Knowledge Representation with Logic

Deepak Jain, Helmut Krawinkler and Kincho H. Law

CIFE TECHNICAL REPORT
Number 13

July, 1989

Stanford University
Copyright © 1989 by
Center for Integrated Facility Engineering

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 95305-4020
Knowledge Representation With Logic

Deepak Jain
Kincho H. Law
Helmut Krawinkler

Department of Civil Engineering
Stanford University, Stanford, CA 94305

Symbolic logic is a representational formalism for expressing knowledge and reasoning about such knowledge, in much the same way as calculus is used to represent and manipulate mathematical relationships in various disciplines of science and engineering. In this report we discuss two logical languages, namely, propositional calculus and predicate calculus, along with a brief introduction to the field of logic programming. A structural engineering example is described in detail to illustrate the concepts discussed in the report. Our emphasis is not so much on mathematical rigor as on conveying informally the essence of how logic can be usefully employed for creating knowledge systems in engineering domains. Thus we do not strive for completeness or mathematical precision; however, references to the appropriate material are provided throughout the report for the inquisitive reader.

This report is divided into four major sections. First, we discuss the syntactical (pertaining to the organization of symbols) and semantical (pertaining to the meaning of the symbols) aspects of propositional and predicate calculus in detail and provide a broad overview of a few tools for logic programming. The next section illustrates a prototypical application of logic for structural analysis. A representative implementation of such an application is the subject of Section 3. The final section presents some concluding remarks and suggestions for further reading.

1. General Description

Logic is a formalized treatment of knowledge and thought. To use logic as a knowledge representational formalism, one starts by asserting some axioms, which embody known facts or self-evident truths in a particular field of interest, in a chosen language. From these axioms, new, useful sentences are inferred through syntactic manipulations, without any reference to the meaning of the symbols involved. This is important because a computer can be programmed to carry out syntactic manipulations. If the range of these manipulations is adequate to cover the spectrum of inferences human beings make, a computer can be made to emulate the human

1This is a modified version of a manuscript submitted for inclusion in a forthcoming ASCE monograph on knowledge representation formalisms.
thought process.

For reasons of ambiguity and imprecision, natural languages, such as English, are not appropriate for expressing knowledge contained in the above mentioned axioms and drawing inferences from them. On the other hand, two languages devised by logicians, namely, propositional calculus and predicate calculus, are quite precise, making them apt candidates for computer representation of knowledge in diverse fields. Besides being precise, however, a suitable representation language must also possess sufficient expressive power; that is, one should be able to state in the given language the things that need to be stated. Of the two languages, predicate calculus has more expressive power than propositional calculus—there are facts that can be expressed in predicate calculus but not in propositional calculus. However, an introduction to propositional calculus is useful as a stepping stone to the understanding of predicate calculus.

A representation language has two components: syntax and semantics. The syntax lays down the rules of grammar that enable one to formulate legal sentences in the language. For example, the sentence “Paris updated times word below problem” is syntactically unacceptable in the English language since it violates the rules of English grammar. The semantics of the language, on the other hand, defines the meaning of the expressions in the language. That is how, for example, we humans understand the essence of spoken words. Thus, the symbol Paris is normally taken to denote the city that is the capital of France. In a manner analogous to the English language, both propositional calculus and predicate calculus have well-defined syntactic and semantic rules.

1.1. Propositional Calculus

1.1.1. Syntax

Propositional calculus is a language of abstract sentences that consist of propositions and connectives. Propositions can either be truth symbols, True and False, or propositional symbols, such as P, ConstructionMaterialIsSteel, BldgIsInSeismicZone, and R2D2. In the notational convention selected here, propositional symbols start with an uppercase letter and can consist of alphanumeric characters. The connectives can be and (\&), or (\lor), not (\neg), implies (\Rightarrow), or equivalent (\equiv).\footnote{Some authors (for instance, [Pospese 74]) prefer alternative symbols for connectives, e.g., \& instead of \&; \neg or \neg instead of \neg; \Rightarrow or \Rightarrow instead of \Rightarrow; and \equiv or \equiv instead of \equiv.} Each proposition in itself is a sentence; thus P and True are both legal sentences in propositional calculus. More complex sentences can be formed using connectives. The types of sentences that can be formed are:

\[
\neg \alpha \\
\alpha \land \beta \\
\alpha \lor \beta \\
\alpha \Rightarrow \beta \\
\alpha \equiv \beta
\]

where \(\alpha\) and \(\beta\) can themselves be simple or complex sentences. Parentheses, ( ), and brackets,
[ ], are used to clarify the structure of a sentence when there is a possibility of ambiguity. Using the above rules, it can be seen that the following is a syntactically correct sentence.

\[((\neg P \land Q) \Rightarrow (Q \lor \text{False})) \equiv \text{True}\]

1.1.2. Semantics

Having discussed the syntax of propositional calculus, we move on to study the semantical issues. Using semantical rules, one can determine the truth value of a sentence—that is, a value of either true or false can be assigned to a sentence based on the truth of its propositions and the connections among the propositions. The following rules are used for this purpose.

- The sentence True is always true.
- The sentence False is always false. (Note the distinction between the truth symbols True and False and the truth values true and false.)
- A sentence consisting of a single proposition and no connectives is true if the proposition is true, and false if the proposition is false. Thus the sentence \(P\) will be true if and only if the proposition \(P\) itself is true.
- The sentence \(\neg \alpha\) is true if \(\alpha\) is false, and false if \(\alpha\) is true.
- The sentence \(\alpha \land \beta\) is true if both \(\alpha\) and \(\beta\) are true, and false otherwise.
- The sentence \(\alpha \lor \beta\) is false if both \(\alpha\) and \(\beta\) are false, and true otherwise.
- The sentence \(\alpha \Rightarrow \beta\) is false if \(\alpha\) is true and \(\beta\) is false, and true otherwise.
- The sentence \(\alpha \equiv \beta\) is true if both \(\alpha\) and \(\beta\) have the same truth value (i.e., they are either both true or both false), and false otherwise.

Thus, for the case when \(P\) is true and \(Q\) is false, the sentence

\((P \land \neg Q) \Rightarrow \text{False}\)

will evaluate to false since according to the above rules:

- \(Q\) will evaluate to false,
- \(\neg\text{false}\) will evaluate to true,
- \(P\) will evaluate to true,
- \(\text{true} \land \text{true}\) will evaluate to true,
- \(\text{False}\) will evaluate to false, and
- \(\text{true} \Rightarrow \text{false}\) will evaluate to false.

Once the knowledge has been expressed as a set of sentences in propositional calculus, new sentences can be deduced from them using rules of inference. Some example rules of inference are given in Fig. 1. One of these rules is modus ponens, which from two sentences of the form \(\alpha \Rightarrow \beta\) and \(\alpha\) permits us to derive \(\beta\). This is one of the most commonly employed rules in our daily reasoning processes.

