Knowledge Representation and Reasoning

Ronald J. Brachman
AT&T Labs – Research
Florham Park, New Jersey
USA 07974
rjb@research.att.com

Hector J. Levesque
Department of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 3H5
hector@cs.toronto.edu

Acknowledgments

Preface

Contents

Chapter 1

Introduction

Intelligence, as exhibited by people anyway, is surely one of the most complex and mysterious phenomena that we are aware of. One striking aspect of intelligent behaviour is that it is clearly conditioned by *knowledge*: for a very wide range of activities, we make decisions about what to do based on what we know (or believe) about the world, effortlessly and unconsciously. Using what we know in this way is so commonplace, that we only really pay attention to it when it is not there. When we say that someone behaved *unintelligently*, like when someone uses a lit match to see if there is any gas in a car's gas tank, what we usually mean is not that there is something that the person did not know, but rather that the person has failed to use what he or she did know. We might say: "You weren't thinking!" Indeed, it is *thinking* that is supposed to bring what is relevant in what we know to bear on what we are trying to do.

One definition of Artificial Intelligence (AI) is that it is the study of intelligent behaviour achieved through computational means. Knowledge Representation and Reasoning, then, is that part of AI that is concerned with how an agent uses what it knows in deciding what to do. It is the study of thinking as a computational process. This book is an introduction to that field and the ways that it has invented to create representations of knowledge, and computational processes that reason by manipulating these knowledge representation structures.

If this book is an introduction to the area, then this chapter is an introduction to the introduction. In it, we will try to address, if only briefly, some big questions that surround the deep and challenging topics of the field: what exactly do we mean by "knowledge," by "representation," and by "reasoning," and why do we think these concepts are useful for building AI systems? In the end, these are philosophical questions, and thorny ones at that; they bear considerable investigation by those with a more philosophical bent and can be the subject matter of whole careers. But the purpose of this chapter is not to cover in any detail what philosophers, logicians, and computer scientists have said about knowledge over the years; it is rather to glance at some of the main issues involved, and examine their bearings on Artificial Intelligence and the prospect of a machine that could think.

1.1 The key concepts: knowledge, representation, and reasoning

Knowledge What is knowledge? This is a question that has been discussed by philosophers since the ancient Greeks, and it is still not totally demystified. We certainly will not attempt to be done with it here. But to get a rough sense of what knowledge is supposed to be, it is useful to look at how we talk about it informally.

First, observe that when we say something like "John knows that . . . ," we fill in the blank with a simple declarative sentence. So we might say that "John knows that Mary will come to the party" or that "John knows that dinosaurs were warm-blooded." This suggests that, among other things, knowledge is a relation between a knower, like John, and a *proposition*, that is, the idea expressed by a simple declarative sentence, like "Mary will come to the party".

Part of the mystery surrounding knowledge is due to the nature of propositions. What can we say about them? As far as we are concerned, what matters about propositions is that they are abstract entities that can be *true* or *false*, right or wrong.[1] When we say "John knows that p," we can just as well say that "John knows that it is true that p." Either way, to say that John knows something is to say that John has formed a judgment of some sort, and has come to realize that the world is one way and not another. In talking about this judgment, we use propositions to classify the two cases.

A similar story can be told about a sentence like "John hopes that Mary will come to the party." The same proposition is involved, but the relationship John has to it is different. Verbs like "knows," "hopes," "regrets," "fears," and "doubts" all denote *propositional attitudes*, relationships between agents and propositions. In all cases, what matters about the proposition is its truth: if John hopes that Mary will come to the party, then John is hoping that the world is one way and not another,

[1]Strictly speaking, we might want to say that the *sentences* expressing the proposition are true or false, and that the propositions themselves are either factual or non-factual. Further, because of linguistic features such as indexicals (that is, words whose referents change with the context in which they are uttered, such as "me" and "yesterday"), we more accurately say that it is actual tokens of sentences or their uses in specific contexts that are true or false, not the sentences themselves.

as classified by the proposition.

