KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Notes on Propositional and Predicate Logic for the Knowledge Representation Framework

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).
The present report, PM-krf-013, can persistently be accessed as follows:

Project Memo URL:	`www.ida.liu.se/ext/caisor/pm-archive/krf/013/`
AIP (Article Index Page):	`http://aip.name/se/Sandewall.Erik.-/2010/007/`
Date of manuscript:	2010-07-29

Related information can also be obtained through the following www sites:

KRFwebsite:	`http://www.ida.liu.se/ext/krf/`
AIP naming scheme:	`http://aip.name/info/`
The author:	`http://www.ida.liu.se/~erisa/`

Introduction

This compendium is intended for those participants in my course on Artificial Intelligence that have not already taken a standard course in logic. It contains the bare minimum information that is needed as prerequisites, and does not replace an regular textbook.

The notation of the Knowledge Representation Framework (KRF) is used for compatibility with the AI course, but we also mention other standard notations in logic. Notice that there is a fairly large variety of notations for formal logic in the literature.

Chapter 1

Propositional Logic

A *logic formula* in propositional logic is either a *proposition symbol* or a *composite formula* which can be on any of the following forms

```
(not p)
(and p q)
(or p q)
(imp p q)
(eqv p q)
```

where the components `p` and `q` are in turn logic formulas, recursively. In the remainder of this chapter, the phrase 'logic formula' means logic formula in propositional logic.

A *vocabulary* of a logic formula or a set of logic formulas is a set of proposition symbols containing all the proposition symbols that occur in it, or in any of them. The vocabulary may also contain other proposition symbols besides those that occur in the given formula(s).

1.1 Evaluation

An *interpretation* of a logic formula is a mapping from a vocabulary for it to the truth-values `T` and `F`. In other words, the interpretation assigns one of the values `T` or `F` to each proposition symbol in the vocabulary, and it may also assign values to some other proposition symbols.

Let `r` be a logic formula and `v` be an interpretation of `r` . The *value of* `r` *in* `v` is written `(val r v)` and is defined recursively as follows. If `r` is a proposition symbol then `(val r v)` is the value that `v` assigns to `r` , that is, `(v r)`. Otherwise, `r` is a composite formula of some of the forms shown above, and its value is defined as follows.

- If `r` is `(not p)`: If `(val p) = T` then `F` otherwise `T`
- If `r` is `(and p q)`: If both `(val p) = T` and `(val q) = T` then `T` otherwise `F`
- If `r` is `(or p q)`: If both `(val p) = F` and `(val q) = F` then `F` otherwise `T`

- If `r` is `(imp p q)`: If both `(val p) = T` and `(val q) = F` then `F` otherwise `T`
- If `r` is `(eqv p q)`: If `(val p) = (val q)` then `T` otherwise `F`

An interpretation for a logic formula is called a *model* if the value of the formula is `T` in that interpretation.

1.2 Equivalence Rules

A *joint vocabulary* for two or more logic formulas is a set of proposition symbols that is a vocabulary for both or all of them, that is, it contains all the proposition symbols that occur in any of the formulas.

Two logic formulas are said to be *equivalent* iff they have the same value in all their joint vocabularies. We write `p == q` to express that `p` and `q` are equivalent. Notice that the expression `p == q` is not *a logic formula,* it expresses a relation between *two* logic formulas.

Do not confuse the connective `eqv` which can occur in logic formulas, with the relation `==` that relates two logic formulas.

The equivalence relation `==` has the following simple properties which can easily be verified, for arbitrary `p, q,` and `r.`

- `p == p`
- If `p == q` then `q == p`
- If `p == q` and `q == r` then `p == r`

The following equivalence rules for logic formulas are also easily verified. In all cases `p, q,` and `r` are arbitrary subformulas.

- `(not (not p)) == p`
- `(and p p) == p`
- `(and p q) == (and q p)`
- `(and p (and q r)) == (and (and p q) r)`
- `(or p p) == p`
- `(or p q) == (or q p)`
- `(or p (or q r)) == (or (or p q) r)`
- `(not (and p q)) == (or (not p)(not q))`
- `(not (or p q)) == (and (not p)(not q))`
- `(and p (or q r)) == (or (and p q)(and p r))`
- `(or p (and q r)) == (and (or p q)(or p r))`
- `(imp p q) == (or (not p) q)`
- `(eqv p q) == (or (and p q)(and (not p)(not q)))`

Finally we have the following *substitution rule:* If `p == q,` the formula `p` contains a subformula `r` where `r == r',` and `p'` is obtained from `p` by replacing `r` by `r',` then `p' == q.`

These rules are used very frequently for rewriting logic formulas to some other, equivalent form, so it is important to know them well.

According to these rules the relation `==` has the properties of the equality relation. The reason why we do not simply write it as equality is that in a later chapter we are going to introduce logic formulas that have the form `[= a b]` so that equality is used *within* logic formulas. Therefore it is strongly recommended to pronounce `==` as 'is equivalent to' and not as 'equals'.