Besides the rules of inference, one can also make use of what are known as the laws of propositional algebra. These laws are of the form \(\alpha \equiv \beta\) meaning that the expressions on both sides of the ‘≡’ symbol are equivalent. Thus, one expression can be freely substituted in place of the other and new sentences can be formulated. A partial list of such laws is given in Fig. 2. (A fuller list can be found in Ref. [Amble 87].)
1. *Modus Ponens*: From two sentences of the form
   \[ \alpha \implies \beta \quad \text{and} \quad \alpha \]
   one can infer the sentence \( \beta \)

2. *Modus Tollens*: From two sentences of the form
   \[ \alpha \implies \beta \quad \text{and} \quad \neg \beta \]
   one can infer the sentence \( \neg \alpha \)

3. *And Introduction*: From two sentences of the form
   \[ \alpha \quad \text{and} \quad \beta \]
   one can infer the sentence \( \alpha \land \beta \)

4. *And Elimination*: From a sentence of the form
   \[ \alpha \land \beta \]
   one can infer the two sentences:
   \[ \alpha \quad \text{and} \quad \beta \]

Figure 1: Sample Rules of Inference

1.1.3. Example

The use of propositional calculus for reasoning is illustrated below with a simple, though contrived, engineering example. We start with the following two axioms.

- If the building has a moment resisting frame and the lateral deflection at the top under the lateral loads is too large, then increase the stiffness of columns or increase the stiffness of beams.
- If the flexural stiffness of columns is much larger than the flexural stiffness of beams, then do not increase the stiffness of columns.

Also, let us assume that for a given building the following is true.

- The building has a moment resisting frame.
- The lateral deflection at the top under the lateral loads is too large.
- The flexural stiffness of columns is much larger than the flexural stiffness of beams.

From the above statements, the reader can derive that the recourse for reducing the lateral deflection in the given case is to increase the stiffness of the beams. We can reach the same conclusion through syntactic manipulations with propositional calculus. In the reasoning process with propositional calculus, we make use of the rules of inference and the laws of propositional algebra. The following symbols are used for the propositions in the proof.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>The building has a moment resisting frame.</td>
</tr>
<tr>
<td>L</td>
<td>The lateral deflection at the top under the lateral loads is too large.</td>
</tr>
<tr>
<td>F</td>
<td>The flexural stiffness of columns is much larger than the flexural stiffness of beams.</td>
</tr>
<tr>
<td>B</td>
<td>Increase the stiffness of the beams.</td>
</tr>
<tr>
<td>C</td>
<td>Increase the stiffness of the columns.</td>
</tr>
</tbody>
</table>

Using these symbols, the given statements can be expressed as follows.

1. \((M \land L) \implies (B \lor C)\)
2. \(F \implies \neg C\)
1. Commutativity Laws
\[ \alpha \lor \beta \equiv \beta \lor \alpha \]
\[ \alpha \land \beta \equiv \beta \land \alpha \]

2. Distributivity Laws
\[ \alpha \land (\beta \lor \gamma) \equiv (\alpha \land \beta) \lor (\alpha \land \gamma) \]
\[ \alpha \lor (\beta \land \gamma) \equiv (\alpha \lor \beta) \land (\alpha \lor \gamma) \]

3. Idempotence Laws
\[ \alpha \lor \alpha \equiv \alpha \]
\[ \alpha \land \alpha \equiv \alpha \]

4. Complementation Laws
\[ \alpha \lor \neg \alpha \equiv \text{True} \]
\[ \alpha \land \neg \alpha \equiv \text{False} \]

5. Annihilator Laws
\[ \text{True} \lor \alpha \equiv \text{True} \]
\[ \text{False} \land \alpha \equiv \text{False} \]

6. Identity Elements
\[ \alpha \lor \text{False} \equiv \alpha \]
\[ \alpha \land \text{True} \equiv \alpha \]

7. DeMorgan's Laws
\[ \neg (\alpha \land \beta) \equiv (\neg \alpha) \lor (\neg \beta) \]
\[ \neg (\alpha \lor \beta) \equiv (\neg \alpha) \land (\neg \beta) \]

8. Cancellation of Negations
\[ \neg (\neg \alpha) \equiv \alpha \]

**Figure 2:** Sample Laws of Propositional Algebra

---

3. M
4. L
5. F

From these statements, the following can be deduced with the aid of the rules of inference and the laws of propositional algebra (mentioned within parentheses for each statement).

6. \( M \land L \) (And Introduction: 3, 4)
7. \( B \lor C \) (Modus Ponens: 6, 1)
8. \( \neg C \) (Modus Ponens: 2, 5)
9. \( (B \lor C) \land (\neg C) \) (And Introduction: 7, 8)
10. \( (B \land \neg C) \lor (C \land \neg C) \) (Distributivity: 9)
11. \( (B \land \neg C) \lor \text{False} \) (Complementation: 10)
12. \( B \land \neg C \) (Identity: 11)
13. \( B \) (And Elimination: 12)

Thus, the desired conclusion (increase the stiffness of the beams) has been reached. The proof may seem cumbersome, but we have employed only the most primitive of algebraic manipulations. Higher level manipulations can also be formulated and shown to be valid using the basic laws. Thus, for instance, we could have used the *Disjunctive Argument* rule, a rule which from \( \alpha \lor \beta \) and \( \neg \beta \) allows us to infer \( \alpha \). Using this particular rule, the result \( B \) could have been

---

3The distinction between the rules of inference and the laws of algebra of propositions is rather arbitrary. For instance, the cancellation of negations law could have been formulated as the double negation rule which permits one to deduce \( \alpha \) from a sentence of the form \( \neg (\neg \alpha) \) and vice versa. Similar statements can be made about DeMorgan's and other laws.
obtained in a single step from statements 7 and 8.

1.2. Predicate Calculus

Although propositional calculus is a useful language for expressing simple concepts, it is usually inadequate for expressing many facts about the real world. The language is too coarse and primitive to express the concept of an object, properties of an object, or relationships among several objects [Manna and Waldinger 85]. These shortcomings arise because the propositions in propositional calculus are a unit as a whole and not decomposable entities. Thus, if the symbol P1 represents the proposition “Frame A has 5 bays” and P2 represents “Frame B has 3 bays,” one still cannot infer that Frame A has more bays than Frame B. In predicate calculus, the number of bays can be expressed as an attribute of each of the objects Frame A and Frame B and it is possible to reason about such an attribute. Predicate calculus enhances the expressibility of propositional calculus by introducing additional concepts such as predicates, functions, variables, and quantifiers.

1.2.1. Syntax

There is a greater variety of symbols available in predicate calculus as compared to propositional calculus. The basic truth symbols are still the same: True and False. In addition, there are constant symbols or constants (e.g., A5, Red, 123, FrameB, Delaware), variable symbols or variables (e.g., x, y, x1, y1), function symbols or functions (e.g., Sin, Distance, Age, Log), and predicate symbols or relations (e.g., Even, Taller, Neighbor). Constants, variables, and functions correspond well with the similar notions in algebra. As the name implies, constants denote those elements whose value never changes. Variables, on the other hand, denote the elements that can assume different values at different instances. Function symbols are used to denote operators or functions which operate on a fixed number of arguments and evaluate to a legal value. Predicate symbols denote relations that hold among a fixed number of arguments.