Of course, there are sentences involving knowledge that do not explicitly mention a proposition. When we say "John knows who Mary is taking to the party," or "John knows how to get there," we can at least imagine the implicit propositions: "John knows that Mary is taking so-and-so to the party", or "John knows that to get to the party, you go two blocks past Main Street, turn left, . . . ," and so on. On the other hand, when we say that John has a skill as in "John knows how to play piano," or a deep understanding of someone or something as in "John knows Bill well," it is not so clear that any useful proposition is involved. While this is certainly challenging subject matter, we will have nothing further to say about this latter form of knowledge in this book.

A related notion that we are concerned with, however, is the concept of *belief*. The sentence "John believes that p" is clearly related to "John knows that p." We use the former when we do not wish to claim that John's judgment about the world is necessarily accurate or held for appropriate reasons. We sometimes use it when we feel that John might not be completely convinced. In fact, we have a full range of propositional attitudes, expressed by sentences like "John is absolutely certain that p," "John is confident that p," "John is of the opinion that p," "John suspects that p," and so on, that differ only in the level of conviction they attribute. For now, we will not distinguish amongst *any* of them. What matters is that they all share with knowledge a very basic idea: John takes the world to be one way and not another.

Representation The concept of representation is as philosophically vexing as that of knowledge. Very roughly speaking, representation is a relationship between two domains where the first is meant to "stand for" or take the place of the second. Usually, the first domain, the representor, is more concrete, immediate, or accessible in some way than the second. For example, a drawing of a milkshake and a hamburger on a sign might stand for a less immediately visible fast food restaurant; the drawing of a circle with a plus below it might stand for the much more abstract concept of womanhood; an elected legislator might stand for his or her constituency.

The type of representor that we will be most concerned with here is the formal *symbol*, that is, a character or group of them taken from some predetermined alphabet. The digit "7," for example, stands for the number 7, as does the group of letters "VII," and in other contexts, the words "*sept*" and "*shichi*." As with all representation, it is assumed to be easier to deal with symbols (recognize them, distinguish them from each other, display them, etc.) than with what the symbols represent. In some cases, a word like "John" might stand for something quite concrete; but many words, like "love" or "truth," stand for abstractions.

Of special concern to us is when a group of formal symbols stands for a proposition: "John loves Mary" stands for the proposition that John loves Mary. Again, the symbolic English sentence is fairly concrete: it has distinguishable parts involving the 3 words, for example, and a recognizable syntax. The proposition, on the other hand, is abstract: it is something like a classification of all the different ways we can imagine the world to be into two groups: those where John loves Mary, and those where he does not.

Knowledge Representation, then, is this: it is the field of study concerned with using formal symbols to represent a collection of propositions believed by some putative agent. As we will see, however, we do not want to insist that these symbols must represent *all* the propositions believed by the agent. There may very well be an infinite number of propositions believed, only a finite number of which are ever represented. It will be the role of *reasoning* to bridge the gap between what is represented and what is believed.

Reasoning So what is reasoning? In general, it is the formal manipulation of the symbols representing a collection of believed propositions to produce representations of new ones. It is here that we use the fact that symbols are more accessible than the propositions they represent: they must be concrete enough that we can manipulate them (move them around, take them apart, copy them, string them together) in such a way as to construct representations of new propositions.

The analogy here is with arithmetic. We can think of binary addition as being a certain formal manipulation: we start with symbols like "1011" and "10," for instance, and end up with "1101." The manipulation here is addition since the final symbol represents the sum of the numbers represented by the initial ones. Reasoning is similar: we might start with the sentences "John loves Mary" and "Mary is coming to the party," and after a certain amount of manipulation produce the sentence "Someone John loves is coming to the party." We would call this form of reasoning *logical inference* because the final sentence represents a logical conclusion of the propositions represented by the initial ones, as we will discuss below. According to this view (first put forward, incidentally, by the philosopher Leibniz in the 17th century), reasoning is a form of calculation, not unlike arithmetic, but over symbols standing for propositions rather than numbers.

1.2 Why knowledge representation and reasoning?

Why is knowledge even relevant at all to AI systems? The first answer that comes to mind is that it is sometimes useful to describe the behaviour of sufficiently complex systems (human or otherwise) using a vocabulary involving terms like "beliefs,"

"goals," "intentions," "hopes," and so on.