1.3 Entailment Rules and Proofs

Equivalence rules are two-way rules: if `p` can be replaced by `q` according to an equivalence rule, then `q` can be replaced by `p.` Entailment rules are corresponding one-way rules and are defined as follows. A logic formula `p` is said to `entail` a logic formula `q` iff `q` is true in all interpretations where `p` is true, where of course one only uses interpretations over a joint vocabulary for `p` and `q.` This is written `p |= q.` Here are some examples of entailment rules.

- `(and p q) |= p`
- `p |= (or p q))`
- `(not p) |= (imp p q)`
- `(and p (not p)) |= q`

Moreover, a pair of logic formulas `p` and `q` is said to entail a third logic formula `r,` which is written `p, q |= r` iff `r` is true in all interpretations where both `p` and `q` are true, considering only interpretations over a joint vocabulary between `p, q` and `r.`

The formulas that are given to the inference rule are called *antecedents* and the result is called a *consequent.* Here are some additional examples of entailment rules, now using two antecedents.

- `p, (imp p q) |= q`
- `p, q |= (and p q)`

The first one of these rules is called *Modus Ponens* and it is a quite important rule.

Entailment rules are used for *proofs.* Proofs are constructed as follows. One is given a set of logic formulas, which are called the *premises* of the proof. Then one adds successively more logic formulas, where each of them must be obtained from some of the previously introduced formulas according to an entailment rule. These are called *inferred* formulas. It is customary to write each premise and each inferred formula on a separate line. The last formula in the proof is called the *conclusion* of the proof.

The following is a simple example of a proof. There are two premises, marked `P1` and `P2`.

```
P1  (and p q)
P2  (imp q r)
    q
    r
```

1.4 Resolution Theorem-Proving

There are several ways of using logic formulas computationally. Some computational methods are based on doing equivalence transformations on logic formulas, or simply on evaluating a given set of logic formulas repeatedly in particular ways. Yet another type of logic-based computation is *theorem-proving,* where the program searches for a proof with a desired conclusion, or a conclusion that belongs to a given class of desired conclusions. The lecture notes on Knowledge Representation describes computational methods of these different types.

Most theorem-proving programs are organized using the *resolution method* for first-order predicate logic. First-order logic is a generalization of propositional logic and is described in the next two chapters. However the resolution method can also be used in the special case of propositional logic, and we shall now describe the resolution method for propositional logic as an introduction.

Consider that a set of logic formulas (the premises) and a specific logic formula (the desired conclusion) are given, and one wishes to use the resolution method for finding a proof of the desired conclusion. The first step is then to rewrite the premises and the desired conclusion on *conjunctive normal form.* In this case one first of all generalizes the `and` and `or` operators so that they can take an arbitrary number of arguments, and not necessarily two. Thus one can write, for example,

```
(or p q r s)
```

This can mean any of the following

```
(or p (or q (or r s)))
(or (or (or p q) r) s)
(or (or p q)(or r s))
```

as well as a number of other configurations. These variants are equivalent so it does not matter which of them is used. The definition of `val` for `or-` and `and-` expressions can easily be generalized for the case of an arbitrary number of arguments.

A logic formula is on conjunctive normal form if it is an `and-` expression where each of the elements is an `or-` expression, and each argument of the `or-` expression is a *literal.* A literal is defined as being either a proposition symbol, or an expression of the form `(not p)` where `p` is a proposition symbol. The following is an example of a formula in conjunctive normal form.

```
(and (or p (not q) r) (or (not p) q)(or r))
```

The procedure for rewriting a formula on conjunctive normal form is quite easy:

- Replace all occurrences of `imp` and `eqv` by expressions using `and, or,` and `not.`
- Move all occurrences of `not` "inwards" using
  - `(not (and p q)) == (or (not p)(not q))`
  - `(not (or p q)) == (and (not p)(not q))`
- Simplify all subexpressions of the form `(not (not p))` to `p`
- Move all occurrences of `or` "inside" occurrences of `and`
- Simplify all `or-` expressions for example by rewriting `(or (or p q) r)` as `(or p q r)` , and similarly for `and`

Each premise is converted to conjunctive normal form in this way. Then the surroundig `and-` expression is removed so that each premise becomes a *set of* `or-` expressions, and one takes the union of those sets. In this way, one obtains the entire set of premises as a set of `or-` expressions.

The same operation is done on the desired conclusion. If it is a set of a single `or-` expression then that is the conclusion that is to be proved. If it is a set of more than one `or-` expression then one makes a separate proof for each of them.

For readability it is convenient to write `(not p)` as `-p.` This is possible since after the transformations, all uses of `not` have a proposition symbol as its argument, and never a composite expression.

