Module 4: FIRST-ORDER LOGIC

.

4.1 REPRESENTATION REVISITED

In this section, we discuss the nature of representation languages. Our discussion motivates the development of first-order logic, a much more expressive language than the propositional logic.
Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Data structures within programs can represent facts; for example, a program could use a 4 × 4 array to represent the contents of the wumpus world. Thus, the programming language statement *World*[2,2] ← *Pit* is a fairly natural way to assert that there is a pit in square [2,2].

First drawback is programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure.

A second drawback of data structures in programs is the lack of any easy way to say, for example, "There is a pit in [2,2] or [3,1]" or "If the wumpus is in [1,1] then he is not in [2,2]." Programs can store a single value for each variable, and some systems allow the value to be "unknown," but they lack the expressiveness required to handle partial information.

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.
Propositional logic has a third property that is desirable in representation languages, namely, **compositionality**. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For example, the meaning of "S1,4 ∧ S1,2" is related to the meanings of "S1,4" and "S1,2."

i. The language of thought

- Natural languages (such as English or Spanish) are very expressive indeed.
- The modern view of natural language is that it serves a as a medium for communication rather than pure representation.
- Natural languages also suffer from ambiguity, a problem for a representation language.

In a first-order logic reasoning system that uses CNF, we can see that the linguistic form "¬(A ∨ B)" and "¬A ∧ ¬B" are the same because we can look inside the system and see that the two sentences are stored as the same canonical CNF form.

ii. Combining the best of formal and natural languages

We can adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, and borrowing representational ideas from natural language while avoiding its drawbacks.

When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to objects (squares, pits, wumpuses) and verbs and verb phrases that refer to relations among objects (is breezy, is adjacent to, shoots). Some of these relations are functions—relations in which there is only one "value" for a given "input."

It is easy to start listing examples of objects, relations, and functions:

• Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...
• Relations: these can be unary relations or properties such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, ...
• Functions: father of, best friend, third inning of, one more than, beginning of ...

Some examples follow:

• "One plus two equals three." Objects: one, two, three, one plus two; Relation: equals; Function: plus. ("One plus two" is a name for the object that is obtained by applying the function "plus" to the objects "one" and "two." "Three" is another name for this object.)

• "Squares neighboring the wumpus are smelly." Objects: wumpus, squares; Property: smelly; Relation: neighboring.

• "Evil King John ruled England in 1200." Objects: John, England, 1200; Relation: ruled; Properties: evil, king.

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of reality. Mathematically, this commitment is expressed through the nature of the formal models with respect to which the truth of sentences is defined.

- For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states: true or false, and each model assigns true or false to each proposition symbol.
- First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold.
- Temporal logic assumes that facts hold at particular times and that those times (which may be points or intervals) are ordered.
- Higher-order logic views the relations and functions referred to by first-order logic as objects in themselves.

Logic can also be characterized by its **epistemological commitments**—the possible states of knowledge that it allows with respect to each fact.
Systems using probability theory, on the other hand, can have any degree of belief, ranging from 0 (total disbelief) to 1 (total belief).

Language	Ontological Commitment (What exists in the world)	Epistemological Commitment (What an agent believes about facts)
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief $\in [0, 1]$
Fuzzy logic	facts with degree of truth $\in [0, 1]$	known interval value

Figure 8.1 Formal languages and their ontological and epistemological commitments.

4.2 Syntax and Semantics of First-Order Logic

i. Models for first-order logic

The models of a logical language are the formal structures that constitute the possible worlds under

consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values.
Models for first-order logic are much more interesting. First, they have objects in them! The domain of a model is the set of objects. The domain is required to be nonempty—every possible world must contain at least one object.

1. Richard the Lionheart, King of England from 1189 to 1199;
2. The evil King John, who ruled from 1199 to 1215;
3. The left legs of Richard
4. The left legs of John;
5. A crown

The objects in the model may be related in various ways. In the figure, Richard and John are brothers.
A relation is just the set of tuples of objects that are related. (A tuple is a collection of objects arranged in a fixed order and is written with angle brackets surrounding the objects.)

Thus, the brotherhood relation in this model is the set
{ <Richard the Lionheart, King John>, <King John, Richard the Lionheart >} (8.1)

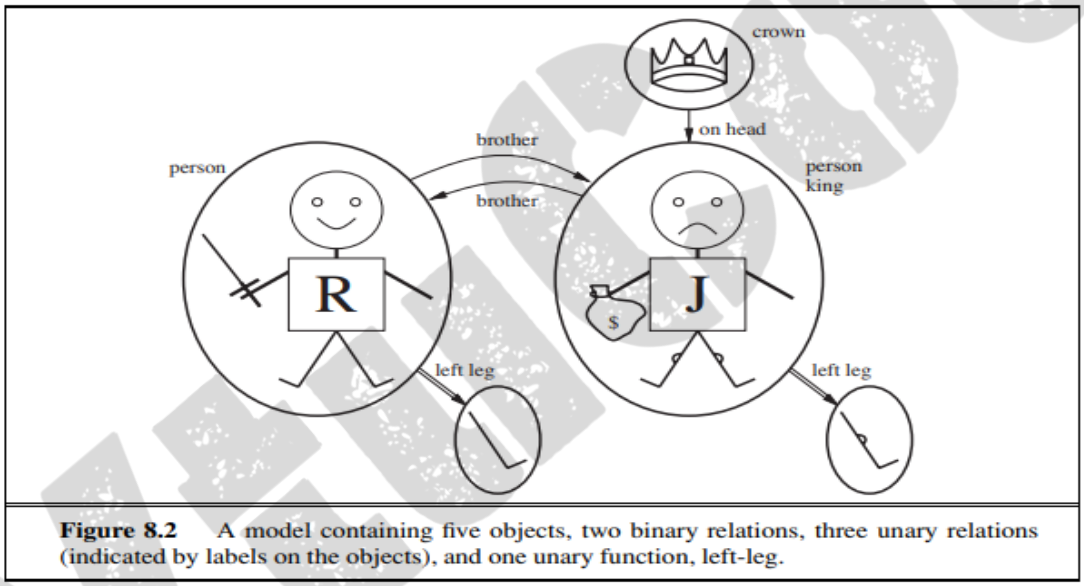


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

The crown is on King John's head, so the "on head" relation contains just one tuple,
<the crown, King John>.

The "brother" and "on head" relations are binary relations—that is, they relate pairs of objects.
The model also contains unary relations, or properties: the "person" property is true of both Richard and John; the "king" property is true only of John (presumably because Richard is dead at this point); and the "crown" property is true only of the crown.

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way.
For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

<Richard the Lionheart> → Richard's left leg (8.2)
<King John> → John's left leg

ii. Symbols and interpretations

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds:

- Constant symbols, which stand for objects;
- Predicate symbols, which stand for relations; and
- Function symbols, which stand for functions.

We adopt the convention that these symbols will begin with uppercase letters.
For example, we might use

- constant symbols Richard and John;
- predicate symbols Brother , OnHead, Person, King, and Crown; and
- function symbol LeftLeg.

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
&\mid \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\textit{Term} &\rightarrow \textit{Function}(\textit{Term}, \ldots) \\
&\mid \textit{Constant} \\
&\mid \textit{Variable} \\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\textit{Predicate} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} &: \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

As in propositional logic, every model must provide the information required to determine if any given sentence is true or false.
Thus, in addition to its objects, relations, and functions, each model includes an **interpretation** that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.
One possible interpretation for our example—which a logician would call the intended interpretation—is as follows:

• Richard refers to Richard the Lionheart and John refers to the evil King John.
• Brother refers to the brotherhood relation, that is, the set of tuples of objects given in Equation (8.1); OnHead refers to the "on head" relation that holds between the crown and King John;

• LeftLeg refers to the "left leg" function, that is, the mapping given in Equation (8.2).

There are many other possible interpretations, of course. For example, one interpretation maps Richard to the crown and John to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols Richard and John.