Imagine, for example, playing a game of chess against a complex chess-playing program. In looking at one of its moves, we might say to ourselves something like this: "It moved this way because it believed its queen was vulnerable, but still wanted to attack the rook." In terms of how the chess-playing program is actually constructed, we might have said something more like, "It moved this way because evaluation procedure P using static evaluation function Q returned a value of +7 after an alpha-beta minimax search to depth d." The problem is that this second description, although perhaps quite accurate, is at the wrong level of detail, and does not help us determine what chess move we should make in reponse. Much more useful is to understand the behaviour of the program in terms of the immediate goals being pursued, relative to its beliefs, long-term intentions, and so on. This is what the philosopher Dennett calls taking an *intentional stance* towards the chess-playing system.

This is not to say that an intentional stance is always appropriate. We might think of a thermostat, to take a classic example, as "knowing" that the room is too cold and "wanting" to warm it up. But this type of anthropomorphization is typically inappropriate: there is a perfectly workable electrical account of what is going on. Moreover, it can often be quite misleading to describe an AI system in intentional terms: using this kind of vocabulary, we could end up fooling ourselves into thinking we are dealing with something much more sophisticated than it actually is.

But there's a more basic question: is *this* what Knowledge Representation is all about? Is all the talk about knowledge just that—talk—a stance one may or may not choose to take towards a complex system?

To understand the answer, first observe that the intentional stance says nothing about what is or is not represented symbolically within a system. In the chess-playing program, the board position might be represented symbolically, say, but the goal of getting a knight out early, for instance, may not be. Such a goal might only emerge out of a complex interplay of many different aspects of the program, its evaluation functions, book move library, and so on. Yet, we may still choose to describe the system as "having" this goal, if this properly explains its behaviour.

So what role is played by a symbolic representation? The hypothesis underlying work in Knowledge Representation is that we will want to construct systems that contain symbolic representations with two important properties. First is that we (from the outside) can understand them as standing for propositions. Second is that the system is designed to behave the way that it does *because* of these symbolic representations. This is what Brian Smith has called the *Knowledge Representation Hypothesis*:

> Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantic attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge.

In other words, the Knowledge Representation Hypothesis implies that we will want to construct systems for which the intentional stance is grounded by design in symbolic representations. We will call such systems *knowledge-based systems* and the symbolic representations involved their *knowledge bases* (KB's).

1.2.1 Knowledge-based systems

To see what a knowledge-based system amounts to, it is helpful to look at two very simple Prolog programs with identical behaviour. Consider the first:

```
printColour(snow) :- !, write("It's white.").
printColour(grass) :- !, write("It's green.").
printColour(sky) :- !, write("It's yellow.").
printColour(X) :- write("Beats me.").
```

And here is an alternate:

```
printColour(X) :- colour(X,Y), !,
      write("It's "), write(Y), write(".").
printColour(X) :- write("Beats me.").

colour(snow,white).
colour(sky,yellow).
colour(X,Y) :- madeof(X,Z), colour(Z,Y).
madeof(grass,vegetation).
colour(vegetation,green).
```

Observe that both programs are able to print out the colour of various items (getting the sky wrong, as it turns out). Taking an intentional stance, both might be said to "know" that the colour of snow is white. The crucial point, however, is that only the second program is designed according to the Knowledge Representation Hypothesis.

Consider the clause `colour(snow,white)`, for example. This is a symbolic structure that we can understand as representing the proposition that snow is white, and moreover, we know, by virtue of knowing how the Prolog interpreter

works, that the system prints out the appropriate colour of snow precisely *because* it bumps into this clause at just the right time. Remove the clause and the system would no longer do so.

There is no such clause in the first program. The one that comes closest is the first clause of the program which says what to print when asked about snow. But we would be hard-pressed to say that this clause literally represents a belief, except perhaps a belief about what ought to be printed.

So what makes a system knowledge-based, as far as we are concerned, is not the use of a logical formalism (like Prolog), or the fact that it is complex enough to merit an intentional description involving knowledge, or the fact that what it believes is true; rather it is the presence of a KB, a collection of symbolic structures representing what it believes and reasons with during the operation of the system.

Much (but not all) of AI involves building systems that are knowledge-based in this way, that is, systems whose ability derives in part from reasoning over explicitly represented knowledge. So-called "expert systems" are a very clear case, but we also find them in the areas of language understanding, planning, diagnosis, and learning. Many AI systems are also knowledge-based to a somewhat lesser extent—some game-playing and high-level vision systems, for instance. And finally, some AI systems are not knowledge-based at all: speech, low-level vision, and motor control systems typically encode what they need to know directly into the programs themselves.