Finally, instead of considering each `or-` expression as having a *sequence* of arguments, one considers it as having a *set* of arguments. This means that the order of the arguments is considered as unimportant, and repeated use of the the same argument does not count. For example, `(or p q p)` is considered as *the same expression as* `(or p q).` This means that a number of equivalence rules that were mentioned above can be eliminated since they are implicit in the notation.

An `or-` expression having a set of literals as its argument set is called a *clause* in the resolution method. After the described transformations one has a set of clauses as the premises, and a single clause as the desired conclusion, and this is the starting point for constructing a proof. One single entailment rule is used for the proof, namely the *resolution rule,* which has the following form:

```
(or p Q), (or -p R) |= (or Q R)
```

where both `Q` and `R` represents no, one or more literals. For example,

```
(or a b -c), (or -a d) |= (or b -c d)
```

It is easily seen that this is a correct entailment rule, in the following ways, and for the example. Consider now an arbitrary interpretation where both `(or a b -c)` and `(or -a d)` are true. The value of `a` in that interpretation can be either `T` or `F.` If it is `T` then `d` must be true. If it is `F` then either `b` or `-c` must be true. In either case at least one of `b, -c` and `d` must be true, which means that the expression `(or b -c d)` must be true. The general form of the rule is explained in the same way.

The resolution rule is a generalization of Modus Ponens, since the latter can be written equivalently as

```
(or p), (or -p q) |= (or q)
```

1.5 Proof by Contradiction

One important way of making proofs is using *proof by contradiction.* Suppose you have a set of premises Γ and a desired conclusion `p`. Let Γ' be obtained by adding `(not p)` to Γ. If it is possible to prove two propositions `q` and `(not q)` from Γ', then one has a proof of `p` from Γ. The argument is that if `(not p)` were true then there would be a contradiction, which is not possible, and therefore `p` must be true. Proofs by contradiction are quite common both in everyday reasoning, and in mathematical texts.

The formal motivation for proof by contradiction is as follows. Consider the set M of models for Γ, i.e. the set of all interpretations where all members of Γ are true. Let M^+ be the subset of M containing those interpretations where p is true, and M^- those where p is false, and let m^- be an arbitrary member of M^-. Then all members of Γ' are true in m^-. All conclusions from the members of Γ' are also true in m^-, according to the definition of entailment rules. Therefore both `q` and `(not q)` are true m^-, but this is not possible, and it follows that M^- must be the empty set, so that $M = M^+$. Therefore p is true in all interpretations where the members of Γ are true.

Proofs by contradiction can be used in different varieties of logic, and in particular they are very often used in the resolution method. Instead of converting the given premises and the desired conclusion separately, one adds the negation of the desired conclusion to the premises, converts the extended premises to clause form, and attempts to find a proof for a contradiction, that is, to prove both `(or p)` and `(or -p)` for some proposition symbol `p`. In fact, according to the definition of the resolution rule, if one has obtained `(or p)` and `(or -p)` as conclusions then one can also obtain the clause `(or)`. This somewhat artificial expression is therefore a representation for "contradiction" in the resolution method.

Chapter 2

Relational Predicate Logic

Relational predicate logic is a generalization of propositional logic that is obtained by allowing proposition symbols to have arguments, and they are then called *predicates.* The concepts and definitions for predicate logic are similar to those for propositional logic. However, in order to establish the convention that all occurrences of a predicate have the same number of arguments one must start with the definition of a vocabulary. The definitions of terms in this chapter apply to relational predicate logic.

A *vocabulary* consists of a *vocabulary of predicates,* a *vocabulary of constants* and a *vocabulary of variables.* The latter two are simply two disjoint sets of symbols. The vocabulary of predicates is a mapping from symbols (called predicate symbols) to non-negative integers, where the predicate symbols can not be members of the other two vocabularies.

In KRF we write variables preceded by a point (for example `.x`) or using fixed-width italic characters (for example ***x***) depending on what is available in the graphic medium at hand.

2.1 Ground Propositions

We begin with the case where there are no variables, merely predicates and constants, so the vocabulary of variables is the empty set. In this case formulas are called *ground formulas.* A *ground atomic formula* for a vocabulary is an expression of the form `[p a b ...]` where `p` is in the vocabulary of predicates and each of the *arguments* `a, b,` etc. is in the vocabulary of constants.

For example, if `is-capital-of` is in the vocabulary of predicates and `finland` and `helsinki` are in the vocabulary of constants, then `[is-capital-of helsinki finland]` is a ground atomic formula for the vocabulary in question.

Notice that it is not necessary that all the predicates and constants in the vocabulary are used in the formula, so for each formula there can be several vocabularies for it, just like for propositional logic.

A *ground formula* for a vocabulary is formed from ground atomic formulas using the propositional operators `not, and, or,` etc. which were defined

in the previous chapter.

An *interpretation* for a vocabulary consists of the following components:

- A set of objects, called the *domain*
- A mapping from constants to objects in the domain, called the *constant mapping*
- A mapping from atoms to the truth-values `T` and `F` , called the *predicate mapping*

In this definition, an *atom* is similar to a ground atomic formula except that the arguments are objects in the domain, rather than constant symbols.

