CHAPTER

4

KNOWLEDGE REPRESENTATION ISSUES

In general we are least aware of what our minds do best.

—Marvin Minsky
(1927-), American cognitive scientist

In Chapter 1, we discussed the role that knowledge plays in AI systems. In succeeding chapters up until now, though, we have paid little attention to knowledge and its importance as we instead focused on basic frameworks for building search-based problem-solving programs. These methods are sufficiently general that we have been able to discuss them without reference to how the knowledge they need is to be represented. For example, in discussing the best-first search algorithm, we hid all the references to domain-specific knowledge in the generation of successors and the computation of the h' function. Although these methods are useful and form the skeleton of many of the methods we are about to discuss, their problem-solving power is limited precisely because of their generality. As we look in more detail at ways of representing knowledge, it becomes clear that particular knowledge representation models allow for more specific, more powerful problem-solving mechanisms that operate on them. In this part of the book, we return to the topic of knowledge and examine specific techniques that can be used for representing and manipulating knowledge within programs.

4.1 REPRESENTATIONS AND MAPPINGS

In order to solve the complex problems encountered in artificial intelligence, one needs both a large amount of knowledge and some mechanisms for manipulating that knowledge to create solutions to new problems. A variety of ways of representing knowledge (facts) have been exploited in AI programs. But before we can talk about them individually, we must consider the following point that pertains to all discussions of representation, namely that we are dealing with two different kinds of entities:

- Facts: truths in some relevant world. These are the things we want to represent.
- Representations of facts in some chosen formalism. These are the things we will actually be able to manipulate.

One way to think of structuring these entities is as two levels:

- The *knowledge level*, at which facts (including each agent's behaviors and current goals) are described.

- The *symbol level*, at which representations of objects at the knowledge level are defined in terms of symbols that can be manipulated by programs.

See Newell [1982] for a detailed exposition of this view in the context of agents and their goals and behaviors. In the rest of our discussion here, we will follow a model more like the one shown in Fig. 4.1. Rather than thinking of one level on top of another, we will focus on facts, on representations, and on the two-way mappings that must exist between them. We will call these links *representation mappings*. The forward representation mapping maps from facts to representations. The backward representation mapping goes the other way, from representations to facts.

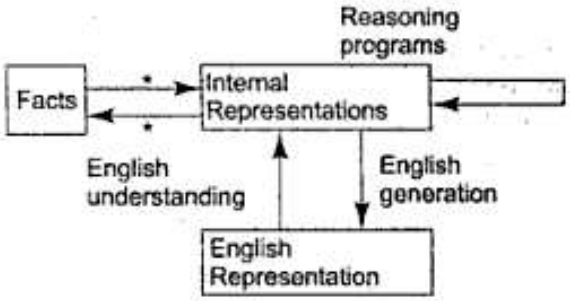

Fig. 4.1 *Mappings between Facts and Representations*

One representation of facts is so common that it deserves special mention: natural language (particularly English) sentences. Regardless of the representation for facts that we use in a program, we may also need to be concerned with an English representation of those facts in order to facilitate getting information into and out of the system. In this case, we must also have mapping functions from English sentences to the representation we are actually going to use and from it back to sentences. Figure 4.1 shows how these three kinds of objects relate to each other.

Let's look at a simple example using mathematical logic as the representational formalism. Consider the English sentence:

Spot is a dog.

The fact represented by that English sentence can also be represented in logic as:

$dog(Spot)$

Suppose that we also have a logical representation of the fact that all dogs have tails:

$\forall x : dog(x) \rightarrow hastail(x)$

Then, using the deductive mechanisms of logic, we may generate the new representation object:

$hastail(Spot)$

Using an appropriate backward mapping function, we could then generate the English sentence:

Spot has a tail.

Or we could make use of this representation of a new fact to cause us to take some appropriate action or to derive representations of additional facts.

It is important to keep in mind that usually the available mapping functions are not one-to-one. In fact, they are often not even functions but rather many-to-many relations. (In other words, each object in the domain may map to several elements in the range, and several elements in the domain may map to the same element of the range.) This is particularly true of the mappings involving English representations of facts. For example, the two sentences "All dogs have tails" and "Every dog has a tail" could both represent the same fact, namely,

that every dog has at least one tail. On the other hand, the former could represent either the fact that every dog has at least one tail or the fact that each dog has several tails. The latter may represent either the fact that every dog has at least one tail or the fact that there is a tail that every dog has. As we will see shortly, when we try to convert English sentences into some other representation, such as logical propositions, we must first decide what facts the sentences represent and then convert those facts into the new representation.