In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects. To get an idea of what the set of all possible models looks like, see Figure 8.4. It shows that models vary in how many objects they contain—from one up to infinity—and in the way the constant symbols map to objects. If there are two constant symbols and one object, then both symbols must refer to the same object; but this can still happen even with more objects. When there are more objects than constant symbols, some of the objects will have no names. Because the number of possible models is unbounded, checking entailment by the enumeration of all possible models is not feasible for first-order logic.

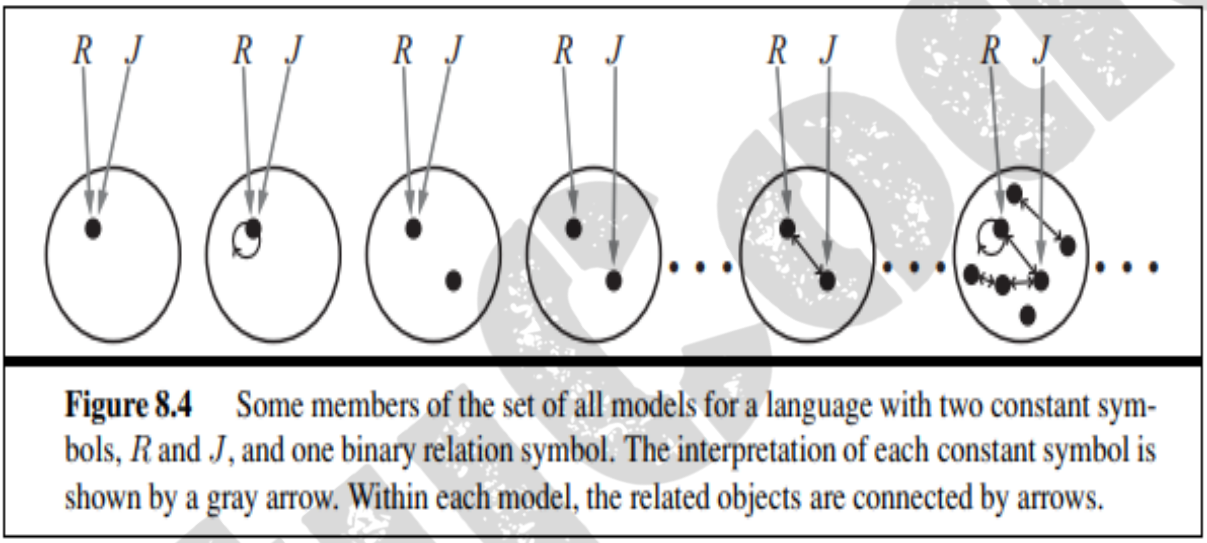


Figure 8.4 Some members of the set of all models for a language with two constant symbols, R and J, and one binary relation symbol. The interpretation of each constant symbol is shown by a gray arrow. Within each model, the related objects are connected by arrows.

iii. **Terms**

- A term is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg.
- This is what function symbols are for: instead of using a constant symbol, we use LeftLeg(John).
- In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol.
- The formal semantics of terms is straightforward. Consider a term f(t1,...,tn). The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them d1,...,dn); and the term as a whole refers to the object that is the value of the function F applied to d1,...,dn.

For example, suppose the LeftLeg function symbol refers to the function shown in Equation (8.2) and John refers to King John, then LeftLeg(John) refers to King John's left leg. In this way, the interpretation fixes the referent of every term.

iv. **Atomic sentences**

Atomic sentence is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as
Brother (Richard, John).

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John. Atomic sentences can have complex terms as arguments.

Thus,
Married(Father (Richard), Mother (John))
states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation).
An atomic sentence is true in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.

v. Complex sentences

We can use logical connectives to construct more complex sentences, with the same syntax and semantics as in propositional calculus.
Here are four sentences that are true in the model of Figure 8.2 under our intended interpretation:

¬Brother (LeftLeg(Richard), John)
Brother (Richard, John) ∧ Brother (John, Richard)
King(Richard) ∨ King(John)
¬King(Richard) ⇒ King(John) .

vi. Quantifiers

Quantifiers are used express properties of entire collections of objects in natural way, instead of enumerating the objects by name.
First-order logic contains two standard quantifiers, called **universal and existential.**

A. Universal quantification (∀)

The rule, "All kings are persons," is written in first-order logic as
∀ x King(x) ⇒ Person(x) . ∀ is usually pronounced "For all ..."
Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a variable. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function—for example, LeftLeg(x). A term with no variables is called a ground term.

The sentence ∀ x P, where P is any logical expression, says that P is true for every object x. More precisely, ∀ x P is true in a given model if P is true in all possible extended interpretations constructed from the interpretation given in the model, where each extended interpretation specifies a domain element to which x refers.

We can extend the interpretation in five ways:

- x → Richard the Lionheart,
- x → King John,
- x → Richard's left leg,
- x → John's left leg,
- x → the crown.

The universally quantified sentence ∀ x King(x) ⇒ Person(x) is true in the original model if the sentence King(x) ⇒ Person(x) is true under each of the five extended interpretations.
That is, the universally quantified sentence is equivalent to asserting the following five sentences:

- Richard the Lionheart is a king ⇒ Richard the Lionheart is a person.
- King John is a king ⇒ King John is a person.
- Richard's left leg is a king ⇒ Richard's left leg is a person.
- John's left leg is a king ⇒ John's left leg is a person.
- The crown is a king ⇒ the crown is a person.

B. Existential quantification (∃)

Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an existential quantifier.
To say, for example, that King John has a crown on his head, we write

$$\exists x \text{ Crown}(x) \land \text{OnHead}(x, \text{John}).$$

∃x is pronounced "There exists an x such that ..." or "For some x...".
Intuitively, the sentence ∃ x P says that P is true for at least one object x. More precisely, ∃ x P is true in a given model if P is true in at least one extended interpretation that assigns x to a domain element.

That is, at least one of the following is true:

- Richard the Lionheart is a crown ∧ Richard the Lionheart is on John's head;
- King John is a crown ∧ King John is on John's head;
- Richard's left leg is a crown ∧ Richard's left leg is on John's head;
- John's left leg is a crown ∧ John's left leg is on John's head;
- The crown is a crown ∧ the crown is on John's head.

Logical connector ⇒ appears to be the natural connective to use with ∀, ∧ is the natural connective to use with ∃.
Using ∧ as the main connective with ∀ led to an overly strong statement in the example in the previous section; using ⇒ with ∃ usually leads to a very weak statement, indeed.

Consider the following sentence: ∃ x Crown(x) ⇒ OnHead(x, John).
Applying the semantics, we see that the sentence says that at least one of the following assertions is true:

- Richard the Lionheart is a crown ⇒ Richard the Lionheart is on John's head;
- King John is a crown ⇒ King John is on John's head;
- Richard's left leg is a crown ⇒ Richard's left leg is on John's head

C. **Nested quantifiers**

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

$$\forall x \forall y \text{ Brother}(x, y) \Rightarrow \text{Sibling}(x, y) .$$

Consecutive quantifiers of the same type can be written as one quantifier with several variables.
For example, to say that siblinghood is a symmetric relationship, we can write

$$\forall x, y \text{ Sibling}(x, y) \Leftrightarrow \text{Sibling}(y, x) .$$

In other cases we will have mixtures. "Everybody loves somebody" means that for every person, there is someone that person loves: $\forall x \exists y \text{ Loves}(x, y)$.
On the other hand, to say "There is someone who is loved by everyone," we write

$$\exists y \forall x \text{ Loves}(x, y) .$$

The order of quantification is therefore very important.
It becomes clearer if we insert parentheses. $\forall x (\exists y \text{ Loves}(x, y))$ says that everyone has a particular property, namely, the property that they love someone.

On the other hand, $\exists y (\forall x \text{ Loves}(x, y))$ says that someone in the world has a particular property, namely the property of being loved by everybody.