The *value* of a ground atomic formula in an interpretation is obtained by replacing each argument in the formula with its value according to the constant mapping, and then obtaining the value of the result using the predicate mapping. The value of an arbitrary ground formula is obtained using the evaluation rules for the propositional operators that were defined in the previous chapter.

Example. Consider an interpretation where the domain consists of the five objects `DK, SE, NO, FI, IS.` These are intended to represent the five Scandinavian countries. Consider then a vocabulary where the predicate vocabulary is a mapping from the predicate symbol `border-between` to the integer 2, indicating that it is a predicate of two arguments, and where the constant vocabulary consists of the following constants:

```
sweden
denmark
finland
suomi
```

Let the constant mapping consist of the following maplets:

```
[: sweden SE]
[: denmark DK]
[: finland FI]
[: suomi FI]
```

Notice that there is no one-to-one correspondence between constants and domain objects: several constants may have the same value, and some domain objects are not the value of any of the constants. The predicate mapping assigns the value `T` to the following atoms:

```
[border-between SE FI]
[border-between FI SE]
[border-between SE NO]
[border-between NO SE]
[border-between NO FI]
[border-between FI NO]
```

It assigns the value `F` to all other atoms using the same vocabulary. (The predicate `border-between` is interpreted as "having a land border in common," and having a bridge in common is not included.

This concludes the specification of the interpretation. The following are the values of some formulas in this interpretation.

```
[border-between sweden suomi]           T
[border-between finland denmark]        F
[border-between finland finland]        F
(or [border-between sweden denmark]
    [border-between suomi sweden])      T
```

2.2 Variables and Quantifiers

Ground propositions can be used for creating databases, but they are not very useful for drawing conclusions. The introduction of *variables* and *quantifiers* provides that additional expressivity. One then uses vocabularies with a non-empty vocabulary of variables. The subsequent definitions are extended accordingly.

An *atomic formula* for a given vocabulary is an expression that has the form `[p a b ...]` where `p` is in the vocabulary of predicates and each of the *arguments* `a, b,` etc. is in the vocabulary of variables or the vocabulary of constants.

A *logic formula* for a vocabulary is a formula that is formed from ground atomic formulas using the propositional operators `not, and, or,` etc., or that is formed as either of the following:

```
[all x p]
[exists x p]
```

where `x` is a variable and `p` is a logic formula. The actual variable my be for example `.s` or `.v` but we write `x` to indicate that `x` can be any of those. Technically it is called a *metavariable* since stands for any of those variables.

The `all` and `exists` operators are called *quantifiers,* and a formula with a quantifier as its top-level operator is called a *quantified formula.* A logic formula is said to be *closed* if all occurrences of variables in predicate argument positions are inside an `all-` or `exists-` expression with that variable as its middle element.

The definition of an *interpretation* is unchanged.

A *variable assignment* for the combination of a vocabulary and an interpretation is a mapping from the variables to objects in the domain.

The *value* of an atomic formula in an interpretation and a variable assignment for that interpretation is obtained by replacing each constant in the formula with its value according to the constant mapping, replacing each variable in the formula with its value according to the variable assignment, and then obtaining the value of the result using the predicate mapping.

The value of an arbitrary formula is obtained using the evaluation rules for the propositional operators that were defined in the previous chapter, together with the evaluation rules for the quantifiers which are as follows. Consider the evaluation of an expression `[all x p]` in an interpretation I and a variable assignment B. For each object d in the domain, construct a modified variable assignment B/d where the variable `x` is bound to the value d and which is otherwise like B. Obtain the value of `p` in I and every such B/d. If the value is `T` for all those B/d then the value of the quantified expression is `T,` otherwise it is `F` .

The value for an `exists-` expression is defined in a similar way except that the value of the quantified expression is `T` iff the value from *at least one* of the evaluations with B/d has the value `T` .

The quantifiers can be understood as counterparts of the summation (Σ) and product operators in mathematics. Just like summation is a repeated application of addition over a range of values for the summation index, so the `all-` operator represents a repeated application of the `and` operator over the domain objects, and the `exists-` operator represents a repeated application of the `or` operator.

2.3 Equivalence Rules for the Quantifiers

All the equivalence rules that were introduced in Chapter 1 apply for propositions in relational predicate logic as well. The following equivalence rules for quantifiers apply as well and their correctness is easily verified. The following definition is used: An occurrence of a variable in a logic formula is said to be a *free occurrence* iff it is not a (direct or indirect) subexpression of a quantification expression having that variable as its middle element.