How much of intelligent behaviour needs to be knowledge-based in this sense? At this point, this remains an open research question. Perhaps the most serious challenge to the Knowledge Representation Hypothesis is the so-called "connectionist" methodology, which attempts to avoid any kind of symbolic representation and reasoning, and instead advocates computing with networks of weighted links between artificial "neurons."

1.2.2 Why knowledge representation?

So an obvious question arises when we start thinking about the two Prolog programs of the previous section: what advantage, if any, does the knowledge-based one have? Wouldn't it be better to "compile out" the KB and distribute this knowledge to the procedures that need it, as we did in the first program? The performance of the system would certainly be better. It can only slow a system down to have to look up facts in a KB and reason with them at runtime in order to decide what actions to take. Indeed advocates within AI of so-called "procedural knowledge" take pretty much this point of view.

When we think about the various skills we have, such as riding a bicycle or playing a piano, it certainly *feels* like we do not reason about the various actions to take (shifting our weight or moving our fingers); it seems much more like we just know what to do, and do it. In fact, if we try to think about what we are doing, we end up making a mess of it. Perhaps (the argument goes), this applies to most of our activities, making a meal, getting a job, staying alive, and so on.

Of course, when we first learn these skills, the case is not so clear: it seems like we need to think deliberately about what we are doing, even riding a bicycle. The philosopher Hubert Dreyfus first observed this paradox of "expert systems." These systems are claimed to be superior precisely because they are knowledge-based, that is, they reason over explicitly represented knowledge. But novices are the ones who think and reason, claims Dreyfus. Experts do not; they learn to recognize and to react. The difference between a chess master and a chess novice is that the novice needs to figure out what is happening and what to do, but the master just "sees" it. For this reason (among others), Dreyfus believes that the development of knowledge-based systems is completely wrong-headed, if it is attempting to duplicate human-level expertise.

So why even consider knowledge-based systems? Unfortunately, no definitive answer can yet be given. We suspect, however, that the answer will emerge in our desire to build systems that deal with a set of tasks that is *open-ended*. For any fixed set of tasks, it might work to "compile out" what the system needs to know; but if the set of tasks is not determined in advance, the strategy will not work. The ability to make behaviour depend on explicitly represented knowledge seems to pay off when we cannot specify in advance how that knowledge will be used.

The best example of this, perhaps, is what happens when we read a book. Suppose we are reading about South American geography. When we find out for the first time that approximately half of the population of Peru lives in the Andes, we are in no position to distribute this piece of knowledge to the various routines that might eventually require it. Instead, it seems pretty clear that we are able to assimilate the fact in declarative form for a very wide variety of potential uses. This is a prototypical case of a knowledge-based approach.

Further, from a system design point of view, the knowledge-based approach seems to have a number of desirable features:

- We can add new tasks and easily make them depend on previous knowledge. In our Prolog program example, we can add the task of enumerating all objects of a given color, or even of painting a picture, by making use of the KB to determine the colours.

- We can extend the existing behaviour by adding new beliefs. For example, by adding a clause saying that canaries are yellow, we automatically propagate this information to any routine that needs it.

- We can debug faulty behaviour by locating the erroneous beliefs of the system. In the Prolog example, by changing the clause for the colour of the sky, we automatically correct any routine that uses colour information.

- We can concisely explain and justify the behaviour of the system. Why did the program say that grass was green? It was because it believed that grass is a form of vegetation and that vegetation is green. We are justified in saying "because" here since if we removed either of the two relevant clauses, the behaviour would indeed change.

Overall, then, the hallmark of a knowledge-based system is that by design it has the ability to be *told* facts about its world and adjust its behaviour correspondingly.

This ability to have some of our actions depend on what we believe is what the cognitive scientist Zenon Pylyshyn has called *cognitive penetrability*. Consider, for example, responding to a fire alarm. The normal response is to get up and leave the building. But we would not do so if we happened to believe that the alarm was being tested, say. There are any number of ways we might come to this belief, but they all lead to the same effect. So our response to a fire alarm is cognitively penetrable since it is conditioned on what we can be made to believe. On the other hand, something like a blinking reflex as an object approaches your eye does not appear to be cognitively penetrable: even if you strongly believe the object will not touch you, you still blink.

