

10 KNOWLEDGE REPRESENTATION

In which we show how to use first-order logic to represent the most important aspects of the real world, such as action, space, time, mental events, and shopping.

The last three chapters described the technology for knowledge-based agents: the syntax, semantics, and proof theory of propositional and first-order logic, and the implementation of agents that use these logics. In this chapter we address the question of what *content* to put into such an agent's knowledge base—how to represent facts about the world.

Section 10.1 introduces the idea of a general ontology, which organizes everything in the world into a hierarchy of categories. Section 10.2 covers the basic categories of objects, substances, and measures. Section 10.3 discusses representations for actions, which are central to the construction of knowledge-based agents, and also explains the more general notion of **events**, or space–time chunks. Section 10.4 discusses knowledge about beliefs, and Section 10.5 brings all the knowledge together in the context of an Internet shopping environment. Sections 10.6 and 10.7 cover specialized reasoning systems for representing uncertain and changing knowledge.

10.1 ONTOLOGICAL ENGINEERING

In “toy” domains, the choice of representation is not that important; it is easy to come up with a consistent vocabulary. On the other hand, complex domains such as shopping on the Internet or controlling a robot in a changing physical environment require more general and flexible representations. This chapter shows how to create these representations, concentrating on general concepts—such as *Actions*, *Time*, *Physical Objects*, and *Beliefs*—that occur in many different domains. Representing these abstract concepts is sometimes called **ontological engineering**—it is related to the **knowledge engineering** process described in Section 8.4, but operates on a grander scale.

The prospect of representing *everything* in the world is daunting. Of course, we won't actually write a complete description of everything—that would be far too much for even a 1000–page textbook—but we will leave placeholders where new knowledge for any domain can fit in. For example, we will define what it means to be a physical object, and the details

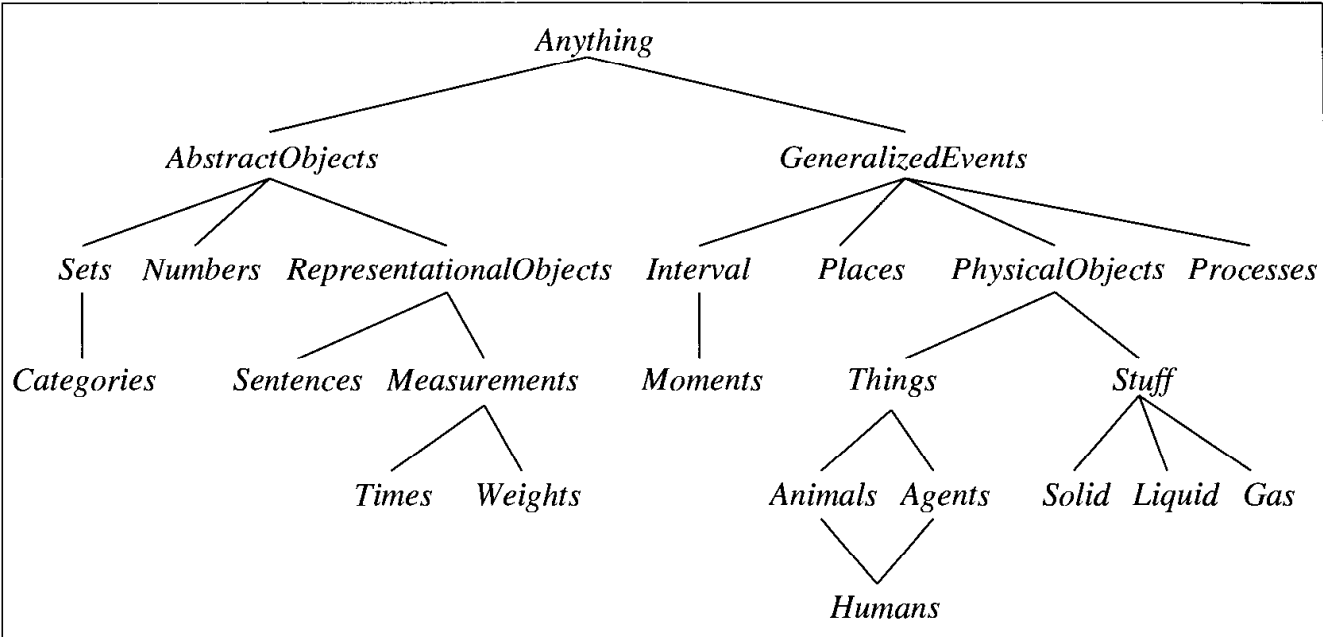


Figure 10.1 The upper ontology of the world, showing the topics to be covered later in the chapter. Each arc indicates that the lower concept is a specialization of the upper one.

UPPER ONTOLOGY

of different types of objects—robots, televisions, books, or whatever—can be filled in later. The general framework of concepts is called an **upper ontology**, because of the convention of drawing graphs with the general concepts at the top and the more specific concepts below them, as in Figure 10.1.

Before considering the ontology further, we should state one important caveat. We have elected to use first-order logic to discuss the content and organization of knowledge. Certain aspects of the real world are hard to capture in FOL. The principal difficulty is that almost all generalizations have exceptions, or hold only to a degree. For example, although “tomatoes are red” is a useful rule, some tomatoes are green, yellow, or orange. Similar exceptions can be found to almost all the general statements in this chapter. The ability to handle exceptions and uncertainty is extremely important, but is orthogonal to the task of understanding the general ontology. For this reason, we will delay the discussion of exceptions until Section 10.6, and the more general topic of uncertain information until Chapter 13.

Of what use is an upper ontology? Consider again the ontology for circuits in Section 8.4. It makes a large number of simplifying assumptions. For example, time is omitted completely. Signals are fixed, and do not propagate. The structure of the circuit remains constant. If we wanted to make this more general, consider signals at particular times, and include the wire lengths and propagation delays. This would allow us to simulate the timing properties of the circuit, and indeed such simulations are often carried out by circuit designers. We could also introduce more interesting classes of gates, for example by describing the technology (TTL, MOS, CMOS, and so on) as well as the input/output specification. If we wanted to discuss reliability or diagnosis, we would include the possibility that the structure of the circuit or the properties of the gates might change spontaneously. To account for stray capacitances, we would need to move from a purely topological representation of connectivity to a more realistic description of geometric properties.

If we look at the wumpus world, similar considerations apply. Although we do include time, it has a very simple structure: Nothing happens except when the agent acts, and all changes are instantaneous. A more general ontology, better suited for the real world, would allow for simultaneous changes extended over time. We also used a *Pit* predicate to say which squares have pits. We could have allowed for different kinds of pits by having several individuals belonging to the class of pits, each having different properties. Similarly, we might want to allow for other animals besides wumpuses. It might not be possible to pin down the exact species from the available percepts, so we would need to build up a wumpus-world biological taxonomy to help the agent predict behavior from scanty clues.

For any special-purpose ontology, it is possible to make changes like these to move toward greater generality. An obvious question then arises: do all these ontologies converge on a general-purpose ontology? After centuries of philosophical and computational investigation, the answer is “Possibly.” In this section, we will present one version, representing a synthesis of ideas from those centuries. There are two major characteristics of general-purpose ontologies that distinguish them from collections of special-purpose ontologies:

- A general-purpose ontology should be applicable in more or less any special-purpose domain (with the addition of domain-specific axioms). This means that, as far as possible, no representational issue can be finessed or brushed under the carpet.
- In any sufficiently demanding domain, different areas of knowledge must be *unified*, because reasoning and problem solving could involve several areas simultaneously. A robot circuit-repair system, for instance, needs to reason about circuits in terms of electrical connectivity and physical layout, and about time, both for circuit timing analysis and estimating labor costs. The sentences describing time therefore must be capable of being combined with those describing spatial layout and must work equally well for nanoseconds and minutes and for angstroms and meters.

After we present the general ontology we use it to describe the Internet shopping domain. This domain is more than adequate to exercise our ontology, and leaves plenty of scope for the reader to do some creative knowledge representation of his or her own. Consider for example that the Internet shopping agent must know about myriad subjects and authors to buy books at Amazon.com, about all sorts of foods to buy groceries at Peapod.com, and about everything one might find at a garage sale to hunt for bargains at Ebay.com.¹

10.2 CATEGORIES AND OBJECTS

CATEGORIES



The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories*. For example, a shopper might have the goal of buying a basketball, rather than a *particular* basketball such as *BB₉*. Categories also serve to make predictions about objects once they are classified. One infers the presence of certain

¹ We apologize if, due to circumstances beyond our control, some of these online stores are no longer functioning by the time you read this.

objects from perceptual input, infers category membership from the perceived properties of the objects, and then uses category information to make predictions about the objects. For example, from its green, mottled skin, large size, and ovoid shape, one can infer that an object is a watermelon; from this, one infers that it would be useful for fruit salad.

There are two choices for representing categories in first-order logic: predicates and objects. That is, we can use the predicate $Basketball(b)$, or we can **reify** the category as an object, $Basketballs$. We could then say $Member(b, Basketballs)$ (which we will abbreviate as $b \in Basketballs$) to say that b is a member of the category of basketballs. We say $Subset(Basketballs, Balls)$ (abbreviated as $Basketballs \subset Balls$) to say that $Basketballs$ is a subcategory, or subset, of $Balls$. So you can think of a category as being the set of its members, or you can think of it as a more complex object that just happens to have the *Member* and *Subset* relations defined for it.

INHERITANCE

Categories serve to organize and simplify the knowledge base through **inheritance**. If we say that all instances of the category *Food* are edible, and if we assert that *Fruit* is a subclass of *Food* and *Apples* is a subclass of *Fruit*, then we know that every apple is edible. We say that the individual apples **inherit** the property of edibility, in this case from their membership in the *Food* category.

TAXONOMY

Subclass relations organize categories into a **taxonomy**, or **taxonomic hierarchy**. Taxonomies have been used explicitly for centuries in technical fields. For example, systematic biology aims to provide a taxonomy of all living and extinct species; library science has developed a taxonomy of all fields of knowledge, encoded as the Dewey Decimal system; and tax authorities and other government departments have developed extensive taxonomies of occupations and commercial products. Taxonomies are also an important aspect of general commonsense knowledge.

First-order logic makes it easy to state facts about categories, either by relating objects to categories or by quantifying over their members:

- An object is a member of a category. For example:
 $BB_9 \in Basketballs$
- A category is a subclass of another category. For example:
 $Basketballs \subset Balls$
- All members of a category have some properties. For example:
 $x \in Basketballs \Rightarrow Round(x)$
- Members of a category can be recognized by some properties. For example:
 $Orange(x) \wedge Round(x) \wedge Diameter(x) = 9.5'' \wedge x \in Balls \Rightarrow x \in Basketballs$
- A category as a whole has some properties. For example:
 $Dogs \in DomesticatedSpecies$

Notice that because *Dogs* is a category and is a member of *DomesticatedSpecies*, the latter must be a category of categories. One can even have categories of categories of categories, but they are not much use.

Although subclass and member relations are the most important ones for categories, we also want to be able to state relations between categories that are not subclasses of each other. For example, if we just say that *Males* and *Females* are subclasses of *Animals*, then

DISJOINT

EXHAUSTIVE
DECOMPOSITION

PARTITION

we have not said that a male cannot be a female. We say that two or more categories are **disjoint** if they have no members in common. And even if we know that males and females are disjoint, we will not know that an animal that is not a male must be a female, unless we say that males and females constitute an **exhaustive decomposition** of the animals. A disjoint exhaustive decomposition is known as a **partition**. The following examples illustrate these three concepts:

Disjoint(*{ Animals, Vegetables }*)
ExhaustiveDecomposition(*{ Americans, Canadians, Mexicans },*
NorthAmericans)
Partition(*{ Males, Females }, Animals*) .

(Note that the *ExhaustiveDecomposition* of *NorthAmericans* is not a *Partition*, because some people have dual citizenship.) The three predicates are defined as follows:

$Disjoint(s) \Leftrightarrow (\forall c_1, c_2 \ c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow Intersection(c_1, c_2) = \{ \})$
 $ExhaustiveDecomposition(s, c) \Leftrightarrow (\forall i \ i \in c \Leftrightarrow \exists c_2 \ c_2 \in s \wedge i \in c_2)$
 $Partition(s, c) \Leftrightarrow Disjoint(s) \wedge ExhaustiveDecomposition(s, c)$.

Categories can also be *defined* by providing necessary and sufficient conditions for membership. For example, a bachelor is an unmarried adult male:

$x \in Bachelors \Leftrightarrow Unmarried(x) \wedge x \in Adults \wedge x \in Males$.

As we discuss in the sidebar on natural kinds, strict logical definitions for categories are neither always possible nor always necessary.

Physical composition

The idea that one object can be part of another is a familiar one. One's nose is part of one's head, Romania is part of Europe, and this chapter is part of this book. We use the general *PartOf* relation to say that one thing is part of another. Objects can be grouped into *PartOf* hierarchies, reminiscent of the *Subset* hierarchy:

PartOf(*Bucharest, Romania*)
PartOf(*Romania, EasternEurope*)
PartOf(*EasternEurope, Europe*)
PartOf(*Europe, Earth*) .

The *PartOf* relation is transitive and reflexive; that is,

$PartOf(x, y) \wedge PartOf(y, z) \Rightarrow PartOf(x, z)$.
 $PartOf(x, x)$.

Therefore, we can conclude *PartOf*(*Bucharest, Earth*).

COMPOSITE OBJECT

Categories of **composite objects** are often characterized by structural relations among parts. For example, a biped has two legs attached to a body:

$Biped(a) \Rightarrow \exists l_1, l_2, b \ Leg(l_1) \wedge Leg(l_2) \wedge Body(b) \wedge$
 $PartOf(l_1, a) \wedge PartOf(l_2, a) \wedge PartOf(b, a) \wedge$
 $Attached(l_1, b) \wedge Attached(l_2, b) \wedge$
 $l_1 \neq l_2 \wedge [\forall l_3 \ Leg(l_3) \wedge PartOf(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)]$.

The notation for “exactly two” is a little awkward; we are forced to say that there are two legs, that they are not the same, and that if anyone proposes a third leg, it must be the same as one of the other two. In Section 10.6, we will see how a formalism called description logic makes it easier to represent constraints like “exactly two.”

We can define a *PartPartition* relation analogous to the *Partition* relation for categories. (See Exercise 10.6.) An object is composed of the parts in its *PartPartition* and can be viewed as deriving some properties from those parts. For example, the mass of a composite object is the sum of the masses of the parts. Notice that this is not the case with categories, which have no mass, even though their elements might.

It is also useful to define composite objects with definite parts but no particular structure. For example, we might want to say, “The apples in this bag weigh two pounds.” The temptation would be to ascribe this weight to the *set* of apples in the bag, but this would be a mistake because the set is an abstract mathematical concept that has elements but does not have weight. Instead, we need a new concept, which we will call a **bunch**. For example, if the apples are *Apple*₁, *Apple*₂, and *Apple*₃, then

$$\text{BunchOf}(\{ \text{Apple}_1, \text{Apple}_2, \text{Apple}_3 \})$$

denotes the composite object with the three apples as parts (not elements). We can then use the bunch as a normal, albeit unstructured, object. Notice that $\text{BunchOf}(\{x\}) = x$. Furthermore, $\text{BunchOf}(\text{Apples})$ is the composite object consisting of all apples—not to be confused with *Apples*, the category or set of all apples.

We can define *BunchOf* in terms of the *PartOf* relation. Obviously, each element of *s* is part of *BunchOf(s)*:

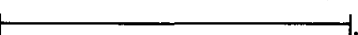
$$\forall x \ x \in s \Rightarrow \text{PartOf}(x, \text{BunchOf}(s)) .$$

Furthermore, *BunchOf(s)* is the smallest object satisfying this condition. In other words, *BunchOf(s)* must be part of any object that has all the elements of *s* as parts:

$$\forall y \ [\forall x \ x \in s \Rightarrow \text{PartOf}(x, y)] \Rightarrow \text{PartOf}(\text{BunchOf}(s), y) .$$

These axioms are an example of a general technique called **logical minimization**, which means defining an object as the smallest one satisfying certain conditions.

Measurements

In both scientific and commonsense theories of the world, objects have height, mass, cost, and so on. The values that we assign for these properties are called **measures**. Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract “measure objects,” such as the *length* that is the length of this line segment: . We can call this length 1.5 inches or 3.81 centimeters. Thus, the same length has different names in our language. Logically, this can be done by combining a **units function** with a number. (An alternative scheme is explored in Exercise 10.8.) If the line segment is called *L*₁, we can write

$$\text{Length}(L_1) = \text{Inches}(1.5) = \text{Centimeters}(3.81) .$$

Conversion between units is done by equating multiples of one unit to another:

$$\text{Centimeters}(2.54 \times d) = \text{Inches}(d) .$$

NATURAL KINDS

Some categories have strict definitions: an object is a triangle if and only if it is a polygon with three sides. On the other hand, most categories in the real world have no clear-cut definition; these are called **natural kind** categories. For example, tomatoes tend to be a dull scarlet; roughly spherical; with an indentation at the top where the stem was; about two to four inches in diameter; with a thin but tough skin; and with flesh, seeds, and juice inside. There is, however, variation: some tomatoes are orange, unripe tomatoes are green, some are smaller or larger than average, and cherry tomatoes are uniformly small. Rather than having a complete definition of tomatoes, we have a set of features that serves to identify objects that are clearly typical tomatoes, but might not be able to decide for other objects. (Could there be a tomato that is furry, like a peach?)

This poses a problem for a logical agent. The agent cannot be sure that an object it has perceived is a tomato, and even if it were sure, it could not be certain which of the properties of typical tomatoes this one has. This problem is an inevitable consequence of operating in partially observable environments.

One useful approach is to separate what is true of all instances of a category from what is true only of typical instances. So in addition to the category *Tomatoes*, we will also have the category *Typical(Tomatoes)*. Here, the *Typical* function maps a category to the subclass that contains only typical instances:

$$\text{Typical}(c) \subseteq c.$$

Most knowledge about natural kinds will actually be about their typical instances:

$$x \in \text{Typical}(\text{Tomatoes}) \Rightarrow \text{Red}(x) \wedge \text{Round}(x).$$

Thus, we can write down useful facts about categories without exact definitions.

The difficulty of providing exact definitions for most natural categories was explained in depth by Wittgenstein (1953), in his book *Philosophical Investigations*. He used the example of *games* to show that members of a category shared “family resemblances” rather than necessary and sufficient characteristics.

The utility of the notion of strict definition was also challenged by Quine (1953). He pointed out that even the definition of “bachelor” as an unmarried adult male is suspect; one might, for example, question a statement such as “the Pope is a bachelor.” While not strictly *false*, this usage is certainly *infelicitous* because it induces unintended inferences on the part of the listener. The tension could perhaps be resolved by distinguishing between logical definitions suitable for internal knowledge representation and the more nuanced criteria for felicitous linguistic usage. The latter may be achieved by “filtering” the assertions derived from the former. It is also possible that failures of linguistic usage serve as feedback for modifying internal definitions, so that filtering becomes unnecessary.

Similar axioms can be written for pounds and kilograms, seconds and days, and dollars and cents. Measures can be used to describe objects as follows:

$$\text{Diameter}(\text{Basketball}_{12}) = \text{Inches}(9.5) .$$

$$\text{ListPrice}(\text{Basketball}_{12}) = \$ (19) .$$

$$d \in \text{Days} \Rightarrow \text{Duration}(d) = \text{Hours}(24) .$$

Note that \$(1) is *not* a dollar bill! One can have two dollar bills, but there is only one object named \$(1). Note also that, while *Inches*(0) and *Centimeters*(0) refer to the same zero length, they are not identical to other zero measures, such as *Seconds*(0).

Simple, quantitative measures are easy to represent. Other measures present more of a problem, because they have no agreed scale of values. Exercises have difficulty, desserts have deliciousness, and poems have beauty, yet numbers cannot be assigned to these qualities. One might, in a moment of pure accountancy, dismiss such properties as useless for the purpose of logical reasoning; or, still worse, attempt to impose a numerical scale on beauty. This would be a grave mistake, because it is unnecessary. The most important aspect of measures is not the particular numerical values, but the fact that measures can be *ordered*.