D. Connections between ∀ and ∃

Two quantifiers are actually intimately connected with each other, through negation.

Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$\forall x \neg\text{Likes}(x,\text{Parsnips})$ is equivalent to $\neg\exists x \text{ Likes}(x,\text{Parsnips})$.

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream: $\forall x \text{ Likes}(x,\text{IceCream})$ is equivalent to $\neg\exists x \neg\text{Likes}(x,\text{IceCream})$.

Because ∀ is really a conjunction over the universe of objects and ∃ is a disjunction, it should not be surprising that they obey De Morgan's rules.

The De Morgan rules for quantified and unquantified sentences are as follows:

$\forall x \neg P \equiv \neg\exists x P$ $\quad\quad$ $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
$\neg\forall x P \equiv \exists x \neg P$ $\quad\quad$ $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
$\forall x P \equiv \neg\exists x \neg P$ $\quad\quad$ $P \wedge Q \equiv \neg(\neg P \vee \neg Q)$
$\exists x P \equiv \neg\forall x \neg P$ $\quad\quad$ $P \vee Q \equiv \neg(\neg P \wedge \neg Q)$.

Thus, we do not really need both ∀ and ∃, just as we do not really need both ∧ and ∨.

vii. Equality

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the equality symbol to signify that two terms refer to the same object.
For example,

$$\text{Father (John)} = \text{Henry}$$

says that the object referred to by Father (John) and the object referred to by Henry are the same. Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.
The equality symbol can be used to state facts about a given function, as we just did for the Father symbol. It can also be used with negation to insist that two terms are not the same object.
To say that Richard has at least two brothers, we would write

$$\exists x, y \text{ Brother}(x, \text{Richard}) \wedge \text{Brother}(y, \text{Richard}) \wedge \neg(x = y) .$$

viii. **An alternative semantics?**

Suppose that we believe that Richard has two brothers, John and Geoffrey
Assert *Brother* (*John, Richard*) ∧ *Brother* (*Geoffrey , Richard*) ? Not quite

1. This assertion is true in a model where Richard has only one brother—we need to add *John* != *Geoffrey* .
2. The sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey.
3. The correct translation of "Richard's brothers are John and Geoffrey" is as follows:
   Brother (*John, Richard*) ∧ *Brother* (*Geoffrey , Richard*) ∧ *John* ≠ *Geoffrey* ∧
   ∀ *x Brother* (*x, Richard*) ⇒ (*x = John* ∨ *x = Geoffrey*)

ix. **Database Semantics**

One proposal that is very popular in database systems works as follows.

- First, we insist that every constant symbol refer to a distinct object—the so-called **unique-names assumption**.
- Second, we assume that atomic sentences not known to be true are in fact false—the **closed-world assumption**.
- Finally, we invoke **domain closure**, meaning that each model contains no more domain elements than those named by the constant symbols.

4.3 USING FIRST-ORDER LOGIC

i. Assertions and queries in first-order logic

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions.
For example, we can assert that John is a king, Richard is a person, and all kings are persons:

TELL(KB, King(John)) .
TELL(KB, Person(Richard)) .
TELL(KB, ∀ x King(x) ⇒ Person(x)) .

We can ask questions of the knowledge base using ASK.
For example,

ASK(KB, King(John)) returns true.

Questions asked with ASK are called queries or goals. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively.

For example, given the two preceding assertions, the query

ASK(KB, Person(John)) should also return true.

We can ask quantified queries, such as

ASK(KB, ∃ x Person(x)) .

The answer is true, but this is perhaps not as helpful as we would like. It is rather like answering "Can you tell me the time?" with "Yes." If we want to know what value of x makes the sentence

true, we will need a different function, ASKVARS, which we call with

ASKVARS(KB,Person(x))

and which yields a stream of answers. In this case there will be two answers: {x/John} and {x/Richard}. Such an answer is called a **substitution or binding list**. ASKVARS is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific value.

ii. The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent."

Clearly, the objects in our domain are people. We have two unary predicates, Male and Female. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle.
We can go through each function and predicate, writing down what we know in terms of the other symbols.
For example,

one's mother is one's female parent:

$\forall$ m, c Mother (c) = m $\Leftrightarrow$ Female(m) $\wedge$ Parent(m, c) .

One's husband is one's male spouse:

$\forall$ w, h Husband(h, w) $\Leftrightarrow$ Male(h) $\wedge$ Spouse(h, w) .

Male and female are disjoint categories:

$\forall$ x Male(x) $\Leftrightarrow$ $\neg$Female(x) .

Parent and child are inverse relations:

$\forall$ p, c Parent(p, c) $\Leftrightarrow$ Child(c, p) .

A grandparent is a parent of one's parent:

$\forall$ g, c Grandparent(g, c) $\Leftrightarrow$ $\exists$ p Parent(g, p) $\wedge$ Parent(p, c) .

A sibling is another child of one's parents:

$\forall$ x, y Sibling(x, y) $\Leftrightarrow$ x = y $\wedge$ $\exists$ p Parent(p, x) $\wedge$ Parent(p, y) .

Each of these sentences can be viewed as an axiom of the kinship domain. Axioms are commonly associated with purely mathematical domains.
Our kinship axioms are also definitions; they have the form $\forall$ x, y P(x, y) $\Leftrightarrow$ …. The axioms define the Mother function and the Husband, Male, Parent, Grandparent, and Sibling predicates in terms of other predicates.

Not all logical sentences about a domain are axioms. Some are theorems—that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric:

$\forall$ x, y Sibling(x, y) $\Leftrightarrow$ Sibling(y, x)

Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully.
For example, there is no obvious definitive way to complete the sentence

$\forall$ x Person(x) $\Leftrightarrow$ ...

Fortunately, first-order logic allows us to make use of the Person predicate without completely defining it. Instead, we can write partial specifications of properties that every person has and

properties that make something a person:

$\forall$ x Person(x) $\Rightarrow$ ... $\forall$ x ... $\Rightarrow$ Person(x) .

Axioms can also be "just plain facts," such as Male(Jim) and Spouse(Jim, Laura). Such facts form the descriptions of specific problem instances, enabling specific questions to be answered.

iii. Numbers, sets, and lists

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of natural numbers or non-negative integers.

- A predicate *NatNum* that will be true of natural numbers.
- One constant symbol, 0; and
- One function symbol, *S* (successor).
- 0 is a natural number, and for every object *n*,
- if *n* is a natural number, then *S*(*n*) is a natural number.
- The natural numbers are 0, *S*(0), *S*(*S*(0)), and so on
- axioms to constrain the successor function

$$\forall\ n\ 0 \neq S(n)\ .$$
$$\forall\ m, n\ m \neq n \Rightarrow S(m) \neq S(n)\ .$$

Natural numbers are defined recursively:

$$\text{NatNum}(0)\ .$$
$$\forall\ \text{n NatNum(n)} \Rightarrow \text{NatNum(S(n))}\ .$$

Addition in terms of the successor function: (0-zero)

$$\forall\ m\ NatNum(m) \Rightarrow +(0, m) = m$$
$$\forall\ m, n\ NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m), n) = S(+(m, n))$$

write $S(n)$ as $n + 1$

$$\forall\ m, n\ NatNum(m) \wedge NatNum(n) \Rightarrow (m + 1) + n = (m + n) + 1\ .$$

- The use of infix notation is an example of **syntactic sugar**, that is, an extension to or abbreviation of the standard syntax that does not change the semantics.
- Addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on.

Sets: The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning.

- Normal vocabulary of set theory as syntactic sugar.
- Empty set { }.
- One unary predicate, *Set*, which is true of sets.
- The binary predicates are $x \in s$ (x is a member of set s) and $s_1 \subseteq s_2$ (set s_1 is a subset, not necessarily proper, of set s_2).
- The binary functions are $s_1 \cap s_2$ (the intersection of two sets), $s_1 \cup s_2$ (the union of two sets), and $\{x|s\}$ (the set resulting from adjoining element x to set s).

