Module 4

8.1 REPRESENTATION REVISITED

Drawbacks of Procedural Languages

- Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent only computational processes. 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].
- What 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 whose details are derived by the programmer from his or her own knowledge of the domain.
- A second drawback of is the lack the expressiveness required to handle partial information . For example data structures in programs lack the easy way to say, “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+.”

Advantages of Propositional Logic

- The declarative nature of propositional logic, specify that knowledge and inference are separate, and inference is entirely domain-independent.
- 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 COMPOSITIONALITY 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 “$S_{1,4} \wedge S_{1,2}$” is related to the meanings of “$S_{1,4}$” and “$S_{1,2}$".

Drawbacks of Propositional Logic

- Propositional logic lacks the expressive power to concisely describe an environment with many objects.
- For example, we were forced to write a separate rule about breezes and pits for each square, such as $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.
- In English, it seems easy enough to say, “Squares adjacent to pits are breezy.”
- The syntax and semantics of English somehow make it possible to describe the environment concisely.

8.1.1 The language of thought:

The modern view of natural language is that it serves a as a medium for **communication** rather than pure representation. When a speaker points and says, “Look!” the listener comes to know that, say, Superman has finally appeared over the rooftops. Yet we would not want to say that the sentence “Look!” represents that fact. Rather, the meaning of the sentence depends both on the sentence itself and on the **context** in which the sentence was spoken. Natural languages also suffer from **ambiguity**, a problem for representation AMBIGUITY language. From the viewpoint of formal logic, representing the same knowledge in two different ways makes absolutely no difference; the same facts will be derivable from either representation. In practice, however, one representation might require fewer steps to derive a conclusion, meaning that a reasoner with limited resources could get to the conclusion using one representation but not the other. For *nondeductive* tasks such as learning from experience, outcomes are *necessarily* dependent on the form of the representations used. We show in Chapter 18 that when a learning program considers two possible theories of the world, both of which are consistent with all the data, the most common way of breaking the tie is to choose the most succinct theory—and that depends on the language used to represent theories. Thus, the influence of language on thought is unavoidable for any agent that does learning.

8.1.2 Combining the best of formal and natural languages

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 . . .

The language of **first-order logic**, whose syntax and semantics, is built around objects and relations. 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. Special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular *times* and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) "first class" status within the logic, rather than simply defining them within the knowledge base. **Higher-order logic** views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about *all* relations.

A logic can also be characterized by its **epistemological commitments**—the possible states of knowledge that it allows with respect to each fact. In both propositional and first order logic, a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion. These logics therefore have three possible states of knowledge regarding any sentence. Systems using **probability theory**, on the other hand, can have any *degree of belief*, ranging from 0 (total disbelief) to 1 (total belief). The ontological and epistemological commitments of five different logics are summarized in Figure 8.1.

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.

8.2 SYNTAX AND SEMANTICS OF FIRST-ORDER LOGIC

8.2.1 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 have objects. The domain of a model is the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object.

A relation is just the set of tuples of objects that are related.

- Unary Relation: Relations relates to single Object.
- Binary Relation: Relation Relates to multiple objects Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object.

For Example:

Richard the Lionheart, King of England from 1189 to 1199; His younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; crown Unary Relation: John is a king Binary Relation : crown is on head of john , Richard is brother ofjohn The unary "left leg" function includes the following mappings:

(Richard the Lionheart) ->Richard's left leg (King John) ->Johns left Leg

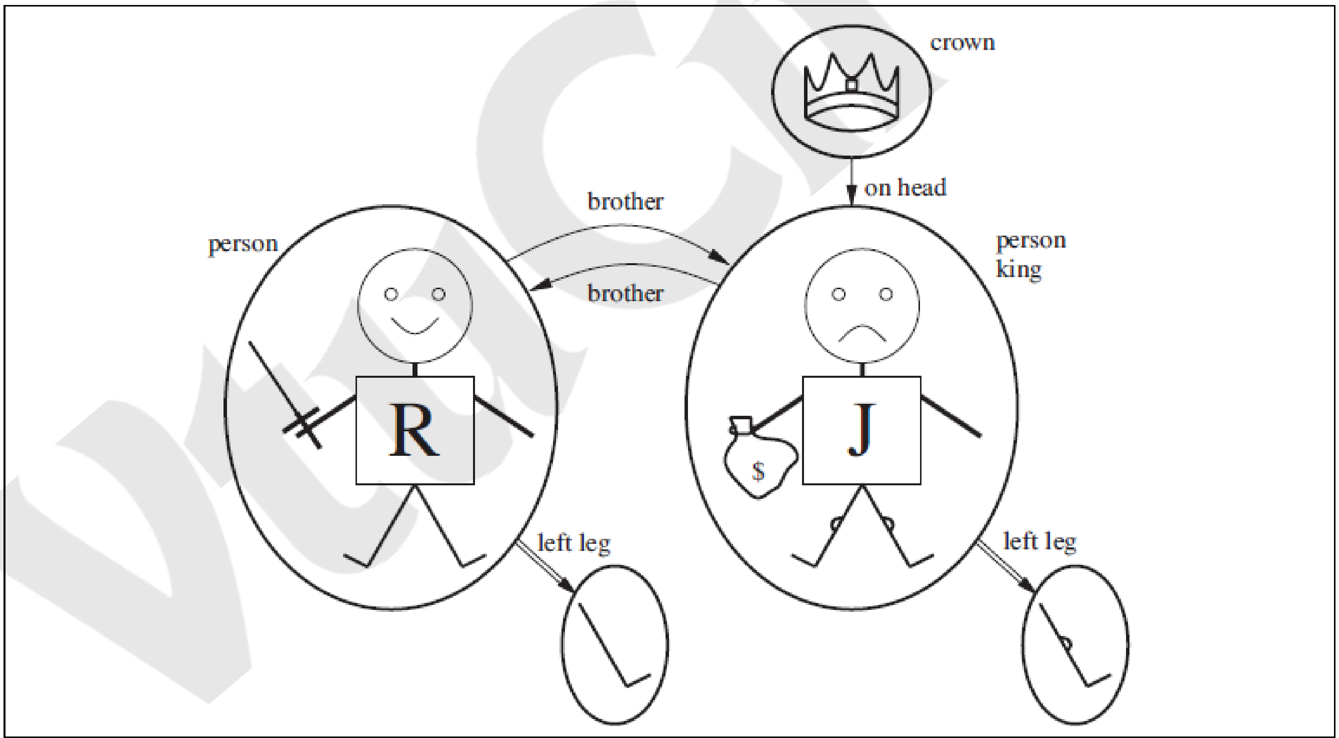


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

8.2.2 Symbols and interpretations

Symbols are the basic syntactic elements of first-order logic. Symbols stand for objects, relations, and functions.

The symbols are of three kinds:

- Constant symbols which stand for objects; Example: John, Richard
- Predicate symbols, which stand for relations; Example: On Head, Person, King, and Crown
- Function symbols, which stand for functions. Example: left leg

Symbols will begin with uppercase letters.

Interpretation The semantics must relate sentences to models in order to determine truth. For this to happen, we need an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.

For Example:

- Richard refers to Richard the Lionheart and John refers to the evil king John.
- Brother refers to the brotherhood relation
- On Head refers to the "on head relation that holds between the crown and King John;
- Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns.
- Left Leg refers to the "left leg" function,

The truth of any sentence is determined by a model and an interpretation for the sentence's symbols. Therefore, entailment, validity, and so on are defined in terms of all possible models and all possible interpretations. The number of domain elements in each model may be unbounded-for example, the domain elements may be integers or real numbers. Hence, the number of possible models is an bounded, as is the number of interpretations.

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

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

8.2.3 Term:

A term is a logical expression that refers to an object. Constant symbols are therefore terms. Complex Terms A complex term is just a complicated kind of name. A complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol For example: "King John's left leg" Instead of using a constant symbol, we use LeftLeg(John). The formal semantics of terms:

Consider a term f (tl,. . . , t,). The function symbol frefers to some function in the model (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 Fapplied to dl, . . . , d,.

For example,: the LeftLeg function symbol refers to the function " (King John) -+ John's left leg" 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.

8.2.4 Atomic sentences

An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms: For Example: Brother(Richard, John).

Atomic sentences can have complex terms as arguments. For Example: Married (Father(Richard), Mother(John)).

An atomic sentence is true in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments

8.2.5 Complex sentences

Complex sentences can be constructed using logical Connectives, just as in propositional calculus. For Example:

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

Quantifiers

Quantifiers express properties of entire collections of objects, instead of enumerating the objects by name.
First-order logic contains two standard quantifiers:

1. Universal Quantifier
2. Existential Quantifier

Universal Quantifier
Universal quantifier is defined as follows:
"Given a 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.

For Example: "All kings are persons," is written in first-order logic as
∀xKing(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. Variables are lowercase letters. A variable is a term all by itself, and can also serve as the argument of a function A term with no variables is called a ground term.

Assume we can extend the interpretation in different 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.

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.

"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." ∃x is pronounced "There exists an x such that . . ." or "For some x . . .".

For example, that King John has a crown on his head, we write ∃xCrown(x) ∧OnHead(x, John)

Given assertions:

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. The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Just as ⇒appears to be the natural connective to use with ∀, ∧is the natural connective to use with ∃.

Nested quantifiers

One can express more complex sentences using multiple quantifiers.

For example, "Brothers are siblings" can be written as ∀x∀y Brother (x, y) ⇒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∀x, y Sibling(x, y) ⇔Sibling(y, x).

In other cases we will have mixtures. For example: 1. "Everybody loves somebody" means that for every person, there is someone that person loves: ∀x∃y Loves(x, y) . 2. On the other hand, to say "There is someone who is loved by everyone," we write ∃y∀x Loves(x, y) .

Connections between ∀and ∃

Universal and Existential quantifiers are actually intimately connected with each other, through negation.