1.2.3 Why reasoning?

To see the motivation behind reasoning in a knowledge-based system, it suffices to observe that we would like action to depend on what the system *believes* about the world, as opposed to just what the system has *explicitly represented*. In the Prolog example, there was no clause representing the belief that the colour of grass was green, but we still wanted the system to know this. In general, much of what we expect to put in a KB will involve quite general facts, which will then need to be applied to particular situations.

For example, we might represent the following two facts explicitly:

1. Patient x is allergic to medication m.

2. Anyone allergic to medication m is also allergic to medication m'.

In trying to decide if it is appropriate to prescribe medication m' for patient x, neither represented fact answers the question. Together, however, they paint a picture of a world where x is allergic to m', and this, together with other represented facts about allergies, might be sufficient to rule out the medication. So we do not want to condition behaviour only on the represented facts that we are able to *retrieve*, like in a database system. The beliefs of the system must go beyond these.

But beyond them to where? There is, as it turns out, a simple answer to this question, but one which, as we will discuss many times in subsequent chapters, is not always practical. The simple answer is that the system should believe p if, according to the beliefs it has represented, the world it is imagining is one where p is true. In the above example, facts (1) and (2) are both represented. If we now imagine what the world would be like if (1) and (2) were both true, then this is a world where

3. Patient x is allergic to medication m'

is also true, even though this fact is only implicitly represented.

This is the concept of *entailment*: we say that the propositions represented by a set of sentences S entail the proposition represented by a sentence p when the truth of p is implicit in the truth of the sentences in S. In other words, if the world is such that every element of S comes out true, then p does as well. All that we require to get some notion of entailment is a language with an account of what it means for a sentence to be true or false. As we argued, if our representation language is to represent knowledge at all, it must come with such an account (again, to know p is to take p to be true). So any knowledge representation language, whatever other features it may have, whatever syntactic form it may take, whatever reasoning procedures we may define over it, ought to have a well-defined notion of entailment.

The simple answer to what beliefs a knowledge-based system should exhibit, then, is that it should believe all and only the entailments of what it has explicitly represented. The job of reasoning, then, according to this account, is to compute the entailments of the KB.

What makes this account simplistic is that there are often quite good reasons not to calculate entailments. For one thing, it can be too *difficult* computationally to decide which sentences are entailed by the kind of KB we will want to use. Any procedure that always gives us answers in a reasonable amount of time will occasionally either miss some entailments or return too many answers. In the former case, the reasoning process is said to be *incomplete*; in the latter case, the reasoning is said to be *unsound.*

But there are also conceptual reasons why we might consider unsound or incomplete reasoning. For example, suppose p is not entailed by a KB, but is a reasonable guess, given what is represented. We might still want to believe that p is true. To use a classic example, suppose all I know about an individual Tweety is that she is a

bird. I might have a number of facts about birds in the KB, but likely none of them would *entail* that Tweety flies. After all, Tweety might turn out to be an ostrich. Nonetheless, it is a reasonable assumption that Tweety flies. This is unsound reasoning since we can imagine a world where everything in the KB is true but where Tweety does not fly.

Alternately, a knowledge-based system might come to believe a collection of facts from various sources which, taken together, cannot all be true. In this case, it would be inappropriate to do logically complete reasoning, since then *every* sentence would be believed: because there are no worlds where the KB is true, every sentence p will be trivially true in all worlds where the KB is true. An incomplete form of reasoning would clearly be more useful here until the contradictions were dealt with, if ever.

But despite all this, it remains the case that the simplistic answer is by far the best starting point for thinking about reasoning, even if we intend to diverge from it. So while it would be a mistake to *identify* reasoning in a knowledge-based system with logically sound and complete inference, it is the right place to begin.

1.3 The role of logic

The reason logic is relevant to knowledge representation and reasoning is simply that, at least according to one view, logic *is* the study of entailment relations—languages, truth conditions, and rules of inference. Not surprisingly, we will borrow heavily from the tools and techniques of formal symbolic logic. Specifically, we will use as our first knowledge representation language a very popular logical language, that of the predicate calculus, or as it sometimes called, the language of first-order logic (FOL). This language was invented by the philosopher Frege at the turn of the (Twentieth) century for the formalization of mathematical inference, but has been co-opted for knowledge representation purposes.

