syntax includes provision of symbols, called the alphabet of the language:

Constants
Functions
Predicates
Variables
Connectives
Quantifiers
Punctuation symbols

2. Semantics of Logic Systems

2.1 The role of semantics

The reason that we are interested in the semantics of logic is that we want to talk about the relationship between formulas in the following way:

Let *W* be a set of formulas (which is intended to mean the conjunction of all the formulas in it) and *A* be a formula.

$$W \models A$$

means for every way to interpret *W* that makes *W* true (again, the conjunction of all the formulas in *W*), *A* is interpreted true. Informally, this means that no matter how you interpret *W*, whenever *W* is interpreted true, so is *A*.

That is why we say

W entails A
W implies A
A follows from W

A is a logical consequence of W
A is a theorem of W

This is a key reason why we can use logic as a tool for problem solving.

Example. Suppose we have a blocks world, which is represented by Predicates

on(A,B): represent that *A* is on *B*;
above(A,B): represent that *A* is above *B*.

Let *W* consist of the following formulas where a, b, c are constants representing individual blocks.

on(a,b)
on(b,c)

For all *X,Y,* *above(X,Y) <- on(X,Y)*
For all *X,Y,Z* *above(X,Z) <- on(X,Y) & above(Y,Z)*

Does *above(a,c)* follow from *W*? That is, is *above(a,c)* a logical consequence of *W*?

We know the answer is yes. But how do we know this is the right answer. How do we know that *above(c,b)* does not follow from *W*. We must define *logic consequence*.

2.2 Semantics of Propositional Logic

Propositional logic consists of propositions and some connectives. The typical connectives are

& (and), v (or), - (not),
<- (implication, we may write it -> too)

You may recall the truth table used to define these connectives, given propositions *A* and *B* (we will use 1 for true and 0 for false

A	*B*	*A & B*	*A v B*	*-A*	*A <- B*
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	1

An *interpretation* of a formula *D* is a truth value assignment

of the propositions in *D*. E.g.

Let *D* be

a <- (a v b)

The following truth table shows all the possible truth value assignments, and under each assignment, the truth value of
a <- (a v b),
according to the definition of the connectives

a	*b*	*a <- (a v b)*
0	0	1
0	1	0
1	0	1
1	1	1

The notion of assignment can be trivially extended to a (finite) set *W* of formulas, which denotes the conjunction of all formulas in *W*.

From now on, a formula may refer to a set of formulas in all the definitions given below.

How do we establish the relation?

W |= D

given *W* as a collection of formulas and *D* as a formula?

For propositional logic, since there are finitely many possible assignments given a finite set of propositions, we can always use a truth table.

Note that such a table could be very large for a large number of propositions; the number of assignments grows exponentially; for N propositions, this number is *2^N*.

For example, let

W = {a <- (a v b)}
D = a v b

Is that the case that

W |= D ?

The answer is NO, since it is not the case that "whenever *W* is interpreted true so is *D*". It is sufficient to find one assignment that makes *W* true and *D* false.

Assignments

a	*b*	*a <- (a v b)*	*a v b*
0	0	1	0
0	1	0	1
1	0	1	1
1	1	1	1

For the first row above, *W* is true and *D* is false.

As another example, let

W = {a, b <- a}
D = a & b

It is easy to see that

W |= D

holds.

3. Inference Rules of Logic Systems

By using *W |= a*, we can talk about the relations between formulas. But how is such a relation established? For propositional logic we can use a truth table, but for predicate logic (as we will see later), there may be an infinite number of ways to interpret a formula, such a "truth table" would be infinitely large. So the truth table method doesn't work.

We can use an inference system, which consists of a collection of inference rules that we can use to derive new formulas.

For example, modus-ponens is a common inference rule:

Whenever we have *x* and *x -> y*, we derive *y*

$$\frac{x,\ \ x \rightarrow y}{y}$$

A proof is a sequence of wffs: *a1*, *a2*, ..., *an*, such that each *ai* is an inference drawn from *W* or some subset of the prior *aj*'s with *j < i*.

We use

W |- D

to mean that *D* can be derived from *W*. Derivations are determined, exclusively, by inference rules.

Of course, we want our inference system to work correctly and not to miss any logic consequence.