Example assertions: 1. " Everyone dislikes medicine" is the same as asserting " there does not exist someone who likes medicine" , and vice versa: "∀x ¬Likes(x, medicine)" is equivalent to "¬∃x Likes(x, medicine)". 2. "Everyone likes ice cream" means that " there is no one who does not like ice cream" : ∀xLikes(x, IceCream) is equivalent to ¬∃x ¬Likes(x, 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:

$$\begin{array}{llll} \forall x\ \neg P & \equiv \neg\exists x\ P & \neg(P \lor Q) & \equiv \neg P \land \neg Q \\ \neg\forall x\ P & \equiv \exists x\ \neg P & \neg(P \land Q) & \equiv \neg P \lor \neg Q \\ \forall x\ P & \equiv \neg\exists x\ \neg P & P \land Q & \equiv \neg(\neg P \lor \neg Q) \\ \exists x\ P & \equiv \neg\forall x\ \neg P & P \lor Q & \equiv \neg(\neg P \land \neg Q)\,. \end{array}$$

Thus, we do not really need both ∀ and ∃, just as we do not really need both ∧ and ∨. Still, readability is more important than parsimony, so we will keep both of the quantifiers.

8.2.7 Equality

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

For example,

"Father (John) =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. It can also be used with negation to insist that two terms are not the same object.

For example,

"Richard has at least two brothers" can be written as, ∃x, y Brother (x,Richard) ∧Brother (y,Richard) ∧¬ (x=y) The sentence ∃x, y Brother (x,Richard) ∧Brother (y,Richard) does not have the intended meaning. In particular, it is true only in the model where Richard has only one brother considering the extended interpretation in which both x and y are assigned to King John. The addition of ¬(x=y) rules out such models.

8.3 USING FIRST-ORDER LOGIC

8.3.1 Assertions and queries in first-order logic

Assertions:

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions.

For example,

John is a king, TELL (KB, King (John)). Richard is a person. TELL (KB, Person (Richard)). All kings are persons: TELL (KB, ∀x King(x) ⇒Person(x)).

Asking Queries:

We can ask questions of the knowledge base using ASK. Questions asked with ASK are called queries or goals.

For example,

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

Substitution or binding list 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 values.

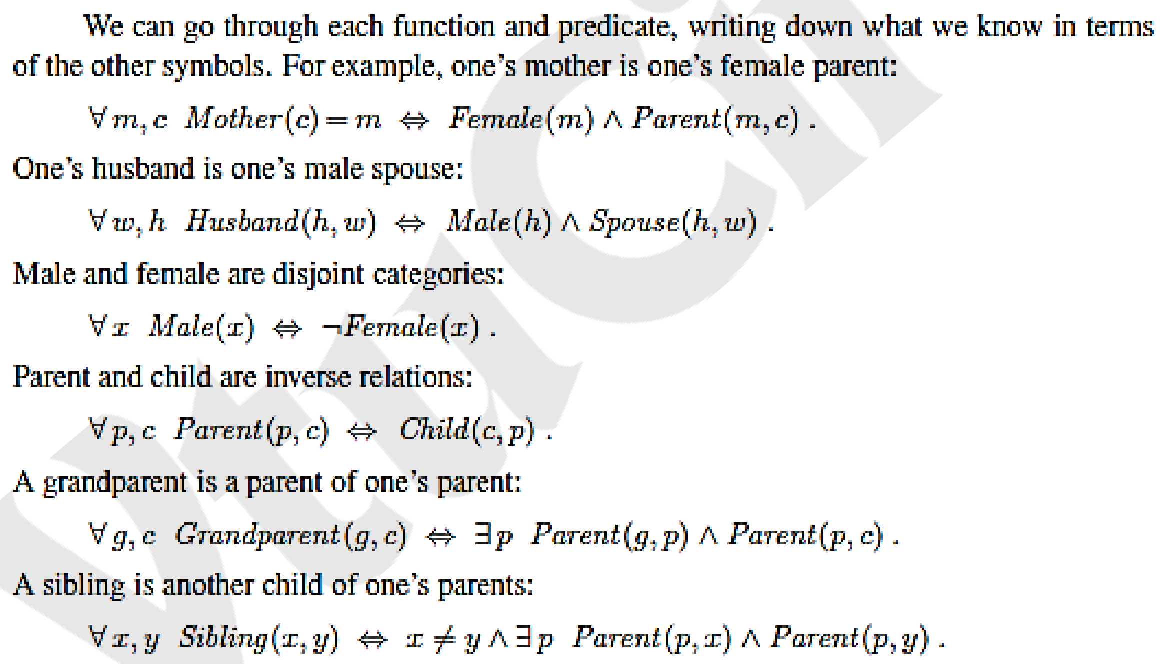
8.3.2 The kinship domain

The objects in Kinship 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 use functions for Mother and Father, because every person has exactly one of each of these.

We can represent each function and predicate, writing down what we know in terms of the other symbols.



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:

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

One’s husband is one’s male spouse:

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

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:

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

A sibling is another child of one’s parents:

$$\forall x, y \;\; Sibling(x, y) \;\Leftrightarrow\; x \neq y \land \exists p \;\; Parent(p, x) \land Parent(p, y) \,.$$

Axioms:

Each of these sentences can be viewed as an axiom of the kinship domain. Axioms are commonly associated with purely mathematical domains. They provide the basic factual information from which useful conclusions can be derived.

Kinship axioms are also definitions; they have the form ∀x, y P(x, y) ⇔. . ..