It must be stressed, however, that FOL itself is also just a starting point. We will have good reason in what follows to consider subsets and supersets of FOL, as well as knowledge representation languages quite different in form and meaning. Just as we are not committed to understanding reasoning as the computation of entailments, even when we do so, we are not committed to any particular language. Indeed, as we shall see, certain representation languages suggest forms of reasoning that go well beyond whatever connections they may have ever had with logic.

Where logic really does pay off from a knowledge representation perspective is at what Allen Newell has called the *knowledge level*. The idea is that we can understand a knowledge-based system at two different levels (at least). At the knowledge level, we ask questions concerning the representation language and its semantics. At the *symbol level*, on the other hand, we ask questions concerning the computational aspects. There are clearly issues of adequacy at each level. At the knowledge level, we deal with the expressive adequacy of a representation language and the characteristics of its entailment relation, including its computational complexity; at the symbol level, we ask questions about the computational architecture and the properties of the data structures and reasoning procedures, including their algorithmic complexity.

The tools of formal symbolic logic seem ideally suited for a knowledge level analysis of a knowledge-based system. In the next chapter, we begin such an analysis using the language of first-order logic, putting aside for now all computational concerns.

1.4 Bibliographic notes

1.5 Exercises

Chapter 2

The Language of First-Order Logic

Before any system aspiring to intelligence can even begin to reason, learn, plan, or explain its behaviour, it must be able to formulate the ideas involved. You will not be able to learn something about the world around you, for example, if it is beyond you to even express what that thing is. So we need to start with a *language* of some sort, in terms of which knowledge can be formulated. In this chapter, we will examine in detail one specific language that can be used for this purpose: the language of first-order logic, or FOL for short. FOL is not the only choice, but is merely a simple and convenient one to begin with.

2.1 Introduction

What does it mean to "have" a language? Once we have a set of words, or a set of symbols of some sort, what more is needed? As far as we are concerned, there are three things:

1. *syntax*: we need to specify which groups of symbols, arranged in what way, are to be considered properly formed. In English, for example, the string of words "the cat my mother loves" is a well-formed noun phrase, but "the my loves mother cat" is not. For knowledge representation, we need to be especially clear about which of these well-formed strings are the *sentences* of the language, since these are what express propositions.

2. *semantics*: we need to specify what the well-formed expressions are supposed to mean. Some well-formed expressions like "the hard-nosed decimal

holiday" might not mean anything. For sentences, we need to be clear about what idea about the world is being expressed. Without such an account, we cannot expect to say what believing one of them amounts to.

3. *pragmatics*: we need to specify how the meaningful expressions in the language are to be used. In English, for example, "There is someone right behind you" could be used as a warning to be careful in some contexts, and a request to move in others. For knowledge representation, this involves how we use the meaningful sentences of a representation language as part of a knowledge base from which inferences will be drawn.

These three aspects apply mainly to declarative languages, the sort we use to represent knowledge. Other languages will have other aspects not discussed here, for example, what the words sound like (for spoken languages), or what actions are being called for (for imperative languages).

We now turn our attention to the specification of FOL.

2.2 The syntax

In FOL, there are two sorts of symbols: the *logical* symbols, and the *non-logical* ones. Intuitively, the logical symbols are those those that have a fixed meaning or use in the language. There are three sorts of logical symbols:

1. punctuation: "(", ")", and ".".
2. connectives: "$\neg$", "$\wedge$", "$\vee$", "$\exists$", "$\forall$", and "=". Note the usual interpretation of these logical symbols: $\neg$ is logical negation, $\wedge$ is logical conjunction ("and"), $\vee$ is logical disjunction ("or"), $\exists$ means "there exists...," $\forall$ means "for all...", and = is logical equality. $\forall$ and $\exists$ are called "quantifiers."
3. variables: an infinite supply of symbols, which will denote here using x, y and z, sometimes with subscripts and superscripts.