One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:
   $\forall$ s Set(s) $\Leftrightarrow$ (s = { }) $\vee$ ($\exists$ x, s2 Set(s2) $\wedge$ s = {x|s2}) .
2. The empty set has no elements adjoined into it. In other words, there is no way to decompose
   { } into a smaller set and an element:
   $\neg\exists$ x, s {x|s} = { } .
3. Adjoining an element already in the set has no effect:
   $\forall$ x, s x$\in$ s $\Leftrightarrow$ s = {x|s} .
4. The only members of a set are the elements that were adjoined into it.

$\forall$ x, s x$\in$ s $\Leftrightarrow$ $\exists$ y, s2 (s = {y|s2} $\wedge$ (x = y $\vee$ x$\in$ s2)) .

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:
   $\forall$ s1, s2 s1 $\subseteq$ s2 $\Leftrightarrow$ ($\forall$ x x$\in$ s1 $\Rightarrow$ x$\in$ s2) .
6. Two sets are equal if and only if each is a subset of the other:
   $\forall$ s1, s2 (s1 = s2) $\Leftrightarrow$ (s1 $\subseteq$ s2 $\wedge$ s2 $\subseteq$ s1) .
7. An object is in the intersection of two sets if and only if it is a member of both sets:
   $\forall$ x, s1, s2 x$\in$ (s1 $\cap$ s2) $\Leftrightarrow$ (x$\in$ s1 $\wedge$ x$\in$s2) .
8. An object is in the union of two sets if and only if it is a member of either set:
   $\forall$ x, s1, s2 x$\in$ (s1 $\cup$ s2) $\Leftrightarrow$ (x$\in$ s1 $\vee$ x$\in$s2) .

Lists are similar to sets. The differences are that lists are ordered and the same element can appear more than once in a list

- *List* ? is a predicate that is true only of lists.
- lists: *Nil* is the constant list with no elements;
- *Cons* , *Append* , *First* , and *Rest* are functions; and
- *Find* is the predicate that does for lists what *Member* does for sets.
- *Cons*(*x, y*), where *y* is a nonempty list, is written [*x*|*y*]
- *Cons*(*x, Nil*) (i.e., the list containing the element *x*) is written as [*x*]
- Nested term *Cons*(*A, Cons* (*B, Cons* (*C, Nil*)))

iv. The wumpus world

Wumpus agent receives a percept vector with five elements:

Percept ([*Stench, Breeze, Glitter , None, None*], 5)

- The actions in the wumpus world can be represented by logical terms:
  Turn(*Right*), *Turn*(*Left*), *Forward* , *Shoot* , *Grab, Climb.*
- To determine which is best, the agent program executes the query
  ASKVARS($\exists$ *a BestAction*(*a*, 5))---------returns a binding list such as {*a/Grab* }.
- The raw percept data implies certain facts about the current state
  example:

$\forall$ *t, s, g, m, c Percept* ([*s, Breeze, g, m, c*], *t*) $\Rightarrow$ *Breeze*(*t*) ,

$\forall$ *t, s, b, m, c Percept*([*s, b, Glitter , m, c*], *t*) $\Rightarrow$ *Glitter* (*t*) *,…….*

- These rules exhibit a trivial form of the reasoning process called **perception**
- Simple "reflex" behavior can also be implemented by quantified implication sentences.
  Example: $\forall$ *t Glitter* (*t*) $\Rightarrow$ *BestAction*(*Grab, t*)
- To represent the environment itself
  Adjacency of any two squares can be defined as

$\forall$ *x, y, a, b Adjacent*([*x, y*], [*a, b*]) $\Leftrightarrow$

(*x* = *a* $\wedge$ (*y* = *b* − 1 $\vee$ *y* = *b* + 1)) $\vee$ (*y* = *b* $\wedge$ (*x* = *a* − 1 $\vee$ *x* = *a* + 1))

- The agent's location changes over time *At*(*Agent , s, t*) to mean that the agent is at square *s* at time *t*
- Fix the wumpus's location with $\forall$*t At*(*Wumpus,* [2, 2], *t*)
- objects can only be at one location at a time

$\forall$ *x*, s_1, s_2,*t At*(*x*, s_1, *t*) $\wedge$ *At*(*x*, s_2, *t*) $\Rightarrow$ s_1 = s_2

- If the agent is at a square and perceives a breeze, then that square is breezy

$\forall$ *s, t At*(*Agent , s, t*) $\wedge$ *Breeze*(*t*) $\Rightarrow$ *Breezy* (*s*)

- First-order logic just needs one axiom to deduce where the pits are :

$\forall$ *s Breezy* (*s*) $\Leftrightarrow$ $\exists$ *r Adjacent* (*r, s*) $\wedge$ *Pit*(*r*)

- The axiom for the arrow

$\forall$ *t HaveArrow* (*t* + 1) $\Leftrightarrow$ (*HaveArrow* (*t*) $\wedge$ ¬*Action*(*Shoot , t*))

.

INFERENCE IN FIRST-ORDER LOGIC

1. Propositional Vs First Order Inference

Earlier inference in first order logic is performed with *Propositionalization* which is a process of converting the Knowledgebase present in First Order logic into Propositional logic and on that using any inference mechanisms of propositional logic are used to check inference.

Inference rules for quantifiers:

There are some Inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers**.** These rules will lead us to make the conversion.

Universal Instantiation (UI):

The rule says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. Let SUBST (θ*)* denote the result of applying the substitution θto the sentence *a*. Then the rule is written

$$\frac{\forall v \;\; a}{\text{SUBST}(\{v/g\}, \alpha)}$$

For any variable v and ground term *g.*

For example, there is a sentence in knowledge base stating that all greedy kings are Evils

$$\forall x \;\; King(x) \; A \; Greedy(x) \Rightarrow Evil(x) \; .$$

For the variable x, with the substitutions like {x/John},{x/Richard}the following sentences can be inferred.

$$King(John) \; A \; Greedy(John) \Rightarrow Evil(John).$$
$$King(Richard) \; A \; Greedy(Richard) \Rightarrow Evil(Richard) \; .$$

Thus a universally quantified sentence can be replaced by the set of *all* possible instantiations.

Existential Instantiation (EI):

The existential sentence says there is some object satisfying a condition, and the instantiation process is just giving a name to that object, that name must not already belong to another object. This new name is called a **Skolem constant.** Existential Instantiation is a special case of a more general process called *"skolemization".*

For any sentence *a*, variable v, and constant symbol *k* that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \;\; \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

For example, from the sentence

$$\exists x \;\; Crown(x) \; A \; OnHead(x, John)$$

So, we can infer the sentence

$$Crown(C_1) \wedge OnHead(C_1, John)$$

As long as C_1 does not appear elsewhere in the knowledge base. Thus an existentially quantified sentence can be replaced by one instantiation

Elimination of Universal and Existential quantifiers should give new knowledge base which can be shown to be *inferentially equivalent*to oldin the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

Reduction to propositional inference:

Once we have rules for inferring non quantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. For example, suppose our knowledge base contains just the sentences

$$\forall x \ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John).$$

Then we apply UI to the first sentence using all possible ground term substitutions from the vocabulary of the knowledge base-in this case, *{xl John)* and *{x/Richard).* We obtain

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John),$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$

We discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences-King *(John), Greedy (John),* and Brother (Richard, *John*) as proposition symbols. Therefore, we can apply any of the complete propositional algorithms to obtain conclusions such as *Evil (John).*

Disadvantage:

If the knowledge base includes a function symbol, the set of possible ground term substitutions is infinite. Propositional algorithms will have difficulty with an infinitely large set of sentences.

NOTE:

Entailment for first-order logic is *semi decidable* which means algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non entailed sentence.

