CASL Single Lecture Notes

SLN - AIB - 03

2006-12-05

Cognitive Autonomous Systems Laboratory
Department of Computer and Information Science
Linköping university, Sweden

Erik Sandewall

Introduction to Logic for AI

Subject Area: Artificial Intelligence Basic

Date of lecture: 2006-11-15

"Single Lecture Notes" are notes corresponding to one lecture.

Related information can be obtained via the following WWW pages:

Linköping university:	http://www.liu.se/
This course:	http://www.ida.liu.se/~TDDA23/
CASL Group:	http://www.ida.liu.se/ext/casl/
The author:	http://www.ida.liu.se/~erisa/

1 The concept of logic

The place of Logic in Science

Knowledge representation, which is considered by many to be the central area of artificial intelligence, combines two kinds of considerations: computational performance and expressivity of notation. Formal logic is the basic tool for addressing the expressivity aspect. In practical work one often has to make compromise between what is the best approach from a purely logical point of view and what is best computationally. However it is important to always have the logic aspect in mind as a frame of reference.

Logic is the study of methods for drawing correct conclusions from available information. *Formal logic* is the study of Logic using formal languages, that is, formulas whose structure is precisely defined.

Formal logic has a short history as a science. (One often hears that logic is very old because it was studied already by the ancient Greeks, but this is not accurate - the logic expounded by Aristotle, for example, was very simple by the standards of contemporary formal logic). It has emerged during the 20th century, partly inside mathematics and partly in theoretical philosophy.

Computer science is the only scientific discipline where logic is applied to any significant extent. Uses of logic in computer science fall in two main areas:

1. In the study of programming languages and real-time systems
2. In the study of knowledge representation (in AI) and for the theory of databases.

There is relatively little exchange of methods and results between these two areas.

Major types of logic languages

The word 'language' is used in logic in a similar way as for programming languages: a language *is* the set of formulas that conform to a particular definition of syntax. Usually these sets are infinite since formulas can be arbitrarily large.

The following are the major types of logic languages.

Propositional logic

In propositional logic formulas are constructed using the operators and ($\wedge$), or ($\vee$), implies ($\rightarrow$), not ($\neg$), and sometimes others. The elements of the formulas are called *proposition symbols.* When one studies logic as such one usually writes them as single letters; in practical use for knowledge representation one often uses entire 'identifiers' for proposition symbols, like in programming languages.

In this series of memos, logic formulas will sometimes be written with typewriter font (for convenience reasons for the author), and in these cases we shall use `&`, `v`,`->`, and `-`, respectively for the symbols mentioned above.

The word 'logic' is often used in engineering (for example in circuit design) to refer to propositional logic. In a computer science context one must be aware that propositional logic is only a very simple special case of formal logic.

In the following I assume that the reader is generally familiar with propositional logic, and in particular that she or he knows the truth-table definitions for the operators of propositional logic.

First-order predicate logic

The following is an example of a formula in first-order predicate logic (often one just says 'first-order logic' or 'predicate logic'):

$$\forall x \forall y[x = father(y) \rightarrow Older(x, y)]$$

Presumably this formula is intended to mean "if x is the father of y then x is older than y". In general, a formula in predicate logic is obtained by composing elements, called *atomic propositions* using the same operators as in propositional logic, plus some more, namely the *quantifiers* 'for all' ($\forall$) and 'there exists' ($\exists$). The atomic formulas may be proposition symbols, but they may also be composite expressions.

In the example, the atomic proposition $Older(x, y)$ is formed using the *predicate Older* with two arguments which are formed as the *variables* x and y. The atomic proposition $x = father(y)$ is formed using the predicate $=$ (written as an infix between its two arguments) where the first argument is again a variable, and the second argument, $father(y)$ is called a *term* and is constructed using the function $father$ and the two argument x.

