First Order Logic

Notes on First Order Predicate Logic

I. Introduction

First order predicate logic (FOL or FOPL) is a language for precisely specifying properties of a domain of interest. As a language, it has a syntax and a semantics. This document gives a quick overview of the syntax, describes several abbreviation mechanisms, talks (informally) about how to interprete FOL sentences, and gives an overview of the structure of proofs in general and program proofs in particular.

II. Propositional Logic

FOPL is built on top of propositional logic by adding quantification. Propositional logic is the logic dealing with the composition of sentences or propositions using connectives such as "and", "or", "not", and "implies". It is also called combinatorial logic. It is not an adequate logical language for making sophisticated inferences because it has no mechanism for quantification. This is what predicate logic adds.

A proposition may be denoted by a name (alphanumeric identifier) or a formula. A propositional formula is one of the following (where P and Q are themselves propositional formulas):

```
+ a name
+ (! P)          // Not P
+ (| P Q)        // P or Q
+ (& P Q)        // P and Q
+ (=> P Q)       // if P then Q or P implies Q
+ (<=> P Q)      // P if and only if Q; P is equivalent to Q
```

Propositional formulas can be interpreted (given meaning) in terms of their truth (T) or falsity (F). To do so requires that each proposition must itself first have a truth value.

The following table can be used to interpret the value of an expression, given the values of its constituents.

P	Q	(| P Q)	(& P Q)	(! P)	(=> P Q)	(<=> P Q)
T	T	T	T	F	T	T
T	F	T	F	F	F	F
F	T	T	F	T	T	F
F	F	F	F	T	T	T

III. FOPL

FOPL adds to propositional logic quantification over logic variables. Each quantification adds a new variable whose meaning obtains over the scope of the quantification. Quantifications themselves are propositions that can be combined with other propositions using the operations of propositional logic, or they may be nested inside of other quantifications.

IV. Syntax of FOPL

1. Sentences (also called propositions) may be simple or complex. Simple sentences include predicates and named references to sentences defined elsewhere. In the latter case, uppercase letters from the middle of the alphabet are normally used (e.g. 'P', 'Q', ...).

2. Predicates (sometimes called assertions) have a name and a list of terms. The term list is placed in parentheses after the predicate name. Terms are separated by commas in the list.

    Eg. married(Bill, Jane)

3. Terms are names denoting either variables or constants. Variables are usually denoted by lower case letters from the end of the alphabet (like 'x', 'y', or 'z');

4. Complex sentences are unary, binary, or quantified.

5. A unary sentence consists of '!' followed by a sentence.

    Eg. (! hot(today))

6. A binary sentence composes two other sentences using one of four infix operators ('&', '|', '=>', or <=>). The sentence itself is normally placed in parentheses to avoid precedence problems.

7. A quantified sentence has three parts: a quantifier, a variable, and a sentence. There are two quantifiers: ('all' or universal quantification and 'exists' or existential quantification). The variable and the sentence are separated by a comma and placed in parentheses.

    Eg. all(x, older(x, 50))

VI. Syntactic Sugar

1. all(x, all(y, bar)) may be abbreviated as all((x, y), bar).
   Likewise for exists. 'x' must be lexically different than 'y'.

2. Parentheses may be dropped when no ambiguity arises.

3. Definitions for sentences can be made using '=='. For example
   P == older(tom, harry) & happier(harry, tom). The definitions
   can be parameterized with variable names. For example,
   Q(x, y) == all(z, (older(x, z) & older(z, y)) =>
   older(x, y)).

4. P(V % E) where 'V' is a name and 'E' is a sentence is a syntactic
   transliteration (substitution) of P with every unbound occurrence
   of the name 'V' replaced by the expression given by 'E'. Think
   of this as a way of making systematic changes to the names used
   in sentences.

VII. Interpretation

1. Predicates are functions producing truth values ('T' or 'F').

2. The unary and binary operators have their meaning given in the usual
   way (propositional logic or truth tables). That is, '!' is 'not';
   '&' is 'and'; '|' is 'or'; '=>' is 'implies'; and '<=>' is
   'equivalence' or 'if and only if'.

3. Quantifiers specify a scope within which the meaning of the
   quantified variable is defined and fixed. By the syntactic rules
   given above, quantified scopes can be nested. As long as the
   variable names being quantified are different, a variable's scope
   includes all of the quantified sentence. If an inner scope
   quantifies a variable with the same name as an outer one, however,
   the outer binding does not hold within the inner scope. It is
   reinstated when the inner scope is ended.

4. Type checking predicates are normally elided.

VIII. Proofs

1. A proof is a series of sentences. The sentences are usually
   numbered. For each sentence, a reason must be given. There are
   four kinds of reasons.

2. The first kind of reason is an ASSUMPTION. That is, the sentence is
   assumed to be true for the duration of the proof without further
   justification. Usually, the assumptions are given at the head of
   the proof, but they may be repeated internally when readability is
   improved.