The axioms define the Mother function, Husband, Male, Parent, Grandparent, and Sibling predicates in terms of other predicates.

Our definitions "bottom out" at a basic set of predicates (Child, Spouse, and Female) in terms of which the others are ultimately defined. This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions.

Theorems:

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:

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

It is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return true. From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time.

Axioms : Axioms without Definition

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

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

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:

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

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. The answers to these questions will then be theorems that follow from the axioms

<u>8.3.3 Numbers, sets, and lists</u>

Number theory

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

- predicate NatNum that will be true of natural numbers;
- one PEANO AXIOMS constant symbol, 0;
- One function symbol, S (successor).
- The Peano axioms define natural numbers and addition.

Natural numbers are defined recursively: $NatNum(0)\ .\ \forall n\ NatNum(n) \Rightarrow NatNum(S(n))\ .$
That is, 0 is a natural number, and for every object n, if n is a natural number, then S(n) is a natural number.
So the natural numbers are 0, S(0), S(S(0)), and so on.
We also need axioms to constrain the successor function: $\forall n\ 0 \mathrel{!=} S(n)\ .\ \forall m, n\ m \mathrel{!=} n \Rightarrow S(m) \mathrel{!=} S(n)\ .$

Now we can define addition in terms of the successor function: ∀m NatNum(m) ⇒ + (0, m) = m . ∀m, n NatNum(m) ∧ NatNum(n) ⇒ + (S(m), n) = S(+(m, n))
The first of these axioms says that adding 0 to any natural number m gives m itself. Addition is represented using the binary function symbol "+" in the term + (m, 0);

To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write S(n) as n + 1, so the second axiom becomes :

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

This axiom reduces addition to repeated application of the successor function. Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

Sets

The domain of sets is also fundamental to mathematics as well as to commonsense reasoning. Sets can be represented as individual sets, including empty sets.

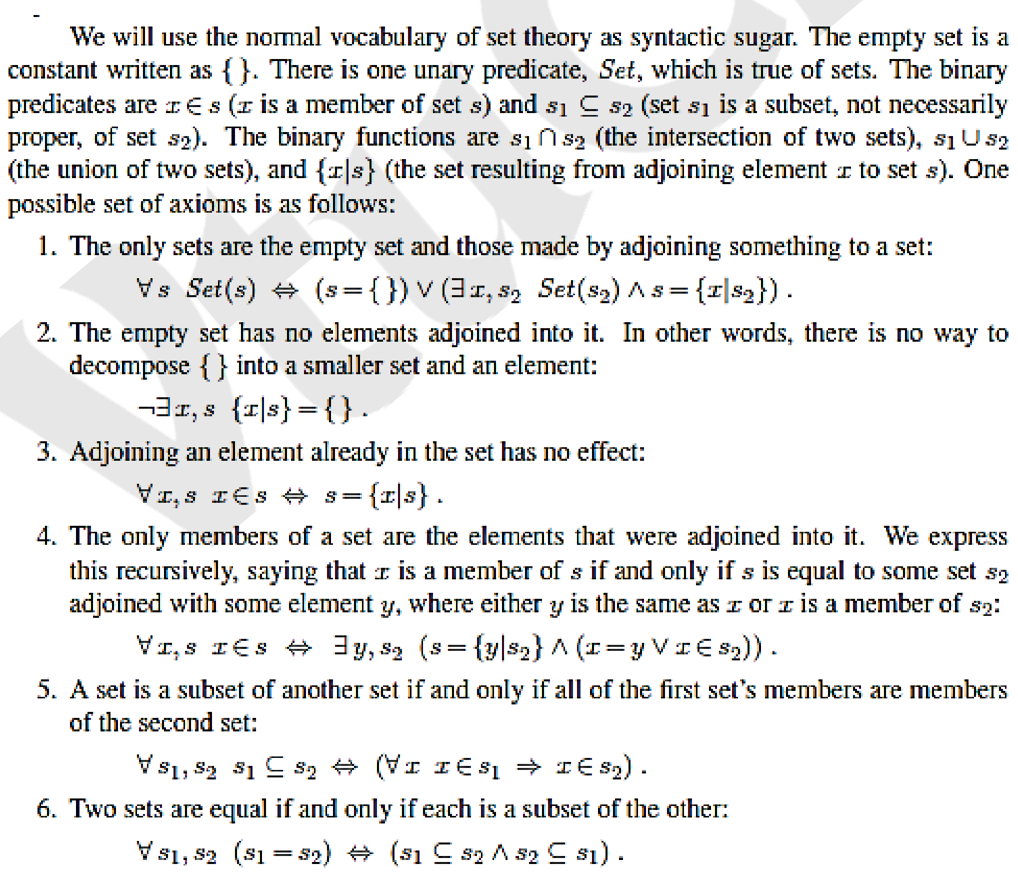
Sets can be built up by:

- adding an element to a set or
- Taking the union or intersection of two sets.

Operations that can be performed on sets are:

- To know whether an element is a member of a set.
- Distinguish sets from objects that are not sets.