In general, a formula in first-order logic is built up in three layers. In the lowest layer, terms can be nested inside terms, since each function takes terms as arguments, and the resulting expression is a new term. In the middle layer, atomic formulas are formed using predicate symbols with terms as arguments. In the third and topmost layer, *propositions* are formed from atomic formulas which are composed, again to any desired depth, using quantifiers and the propositional connectives (and, or, etc).

The separation between these layers must be respected. The arguments of functions must be terms, they can not be e.g. atomic propositions, non-atomic propositions, or predicate symbols, for example. Similarly, terms can not be used directly as arguments of the propositional connectives when propositions are formed, they can only be used as arguments of functions and predicates.

Some of these constructs have several names, according to the preferences of the author. For example, propositions are often called 'well-formed formulas'.

Second-order predicate logic

The difference between first-order and second-order predicate logic is that in the latter it is permitted to let functions or predicates be arguments of functions and predicates. This allows one to write, for example, the definition of the transitivity property for a binary predicate as follows:

$$\forall R[Transitive(R) \rightarrow \forall x \forall y \forall z[R(x,y) \wedge R(y,z) \rightarrow R(x,z)]]$$

This greater expressive freedom still has some restrictions, however. One has to designate some functions and predicates as first order and others as second order. A second-order function can have first-order functions and predicates as arguments, but it can not have itself or other second-order functions as arguments. A first-order function is like the functions in first-order logic: it can only have variables and first-order terms as arguments. In the example $Transitive$ is a second-order predicate and R is a variable whose values shall be first-order predicates.

As a second example, consider the definition of the limit value of a function f as its single argument x approaches a certain value c from below. In second-order logic we might write this as $leftlimit(f,a,c)$ where f is a first-order function and $leftlimit$ is a second-order predicate. The definition of the left limit value can be expressed as:

$$\forall f \forall a \forall c[leftlimit(f,a,c) \leftrightarrow$$
$$\forall \delta \exists \epsilon \forall \epsilon'[\epsilon > 0 \wedge (0 < \epsilon' < \epsilon \rightarrow | f(x-\epsilon') - c| < \delta)]]$$

This expression uses numerical functions such as - and | . | and numerical predicates such as $<$.

Modal logic

Modal logic, as an extension of first-order predicate logic, allows one to write expressions such as

$$Believes(john, Older(mary, peter))$$

where *Older* is a predicate, as before, and *Believes* is a modal operator. Thus modal operators can take entire propositions as arguments, which is not allowed in the previously mentioned cases. (Notice that in second-order logic it is possible to take the first-order predicate *Older* as an argument for a function or a predicate, but not a proposition formed using *Older* as its leading predicate).

There is a difference between functions and predicates on one hand and modal operators on the other: For every application one can choose one's own vocabulary of functions and predicates and describe their properties using axioms. As more of them are added one just gets another usage of first-order logic, or more precisely one obtains *another first-order logic*. The introduction of modal operators is quite another thing and much more complex, and having more than one modal operator in an application causes more complexity.

2 Drawing Conclusions

There are three ways of analyzing and working with propositions in formal logic, namely by *rewriting* and by using the *proof-oriented* and the *semantics-oriented* perspectives.

Rewriting

Rewriting is what happens when a proposition in formal logic is replaced by another, equivalent proposition. Informally speaking, two propositions are equivalent if they have the same meaning. If A and B are equivalent we write

$$A \Leftrightarrow B$$

Notice that the expression $A \Leftrightarrow B$ is not a proposition in logic: it expresses *a relation between two formulas, two propositions*, and it is not itself a proposition.

Most rewriting rules for propositional and predicate logic are easily understood once one knows (a) the truth-tables for the propositional operators and (b) the meaning of the quantifiers. The following are some examples of equivalence rules for the propositional operators:

$$A \wedge B \Leftrightarrow B \wedge A$$

$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$$

$$A \rightarrow B \Leftrightarrow \neg A \vee B$$

The following are some equivalence rules for quantifiers:

$$\forall x[A \wedge B] \Leftrightarrow \forall x[A] \wedge \forall x[B]$$

$$\neg \forall x[A] \Leftrightarrow \exists x[\neg A]$$

The full set of such rewrite rules is learnt in a logic course. For the purpose of the present course we will not teach these rules systematically. A few rules for quantifiers will be needed later on in the course, and then we shall introduce the rules that we need.