The non-logical symbols are those that have an application-dependent meaning or use. In FOL, there are two sorts of non-logical symbols:

1. function symbols, an infinite supply of symbols, which we will denote using a, b, c, f, g, and h, with subscripts and superscripts.
2. predicate symbols: an infinite supply of symbols, which we will denote using P, Q and R, with subscripts and superscripts.

One distinguishing feature of non-logical symbols is that each one is assumed to have an *arity*, that is, a non-negative integer indicating how many "arguments" it takes. (This number is used in the syntax of the language below.) It is assumed that there is an infinite supply of function and predicate symbols of each arity. By convention, a, b, c are only used for function symbols of arity 0, which are called *constants*, and g and h are only used for function symbols of non-zero arity. Predicate symbols of arity 0 are sometimes called *propositional symbols*.

If you think of the logical symbols as the reserved keywords of a programming language, then non-logical symbols are like its identifiers. For example, we might have "Dog" as a predicate symbol of arity 1, "OlderThan" as a predicate symbol of arity 2, "bestFriend" as a function symbol of arity 1, and "johnSmith" as a constant. Note that we are treating "=" not as a predicate symbol, but as a logical connective (unlike the way that it is handled in some logic textbooks).

There are two types of legal syntactic expressions in FOL: *terms* and *formulas*. Intuitively, a term will be used to refer to something in the world, and a formula will be used to express a proposition. The set of terms of FOL is the least set satisfying these conditions:

- every variable is a term;
- if $t_1, \ldots, t_n$ are terms, and f is a function symbol of arity n, then $f(t_1, \ldots, t_n)$ is a term.

The set of formulas of FOL is the least set satisfying these constraints:

- if $t_1, \ldots, t_n$ are terms, and P is a predicate symbol of arity n, then $P(t_1, \ldots, t_n)$ is a formula;
- if t_1 and t_2 are terms, then $t_1 = t_2$ is a formula;
- if α and β are formulas, and x is variable, then $\neg\alpha$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, $\forall x.\alpha$, and $\exists x.\alpha$ are formulas.

Formulas of the first two types (containing no other simpler formulas) are called *atomic formulas* or *atoms*.

At this point, it is useful to introduce some notational abbreviations and conventions. First of all, we will add or omit matched parentheses and periods freely, and also use square and curly brackets to improve readability. In the case of predicates or function symbols of arity 0, we will usually omit the parentheses since there are no arguments to enclose.

By the *propositional subset* of FOL, we mean the language with no terms, no quantifiers, and where only propositional symbols are used. So, for example,

$$(P \wedge \neg(Q \vee R)),$$

where P, Q, and R are propositional symbols, would be a formula in this subset.

We also use the following abbreviations:

- $(\alpha \supset \beta)$ for $(\neg\alpha \vee \beta)$, and
- $(\alpha \equiv \beta)$ for $((\alpha \supset \beta) \wedge (\beta \supset \alpha))$

We also need to discuss the scope of quantifiers. We say that a variable appears *bound* in a formula if it is within the scope of a quantifier, and *free* otherwise. That is, x appears bound if it appears in a subformula $\forall x.\alpha$ or $\exists x.\alpha$ of the formula. So, for example, in a formula of the form

$$\forall y.P(x) \wedge \exists x[P(y) \vee Q(x)],$$

the first occurrence of the variable x is free, and the final two occurrences are bound. If x is a variable, t is a term, and α is a formula, we use the notation α^x_t to stand for the formula that results from replacing all free occurrences of x in α by t.

Finally, a *sentence* of FOL is any formula without free variables. The sentences of FOL are what we use to represent knowledge, and the rest is merely supporting syntactic machinery.

2.3 The semantics

As noted above, the concern of semantics is to explain what the expressions of a language mean. As far as we are concerned, this involves specifying what claim a sentence of FOL makes about the world, so that we can understand what believing it amounts to.

Unfortunately, there is a bit of a problem here. We cannot realistically expect to specify once and for all what a sentence of FOL means, for the simple reason that the non-logical symbols are used in an application-dependent way. I might use the constant “john” to mean one individual, and you might use it to mean another. So there’s no way we can possibly agree on what the sentence “Happy(john)” claims about the world, even if we were to agree on what “Happy” means.