2. Unification and Lifting

Consider the above discussed example, if we add Siblings (Peter, Sharon) to the knowledge base then it will be

$$\forall x \ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John)$$
$$\mathbf{Siblings(Peter, Sharon)}$$

Removing Universal Quantifier will add new sentences to the knowledge base which are not necessary for the query *Evil (John)?*

$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$
$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$
$King(Peter) \wedge Greedy(Peter) \Rightarrow Evil(Peter)$
$King(Sharon) \wedge Greedy(Sharon) \Rightarrow Evil(Sharon)$

Hence we need to teach the computer to make better inferences. For this purpose Inference rules were used.

First Order Inference Rule:

The key advantage of lifted inference rules over *propositionalization* is that they make only those substitutions which are required to allow particular inferences to proceed.

Generalized Modus Ponens:

If there is some substitution **θ** that makes the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying **θ**. This inference process can be captured as a single inference rule called Generalized Modus Ponens which is a ***lifted***version of Modus Ponens-it raises Modus Ponens from propositional to first-order logic

For atomic sentences pi, pi ', and q, where there is a substitution θ such that SUBST(θ , pi) = SUBST(θ , pi '), for all i,

$$\frac{\textbf{p1 ', p2 ', …, pn ', (p1} \wedge \textbf{p2} \wedge \textbf{…} \wedge \textbf{pn} \Rightarrow \textbf{q)}}{\textbf{SUBST (θ, q)}}$$

There are N + 1 premises to this rule, N atomic sentences + one implication. Applying SUBST (θ, q) yields the conclusion we seek. It is a sound inference rule.

Suppose that instead of knowing Greedy (John) in our example we know that everyone is greedy:

∀y Greedy(y)

We would conclude that Evil(John).

Applying the substitution {x/John, y / John) to the implication premises King (x) and Greedy (x) and the knowledge base sentences King(John) and Greedy(y)will make them identical. Thus, we can infer the conclusion of the implication.

For example,

p_1' is $King(John)$ — p_1 is $King(x)$
p_2' is $Greedy(y)$ — p_2 is $Greedy(x)$
θ is $\{x/John, y/John)$ — q is $Evil(x)$
SUBST(θ, q) is $Evil(John)$.

Unification:

It is the process used to find substitutions that make different logical expressions look identical. **Unification** is a key component of all first-order Inference algorithms.

UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q) θ is our unifier value (if one exists). Ex: ―Who does John know?‖
UNIFY (Knows (John, x), Knows (John, Jane)) = {x/ Jane}. UNIFY (Knows (John, x), Knows (y, Bill)) = {x/Bill, y/ John}.
UNIFY (Knows (John, x), Knows (y, Mother(y))) = {x/Bill, y/ John} UNIFY (Knows (John, x), Knows (x, Elizabeth)) = FAIL

- The last unification fails because both use the same variable, X. X can't equal both John and Elizabeth. To avoid this change the variable X to Y (or any other value) in Knows(X, Elizabeth)

Knows(X, Elizabeth) → Knows(Y, Elizabeth)

Still means the same. This is called **standardizing apart.**

- sometimes it is possible for more than one unifier returned:

UNIFY (Knows (John, x), Knows(y, z)) =???

This can return two possible unifications: {y/ John, x/ z} which means Knows (John, z) OR {y/ John, x/ John, z/ John}. For each unifiable pair of expressions there is a single **most general unifier (MGU)**, In this case it is *{y/ John, x/z)*.
An algorithm for computing most general unifiers is shown below.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 2.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution 0 that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression, such as F (A, B), the function OP picks out the function symbol F and the function ARCS picks out the argument list (A, B).

The process is very simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. **Occur check** step makes sure same variable isn't used twice.

Storage and retrieval

- STORE(s) stores a sentence *s* into the knowledge base
- FETCH(s) returns all unifiers such that the query q unifies with some sentence in the knowledge base.

  Easy way to implement these functions is Store all sentences in a long list, browse list one sentence at a time with UNIFY on an ASK query. But this is inefficient.

  To make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. (i.e. Knows(John, x) vs. Brother(Richard, John) are not compatible for unification)
- To avoid this, a simple scheme called ***predicate indexing***puts all the *Knows* facts in one bucket and all the *Brother* facts in another.
- The buckets can be stored in a hash table for efficient access. Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol.

  But if we have many clauses for a given predicate symbol, facts can be stored under multiple index keys.

  For the fact *Employs (AIMA.org, Richard),* the queries are *Employs (A IMA. org, Richard)* Does AIMA.org employ Richard? *Employs (x, Richard)* who employs Richard?
  Employs (AIMA.org, y) whom does AIMA.org employ?
  Employs Y(x), who employs whom?

  We can arrange this into a **subsumption lattice,** as shown below**.**

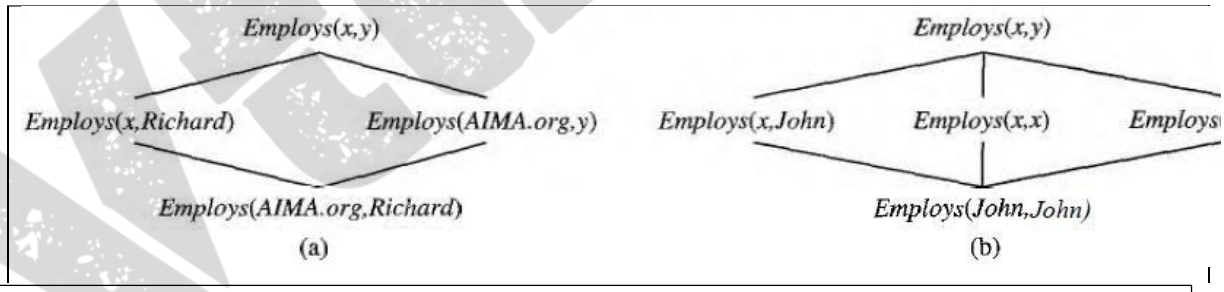


Figure 2.2 *(a) The subsumption lattice whose lowest node is the sentence Employs (AIMA.org, Richard). (b) The subsumption lattice for the sentence Employs (John, John).*

A subsumption lattice has the following properties:

- ✓ child of any node obtained from its parents by one substitution

- ✓ the ―highest‖ common descendant of any two nodes is the result of applying their most general unifier.
- ✓ predicate with n arguments contains O(2n) nodes (in our example, we have two arguments, so our lattice has four nodes)
- ✓ Repeated constants = slightly different lattice.

3. Forward Chaining

First-Order Definite Clauses:

A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses: Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Consider the following problem;

"The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

We will represent the facts as first-order definite clauses

". . . It is a crime for an American to sell weapons to hostile nations":

Example Knowledge Base:

- ….it is a crime for an American to sell weapons to hostile nations:
  Rule 1. $American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$
- Nono . . . has some missiles,
  i.e., $\exists x\ Owns(Nono, x) \wedge Missile(x)$:
  Rule 2. $Owns(Nono, M_1)$ **and**
  Rule 3. $Missile(M_1)$
- . . . all of its missiles were sold to it by Colonel West
  Rule 4. $\forall x\ Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$
- Missiles are weapons:
  Rule 5. $Missile(x) \Rightarrow Weapon(x)$
- An enemy of America counts as "hostile":
  Rule 6. $Enemy(x, America) \Rightarrow Hostile(x)$
- West, who is American . . .
  Rule 7. $American(West)$
- The country Nono, an enemy of America . . .
  Rule 8. $Enemy(Nono, America)$

A simple forward-chaining algorithm:

- Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions lo the known facts
- The process repeats until the query is answered or no new facts are added. Notice that a fact is not "new" if it is just *renaming*of a known fact.