We adopt the following commonly used notational convention. A constant symbol is a sequence of alphanumeric characters in which the first character is either numeric or uppercase alphabetic. A variable symbol is a sequence of alphanumeric characters in which the first character is lowercase alphabetic. Function symbols can either be a sequence of alphanumeric characters in which the first character is uppercase alphabetic, or one of the following functional operators: +, -, *, and /. Predicate symbols can either be a sequence of alphanumeric characters in which the first character is uppercase alphabetic, or one of the following mathematical operators: =, <, >, ≤, and ≥. In predicate calculus, a couple of punctuation marks are needed as well: (i) commas to separate the multiple arguments of a function or a predicate, and (ii) parentheses to enclose such arguments.

Functions and relations have associated arities (the number of arguments the function or the relation requires) which are always positive integers. The arguments that functions and relations take are called terms. A term can be either a constant, a variable, or a functional expression
where a functional expression consists of a function symbol of arity $n$ followed by a list of $n$ terms enclosed in parentheses and separated by commas. The following are legal functional expressions:

$$\text{MomentOfInertia(section,YY)}$$
$$\text{Sqrt(Fy)}$$
$$/(640,\text{Sqrt(Fy)})^4$$

(Note that the functional expressions themselves can serve as arguments to other functions.) These functional expressions and the constants and variables such as $A5$, $123$, and $y_1$ are all legal examples of terms.

From these terms, sentences can be formed. A sentence can be of one of three types: atomic sentence, logical sentence, or quantified sentence. An atomic sentence can either be a truth symbol or a predicate symbol of arity $n$ followed by a list of $n$ terms enclosed in parenthesis and separated by commas. Examples of atomic sentences are:

$$\text{NumberOfBays(FrameA,5)}$$
$$>(x,0).^5$$

A logical sentence is formed by combining sentences using logical connectives such as and ($\land$), or ($\lor$), not ($\neg$), implies ($\Rightarrow$), and equivalent ($\equiv$). The types of sentences that can be formed with these connectives are the same as those for propositional calculus. An example of a logical sentence is:

$$\text{NumberOfBays(FrameA,5)} \land \text{NumberOfBays(FrameB,3)}$$

A quantified sentence can either be a universally quantified or an existentially quantified sentence. A universally quantified sentence consists of the universal quantifier, for all ($\forall$), a variable, followed by a sentence. For instance, the sentence

$$\forall y \text{ Partition}(y) \Rightarrow \text{PotentialColLine}(y)$$

is a universally quantified sentence. The intended meaning of the sentence is that every partition is a potential column line. An existentially quantified sentence consists of the existential quantifier, for some ($\exists$), a variable, followed by a sentence. For instance, the sentence

$$\exists x \text{ Eccentricity}(Load,x) \land \neg(x=0)$$

is an existentially quantified sentence. The intended meaning of the sentence is that there is a load with non-zero eccentricity.

For clarity, parentheses and brackets are used to specify the scope of quantifiers. A sentence that uses both universal and existential quantifiers is the following.

---

^4For readability purposes, the infix notation is sometimes preferred in the case of common mathematical functions; for instance, ($x/y$), or $x/y$, may be used in place of $/(x,y)$.

^5For readability purposes, the infix notation is sometimes preferred in the case of common mathematical relations; for instance, ($x>y$), or $x>y$, may be used in place of $>(x,y)$.

^6Also termed as there exists.
\[\forall \text{line} \left( \text{PotentialColLine}(\text{line}) \land \neg \exists \text{otherLine} \left( \text{ColLine}(\text{otherLine}) \land \neg (\text{line} = \text{otherLine}) \land \text{Distance}(\text{otherLine}, \text{line}) < \text{MinDistance}) \right) \Rightarrow \text{ColLine}(\text{line}) \]

The intended meaning of the sentence is that all potential column lines which are no closer than a certain minimum distance to any other column line, become column lines themselves.

A language in which functions and predicates are constants is called a \textit{first-order} language. A language in which functions and predicates also vary is called \textit{second-order}. In a second-order language, functions and relations can serve as quantifying variables to quantifiers. In this report, however, we will restrict ourselves to first-order predicate calculus.

\section*{1.2.2. Semantics}

Now that the formulation of sentences in predicate calculus has been discussed, let us examine how their truth values can be evaluated. For this purpose, however, introduction of some additional notions—such as interpretation and variable assignment—is necessary. We first examine the truth or falsehood of an atomic sentence. Later in this section we discuss evaluation of truth values of logical and quantified sentences.

The motivation for formulating sentences in logic is to conceptualize a world about which one is trying to reason. In the process, certain symbols are used to represent the elements of the world (for the simple reason that it is often not possible to substitute the actual elements—which can be physical and conceptual entities—in the text). Thus, in the example in Section 1.1.3, we expressed certain conceptual ideas about the world in terms of single letter propositional symbols. An \textit{interpretation} maps the elements of the language to the elements of the world that one is trying to conceptualize. Interpretation is similar to dereferencing the symbols—bringing into consideration the object the symbol stands for rather than the symbol itself. Indeed, the idea of interpretation is so natural to us humans that we do it frequently without even being aware of it. Thus, when we say that an I-beam has three parts (two flanges and one web), we mean that the members of the category of objects represented by the word \textit{I-beam} each have three parts, rather than that the word \textit{I-beam} has three parts.

With reference to predicate calculus, an interpretation over a domain (a set of objects, functions, and relations) provides a mapping for the constant symbols, the function symbols, and the predicate symbols. To each constant symbol in the language, the interpretation assigns an object in the domain. For each function symbol of arity \(n\), the interpretation assigns an \(n\)-ary function defined over the objects in the domain. A functional expression evaluates to a value which itself is an object of the domain. For instance, the function symbol \texttt{Abs} of arity 1 can be assigned the unary mathematical function ‘absolute’ over the set of real numbers. The value of \texttt{Abs}(x) will always evaluate to a real number for all values of \(x\). For each predicate symbol of arity \(n\), the interpretation assigns an \(n\)-ary relation defined over the objects in the domain such that the value of an atomic sentence composed of the relation and \(n\) terms always evaluates to either \textit{true} or \textit{false} depending upon whether the relation holds for the specified objects or not. For instance,
the predicate symbol $\geq z$ of arity 1 can be assigned, over the set of real numbers, a relation ‘greater-than-zero’ which is true for all positive numbers and false for nonpositive numbers. The value of $\geq z(x)$ will always evaluate to either true or false for all values of $x$.

Analogous to the concept of interpretation is the concept of variable assignment. A variable assignment maps the free variables in the language to the elements of the domain. Free variables in a sentence are those variables that are not within the scope of any quantifier.