Vocabulary of set theory:



We will use the normal vocabulary of set theory as syntactic sugar. The empty set is a constant written as { }. There is 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:
$$\forall s\ Set(s) \Leftrightarrow (s = \{\}) \lor (\exists x, s_2\ Set(s_2) \land s = \{x|s_2\}) .$$
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. We express this recursively, saying that x is a member of s if and only if s is equal to some set s_2 adjoined with some element y, where either y is the same as x or x is a member of s_2:
$$\forall x, s\ x \in s \Leftrightarrow \exists y, s_2\ (s = \{y|s_2\} \land (x = y \lor x \in s_2)) .$$
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:
$$\forall s_1, s_2\ s_1 \subseteq s_2 \Leftrightarrow (\forall x\ x \in s_1 \Rightarrow x \in s_2) .$$
6. Two sets are equal if and only if each is a subset of the other:
$$\forall s_1, s_2\ (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \land s_2 \subseteq s_1) .$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cap s_2) \;\Leftrightarrow\; (x \in s_1 \land x \in s_2) .$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cup s_2) \;\Leftrightarrow\; (x \in s_1 \lor x \in s_2) .$$

8.3.4 The wumpus world

The wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what.We use integers for time steps. A typical percept sentence would be

$$Percept([Stench, Breeze, Glitter, None, None], 5) .$$

Here, *Percept* is a binary predicate, and *Stench* and so on are constants placed in a list. 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

$$\text{ASKVARS}(\exists\, a \;\; BestAction(a, 5)) ,$$

which returns a binding list such as $\{a/Grab\}$. The agent program can then return *Grab* as the action to take. The raw percept data implies certain facts about the current state. For example:

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

and so on. These rules exhibit a trivial form of the reasoning process called **perception**, which we study in depth in Chapter 24. Notice the quantification over time t. In propositional logic, we would need copies of each sentence for each time step.

Simple "reflex" behavior can also be implemented by quantified implication sentences. For example, we have

$$\forall t \;\; Glitter(t) \;\Rightarrow\; BestAction(Grab, t) .$$

8.4 KNOWLEDGE ENGINEERING IN FIRST-ORDER LOGIC

A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.

8.4.1 The knowledge-engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. *Identify the task.* The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions or is it required to answer questions only about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers. This step is analogous to the PEAS process for designing agents in Chapter 2.
2. *Assemble the relevant knowledge.* The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know—a process called **knowledge acquisition**. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works.

   For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.) For real domains, the issue of relevance can be quite difficult—for example, a system for simulating VLSI designs might or might not need to take into account stray capacitances and skin effects.

3. *Decide on a vocabulary of predicates, functions, and constants.* That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering *style*. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the **ontology** of the domain. The word *ontology* means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.
4. *Encode general knowledge about the domain.* The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.
5. *Encode a description of the specific problem instance.* If the ontology is well thought out, this step will be easy. It will involve writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is supplied with additional sentences in the same way that traditional programs are supplied with input data.
6. *Pose queries to the inference procedure and get answers.* This is where the reward is: we can let the inference procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.
7. *Debug the knowledge base.* Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct *for the knowledge base as written*, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue. Missing axioms or axioms that are too weak can be easily identified by noticing places where the chain of reasoning stops unexpectedly. For example, if the knowledge base includes a diagnostic rule (see Exercise 8.13) for finding the wumpus,

$$\forall s \ Smelly(s) \Rightarrow Adjacent(Home(Wumpus), s) ,$$

8.4.2 The electronic circuits domain

We will develop an ontology and knowledge base that allow us to reason about digital circuits of the kind shown in Figure 8.6. We follow the seven-step process for knowledge engineering.

Identify the task

There are many reasoning tasks associated with digital circuits. At the highest level, one analyzes the circuit's functionality. For example, does the circuit in Figure 8.6 actually add properly? If all the inputs are high, what is the output of gate A2? Questions about the circuit's structure are also interesting. For example, what are all the gates connected to the first input terminal? Does the circuit contain feedback loops? These will be our tasks. There are more detailed levels of analysis, including those related to timing delays, circuit area, power consumption, production cost, and so on. Each of these levels would require additional knowledge.

Assemble the relevant knowledge

What do we know about digital circuits? For our purposes, they are composed of wires and gates. Signals flow along wires to the input terminals of gates, and each gate produces a signal on the output terminal that flows along another wire. To determine what these signals will be, we need to know how the gates transform their input signals. There are four types of gates: AND, OR, and XOR gates have two input terminals, and NOT gates have one. All gates have one output terminal. Circuits, like gates, have input and output terminals.

