## 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

A logic is a language. Like any language, it has syntax. The syntax of a logic system defines the legal logical expressions (well formed formulas or wff) 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 a 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 ⊨ 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 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)** to represent that A is on B;

**above(A,B)** to represent that A is above B.

Let W consists 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), ∨ (or), ¬ (not),

← (implication, we may write it → too)

A truth table can be used to define these connectives, given propositions A and B. We use 1 for *true* and 0 for *false*.

A B | A & B A ∨ 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 ∨ b)

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

a b | a ← (a ∨ 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 formuals 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 2N.

For example, let

W = {a ← (a ∨ b)}

D = a ∨ b

Is it 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.

a b | a ← (a ∨ b) a ∨ b

-----------|----------------------------------------

0 0 | 1 0

0 1 | 0 1

1 0 | 1 1

1 1 | 1 1

In 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

x, x → 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 assume 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

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" to which we can apply predicates.

**The set of all ground terms**

Given a set C of constants and a set F of functions, the Herbrand universe, denoted H, is defined inductively as:

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

This set is called **the Herbrand universe**.

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 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? 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 defined over the set of objects

[], [a], [b], [a,b], [b,a], [[a],a], ....

These are infinitely many objects: 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 called 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.

**The set of all possible ground atoms**

Once we have the set of ground terms as objects, we need to define some relations/predicates over this domain. Then, what is the set of all such predicates? Well, it is precisely what we can say about a relation/predicate over ground terms. For example, for the plus example,

plus(X,Y,Z) for any X,Y,Z taken 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. Again given a constant set {a, b} and set of unary functions {f, g}.

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 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.

**The Least Model**

The set of all ground atoms that are logic consequences of a program P is called the "least model". It is the smallest set of atoms that is consistent with the program.

The least model can be iteratively constructed. The construction process terminates for finite Herbrand universe (thus finite Herbrand base). 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.

* S0 = the empty set
* To compute Si+1 from Si:   
  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. (We add the head of a clause if we have all the atoms on the right side already.)
* Stop if at any step n, Sn = Sn+1 (we reached a fixed-point, nothing more found to add)

The construction of the set can be summarized informally as follows: Repeatedly, if the body of a clause is obtained then include the head; stop when no more atoms can be obtained.

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 model of this program contains 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)

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

Example:

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: 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 is nonterminating.

a b | a ← (a ∨ b) | a v b

------|-------------------------------

0 0 | 1 0

0 1 | 0 1

1 0 | 1 1

1 1 | 1 1

W = {a ← (a ∨ b)}

D = a ∨ b

Does W imply D?

"Whenever W is true, then D is also true"

No.

In line 1, W is true but D is false.

Example 2:

W = {a ← (a ∨ b), a}

Does W imply D?

"Whenever W is true, then D is also true"

W is true in lines 3 and 4.

In both lines 3 and 4, D is also true.

IN this example, W implies D.

W = {a & b & c}

D = b v c

Does W imply D?

a b c b v c

1 1 1 1

W = {a & b}

D = b v c

a b c b v c

1 1 1 1

1 1 0 1

W = {a & b}

D = b & c

a b c b & c

1 1 1 1

1 1 0 0

Example from notes:

W = {a, b ← a}

D = a & b

a b | b ← a | a & b

------|-------------------------------

0 0 | 1 0

0 1 | 1 0

1 0 | 0 0

1 1 | 1 1

a true:

1 0 | 0 0

1 1 | 1 1

b ← a true:

0 0 | 1 0

0 1 | 1 0

1 1 | 1 1

both a and b ← a true:

1 1 | 1 1

Yes, W implies D.

Herbrand universe

constants: C = { 0 }

F = {s} .. unary function

Rule 1: 0 is in H H = { 0}

Rule 2: s(0) is in H, H = {0, s(0)}

Rule 2: s(s(0)) is in H, H = {0, s(0), s(s(0))}

etc.

Example:

constants: { [], a, b}

function: 2-ary function .(a,[]) as in Lisp, but written in usual Prolog list notation, [a].

H = { a, b, [],

[[]], [a], [b], ..., [a,b], ..., [[[a],[[b,[]]]],...

Example:

constants C = { a,b }

functions F = {f, g}

f unary

g 2-ary

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

H includes any arbitrarily nested combination of f,g,a,b you can put together that respects the syntax.