Proofs

In the proof-oriented perspective one in concerned about *inference rules* that specify how one can take some propositions and *infer* other propositions from them, that is, to draw conclusions. The following is an example of an inference rule in one variety of propositional calculus:

```
  A & B
------------
  A
```

The following is another one:

```
  A
  B
------------
```

```
    A & B
```

In general, inference rules are used in *proofs* where each proof is a sequence of formulas that are usually written on successive lines. One starts with a number of *axioms*, that is, propositions that one wishes to take for granted, each axiom being written on a separate line. After the axioms, each successive line in the proof must be motivated by an inference rule and some earlier lines. For example, the second inference rule above says that if you have two formulas, A and B, on some earlier lines (in whichever order) then one is entitled to write A & B on a later line.

The two rules shown above plus the following rule

```
    A & B
 ------------
    B
```

are sufficient for drawing the conclusion B & A from the axiom A & B, as follows:

```
    A & B          (axiom)
    A
    B
    B & A
```

Practical proofs are of course much more complex but this shows the general idea.

The following notation is used. If Γ is a set of propositions (the axioms) and B is a consequence from Γ according to the inference rules (the chosen set of inference rules, if one has several to choose between) then one writes

$$\Gamma \vdash B$$

The is pronounced "Gamma leads to B", or "Gamma leder till B" in Swedish.

Introduction to semantics: Models for equations

In the *semantics* oriented perspective, finally, one asks first of all what is the *interpretation* of propositions in the logic at hand, and then one defines what are valid conclusions from given axioms in terms of those interpretations.

We shall introduce the basic concepts of semantics by making a comparison with something well-known, namely algebraic equations. The following equation

$$x + 4 = 13$$

determines one value for x that is a solution for the equation, but the following one

$$x^2 + 4 = 13$$

allows two solutions. One way of understanding this is to think of the equality operator as a function that has the value T or F depending on whether the two arguments are equal or not. Then a solution for the equation is an assignment of values to the variable(s) in the equation whereby the value of the entire equation is T.

The advantage with looking at things this way is that it makes it possible to generalize to the use of formulas in predicate logic. Consider for example the following one:

$$x + 2 = 7 \lor x - 3 = 6$$

It is easily verified that if x is 5 or it is 9 then the value of the entire expression is T, and for all other cases it is F, since the value of an or-expression is T when at least one of its arguments is T. In this way one can define the concept of solutions for propositions in logic.

On this basis we can proceed to how inference is made. Suppose that we know that

$$x^2 + 4 = 13$$

but we do not know the exact value of x. Anyway we can certainly draw the conclusion that x is less than 10:

$$x < 10$$

Here is a simple example of a conclusion: we do not know everything about x, but we know enough about it to be able to say that it is less than 10. How can we motivate drawing that conclusion? Along the line of proofs, as briefly described above, we would expect to have a number of inference rules whereby we can go from the axiom ($x^2 + 4 = 13$) to the conclusion ($x < 10$). However this would hardly be a natural way of doing things.

Instead, consider what are the solutions of the desired conclusion,

$$x < 10$$

In other words, what are the values for x whereby this formula obtains the value T? Clearly this is the set of all numbers from 9 and down towards minus infinity. What is significant is that this set has the set of solutions for $x^2 + 4 = 13$ as a subset.

These notions apply just as easily when a solution is an assignment of values to several variables, instead of just one. They also apply for formulas in propositional calculus, where each proposition contains a number of proposition symbols. In this case they shall of course be assigned the value T or F, rather than numbers as in our first examples.