We will use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (1), (4), (5), and (6). Two iterations are required:

- ✓ On the first iteration, rule (1) has unsatisfied premises.
  Rule (4) is satisfied with *{x/Ml),* and *Sells (West,* M1, *Nono)* is added. Rule (5) is satisfied with *{x/M1) a*nd *Weapon (M1)* is added.
  Rule (6) is satisfied with *{x/Nono},* and *Hostile (Nono)* is added.
- ✓ On the second iteration, rule (1) is satisfied with *{x/West, Y/MI, z /Nono),* and *Criminal* (*West)* is added.
  It is ***sound,*** because every inference is just an application of Generalized Modus Ponens, it is ***complete***for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each rule in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new then
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

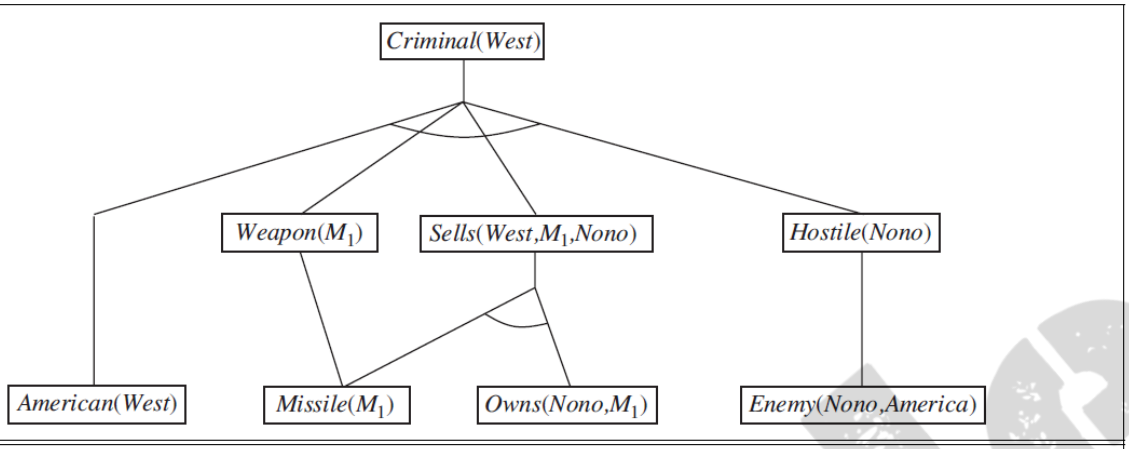


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Efficient forward chaining:

The above given forward chaining algorithm was lack with efficiency due to the the three sources of complexities:

- ✓ Pattern Matching
- ✓ Rechecking of every rule on every iteration even a few additions are made to rules
- ✓ Irrelevant facts

1. **Matching rules against known facts:**

For example, consider this rule,

Missile(x) A Owns (Nono, x) =>Sells (West, x, Nono).

The algorithm will check all the objects owned by Nono in and then for each object, it could check whether it is a missile. This is the ***conjunct ordering problem:***

―Find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized‖. The **most constrained variable** heuristic used for CSPs would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than objects that are owned by Nono.

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains-for example, Missile(x) is a unary constraint on x. Extending this idea, we can express everyfinite-domain CSP as a single definite clause together with some associated ground facts. Matching a definite clause against a set of facts is NP-hard

2. **Incremental forward chaining:**

On the second iteration, the rule ***Missile (x) =>Weapon (x)***

Matches against Missile (M1) (again), and of course the conclusion Weapon(x/M1) is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation:

―Every new fact inferred on iteration t must be derived from at leastone new fact inferred on iteration t – 1‖.

This observation leads naturally to an incremental forward chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p, that unifies with a fact p: newly inferred at iteration t - 1. The rule matching step then fixes p, to match with p', but allows the other conjuncts of the rule to match with facts from any previous iteration.

3. **Irrelevant facts:**

- One way to avoid drawing irrelevant conclusions is to use backward chaining.
- Another solution is to restrict forward chaining to a selected subset of rules
- A third approach, is to rewrite the rule set, using information from the goal.so that only relevant variable bindings-those belonging to a so-called **magic** set-are considered during forward inference.

For example, if the goal is Criminal (West), the rule that concludes Criminal (x) will be rewritten to include an extra conjunct that constrains the value of x:

Magic(x) A American(z) A Weapon(y)A Sells(x, y, z) A Hostile(z) =>Criminal(x)

The fact *Magic (West)* is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process.

4. Backward Chaining

This algorithm work backward from the goal, chaining through rules to find known facts that support the proof. It is called with a list of goals containing the original query, and returns the set of all substitutions satisfying the query. The algorithm takes the first goal in the list and finds every clause in the knowledge base whose **head,** unifies with the goal. Each such clause creates a new recursive call in which **body,** of the clause is added to the goal stack .Remember that facts are clauses with a head but no body, so when a goal unifies with a known fact, no new sub goals are added to the stack and the goal is solved. The algorithm for backward chaining and proof tree for finding criminal (West) using backward chaining are given below.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ′ in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ′

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first, rest ← FIRST(goals), REST(goals)
    for each θ′ in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ″ in FOL-BC-AND(KB, rest, θ′) do
        yield θ″
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

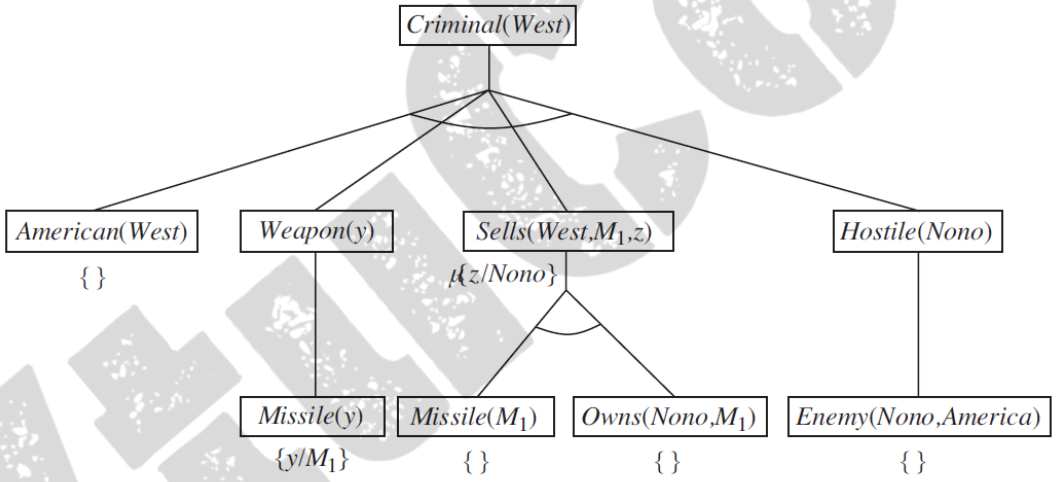


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to $Nono$.

Logic programming:

- **Prolog** is by far the most widely used logic programming language.
- Prolog programs are sets of definite clauses written in a notation different from standard first-order logic.
- Prolog uses uppercase letters for variables and lowercase for constants.
- Clauses are written with the head preceding the body; " : -" is used for left implication, commas separate literals in the body, and a period marks the end of a sentence
  Prolog includes "syntactic sugar" for list notation and arithmetic. Prolog program for append

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z)
```

  (X, Y, Z), which succeeds if list Z is the result of appending lists x and Y

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z)
```

  For example, we can ask the query append (A, B, [1, 2]): what two lists can be appended to give [1, 2]? We get back the solutions

```
A=[]     B=[1,2]
A=[1]    B=[2]
A=[1,2]  B=[]
```

- The execution of Prolog programs is done via depth-first backward chaining

- Prolog allows a form of negation called **negation as failure.** A negated goal not P is considered proved if the system fails to prove p. Thus, the sentence
  Alive (X) : - not dead(X) can be read as "Everyone is alive if not provably dead."
- Prolog has an equality operator, =, but it lacks the full power of logical equality. An equality goal succeeds if the two terms are *unifiable* and fails otherwise. So X+Y=2+3 succeeds with x bound to *2* and Y bound to 3, but Morningstar=evening star fails.
- The occur check is omitted from Prolog's unification algorithm.

Efficient implementation of logic programs:

The execution of a Prolog program can happen in two modes: interpreted and compiled.