Although measures are not numbers, we can still compare them using an ordering symbol such as $>$. For example, we might well believe that Norvig's exercises are tougher than Russell's, and that one scores less on tougher exercises:

$$e_1 \in \text{Exercises} \wedge e_2 \in \text{Exercises} \wedge \text{Wrote}(\text{Norvig}, e_1) \wedge \text{Wrote}(\text{Russell}, e_2) \Rightarrow \\ \text{Difficulty}(e_1) > \text{Difficulty}(e_2) .$$

$$e_1 \in \text{Exercises} \wedge e_2 \in \text{Exercises} \wedge \text{Difficulty}(e_1) > \text{Difficulty}(e_2) \Rightarrow \\ \text{ExpectedScore}(e_1) < \text{ExpectedScore}(e_2) .$$

This is enough to allow one to decide which exercises to do, even though no numerical values for difficulty were ever used. (One does, however, have to discover who wrote which exercises.) These sorts of monotonic relationships among measures form the basis for the field of **qualitative physics**, a subfield of AI that investigates how to reason about physical systems without plunging into detailed equations and numerical simulations. Qualitative physics is discussed in the historical notes section.

Substances and objects

The real world can be seen as consisting of primitive objects (particles) and composite objects built from them. By reasoning at the level of large objects such as apples and cars, we can overcome the complexity involved in dealing with vast numbers of primitive objects individually. There is, however, a significant portion of reality that seems to defy any obvious **individuation**—division into distinct objects. We give this portion the generic name **stuff**. For example, suppose I have some butter and an aardvark in front of me. I can say there is one aardvark, but there is no obvious number of “butter-objects,” because any part of a butter-object is also a butter-object, at least until we get to very small parts indeed. This is the major distinction between stuff and things. If we cut an aardvark in half, we do not get two aardvarks (unfortunately).

The English language distinguishes clearly between stuff and things. We say “an aardvark,” but, except in pretentious California restaurants, one cannot say “a butter.” Linguists

COUNT NOUNS

MASS NOUNS

distinguish between **count nouns**, such as aardvarks, holes, and theorems, and **mass nouns**, such as butter, water, and energy. Several competing ontologies claim to handle this distinction. We will describe just one; the others are covered in the historical notes section.

To represent stuff properly, we begin with the obvious. We will need to have as objects in our ontology at least the gross “lumps” of stuff we interact with. For example, we might recognize a lump of butter as the same butter that was left on the table the night before; we might pick it up, weigh it, sell it, or whatever. In these senses, it is an object just like the aardvark. Let us call it *Butter*₃. We will also define the category *Butter*. Informally, its elements will be all those things of which one might say “It’s butter,” including *Butter*₃. With some caveats about very small parts that we will omit for now, any part of a butter-object is also a butter-object:

$$x \in \textit{Butter} \wedge \textit{PartOf}(y, x) \Rightarrow y \in \textit{Butter} .$$

We can now say that butter melts at around 30 degrees centigrade:

$$x \in \textit{Butter} \Rightarrow \textit{MeltingPoint}(x, \textit{Centigrade}(30)) .$$

We could go on to say that butter is yellow, is less dense than water, is soft at room temperature, has a high fat content, and so on. On the other hand, butter has no particular size, shape, or weight. We can define more specialized categories of butter such as *UnsaltedButter*, which is also a kind of stuff. On the other hand, the category *PoundOfButter*, which includes as members all butter-objects weighing one pound, is not a substance! If we cut a pound of butter in half, we do not, alas, get two pounds of butter.

INTRINSIC

EXTRINSIC

What is actually going on is this: there are some properties that are **intrinsic**: they belong to the very substance of the object, rather than to the object as a whole. When you cut a substance in half, the two pieces retain the same set of intrinsic properties—things like density, boiling point, flavor, color, ownership, and so on. On the other hand, **extrinsic** properties are the opposite: properties such as weight, length, shape, function, and so on are not retained under subdivision.

A class of objects that includes in its definition only *intrinsic* properties is then a substance, or mass noun; a class that includes *any* extrinsic properties in its definition is a count noun. The category *Stuff* is the most general substance category, specifying no intrinsic properties. The category *Thing* is the most general discrete object category, specifying no extrinsic properties. All physical objects belong to both categories, so the categories are coextensive—they refer to the same entities.

10.3 ACTIONS, SITUATIONS, AND EVENTS

Reasoning about the results of actions is central to the operation of a knowledge-based agent. Chapter 7 gave examples of propositional sentences describing how actions affect the wumpus world—for example, Equation (7.3) on page 227 states how the agent’s location is changed by forward motion. One drawback of propositional logic is the need to have a different copy of the action description for each time at which the action might be executed. This section describes a representation method that uses first-order logic to avoid that problem.

SITUATION
CALCULUS

SITUATIONS

FLUENTS

The ontology of situation calculus

One obvious way to avoid multiple copies of axioms is simply to quantify over time—to say, “ $\forall t$, such-and-such is the result at $t + 1$ of doing the action at t .” Instead of dealing with explicit times like $t + 1$, we will concentrate in this section on *situations*, which denote the states resulting from executing actions. This approach is called **situation calculus** and involves the following ontology:

- As in Chapter 8, actions are logical terms such as *Forward* and *Turn(Right)*. For now, we will assume that the environment contains only one agent. (If there is more than one, an additional argument can be inserted to say which agent is doing the action.)
- **Situations** are logical terms consisting of the initial situation (usually called S_0) and all situations that are generated by applying an action to a situation. The function *Result(a, s)* (sometimes called *Do*) names the situation that results when action a is executed in situation s . Figure 10.2 illustrates this idea.
- **Fluents** are functions and predicates that vary from one situation to the next, such as the location of the agent or the aliveness of the wumpus. The dictionary says a fluent is something that flows, like a liquid. In this use, it means flowing or changing across situations. By convention, the situation is always the last argument of a fluent. For example, $\neg Holding(G_1, S_0)$ says that the agent is not holding the gold G_1 in the initial situation S_0 . *Age(Wumpus, S_0)* refers to the wumpus’s age in S_0 .
- **Atemporal** or **eternal** predicates and functions are also allowed. Examples include the predicate *Gold(G_1)* and the function *LeftLegOf(Wumpus)*.

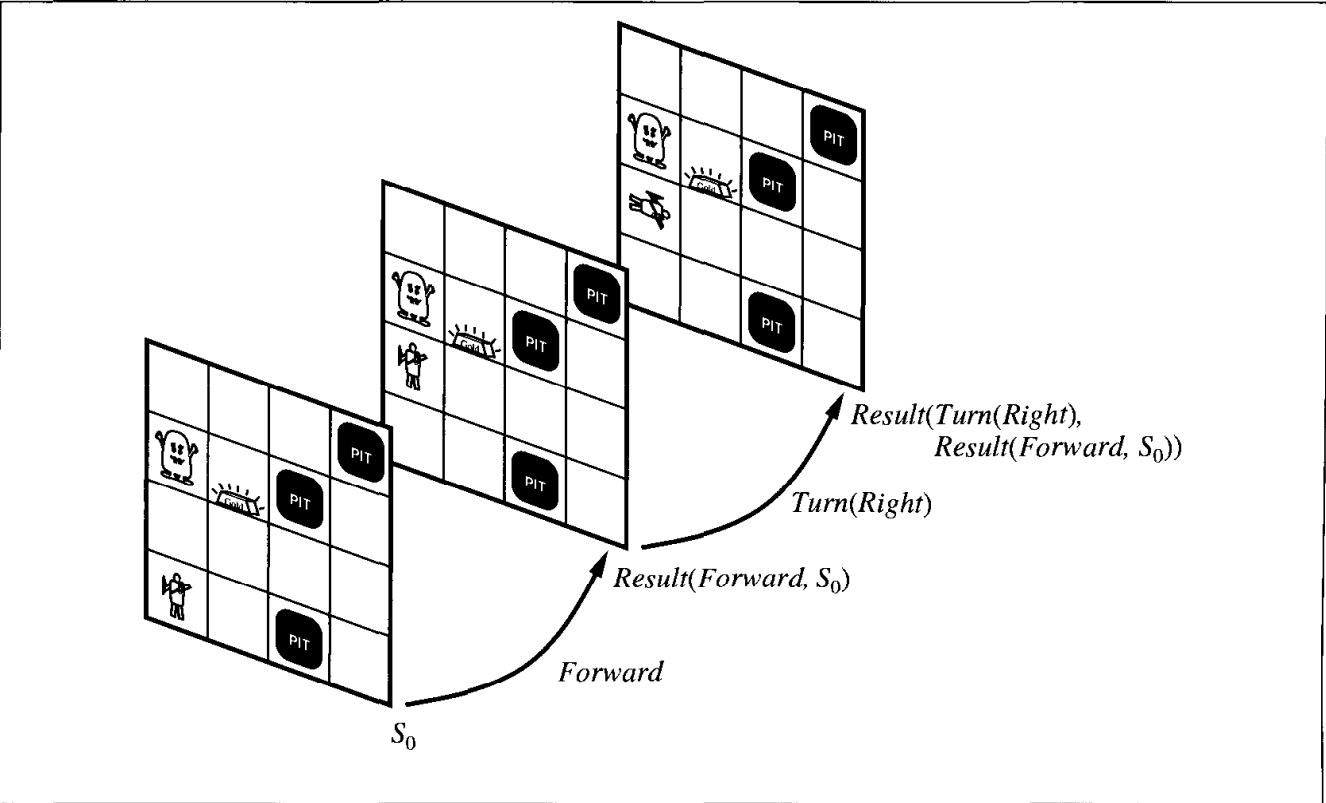


Figure 10.2 In situation calculus, each situation (except S_0) is the result of an action.

In addition to single actions, it is also helpful to reason about action sequences. We can define the results of sequences in terms of the results of individual actions. First, we say that executing an empty sequence leaves the situation unchanged:

$$Result([], s) = s .$$

Executing a nonempty sequence is the same as executing the first action and then executing the rest in the resulting situation:

$$Result([a|seq], s) = Result(seq, Result(a, s)) .$$

A situation calculus agent should be able to deduce the outcome of a given sequence of actions; this is the **projection** task. With a suitable constructive inference algorithm, it should also be able to *find* a sequence that achieves a desired effect; this is the **planning** task.

We will use an example from a modified version of the wumpus world where we do not worry about the agent's orientation and where the agent can *Go* from one location to an adjacent one. Suppose the agent is at $[1, 1]$ and the gold is at $[1, 2]$. The aim is to have the gold in $[1, 1]$. The fluent predicates are $At(o, x, s)$ and $Holding(o, s)$. Then the initial knowledge base might include the following description:

$$At(Agent, [1, 1], S_0) \wedge At(G_1, [1, 2], S_0) .$$

This is not quite enough, however, because it doesn't say what what *isn't* true in S_0 . (See page 355 for further discussion of this point.) The complete description is as follows:

$$At(o, x, S_0) \Leftrightarrow [(o = Agent \wedge x = [1, 1]) \vee (o = G_1 \wedge x = [1, 2])] . \\ \neg Holding(o, S_0) .$$

We also need to state that G_1 is gold and that $[1, 1]$ and $[1, 2]$ are adjacent:

$$Gold(G_1) \wedge Adjacent([1, 1], [1, 2]) \wedge Adjacent([1, 2], [1, 1]) .$$

One would like to be able to prove that the agent achieves its aim by going to $[1, 2]$, grabbing the gold, and returning to $[1, 1]$. That is,

$$At(G_1, [1, 1], Result([Go([1, 1], [1, 2]), Grab(G_1), Go([1, 2], [1, 1])], S_0)) .$$

More interesting is the possibility of constructing a plan to get the gold, which is achieved by answering the query "what sequence of actions results in the gold being at $[1, 1]$?"

$$\exists seq \ At(G_1, [1, 1], Result(seq, S_0)) .$$

Let us see what has to go into the knowledge base for these queries to be answered.

Describing actions in situation calculus

In the simplest version of situation calculus, each action is described by two axioms: a **possibility axiom** that says when it is possible to execute the action, and an **effect axiom** that says what happens when a possible action is executed. We will use $Poss(a, s)$ to mean that it is possible to execute action a in situation s . The axioms have the following form:

$$\text{POSSIBILITY AXIOM: } Preconditions \Rightarrow Poss(a, s) .$$

$$\text{EFFECT AXIOM: } Poss(a, s) \Rightarrow Changes \text{ that result from taking action.}$$

We will present these axioms for the modified wumpus world. To shorten our sentences, we will omit universal quantifiers whose scope is the entire sentence. We assume that the variable s ranges over situations, a ranges over actions, o over objects (including agents), g over gold, and x and y over locations.

The possibility axioms for this world state that an agent can go between adjacent locations, grab a piece of gold in the current location, and release some gold that it is holding:

$$\begin{aligned} At(Agent, x, s) \wedge Adjacent(x, y) &\Rightarrow Poss(Go(x, y), s) . \\ Gold(g) \wedge At(Agent, x, s) \wedge At(g, x, s) &\Rightarrow Poss(Grab(g), s) . \\ Holding(g, s) &\Rightarrow Poss(Release(g), s) . \end{aligned}$$

The effect axioms state that, if an action is possible, then certain properties (fluents) will hold in the situation that results from executing the action. Going from x to y results in being at y , grabbing the gold results in holding the gold, and releasing the gold results in not holding it:

$$\begin{aligned} Poss(Go(x, y), s) &\Rightarrow At(Agent, y, Result(Go(x, y), s)) . \\ Poss(Grab(g), s) &\Rightarrow Holding(g, Result(Grab(g), s)) . \\ Poss(Release(g), s) &\Rightarrow \neg Holding(g, Result(Release(g), s)) . \end{aligned}$$

Having stated these axioms, can we prove that our little plan achieves the goal? Unfortunately not! At first, everything works fine; $Go([1, 1], [1, 2])$ is indeed possible in S_0 and the effect axiom for Go allows us to conclude that the agent reaches $[1, 2]$:

$$At(Agent, [1, 2], Result(Go([1, 1], [1, 2]), S_0)) .$$

Now we consider the $Grab(G_1)$ action. We have to show that it is possible in the new situation—that is,

$$At(G_1, [1, 2], Result(Go([1, 1], [1, 2]), S_0)) .$$

Alas, nothing in the knowledge base justifies such a conclusion. Intuitively, we understand that the agent's Go action should have no effect on the gold's location, so it should still be at $[1, 2]$, where it was in S_0 . *The problem is that the effect axioms say what changes, but don't say what stays the same.*

Representing all the things that stay the same is called the **frame problem**. We must find an efficient solution to the frame problem because, in the real world, almost everything stays the same almost all the time. Each action affects only a tiny fraction of all fluents.

One approach is to write explicit **frame axioms** that *do* say what stays the same. For example, the agent's movements leave other objects stationary unless they are held:

$$At(o, x, s) \wedge (o \neq Agent) \wedge \neg Holding(o, s) \Rightarrow At(o, x, Result(Go(y, z), s)) .$$

If there are F fluent predicates and A actions, then we will need $O(AF)$ frame axioms. On the other hand, if each action has at most E effects, where E is typically much less than F , then we should be able to represent what happens with a much smaller knowledge base of size $O(AE)$. This is the **representational frame problem**. The closely related **inferential frame problem** is to project the results of a t -step sequence of actions in time $O(Et)$, rather than time $O(Ft)$ or $O(AEt)$. We will address each problem in turn. Even then, another problem remains—that of ensuring that *all* necessary conditions for an action's success have been specified. For example, Go fails if the agent dies *en route*. This is the **qualification problem**, for which there is no complete solution.



FRAME PROBLEM

FRAME AXIOM

REPRESENTATIONAL
FRAME PROBLEM
INFERENTIAL FRAME
PROBLEMQUALIFICATION
PROBLEM

Solving the representational frame problem

The solution to the representational frame problem involves just a slight change in viewpoint on how to write the axioms. Instead of writing out the effects of each action, we consider how each fluent predicate evolves over time.³ The axioms we use are called **successor-state axioms**. They have the following form:

SUCCESSOR-STATE
AXIOM

SUCCESSOR-STATE AXIOM:

Action is possible \Rightarrow

*(Fluent is true in result state \Leftrightarrow Action's effect made it true
 \vee It was true before and action left it alone) .*

After the qualification that we are not considering impossible actions, notice that this definition uses \Leftrightarrow , not \Rightarrow . This means that the axiom says that the fluent will be true if *and only if* the right-hand side holds. Put another way, we are specifying the truth value of each fluent in the next state as a function of the action and the truth value in the current state. This means that the next state is completely specified from the current state and hence that there are no additional frame axioms needed.

The successor-state axiom for the agent's location says that the agent is at y after executing an action either if the action is possible and consists of moving to y or if the agent was already at y and the action is not a move to somewhere else:

$$\begin{aligned} Poss(a, s) \Rightarrow \\ (At(Agent, y, Result(a, s)) \Leftrightarrow a = Go(x, y) \\ \vee (At(Agent, y, s) \wedge a \neq Go(y, z))) . \end{aligned}$$

The axiom for *Holding* says that the agent is holding g after executing an action if the action was a grab of g and the grab is possible or if the agent was already holding g , and the action is not releasing it:

$$\begin{aligned} Poss(a, s) \Rightarrow \\ (Holding(g, Result(a, s)) \Leftrightarrow a = Grab(g) \\ \vee (Holding(g, s) \wedge a \neq Release(g))) . \end{aligned}$$

Successor-state axioms solve the representational frame problem because the total size of the axioms is $O(AE)$ literals: each of the E effects of each of the A actions is mentioned exactly once. The literals are spread over F different axioms, so the axioms have average size AE/F .

The astute reader will have noticed that these axioms handle the *At* fluent for the agent, but not for the gold; thus, we still cannot prove that the three-step plan achieves the goal of having the gold in $[1, 1]$. We need to say that an **implicit effect** of an agent moving from x to y is that any gold it is carrying will move too (as will any ants on the gold, any bacteria on the ants, etc.). Dealing with implicit effects is called the **ramification problem**. We will discuss the problem in general later, but for this specific domain, it can be solved by writing a more general successor-state axiom for *At*. The new axiom, which subsumes the previous version, says that an object o is at y if the agent went to y and o is the agent or something the

³ This is essentially the approach we took in building the Boolean circuit-based agent in Chapter 7. Indeed, axioms such as Equation (7.4) and Equation (7.5) can be viewed as successor-state axioms.



IMPLICIT EFFECT

RAMIFICATION
PROBLEM

agent was holding; or if o was already at y and the agent didn't go elsewhere, with o being the agent or something the agent was holding.

$$\begin{aligned} Poss(a, s) \Rightarrow \\ At(o, y, Result(a, s)) \Leftrightarrow & (a = Go(x, y) \wedge (o = Agent \vee Holding(o, s))) \\ & \vee (At(o, y, s) \wedge \neg(\exists z \ y \neq z \wedge a = Go(y, z) \wedge \\ & (o = Agent \vee Holding(o, s)))) . \end{aligned}$$

There is one more technicality: an inference process that uses these axioms must be able to prove nonidentities. The simplest kind of nonidentity is between constants—for example, $Agent \neq G_1$. The general semantics of first-order logic allows distinct constants to refer to the same object, so the knowledge base must include an axiom to prevent this. The **unique names axiom** states a disequality for every pair of constants in the knowledge base. When this is assumed by the theorem prover, rather than written down in the knowledge base, it is called a **unique names assumption**. We also need to state disequalities between action terms: $Go([1, 1], [1, 2])$ is a different action from $Go([1, 2], [1, 1])$ or $Grab(G_1)$. First, we say that each type of action is distinct—that no Go action is a $Grab$ action. For each pair of action names A and B , we would have

$$A(x_1, \dots, x_m) \neq B(y_1, \dots, y_n) .$$

Next, we say that two action terms with the same action name refer to the same action only if they involve all the same objects:

$$A(x_1, \dots, x_m) = A(y_1, \dots, y_m) \Leftrightarrow x_1 = y_1 \wedge \dots \wedge x_m = y_m .$$

These are called, collectively, the **unique action axioms**. The combination of initial state description, successor-state axioms, unique name axiom, and unique action axioms suffices to prove that the proposed plan achieves the goal.

Solving the inferential frame problem

Successor-state axioms solve the representational frame problem, but not the inferential frame problem. Consider a t -step plan p such that $S_t = Result(p, S_0)$. To decide which fluents are true in S_t , we need to consider each of the F frame axioms on each of the t time steps. Because the axioms have average size AE/F , this gives us $O(AEt)$ inferential work. Most of the work involves copying fluents unchanged from one situation to the next.