A set of inference rules is ***sound*** if they do not generate a proof from *W* to *D* when it is not true that *W |= D.*

That is, given *W* and suppose *W |- D,* a sound system guarantees that anything proved from *W* is a logic consequence of *W*.

A set of inference rules is ***complete*** if whenever *W |= D,* there is a proof from *W* to *D*.

That is, given *W* and let *W |= D*, a complete system can always find a proof *W |- D.*

4. Predicate Logic

Let's focus on Horn clauses, namely the logic programs consisting of clauses of the form

L1 :- L2, ..., Lm.

This corresponds to a formula in implication.

L1 <- L2 & ...& Lm

where all variables are universally quantified. The question is: what are the logic consequences of such a program?

Let's consider only those logic consequences that are **ground atoms (**atoms without variables).

When we write a logic program, we may write constants, use function symbols, and of course, predicates.

First, the question is: what are the "objects" which we use predicates to relate. Recall a predicate is just a relation. Then, what are possible tuples of objects?

Herbrand Universe: The set of all ground terms

Given a set *C* of constants and a set F of functions (we often call them function symbols, as they are not concrete functions that we see in math; they are just symbols), the *Herbrand universe*, denoted ***H_u***, is defined inductively as:

1) any constant in *C* is in *H_u*;
2) if *f* is an n-ary function in *F*, and *t1, ..., tn* are in *H_u*, then *f(t1,...,tn)* is in *H_u*;
3) nothing else except those constructed from 1) and 2) are in *H_u*.

Example. Consider the following Horn clause program:

```
plus(0,I,I).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
```

Assume 0 is the only constant and s/1 the only function. Clearly, the relation plus/3 is defined over the objects

```
0, s(0), s(s(0)), ....
```

That is what we use to represent natural numbers. We define a predicate plus/3, which is true on some of these objects, e.g. the following atoms logically follow from our program:

```
plus(0,0,0)
plus(0,s(0),s(0))
plus(0,s(s(0)),s(s(0)))
plus(s(0),0,s(0))
......
```

Sometimes, constants may not appear in our program, but in goals. E.g. when we define the predicate append/3

```
append([],L,L).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

the only constant is the empty list []. Constants may appear in a goal, e.g.

```
?- append([a,b,c], [1,2,3], W).
```

So, the objects over which the predicates in our program are defined depend on some constants. This is why sometimes we say explicitly what are the constants in our language, not only those appearing in our program.

Then, what are the objects for append/3?

Well, given a set of constants, append/3 is defined over the set of all lists composed from these constants. E.g. given constants a and b, append/3 is define over the set of objects

[], [a], [b], [[a], a], …

There are infinitely many objects including any list constructed from a and b.

Example.
Suppose we have constant set {a b} and function set {f, g}, both of which are unary functions. Then, H is an infinite set

H = {a, b, f(a), g(a), f(b), g(b), f(f(a)), f(f(b)), f(g(a)), f(g(b)), g(g(a)), g(g(b)), g(f(a)), g(f(b)), ...}

The terms in *H* are *ground terms*; i.e. they contain no variables. Thus, in one sentence, the Herbrand universe is the set of all the ground terms that can be constructed from the given constants and function symbols.

Herbrand Base, denoted H_b:

Given a set of constants C, a set of function symbols F, we have Herbrand universe H_u. Once we have the set of ground terms in H_u as objects, we need to define relations/predicates over this domain. Then, what is the set of all instances of such predicates?

Given a set of predicate symbols Pr, we define

H_b = {p(t1,…,tn) | p is an n-nary predicate symbol in Pr and t1,…,tn are in H_u}

That is, the Herbrand base is the set of all instances generated from applying predicates in Pr to terms in H_u. This is the set of ground atoms that can be interpreted to be true or false. For example, for the plus example,

{plus(X,Y,Z) | any X,Y,Z from the set {0, s(0), s(s(0)), ...}}

For instance

plus(0,0,0)
plus(s(0),0,0)
....

The set of all possible ground atoms is called **the Herbrand base**, which is the set of all predicate symbols applied to all possible tuples of elements from the Herbrand universe.

Example.

Suppose again constant set {a b} and function set {f, g}, both of which are unary functions.

H = {a, b, f(a), g(a), f(b), g(b), f(f(a)), f(f(b)), f(g(a)), f(g(b)), g(g(a)), g(g(b)), g(f(a)), g(f(b)), ...}

Suppose we have two (unary) predicate symbols p and q. Then the Herbrand base, denoted by B, is

B = {p(a), q(a), p(b), q(b), p(f(a)), q(f(a)), ...}

The Herbrand base is the set of all the predicates of interest; the set of all the atomic assertions that we are interested in their truth or falsity.

Given a logic program P, the set of all ground atoms that are logic consequences of P can be iteratively constructed. The construction process terminates for finite Herbrand universe (thus finite Herbrand base). If it is infinite, we can talk about the set “when n is tending to infinity”.

Given a program P, the iterative construction of this set is by the following "loop" to construct the sequence of sets

S0, S1, ... Sn

such that Sn = Sn+1

- to start, S0 = the empty set

- in every step of iteration, suppose we have Si;
  then Si+1 contains all atoms in Si, in addition,
  if there is a ground instance of a clause from P

H :- B1, ..., Bn

  such that {B1, ..., Bn} is a subset of Si, then H is in Si+1.

- stops if at any step n,

Sn = Sn+1

The construction of the set can be summarized informally as follows:

Repeatedly, if the body of an (instance of a) clause is in Si, then Si+1 contains the head; stop when no more atoms can be obtained.

We can define this sequence of computational steps as applying an operator iteratively.

Let S be a set of ground atoms. First, let us define the notion of a *ground instance of a clause.*

Let A <- B1,...,Bn be a clause. If there exists a (ground) substitution u = {X1 <- t1, Xm <- tm}, where Xi are the variables appearing in the clause and ti are elements from the underlying Herbrand universe, then u(A <- B1,...,Bn) is a ground instance of clause A <- B1,...,Bn.

That is, a ground instance of a clause is an instance of the clause obtained by replacing variables by ground terms.

Now we can define what is called the T_P operator:

```
T_P(S) = {a | a <- b1, ..., bn is a ground instance
              of a clause in P, and b1,...,bn are in S}
```

In mathematics, there is a beautiful theorem, called **Knaster-Tarski fixpoint theorem,** which is established decades ago **and** says that if an operator defined on a *complete lattice* is *monotonic*, then its *least fixpoint* exists and can be computed iteratively by applying the operator starting from the least element of the domain.

Without worrying about all the details, a fixpoint S of an operator T is one that satisfies the equation T(S) = S, i.e., applying T to a fixpoint yields the fixpoint itself.

A lattice is a partial order – being complete means the greatest lower bound and least upper bound exists.

The complete lattice in our case is the power set of the Herbrand universe H_u. It is a partial order in terms of the subset relation. The lattice is complete with H_u itself being the superset of any other set in the power set and the empty set being the least element.

Monotone: in our case, T_P being monotone means, if S1 is a subset of S2, then T_P(S1) is a subset of T_P(S2). This property can be proved easily.

In addition, it can be shown that *the least fixpoint of operator T_P for a program P is precisely the set of the ground atoms that are logic consequences of P.*

Conclusion: A pure Prolog program tries to answer queries according to this fixpoint by the inference rule named resolution.

Example. Transitive closure

```
path(X,Y) :- link(X,Y).
path(X,Z) :- link(X,Y),path(Y,Z).

link(a,b).
link(b,d).
link(d,a).
link(d,e).
```

The least fixoint of this program constains all the atoms directly obtained from the link predicate

```
link(a,b)
link(b,d)
link(d,a)
link(d,e)
```

and atoms about path obtained using the first clause

```
path(a,b)
path(b,d)
path(d,a)
path(d,e)
```

and atoms about path obtained using the second clause, possibly repeatedly

```
path(a,d)
path(a,a)
path(b,a)
link(b,b)
path(b,e)
path(d,b)
path(d,d)
path(d,e)
```

According to the fixpoint theorem, this set can be obtained by applying the operator T_P iteratively from the empty set.

In general, the set of ground atoms that follow from our program may be infinite. This is one reason why we use resolution to answer goals other than computing the fixpoint iteratively.

E.g.

```
p(a).
p((f(X)) :- p(X).
```

The set of ground atoms that are logic consequences of the program contains

p(a), p(f(a)), p(f(f(a))), ...

Question: Could you figure out how is the set of all ground atoms that follow from our program

```
plus(0,I,I).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

constructed? Of course, the set is infinite and the process runs forever. But when the iteration tends to infinity, what would be the set of ground atoms that are computed?