- In a formula `[all x p]` if `p'` is obtained from `p` by replacing all free occurrences of the variable `x` by the variable `y` , and if none of these occurrences is inside a quantification of the variable `y` then `[all x p] == [all y p']` , and similarly for `exists.`
- If a formula `p` does not contain any free occurrences of a variable `x` then `p == [all x p]` and `p == [exists x p]`
- `(not [all x p] ) == [exists x (not p)]`
- `(and [all x p] [all x q] ) == [all x (and p q)]`
- `(or [exists x p] [exists x q] ) == [exists x (or p q)]`
- If `q` does not contain any free occurrence of the variable `x` then `(or [all x p] q) == [all x (or p q)]` and `(and [exists x p] q) == [exists x (and p q)]`

These equivalence rules are used widely when working with logical formulas containing quantifiers. They are also used when transforming a set of premises to clause form in order to use the resolution method, but in that case one needs a more general form of the logic which will be introduced in the next chapter.

2.4 Entailment Rules for the Quantifiers

The following are some entailment rules involving quantifiers.

- `[all x p] |= p'` where `p'` is obtained from `p` by replacing all free occurrences of the variable `x` by an arbitrary constant.
- `[all x p] |= p'` where `p'` is obtained from `p` by replacing all free occurrences of the variable `x` by another variable `y` , provided that none of those free occurrences of `x` is inside the scope of a quantification that binds `y.`

The following is an example of a simple proof using the first of these entailment rules. Each line begins with a line number, which is preceded by the letter P for the premises, and each non-premise line ends with an indication of which earlier line(s) is/are used for obtaining the present line.

```
P1   [all .x (imp [P .x][Q .x])]
P2   [all .y (imp [Q .y][R .y])]
P3   [all .x (imp [R .x][V .x])]
P4   [P green]
 5   (imp [P green][Q green])     P1
 6   [Q green]                    P4,5
 7   (imp [Q green][R green])     P2
 8   [R green]                    6,7
 9   (imp [R green][V green])     P3
10   [V green]                    8,9
```

Chapter 3

First-Order Predicate Logic

In this chapter we make one more important extension of the logic language from the previous section, namely, introducing the use of *functions.* In this way the arguments of predicates can be composite expressions, called *terms,* and not merely single constants or variables. The resulting logic is called *First-Order Predicate Logic* (FOPL) or *First-Order Predicate Calculus* (FOPC).

First-order predicate logic can be used for expressing many kinds of formulas that arise in mathematics. A simple example would be

```
[all .x [all .y [= (+ .x .y) (+ .y .x)]]]
```

expressing that for all values of `x` and `y` , `x+y = y+x.` In the present compendium we use prefix notation throughout for predicates and functions, but if infix are also allowed as described in the *KRF Framework* report one can write the same logic formula as

```
[all .x [all .y [(.x + .y) = (.y + .x)]]]
```

However, the major use of predicate logic in Artificial Intelligence and Knowledge Representation is not for expressing mathematical information, but for expressing application knowledge of other kinds. In those cases one tends to obtain a fairly large number of predicates and of functions, as well as a large number of constants, and in those uses it is convenient to use the prefix-oriented and fully parenthesized notation of the Knowledge Representation Framework.

3.1 Formulas and Evaluation

The definitions for first-order predicate logic follow the same pattern as for relational predicate logic in the previous chapter, and some of the definitions are unaffected by the generalization and therefore unchanged. They proceed as follows.

A *vocabulary* consists of a *vocabulary of predicates,* a *vocabulary of functions* and a *vocabulary of variables.* The last one is simply a set of symbols.

The vocabularies of predicates and of functions are mappings from symbols (called predicate symbols and function symbols, respectively) to non-negative integers. The sets of variables, function symbols and predicate symbols must be disjoint.

A function that is mapped to zero in the vocabulary of functions is called a constant, and is considered the same as a constant as defined in the previous chapter.

A *term* for a vocabulary is either a constant, a variable, or an expression of the form `(f a b ...)` where `f` is a function symbol, each of the arguments `a, b,...` is a term, and the number of terms is the same as the number specified for `f` by the vocabulary of functions.

The definition of an *atom* remains as before. It has a counterpart in the definition of a *term-atom* which is similar to a term except that the arguments must be objects in the domain.

An *interpretation* for a vocabulary consists of the following components:

- A set of objects, called the *domain*
- A mapping from constants and term-atoms to objects in the domain, called the *function mapping* , where `f` and `(f)` are mapped to the same object if `f` is a function symbol that is specified to have zero arguments
- A mapping from atoms to the truth-values `T` and `F` , called the *predicate mapping*

The definition of a *variable assignment* is unchanged.

The *value of a term* for an interpretation and a variable assignment for that assignment is obtained as follows. If the term is a variable then its value is obtained from the variable assignment. If the term is a constant `f` then the value is obtained by applying the function mapping to the expression `(f)` If the term has the form `(f a b ...)` then the value is obtained by applying the function mapping to the term-atom `(f a' b' ...)` where `a'` is the value of `a` with the same interpretation and variable assignment, and similarly for the other arguments.