To solve the inferential frame problem, we have two possibilities. First, we could discard situation calculus and invent a new formalism for writing axioms. This has been done with formalisms such as the **fluent calculus**. Second, we could alter the inference mechanism to handle frame axioms more efficiently. A hint that this should be possible is that the simple approach is $O(AEt)$; why should it depend on the number of actions, A , when we know exactly which one action is executed at each time step? To see how to improve matters, we first look at the format of the frame axioms:

$$\begin{aligned} Poss(a, s) \Rightarrow \\ F_i(Result(a, s)) \Leftrightarrow & (a = A_1 \vee a = A_2 \dots) \\ & \vee F_i(s) \wedge (a \neq A_3) \wedge (a \neq A_4) \dots \end{aligned}$$

UNIQUE NAMES
AXIOM

UNIQUE ACTION
AXIOMS

That is, each axiom mentions several actions that can make the fluent true and several actions that can make it false. We can formalize this by introducing the predicate $PosEffect(a, F_i)$, meaning that action a causes F_i to become true, and $NegEffect(a, F_i)$ meaning that a causes F_i to become false. Then we can rewrite the foregoing axiom schema as:

$$\begin{aligned}
 & Poss(a, s) \Rightarrow \\
 & \quad F_i(Result(a, s)) \Leftrightarrow PosEffect(a, F_i) \vee [F_i(s) \wedge \neg NegEffect(a, F_i)] \\
 & PosEffect(A_1, F_i) \\
 & PosEffect(A_2, F_i) \\
 & NegEffect(A_3, F_i) \\
 & NegEffect(A_4, F_i) .
 \end{aligned}$$

Whether this can be done automatically depends on the exact format of the frame axioms. To make an efficient inference procedure using axioms like this, we need to do three things:

1. Index the $PosEffect$ and $NegEffect$ predicates by their first argument so that when we are given an action that occurs at time t , we can find its effects in $O(1)$ time.
2. Index the axioms so that once you know that F_i is an effect of an action, you can find the axiom for F_i in $O(1)$ time. Then you need not even consider the axioms for fluents that are not an effect of the action.
3. Represent each situation as a previous situation plus a delta. Thus, if nothing changes from one step to the next, we need do no work at all. In the old approach, we would need to do $O(F)$ work in generating an assertion for each fluent $F_i(Result(a, s))$ from the preceding $F_i(s)$ assertions.

Thus at each time step, we look at the current action, fetch its effects, and update the set of true fluents. Each time step will have an average of E of these updates, for a total complexity of $O(Et)$. This constitutes a solution to the inferential frame problem.

Time and event calculus

Situation calculus works well when there is a single agent performing instantaneous, discrete actions. When actions have duration and can overlap with each other, situation calculus becomes somewhat awkward. Therefore, we will cover those topics with an alternative formalism known as **event calculus**, which is based on points in time rather than on situations. (The terms “event” and “action” may be used interchangeably. Informally, “event” connotes a wider class of actions, including ones with no explicit agent. These are easier to handle in event calculus than in situation calculus.)

In event calculus, fluents hold at points in time rather than at situations, and the calculus is designed to allow reasoning over intervals of time. The event calculus axiom says that a fluent is true at a point in time if the fluent was initiated by an event at some time in the past and was not terminated by an intervening event. The *Initiates* and *Terminates* relations play a role similar to the *Result* relation in situation calculus; $Initiates(e, f, t)$ means that the occurrence of event e at time t causes fluent f to become true, while $Terminates(w, f, t)$ means that f ceases to be true. We use $Happens(e, t)$ to mean that event e happens at time t ,

and we use $Clipped(f, t, t_2)$ to mean that f is terminated by some event sometime between t and t_2 . Formally, the axiom is:

EVENT CALCULUS AXIOM:

$$T(f, t_2) \Leftrightarrow \exists e, t \text{ Happens}(e, t) \wedge \text{Initiates}(e, f, t) \wedge (t < t_2) \\ \wedge \neg Clipped(f, t, t_2)$$

$$Clipped(f, t, t_2) \Leftrightarrow \exists e, t_1 \text{ Happens}(e, t_1) \wedge \text{Terminates}(e, f, t_1) \\ \wedge (t < t_1) \wedge (t_1 < t_2) .$$

This gives us functionality that is similar to situation calculus, but with the ability to talk about time points and intervals, so we can say $\text{Happens}(\text{TurnOff}(\text{LightSwitch}_1), 1:00)$ to say that a lightswitch was turned off at exactly 1:00.

Many extensions to event calculus have been made to address problems of indirect effects, events with duration, concurrent events, continuously changing events, nondeterministic effects, causal constraints, and other complications. We will revisit some of these issues in the next subsection. It is fair to say that, at present, completely satisfactory solutions are not yet available for most of them, but no insuperable obstacles have been encountered.

Generalized events

So far, we have looked at two main concepts: actions and objects. Now it is time to see how they fit into an encompassing ontology in which both actions and objects can be thought of as aspects of a physical universe. We think of a particular universe as having both a spatial and a temporal dimension. The wumpus world has its spatial component laid out in a two-dimensional grid and has discrete time; our world has three spatial dimensions and one temporal dimension,³ all continuous. A **generalized event** is composed from aspects of some “space–time chunk”—a piece of this multidimensional space–time universe. This abstraction generalizes most of the concepts we have seen so far, including actions, locations, times, fluents, and physical objects. Figure 10.3 gives the general idea. From now on, we will use the simple term “event” to refer to generalized events.

For example, World War II is an event that took place at various points in space–time, as indicated by the irregularly shaped grey patch. We can break it down into **subevents**:⁴

$$\text{SubEvent}(\text{BattleOfBritain}, \text{WorldWarII}) .$$

Similarly, World War II is a subevent of the 20th century:

$$\text{SubEvent}(\text{WorldWarII}, \text{TwentiethCentury}) .$$

The 20th century is an *interval* of time. Intervals are chunks of space–time that include all of space between two time points. The function $\text{Period}(e)$ denotes the smallest interval enclosing the event e . $\text{Duration}(i)$ is the length of time occupied by an interval, so we can say $\text{Duration}(\text{Period}(\text{WorldWarII})) > \text{Years}(5)$.

³ Some physicists studying string theory argue for 10 dimensions or more, and some argue for a discrete world, but 4-D continuous space–time is an adequate representation for commonsense reasoning purposes.

⁴ Note that SubEvent is a special case of the PartOf relation and is also transitive and reflexive.

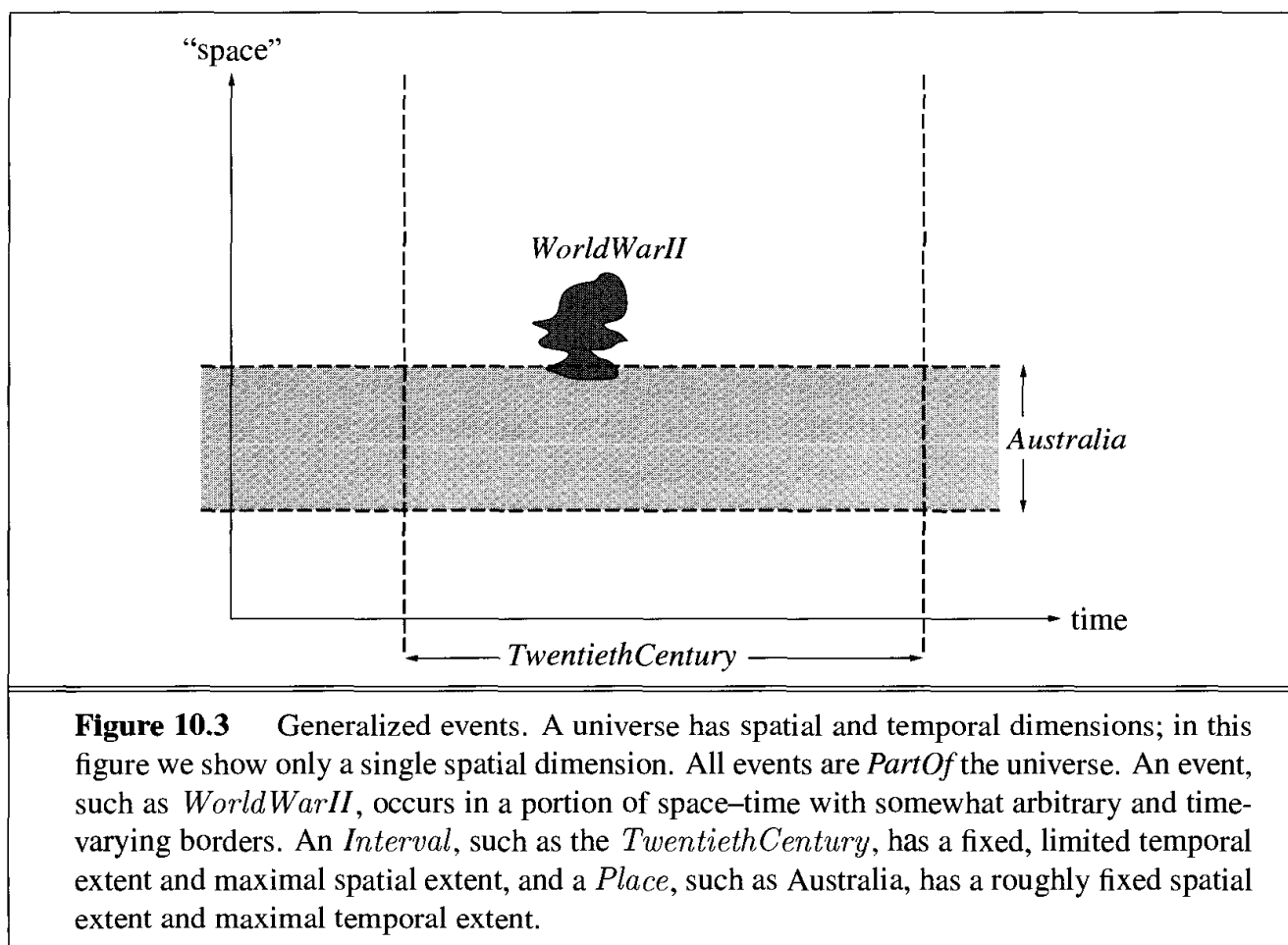


Figure 10.3 Generalized events. A universe has spatial and temporal dimensions; in this figure we show only a single spatial dimension. All events are *PartOf* the universe. An event, such as *WorldWarII*, occurs in a portion of space–time with somewhat arbitrary and time-varying borders. An *Interval*, such as the *TwentiethCentury*, has a fixed, limited temporal extent and maximal spatial extent, and a *Place*, such as *Australia*, has a roughly fixed spatial extent and maximal temporal extent.

Australia is a *place*; a chunk with some fixed spatial borders. The borders can vary over time, due to geological or political changes. We use the predicate *In* to denote the subevent relation that holds when one event’s spatial projection is *PartOf* of another’s:

$In(Sydney, Australia)$.

The function $Location(e)$ denotes the smallest place that encloses the event e .

Like any other sort of object, events can be grouped into categories. For example, *WorldWarII* belongs to the category *Wars*. To say that a civil war occurred in England in the 1640s, we would say

$\exists w \ w \in CivilWars \wedge SubEvent(w, 1640s) \wedge In(Location(w), England)$.

The notion of a category of events answers a question that we avoided when we described the effects of actions in Section 10.3: what exactly do logical terms such as $Go([1, 1], [1, 2])$ refer to? Are they events? The answer, perhaps surprisingly, is *no*. We can see this by considering a plan with two “identical” actions, such as

$[Go([1, 1], [1, 2]), Go([1, 2], [1, 1]), Go([1, 1], [1, 2])]$.

In this plan, $Go([1, 1], [1, 2])$ cannot be the name of an event, because there are *two different events* occurring at different times. Instead, $Go([1, 1], [1, 2])$ is the name of a *category* of events—all those events where the agent goes from $[1, 1]$ to $[1, 2]$. The three-step plan says that instances of these three event categories will occur.

Notice that this is the first time we have seen categories named by complex terms rather than just constant symbols. This presents no new difficulties; in fact, we can use the argument structure to our advantage. Eliminating arguments creates a more general category:

$$Go(x, y) \subseteq GoTo(y) \qquad Go(x, y) \subseteq GoFrom(x) .$$

Similarly, we can add arguments to create more specific categories. For example, to describe actions by other agents, we can add an agent argument. Thus, to say that Shankar flew from New York to New Delhi yesterday, we would write:

$$\exists e \ e \in Fly(Shankar, NewYork, NewDelhi) \wedge SubEvent(e, Yesterday) .$$

The form of this formula is so common that we will create an abbreviation for it: $E(c, i)$ will mean that an element of the category of events c is a subevent of the event or interval i :

$$E(c, i) \Leftrightarrow \exists e \ e \in c \wedge SubEvent(e, i) .$$

Thus, we have:

$$E(Fly(Shankar, NewYork, NewDelhi), Yesterday) .$$

Processes

DISCRETE EVENTS

The events we have seen so far are what we call **discrete events**—they have a definite structure. Shankar’s trip has a beginning, middle, and end. If interrupted halfway, the event would be different—it would not be a trip from New York to New Delhi, but instead a trip from New York to somewhere over Europe. On the other hand, the category of events denoted by $Flying(Shankar)$ has a different quality. If we take a small interval of Shankar’s flight, say, the third 20-minute segment (while he waits anxiously for a second bag of peanuts), that event is still a member of $Flying(Shankar)$. In fact, this is true for any subinterval.

PROCESS

LIQUID EVENT

Categories of events with this property are called **process** categories or **liquid event** categories. Any subinterval of a process is also a member of the same process category. We can employ the same notation used for discrete events to say that, for example, Shankar was flying at some time yesterday:

$$E(Flying(Shankar), Yesterday) .$$

We often want to say that some process was going on *throughout* some interval, rather than just in some subinterval of it. To do this, we use the predicate T :

$$T(Working(Stuart), TodayLunchHour) .$$

$T(c, i)$ means that some event of type c occurred over exactly the interval i —that is, the event begins and ends at the same time as the interval.

The distinction between liquid and nonliquid events is exactly analogous to the difference between substances, or *stuff*, and individual objects. In fact, some have called liquid event types **temporal substances**, whereas things like butter are **spatial substances**.

TEMPORAL
SUBSTANCES
SPATIAL
SUBSTANCES
STATES

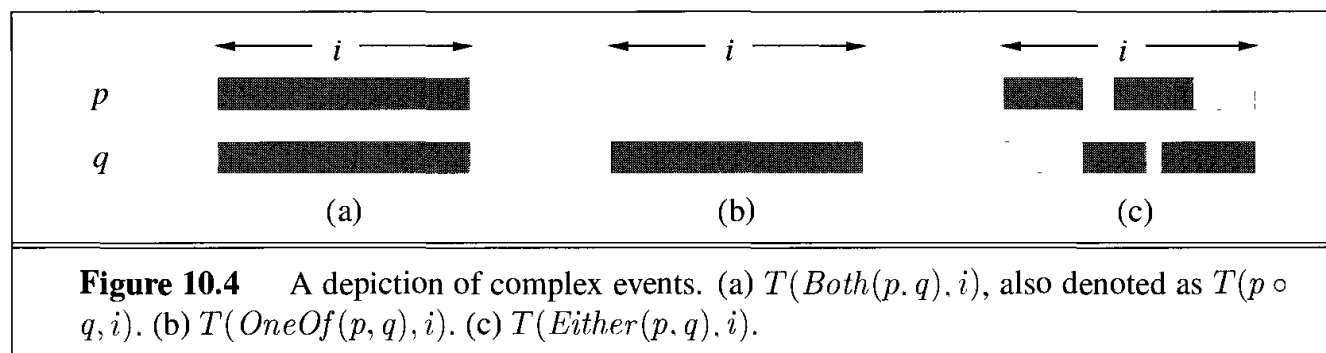
As well as describing processes of continuous change, liquid events can describe processes of continuous non-change. These are often called **states**. For example, “Shankar being in New York” is a category of states that we denote by $In(Shankar, NewYork)$. To say he was in New York all day, we would write

$$T(In(Shankar, NewYork), Today) .$$

We can form more complex states and events by combining primitive ones. This approach is called **fluent calculus**. Fluent calculus reifies combinations of fluents, not just individual fluents. We have already seen a way of representing the event of two things happening at once, namely, the function $Both(e_1, e_2)$. In fluent calculus, this is usually abbreviated with the infix notation $e_1 \circ e_2$. For example, to say that someone walked and chewed gum at the same time, we can write

$$\exists p, i \ (p \in People) \wedge T(Walk(p) \circ ChewGum(p), i) .$$

The “ \circ ” function is commutative and associative, just like logical conjunction. We can also define analogs of disjunction and negation, but we have to be more careful—there are two reasonable ways of interpreting disjunction. When we say “the agent was either walking or chewing gum for the last two minutes” we might mean that the agent was doing one of the actions for the whole interval, or we might mean that the agent was alternating between the two actions. We will use *OneOf* and *Either* to indicate these two possibilities. Figure 10.4 diagrams the complex events.



Intervals

Time is important to any agent that takes action, and there has been much work on the representation of time intervals. We will consider two kinds: moments and extended intervals. The distinction is that only moments have zero duration:

$$\begin{aligned} & Partition(\{Moments, ExtendedIntervals\}, Intervals) \\ & i \in Moments \Leftrightarrow Duration(i) = Seconds(0) . \end{aligned}$$

Next we invent a time scale and associate points on that scale with moments, giving us absolute times. The time scale is arbitrary; we will measure it in seconds and say that the moment at midnight (GMT) on January 1, 1900, has time 0. The functions *Start* and *End* pick out the earliest and latest moments in an interval, and the function *Time* delivers the point on the time scale for a moment. The function *Duration* gives the difference between the end time and the start time.

$$\begin{aligned} Interval(i) & \Rightarrow Duration(i) = (Time(End(i)) - Time(Start(i))) . \\ Time(Start(AD1900)) & = Seconds(0) . \\ Time(Start(AD2001)) & = Seconds(3187324800) . \\ Time(End(AD2001)) & = Seconds(3218860800) . \\ Duration(AD2001) & = Seconds(31536000) . \end{aligned}$$

To make these numbers easier to read, we also introduce a function *Date*, which takes six arguments (hours, minutes, seconds, day, month, and year) and returns a time point:

$$\text{Time}(\text{Start}(\text{AD2001})) = \text{Date}(0, 0, 0, 1, \text{Jan}, 2001)$$

$$\text{Date}(0, 20, 21, 24, 1, 1995) = \text{Seconds}(3000000000) .$$

Two intervals *Meet* if the end time of the first equals the start time of the second. It is possible to define predicates such as *Before*, *After*, *During*, and *Overlap* solely in terms of *Meet*, but it is more intuitive to define them in terms of points on the time scale. (See Figure 10.5 for a graphical representation.)

$$\text{Meet}(i, j) \Leftrightarrow \text{Time}(\text{End}(i)) = \text{Time}(\text{Start}(j)) .$$

$$\text{Before}(i, j) \Leftrightarrow \text{Time}(\text{End}(i)) < \text{Time}(\text{Start}(j)) .$$

$$\text{After}(j, i) \Leftrightarrow \text{Before}(i, j) .$$

$$\begin{aligned} \text{During}(i, j) \Leftrightarrow & \text{Time}(\text{Start}(j)) \leq \text{Time}(\text{Start}(i)) \\ & \wedge \text{Time}(\text{End}(i)) \leq \text{Time}(\text{End}(j)) . \end{aligned}$$

$$\text{Overlap}(i, j) \Leftrightarrow \exists k \text{ During}(k, i) \wedge \text{During}(k, j) .$$

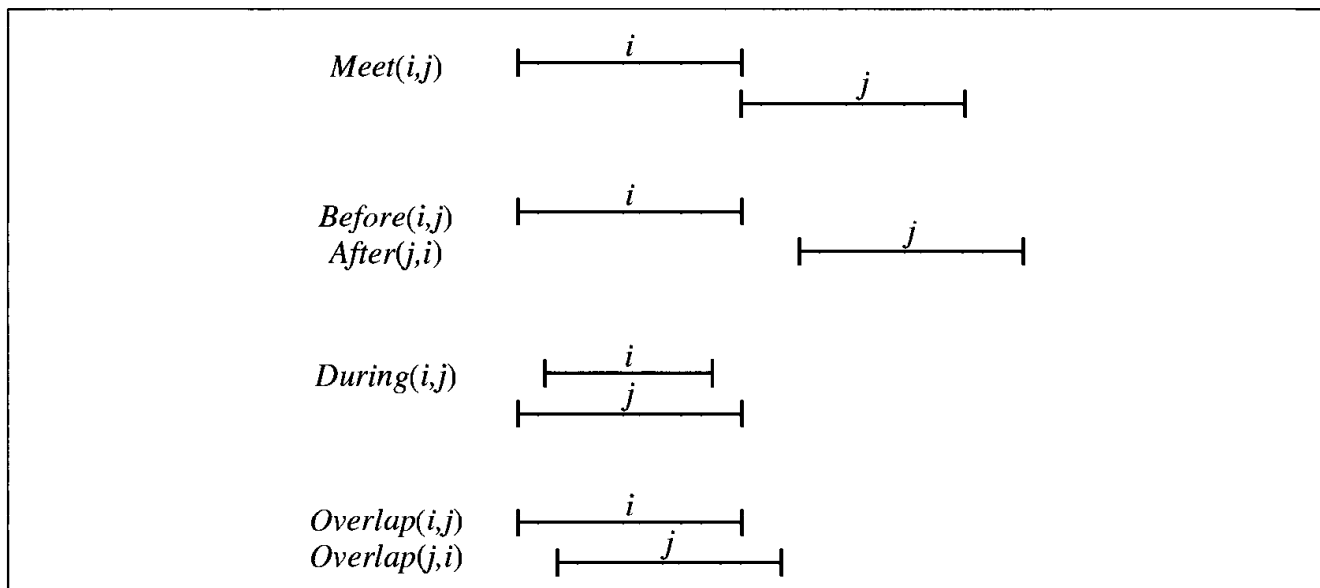


Figure 10.5 Predicates on time intervals.