- Interpretation essentially amounts to running the FOL-BC-ASK algorithm, with the program as the knowledge base. These are designed to maximize speed.

  First, instead of constructing the list of all possible answers for each sub goal before continuing to the next, Prolog interpreters generate one answer and a "promise" to generate the rest when the current answer has been fully explored. This promise is called a **choice point.**FOL-BC-ASK spends a good deal of time in generating and composing substitutions
  when a path in search fails. Prolog will backup to previous choice point and unbind some variables.

This is called ―TRAIL‖. So, new variable is bound by UNIFY-VAR and it is pushed on to trail.

- Prolog Compilers compile into an intermediate language i.e., Warren Abstract Machine or WAM named after David. H. D. Warren who is one of the implementers of first prolog compiler. So, WAM is an abstract instruction set that is suitable for prolog and can be either translated or interpreted into machine language.

  Continuations are used to implement choice point'scontinuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds.
- Parallelization can also provide substantial speedup. There are two principal sources of parallelism

1. The first, called **OR-parallelism,** comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel.
2. The second, called **AND-parallelism,** comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables.

Redundant inference and infinite loops:

Consider the following logic program that decides if a path exists between two points on a directed graph.

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z)
```

A simple three-node graph, described by the facts link (a, b) and link (b, c)

It generates the query path (a, c)

Hence each node is connected to two random successors in the next layer.

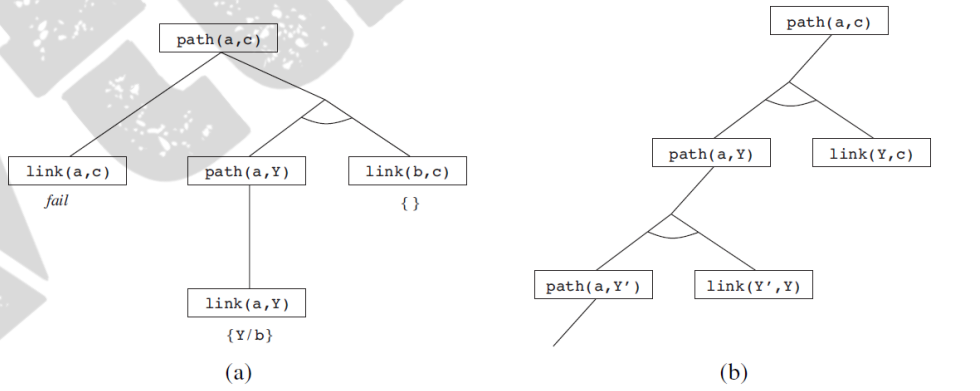


Figure 9.10 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

Constraint logic programming:

The Constraint Satisfaction problem can be solved in prolog as same like backtracking algorithm.

Because it works only for finite domain CSP's in prolog terms there must be finite number of solutions for any goal with unbound variables.