The following notation is used. If A is a proposition then $Mod(A)$ is the set of solutions for A. The symbol $\models$ is defined by

$$A \models B \quad \text{if and only if} \quad Mod(A) \subseteq Mod(B)$$

which means that from A one can draw the conclusion B. For example, in the example above we have

$$x^2 + 4 = 13 \models x < 10$$

If we have a set of propositions, for example a set of equations, then of course the solutions for the entire set are those solutions that are true in all the propositions, that is, the intersection of the solution sets for the individual propositions. Thus we have

$$Mod(\{A, B\}) = Mod(A) \cap Mod(B)$$

and similarly for a set of more than two propositions. If Γ is a set of propositions then we write

$$Mod(\Gamma) \models Mod(B)$$

when B is a consequence of Γ. This is pronounced "Gamma entails B", or "Gamma innebär B" in Swedish.

Models for first-order logic

The same concepts as have now been explained for equations apply to predicate logic as well. However in this case we have to generalize the concept of 'solution' and in logic one instead uses the following two terms. An *interpretation* is a structure that assigns appropriate values to the symbols in a proposition (the function symbols and predicate symbols, in particular) and a *model* is an interpretation for which the proposition has the value T.

By analogy, the value assignment $\{x : 4, y : 7\}$ is an interpretation for the equation $x + y = 9$ since it assigns values to both the variables that occur in the equation, but it is not a model since the value of the equation with this assignment is F and not T.

An interpretation for first-order predicate logic consists essentially of three parts: A non-empty set of objects called the *domain*, assignments of functions to the function symbols, and assignments of predicates (functions whose value is T or F) to the predicate symbols in the particular first-order logic in question.

For example, for the formula

$$\forall x \forall y [x = father(y) \rightarrow Older(x, y)]$$

the following is an interpretation:

```
Domain: the objects P1, P2, P3, and NIL

Functions: the function symbol f is assigned the function
   { <P1>:NIL, <P2>:P1, <P3>:P2, <NIL>:NIL }

Predicates: the predicate symbol Older is assigned
   { <P1,P1>:F, <P2,P2>:F, <P3,P3>:F,
     <P1,P2>:T, <P1,P3>:T, <P2,P3>:T,
     <P2,P1>:F, <P3,P1>:F, <P3,P2>:F
       ... }
and the predicate symbol = is assigned
   { <P1,P1>:T, <P2,P2>:T, <P3,P3>:T,
     <P1,P2>:F, <P1,P3>:F, <P2,P3>:F,
     <P2,P1>:F, <P3,P1>:F, <P3,P2>:F,
       ... }
```

where the ... indicate that the seven argument combinations involving NIL must also be assigned values. The idea here is that you have three intended persons, namely P1, P2, and P3, where P1 is the father of P2 and P2 is the father of P3, but since every function must have a value for every combination of arguments it becomes necessary to introduce an extra object NIL that represents the father of P1 that remains unspecified for the purpose of the example.

In general, then, for every function symbol the interpretation must specify a function that maps sequences of arguments with the right number of elements to corresponding values. Each argument as well as the value must be members of the domain in the interpretation, and the value must also be so. Similarly for predicates, except that the value is a truth-value.

For every combination of a proposition and an interpretation for that proposition it is then straight-forward to define the value of the proposition in the interpretation. As already said, a model is an interpretation where the proposition has the value T, and the model set $Mod(\Gamma)$ for a set of propositions is the set of interpretations where all the propositions in Γ are true.

The generalization from solutions to equations or sets of equations, to propositions in predicate logic, is quite similar to the generalization from equations in high-school algebra to those in matrix algebra or in differential equations. In all cases one allows the solutions, or models, to be structured objects. In logic they are structured as the interpretations that have just been defined; in matrix algebra they are vectors or matrices, and in differential equations they are functions of some variable, for example time.

Relationship between the three perspectives

We have now seen three ways of analyzing and working with propositions: rewriting using equivalence rules, proofs, and semantics. They have important relationships. To begin with, it is straightforward to specify equivalence rules in terms of semantics: one shall have $A \Leftrightarrow B$ iff and only if $Mod(A) = Mod(B)$. This is easily seen if one thinks of A as an equation so that $Mod(A)$ is the set of solutions for A.