The definition of the value of an atomic formula is like above, except that for the arguments one must now use the definition for the value of a term that allows for terms consisting of a function with arguments.

The definition of the value of a composite logic formula is unchanged, including both formulas formed using the propositional connectives and formulas formed using the quantifiers.

3.2 Equivalence Rules and Entailment Rules

No additional rules are needed in principle, and the existing rules generalize in their natural ways. However, when the substitution of a variable by another variable is generalized to the substitution of a variable by a term that may contain one or more variables, then the scope restriction will apply to each of those variables. For example, if an expression `[all x P]`

is instantiated by replacing all free occurrences of `x` in `P` by the term `(h y z)` then this is only allowed if none of the free occurrences of `x` in `P` is a direct or indirect subexpression of a quantification over the variable `y` or the variable `z`.

3.3 Resolution Operator for Predicate Logic

The resolution operator is defined for predicate logic, and the case of propositional logic that was described above is just a special case. For predicate logic, like for propositional logic, one uses clauses that represent the disjunction (i.e., `or-` expression) of a set of literals. The difference now is that each literal is a plain or negated atomic proposition, where the arguments can be arbitrary terms so that they can use functions and variables.

Negated atoms can be represented by preceding the predicate symbol with a dash, so for example `(not [P a b] )` can be written as `[-P a b]` like in the propositional case.

Predicate-logic clauses do not contain quantifiers, so all variables are "free variables," and the interpretation is such that there is an implicit universal quantifier (i.e., `all-` quantifier) around the entire clause, and for every variable that occurs there. In principle there are two entailment rules, namely, the propositional resolution operator that was introduced above, and the instantiation rule which says that any variable can be replaced by an arbitrary term, provided that this is done for all occurrences of the variable in the clause. For example, the following clause

```
(or [P x y][R y x])
```

entails each one of the following, given that `x` and `y` are variables and `a` and `b` are constants:

```
(or [P a b][R b a])
(or [P a a][R a a])
(or [P y x][R x y])
(or [P (f a)(g b)][R (g b)(f a)])
```

as well as arbitrarily many others.

However, in practice one combines propositional resolution with instantiation so as to make exactly as much instantiation as is needed in order to *unify* terms, that is, make them equal. Suppose for example that the following two clauses are given.

```
(or [P x (g y)][Q x y])
(or [-P (h b) v][R v])
```

Here one can instantiate the clauses so that the two literals for `P` become equal:

```
(or [P (h b)(g y)][Q (h b) y])
(or [-P (h b) (g y)][R (g y)])
```

and then the resolution operator gives:

```
(or [Q (h b) y][R (g y)])
```

It has been proved that it is *sufficient to consider the most general instantiations.* Other and more specialized instantiations are also possible, for example substituting `y` by `(f z),` but according to that result no additional conclusions can be obtained by also considering that substitution, besides the one showed above. The idea is that if that more specialized substitution will ever be needed later on in a proof, it can always be obtained when needed by doing an additional instantiation then.

On the other hand, there are in fact cases where two given clauses can be resolved in more than one way, namely if they contain more than one pair of resolvable literals. For example, if one clause contains both a literal with `P` and one with `Q` , and the other one contains both a literal with `-P` and a literal with `-Q` then one can choose either to resolve with the `P` literals, or to resolve with the `Q` literals. Both of these resolutions with their separate resulting clauses have to be considered in order not to miss any conclusions. Notice that one can not resolve both literal pairs at the same time!

3.4 Existential Quantifier Elimination

The transformation of given premises to clause form is done in a similar way for the case of predicate logic as for the case of proposition logic, except there is a major problem namely the elimination of quantifiers. The equivalence rules for quantifiers make it possible to move all quantifiers outwards, so that the result is a formula that contains one or more quantifiers inside each other on the outermost levels, and then a quantifier-free formula inside. That inside part can be transformed to conjunctive normal form without problems, but the problem remains with the quantifiers. If all of them are `all-` quantifiers then nothing needs to be done, but if some of them are existential quantifiers then one does not have the clause form that is required for the resolution operator.

The solution to this problem assumes first of all that the proof is done by contradiction, so that one has added the negation of the desired conclusion to the given premises, and that the composite of those is being transformed to clause form. Then, after having done the standard transformations, suppose for example that one has a formula of the form

```
[all x [all y [exists z [all v P]]]]
```

Now the following transformation is done: for each existential quantification, introduce one more function symbol that is not already in the vocabulary, for example `g` in the example, and let it have the number of arguments that is specified by the number of universal quantifications outside the existential quantification in question. In the example one would let `g` have two arguments. Every occurrence of `z` in `P` is replaced by the term `(g x y)` .

A function that is introduced in this way is called a *Skolem function* and the transformation is called *Skolemization.* (Thoralf Skolem was a Norwegian logician who invented this technique). The idea with this is that one can always extend a function mapping so that for each combination of values of `x` and `y` one picks one of the existing `z` as the value of `(g x y).` The existential operator says that there exists `at least one` such object, and even if there are several one can choose one of them for the value of `g` with those arguments.