For example, to say that the reign of Elizabeth II followed that of George VI, and the reign of Elvis overlapped with the 1950s, we can write the following:

$$\text{After}(\text{ReignOf}(\text{ElizabethII}), \text{ReignOf}(\text{George VI})) .$$

$$\text{Overlap}(\text{Fifties}, \text{ReignOf}(\text{Elvis})) .$$

$$\text{Start}(\text{Fifties}) = \text{Start}(\text{AD1950}) .$$

$$\text{End}(\text{Fifties}) = \text{End}(\text{AD1959}) .$$

Fluents and objects

We mentioned that physical objects can be viewed as generalized events, in the sense that a physical object is a chunk of space–time. For example, *USA* can be thought of as an event that began in, say, 1776 as a union of 13 states and is still in progress today as a union of 50.

We can describe the changing properties of *USA* using state fluents. For example, we can say that at some point in 1999 its population was 271 million:

$$E(\text{Population}(\text{USA}, 271000000), \text{AD1999}) .$$

Another property of the USA that changes every four or eight years, barring mishaps, is its president. One might propose that $\text{President}(\text{USA})$ is a logical term that denotes a different object at different times. Unfortunately, this is not possible, because a term denotes exactly one object in a given model structure. (The term $\text{President}(\text{USA}, t)$ can denote different objects, depending on the value of t , but our ontology keeps time indices separate from fluents.) The only possibility is that $\text{President}(\text{USA})$ denotes a single object that consists of different people at different times. It is the object that is George Washington from 1789 to 1796, John Adams from 1796 to 1800, and so on, as in Figure 10.6.

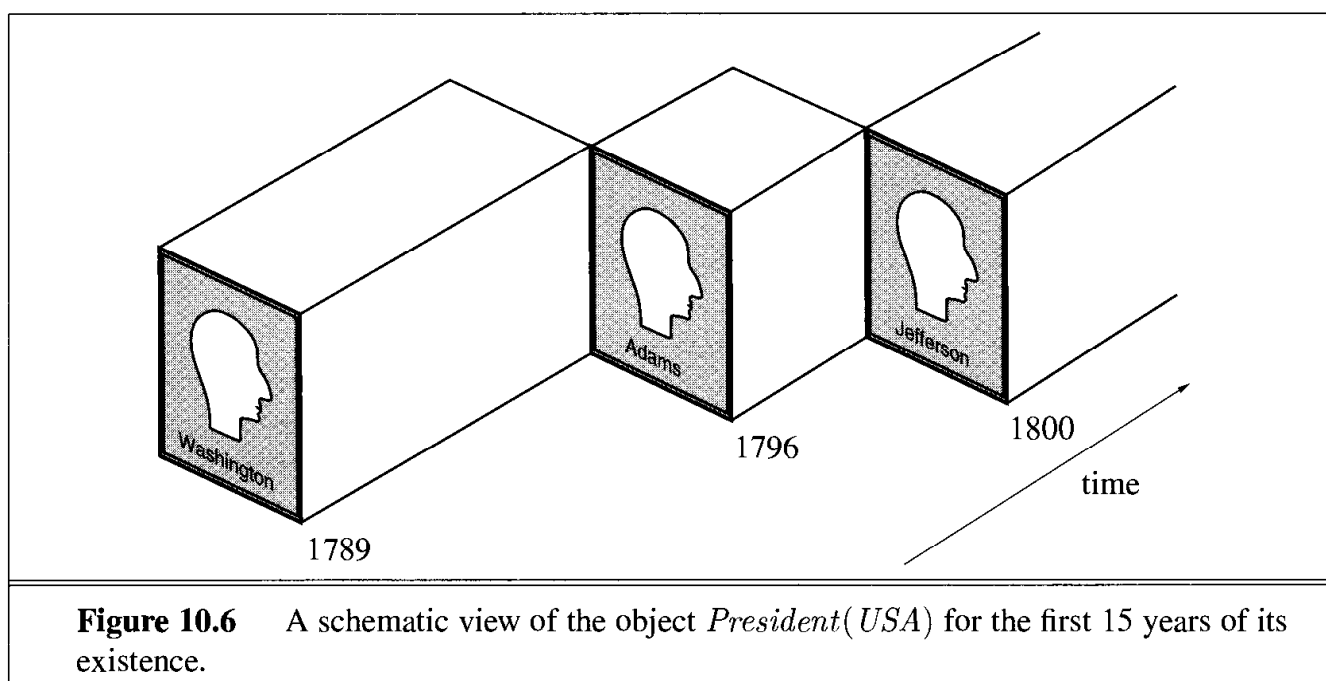


Figure 10.6 A schematic view of the object $\text{President}(\text{USA})$ for the first 15 years of its existence.

To say that George Washington was president throughout 1790, we can write

$$T(\text{President}(\text{USA}) = \text{George Washington}, \text{AD1790}) .$$

We need to be careful, however. In this sentence, “=” must be a function symbol rather than the standard logical operator. The interpretation is *not* that *George Washington* and $\text{President}(\text{USA})$ are logically identical in 1790; logical identity is not something that can change over time. The logical identity exists between the subevents of each object that are defined by the period 1790.

Don’t confuse the physical object *George Washington* with a collections of atoms. George Washington is not logically identical to *any* specific collection of atoms, because the set of atoms of which he is constituted varies considerably over time. He has his short lifetime, and each atom has its own very long lifetime. They intersect for some period, during which the temporal slice of the atom is *PartOf* George, and then they go their separate ways.

10.4 MENTAL EVENTS AND MENTAL OBJECTS

The agents we have constructed so far have beliefs and can deduce new beliefs. Yet none of them has any knowledge *about* beliefs or *about* deduction. For single-agent domains, knowledge about one's own knowledge and reasoning processes is useful for controlling inference. For example, if one knows that one does not know anything about Romanian geography, then one need not expend enormous computational effort trying to calculate the shortest path from Arad to Bucharest. One can also reason about one's own knowledge in order to construct plans that will change it—for example by buying a map of Romania. In multiagent domains, it becomes important for an agent to reason about the mental states of the other agents. For example, a Romanian police officer might well know the best way to get to Bucharest, so the agent might ask for help.

In essence, what we need is a model of the mental objects that are in someone's head (or something's knowledge base) and of the mental processes that manipulate those mental objects. The model should be faithful, but it does not have to be detailed. We do not have to be able to predict how many milliseconds it will take for a particular agent to make a deduction, nor do we have to predict what neurons will fire when an animal is faced with a particular visual stimulus. We will be happy to conclude that the Romanian police officer will tell us how to get to Bucharest if he or she knows the way and believes we are lost.

A formal theory of beliefs

We begin with the relationships between agents and “mental objects”—relationships such as *Believes*, *Knows*, and *Wants*. Relations of this kind are called **propositional attitudes**, because they describe an attitude that an agent can take toward a proposition. Suppose that Lois believes something—that is, $Believes(Lois, x)$. What kind of thing is x ? Clearly, x cannot be a logical sentence. If $Flies(Superman)$ is a logical sentence, we can't say $Believes(Lois, Flies(Superman))$, because only terms (not sentences) can be arguments of predicates. But if *Flies* is a function, then $Flies(Superman)$ is a candidate for being a mental object, and *Believes* can be a relation between an agent and a propositional fluent. Turning a proposition into an object is called **reification**.⁶

This appears to give us what we want: the ability for an agent to reason about the beliefs of agents. Unfortunately, there is a problem with that approach: If Clark and Superman are one and the same (i.e., $Clark = Superman$) then Clark's flying and Superman's flying are one and the same event category, i.e., $Flies(Clark) = Flies(Superman)$. Hence, we must conclude that if Lois believes that Superman can fly, she also believes that Clark can fly, *even if she doesn't believe that Clark is Superman*. That is,

$$(Superman = Clark) \models (Believes(Lois, Flies(Superman)) \Leftrightarrow Believes(Lois, Flies(Clark)))$$

There is a sense in which this is right: Lois does believe of a certain person, who happens

⁶ The term “reification” comes from the Latin word *res*, or thing. John McCarthy proposed the term “thingification,” but it never caught on.

to be called Clark sometimes, that that person can fly. But there is another sense in which it is wrong: if you asked Lois “Can Clark fly?” she would certainly say no. Reified objects and events work fine for the first sense of *Believes*, but for the second sense we need to reify *descriptions* of those objects and events, so that *Clark* and *Superman* can be different descriptions (even though they refer to the same object).

Technically, the property of being able to substitute a term freely for an equal term is called **referential transparency**. In first-order logic, every relation is referentially transparent. We would like to define *Believes* (and the other propositional attitudes) as relations whose second argument is referentially **opaque**—that is, one cannot substitute an equal term for the second argument without changing the meaning.

There are two ways to achieve this. The first is to use a different form of logic called **modal logic**, in which propositional attitudes such as *Believes* and *Knows* become **modal operators** that are referentially opaque. This approach is covered in the historical notes section. The second approach, which we will pursue, is to achieve effective opacity within a referentially transparent language using a **syntactic theory** of mental objects. This means that mental objects are represented by **strings**. The result is a crude model of an agent’s knowledge base as consisting of strings that represent sentences believed by the agent. A string is just a complex term denoting a list of symbols, so the event *Flies*(*Clark*) can be represented by the list of characters $[F, l, i, e, s, (, C, l, a, r, k,)]$, which we will abbreviate as a quoted string, “*Flies*(*Clark*)”. The syntactic theory includes a **unique string axiom** stating that strings are identical if and only if they consist of identical characters. In this way, even if *Clark* = *Superman*, we still have “*Clark*” \neq “*Superman*”.

Now all we have to do is provide a syntax, semantics, and proof theory for the string representation language, just as we did in Chapter 7. The difference is that we have to define them all in first-order logic. We start by defining *Den* as the function that maps a string to the object that it denotes and *Name* as a function that maps an object to a string that is the name of a constant that denotes the object. For example, the denotation of both “*Clark*” and “*Superman*” is the object referred to by the constant symbol *ManOfSteel*, and the name of that object within the knowledge base could be either “*Superman*”, “*Clark*”, or some other constant, such as “*X₁₁*”:

$$\begin{aligned} Den(\text{“Clark”}) &= ManOfSteel \wedge Den(\text{“Superman”}) = ManOfSteel . \\ Name(ManOfSteel) &= \text{“X}_{11}\text{”} . \end{aligned}$$

The next step is to define inference rules for logical agents. For example, we might want to say that a logical agent can do Modus Ponens: if it believes *p* and believes *p* \Rightarrow *q*, then it will also believe *q*. The first attempt at writing this axiom is

$$LogicalAgent(a) \wedge Believes(a, p) \wedge Believes(a, \text{“}p \Rightarrow q\text{”}) \Rightarrow Believes(a, q) .$$

But this is not right because the string “*p* \Rightarrow *q*” contains the letters ‘*p*’ and ‘*q*’ but has nothing to do with the strings that are the values of the variables *p* and *q*. The correct formulation is

$$\begin{aligned} LogicalAgent(a) \wedge Believes(a, p) \wedge Believes(a, Concat(p, \text{“}\Rightarrow\text{”}, q)) \\ \Rightarrow Believes(a, q) . \end{aligned}$$

where *Concat* is a function on strings that concatenates their elements. We will abbreviate *Concat*(*p*, “ \Rightarrow ”, *q*) as “*p* \Rightarrow *q*”. That is, an occurrence of *x* within a string is **unquoted**,

meaning that we are to substitute in the value of the variable x . Lisp programmers will recognize this as the comma/backquote operator, and Perl programmers will recognize it as $\$$ -variable interpolation.

Once we add in the other inference rules besides Modus Ponens, we will be able to answer questions of the form “given that a logical agent knows these premises, can it draw that conclusion?” Besides the normal inference rules, we need some rules that are specific to belief. For example, the following rule says that if a logical agent believes something, then it believes that it believes it.

$$\text{LogicalAgent}(a) \wedge \text{Believes}(a, p) \Rightarrow \text{Believes}(a, \text{“Believes}(\text{Name}(a), p)\text{”}) .$$

Now, according to our axioms, an agent can deduce any consequence of its beliefs infallibly. This is called **logical omniscience**. A variety of attempts have been made to define limited rational agents, which can make a limited number of deductions in a limited time. None is completely satisfactory, but these formulations do allow a highly restricted range of predictions about limited agents.

Knowledge and belief

The relation between believing and knowing has been studied extensively in philosophy. It is commonly said that knowledge is justified true belief. That is, if you believe something for an unassailably good reason, and if it is actually true, then you know it. The “unassailably good reason” is necessary to prevent you from saying “I know this coin flip will come up heads” and being right half the time.

Let $\text{Knows}(a, p)$ mean that agent a knows that proposition p is true. It is also possible to define other kinds of knowing. For example, here is a definition of “knowing whether”:

$$\text{KnowsWhether}(a, p) \Leftrightarrow \text{Knows}(a, p) \vee \text{Knows}(a, \neg p) .$$

Continuing our example, Lois knows whether Clark can fly if she either knows that Clark can fly or knows that he cannot.

The concept of “knowing what” is more complicated. One is tempted to say that an agent knows what Bob’s phone number is if there is some x for which the agent knows $\text{PhoneNumber}(\text{Bob}) = x$. But that is not enough, because the agent might know that Alice and Bob have the same number (i.e., $\text{PhoneNumber}(\text{Bob}) = \text{PhoneNumber}(\text{Alice})$), but if Alice’s number is unknown, that isn’t much help. A better definition of “knowing what” says that the agent has to be aware of some x that is a string of digits and that is Bob’s number:

$$\begin{aligned} \text{KnowsWhat}(a, \text{“PhoneNumber}(\underline{b})\text{”}) &\Leftrightarrow \\ &\exists x \text{ Knows}(a, \text{“}\underline{x} = \text{PhoneNumber}(\underline{b})\text{”}) \wedge x \in \text{DigitStrings} . \end{aligned}$$

Of course, for other questions we have different criteria for what is an acceptable answer. For the question “what is the capital of New York,” an acceptable answer is a proper name, “Albany,” not something like “the city where the state house is.” To handle this, we will make KnowsWhat a three-place relation: it takes an agent, a string representing a term, and a category to which the answer must belong. For example, we might have the following:

$$\begin{aligned} \text{KnowsWhat}(\text{Agent}, \text{“Capital(NewYork)”}, \text{ProperNames}) . \\ \text{KnowsWhat}(\text{Agent}, \text{“PhoneNumber(Bob)”}, \text{DigitStrings}) . \end{aligned}$$

Knowledge, time, and action

In most real situations, an agent will be dealing with beliefs—its own or those of other agents—that change over time. The agent will also have to make plans that involve changes to its own beliefs, such as buying a map to find out how to get to Bucharest. As with other predicates, we can reify *Believes* and talk about beliefs occurring over some period. For example, to say that Lois believes today that Superman can fly, we write

$$T(\text{Believes}(\text{Lois}, \text{"Flies}(\text{Superman})"), \text{Today}) .$$

If the object of belief is a proposition that can change over time, then it too can be described using the *T* operator within the string. For example, Lois might believe today that that Superman could fly yesterday:

$$T(\text{Believes}(\text{Lois}, \text{"T(Flies(Superman), Yesterday)"}, \text{Today}) .$$

Given a way to describe beliefs over time, we can use the machinery of event calculus to make plans involving beliefs. Actions can have **knowledge preconditions** and **knowledge effects**. For example, the action of dialing a person's number has the precondition of knowing the number, and the action of looking up the number has the effect of knowing the number. We can describe the latter action using the machinery of event calculus:

$$\begin{aligned} &\text{Initiates}(\text{Lookup}(a, \text{"PhoneNumber}(\underline{b})"), \\ &\quad \text{KnowsWhat}(a, \text{"PhoneNumber}(\underline{b})", \text{DigitStrings}), t) . \end{aligned}$$

Plans to gather and use information are often represented using a shorthand notation called **runtime variables**, which is closely related to the unquoted-variable convention described earlier. For example, the plan to look up Bob's number and then dial it can be written as

$$[\text{Lookup}(\text{Agent}, \text{"PhoneNumber}(\text{Bob})", \underline{n}), \text{Dial}(\underline{n})] .$$

Here, \underline{n} is a runtime variable whose value will be bound by the *Lookup* action and can then be used by the *Dial* action. Plans of this kind occur frequently in partially observable domains. We will see examples in the next section and in Chapter 12.

10.5 THE INTERNET SHOPPING WORLD

In this section we will encode some knowledge related to shopping on the Internet. We will create a shopping research agent that helps a buyer find product offers on the Internet. The shopping agent is given a product description by the buyer and has the task of producing a list of Web pages that offer such a product for sale. In some cases the buyer's product description will be precise, as in *Coolpix 995 digital camera*, and the task is then to find the store(s) with the best offer. In other cases the description will be only partially specified, as in *digital camera for under \$300*, and the agent will have to compare different products.

The shopping agent's environment is the entire World Wide Web—not a toy simulated environment, but the same complex, constantly evolving environment that is used by millions of people every day. The agent's percepts are Web pages, but whereas a human Web user would see pages displayed as an array of pixels on a screen, the shopping agent will perceive

KNOWLEDGE
PRECONDITIONS
KNOWLEDGE
EFFECTS

RUNTIME VARIABLES

Generic Online Store

Select from our fine line of products:

- Computers
- Cameras
- Books
- Videos
- Music

```
<h1>Generic Online Store</h1>
<i>Select</i> from our fine line of products:
<ul>
<li> <a href="http://gen-store.com/compu">Computers</a>
<li> <a href="http://gen-store.com/camer">Cameras</a>
<li> <a href="http://gen-store.com/books">Books</a>
<li> <a href="http://gen-store.com/video">Videos</a>
<li> <a href="http://gen-store.com/music">Music</a>
</ul>
```

Figure 10.7 A Web page from a generic online store in the form perceived by the human user of a browser (top), and the corresponding HTML string as perceived by the browser or the shopping agent (bottom). In HTML, characters between `<` and `>` are markup directives that specify how the page is displayed. For example, the string `<i>Select</i>` means to switch to italic font, display the word *Select*, and then end the use of italic font. A page identifier such as `http://gen-store.com/books` is called a **uniform resource locator (URL)** or URL. The markup `anchor` means to create a hypertext link to *url* with the **anchor text** *anchor*.

a page as a character string consisting of ordinary words interspersed with formatting commands in the HTML markup language. Figure 10.7 shows a Web page and a corresponding HTML character string. The perception problem for the shopping agent involves extracting useful information from percepts of this kind.

Clearly, perception on Web pages is easier than, say, perception while driving a taxi in Cairo. Nonetheless, there are complications to the Internet perception task. The web page in Figure 10.7 is very simple compared to real shopping sites, which include cookies, Java, Javascript, Flash, robot exclusion protocols, malformed HTML, sound files, movies, and text that appears only as part of a JPEG image. An agent that can deal with *all* of the Internet is almost as complex as a robot that can move in the real world. We will concentrate on a simple agent that ignores most of these complications.