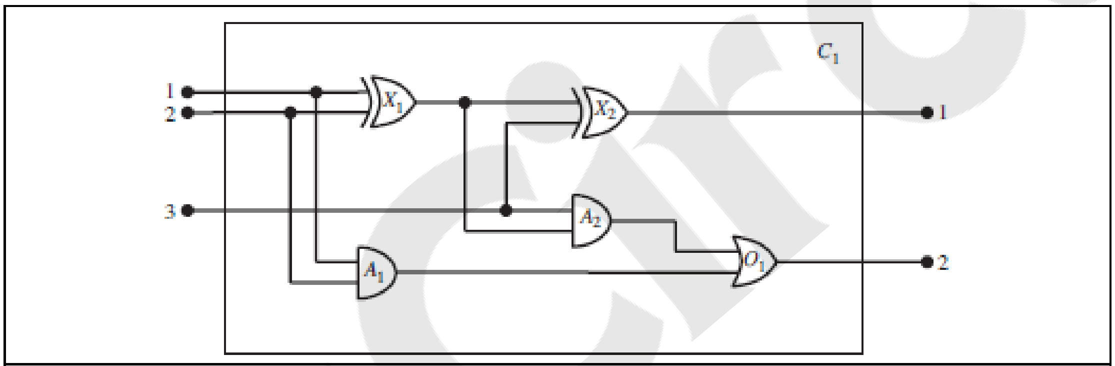


Figure 8.6 A digital circuit C1, purporting to be a one-bit full adder. The first two inputs are the two bits to be added, and the third input is a carry bit. The first output is the sum, and the second output is a carry bit for the next adder. The circuit contains two XOR gates, two AND gates, and one OR gate.

Decide on a vocabulary

We now know that we want to talk about circuits, terminals, signals, and gates. The next step is to choose functions, predicates, and constants to represent them.

First, we need to be able to distinguish gates from each other and from other objects. Each gate is represented an object named by a constant, about which we assert that it is a gate with, say, Gate(X1). The behavior of each gate is determined by its type: one of the constants AND,OR, XOR, or NOT. Because a gate has exactly one type, a function is appropriate: Type(X1)=XOR. Circuits, like gates, are identified by a predicate: Circuit(C1).

Next we consider terminals, which are identified by the predicate Terminal (x). A gate or circuit can have one or more input terminals and one or more output terminals. We use the function In(1,X1) to denote the first input terminal for gate X1. A similar function Out is used for output terminals. The function Arity(c, i, j) says that circuit c has i input and j output terminals. The connectivity between gates can be represented by a predicate, Connected, which takes two terminals as arguments, as in Connected(Out(1,X1), In(1,X2)).

Finally, we need to know whether a signal is on or off. One possibility is to use a unary predicate, On(t), which is true when the signal at a terminal is on. This makes it a little difficult, however, to pose questions such as "What are all the possible values of the signals at the output terminals of circuit C1 ?" We therefore introduce as objects two signal values, 1 and 0, and a function Signal (t) that denotes the signal value for the terminal t.

Encode general knowledge of the domain

One sign that we have a good ontology is that we require only a few general rules, which can be stated clearly and concisely. These are all the axioms we will need:

1. If two terminals are connected, then they have the same signal:
   $\forall t_1, t_2 \;\; Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \Rightarrow$
   $Signal(t_1) = Signal(t_2)$.
2. The signal at every terminal is either 1 or 0:
   $\forall t \;\; Terminal(t) \Rightarrow Signal(t) = 1 \vee Signal(t) = 0$.
3. Connected is commutative:
   $\forall t_1, t_2 \;\; Connected(t_1, t_2) \Leftrightarrow Connected(t_2, t_1)$.
4. There are four types of gates:
   $\forall g \;\; Gate(g) \wedge k = Type(g) \Rightarrow k = AND \vee k = OR \vee k = XOR \vee k = NOT$.
5. An AND gate's output is 0 if and only if any of its inputs is 0:
   $\forall g \;\; Gate(g) \wedge Type(g) = AND \Rightarrow$
   $Signal(Out(1, g)) = 0 \Leftrightarrow \exists n \;\; Signal(In(n, g)) = 0$.
6. An OR gate's output is 1 if and only if any of its inputs is 1:
   $\forall g \;\; Gate(g) \wedge Type(g) = OR \Rightarrow$
   $Signal(Out(1, g)) = 1 \Leftrightarrow \exists n \;\; Signal(In(n, g)) = 1$.
7. An XOR gate's output is 1 if and only if its inputs are different:
   $\forall g \;\; Gate(g) \wedge Type(g) = XOR \Rightarrow$
   $Signal(Out(1, g)) = 1 \Leftrightarrow Signal(In(1, g)) \neq Signal(In(2, g))$.
8. A NOT gate's output is different from its input:
   $\forall g \;\; Gate(g) \wedge (Type(g) = NOT) \Rightarrow$
   $Signal(Out(1, g)) \neq Signal(In(1, g))$.
9. The gates (except for NOT) have two inputs and one output.
   $\forall g \;\; Gate(g) \wedge Type(g) = NOT \Rightarrow Arity(g, 1, 1)$.
   $\forall g \;\; Gate(g) \wedge k = Type(g) \wedge (k = AND \vee k = OR \vee k = XOR) \Rightarrow$
   $Arity(g, 2, 1)$
10. A circuit has terminals, up to its input and output arity, and nothing beyond its arity:
    $\forall c, i, j \;\; Circuit(c) \wedge Arity(c, i, j) \Rightarrow$
    $\forall n \;\; (n \leq i \Rightarrow Terminal(In(c, n))) \wedge (n > i \Rightarrow In(c, n) = Nothing) \wedge$
    $\forall n \;\; (n \leq j \Rightarrow Terminal(Out(c, n))) \wedge (n > j \Rightarrow Out(c, n) = Nothing)$
11. Gates, terminals, signals, gate types, and *Nothing* are all distinct.
    $\forall g, t \;\; Gate(g) \wedge Terminal(t) \Rightarrow$
    $g \neq t \neq 1 \neq 0 \neq OR \neq AND \neq XOR \neq NOT \neq Nothing$.
12. Gates are circuits.
    $\forall g \;\; Gate(g) \Rightarrow Circuit(g)$