```
triangle(X,Y,Z) :-
    X>=0, Y>=0, Z>=0, X+Y>=Z, Y+Z>=X, X+Z>=Y.
```

- If we have a query, triangle (3, 4, and 5) works fine but the query like, triangle (3, 4, Z) no solution.
- The difficulty is variable in prolog can be in one of two states i.e., Unbound or bound.
- Binding a variable to a particular term can be viewed as an extreme form of constraint namely ―equality‖.CLP allows variables to be constrained rather than bound.

The solution to triangle (3, 4, Z) is Constraint 7>=Z>=1.

5. Resolution

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF) that is, a conjunction of clauses, where each clause is a disjunction ofliterals.

Literals can contain variables, which are assumed to be universally quantified. Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. We will illustrate the procedure by translating the sentence

"Everyone who loves all animals is loved by someone," or

$$\forall x \ [\forall y \ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y \ Loves(y, x)]$$

The steps are as follows:

Step 1. Eliminate implications:

$$ti\, x \ [\neg \forall y \ \neg Animal(y) \vee Loves(x, y)] \ v \ [\exists y \ Loves(y, x)]$$

Step 2. Move Negation inwards: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$$\neg \forall x \ p \quad \text{becomes} \quad 3\, x \ \neg p$$
$$\neg \exists x \ p \quad \text{becomes} \quad \forall x \ \neg p$$

Our sentence goes through the following transformations:

$$\forall x \ [\exists y \ \neg(\neg Animal(y) \vee Loves(x, y))] \vee [\exists y \ Loves(y, x)] .$$
$$\forall x \ [\exists y \ \neg\neg Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y \ Loves(y, x)].$$
$$\forall x \ [\exists y \ Animal(y) \wedge \neg Loves(x, y)] \ v \ [\exists y \ Loves(y, x)] .$$

Step 3. Standardize variables: For sentences like $(\forall x \quad P(x)) \vee (\exists x \quad Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

$$\forall x \ [\exists y \ Animal(y) \wedge \neg Loves(x, y)] \vee [\exists z \ Loves(z, x)]$$

Step 4. Skolemize: Skolemization is the process of removing existential quantifiers by elimination. Translate *3 x P(x)* into *P(A),* where *A* is a new constant. If we apply this rule to our sample sentence, however, we obtain

$\forall x \ [Animal(A) \wedge \neg Loves(x, A)] \vee Loves(B, x)$

Which has the wrong meaning entirely: it says that everyone either fails to love a particular animal *A* or is loved by some particular entity *B.* In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person.

Thus, we want the Skolem entities to depend on *x:*

$$\forall x \ [Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x), x)$$

Here *F* and *G* are Skolem functions. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears.

Step 5. Drop universal quantifiers: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers

$$[Animal(F(x)) \wedge \neg Loves(x, F(x))] \vee Loves(G(x), x)$$

Step 6. Distribute ∨ over ∧

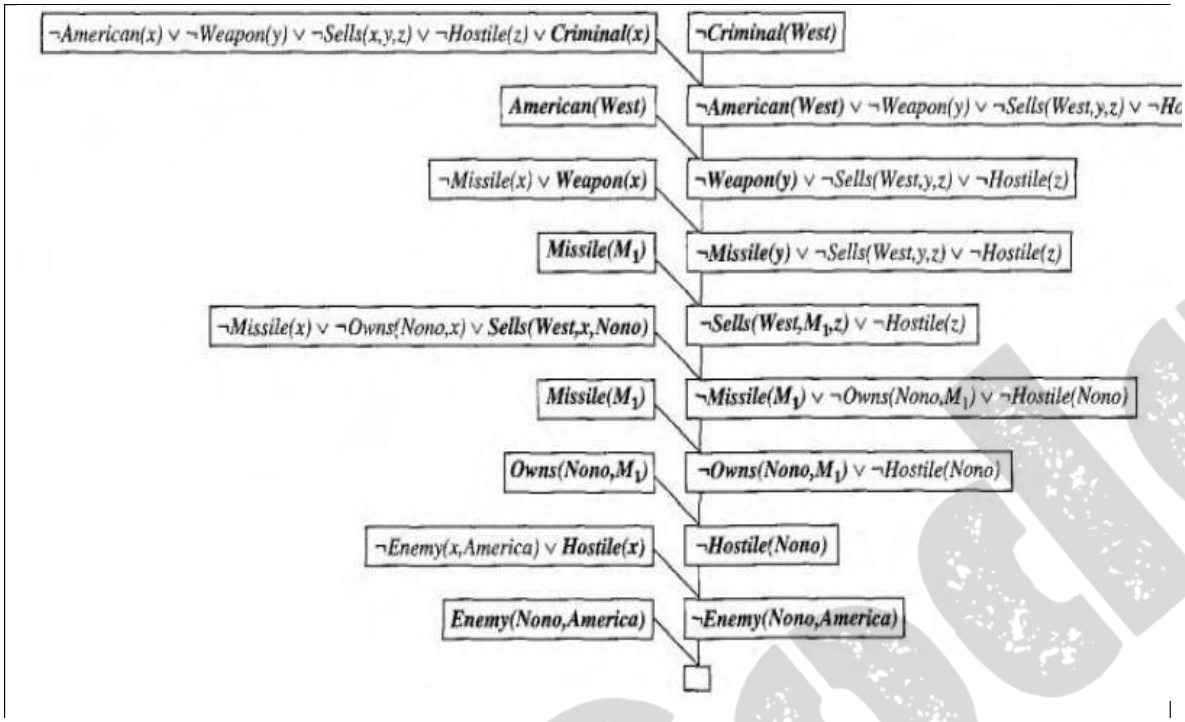
$$[Animal(F(x)) \vee Loves(G(x), x)] \wedge [\neg Loves(x, F(x)) \vee Loves(G(x), x)].$$

This is the CNF form of given sentence.

The resolution inference rule:

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one ***unifies with*** the negation of the other. Thus we have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$



Where UNIFY (l_i, m_j) == θ.

For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

By eliminating the complementary literals *Loves (G(x), x)* and ¬*Loves (u, v),* with unifier θ =

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

{u/G(x), v/x), to produce the resolvent clause

Example proofs:

Resolution proves that KB /= *a* by proving *KB* A *la* unsatisfiable, i.e., by deriving the empty clause. The sentences in CNF are

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$$
$$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono) .$$
$$\neg Enemy(x, America) \vee Hostile(x) .$$
$$\neg Missile(x) \vee Weapon(x) .$$
$$Owns(Nono, M_1) . \qquad Missile(M_1) .$$
$$American(West) . \qquad Enemy(Nono, America) .$$

The resolution proof is shown in below figure;

Figure 5.1 A resolution proof that West is a criminal.

Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. Backward chaining is really just a special case of resolution with a particular control strategy to decide which resolution to perform next.

Example 2:

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

First, express the original sentences, some background knowledge, and the negated goal G in first-order logic:

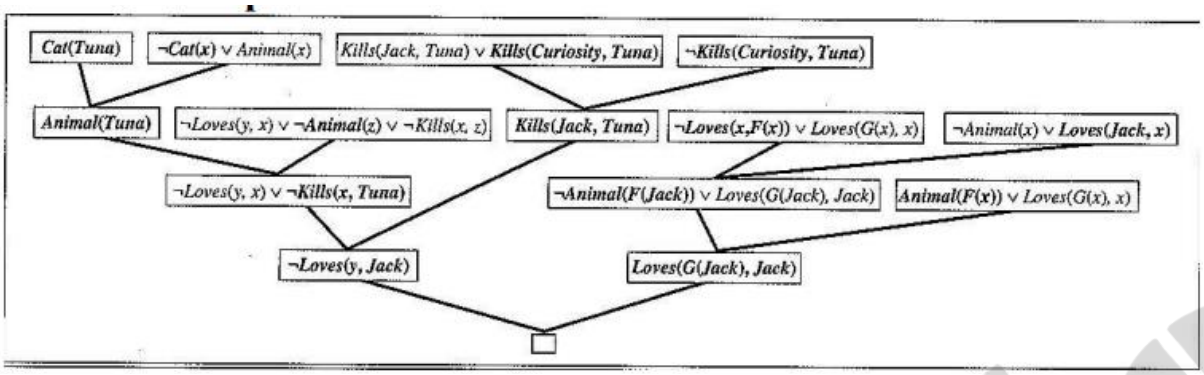
- $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$
- $\forall x\ [\exists z\ Animal(z) \wedge Kills(x,z)] \Rightarrow [\forall y\ \neg Loves(y,x)]$
- $\forall x\ Animal(x) \Rightarrow Loves(Jack,x)$
- $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
- $Cat(Tuna)$
- $\forall x\ Cat(x) \Rightarrow Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

Now apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \vee Loves(G(x), x)$
A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$
B. $\neg Loves(y,x) \vee \neg Animal(z) \vee \neg Kills(x,z)$
C. $\neg Animal(x) \vee Loves(Jack,x)$
D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
E. $Cat(Tuna)$
F. $\neg Cat(x) \vee Animal(x)$
¬G. $\neg Kills(Curiosity, Tuna)$

The resolution proof that Curiosity killed the cat is given in Figure 9.12.



Dealing with equality

There are several ways to deal with $t_1 = t_2$. One of them is *Paramodulation*:

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee t_1 = t_2, \qquad \ell'_1 \vee \cdots \vee \ell'_n[t_3]}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell'_n[y])}$$

where

$$\text{UNIFY}\ (t_1, t_3) = \theta$$

This inference rule can be used during the resolution algorithm.

Example 13.

$$\frac{Father(John) = Father(Richard) \quad Male(Father(x))}{Male(Father(Richard))}$$

$$\theta = \{x/John\} = \text{UNIFY}(Father(John), Father(x))$$

Theorem provers

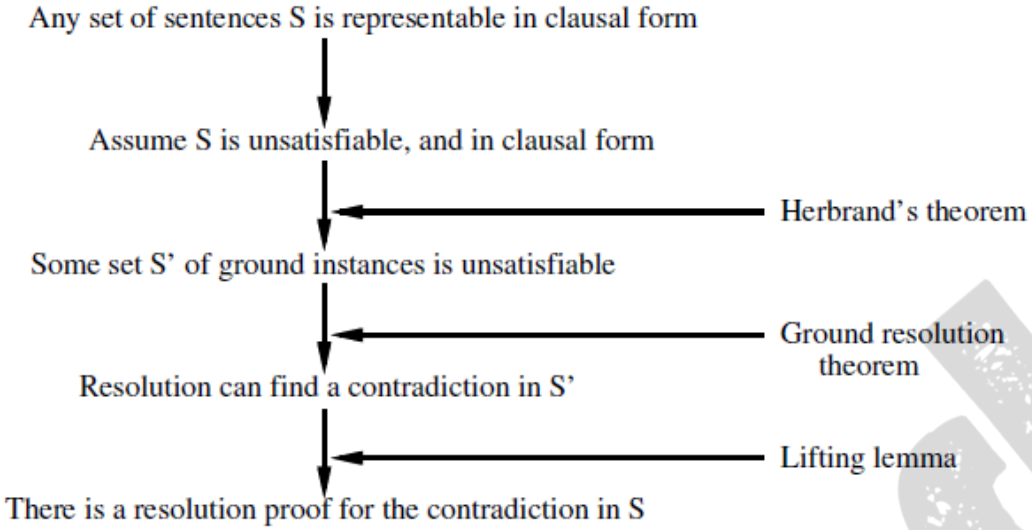
Unlike logic programming language, *Theorem provers* cover FOL (no restriction on Definite Clauses). Their algorithm is based on resolution.

Theorem prover: OTTER

Using the resolution algorithm in a "clever" way.

- *Unit preference*: Inference of sentences with a minimal number of literals (more chance to get the empty clause)
- *Set of support*: What is the set of clauses in KB that will be *useful*?
- *Input resolution*: Always using a sentence from KB or α to apply the resolution rule.
- *Subsumption*: Elimination of sentences that are subsumed by (more specific than) an existing sentence in the KB.

Completeness of resolution



To carry out the first step, we need three new concepts:

- **Herbrand universe**: If S is a set of clauses, then H_S, the Herbrand universe of S, is the set of all ground terms constructable from the following:
  a. The function symbols in S, if any.
  b. The constant symbols in S, if any; if none, then the constant symbol A.

  For example, if S contains just the clause $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, then H_S is the following infinite set of ground terms:

  $$\{A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), \ldots\} .$$
- **Saturation**: If S is a set of clauses and P is a set of ground terms, then $P(S)$, the saturation of S with respect to P, is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P with variables in S.
- **Herbrand base**: The saturation of a set S of clauses with respect to its Herbrand uni- verse is called the Herbrand base of S, written as $H_S(S)$. For example, if S contains solely the clause just given, then $H_S(S)$ is the infinite set of clauses

$$\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B),$$
$$\neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B),$$
$$\neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B),$$
$$\neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \ldots \}$$