What about the relation between proofs and semantics? We have seen two ways of specifying conclusions: "leads to" ($\vdash$) and "entails" ($\models$). Different scholars have different opinions about which of these is the most fundamental one: some think that inference rules shall come first, others think that semantics shall come first. In any case, however, there is an important theorem called *completeness* for first-order predicate logic:

$$\Gamma \vdash B \quad \text{if and only if} \quad \Gamma \models B$$

For the purpose of knowledge representation and A I, this is used as follows. Both equivalence rules and proofs are important computationally, because they are implemented in actual programs for theorem-proving and other computations using propositions in formal logic. Semantics and models are important in the design stage for a knowledge-based system, since they are used for defining and analyzing the formalism that one plans to use for the actual system. The fact that "leads to" and "entails" are equivalent is the key for applying the results of that analysis to the actual system.

3 The Resolution Method

Resolution is an inference method that is extensively used in theorem-proving programs. It is also the basis for logic programming, which is a technique that is intermediate between a programming language and an inference-drawing software system.

Resolution in propositional logic

We first describe resolution for propositional logic and then generalize to first-order predicate logic.

One basic idea in resolution is to reduce the possibilities of equivalence. In ordinary propositional and predicate logic, for every given formula there is a considerable number of other formulas that are equivalent with the given one. This tends to reduce the efficiency of theorem-provers since a lot of work is used for conversions between equivalent formulas, and many of those conversions may be unnecessary.

Resolution uses a combination of two techniques for reducing this redundancy. The first technique is to require that given axioms as well as their conclusions shall always be written on *clause form*, that is, each axiom shall be a disjunction (an or-expression) between literals. (Recall that a literal is a proposition symbol or its negation). The following is an example of a clause:

$$a \vee \neg b \vee c$$

It is always possible to rewrite a given set of axioms on clause form, although sometimes it may be necessary to split up a given axiom into several clauses. For example, even $a \wedge b$ is split up into two clauses: a is one clause and b is another clause. (A clause may be disjunction of just one single literal).

The second technique that is used in resolution is to consider a clause as a *set* of literals, instead of as a single formula where the literals are connected by or-signs. When this technique is applied, the clauses $a \vee b$ and $b \vee a$ are considered as *the same* clause. A theorem-proving program therefore does not need to make conversions between clauses such as those as steps of inference. Instead, the treatment of sets of clauses is pushed down to the implementation level, where it can be dealt with e.g. by always ordering the literals 'alphabetically' in all clauses.

From the perspective of traditional logic it is a bit odd that a formula is a set of things, instead of a sequence of symbols as one usually thinks. However this generalizaton is quite OK, besides helping efficiency.

Given that every axiom and every conclusion is a clause, resolution uses one single inference rule, called the resolution rule. The following is an example of how it is used:

```
    a v b v d
    e v f v -d
  --------------
    a v b v e v f
```

(We shall write the clauses with the or-signs still present in order to facilitate understanding, but please keep in mind that the order of the literals in a clause is irrelevant).

Here is another case, representing the rule that if one knows a and $a \rightarrow b$ then one can conclude b:

```
    a
    -a v b
  --------------
    b
```

In general, two clauses can be resolved if there is some literal that occurs in both of them, but with opposite 'sign': it is negated in one of the clauses and un-negated in the other one. The validity of this rule is easily understood. In the first example, for example, either d must be true or it must be false. If it is true, then $\neg d$ is false so $e \vee f$ must be true. If it is false, then $a \vee b$ must be true. Therefore, regardless of d, the proposition $a \vee b \vee e \vee f$ must be true.