The agent's first task is to find relevant product offers (we'll see later how to choose the best of the relevant offers). Let *query* be the product description that the user types in (e.g., "laptops"); then a page is a relevant offer for *query* if the page is relevant and the page is indeed an offer. We will also keep track of the URL associated with the page:

$$\text{RelevantOffer}(\text{page}, \text{url}, \text{query}) \Leftrightarrow \text{Relevant}(\text{page}, \text{url}, \text{query}) \wedge \text{Offer}(\text{page}) .$$

A page with a review of the latest high-end laptop would be relevant, but if it doesn't provide a way to buy, it isn't an offer. For now, we can say a page is an offer if it contains the word “buy” or “price” within an HTML link or form on the page. In other words, if the page contains a string of the form “<a ...buy ...</a” then it is an offer; it could also say “price” instead of “buy” or use “form” instead of “a”. We can write axioms for this:

$$\begin{aligned}
 \text{Offer}(\text{page}) &\Leftrightarrow (\text{InTag}(\text{"a"}, \text{str}, \text{page}) \vee \text{InTag}(\text{"form"}, \text{str}, \text{page})) \\
 &\quad \wedge (\text{In}(\text{"buy"}, \text{str}) \vee \text{In}(\text{"price"}, \text{str})) . \\
 \text{InTag}(\text{tag}, \text{str}, \text{page}) &\Leftrightarrow \text{In}(\text{"<" + tag + str + "< /" + tag}, \text{page}) . \\
 \text{In}(\text{sub}, \text{str}) &\Leftrightarrow \exists i \text{ str}[i : i + \text{Length}(\text{sub})] = \text{sub} .
 \end{aligned}$$

Now we need to find relevant pages. The strategy is to start at the home page of an online store and consider all pages that can be reached by following relevant links.⁷ The agent will have knowledge of a number of stores, for example:

$$\begin{aligned}
 \text{Amazon} &\in \text{OnlineStores} \wedge \text{Homepage}(\text{Amazon}, \text{"amazon.com"}) . \\
 \text{Ebay} &\in \text{OnlineStores} \wedge \text{Homepage}(\text{Ebay}, \text{"ebay.com"}) . \\
 \text{GenStore} &\in \text{OnlineStores} \wedge \text{Homepage}(\text{GenStore}, \text{"gen-store.com"}) .
 \end{aligned}$$

These stores classify their goods into product categories, and provide links to the major categories from their home page. Minor categories can be reached by following a chain of relevant links, and eventually we will reach offers. In other words, a page is relevant to the query if it can be reached by a chain of relevant category links from a store's home page, and then following one more link to the product offer:

$$\begin{aligned}
 \text{Relevant}(\text{page}, \text{url}, \text{query}) &\Leftrightarrow \\
 &\exists \text{store, home} \text{ store} \in \text{OnlineStores} \wedge \text{Homepage}(\text{store}, \text{home}) \\
 &\wedge \exists \text{url}_2 \text{ RelevantChain}(\text{home}, \text{url}_2, \text{query}) \wedge \text{Link}(\text{url}_2, \text{url}) \\
 &\wedge \text{page} = \text{GetPage}(\text{url}) .
 \end{aligned}$$

Here the predicate $\text{Link}(\text{from}, \text{to})$ means that there is a hyperlink from the *from* URL to the *to* URL. (See Exercise 10.13.) To define what counts as a *RelevantChain*, we need to follow not just any old hyperlinks, but only those links whose associated anchor text indicates that the link is relevant to the product query. For this, we will use $\text{LinkText}(\text{from}, \text{to}, \text{text})$ to mean that there is a link between *from* and *to* with *text* as the anchor text. A chain of links between two URLs, *start* and *end*, is relevant to a description *d* if the anchor text of each link is a relevant category name for *d*. The existence of the chain itself is determined by a recursive definition, with the empty chain ($\text{start} = \text{end}$) as the base case:

$$\begin{aligned}
 \text{RelevantChain}(\text{start}, \text{end}, \text{query}) &\Leftrightarrow (\text{start} = \text{end}) \\
 &\vee (\exists u, \text{text} \text{ LinkText}(\text{start}, u, \text{text}) \wedge \text{RelevantCategoryName}(\text{query}, \text{text}) \\
 &\quad \wedge \text{RelevantChain}(u, \text{end}, \text{query})) .
 \end{aligned}$$

Now we must define what it means for *text* to be a *RelevantCategoryName* for *query*. First, we need to relate strings to the categories they name. This is done using the predicate $\text{Name}(s, c)$, which says that string *s* is a name for category *c*—for example, we might assert that $\text{Name}(\text{"laptops"}, \text{LaptopComputers})$. Some more examples of the *Name* predicate

⁷ An alternative to the link-following strategy is to use an Internet search engine; the technology behind Internet search, information retrieval, will be covered in Section 23.2.

<i>Books</i> \subset <i>Products</i>	<i>Name</i> ("books", <i>Books</i>)
<i>MusicRecordings</i> \subset <i>Products</i>	<i>Name</i> ("music", <i>MusicRecordings</i>)
<i>MusicCDs</i> \subset <i>MusicRecordings</i>	<i>Name</i> ("CDs", <i>MusicCDs</i>)
<i>MusicTapes</i> \subset <i>MusicRecordings</i>	<i>Name</i> ("tapes", <i>MusicTapes</i>)
<i>Electronics</i> \subset <i>Products</i>	<i>Name</i> ("electronics", <i>Electronics</i>)
<i>DigitalCameras</i> \subset <i>Electronics</i>	<i>Name</i> ("digital cameras", <i>DigitalCameras</i>)
<i>StereoEquipment</i> \subset <i>Electronics</i>	<i>Name</i> ("stereos", <i>StereoEquipment</i>)
<i>Computers</i> \subset <i>Electronics</i>	<i>Name</i> ("computers", <i>Computers</i>)
<i>LaptopComputers</i> \subset <i>Computers</i>	<i>Name</i> ("laptops", <i>LaptopComputers</i>)
<i>DesktopComputers</i> \subset <i>Computers</i>	<i>Name</i> ("desktops", <i>DesktopComputers</i>)
...	...
(a)	(b)

Figure 10.8 (a) Taxonomy of product categories. (b) Referring words for those categories.

appear in Figure 10.8(b). Next, we define relevance. Suppose that *query* is “laptops.” Then *RelevantCategoryName(query, text)* is true when one of the following holds:

- The *text* and *query* name the same category—e.g., “laptop computers” and “laptops.”
- The *text* names a supercategory such as “computers.”
- The *text* names a subcategory such as “ultralight notebooks.”

The logical definition of *RelevantCategoryName* is as follows:

RelevantCategoryName(*query*, *text*) \Leftrightarrow
 $\exists c_1, c_2 \text{ } Name(query, c_1) \wedge Name(text, c_2) \wedge (c_1 \subseteq c_2 \vee c_2 \subseteq c_1) .$

(10.1)

Otherwise, the anchor text is irrelevant because it names a category outside this line, such as “mainframe computers” or “lawn & garden.”

To follow relevant links, then, it is essential to have a rich hierarchy of product categories. The top part of this hierarchy might look like Figure 10.8(a). It will not be feasible to list *all* possible shopping categories, because a buyer could always come up with some new desire and manufacturers will always come out with new products to satisfy them (electric kneecap warmers?). Nonetheless, an ontology of about a thousand categories will serve as a very useful tool for most buyers.

In addition to the product hierarchy itself, we also need to have a rich vocabulary of names for categories. Life would be much easier if there were a one-to-one correspondence between categories and the character strings that name them. We have already seen the problem of **synonymy**—two names for the same category, such as “laptop computers” and “laptops.” There is also the problem of **ambiguity**—one name for two or more different categories. For example, if we add the sentence

Name("CDs", *CertificatesOfDeposit*)

to the knowledge base in Figure 10.8(b), then “CDs” will name two different categories.

Synonymy and ambiguity can cause a significant increase in the number of paths that the agent has to follow, and can sometimes make it difficult to determine whether a given

page is indeed relevant. A much more serious problem is that there is a very broad range of descriptions that a user can type, or category names that a store can use. For example, the link might say “laptop” when the knowledge base has only “laptops;” or the user might ask for “a computer I can fit on the tray table of an economy-class seat in a Boeing 737.” It is impossible to enumerate in advance all the ways a category can be named, so the agent will have to be able to do additional reasoning in some cases to determine if the *Name* relation holds. In the worst case, this requires full natural language understanding, a topic that we will defer to Chapter 22. In practice, a few simple rules—such as allowing “laptop” to match a category named “laptops”—go a long way. Exercise 10.15 asks you to develop a set of such rules after doing some research into online stores.

Given the logical definitions from the preceding paragraphs and suitable knowledge bases of product categories and naming conventions, are we ready to apply an inference algorithm to obtain a set of relevant offers for our query? Not quite! The missing element is the *GetPage(url)* function, which refers to the HTML page at a given URL. The agent doesn’t have the page contents of every URL in its knowledge base; nor does it have explicit rules for deducing what those contents might be. Instead, we can arrange for the right HTTP procedure to be executed whenever a subgoal involves the *GetPage* function. In this way, it appears to the inference engine as if the entire Web is inside the knowledge base. This is an example of a general technique called **procedural attachment**, whereby particular predicates and functions can be handled by special-purpose methods.

Comparing offers

Let us assume that the reasoning processes of the preceding section have produced a set of offer pages for our “laptops” query. To compare those offers, the agent must extract the relevant information—price, speed, disk size, weight, and so on—from the offer pages. This can be a difficult task with real web pages, for all the reasons mentioned previously. A common way of dealing with this problem is to use programs called **wrappers** to extract information from a page. The technology of information extraction is discussed in Section 23.3. For now we assume that wrappers exist, and when given a page and a knowledge base, they add assertions to the knowledge base. Typically a hierarchy of wrappers would be applied to a page: a very general one to extract dates and prices, a more specific one to extract attributes for computer-related products, and if necessary a site-specific one that knows the format of a particular store. Given a page on the gen-store.com site with the text

YVM ThinkBook 970. Our price: \$1449.00

followed by various technical specifications, we would like a wrapper to extract information such as the following:

$$\begin{aligned} \exists lc, offer \quad & lc \in LaptopComputers \wedge offer \in ProductOffers \wedge \\ & ScreenSize(lc, Inches(14)) \wedge ScreenType(lc, ColorLCD) \wedge \\ & MemorySize(lc, Megabytes(512)) \wedge CPUSpeed(lc, GHz(2.4)) \wedge \\ & OfferedProduct(offer, lc) \wedge Store(offer, GenStore) \wedge \\ & URL(offer, "genstore.com/comps/34356.html") \wedge \\ & Price(offer, $(449)) \wedge Date(offer, Today) . \end{aligned}$$

This example illustrates several issues that arise when we take seriously the task of knowledge engineering for commercial transactions. For example, notice that the price is an attribute of the *offer*, not the product itself. This is important because the offer at a given store may change from day to day even for the same individual laptop; for some categories—such as houses and paintings—the same individual object may even be offered simultaneously by different intermediaries at different prices. There are still more complications that we have not handled, such as the possibility that the price depends on method of payment and on the buyer's qualifications for certain discounts. All in all, there is much interesting work to do.

The final task is to compare the offers that have been extracted. For example, consider these three offers:

A : 2.4 GHz CPU, 512MB RAM, 80 GB disk, DVD, CDRW, \$1695 .

B : 2.0 GHz CPU, 1GB RAM, 120 GB disk, DVD, CDRW, \$1800 .

C : 2.2 GHz CPU, 512MB RAM, 80 GB disk, DVD, CDRW, \$1800 .

C is **dominated** by *A*; that is, *A* is cheaper and faster, and they are otherwise the same. In general, *X* dominates *Y* if *X* has a better value on at least one attribute, and is not worse on any attribute. But neither *A* nor *B* dominates the other. To decide which is better we need to know how the buyer weighs CPU speed and price against memory and disk space. The general topic of preferences among multiple attributes is addressed in Section 16.4; for now, our shopping agent will simply return a list of all undominated offers that meet the buyer's description. In this example, both *A* and *B* are undominated. Notice that this outcome relies on the assumption that everyone prefers cheaper prices, faster processors, and more storage. Some attributes, such as screen size on a notebook, depend on the user's particular preference (portability versus visibility); for these, the shopping agent will just have to ask the user.

The shopping agent we have described here is a simple one; many refinements are possible. Still, it has enough capability that with the right domain-specific knowledge it can actually be of use to a shopper. Because of its declarative construction, it extends easily to more complex applications. The main point of this section is to show that some knowledge representation—in particular, the product hierarchy—is necessary for an agent like this, and that once we have some knowledge in this form, it is not too hard to do the rest as a knowledge-based agent.

10.6 REASONING SYSTEMS FOR CATEGORIES

We have seen that categories are the primary building blocks of any large-scale knowledge representation scheme. This section describes systems specially designed for organizing and reasoning with categories. There are two closely related families of systems: **semantic networks** provide graphical aids for visualizing a knowledge base and efficient algorithms for inferring properties of an object on the basis of its category membership; and **description logics** provide a formal language for constructing and combining category definitions and efficient algorithms for deciding subset and superset relationships between categories.

Semantic networks

In 1909, Charles Peirce proposed a graphical notation of nodes and arcs called **existential graphs** that he called “the logic of the future.” Thus began a long-running debate between advocates of “logic” and advocates of “semantic networks.” Unfortunately, the debate obscured the fact that semantics networks—at least those with well-defined semantics—are a form of logic. The notation that semantic networks provide for certain kinds of sentences is often more convenient, but if we strip away the “human interface” issues, the underlying concepts—objects, relations, quantification, and so on—are the same.

There are many variants of semantic networks, but all are capable of representing individual objects, categories of objects, and relations among objects. A typical graphical notation displays object or category names in ovals or boxes, and connects them with labeled arcs. For example, Figure 10.9 has a *MemberOf* link between *Mary* and *FemalePersons*, corresponding to the logical assertion $Mary \in FemalePersons$; similarly, the *SisterOf* link between *Mary* and *John* corresponds to the assertion $SisterOf(Mary, John)$. We can connect categories using *SubsetOf* links, and so on. It is such fun drawing bubbles and arrows that one can get carried away. For example, we know that persons have female persons as mothers, so can we draw a *HasMother* link from *Persons* to *FemalePersons*? The answer is no, because *HasMother* is a relation between a person and his or her mother, and categories do not have mothers.⁸ For this reason, we have used a special notation—the double-boxed link—in Figure 10.9. This link asserts that

$$\forall x \ x \in Persons \Rightarrow [\forall y \ HasMother(x, y) \Rightarrow y \in FemalePersons] .$$

We might also want to assert that persons have two legs—that is,

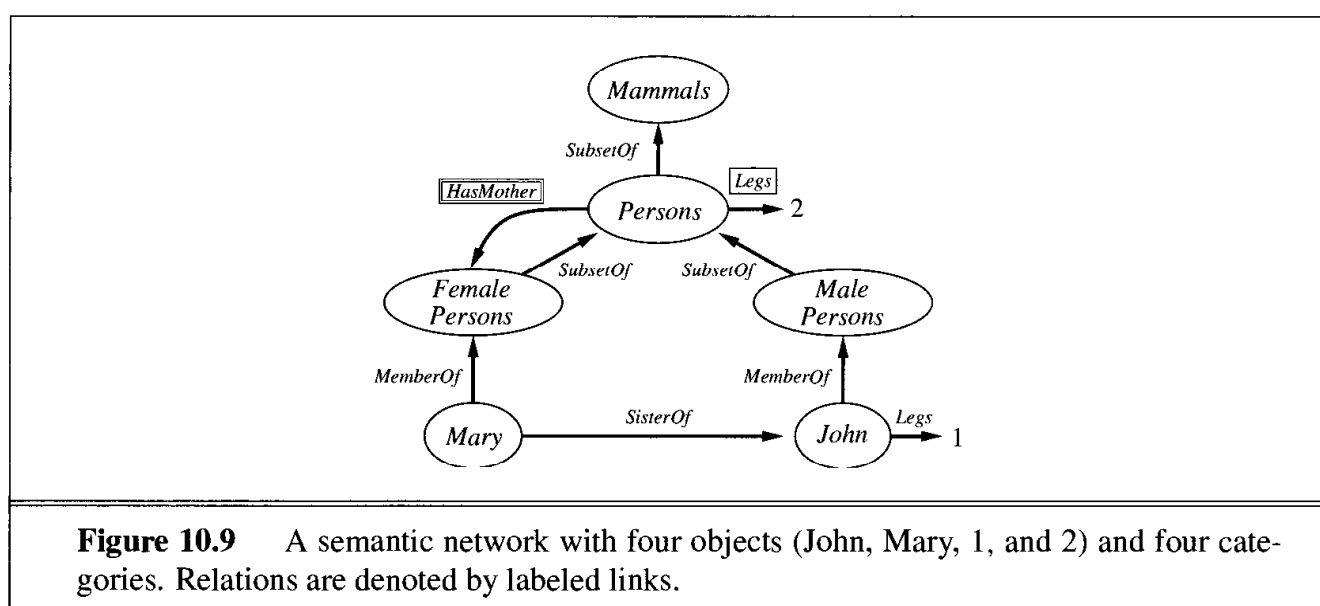
$$\forall x \ x \in Persons \Rightarrow Legs(x, 2) .$$

As before, we need to be careful not to assert that a category has legs; the single-boxed link in Figure 10.9 is used to assert properties of every member of a category.

The semantic network notation makes it very convenient to perform **inheritance** reasoning of the kind introduced in Section 10.2. For example, by virtue of being a person, Mary inherits the property of having two legs. Thus, to find out how many legs Mary has, the inheritance algorithm follows the *MemberOf* link from *Mary* to the category she belongs to, and then follows *SubsetOf* links up the hierarchy until it finds a category for which there is a boxed *Legs* link—in this case, the *Persons* category. The simplicity and efficiency of this inference mechanism, compared with logical theorem proving, has been one of the main attractions of semantic networks.

Inheritance becomes complicated when an object can belong to more than one category or when a category can be a subset of more than one other category; this is called **multiple inheritance**. In such cases, the inheritance algorithm might find two or more conflicting values

⁸ Several early systems failed to distinguish between properties of members of a category and properties of the category as a whole. This can lead directly to inconsistencies, as pointed out by Drew McDermott (1976) in his article “Artificial Intelligence Meets Natural Stupidity.” Another common problem was the use of *IsA* links for both subset and membership relations, in correspondence with English usage: “a cat is a mammal” and “Fifi is a cat.” See Exercise 10.25 for more on these issues.



answering the query. For this reason, multiple inheritance is banned in some **object-oriented programming** (OOP) languages, such as Java, that use inheritance in a class hierarchy. It is usually allowed in semantic networks, but we defer discussion of that until Section 10.7.

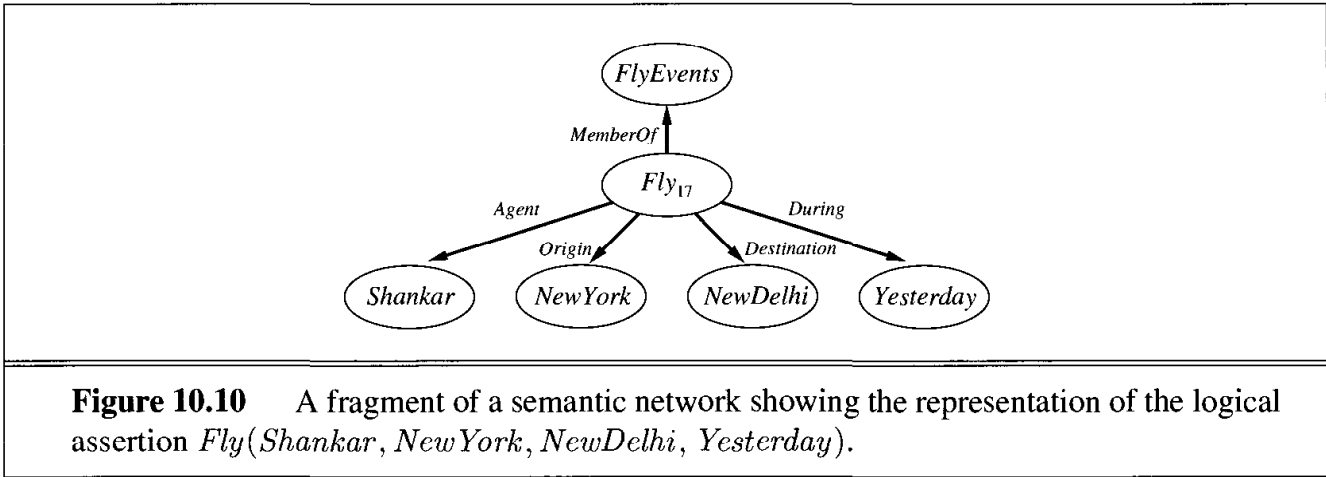
Another common form of inference is the use of **inverse links**. For example, *HasSister* is the inverse of *SisterOf*, which means that

$$\forall p, s \text{ } HasSister(p, s) \Leftrightarrow SisterOf(s, p) .$$

This sentence can be asserted in a semantic network if links are **reified**—that is, made into objects in their own right. For example, we could have a *SisterOf* object, connected by an *Inverse* link to *HasSister*. Given a query asking who is a *SisterOf* John, the inference algorithm can discover that *HasSister* is the inverse of *SisterOf* and can therefore answer the query by following the *HasSister* link from *John* to *Mary*. Without the inverse information, it might be necessary to check every female person to see whether that person has a *SisterOf* link to John. This is because semantic networks provide direct indexing only for objects, categories, and the links emanating from them; in the vocabulary of first-order logic, it is as if the knowledge base were indexed only on the first argument of each predicate.