When we write sentences, we have a certain intended interpretation in mind, that is, we associate the elements of the language with specific elements in the world being conceptualized. For instance, in the intended interpretation in the example of Section 1.1.3, the symbol $B$ represented the concept that the stiffness of the beams should be increased. But it is equally acceptable to assign a totally different interpretation (as well as variable assignment) over an entirely different domain to the same set of sentences. To illustrate, let us consider the sentence

$$\forall y \ P(y) \Rightarrow Q(F(y), z)$$

under two different interpretations. The sentence has two predicate symbols ($P$ and $Q$), one function symbol ($F$), and one free variable ($z$). Let us first consider the following interpretation and variable assignment over the domain of steel sections and real numbers.

- $P$ is ‘rolled-section’ relation,
- $Q$ is ‘greater-than’ relation,
- $F$ is ‘cross-sectional-area’ function, and
- $z$ is ‘0.’

The given sentence intuitively transforms to:

the cross-sectional area of all rolled-sections is greater than zero.

This statement turns out to be true. However, consider another interpretation and variable assignment, this time over the domain of beams.

- $P$ is ‘simply-supported-beam’ relation,
- $Q$ is ‘is-a’ relation,
- $F$ is ‘left-support’ function, and
- $z$ is ‘roller.’

The intuitive meaning of the sentence becomes:

the left support of all simply supported beams is a roller

which certainly is false.

In general, the truth of a sentence depends on the particular interpretation and the particular variable assignment being employed. A sentence may be true under some interpretation and variable assignment and false under a different interpretation and variable assignment. Normally, symbols for predicates and functions are so chosen that they correspond well with what they represent in the intended interpretation. (We rely on such practice throughout the report. Our prime motive, however, is brevity—instead of stating every time ‘consider the interpretation over the domain ... where predicate and function symbols ... and ... represent ... and ... respectively, and consider the variable assignment ...’ we instead appeal to intuition to convey the meaning of symbols involved.) However, as the above example illustrates, it is
important to distinguish between the symbols and the referents.

The preceding paragraphs define the semantics of atomic sentences. The truth values of logical sentences can be determined by following the rules for logical connectives presented earlier in Section 1.1.2. The truth values of quantified sentences can be ascertained as follows. A universally quantified sentence $\forall \mu \alpha$ is true under a given interpretation if and only if the sentence $\alpha$ is true for all variable assignments of $\mu$. An existentially quantified sentence $\exists \mu \alpha$ is true if and only if there exists a variable assignment for $\mu$ for which the sentence $\alpha$ is true. In essence, the universal quantifier enables us to express some property which is true of all the objects in the domain while the existential quantifier enables us to express some property of an individual object without specifying the object. For example, consider the sentence

$$\forall x \text{TrussMember}(x) \Rightarrow \text{BendingMoment}(x,0)$$

With this sentence we have expressed that the bending moment in all the truss members is always zero, without enumerating any of the truss members. Similarly, we used existential quantifier earlier to express that a load has certain eccentricity without specifying the eccentricity.

A sentence is termed satisfiable if and only if there exists an interpretation and variable assignment for which the sentence is true. Otherwise, the sentence is unsatisfiable (or contradictory). If a sentence is satisfied by every interpretation and variable assignment, it is termed as a valid sentence. For example, the propositional sentence $P \lor \neg P$ is a valid sentence (it will always evaluate to true irrespective of the truth value of $P$). In a manner analogous to individual sentences, these definitions can be extended to sets of sentences. In particular, a set of sentences is satisfiable or consistent if and only if for some interpretation and variable assignment, all the sentences in the set are true. Otherwise, the set is unsatisfiable or inconsistent.

1.2.3. Inference

The rules of inference listed in Fig. 1 are applicable to predicate calculus as well. In addition, there are some more rules of inference for predicate calculus. One such rule is the universal instantiation rule, which is quite intuitive. According to this rule, from a universally quantified sentence one can infer any other sentence that has been obtained by replacing the universally quantified variable with a suitable term. For example, from

$$\forall x \text{TrussMember}(x) \Rightarrow \text{BendingMoment}(x,0)$$

the following (and many other such sentences) can be inferred.

$$\text{TrussMember}(\text{AB}) \Rightarrow \text{BendingMoment}(\text{AB},0)$$
$$\text{TrussMember}(\text{TopChord(Truss1)}) \Rightarrow \text{BendingMoment}(\text{TopChord(Truss1)},0)$$

The term used for replacing the universally quantified variable can be a constant, a functional expression, or a variable. However, if the variable is a free variable or the functional expression contains free variables, it has to be ensured that the names of such free variables do not match with the names of other variables already present in the sentence.

The existential instantiation rule is similar to the universal instantiation rule and is applicable to existentially quantified sentences. The restrictions on the terms that can be used for replace-
Another rule of inference is resolution. To understand resolution, it is necessary to first understand two other concepts: clausal form and unification. A detailed description of clausal form and conversion of ordinary sentences to clausal form can be found in Ref. [Genesereth and Nilsson 87]. For our purposes, it suffices to say that any sentence formed following the rules of syntax for predicate calculus described earlier can be equivalently represented in a simpler form called clausal form which consists of clauses. A clause is a set of literals (atomic sentences or their negations) that represents the disjunction of such literals. For example, the clause

\[ \neg \text{TrussMember}(x) \lor \text{BendingMoment}(x,0) \]

is equivalent to the sentence

\[ \neg \text{TrussMember}(x) \lor \text{BendingMoment}(x,0) \]

The reader can find the procedure for converting an ordinary sentence into the clausal form in the book by Genesereth and Nilsson (1987).

Unification in some sense is like pattern matching. We say that two expressions can be unified if they can be made identical by replacing some or all of their variables with other terms. For instance, the two expressions

\[
\begin{align*}
\text{BeamSupport}(\text{Beam1,Left, supportingBeam}) & \quad \text{and} \\
\text{BeamSupport}(\text{someBeam, Left, Beam5})
\end{align*}
\]

can be unified by replacing supportingBeam by Beam5 in the first expression and someBeam by Beam1 in the second. Such replacements, or substitutions, for variables (written, for example, as \([\text{supportingBeam} \leftarrow \text{Beam5}, \text{someBeam} \leftarrow \text{Beam1}]\)) that unify two expressions are called unifiers. There may be more than one unifier for two expressions that are unifiable. In such a case, a most general unifier (mgu) is one from which other unifiers can be obtained after appropriate substitutions.

After this introductory background, we can now proceed to discuss the resolution principle.\(^7\) The resolution principle can be applied to any two clauses, \(\Theta\) and \(\Omega\), if one clause contains a positive literal, \(\alpha\), while another contains a negative literal, \(\neg \beta\), such that \(\alpha\) and \(\beta\) can be unified by a most general unifier \(\nu\). If these conditions are met, then from the two clauses \(\Theta\) and \(\Omega\) we can infer another clause which is obtained by applying the substitution \(\nu\) to the union of \(\Theta\) and \(\Omega\) minus \(\alpha\) and \(\neg \beta\). Mathematically, this can be written as

\[
\begin{align*}
\text{From} & \quad \Theta & (\alpha \varepsilon \Theta) \\
\text{and} & \quad \Omega & (\neg \beta \varepsilon \Omega) \\
\text{infer} & \quad [(\Theta - \{\alpha\}) \cup (\Omega - \{\neg \beta\})] \leftarrow \nu & \text{where } \alpha \leftarrow \nu = \neg \beta \leftarrow \nu \\
\text{(The symbol \(\leftarrow\) represents the substitution operator.)}
\end{align*}
\]

As an illustration of application of resolution, consider the following two clauses.