The starred links of Fig. 4.1 are key components of the design of any knowledge-based program. To see why, we need to understand the role that the internal representation of a fact plays in a program. What an AI program does is to manipulate the internal representations of the facts it is given. This manipulation should result in new structures that can also be interpreted as internal representations of facts. More precisely, these structures should be the internal representations of facts that correspond to the answer to the problem described by the starting set of facts.

Sometimes, a good representation makes the operation of a reasoning program not only correct but trivial. A well-known example of this occurs in the context of the mutilated checker board problem, which can be stated as follows:

> **The Mutilated Checker board Problem.** Consider a normal checker board from which two squares, in opposite corners, have been removed. The task is to cover all the remaining squares exactly with dominoes, each of which covers two squares. No overlapping, either of dominoes on top of each other or of dominoes over the boundary of the mutilated board are allowed. Can this task be done?

One way to solve this problem is to try to enumerate, exhaustively, all possible tilings to see if one works. But suppose one wants to be more clever. Figure 4.2 shows three ways in which the mutilated checker board could be represented (to a person). The first representation does not directly suggest the answer to the problem. The second may; the third does, when combined with the single additional fact that each domino must cover exactly one white square and one black square. Even for human problem solvers a representation shift may make an enormous difference in problem-solving effectiveness. Recall that we saw a slightly less dramatic version of this phenomenon with respect to a problem-solving program in Section 1.3.1, where we considered two different ways of representing a tic-tac-toe board, one of which was as a magic square.

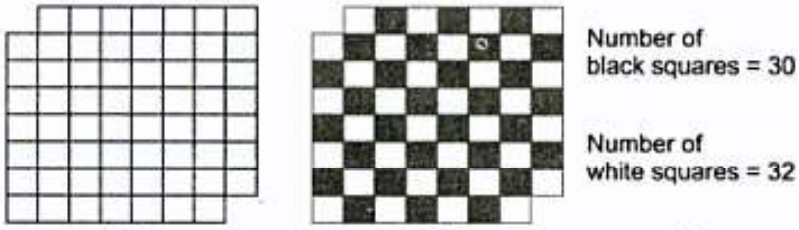

Fig. 4.2 *Three Representations of a Mutilated Checker board*

Figure 4.3 shows an expanded view of the starred part of Fig. 4.1 The dotted line across the top represents the abstract reasoning process that a program is intended to model. The solid line across the bottom represents the concrete reasoning process that a particular program performs. This program successfully models the abstract process to the extent that, when the backward representation mapping is applied to the program's output, the appropriate final facts are actually generated. If either the program's operation or one of the representation mappings is not faithful to the problem that is being modeled, then the final facts will probably not be the desired ones. The key role that is played by the nature of the representation mapping is apparent from this figure. If no good mapping can be defined for a problem, then no matter how good the program to solve the problem is, it will not be able to produce answers that correspond to real answers to the problem.

It is interesting to note that Fig. 4.3 looks very much like the sort of figure that might appear in a general programming book as a description of the relationship between an abstract data type (such as a set) and a concrete implementation of that type (e.g., as a linked list of elements). There are some differences, though, between this figure and the formulation usually used in programming texts (such as Aho *et al.* [1983]). For example, in data type design it is expected that the mapping that we are calling the backward representation mapping is a function (i.e., every representation corresponds to only one fact) and that it is onto (i.e., there is at least one representation for every fact). Unfortunately, in many AI domains, it may not be possible to come up with such a representation mapping, and we may have to live with one that gives less ideal results. But the main idea of what we are doing is the same as what programmers always do, namely to find concrete implementations of abstract concepts.