This process is repeated for all the existential quantifications. Notice that if there are several existential quantifications in the same formula, then it is only the universally quantified variables outside a given existential quantification that shall be used for the function being introduced. Existentially quantified variables outside the given one will go away by the transformation, so they shall not be used in this way.

In the example one obtains `[all x [all y [all v P'] ] ]` where `P'` has been obtained from `P` by replacing each occurrence of the variable `z` by `(g x y)` .

There is one important thing that one must remember about this transformation: *it does not result in an equivalent formula.* It is *not* the case that the obtained formula is equivalent to the given one under the formula equivalence relation that we write as `==`. However, it is useful anyway because there is another, weaker relation between the formula before and after the transformation, namely *one of them is inconsistent if and only if the other one is.* This is all we need given that the resolution method is used for a proof by contradiction. If resolution is able to produce a contradiction then we know that the transformed formula is inconsistent, and then the original formula is inconsistent as well.

Chapter 4

Representation of Application Domains

The uses of predicate logic for expressing information about computer applications is a broad and complex topic and it is neither possible nor reasonable to address it in the present notes. We shall only make a few points of general relevance; more follows in the lecture notes on knowledge representation.

4.1 Partial Information

When logic is used for knowledge representation it is usually the case that one describes the application using a *knowledgebase* i.e. a set of logic formulas which are used for theorem-proving or other computations. This knowledgebase may contain a large set of ground literals that express elementary facts about the application; it may also contain a moderately sized set of non-ground formulas (i.e. formulas that use quantifiers and variables) that express general properties of the application.

Applications may also be described using ordinary databases. One important difference between using a database and using a logicist knowledgebase is that the latter can express partial and incomplete information in more flexible ways. The following is a simple example to illustrate this point. Suppose we let the object domains in our interpretations consist of the integers, we use one single predicate namely equality, and we use one single function namely multiplication, with two arguments. We may need a number of axioms that characterize equality and multiplication (more about this later), but our application also dictates the use of one constant called `foo` as well as the following premise that is included in the knowledgebase.

```
[= (* foo foo) 25]
```

Since `foo` is a constant, each interpretation shall assign a value to it, and by our assumption this value shall be an integer. It is easily seen that in order for this premise to be true in an interpretation, it is necessary for the value of `foo` to be either 5 or -5. Therefore, if `G` is the conjunction of all the other axioms that are needed, we will have

```
[= (* foo foo) 25], G |= (or [= foo 5][= foo -5])
```

In other words, with this knowledgebase we do not know exactly what is the value of `foo,` and we can not conclude it, but we are able to conclude and to express that it is either of two possible values.

This example is suggestive but it is also incomplete, since it has left open a number of important questions: how can we restrict the object domain to consist of the integers; how can we define the properties of the multiplication function, and so on. The following sections will address some of those questions.

4.2 Equality

It is very common that one wishes to use *equality* as a predicate in one's domain model. However, equality must then be understood as saying that *the same domain object* is being referred to. In our Scandinavian countries example above, for example, the formula `[= finland suomi]` should have the value `T` since the two constants `finland` and `suomi` (the name for Finland in Finnish language) refer to the same object, according to the selected interpretation. The fact that the two arguments are different constant symbols is not a reason for saying the the formula has the value `F.`

The familiar properties of the equality relation must be specified explicitly using axioms if one uses standard first-order predicate logic. (Alternatively one may use a more sophisticated logic where equality is "built in" but this is outside the scope of these notes).

```
[all .x [= .x .x]]
[all .x [all .y (imp [= .x .y][= .y .x])]]
[all .x [all .y [all .z (imp (and [= .x .y][= .y .z])
                             [= .x .z] )]]]
```

Moreover, for every argument position of every predicate symbol there must be an axiom similar to the following one:

```
[all x [all y [all z (imp [= x y](eqv [P x z][P y z]))]]]
```

Similarly, for every argument position of every function symbol there must be an axiom similar to the following one:

```
[all x [all y [all z (imp [= x y][= (f x z) (f y z)])]]]
```

These rules that do not specify a single axiom but a set of similar axioms are called *axioms schemas.*

4.3 Closed Domains and the Unique Names Assumption

Predicate logic is defined in general in such a way that several constants may have the same object as value, and there may be objects that are not named by any constant. However in some applications one may wish to make the opposite assumption, that is, to assume a one-to-one correspondence between objects and constants. There are separate axioms for the two parts of this which should be included in the domain model.

The *closed domains assumption* says that there is no domain object besides those that are named by the constants. It is expressed using an axiom of the form

```
[all x (or [= x a][= x b][= x c] ... [= x g])]
```

where the constant symbols `a` and onwards are all the members of the vocabulary of constants.