\(^7\)The definition here does not include the notion of factors. For a more general definition see Ref. [Genesereth and Nilsson 87].
\{-\text{TrussMember}(x), \text{BendingMoment}(x,0)\}
\{-\text{TrussMember}({\text{TopChord(Truss1)})}\}

The mgu \([x \leftarrow \text{TopChord(Truss1)}]\) will unify the two literals \text{TrussMember}(x) and \text{TrussMember}(\text{TopChord(Truss1)})]. Thus, the two clauses above will resolve to produce
\{-\text{BendingMoment}(\text{TopChord(Truss1)},0)\}

The resolution principle is quite powerful. Logic programming languages, which we shall discuss shortly, exploit the generality and the power of resolution to deduce facts and, in the process, instantiate variables (using the coupled notion of unification). In many cases, unifiers are of prime importance as the problem solver may be interested in the values of variables that will satisfy certain conditions. In other cases, one may be interested in finding out if a statement logically follows from a consistent set of statements. In such instances, the statement to be proved is negated and is added to the existing fact base of clauses. Using resolution, if the empty clause (equivalent of a \textit{false} statement), \{\}, can be deduced, it will mean that the set of sentences has become unsatisfiable because of the additional negation—implying that the original statement does indeed follow from the original fact base. This process is called \textit{resolution refutation}.

How can one be sure that the conclusions derived from a set of sentences using a particular rule of inference will indeed be correct (or will logically be implied by the set of sentences)? How can one be sure that \textit{all} the conclusions that logically follow from a given set of sentences can be derived using a particular rule of inference? In logicians' parlance, these properties of inference rules are known as \textit{soundness} and \textit{completeness}, respectively. All the rules of inference given in this report are sound but none is complete. Resolution, though, has an important property of being \textit{refutation complete}, i.e., if the given set of sentences is unsatisfiable, the empty clause can always be deduced using resolution alone.

1.3. Logic Programming

Logic programming is a form of declarative programming based on predicate calculus. Logic programs are essentially a set of specifications; they emphasize the \textit{what} of the solution rather than the \textit{how}. The programmer specifies (in terms of predicate calculus-like sentences) the conditions that the acceptable solution(s) must meet and it is the task of the tool to find such solution(s). Performance design standards are an example of knowledge of this form. Such standards specify the restrictions on the acceptable designs rather than the process of design. Thus it would appear that standards processing will be a natural metaphor for logic programming. Indeed, one such effort [Jain, Law, and Krawinkler 89] is underway to exploit the power of logic programming for processing design standards in structural engineering.

Admittedly, the choice of tools available for logic programming is more limited than the range of tools for some other representational formalisms such as rule-based and frame-based systems. However, the options available are powerful and quite capable of meeting the needs of serious programmers. In the following subsections, we discuss three such tools. Because symbols like \(\land\), \(\neg\), \(\forall\), etc., are typically not available on a keyboard, and because of other reasons of implementation, the syntax in these tools is a variant of the one described in Section 1.2.1 and
not identical.

1.3.1. Prolog

Of all the logic programming languages, Prolog (from programming in logic) unquestionably is the most popular. One programs in Prolog using a combination of atomic sentences and rules. The rules are restricted to the statements that can be expressed as Horn Clauses—clauses that have at most one positive literal. The notational convention regarding the upper and lowercase in Prolog is, however, different from the one adopted in Section 1.2.1. Variable symbols in Prolog begin with an uppercase letter while constant and predicate symbols begin with a lowercase letter. The following are some acceptable statements in the syntax of Prolog.

\[
\text{numberOfBays(frameA, 5).} \\
\text{numberOfBays(frameB, 3).} \\
\text{moreBays(Frame1, Frame2) :- numberOfBays(Frame1, X),} \\
\text{numberOfBays(Frame2, Y), X > Y.}
\]

The symbol ‘:-’ is read as if and is like an implication in the reverse direction. The period symbol ‘.’ denotes the end of a statement. In a rule, the symbol ‘,’ between atomic sentences (between \text{numberOfBays(Frame1, X)} and \text{numberOfBays(Frame2, Y)}, for example) signifies their conjunction. The or connective is specified by the symbol ‘;’ while not by the symbol ‘not’. Parentheses can be used to clarify the scope of connectives. The third statement above is thus equivalently stated in our syntactical notation as

\[
\text{(NumberOfBays(frame1,x)∧NumberOfBays(frame2,y)∧x>y) } \\
\rightarrow \text{MoreBays(frame1,frame2)}
\]

Common mathematical functions (such as +, −, *, and /) and relations (such as =, <, and >) are available in Prolog to facilitate processing of numbers. A programmer creates a fact base containing atomic sentences and rules that meet the syntactic requirements described herein. The facts in the fact base can contain variables; such variables are taken to be universally quantified. Thus, in the example rule given earlier, Frame1, Frame2, X, and Y are all universally quantified variables.

Once the fact base is created, one can query the fact base. Continuing with the example above, the query \text{numberOfBays(frameA, HowMany)}., for instance, will resolve with the fact \text{numberOfBays(frameA, 5)}., resulting in the unifier [HowMany = 5]. The query, \text{numberOfBays(Frame, HowMany)}, on the other hand, will result in two possible unifiers: [Frame = frameA, HowMany = 5] and [Frame = frameB, HowMany = 3]. Variables in queries are taken to be existentially quantified—if the query can unify with some fact in the fact base for at least one instance of variables, the query is successful; otherwise it is unsuccessful.

We can also query \text{moreBays(frameA, frameB)}., which will cause Prolog to prove the goals \text{numberOfBays(frameA, X)}, \text{numberOfBays(frameB, Y)}, and \text{X > Y} in that order. On proving \text{numberOfBays(frameA, X)}, the unifier will be [X = 5] while the second goal will result in the unifier [Y = 3]. The third goal, now reduced to 5 > 3, will succeed, resulting in \text{yes} (a sign of success of the query) along with the combined unifier [X = 5, Y = 3]. The query \text{moreBays(frameB, frameA)}., on the other hand, will be unsuccessful.
(since for no instance of the variables X and Y will all three goals succeed simultaneously) and result in no. This example illustrates how Prolog proceeds to prove each of the goals in a rule in the order in which they are present. If some goal fails, Prolog backtracks to a previous goal and starts with a different substitution for one or more variables. In matching the query with the facts in the fact base, Prolog starts at the top of the fact base and proceeds down until some atomic sentence or the part to the left of ' :- ' in a rule can unify with the query.

We will illustrate more details of Prolog in the example problem described in Section 3. In particular, the list data structure, anonymous variables, recursion, extra-logical facilities, metalogical predicates, and second-order programming will be illustrated. For an excellent treatment of the Prolog language and its underlying foundations, the reader should consult Ref. [Sterling and Shapiro 86].