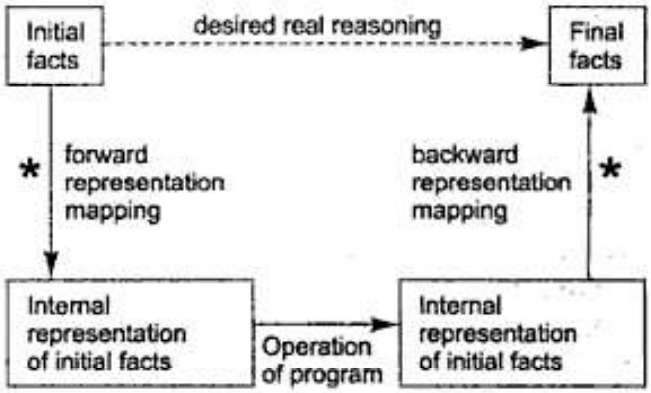

Fig. 4.3 *Representation of Facts*

4.2 APPROACHES TO KNOWLEDGE REPRESENTATION

A good system for the representation of knowledge in a particular domain should possess the following four properties:

Representational Adequacy — the ability to represent all of the kinds of knowledge that are needed in that domain.

- Inferential Adequacy — the ability to manipulate the representational structures in such a way as to derive new structures corresponding to new knowledge inferred from old.
- Inferential Efficiency — the ability to incorporate into the knowledge structure additional information that can be used to focus the attention of the inference mecha- nisms in the most promising directions.
- Acquisitional Efficiency — the ability to acquire new information easily. The simplest case involves direct insertion, by a person, of new knowledge into the database. Ideally, the program itself would be able to control knowledge acquisition.

Unfortunately, no single system that optimizes all of the capabilities for all kinds of knowledge has yet been found. As a result, multiple techniques for knowledge representation exist. Many programs rely on more than one technique. In the chapters that follow, the most important of these techniques are described in detail. But in this section, we provide a simple, example-based introduction to the important ideas.

Simple Relational Knowledge

The simplest way to represent declarative facts is as a set of relations of the same sort used in database systems. Figure 4.4 shows an example of such a relational system.

Player	Height	Weight	Bats-Throws
Hank Aaron	6-0	180	Right-Right
Willie Mays	5-10	170	Right-Right
Babe Ruth	6-2	215	Left-Left
Ted Williams	6-3	205	Left-Right
player_info('hank aaron', '6-0', 180,right-right).			

Fig. 4.4 *Simple Relational Knowledge and a sample fact in Prolog*

The reason that this representation is simple is that standing alone it provides very weak inferential capabilities But knowledge represented in this form may serve as the input to more powerful inference engines. For example, given just the facts of Fig. 4.4, it is not possible even to answer the simple question, "Who is the heaviest player?" But if a procedure for finding the heaviest player is provided, then these facts will enable the procedure to compute an answer. If, instead, we are provided with a set of rules for deciding which hitter to put up against a given pitcher (based on right- and left-handedness, say), then this same relation can provide at least some of the information required by those rules.

Providing support for relational knowledge is what database systems are designed to do. Thus we do not need to discuss this kind of knowledge representation structure further here. The practical issues that arise in linking a database system that provides this kind of support to a knowledge representation system that provides some of the other capabilities that we are about to discuss have already been solved in several commercial products.

Inheritable Knowledge

The relational knowledge of Fig. 4.4 corresponds to a set of attributes and associated values that together describe the objects of the knowledge base. Knowledge about objects, their attributes, and their values need not be as simple as that shown in our example. In particular, it is possible to augment the basic representation with inference mechanisms that operate on the structure of the representation. For this to be effective, the structure must be designed to correspond to the inference mechanisms that are desired. One of the most useful forms of inference is *property inheritance*, in which elements of specific classes inherit attributes and values from more general classes in which they are included.

In order to support property inheritance, objects must be organized into classes and classes must be arranged in a generalization hierarchy. Figure 4.5 shows some additional baseball knowledge inserted into a structure that is so arranged. Lines represent attributes. Boxed nodes represent objects and values of attributes of objects. These values can also be viewed as objects with attributes and values, and so on. The arrows on the lines point from an object to its value along the corresponding attribute line, The structure shown in the figure is a *slot-and-filler structure*. It may also be called a *semantic network* or a collection *of frames*. In the latter case, each individual frame represents the collection of attributes and values associated with a particular node. Figure 4.6 shows the node for baseball player displayed as a frame.