3. The second kind of reason has to do with the laws of BOOLEAN ALGEBRA. That is, we know certain rules for manipulating truth values, and this often allows us to make certain simplifications, like 'P & T' allow us to conclude 'P'.

4. The third kind of reason has to do with LOGIC itself. This kind of reason is called a "rule of inference". Here are some examples of rules of inference.

   a. all-elimination: all(x, P(x)) allows you to conclude P(a) for a specific constant or variable 'a'. The term "allows you to conclude" can be written in two other ways: with '|-' or by placing the sentence that comes before it on a line above the sentence that comes after it.

```
all( x, P( x ) )
----------------
      P( a )
```

   What this rule of inference says is that if a sentence is true for all x, then it is true for a specific one named a.

   b. all-introduction:
   P(a) for an arbitrary a -> all(x, P(x)).
   That is, if you have made no assumptions about a, then a has really served the role of a universally quantified variable.

   c. exists-elimination:
   exists(x, P(x)), all(y, P(y) => Q) |- Q.

   Note that in the case where more than one sentence preceeds the '->', the sentences are conventionally separated by commas instead of 'and's.

   d. exists-introduction: P(a) |- exists(x, P(x)).

   e. modus ponens: P & (P => Q) |- Q.

   f. There are lots more of these.

5. The fourth kind of reason has to do with the particular domain of values about which the sentences are making assertions. For example, if we are talking about numbers, we can draw certain simplifying conclusions. For example, greater(plus(x, 0), y) |- greater(x, y). Furthermore, for the domain of numbers, we will allow further syntactic simplifications, like giving the above with (x + 0) > y |- x > y.

IX. Program Proofs

1. For the particular domain of programs, we have basic rules for statements. Note that we will place sentences within '{' and '}' in order to distinguish them from program text. Also, the rules assume that the described statement terminates.

   a. The assignment statement: {P(V % E)} V := E; {P(V)}.

   b. Two consecutive statements: ({P} S {Q}), ({Q} T {R}) |- {P} S; T {R}.

   c. Conditional statement: ({P & B} S1 {Q}), ({P & !B} S2 {Q}) |- {P} if B then S1 else S2 end if {Q}.

   d. While loop statement: ({P & B} S {P}) |- {P} while B do S end while {P & !B}.

   Most other programming language control statements can be defined in terms of these four rules.

2. Programs normally have 'preconditions'. A precondition is an assertion describing the allowable circumstances for valid program execution.

3. Likewise, programs have 'postconditions', assertions describing the state of the program variables after the program has finished executing. Postconditions assume that the program terminates. A program's meaning can be given by the relationship between its preconditions and its postconditions.

4. One other special assertion is called a 'loop invariant'. This is an assertion that holds true at some point within the scope of a loop (e.g. while) construct. Note that it must hold true both for the first loop execution and for each subsequent one. There is a intentional similarity here to the property of induction that holds for assertions about well-ordered sets like the integers.

5. Program termination is normally proven separately from program correctness. The only one of the program constructs that might not terminate is the while loop, for which the Boolean condition may never become false. To prove program termination, it is sufficient to show that each constituent loop terminates.

6. Loop termination is proven by using the following property of the integers. Consider a sequence of integers such that each element of the sequence is strictly greater than than its successor. If the sequence is long enough, it must eventually become negative. For example, consider a sequence of numbers starting from 100 where each element is three less than its predecessor (100, 97, 94, ...). Then if the sequence goes on long enough, it must eventually contain a negative number (4, 1, -2, ...). Of course, once it goes negative, it will stay negative forever.

7. This property can be used in proofs of termination. Consider the following loop skeleton.

```
I := 0;
while (I < N) {
  S1;          // ********
  S2;
  .
  .
  .
  Sn;
  I := I + 1;
}
```

Consider also the function f(I, N) == N - I at the point just above the line marked with '*'s. If this function were computed on each loop iteration, a sequence of values would be produced (N, N - 1, ... 2, 1, ...). The sequence has exactly the properties required to show that it must eventually go negative. That is, each time through the loop, N - I is smaller than the time before.

8. Therefore, to show termination for a loop, you must do the following. Devise a function whose arguments are (integer valued) program variables. The function must have the following properties.

a. Regardless of the path taken through the loop, the function's value must decrease on each iteration.

b. The loop termination condition must be such that the loop cannot execute if the function's value is negative. In the example of the loop skeleton given above, the loop cannot execute if I >= N. This is equivalent to 0 >= N - I which is the definition of f(I, N).

If you can devise such a function and demonstrate the two properties, then you have proven that the loop must terminate.

9. In general, termination for an entire program is proven from the inside out. That is, the innermost loop is proven to terminate first. Then the loop which most closely contains it, and so on until the outermost loop.