Resolution in predicate logic

In predicate logic, clause form is defined so that every clause must be a disjunction of literals, but in addition it can only have universal quantifiers ($\forall$) and it can not have any existential quantifiers ($\exists$). These universal quantifiers must furthermore be placed outermost in the clause, and not in sub-expressions inside it.

Since all variables are universally quantified anyway, one usually does not write out the quantifiers at the beginning of the clauses. All variables that occur in a clause are presumed to be universally quantified.

The rewriting of a given proposition in first-order logic to clause form sometimes requires one to introduce a new function for replacing an existential quantifier. For example,

$$\forall x \exists y [P(x, y)]$$

is rewritten as

$$\forall x [P(x, f(x))]$$

where f is a function symbol that is not previously used in the axioms. The intuitive meaning is that $f(x)$ shall be one of those objects that is stated to exist, for the given x, according to the first version of the axiom. Functions that are introduced in this way are called *Skolem functions*. Some additional details about the properties of this transformation is at the end of this memo.

In the case of predicate logic, resolution uses two inference rules: resolution and instantiation. Resolution is like before. Instantiation is a rule that says that a given clause containing a variable, for example x, can be succeeded by a clause where all occurrences of x have been replaced by an arbitrary term. This term can be another variable (for example y), it can be a term containing x (for example $h(x)$), or it can be a term using several variables including both x and others, or only others.

There is only one kind of usage of the instantiation rule, namely to modify two clauses so that resolution is possible between them. Consider for example:

```
P(f(y))
-P(x) v Q(x)
```

where the second clauses expresses of course that $P(x) \rightarrow Q(x)$ for all x. Here the second clause can be instantiated into

```
P(f(y))
-P(f(y)) v Q(f(y))
```

and then resolution is possible.

In practical operation one therefore integrates resolution and instantiation into one single rule, called resolution, where instantiation is done on one or both of the given clauses in such a way that resolution then becomes possible.

The operation of *unification* is important in this context: unification is the task where two literals with the same predicate are given, and one obtains one substitution for each of them whereby they become equal.

The empty clause

Since a clause is a set of literals, one may ask whether the empty set is also a clause. The answer is yes, and in fact the empty clause plays an important role when resolution is used. It arises in situations like the following:

```
  {a}
  {-a}
---------
  {}
```

which shows that the empty clause represents *contraction.* But why should one expect contradictions to occur in a proof - one would expect that the given axioms do not contain any contradiction? The reason is that in order to prove a proposed conclusion B from a given set Γ of axioms, one usually *adds the negation of B to the axioms and tries to prove contradiction.* This can be done in several ways. One can use the axioms to prove B, and then B and its negation give contradiction in the last step. This is a 'forward' proof of B. However, one can also use B together with axioms such as $C \rightarrow B$ to obtain C as a subgoal, thereby obtaining a 'reverse' proof from B back to the axioms. The following shows how this works:

```
  -B          (negated goal proposition)
  -C v B      (C implies B, on clause form)
---------
  -C          (here C has become a subgoal)
```

The lecture notes for a later lecture will show an example of the use of this method.

4 The Skolem transformation

The introduction of Skolem functions that was described above requires some additional comment. Clearly the result of the Skolem transformation is not equivalent to the original clause, in the sense of ⇔, since one even introduces a new function symbol. However, the transformation is useful and appropriate anyway in view of the following two theorems.

I. $A \vdash B$ if and only if $A \wedge \neg B \vdash$ contradiction

II. If A' is the Skolem transformation of A, then $A \vdash$ contradiction iff $A' \vdash$ contradiction.

Therefore, if we can do the following: (1) take the given axioms and add the negation of the proposed conclusion to them, then (2) convert the entire set to clause form, using Skolemization when necessary, (3) derive a contradiction (empty clause) from the result of step 2, then we can be sure that the result of step 1 also has a proof for contradition. Then by the first of the two theorems we know that B follows from the given A. Also, the converse holds as well, so that if there exists a proof for the desired conclusion from the given axioms (for example, without using resolution), then the three steps 1 - 3 that were just mentioned can be carried out.