1.3.2. Epikit

Epikit [Epistemics 88] is a tool for knowledge representation and automated reasoning. For representing knowledge, Epikit employs KIF (Knowledge Interchange Format) language which is an extension of first-order predicate calculus. An important feature of Epikit is the meta-level facility which permits one to state how the facts in the knowledge base should be used. Thus actions of inference procedures can be declaratively specified without embedding the control knowledge in base-level statements (as opposed to Prolog in which a programmer often orders the rules and goals within a rule to incorporate peculiarities of the interpreter).

A wide spectrum of inference procedures and control strategies is available in Epikit. The inference procedures include ordered resolution, general resolution, unit resolution, dynamically ordered resolution, and user supplied inference procedures. The control strategies include depth first, breadth first, best first, and iterative deepening.

1.3.3. A Nonclausal Connection-Graph Resolution Theorem Prover

As the name implies, this theorem prover also uses resolution for making conclusions from a given fact base. However, instead of the clausal resolution we have described, this tool uses nonclausal resolution as the inference procedure leading to easy readability of statements and a potential reduction in the resolution search space [Stickel 82]. At any stage in a resolution deduction, there may be many clauses which will resolve with each other, implying that there can be more than one choice for the resolution to be carried out in a particular step. A connection-graph is a graphical representation of syntactically acceptable sentences and links among such sentences. The links denote permitted resolution operations among the sentences joined by such links. Using a graph-searching algorithm, one can traverse the connection-graph to plan which deductions to perform in order to construct a proof.

An important feature of this tool is the use of special connectives to impose restrictions on the use of particular assertions. For instance, instead of using the unrestricted implication \( \Rightarrow \), one can use the forward arrow \( \rightarrow \) to signify that the assertion can only be used for forward chaining, or can use the back arrow \( \leftarrow \) to signify that the assertion can only be used for back-
ward chaining. The input to the theorem prover is in the form of predicate calculus sentences. The program allows for predicate variables and thus is not strictly within the realm of first-order predicate calculus.

2. An Engineering Example

In this section, we illustrate an example of structural analysis in the context of the concepts presented earlier. We contrast the declarative approach to the solution of the problem with the procedural approach. The problem is a part of a larger problem discussed in Ref. [Fenves 79] and involves determining reactions at the ends of simply supported beams that comprise an idealized floor system. The information about the floor geometry and loading is provided to the program as input. The floor itself is composed of various rectangular areas that are loaded with uniformly distributed area loads and behave as one-way slabs, transferring their accumulated loads to two parallel supporting beams (specified by the engineer in the input) at the edges. The beams lie on a rectangular grid and form a hierarchy, transferring loads through the beams lower in the hierarchy ultimately to the columns. One sample floor layout is shown in Fig. 3. The problem appears to be trivial, but its solution requires knowledge about beam hierarchy since the computation of end reactions of a beam can proceed only if the reactions of all other beams supported by the beam under consideration have been determined.

![Figure 3: A Sample Floor Layout](image)

To appreciate the difference between the procedural approach and the declarative approach, first consider the following abstract procedure (adapted from Ref. [Fenves 79]) that can be used
to solve the problem.

**Convert Area Loads**
For each area load do
locate edge beams under area from grid
assign a line load to supporting beams

**Carry Down Loads**
For each beam do
initialize counter to 0
For each beam do
increase the counters of the left and right supports by 1
repeat
For each beam do
if counter = 0 then
    place beam designation into the next location of a stack
    decrease the counters of the left and right supports by 1
until all beams are in stack

**Solve for End Reactions**
while stack is not empty
    pop a beam from the top of the stack
    compute its end reactions
    assign the end reactions as point loads to underlying beams
    (or columns)

Effectively, the algorithm first explicitly determines the sequence in which the beams should be solved for end reactions, and then proceeds to determine the reactions. However, in solving problems of this type, a structural engineer does not typically determine such a sequence beforehand but relies on the following fact:

- If the end reactions of all the beams that a beam under consideration supports have been computed then the supporting beam can be solved.

Other knowledge that is needed for the solution to this problem is the following:

- Each of the two beams acting as supports to an area carries a uniformly distributed line load (udll) of intensity equal to half of the area's dimension in the direction perpendicular to supporting beams times the loading intensity on the area.

- The extent (portion of the span) of loading on the beam resulting from such area loads coincides with that portion of the beam which corresponds to the edge of the area.

- For purposes of computing end reactions on a simply supported beam, a udll can be treated as an equivalent point load acting at the midpoint of the udll and having a magnitude equal to intensity times the extent of udll.

In addition, laws of static equilibrium are also needed to determine the end reactions of a beam.

In contrast to the procedural approach, we will encode the above pieces of knowledge as declarative statements in predicate calculus and reason from them. However, before the statements can be formulated in predicate calculus syntax, we must decide on a conceptualization; for instance, we must choose what relations and functions will be used for presenting the information about the floor geometry and loading. One possibility is the following.

AreaLoad(areaName,loadIntensity)
AreaSupport(areaName,Left,suptgBeamOnLeft)
AreaSupport(areaName,Right,suptgBeamOnRight)
AreaEdges(areaName,xLeft,yLeft,xRight,yRight)
BeamSupport(beamName, Left, leftSupport)
BeamSupport(beamName, Right, rightSupport)
BeamEnds(beamName, xLeft, yLeft, xRight, yRight)
UDLLoad(beamName, from, to, loadIntensity)
PtLoad(beamName, location, magnitude)

The importance of proper conceptualization cannot be overemphasized. The one shown above is not unique; there are many others. However, with experience, one develops intuition and judgement as to which ones are natural as well as efficient for reasoning. For the area support information, for instance, we could have chosen relations like AreaSupportLeft and AreaSupportRight, eliminating one of the arguments. These relations, however, are undesirable because they embed the semantics of arguments in the predicate names where they are not amenable for manipulation.

Using the proposed choice of relations for presenting the floor geometry and loading information, the items of knowledge presented earlier can be formulated as the following predicate calculus sentences. Note that there is a direct correspondence between the English sentences and the predicate calculus sentences.