The reader might have noticed an obvious drawback of semantic network notation, compared to first-order logic: the fact that links between bubbles represent only *binary* relations. For example, the sentence *Fly(Shankar, NewYork, NewDelhi, Yesterday)* cannot be asserted directly in a semantic network. Nonetheless, we *can* obtain the effect of *n*-ary assertions by reifying the proposition itself as an event (see Section 10.3) belonging to an appropriate event category. Figure 10.10 shows the semantic network structure for this particular event. Notice that the restriction to binary relations forces the creation of a rich ontology of reified concepts; indeed, much of the ontology developed in this chapter originated in semantic network systems.

Reification of propositions makes it possible to represent every ground, function-free atomic sentence of first-order logic in the semantic network notation. Certain kinds of universally quantified sentences can be asserted using inverse links and the singly boxed and doubly boxed arrows applied to categories, but that still leaves us a long way short of full first-order



logic. Negation, disjunction, nested function symbols, and existential quantification are all missing. Now it is *possible* to extend the notation to make it equivalent to first-order logic—as in Peirce’s existential graphs or Hendrix’s (1975) partitioned semantic networks—but doing so negates one of the main advantages of semantic networks, which is the simplicity and transparency of the inference processes. Designers can build a large network and still have a good idea about what queries will be efficient, because (a) it is easy to visualize the steps that the inference procedure will go through and (b) in some cases the query language is so simple that difficult queries cannot be posed. In cases where the expressive power proves to be too limiting, many semantic network systems provide for **procedural attachment** to fill in the gaps. Procedural attachment is a technique whereby a query about (or sometimes an assertion of) a certain relation results in a call to a special procedure designed for that relation rather than a general inference algorithm.

One of the most important aspects of semantic networks is their ability to represent **default values** for categories. Examining Figure 10.9 carefully, one notices that John has one leg, despite the fact that he is a person and all persons have two legs. In a strictly logical KB, this would be a contradiction, but in a semantic network, the assertion that all persons have two legs has only default status; that is, a person is assumed to have two legs unless this is contradicted by more specific information. The default semantics is enforced naturally by the inheritance algorithm, because it follows links upwards from the object itself (John in this case) and stops as soon as it finds a value. We say that the default is **overridden** by the more specific value. Notice that we could also override the default number of legs by creating a category of *OneLeggedPersons*, a subset of *Persons* of which *John* is a member.

We can retain a strictly logical semantics for the network if we say that the *Legs* assertion for *Persons* includes an exception for John:

$$\forall x \ x \in Persons \wedge x \neq John \Rightarrow Legs(x, 2) .$$

For a *fixed* network, this is semantically adequate, but will be much less concise than the network notation itself if there are lots of exceptions. For a network that will be updated with more assertions, however, such an approach fails—we really want to say that any persons as yet unknown with one leg are exceptions too. Section 10.7 goes into more depth on this issue and on default reasoning in general.

DEFAULT VALUES

OVERRIDING

Description logics

DESCRIPTION
LOGICS

The syntax of first-order logic is designed to make it easy to say things about objects. **Description logics** are notations that are designed to make it easier to describe definitions and properties of categories. Description logic systems evolved from semantic networks in response to pressure to formalize what the networks mean while retaining the emphasis on taxonomic structure as an organizing principle.

SUBSUMPTION
CLASSIFICATION

The principal inference tasks for description logics are **subsumption**—checking if one category is a subset of another by comparing their definitions—and **classification**—checking whether an object belongs to a category. Some systems also include **consistency** of a category definition—whether the membership criteria are logically satisfiable.

The CLASSIC language (Borgida *et al.*, 1989) is a typical description logic. The syntax of CLASSIC descriptions is shown in Figure 10.11.⁹ For example, to say that bachelors are unmarried adult males we would write

$$\text{Bachelor} = \text{And}(\text{Unmarried}, \text{Adult}, \text{Male}) .$$

The equivalent in first-order logic would be

$$\text{Bachelor}(x) \Leftrightarrow \text{Unmarried}(x) \wedge \text{Adult}(x) \wedge \text{Male}(x) .$$

Notice that the description logic effectively allows direct logical operations on predicates, rather than having to first create sentences to be joined by connectives. Any description in CLASSIC can be written in first-order logic, but some descriptions are more straightforward in CLASSIC. For example, to describe the set of men with at least three sons who are all unemployed and married to doctors and at most two daughters who are all professors in physics or math departments, we would use

$$\begin{aligned} &\text{And}(\text{Man}, \text{AtLeast}(3, \text{Son}), \text{AtMost}(2, \text{Daughter}), \\ &\quad \text{All}(\text{Son}, \text{And}(\text{Unemployed}, \text{Married}, \text{All}(\text{Spouse}, \text{Doctor}))), \\ &\quad \text{All}(\text{Daughter}, \text{And}(\text{Professor}, \text{Fills}(\text{Department}, \text{Physics}, \text{Math})))) . \end{aligned}$$

We leave it as an exercise to translate this into first-order logic.

Perhaps the most important aspect of description logics is their emphasis on tractability of inference. A problem instance is solved by describing it and then asking if it is subsumed by one of several possible solution categories. In standard first-order logic systems, predicting the solution time is often impossible. It is frequently left to the user to engineer the representation to detour around sets of sentences that seem to be causing the system to take several weeks to solve a problem. The thrust in description logics, on the other hand, is to ensure that subsumption-testing can be solved in time polynomial in the size of the descriptions.¹⁰

This sounds wonderful in principle, until one realizes that it can only have one of two consequences: either hard problems cannot be stated at all, or they require exponentially large descriptions! However, the tractability results do shed light on what sorts of constructs cause problems and thus help the user to understand how different representations behave.

⁹ Notice that the language does *not* allow one to simply state that one concept, or category, is a subset of another. This is a deliberate policy: subsumption between categories must be derivable from some aspects of the descriptions of the categories. If not, then something is missing from the descriptions.

¹⁰ CLASSIC provides efficient subsumption testing in practice, but the worst-case runtime is exponential.

```

Concept → Thing | ConceptName
           | And(Concept, ...)
           | All(RoleName, Concept)
           | AtLeast(Integer, RoleName)
           | AtMost(Integer, RoleName)
           | Fills(RoleName, IndividualName, ...)
           | SameAs(Path, Path)
           | OneOf(IndividualName, ...)
Path → [RoleName, ...]

```

Figure 10.11 The syntax of descriptions in a subset of the CLASSIC language.

For example, description logics usually lack *negation* and *disjunction*. Each forces first-order logical systems to go through a potentially exponential case analysis in order to ensure completeness. For the same reason, they are excluded from Prolog. CLASSIC allows only a limited form of disjunction in the *Fills* and *OneOf* constructs, which permit disjunction over explicitly enumerated individuals but not over descriptions. With disjunctive descriptions, nested definitions can lead easily to an exponential number of alternative routes by which one category can subsume another.

10.7 REASONING WITH DEFAULT INFORMATION

In the preceding section, we saw a simple example of an assertion with default status: people have two legs. This default can be overridden by more specific information, such as that Long John Silver has one leg. We saw that the inheritance mechanism in semantic networks implements the overriding of defaults in a simple and natural way. In this section, we study defaults more generally, with a view toward understanding the *semantics* of defaults rather than just providing a procedural mechanism.

Open and closed worlds

Suppose you were looking at a bulletin board in a university computer science department and saw a notice saying, “The following courses will be offered: CS 101, CS 102, CS 106, EE 101.” Now, how many courses will be offered? If you answered “Four,” you would be in agreement with a typical database system. Given a relational database with the equivalent of the four assertions

$$\text{Course}(\text{CS}, 101), \text{Course}(\text{CS}, 102), \text{Course}(\text{CS}, 106), \text{Course}(\text{EE}, 101), \quad (10.2)$$

the SQL query `count * from Course` returns 4. On the other hand, a first-order logical system would answer “Somewhere between one and infinity,” not “four.” The reason is that

the *Course* assertions do not deny the possibility that other unmentioned courses are also offered, nor do they say that the courses mentioned are different from each other.

This example shows that database systems and human communication conventions differ from first-order logic in at least two ways. First, databases (and people) assume that the information provided is *complete*, so that ground atomic sentences not asserted to be true are assumed to be false. This is called the **closed-world assumption**, or CWA. Second, we usually assume that distinct names refer to distinct objects. This is the **unique names assumption**, or UNA, which we introduced first in the context of action names in Section 10.3.

First-order logic does not assume these conventions, and thus needs to be more explicit. To say that *only* the four distinct courses are offered, we would write:

$$\begin{aligned} \text{Course}(d, n) \quad \Leftrightarrow \quad & [d, n] = [CS, 101] \vee [d, n] = [CS, 102] \\ & \vee [d, n] = [CS, 106] \vee [d, n] = [EE, 101] . \end{aligned} \quad (10.3)$$

Equation 10.3 is called the **completion**¹¹ of 10.2. In general, the completion will contain a definition—an if-and-only-if sentence—for each predicate, and each definition will contain a disjunct for each definite clause having that predicate in its head.¹² In general, the completion is constructed as follows:

1. Gather up all the clauses with the same predicate name (P) and the same arity (n).
2. Translate each clause to **Clark Normal Form**: replace

$$P(t_1, \dots, t_n) \leftarrow \text{Body} ,$$

where t_i are terms, with

$$P(v_1, \dots, v_n) \leftarrow \exists w_1 \dots w_m \ [v_1, \dots, v_n] = [t_1, \dots, t_n] \wedge \text{Body} ,$$

where v_i are newly invented variables and w_i are the variables that appear in the original clause. Use the same set of v_i for every clause. This gives us a set of clauses

$$\begin{aligned} P(v_1, \dots, v_n) &\leftarrow B_1 \\ &\vdots \\ P(v_1, \dots, v_n) &\leftarrow B_k . \end{aligned}$$

3. Combine these together into one big disjunctive clause:

$$P(v_1, \dots, v_n) \leftarrow B_1 \vee \dots \vee B_k .$$

4. Form the completion by replacing the \leftarrow with an equivalence:

$$P(v_1, \dots, v_n) \Leftrightarrow B_1 \vee \dots \vee B_k .$$

Figure 10.12 shows an example of the Clark completion for a knowledge base with both ground facts and rules. To add in the unique names assumption, we simply construct the Clark completion for the equality relation, where the only known facts are that $CS = CS$, $101 = 101$, and so on. This is left as an exercise.

The closed-world assumption allows us to find a **minimal model** of a relation. That is, we can find the model of the relation *Course* with the fewest elements. In Equation (10.2)

¹¹ Sometimes called “Clark Completion” after the inventor, Keith Clark.

¹² Notice that this is also the form of the successor-state axioms given in Section 10.3.

CLOSED-WORLD
ASSUMPTION

COMPLETION

CLARK NORMAL
FORM

Horn Clauses	Clark Completion
$Course(CS, 101)$	$Course(d, n) \Leftrightarrow [d, n] = [CS, 101]$
$Course(CS, 102)$	$\vee [d, n] = [CS, 102]$
$Course(CS, 106)$	$\vee [d, n] = [CS, 106]$
$Course(EE, 101)$	$\vee [d, n] = [EE, 101]$
$Course(EE, i) \leftarrow Integer(i)$	$\vee \exists i [d, n] = [EE, i] \wedge Integer(i)$
$\wedge 101 \leq i \wedge i \leq 130$	$\wedge 101 \leq i \wedge i \leq 130$
$Course(CS, m + 100) \Leftarrow$	$\vee \exists m [d, n] = [CS, m + 100]$
$Course(CS, m) \wedge 100 \leq m$	$\wedge Course(CS, m) \wedge 100 \leq m$
$\wedge m < 200$	$\wedge m < 200$

Figure 10.12 The Clark Completion of a set of Horn clauses. The original Horn program (left) lists four courses explicitly and also asserts that there is a math class for every integer from 101 to 130, and that for every CS class in the 100 (undergraduate) series, there is a corresponding class in the 200 (graduate) series. The Clark completion (right) says that there are no other classes. With the completion and the unique names assumption (and the obvious definition of the *Integer* predicate), we get the desired conclusion that there are exactly 36 courses: 30 math courses and 6 CS courses.

the minimal model of *Course* has four elements; any less and we'd have a contradiction. For Horn knowledge bases, there is always a *unique* minimal model. Notice that, with the unique names assumption, this applies to the equality relation too: each term is equal only to itself. Paradoxically, this means that minimal models are maximal in the sense of having as many objects as possible.

It is possible to take a Horn program, generate the Clark completion, and hand that to a theorem prover to do inference. But it is usually more efficient to use a special-purpose inference mechanism such as Prolog, which has the closed world and unique names assumptions built into the inference mechanism.

Those who make the closed-world assumption must be careful about what kind of reasoning they will be doing. For example, in a census database it would be reasonable to make the CWA when reasoning about the current population of cities, but it would be wrong to conclude that no baby will ever be born in the future just because the database contains no entries with future birthdates. The CWA makes the database **complete**, in the sense that every atomic query is answered either positively or negatively; when we are genuinely ignorant of facts (such as future births) we cannot use the CWA. A more sophisticated knowledge representation system might allow the user to specify rules for when to apply the CWA.

Negation as failure and stable model semantics

We saw in Chapters 7 and 9 that Horn-form knowledge bases have desirable computational properties. In many applications, however, the requirement that every literal in the body of a clause be positive is rather inconvenient. We would like to say "You can go outside if it's not raining," without having to concoct predicates such as *NotRaining*. In this section, we explore the addition of a form of explicit negation to Horn clauses using the idea using of

NEGATION AS
FAILURE

negation as failure. The idea is that a negative literal, *not P*, can be “proved” true just in case the proof of *P* fails. This is a form of default reasoning closely related to the closed world assumption: we assume something is false if it cannot be proved true. We use “*not*” to distinguish negation as failure from the logical “ \neg ” operator.

Prolog allows the *not* operator in the body of a clause. For example, consider the following Prolog program:

$$\begin{aligned} IDEdrive &\leftarrow Drive \wedge not\ SCSIdrive . \\ SCSIdrive &\leftarrow Drive \wedge not\ IDEdrive . \\ SCSIcontroller &\leftarrow SCSIdrive . \\ Drive &. \end{aligned} \tag{10.4}$$

The first rule says that if we have a hard drive on a computer and it is not SCSI, then it must be IDE. The second says if it is not IDE it must be SCSI. The third says that having a SCSI drive implies having a SCSI controller, and the fourth says that we do indeed have a drive. This program has *two* minimal models:

$$\begin{aligned} M_1 &= \{Drive, IDEdrive\} , \\ M_2 &= \{Drive, SCSIdrive, SCSIcontroller\} . \end{aligned}$$

Minimal models do not capture the intended semantics of programs with negation as failure. Consider the program

$$P \leftarrow not\ Q. \tag{10.5}$$

This has two minimal models, $\{P\}$ and $\{Q\}$. From an FOL point of view this makes sense, since $P \leftarrow \neg Q$ is equivalent to $P \vee Q$. But from a Prolog point of view it is worrisome: *Q* never appears on the left hand side of an arrow, so how can it be a consequence?

STABLE MODEL

JUSTIFICATION

REDUCT

An alternative is the idea of a **stable model**, which is a minimal model where every atom in the model has a **justification**: a rule where the head is the atom and where every literal in the body is satisfied. Technically, we say that *M* is a stable model of a program *H* if *M* is the unique minimal model of the **reduct** of *H* with respect to *M*. The reduct of a program *H* is defined by first deleting from *H* any rule that has a literal *not A* in the body, where *A* is in the model, and then deleting any negative literals in the remaining rules. Since the reduct of *H* is now a list of Horn clauses, it must have a unique minimal model.

The reduct of $P \leftarrow not\ Q$ with respect to $\{P\}$ is P , which has minimal model $\{P\}$. Therefore $\{P\}$ is a stable model. The reduct with respect to $\{Q\}$ is the empty program, which has minimal model $\{\}$. Therefore $\{Q\}$ is not a stable model because *Q* has no justification in Equation (10.5). As another example, the reduct of 10.4 with respect to M_1 is as follows:

$$\begin{aligned} IDEdrive &\leftarrow Drive . \\ SCSIcontroller &\leftarrow SCSIdrive . \\ Drive &. \end{aligned}$$

ANSWER SET
PROGRAMMING

ANSWER SETS

This has minimal model M_1 , so M_1 is a stable model. **Answer set programming** is a kind of logic programming with negation as failure that works by translating the logic program into ground form and then searching for stable models (also known as **answer sets**) using propositional model checking techniques. Thus answer set programming is a descendant both of Prolog and of the fast propositional satisfiability provers such as WALKSAT. Indeed,

answer set programming has been successfully applied to problems in planning just as the propositional satisfiability provers have. The advantage of answer set planning over other planners is the degree of flexibility: the planning operators and constraints can be expressed as logic programs and are not bound to the restricted format of a particular planning formalism. The disadvantage of answer set planning is the same as for other propositional techniques: if there are very many objects in the universe, then there can be an exponential slow-down.

Circumscription and default logic

We have seen two examples where apparently natural reasoning processes violate the **monotonicity** property of logic that was proved in Chapter 7.¹³ In the first example, a property inherited by all members of a category in a semantic network could be overridden by more specific information for a subcategory. In the second example, negated literals derived from a closed-world assumption could be overridden by the addition of positive literals.

Simple introspection suggests that these failures of monotonicity are widespread in commonsense reasoning. It seems that humans often “jump to conclusions.” For example, when one sees a car parked on the street, one is normally willing to believe that it has four wheels even though only three are visible. (If you feel that the existence of the fourth wheel is dubious, consider also the question as to whether the three visible wheels are real or merely cardboard facsimiles.) Now, probability theory can certainly provide a conclusion that the fourth wheel exists with high probability, yet, for most people, the possibility of the car’s not having four wheels *does not arise unless some new evidence presents itself*. Thus, it seems that the four-wheel conclusion is reached *by default*, in the absence of any reason to doubt it. If new evidence arrives—for example, if one sees the owner carrying a wheel and notices that the car is jacked up—then the conclusion can be retracted. This kind of reasoning is said to exhibit **nonmonotonicity**, because the set of beliefs does not grow monotonically over time as new evidence arrives. **Nonmonotonic logics** have been devised with modified notions of truth and entailment in order to capture such behavior. We will look at two such logics that have been studied extensively: circumscription and default logic.

Circumscription can be seen as a more powerful and precise version of the closed-world assumption. The idea is to specify particular predicates that are assumed to be “as false as possible”—that is, false for every object except those for which they are known to be true. For example, suppose we want to assert the default rule that birds fly. We would introduce a predicate, say $Abnormal_1(x)$, and write

$$Bird(x) \wedge \neg Abnormal_1(x) \Rightarrow Flies(x).$$

If we say that $Abnormal_1$ is to be **circumscribed**, a circumscriptive reasoner is entitled to assume $\neg Abnormal_1(x)$ unless $Abnormal_1(x)$ is known to be true. This allows the conclusion $Flies(Tweety)$ to be drawn from the premise $Bird(Tweety)$, but the conclusion no longer holds if $Abnormal_1(Tweety)$ is asserted.

Circumscription can be viewed as an example of a **model preference** logic. In such logics, a sentence is entailed (with default status) if it is true in all *preferred* models of the KB,

¹³ Recall that monotonicity requires all entailed sentences to remain entailed after new sentences are added to the KB. That is, if $KB \models \alpha$ then $KB \wedge \beta \models \alpha$.

as opposed to the requirement of truth in *all* models in classical logic. For circumscription, one model is preferred to another if it has fewer abnormal objects.¹⁴ Let us see how this idea works in the context of multiple inheritance in semantic networks. The standard example for which multiple inheritance is problematic is called the “Nixon diamond.” It arises from the observation that Richard Nixon was both a Quaker (and hence by default a pacifist) and a Republican (and hence by default not a pacifist). We can write this as follows:

$$\begin{aligned} & Republican(Nixon) \wedge Quaker(Nixon) . \\ & Republican(x) \wedge \neg Abnormal_2(x) \Rightarrow \neg Pacifist(x) . \\ & Quaker(x) \wedge \neg Abnormal_3(x) \Rightarrow Pacifist(x) . \end{aligned}$$

If we circumscribe $Abnormal_2$ and $Abnormal_3$, there are two preferred models: one in which $Abnormal_2(Nixon)$ and $Pacifist(Nixon)$ hold and one in which $Abnormal_3(Nixon)$ and $\neg Pacifist(Nixon)$ hold. Thus, the circumscriptive reasoner remains properly agnostic as to whether Nixon is a pacifist. If we wish, in addition, to assert that religious beliefs take precedence over political beliefs, we can use a formalism called **prioritized circumscription** to give preference to models where $Abnormal_3$ is minimized.