Encode the specific problem instance

The circuit shown in Figure 8.6 is encoded as circuit C_1 with the following description. First, we categorize the circuit and its component gates:

$$\begin{array}{l}
Circuit(C_1) \land Arity(C_1, 3, 2) \\
Gate(X_1) \land Type(X_1) = XOR \\
Gate(X_2) \land Type(X_2) = XOR \\
Gate(A_1) \land Type(A_1) = AND \\
Gate(A_2) \land Type(A_2) = AND \\
Gate(O_1) \land Type(O_1) = OR\ .
\end{array}$$

Then, we show the connections between them:

$$\begin{array}{ll}
Connected(Out(1, X_1), In(1, X_2)) & Connected(In(1, C_1), In(1, X_1)) \\
Connected(Out(1, X_1), In(2, A_2)) & Connected(In(1, C_1), In(1, A_1)) \\
Connected(Out(1, A_2), In(1, O_1)) & Connected(In(2, C_1), In(2, X_1)) \\
Connected(Out(1, A_1), In(2, O_1)) & Connected(In(2, C_1), In(2, A_1)) \\
Connected(Out(1, X_2), Out(1, C_1)) & Connected(In(3, C_1), In(2, X_2)) \\
Connected(Out(1, O_1), Out(2, C_1)) & Connected(In(3, C_1), In(1, A_2))\ .
\end{array}$$

9.INFERENCE IN FIRST-ORDER LOGIC

9.1 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.

∀ x King(x) ∧ Greedy(x) ⇒ Evil(x)

Then it seems quite permissible to infer any of the following sentences:

King(John) ∧ Greedy(John) ⇒ Evil(John)
King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)
King(Father (John)) ∧ Greedy(Father (John)) ⇒ Evil(Father (John)) .

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

$$\frac{\forall v \ \ \alpha}{\textsc{Subst}(\{v/g\}, \alpha)}$$

For any variable v and ground term *g.*

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

∀ x King(x) ∧ Greedy(x) ⇒ Evil(x)

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

King(John) ∧ Greedy(John) ⇒ Evil(John)
King(Richard) ∧ Greedy(Richard) ⇒ 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,

$$\frac{\exists v \ \ \alpha}{\textsc{Subst}(\{v/k\}, \alpha)} .$$

For example, from the sentence

$$\exists x \ \ Crown(x) \land OnHead(x, John)$$

we can infer the sentence

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

As long as *C1* 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 old in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

9.1.2 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

$$\begin{aligned}&\forall x\ King(x) A\ Greedy(x) \Rightarrow Evil(x)\\&King(John)\\&Greedy(John)\\&Brother(Richard, John).\end{aligned}$$

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

$$\begin{aligned}&King(John)\ A\ Greedy(John) \Rightarrow Evil(John),\\&King(Richard) A\ Greedy(Richard) \Rightarrow Evil(Richard)\end{aligned}$$

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

9.2 UNIFICATION AND LIFTING

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

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

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

$$\begin{aligned}&King(John) \wedge Greedy(John) \Rightarrow Evil(John)\\&King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)\\&King(Peter) \wedge Greedy(Peter) \Rightarrow Evil(Peter)\\&King(Sharon) \wedge Greedy(Sharon) \Rightarrow Evil(Sharon)\end{aligned}$$

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,

$$\frac{p_1{}',\ \ p_2{}',\ \ \ldots,\ \ p_n{}',\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}\ .$$

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 our example,

$$\begin{array}{ll} p_1{}' \text{ is } King(John) & p_1 \text{ is } King(x) \\ p_2{}' \text{ is } Greedy(y) & p_2 \text{ is } Greedy(x) \\ \theta \text{ is } \{x/John, y/John\} & q \text{ is } Evil(x) \\ \text{SUBST}(\theta, q) \text{ is } Evil(John)\ . & \end{array}$$

9.2.2 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($x, y, \theta$) returns a substitution to make $x$ and $y$ identical
   inputs: $x$, a variable, constant, list, or compound expression
           $y$, a variable, constant, list, or compound expression
           $\theta$, the substitution built up so far (optional, defaults to empty)

   if $\theta$ = failure then return failure
   else if $x = y$ then return $\theta$
   else if VARIABLE?($x$) then return UNIFY-VAR($x, y, \theta$)
   else if VARIABLE?($y$) then return UNIFY-VAR($y, x, \theta$)
   else if COMPOUND?($x$) and COMPOUND?($y$) then
       return UNIFY($x$.ARGS, $y$.ARGS, UNIFY($x$.OP, $y$.OP, $\theta$))
   else if LIST?($x$) and LIST?($y$) then
       return UNIFY($x$.REST, $y$.REST, UNIFY($x$.FIRST, $y$.FIRST, $\theta$))
   else return failure
```