A beam is solvable if it has not already been solved and there does not exist another unsolved beam that is supported by this beam.

\[\neg \text{Solved}(\text{beam}) \land
\neg \exists \text{higher}(\text{BeamSupport}(\text{higher}, \text{direction}, \text{beam}) \land \neg \text{Solved}(\text{higher}))\]
\[\Rightarrow \text{Solvable}(\text{beam})\]

If a beam supports an area on which there is certain loading, and the extent and intensity of loading resulting on the supporting beam have been determined, then the beam can be assigned a udll of such intensity and extent.

\[\text{AreaSupport}(\text{area}, \text{side}, \text{beam}) \land \text{AreaLoad}(\text{area}, \text{load}) \land
\text{LoadingExtents}(\text{area}, \text{beam}, \text{from}, \text{to}, \text{leftEdge}, \text{rightEdge}) \land
(\text{intensity} = \text{load} \times \text{Abs}(\text{rightEdge-} \text{leftEdge})/2)\]
\[\Rightarrow \text{UDLLoad}(\text{beam}, \text{from}, \text{to}, \text{intensity})\]

If the y coordinates of the two ends of a beam are the same (or, in other words, the beam is parallel to the x-axis) then the x coordinates of the two edges of the area that are perpendicular to the beam can be used to determine the extent of the loading.

\[\text{BeamEnds}(\text{beam}, x\text{Left}, y, x\text{Right}, y) \land \text{AreaEdges}(\text{area}, x1, y1, x2, y2)\]
\[\Rightarrow \text{LoadingExtents}(\text{area}, \text{beam}, x1, x2, y1, y2)\]

If the x coordinates of the two ends of a beam are the same (or, in other words, the beam is parallel to the y-axis) then the y coordinates of the two edges of the area that are perpendicular to the beam can be used to determine the extent of the loading.

\[\text{BeamEnds}(\text{beam}, x, y\text{Left}, x, y\text{Right}) \land \text{AreaEdges}(\text{area}, x1, y1, x2, y2)\]
\[\Rightarrow \text{LoadingExtents}(\text{area}, \text{beam}, y1, y2, x1, x2)\]

A udll is equivalent to a point load of magnitude equal to intensity times span and location at the midpoint of the extent of udll.
UDLLoad(beam,from,to,intensity)  
⇒PtLoad(beam,(from + to)/2,intensity * Abs(to - from))

As shown in the next section, these statements can be translated into Prolog syntax in a fairly straightforward manner. In the actual program, however, one also has to do such mundane things as display the results, assign the end reactions of a beam as point loads to supporting beams, and keep track of the beams whose end reactions have been determined (to avoid duplicate processing). Accordingly, the program presented in the next section consists of more than the five statements listed above.

3. A Representative Implementation

In Prolog syntax, the first four predicate calculus statements of the previous section can be expressed as follows. (The fifth statement requires additional concepts and is presented later.)

\[
\text{solvable(} \text{Beam}) : -
\]
\[
\text{not solved(} \text{Beam}),
\]
\[
\text{not (beamSupport(} \text{Higher}, _, \text{Beam}),
\]
\[
\text{not solved(} \text{Higher})).
\]

\[
\text{areaLoadAssigned(} \text{Area}, \text{Side}) : -
\]
\[
\text{areaSupport(} \text{Area}, \text{Side}, \text{Beam}), \text{areaLoad(} \text{Area}, \text{Load}),
\]
\[
\text{loadingExtents(} \text{Area}, \text{Beam}, \text{From}, \text{To}, \text{LeftEdge}, \text{RightEdge}),
\]
\[
\text{abs(} \text{RightEdge} - \text{LeftEdge}, \text{Span}),
\]
\[
\text{Intensity is Load} \times \text{Span} / 2,
\]
\[
\text{assert(} \text{udlLoad(} \text{Beam}, \text{From}, \text{To}, \text{Intensity})\).
\]

\[
\text{loadingExtents(} \text{Area}, \text{Beam}, \text{From}, \text{To}, \text{LeftCoord}, \text{RightCoord}) : -
\]
\[
\text{beamEnds(} \text{Beam}, _, \text{Y}, _, \text{Y}),
\]
\[
\text{areaEdges(} \text{Area}, \text{From}, \text{LeftCoord}, \text{To}, \text{RightCoord}).
\]

\[
\text{loadingExtents(} \text{Area}, \text{Beam}, \text{From}, \text{To}, \text{LeftCoord}, \text{RightCoord}) : -
\]
\[
\text{beamEnds(} \text{Beam}, \text{X}, _, \text{X}, _),
\]
\[
\text{areaEdges(} \text{Area}, \text{LeftCoord}, \text{From}, \text{RightCoord}, \text{To}).
\]

Note that there are a few minor modifications in the sentences above as compared to the format of sentences in Section 2. These are to accommodate the functioning of the Prolog interpreter. In particular, the meta-logical predicate assert mentioned in the second statement adds its argument to the current fact base. Thus, besides the facts contained in the program and data files, any facts added to the fact base by assert will also be used for potential unification with subsequent queries.

Another modification is the use of underscores ( _ ). Underscores in a Prolog statement provide anonymous variables—they can be used in place of variables that are not needed for subsequent reference within the same rule and thus need not be named. The is operator in the rule for areaLoadAssigned is the assignment operator; it assigns the value of the expression to its right to the variable on its left. Since Prolog does not provide an ‘absolute’ function, one can write the abs relation as follows.

\[
\text{abs(} \text{X}, \text{X}) : - \text{X} > 0.
\]
\[
\text{abs(} \text{X}, \text{Y}) : - \text{X} <= 0, \text{Y} \text{ is } \text{X}.
\]
In keeping with the spirit of declarative programming, the ordering of rules in Prolog is generally not important—except in cases where the consequent of two or more rules affect the same predicate. In such cases, a rule that appears earlier in the fact base is tried first. The same is true for atomic sentences.

Prolog provides (among others) a second-order predicate, bagof. The query

```
bagof(Supports, beamSupport(someBeam, _, Supports), ListOfSupports)
```

will find out all the values of Supports such that beamSupport(someBeam, _, Supports) succeeds, and store those values in ListOfSupports. We use this feature to collect all the udll's on a beam and convert them one by one into equivalent point load as follows.

```
convertUDLLs(Beam) :-
  bagof([], udLoad(Beam, from, to, intensity), udllloads),
  convertToPtLoad(Beam, udllloads).
```

```
convertToPtLoad(Beam, [ ]).
convertToPtLoad(Beam, [[from, to, intensity] | Rest]) :-
  location is (from + to) / 2, abs(to - from, span),
  magnitude is intensity * span,
  assert(ptLoad(Beam, location, magnitude)),
  convertToPtLoad(Beam, Rest).
```

In the above rules, a special data structure, list, has been used. A list is a collection of elements separated by commas and enclosed in square brackets. As the above rules illustrate, the elements of a list can be accessed in the following two fashions or combinations thereof.

1. By relative position: From a list of three elements, we can access the second with [_, Element, _].

2. By head|tail notation: The first element of a list is called its head and the list of remaining elements, its tail. Thus, [H|T] for the list [a, b, c] will give {H = a, T = [b, c]}.