PRIORITIZED
CIRCUMSCRIPTION

Default logic is a formalism in which **default rules** can be written to generate contingent, nonmonotonic conclusions. A default rule looks like this:

DEFAULT LOGIC

DEFAULT RULES

$$Bird(x) : Flies(x) / Flies(x) .$$

This rule means that if $Bird(x)$ is true, and if $Flies(x)$ is consistent with the knowledge base, then $Flies(x)$ may be concluded by default. In general, a default rule has the form

$$P : J_1, \dots, J_n / C$$

where P is called the prerequisite, C is the conclusion, and J_i are the justifications—if any one of them can be proven false, then the conclusion cannot be drawn. Any variable that appears in J_i or C must also appear in P . The Nixon-diamond example can be represented in default logic with one fact and two default rules:

$$\begin{aligned} & Republican(Nixon) \wedge Quaker(Nixon) . \\ & Republican(x) : \neg Pacifist(x) / \neg Pacifist(x) . \\ & Quaker(x) : Pacifist(x) / Pacifist(x) . \end{aligned}$$

EXTENSION

To interpret what the default rules mean, we define the notion of an **extension** of a default theory to be a maximal set of consequences of the theory. That is, an extension S consists of the original known facts and a set of conclusions from the default rules, such that no additional conclusions can be drawn from S and the justifications of every default conclusion in S are consistent with S . As in the case of the preferred models in circumscription, we have two possible extensions for the Nixon diamond: one wherein he is a pacifist and one wherein he is not. Prioritized schemes exist in which some default rules can be given precedence over others, allowing some ambiguities to be resolved.

Since 1980, when nonmonotonic logics were first proposed, a great deal of progress has been made in understanding their mathematical properties. Beginning in the late 1990s,

¹⁴ For the closed-world assumption, one model is preferred to another if it has fewer true atoms—that is, preferred models are **minimal** models. There is a natural connection between the CWA and definite clause KBs, because the fixed point reached by forward chaining on such KBs is the unique minimal model. (See page 219.)

practical systems based on logic programming have shown promise as knowledge representation tools. There are still unresolved questions, however. For example, if “Cars have four wheels” is false, what does it mean to have it in one’s knowledge base? What is a good set of default rules to have? If we cannot decide, for each rule separately, whether it belongs in our knowledge base, then we have a serious problem of nonmodularity. Finally, how can beliefs that have default status be used to make decisions? This is probably the hardest issue for default reasoning. Decisions often involve tradeoffs, and one therefore needs to compare the *strengths* of belief in the outcomes of different actions. In cases where the same kinds of decisions are being made repeatedly, it is possible to interpret default rules as “threshold probability” statements. For example, the default rule “My brakes are always OK” really means “The probability that my brakes are OK, given no other information, is sufficiently high that the optimal decision is for me to drive without checking them.” When the decision context changes—for example, when one is driving a heavily laden truck down a steep mountain road—the default rule suddenly becomes inappropriate, even though there is no new evidence to suggest that the brakes are faulty. These considerations have led some researchers to consider how to embed default reasoning in probability theory.

10.8 TRUTH MAINTENANCE SYSTEMS

BELIEF REVISION

The previous section argued that many of the inferences drawn by a knowledge representation system will have only default status, rather than being absolutely certain. Inevitably, some of these inferred facts will turn out to be wrong and will have to be retracted in the face of new information. This process is called **belief revision**.¹⁵ Suppose that a knowledge base KB contains a sentence P —perhaps a default conclusion recorded by a forward-chaining algorithm, or perhaps just an incorrect assertion—and we want to execute $TELL(KB, \neg P)$. To avoid creating a contradiction, we must first execute $RETRACT(KB, P)$. This sounds easy enough. Problems arise, however, if any *additional* sentences were inferred from P and asserted in the KB. For example, the implication $P \Rightarrow Q$ might have been used to add Q . The obvious “solution”—retracting all sentences inferred from P —fails because such sentences may have other justifications besides P . For example, if R and $R \Rightarrow Q$ are also in the KB, then Q does not have to be removed after all. **Truth maintenance systems**, or TMSs, are designed to handle exactly these kinds of complications.

TRUTH
MAINTENANCE
SYSTEM

One very simple approach to truth maintenance is to keep track of the order in which sentences are told to the knowledge base by numbering them from P_1 to P_n . When the call $RETRACT(KB, P_i)$ is made, the system reverts to the state just before P_i was added, thereby removing both P_i and any inferences that were derived from P_i . The sentences P_{i+1} through P_n can then be added again. This is simple, and it guarantees that the knowledge base will be consistent, but retracting P_i requires retracting and reasserting $n - i$ sentences as well as

¹⁵ Belief revision is often contrasted with **belief update**, which occurs when a knowledge base is revised to reflect a change in the world rather than new information about a fixed world. Belief update combines belief revision with reasoning about time and change; it is also related to the process of **filtering** described in Chapter 15.

undoing and redoing all the inferences drawn from those sentences. For systems to which many facts are being added—such as large commercial databases—this is impractical.

JTMS

JUSTIFICATION

A more efficient approach is the justification-based truth maintenance system, or **JTMS**. In a JTMS, each sentence in the knowledge base is annotated with a **justification** consisting of the set of sentences from which it was inferred. For example, if the knowledge base already contains $P \Rightarrow Q$, then $\text{TELL}(P)$ will cause Q to be added with the justification $\{P, P \Rightarrow Q\}$. In general, a sentence can have any number of justifications. Justifications are used to make retraction efficient. Given the call $\text{RETRACT}(P)$, the JTMS will delete exactly those sentences for which P is a member of every justification. So, if a sentence Q had the single justification $\{P, P \Rightarrow Q\}$ it would be removed, if it had the additional justification $\{P, P \vee R \Rightarrow Q\}$ it would still be removed, but if it also had the justification $\{R, P \vee R \Rightarrow Q\}$, then it would be spared. In this way, the time required for retraction of P depends only on the number of sentences derived from P rather than on the number of other sentences added since P entered the knowledge base.

The JTMS assumes that sentences that are considered once will probably be considered again, so rather than deleting a sentence from the knowledge base entirely when it loses all justifications, we merely mark the sentence as being *out* of the knowledge base. If a subsequent assertion restores one of the justifications, then we mark the sentence as being back *in*. In this way, the JTMS retains all of the inference chains that it uses and need not rederive sentences when a justification becomes valid again.

In addition to handling the retraction of incorrect information, TMSs can be used to speed up the analysis of multiple hypothetical situations. Suppose, for example, that the Romanian Olympic Committee is choosing sites for the swimming, athletics, and equestrian events at the 2048 Games to be held in Romania. For example, let the first hypothesis be $\text{Site}(\text{Swimming}, \text{Pitesti})$, $\text{Site}(\text{Athletics}, \text{Bucharest})$, and $\text{Site}(\text{Equestrian}, \text{Arad})$. A great deal of reasoning must then be done to work out the logistical consequences and hence the desirability of this selection. If we want to consider $\text{Site}(\text{Athletics}, \text{Sibiu})$ instead, the TMS avoids the need to start again from scratch. Instead, we simply retract $\text{Site}(\text{Athletics}, \text{Bucharest})$ and assert $\text{Site}(\text{Athletics}, \text{Sibiu})$ and the TMS takes care of the necessary revisions. Inference chains generated from the choice of Bucharest can be reused with Sibiu, provided that the conclusions are the same.

ATMS

An assumption-based truth maintenance system, or **ATMS**, is designed to make this type of context-switching between hypothetical worlds particularly efficient. In a JTMS, the maintenance of justifications allows you to move quickly from one state to another by making a few retractions and assertions, but at any time only one state is represented. An ATMS represents *all* the states that have ever been considered at the same time. Whereas a JTMS simply labels each sentence as being *in* or *out*, an ATMS keeps track, for each sentence, of which assumptions would cause the sentence to be true. In other words, each sentence has a label that consists of a set of assumption sets. The sentence holds just in those cases where all the assumptions in one of the assumption sets hold.

EXPLANATIONS

Truth maintenance systems also provide a mechanism for generating **explanations**. Technically, an explanation of a sentence P is a set of sentences E such that E entails P . If the sentences in E are already known to be true, then E simply provides a sufficient ba-

ASSUMPTIONS

sis for proving that P must be the case. But explanations can also include **assumptions**—sentences that are not known to be true, but would suffice to prove P if they were true. For example, one might not have enough information to prove that one's car won't start, but a reasonable explanation might include the assumption that the battery is dead. This, combined with knowledge of how cars operate, explains the observed nonbehavior. In most cases, we will prefer an explanation E that is minimal, meaning that there is no proper subset of E that is also an explanation. An ATMS can generate explanations for the “car won't start” problem by making assumptions (such as “gas in car” or “battery dead”) in any order we like, even if some assumptions are contradictory. Then we look at the label for the sentence “car won't start” to read off the sets of assumptions that would justify the sentence.

The exact algorithms used to implement truth maintenance systems are a little complicated, and we do not cover them here. The computational complexity of the truth maintenance problem is at least as great as that of propositional inference—that is, NP-hard. Therefore, you should not expect truth maintenance to be a panacea. When used carefully, however, a TMS can provide a substantial increase in the ability of a logical system to handle complex environments and hypotheses.

10.9 SUMMARY

This has been the most detailed chapter of the book so far. By delving into the details of how one represents a variety of knowledge, we hope we have given the reader a sense of how real knowledge bases are constructed. The major points are as follows:

- Large-scale knowledge representation requires a general-purpose ontology to organize and tie together the various specific domains of knowledge.
- A general-purpose ontology needs to cover a wide variety of knowledge and should be capable, in principle, of handling any domain.
- We presented an **upper ontology** based on categories and the event calculus. We covered structured objects, time and space, change, processes, substances, and beliefs.
- Actions, events, and time can be represented either in situation calculus or in more expressive representations such as event calculus and fluent calculus. Such representations enable an agent to construct plans by logical inference.
- The mental states of agents can be represented by strings that denote beliefs.
- We presented a detailed analysis of the Internet shopping domain, exercising the general ontology and showing how the domain knowledge can be used by a shopping agent.
- Special-purpose representation systems, such as **semantic networks** and **description logics**, have been devised to help in organizing a hierarchy of categories. **Inheritance** is an important form of inference, allowing the properties of objects to be deduced from their membership in categories.
- The **closed-world assumption**, as implemented in logic programs, provides a simple way to avoid having to specify lots of negative information. It is best interpreted as a **default** that can be overridden by additional information.

- **Nonmonotonic logics**, such as **circumscription** and **default logic**, are intended to capture default reasoning in general. **Answer set programming** speeds up nonmonotonic inference, much as WALKSAT speeds up propositional inference.
- **Truth maintenance systems** handle knowledge updates and revisions efficiently.

BIBLIOGRAPHICAL AND HISTORICAL NOTES

There are plausible claims (Briggs, 1985) that formal knowledge representation research began with classical Indian theorizing about the grammar of Shastric Sanskrit, which dates back to the first millennium B.C. In the West, the use of definitions of terms in ancient Greek mathematics can be regarded as the earliest instance. Indeed, the development of technical terminology in any field can be regarded as a form of knowledge representation.

Early discussions of representation in AI tended to focus on “*problem* representation” rather than “*knowledge* representation.” (See, for example, Amarel’s (1968) discussion of the Missionaries and Cannibals problem.) In the 1970s, AI emphasized the development of “expert systems” (also called “knowledge-based systems”) that could, if given the appropriate domain knowledge, match or exceed the performance of human experts on narrowly defined tasks. For example, the first expert system, DENDRAL (Feigenbaum *et al.*, 1971; Lindsay *et al.*, 1980), interpreted the output of a mass spectrometer (a type of instrument used to analyze the structure of organic chemical compounds) as accurately as expert chemists. Although the success of DENDRAL was instrumental in convincing the AI research community of the importance of knowledge representation, the representational formalisms used in DENDRAL are highly specific to the domain of chemistry. Over time, researchers became interested in standardized knowledge representation formalisms and ontologies that could streamline the process of creating new expert systems. In so doing, they ventured into territory previously explored by philosophers of science and of language. The discipline imposed in AI by the need for one’s theories to “work” has led to more rapid and deeper progress than was the case when these problems were the exclusive domain of philosophy (although it has at times also led to the repeated reinvention of the wheel).

The creation of comprehensive taxonomies or classifications dates back to ancient times. Aristotle (384–322 B.C.) strongly emphasized classification and categorization schemes. His *Organon*, a collection of works on logic assembled by his students after his death, included a treatise called *Categories* in which he attempted to construct what we would now call an upper ontology. He also introduced the notions of **genus** and **species** for lower-level classification, although not with their modern, specifically biological meaning. Our present system of biological classification, including the use of “binomial nomenclature” (classification via genus and species in the technical sense), was invented by the Swedish biologist Carolus Linnaeus, or Carl von Linne (1707–1778). The problems associated with natural kinds and inexact category boundaries have been addressed by Wittgenstein (1953), Quine (1953), Lakoff (1987), and Schwartz (1977), among others.

Interest in larger-scale ontologies is increasing. The CYC project (Lenat, 1995; Lenat and Guha, 1990) has released a 6,000-concept upper ontology with 60,000 facts, and licenses

a much larger global ontology. The IEEE has established subcommittee P1600.1, the Standard Upper Ontology Working Group, and the Open Mind Initiative has enlisted over 7,000 Internet users to enter more than 400,000 facts about commonsense concepts. On the Web, standards such as RDF, XML, and the Semantic Web (Berners-Lee *et al.*, 2001) are emerging, but are not yet widely used. The conferences on *Formal Ontology in Information Systems* (FOIS) contain many interesting papers on both general and domain-specific ontologies.

The taxonomy used in this chapter was developed by the authors and is based in part on their experience in the CYC project and in part on work by Hwang and Schubert (1993) and Davis (1990). An inspirational discussion of the general project of commonsense knowledge representation appears in Hayes's (1978, 1985b) "The Naive Physics Manifesto."

The representation of time, change, actions, and events has been studied extensively in philosophy and theoretical computer science as well as in AI. The oldest approach is **temporal logic**, which is a specialized logic in which each model describes a complete trajectory through time (usually either linear or branching), rather than just a static relational structure. The logic includes **modal operators** that are applied to formulas; $\Box p$ means " p will be true at all times in the future," and $\Diamond p$ means " p will be true at some time in the future." The study of temporal logic was initiated by Aristotle and the Megarian and Stoic schools in ancient Greece. In modern times, Findlay (1941) was the first to suggest a formal calculus for reasoning about time, but the work of Arthur Prior (1967) is considered the most influential. Textbooks on temporal logic include those by Rescher and Urquhart (1971) and van Benthem (1983).

Theoretical computer scientists have long been interested in formalizing the properties of programs, viewed as sequences of computational actions. Burstall (1974) introduced the idea of using modal operators to reason about computer programs. Soon thereafter, Vaughan Pratt (1976) designed **dynamic logic**, in which modal operators indicate the effects of programs or other actions (see also Harel, 1984). For instance, in dynamic logic, if α is the name of a program, then " $[\alpha]p$ " means " p would be true in all world states resulting from executing program α in the current world state", and " $\langle\alpha\rangle p$ " means " p would be true in at least one world state resulting from executing program α in the current world state." Dynamic logic was applied to the actual analysis of programs by Fischer and Ladner (1977). Pnueli (1977) introduced the idea of using classical temporal logic to reason about programs.

Whereas temporal logic puts time directly into the model theory of the language, representations of time in AI have tended to incorporate axioms about times and events explicitly in the knowledge base, giving time no special status in the logic. This approach can allow for greater clarity and flexibility in some cases. Also, temporal knowledge expressed in first-order logic can be more easily integrated with other knowledge that has been accumulated in that notation.

The earliest treatment of time and action in AI was John McCarthy's (1963) situation calculus. The first AI system to make substantial use of general-purpose reasoning about actions in first-order logic was QA3 (Green, 1969b). Kowalski (1979b) developed the idea of reifying propositions within situation calculus.

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem insoluble within first-order logic, and it spurred a great

deal of research into nonmonotonic logics. Philosophers from Dreyfus (1972) to Crockett (1994) have cited the frame problem as one symptom of the inevitable failure of the entire AI enterprise. The partial solution of the representational frame problem using successor-state axioms is due to Ray Reiter (1991); a solution of the inferential frame problem can be traced to work by Holldobler and Schneeberger (1990) on what became known as fluent calculus (Thielscher, 1999). The discussion in this chapter is based partly on the analyses by Lin and Reiter (1997) and Thielscher (1999). Books by Shanahan (1997) and Reiter (2001b) give complete, modern treatments of reasoning about action in situation calculus.

The partial resolution of the frame problem has rekindled interest in the declarative approach to reasoning about actions, which had been eclipsed by special-purpose planning systems since the early 1970s. (See Chapter 11.) Under the banner of **cognitive robotics**, much progress has been made on logical representations of action and time. The GOLOG language uses the full expressive power of logic programming to describe actions and plans (Levesque *et al.*, 1997a) and has been extended to handle concurrent actions (Giacomo *et al.*, 2000), stochastic environments (Boutilier *et al.*, 2000), and sensing (Reiter, 2001a).

The event calculus was introduced by Kowalski and Sergot (1986) to handle continuous time, and there have been several variations (Sadri and Kowalski, 1995). Shanahan (1999) presents a good short overview. James Allen introduced time intervals for the same reason (Allen, 1983, 1984), arguing that intervals were much more natural than situations for reasoning about extended and concurrent events. Peter Ladkin (1986a, 1986b) introduced “concave” time intervals (intervals with gaps; essentially, unions of ordinary “convex” time intervals) and applied the techniques of mathematical abstract algebra to time representation. Allen (1991) systematically investigates the wide variety of techniques available for time representation. Shoham (1987) describes the reification of events and sets forth a novel scheme of his own for the purpose. There are significant commonalities between the event-based ontology given in this chapter and an analysis of events due to the philosopher Donald Davidson (1980). The **histories** in Pat Hayes’s (1985a) ontology of liquids also have much the same flavor.

The question of the ontological status of substances has a long history. Plato proposed that substances were abstract entities entirely distinct from physical objects; he would say *MadeOf(Butter₃, Butter)* rather than *Butter₃ ∈ Butter*. This leads to a substance hierarchy in which, for example, *UnsaltedButter* is a more specific substance than *Butter*. The position adopted in this chapter, in which substances are categories of objects, was championed by Richard Montague (1973). It has also been adopted in the CYC project. Copeland (1993) mounts a serious, but not invincible, attack. The alternative approach mentioned in the chapter, in which butter is one object consisting of all buttery objects in the universe, was proposed originally by the Polish logician Leśniewski (1916). His **mereology** (the name is derived from the Greek word for “part”) used the part–whole relation as a substitute for mathematical set theory, with the aim of eliminating abstract entities such as sets. A more readable exposition of these ideas is given by Leonard and Goodman (1940), and Goodman’s *The Structure of Appearance* (1977) applies the ideas to various problems in knowledge representation. While some aspects of the mereological approach are awkward—for example, the need for a separate inheritance mechanism based on part–whole relations—the approach gained the

support of Quine (1960). Harry Bunt (1985) has provided an extensive analysis of its use in knowledge representation.

Mental objects and states have been the subject of intensive study in philosophy and AI. **Modal logic** is the classical method for reasoning about knowledge in philosophy. Modal logic augments first-order logic with modal operators, such as *B* (believes) and *K* (knows), that take *sentences* rather than terms as arguments. The proof theory for modal logic restricts substitution within modal contexts, thereby achieving referential opacity. The modal logic of knowledge was invented by Jaakko Hintikka (1962). Saul Kripke (1963) defined the semantics of the modal logic of knowledge in terms of **possible worlds**. Roughly speaking, a world is possible for an agent if it is consistent with everything the agent knows. From this, one can derive rules of inference involving the *K* operator. Robert C. Moore relates the modal logic of knowledge to a style of reasoning about knowledge that refers directly to possible worlds in first-order logic (Moore, 1980, 1985). Modal logic can be an intimidatingly arcane field, but it has found significant applications in reasoning about information in distributed computer systems. The book *Reasoning about Knowledge* by Fagin *et al.* (1995) provides a thorough introduction to the modal approach. The biennial conference on *Theoretical Aspects of Reasoning About Knowledge* (TARK) covers applications of the theory of knowledge in AI, economics, and distributed systems.

The syntactic theory of mental objects was first studied in depth by Kaplan and Montague (1960), who showed that it led to paradoxes if not handled carefully. Because it has a natural model in terms of beliefs as physical configurations of a computer or a brain, it has been popular in AI in recent years. Konolige (1982) and Haas (1986) used it to describe inference engines of limited power, and Morgenstern (1987) showed how it could be used to describe knowledge preconditions in planning. The methods for planning observation actions in Chapter 12 are based on the syntactic theory. Ernie Davis (1990) gives an excellent comparison of the syntactic and modal theories of knowledge.