```
function UNIFY-VAR($var, x, \theta$) returns a substitution

   if $\{var/val\} \in \theta$ then return UNIFY($val, x, \theta$)
   else if $\{x/val\} \in \theta$ then return UNIFY($var, val, \theta$)
   else if OCCUR-CHECK?($var, x$) then return failure
   else return add $\{var/x\}$ to $\theta$
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ 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 OP field picks out the function symbol F and the ARGS field 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**.**

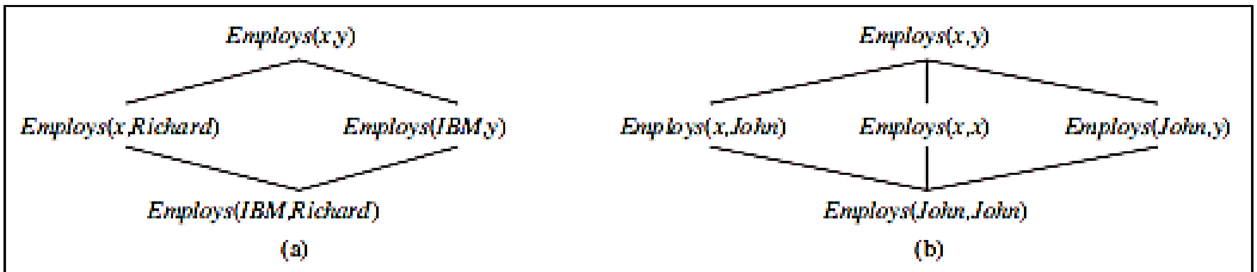


Figure 9.2 (a) The subsumption lattice whose lowest node is *Employs*(*IBM*, *Richard*). (b) The subsumption lattice for the sentence *Employs*(*John*, *John*).

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.

9.3 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:

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

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.

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":

American(x) ∧Weapon(y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal (x) ----------(1)

"Nono . . . has some missiles." The sentence 3 x *Owns (Nono,* .rc) *A Missile* (x) is transformed into two definite clauses by Existential Elimination, introducing a new constant *M1:*

Owns (Nono, M1) ----------------- (2)

Missile (Ml) ------------------------- (3)

"All of its missiles were sold to it by Colonel West":

***Missile (x) A Owns (Nono, x) =>Sells (West,* z*, Nono)* ----------------- (4)**

We will also need to know that missiles are weapons:

Missile (x) =>Weapon (x) ---------- (5)

We must know that an enemy of America counts as "hostile":

***Enemy (x, America) =>Hostile(x)* ----------- (6)**

"West, who is American":

***American (West)* --------------- (7)**

"The country Nono, an enemy of America ":

Enemy (Nono, America) ------------ (8)

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) an*d *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($KB, \alpha$) returns a substitution or $false$
   inputs: $KB$, the knowledge base, a set of first-order definite clauses
           $\alpha$, the query, an atomic sentence
   local variables: $new$, the new sentences inferred on each iteration

   repeat until $new$ is empty
      $new \leftarrow \{\}$
      for each $rule$ in $KB$ do
         $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
         for each $\theta$ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
                     for some $p'_1, \ldots, p'_n$ in $KB$
            $q' \leftarrow$ SUBST($\theta, q$)
            if $q'$ does not unify with some sentence already in $KB$ or $new$ then
               add $q'$ to $new$
               $\phi \leftarrow$ UNIFY($q', \alpha$)
               if $\phi$ is not $fail$ then return $\phi$
      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.

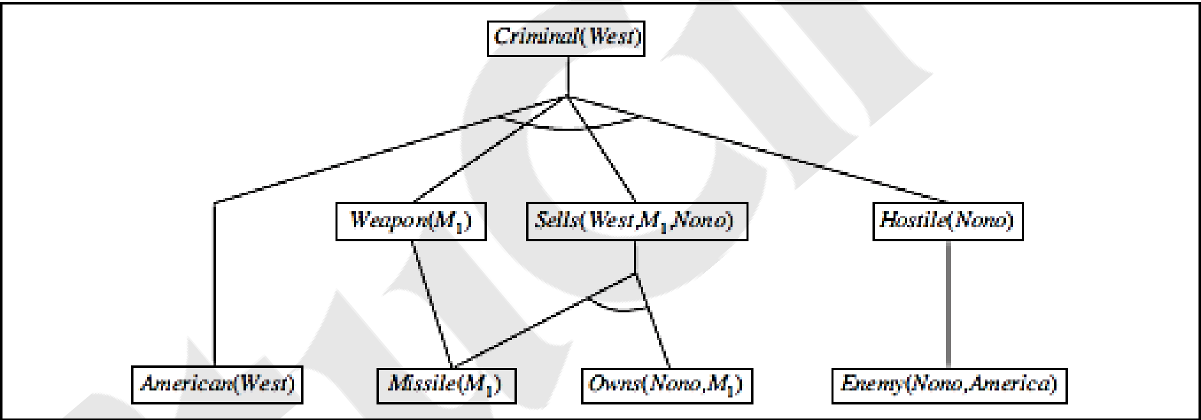


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Efficient forward chaining:

The above given forward chaining algorithm was lack with efficiency due to the 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 every finite-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 least one 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.