Now that the area loads have been transferred and uniformly distributed loads have been converted, the equilibrium equations can be applied to compute the end reactions. The following set of statements defines these calculations.

```
endReactions(Beam) :-
  endCoords(Beam, LeftEnd, RtEnd, Level),
  abs(RightEnd - LeftEnd, L),
  convertUDLLs(Beam),
  bagof([], load(Beam, Location, Magnitude), Loads),
  reaction(Beam, L, LeftEnd, RightEnd, Loads, LeftReaction, RtReaction),
  display([Beam, LeftReaction, RtReaction]),
  beamSupport(Beam, left, LeftSupport),
  assert(ptLoad(LeftSupport, Level, LeftReaction)),
  beamSupport(Beam, right, RightSupport),
  assert(ptLoad(RightSupport, Level, RtReaction)),
  assert(solved(Beam)).
```

```
endCoords(Beam, LeftEnd, RightEnd, Level) :-
  beamEnds(Beam, Level, LeftEnd, Level, RightEnd).
```
endCoords(Beam, LeftEnd, RightEnd, Level) :-
  beamEnds(Beam, LeftEnd, Level, RightEnd, Level).

reaction(_, _, LeftEnd, RightEnd, [], LeftReaction, RightReaction) :-
  LeftReaction is 0, RightReaction is 0.
reaction(Beam, L, LeftEnd, RtEnd, [[Locn, Mag] | Loads], LeftRctn, RtRctn) :-
  reaction(Beam, L, LeftEnd, RtEnd, Loads, AddlLeft, AddlRt),
  abs(Locn - LeftEnd, L1), abs(RtEnd - Locn, L2),
  LeftRctn is (L2 * Mag / L) + AddlLeft,
  RtRctn is (L1 * Mag / L) + AddlRt.

On computing the end reactions in the first statement, the extra-logical predicate display, which displays its argument on the terminal screen, is used to communicate the computed reactions to the user. The two rules for endCoords will be applicable in two different cases: when the beam is parallel to the x-axis and when the beam is parallel to the y-axis, respectively. End reaction is computed by means of recursion through the two rules for the reaction predicate.

Three more rules are still needed: one to ensure that all area loads have indeed been assigned to the supporting beams, one for making a pass through the data to solve all the solvable beams at a given stage, and the third whose consequent can be called by the user to start the solution process. These rules are given below.

allAreaLoadsAssigned :-
  not (areaLoad(Area, _),
       not (areaLoadAssigned(Area, left),
            areaLoadAssigned(Area, right))).

solveBeams(_) :-
  not (beamEnds(Beam, _, _, _, _),
       solvable(Beam),
       not endReactions(Beam)).

allBeamsSolved :-
  allAreaLoadsAssigned,
  beamEnds(Beam, _, _, _, _),
  solveBeams(Beam),
  not (beamEnds(Beam, _, _, _, _),
       not solved(Beam)).

The query allBeamsSolved will initiate the solution process for a given input and will display the reactions on the terminal screen as they are computed.

4. Concluding Remarks

In this report we have deliberately emphasized informality rather than strict exactness in our treatment of logic. The reader may find minor variations in the description of logic in some other works. For example, many authors exclude the concept of truth values (true and false) when describing formal logic systems and proof techniques. Despite the adequacy of such an approach, we have chosen to include the notion of truth values to impart an intuitive flavor to the syntactic manipulations.

We should also note that propositional and predicate calculus are not the only logical lan-
languages. The governing characteristic of a logical language is that it must possess declarative semantics, i.e., one should be able to evaluate the truth or falsehood of statements in the language for a given interpretation of the symbols. Decision tables, semantic nets, even frames and rules can also be regarded as logical languages. However, all these specialized languages provide only a subset of the expressive power and inferencing capabilities of predicate calculus, and can be defined in terms of predicate calculus. Even so, no single language is best for all types of knowledge; a specialized language may be more succinct and natural than predicate calculus for many applications.

One engineering domain in which logic is presently being employed for knowledge representation is integrated structural design of steel office buildings [Jain 88]. Design can broadly be considered as a deductive process wherein from the constraints on the final solution (e.g., deflection limitation) and information about the initial state in problem solving (e.g., occupancy of the building, material properties), one deduces a design description. This is true for both conceptual as well as detailed design phases. In the detailed design phase, the constraints are more formal (in the form of standards) while in the conceptual design phase, text book knowledge, standards, and accumulated experience of the engineer all play an important role. Logic provides the form with which to represent the above content. With the reasoning mechanisms described earlier, this content can be utilized to deduce design descriptions that meet the specified requirements.

4.1. Further Reading

In addition to the rules of inference and laws of propositional algebra described in Sec. 1.1.2, there are several other techniques for proving validity (and invalidity) of propositional calculus sentences. These include truth tables, semantic nets, and falsification. A good description of these techniques can be found in Ref. [Manna and Waldinger 85]. Ref. [Enderton 72; Pospesel 74; Pospesel 76] are other excellent book-length treatments of the two languages discussed in this report. We have described clausal resolution in this report. Description of non-clausal resolution can be found in Ref. [Manna and Waldinger 79].

The book [Amble 87] provides a good introduction to the field of logic programming and knowledge engineering. For a wider perspective on applications of logic to the field of Artificial Intelligence, the reader is encouraged to refer to [Genesereth and Nilsson 87]. Proceedings of the International Conferences on Logic Programming [Logic Programming 87; Logic Programming 88] are a good source of the latest developments in the area of logic programming.

In this report, we confined ourselves to describing monotonic reasoning. In monotonic reasoning, sound rules of inference are used that lead to conclusions that will be true under all circumstances. As it turns out, this is not enough to model common-sense reasoning. In non-monotonic reasoning one makes certain provisional inferences using rules of inference that are not sound. These provisional inferences are not guaranteed to be correct and may have to be later retracted in the presence of additional evidence. This may lead to a nonmonotonic growth of the fact base. The field of nonmonotonic reasoning has attracted a lot of attention recently and there is a growing body of work in this area. There are several approaches being pursued,
including modal operators [McDermott and Doyle 80; Moore 85], default theories [Reiter 80], and circumscription [McCarthy 80; Lifschitz 85].

The deductive power offered by logic comes at a price—it is computationally expensive. To overcome the drawback of slow execution and consequent long run-times with increasing size of the fact base, semantic attachment, wherein appropriate computational structures are used in place of rules of inference, can be employed. The idea of semantic attachment was first proposed by Green [Green 69] and later expounded by Weyhrauch [Weyhrauch 80]. The primary concept behind semantic attachment is the following. Using sound rules of inference ensures that any sentence derived from a given fact base will be consistent for all the possible interpretations for which the database is consistent. However, we may be equally well off with conclusions of less strength for most of our purposes, i.e., even when the derived sentence does not hold true for all the possible interpretations as long as it holds true in the intended interpretation. Engineering systems of significant size will have to rely on semantic attachment extensively.

Some researchers have expressed reservations about the adequacy of logic. In particular, they point to the inability of logic to model other types of reasoning besides deduction, and inability to provide guidance regarding which deductions to perform out of the possible candidates. Nonmonotonic reasoning and meta-level architecture [Genesereth 83] are some of the proposed solutions in response to such reservations. One of the more extensive discussions about the suitability of logic appears in a special volume of Computational Intelligence which includes a critique of logic by McDermott [McDermott 87] and commentary by leading researchers on both sides.

Acknowledgments

This work was supported in part by a grant from the Center for Integrated Facilities Engineering at Stanford University. Greg Luth provided helpful suggestions on an earlier draft of this report. Hossam El-Bibany's assistance with the Prolog implementation is gratefully acknowledged.

References