The Greek philosopher Porphyry (c. 234–305 A.D.), commenting on Aristotle's *Categories*, drew what might qualify as the first semantic network. Charles S. Peirce (1909) developed existential graphs as the first semantic network formalism using modern logic. Ross Quillian (1961), driven by an interest in human memory and language processing, initiated work on semantic networks within AI. An influential paper by Marvin Minsky (1975) presented a version of semantic networks called **frames**; a frame was a representation of an object or category, with attributes and relations to other objects or categories. Although the paper served to initiate interest in the field of knowledge representation *per se*, it was criticized as a recycling of earlier ideas developed in object-oriented programming, such as inheritance and the use of default values (Dahl *et al.*, 1970; Birtwistle *et al.*, 1973). It is not clear to what extent the latter papers on object-oriented programming were influenced in turn by early AI work on semantic networks.

The question of semantics arose quite acutely with respect to Quillian's semantic networks (and those of others who followed his approach), with their ubiquitous and very vague "IS-A links," as well as other early knowledge representation formalisms such as that of MERLIN (Moore and Newell, 1973) with its mysterious "flat" and "cover" operations. Woods' (1975) famous article "What's In a Link?" drew the attention of AI researchers to the

need for precise semantics in knowledge representation formalisms. Brachman (1979) elaborated on this point and proposed solutions. Patrick Hayes's (1979) "The Logic of Frames" cut even deeper, claiming that "Most of 'frames' is just a new syntax for parts of first-order logic." Drew McDermott's (1978b) "Tarskian Semantics, or, No Notation without Denotation!" argued that the model-theoretic approach to semantics used in first-order logic should be applied to all knowledge representation formalisms. This remains a controversial idea; notably, McDermott himself has reversed his position in "A Critique of Pure Reason" (McDermott, 1987). NETL (Fahlman, 1979) was a sophisticated semantic network system whose IS-A links (called "virtual copy," or VC, links) were based more on the notion of "inheritance" characteristic of frame systems or of object-oriented programming languages than on the subset relation and were much more precisely defined than Quillian's links from the pre-Woods era. NETL is particularly intriguing because it was intended to be implemented in parallel hardware to overcome the difficulty of retrieving information from large semantic networks. David Touretzky (1986) subjects inheritance to rigorous mathematical analysis. Selman and Levesque (1993) discuss the complexity of inheritance with exceptions, showing that in most formulations it is NP-complete.

The development of description logics is the most recent stage in a long line of research aimed at finding useful subsets of first-order logic for which inference is computationally tractable. Hector Levesque and Ron Brachman (1987) showed that certain logical constructs—notably, certain uses of disjunction and negation—were primarily responsible for the intractability of logical inference. Building on the KL-ONE system (Schmolze and Lipkis, 1983), a number of systems have been developed whose designs incorporate the results of theoretical complexity analysis, most notably KRYPTON (Brachman *et al.*, 1983) and Classic (Borgida *et al.*, 1989). The result has been a marked increase in the speed of inference and a much better understanding of the interaction between complexity and expressiveness in reasoning systems. Calvanese *et al.* (1999) summarize the state of the art. Against this trend, Doyle and Patil (1991) have argued that restricting the expressiveness of a language either makes it impossible to solve certain problems or encourages the user to circumvent the language restrictions through nonlogical means.

The three main formalisms for dealing with nonmonotonic inference—circumscription (McCarthy, 1980), default logic (Reiter, 1980), and modal nonmonotonic logic (McDermott and Doyle, 1980)—were all introduced in one special issue of the AI Journal. Answer set programming can be seen as an extension of negation as failure or as a refinement of circumscription; the underlying theory of stable model semantics was introduced by Gelfond and Lifschitz (1988) and the leading answer set programming systems are DLV (Eiter *et al.*, 1998) and SMODELS (Niemelä *et al.*, 2000). The disk drive example comes from the SMODELS user manual (Syrjänen, 2000). Lifschitz (2001) discusses the use of answer set programming for planning. Brewka *et al.* (1997) give a good overview of the various approaches to nonmonotonic logic. Clark (1978) covers the negation-as-failure approach to logic programming and Clark completion. Van Emden and Kowalski (1976) show that every Prolog program without negation has a unique minimal model. Recent years have seen renewed interest in applications of nonmonotonic logics to large-scale knowledge representation systems. The BENINQ systems for handling insurance benefits inquiries was perhaps the first commercially success-

ful application of a nonmonotonic inheritance system (Morgenstern, 1998). Lifschitz (2001) discusses the application of answer set programming to planning. A variety of nonmonotonic reasoning systems based on logic programming are documented in the proceedings of the conferences on *Logic Programming and Nonmonotonic Reasoning* (LPNMR).

The study of truth maintenance systems began with the TMS (Doyle, 1979) and RUP (McAllester, 1980) systems, both of which were essentially JTMSs. The ATMS approach was described in a series of papers by Johan de Kleer (1986a, 1986b, 1986c). *Building Problem Solvers* (Forbus and de Kleer, 1993) explains in depth how TMSs can be used in AI applications. Nayak and Williams (1997) show how an efficient TMS makes it feasible to plan the operations of a NASA spacecraft in real time.

For obvious reasons, this chapter does not cover *every* area of knowledge representation in depth. The three principal topics omitted are the following:

QUALITATIVE PHYSICS

- ◇ **Qualitative physics:** Qualitative physics is a subfield of knowledge representation concerned specifically with constructing a logical, nonnumeric theory of physical objects and processes. The term was coined by Johan de Kleer (1975), although the enterprise could be said to have started in Fahlman's (1974) BUILD, a sophisticated planner for constructing complex towers of blocks. Fahlman discovered in the process of designing it that most of the effort (80%, by his estimate) went into modeling the physics of the blocks world to calculate the stability of various subassemblies of blocks, rather than into planning per se. He sketches a hypothetical naive-physics-like process to explain why young children can solve BUILD-like problems without access to the high-speed floating-point arithmetic used in BUILD's physical modeling. Hayes (1985a) uses "histories"—four-dimensional slices of space-time similar to Davidson's events—to construct a fairly complex naive physics of liquids. Hayes was the first to prove that a bath with the plug in will eventually overflow if the tap keeps running and that a person who falls into a lake will get wet all over. De Kleer and Brown (1985) and Ken Forbus (1985) attempted to construct something like a general-purpose theory of the physical world, based on qualitative abstractions of physical equations. In recent years, qualitative physics has developed to the point where it is possible to analyze an impressive variety of complex physical systems (Sacks and Joskowicz, 1993; Yip, 1991). Qualitative techniques have been used to construct novel designs for clocks, windscreen wipers, and six-legged walkers (Subramanian, 1993; Subramanian and Wang, 1994). The collection *Readings in Qualitative Reasoning about Physical Systems* (Weld and de Kleer, 1990) provides a good introduction to the field.

SPATIAL REASONING

- ◇ **Spatial reasoning:** The reasoning necessary to navigate in the wumpus world and shopping world is trivial in comparison to the rich spatial structure of the real world. The earliest serious attempt to capture commonsense reasoning about space appears in the work of Ernest Davis (1986, 1990). The region connection calculus of Cohn *et al.* (1997) supports a form of qualitative spatial reasoning and has led to new kinds of geographical information system. As with qualitative physics, an agent can go a long way, so to speak, without resorting to a full metric representation. When such a representation is necessary, techniques developed in robotics (Chapter 25) can be used.

◇ **Psychological reasoning:** Psychological reasoning involves the development of a working *psychology* for artificial agents to use in reasoning about themselves and other agents. This is often based on so-called folk psychology, the theory that humans in general are believed to use in reasoning about themselves and other humans. When AI researchers provide their artificial agents with psychological theories for reasoning about other agents, the theories are frequently based on the researchers' description of the logical agents' own design. Psychological reasoning is currently most useful within the context of natural language understanding, where divining the speaker's intentions is of paramount importance.

The proceedings of the international conferences on *Principles of Knowledge Representation and Reasoning* provide the most up-to-date sources for work in this area. *Readings in Knowledge Representation* (Brachman and Levesque, 1985) and *Formal Theories of the Commonsense World* (Hobbs and Moore, 1985) are excellent anthologies on knowledge representation; the former focuses more on historically important papers in representation languages and formalisms, the latter on the accumulation of the knowledge itself. Davis (1990), Stefik (1995), and Sowa (1999) provide textbook introductions to knowledge representation.

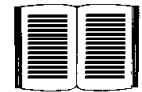
EXERCISES

10.1 Write sentences to define the effects of the *Shoot* action in the wumpus world. Describe its effects on the wumpus and remember that shooting uses the agent's arrow.

10.2 Within situation calculus, write an axiom to associate time 0 with the situation S_0 and another axiom to associate the time t with any situation that is derived from S_0 by a sequence of t actions.

10.3 In this exercise, we will consider the problem of planning a route for a robot to take from one city to another. The basic action taken by the robot is $Go(x, y)$, which takes it from city x to city y if there is a direct route between those cities. $DirectRoute(x, y)$ is true if and only if there is a direct route from x to y ; you can assume that all such facts are already in the KB. (See the map on page 63.) The robot begins in Arad and must reach Bucharest.

- a. Write a suitable logical description of the initial situation of the robot.
- b. Write a suitable logical query whose solutions will provide possible paths to the goal.
- c. Write a sentence describing the *Go* action.
- d. Now suppose that following the direct route between two cities consumes an amount of fuel equal to the distance between the cities. The robot starts with fuel at full capacity. Augment your representation to include these considerations. Your action description should be such that the query you specified earlier will still result in feasible plans.
- e. Describe the initial situation, and write a new rule or rules describing the *Go* action.
- f. Now suppose some of the vertices are also gas stations, at which the robot can fill its tank. Extend your representation and write all the rules needed to describe gas stations, including the *Fillup* action.



10.4 Investigate ways to extend the event calculus to handle *simultaneous* events. Is it possible to avoid a combinatorial explosion of axioms?

10.5 Represent the following seven sentences using and extending the representations developed in the chapter:

- a. Water is a liquid between 0 and 100 degrees.
- b. Water boils at 100 degrees.
- c. The water in John's water bottle is frozen.
- d. Perrier is a kind of water.
- e. John has Perrier in his water bottle.
- f. All liquids have a freezing point.
- g. A liter of water weighs more than a liter of alcohol.

Now repeat the exercise using a representation based on the mereological approach, in which, for example, *Water* is an object containing as parts all the water in the world.

10.6 Write definitions for the following:

- a. *ExhaustivePartDecomposition*
- b. *PartPartition*
- c. *PartwiseDisjoint*

These should be analogous to the definitions for *ExhaustiveDecomposition*, *Partition*, and *Disjoint*. Is it the case that $PartPartition(s, BunchOf(s))$? If so, prove it; if not, give a counterexample and define sufficient conditions under which it does hold.

10.7 Write a set of sentences that allows one to calculate the price of an individual tomato (or other object), given the price per pound. Extend the theory to allow the price of a bag of tomatoes to be calculated.

10.8 An alternative scheme for representing measures involves applying the units function to an abstract length object. In such a scheme, one would write $Inches(Length(L_1)) = 1.5$. How does this scheme compare with the one in the chapter? Issues include conversion axioms, names for abstract quantities (such as "50 dollars"), and comparisons of abstract measures in different units (50 inches is more than 50 centimeters).

10.9 Construct a representation for exchange rates between currencies that allows fluctuations on a daily basis.

10.10 This exercise concerns the relationships between event categories and the time intervals in which they occur.

- a. Define the predicate $T(c, i)$ in terms of *During* and \in .
- b. Explain precisely why we do not need two different notations to describe conjunctive event categories.
- c. Give a formal definition for $T(OneOf(p, q), i)$ and $T(Either(p, q), i)$.
- d. Explain why it makes sense to have two forms of negation of events, analogous to the two forms of disjunction. Call them *Not* and *Never* and give them formal definitions.

10.11 Define the predicate *Fixed*, where $Fixed(Location(x))$ means that the location of object x is fixed over time.

10.12 Define the predicates *Before*, *After*, *During*, and *Overlap*, using the predicate *Meet* and the functions *Start* and *End*, but not the function *Time* or the predicate $<$.

10.13 Section 10.5 used the predicates *Link* and *LinkText* to describe connections between web pages. Using the *InTag* and *GetPage* predicates, among others, write definitions for *Link* and *LinkText*.

10.14 One part of the shopping process that was not covered in this chapter is checking for compatibility between items. For example, if a customer orders a computer, is it matched with the right peripherals? If a digital camera is ordered, does it have the right memory card and batteries? Write a knowledge base that will decide whether a set of items is compatible and that can be used to suggest replacements or additional items if they are not compatible. Make sure that the knowledge base works with at least one line of products, and is easily extensible to other lines.

10.15 Add rules to extend the definition of the predicate $Name(s, c)$ so that a string such as “laptop computer” matches against the appropriate category names from a variety of stores. Try to make your definition general. Test it by looking at ten online stores, and at the category names they give for three different categories. For example, for the category of laptops, we found the names “Notebooks,” “Laptops,” “Notebook Computers,” “Notebook,” “Laptops and Notebooks,” and “Notebook PCs.” Some of these can be covered by explicit *Name* facts, while others could be covered by rules for handling plurals, conjunctions, etc.

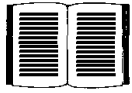
10.16 A complete solution to the problem of inexact matches to the buyer’s description in shopping is very difficult and requires a full array of natural language processing and information retrieval techniques. (See Chapters 22 and 23.) One small step is to allow the user to specify minimum and maximum values for various attributes. We will insist that the buyer use the following grammar for product descriptions:

<i>Description</i>	\rightarrow	<i>Category</i> [<i>Connector</i> <i>Modifier</i>]*
<i>Connector</i>	\rightarrow	“with” “and” “,”
<i>Modifier</i>	\rightarrow	<i>Attribute</i> <i>Attribute Op Value</i>
<i>Op</i>	\rightarrow	“=” “>” “<”

Here, *Category* names a product category, *Attribute* is some feature such as “CPU” or “price,” and *Value* is the target value for the attribute. So the query “computer with at least a 2.5-GHz CPU for under \$1000” must be re-expressed as “computer with CPU > 2.5 GHz and price < \$1000.” Implement a shopping agent that accepts descriptions in this language.

10.17 Our description of Internet shopping omitted the all-important step of actually *buying* the product. Provide a formal logical description of buying, using event calculus. That is, define the sequence of events that occurs when a buyer submits a credit card purchase and then eventually gets billed and receives the product.

10.18 Describe the event of trading something for something else. Describe buying as a kind of trading in which one of the objects traded is a sum of money.



10.19 The two preceding exercises assume a fairly primitive notion of ownership. For example, the buyer starts by *owning* the dollar bills. This picture begins to break down when, for example, one's money is in the bank, because there is no longer any specific collection of dollar bills that one owns. The picture is complicated still further by borrowing, leasing, renting, and bailment. Investigate the various commonsense and legal concepts of ownership, and propose a scheme by which they can be represented formally.

10.20 You are to create a system for advising computer science undergraduates on what courses to take over an extended period in order to satisfy the program requirements. (Use whatever requirements are appropriate for your institution.) First, decide on a vocabulary for representing all the information, and then represent it; then use an appropriate query to the system, that will return a legal program of study as a solution. You should allow for some tailoring to individual students, in that your system should ask what courses or equivalents the student has already taken, and not generate programs that repeat those courses.

Suggest ways in which your system could be improved—for example to take into account knowledge about student preferences, the workload, good and bad instructors, and so on. For each kind of knowledge, explain how it could be expressed logically. Could your system easily incorporate this information to find the *best* program of study for a student?

10.21 Figure 10.1 shows the top levels of a hierarchy for everything. Extend it to include as many real categories as possible. A good way to do this is to cover all the things in your everyday life. This includes objects and events. Start with waking up, and proceed in an orderly fashion noting everything that you see, touch, do, and think about. For example, a random sampling produces music, news, milk, walking, driving, gas, Soda Hall, carpet, talking, Professor Fateman, chicken curry, tongue, \$7, sun, the daily newspaper, and so on.

You should produce both a single hierarchy chart (on a large sheet of paper) and a listing of objects and categories with the relations satisfied by members of each category. Every object should be in a category, and every category should be in the hierarchy.

10.22 (Adapted from an example by Doug Lenat.) Your mission is to capture, in logical form, enough knowledge to answer a series of questions about the following simple sentence:

Yesterday John went to the North Berkeley Safeway supermarket and bought two pounds of tomatoes and a pound of ground beef.

Start by trying to represent the content of the sentence as a series of assertions. You should write sentences that have straightforward logical structure (e.g., statements that objects have certain properties, that objects are related in certain ways, that all objects satisfying one property satisfy another). The following might help you get started:

- Which classes, objects, and relations would you need? What are their parents, siblings and so on? (You will need events and temporal ordering, among other things.)
- Where would they fit in a more general hierarchy?
- What are the constraints and interrelationships among them?
- How detailed must you be about each of the various concepts?

The knowledge base you construct must be capable of answering a list of questions that we will give shortly. Some of the questions deal with the material stated explicitly in the story,

but most of them require one to have other background knowledge—to read between the lines. You’ll have to deal with what kind of things are at a supermarket, what is involved with purchasing the things one selects, what will purchases be used for, and so on. Try to make your representation as general as possible. To give a trivial example: don’t say “People buy food from Safeway,” because that won’t help you with those who shop at another supermarket. Don’t say “Joe made spaghetti with the tomatoes and ground beef,” because that won’t help you with anything else at all. Also, don’t turn the questions into answers; for example, question (c) asks “Did John buy any meat?”—not “Did John buy a pound of ground beef?”

Sketch the chains of reasoning that would answer the questions. In the process of doing so, you will no doubt need to create additional concepts, make additional assertions, and so on. If possible, use a logical reasoning system to demonstrate the sufficiency of your knowledge base. Many of the things you write might be only approximately correct in reality, but don’t worry too much; the idea is to extract the common sense that lets you answer these questions at all. A truly complete answer to this question is *extremely* difficult, probably beyond the state of the art of current knowledge representation. But you should be able to put together a consistent set of axioms for the limited questions posed here.

- a. Is John a child or an adult? [Adult]
- b. Does John now have at least two tomatoes? [Yes]
- c. Did John buy any meat? [Yes]
- d. If Mary was buying tomatoes at the same time as John, did he see her? [Yes]
- e. Are the tomatoes made in the supermarket? [No]
- f. What is John going to do with the tomatoes? [Eat them]
- g. Does Safeway sell deodorant? [Yes]
- h. Did John bring any money to the supermarket? [Yes]
- i. Does John have less money after going to the supermarket? [Yes]

10.23 Make the necessary additions or changes to your knowledge base from the previous exercise so that the questions that follow can be answered. Show that they can indeed be answered by the KB, and include in your report a discussion of the fixes, explaining why they were needed, whether they were minor or major, and so on.

- a. Are there other people in Safeway while John is there? [Yes—staff!]
- b. Is John a vegetarian? [No]
- c. Who owns the deodorant in Safeway? [Safeway Corporation]
- d. Did John have an ounce of ground beef? [Yes]
- e. Does the Shell station next door have any gas? [Yes]
- f. Do the tomatoes fit in John’s car trunk? [Yes]

10.24 Recall that inheritance information in semantic networks can be captured logically by suitable implication sentences. In this exercise, we will consider the efficiency of using such sentences for inheritance.

- a. Consider the information content in a used-car catalog such as Kelly's Blue Book—for example, that 1973 Dodge Vans are worth \$575. Suppose all this information (for 11,000 models) is encoded as logical rules, as suggested in the chapter. Write down three such rules, including that for 1973 Dodge Vans. How would you use the rules to find the value of a *particular* car (e.g., JB, which is a 1973 Dodge Van), given a backward-chaining theorem prover such as Prolog?
- b. Compare the time efficiency of the backward-chaining method for solving this problem with the inheritance method used in semantic nets.
- c. Explain how forward chaining allows a logic-based system to solve the same problem efficiently, assuming that the KB contains only the 11,000 rules about prices.
- d. Describe a situation in which neither forward nor backward chaining on the rules will allow the price query for an individual car to be handled efficiently.
- e. Can you suggest a solution enabling this type of query to be solved efficiently in all cases in logic systems? [*Hint:* Remember that two cars of the same category have the same price.]

10.25 One might suppose that the syntactic distinction between unboxed links and singly boxed links in semantic networks is unnecessary, because singly boxed links are always attached to categories; an inheritance algorithm could simply assume that an unboxed link attached to a category is intended to apply to all members of that category. Show that this argument is fallacious, giving examples of errors that would arise.