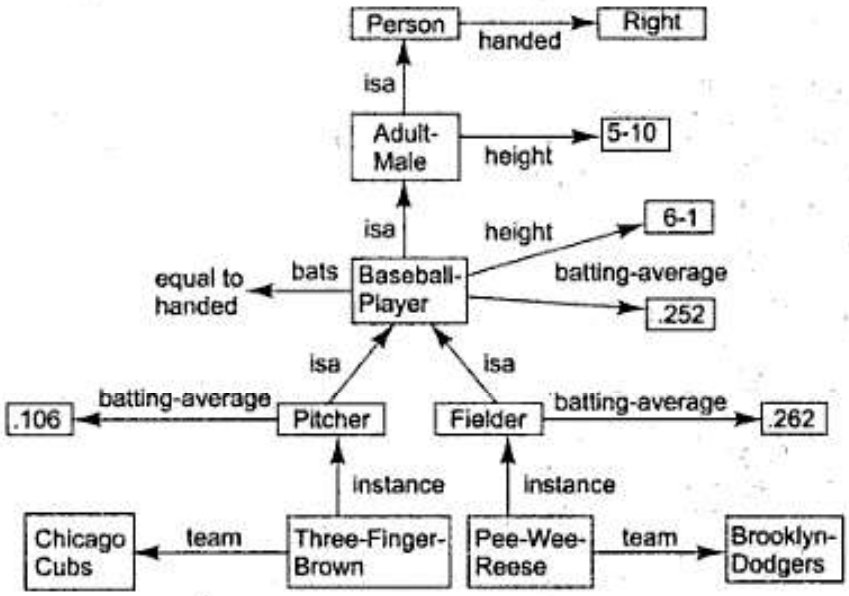

Figure 4.5 *Inheritable Knowledge*

Baseball-Player

isa:	*Adult-Male*
bats:	(EQUAL *handed)*
height:	6-1
batting-average:	.252

Fig. 4.6 *Viewing a Node as a Frame*

Do not be put off by the confusion in terminology here. There is so much flexibility in the way that this (and the other structures described in this section) can be used to solve particular representation problems that it is difficult to reserve precise words for particular representations. Usually the use of the term *frame system* implies somewhat more structure on the attributes and the inference mechanisms that are available to apply to them than does the term *semantic network.*

In Chapter 9 we discuss structures such as these in substantial detail. But to get an idea of how these structures support inference using the knowledge they contain, we discuss them briefly here. All of the objects and most of the attributes shown in this example have been chosen to correspond to the baseball domain, and they have no general significance. The two exceptions to this are the attribute *isa,* which is being used to show class inclusion, and the attribute *instance,* which is being used to show class membership. These two specific (and generally useful) attributes provide the basis for property inheritance as an inference technique. Using this technique, the knowledge base can support retrieval both of facts that have been explicitly stored and of facts that can be derived from those that are explicitly stored.

An idealized form of the property inheritance algorithm can be stated as follows:

Algorithm: Property Inheritance

To retrieve a value *V* for attribute A of an instance object *O:*

1. Find *O* in the knowledge base.
2. If there is a value there for the attribute A, report that value.
3. Otherwise, see if there is a value for the attribute *instance.* If not, then fail.
4. Otherwise, move to the node corresponding to that value and look for a value for the attribute *A.* If one is found, report it.
5. Otherwise, do until there is no value for the *isa* attribute or until an answer is found:
   (a) Get the value of the *isa* attribute and move to that node.
   (b) See if there is a value for the attribute *A.* If there is, report it.

This procedure is simplistic. It does not say what we should do if there is more than one value of the *instance* or *isa* attribute. But it does describe the basic mechanism of inheritance. We can apply this procedure to our example knowledge base to derive answers to the following queries:

- *team(Pee-Wee-Reese) = Brooklyn-Dodgers.* This attribute had a value stored explicitly in the knowledge base.
- *batting-average(Three-Finger Brown)* = .106. Since there is no value for batting average stored explicitly for Three Finger Brown, we follow the *instance* attribute to *Pitcher* and extract the value stored there. Now we observe one of the critical characteristics of property inheritance, namely that it may produce default values that are not guaranteed to be correct but that represent "best guesses" in the face of a lack of more precise information. In fact, in 1906, Brown's batting average was .204.
- *height(Pee-Wee-Reese)* = 6-1. This represents another default inference. Notice here that because we get to it first, the more specific fact about the height of baseball players overrides a more general fact about the height of adult males.