The *unique names assumption* says that no domain object can be the value of more than one constant. It is expressed using an axiom of the form

```
(and [/= a b][/= a c] ... [/= a g][/= b c] ... )
```

enumerating all pairs of different constant symbols. In the particular case of the equality relation we write its negation as `/=` and not as `-=` since the former looks more natural.

Notice that both of these require the use of the equality axioms in order to be of any use.

4.4 Standard Domains, Predicates and Functions

Suppose our application makes use of a number of numerical, integer quantities and we wish to characterize them using logic formulas. For example, the application may involve persons each of which has an age, expressed as an integer number of years, and the general rules may include a rule saying that the age of a person is less than the age of the person's parents, provided that the latter exist in the sense of being alive.

One way of organizing this is to decide that the vocabulary shall include, among other things, the textual representations of all integers, for example the symbols `23` and `-12,` and each interpretation shall map each of these symbols to the corresponding number. For example, the *symbol* `23` that consists of the digit 2 and the digit 3 shall be mapped to the *number* 23 which is equal to 22+1. This decision can be partly enforced using the techniques of the previous section. (Actually it is problematic for a number of reasons, such as that there is an infinite number of integers, but we ignore that at this point).

Then we shall also need functions, such as the function `+` for addition and the function `*` for multiplication. The arrangement in "pure" predicate logic is that one shall include these function symbols in the function vocabulary, and one shall introduce premises which are then also called *axioms* that characterize the properties of these functions, so that only those interpretations are admitted where the function mapping assigns the intended functions to these function symbols. This is however a clumsy way of doing things, and it is more convenient to use *standard functions* in such cases. The idea is simple: besides specifying that the object domain shall contain the integers, one specifies also that the function mapping shall map the function symbol `+` to a mapping representing addition, that is, a mapping consisting of all maplets of the form

```
[: <x y> x+y]
```

for all combinations of `x` and `y` . Then one does not need to write out any axioms that characterize the properties of the addition function.

This approach is computationally straightforward as long as one merely has to implement the evaluation operator, which was called `val` above, since when it is required to evaluate a term having `+` as its leading function, then instead of looking up the value of the term for the evaluated arguments using the function mapping, one will simply invoke a plug-in the implements the definition of addition of integers. Unfortunately, however, the problem comes back in another form when one implements theorem-proving, for example using resolution. Consider the problem of resolving the following two clauses:

```
(or [P 24] G)
(or [-P (+ v 2)] [H v])
```

Clearly in order to unify the literals for `P` one must select `v` as 22, obtaining the resolution result

```
(or G [H 22])
```

In other words, it is not sufficient to have a plug-in that is able to perform addition, one also needs another plug-in that is able to perform the inverse operation of addition with respect to one of the arguments. Furthermore, consider the problem of resolving the following two clauses:

```
(or [P (+ x 4)] [G x])
(or [-P (+ y -3)] [H y])
```

These can be resolved as follows in a step-by-step process. First rewrite them as

```
(or [P (+ (+ y -7) 4)] [G (+ y -7)])
(or [-P (+ y -3)] [H y])
```

Simplify to

```
(or [P (+ y -3)] [G (+ y -7)])
(or [-P (+ y -3)] [H y])
```

Now apply the elementary resolution operator, obtaining

```
(or [G (+ y -7)][H y])
```

This example shows how resolution of literals that use a standard function may require a nontrivial combination of operations that capture the properties of that function in a variety of ways. In practice it will not be done using the sequence of steps that were shown here but in a more aggregated way, but the basic point remains.

On the other hand, if a function like `+` in this example is characterized using axioms then the manipulations that were shown in the example will come out through successive uses of the resolution operator.

Standard predicates are introduced and used in a similar way. For example, it may be appropriate to define the less-than relation between integers as a standard predicate.

4.5 Direct Axiomatization of the Non-negative Integer Domain

For comparison, and not for practical use, it may be interesting to see how the integers from 0 and up can be characterized using axioms. In this case, one introduces one constant called 0, one function called `1+` which is intended as the function of adding one to an integer, and one function of two arguments called `+`. (This is the same name of the function `1+` as is used in CommonLisp). The following axioms are used, with universal quantifiers omitted:

```
(imp [= (1+ x) (1+ y)] [= x y])
[= (+ x 0) x]
[= (+ (1+ x) y) (+ x (1+ y))]
A few more?
```

In addition there is an *induction rule:* If `P` is an arbitrary predicate of one argument and the following two formulas hold

```
[P 0]
(imp [P x] [P (1+ x)])
```

then it follows `[all x [P x] ]` From these axioms together with the axioms for equality it is possible to prove for example

```
[= (+ x y)(+ y x)]
```

Doing so is very cumbersome, however, so this is not a useful method from a practical point of view.

The bottom line is that when characterizing an application using a knowledgebase in the sense of a set of logic formulas, it is important to find a balance between what is expressed using axioms and what is expressed using standard domains, functions and predicates.