But here is what we can agree to: the sentence “Happy(john)” claims that the individual named by “john” (whoever that might be) has the property named by

“Happy” (whatever that might be). In other words, we can agree once and for all on how the meaning of the sentence derives from the interpretation of the non-logical symbols involved. Of course, what we have in mind for these non-logical symbols can be quite complex and hard to make precise. For example, our list of non-logical symbols might include terms like

DemocraticCountry, IsABetterJudgeOfCharacterThan,
favouriteIceCreamFlavourOf, puddleOfwater27,

and the like. We should not (and cannot) expect the semantic specification of FOL to tell us precisely what terms like these mean. What we are after, then, is a clear specification of the meaning of sentences *as a function of the interpretation of the predicate and function symbols*.

To get to such a specification, we take the following (simplistic) view of what the world could be like:

> There are objects in the world.
>
> For any predicate P of arity 1, some of the objects will satisfy P and some will not. An *interpretation* of P settles the question, deciding for each object whether it has or does not have the property in question. (So borderline cases are ruled in separate interpretations: in one, it has the property; in another it does not.) Predicates of other arity are handled similarly. For example, an interpretation of a predicate of arity 3 decides on which triples of objects stand in the ternary relation. Similarly, a function symbol of arity 3 is interpreted as a mapping from triples of objects to objects.
>
> No other aspects of the world matter.

The assumption made in FOL is that this is all you need to say regarding the meaning of the non-logical symbols, and hence the meaning of all sentences.

For example, we might imagine that there are objects that include people, countries, and flavours of ice cream. The meaning of “DemocraticCountry” in some interpretation will be no more and no less than those objects that are countries that we consider to be democratic. We may disagree on which those are, of course, but then we are simply talking about different interpretations. Similarly, the meaning of “favouriteIceCreamFlavourOf” would be a specific mapping from people to flavours of ice cream (and from non-people to some other arbitrarily chosen object, say). Note that as far as FOL is concerned, we do not try to say what “DemocraticCountry” means the way a dictionary would, in terms of free elections, representative governments, majority rule, and so on; all we need to say is which objects are and are not democratic countries. This is clearly a simplifying assumption, and other languages would handle the terms differently.

2.3.1 Interpretations

Meanings are typically captured by specific interpretations, and we can now be precise about them. An *interpretation* $\Im$ in FOL is a pair $\langle\mathcal{D},\mathcal{I}\rangle$ where $\mathcal{D}$ is any non-empty set of objects called the *domain* of the interpretation, and $\mathcal{I}$ is a mapping called the *interpretation mapping* from the non-logical symbols to functions and relations over $\mathcal{D}$, as described below.

It is important to stress that an interpretation need not only involve mathematical objects. $\mathcal{D}$ can be *any* set, including people, garages, numbers, sentences, fairness, unicorns, chunks of peanut butter, situations, and the universe, among others things.

The interpretation mapping $\mathcal{I}$ will assign meaning to the predicate symbols as follows: to every predicate symbol P of arity n, $\mathcal{I}[P]$ is an n-ary relation over $\mathcal{D}$; that is,

$$\mathcal{I}[P] \subseteq [\mathcal{D} \times \cdots \times \mathcal{D}].$$

So for example, consider a unary predicate symbol Dog. Here, $\mathcal{I}$[Dog] would be some subset of $\mathcal{D}$, presumably the set of dogs in that interpretation. Similarly, $\mathcal{I}$[OlderThan] would be some subset of $[\mathcal{D} \times \mathcal{D}]$, presumably the set of pairs of objects in $\mathcal{D}$ where the first element of the pair is older than the second.

The interpretation mapping $\mathcal{I}$ will assign meaning to the function symbols as follows: to every function symbol f of arity n, $\mathcal{I}[f]$ is an n-ary function[1] over $\mathcal{D}$; that is,

$$\mathcal{I}[f] \in [\mathcal{D} \times \cdots \times \mathcal{D} \to \mathcal{D}].$$

So for example, $\mathcal{I}$[bestFriend] would be some function $[\mathcal{D} \to \mathcal{D}]$, presumably the function that maps a person to his or her best friend (and does something reasonable with non-persons). Similarly, $\mathcal{I}$[johnSmith] would be some element of $\mathcal{D}$, presumably somebody called John Smith.

It is sometimes useful to think of the interpretation of predicates in terms of their characteristic function. In this case, when P is a predicate of arity n, we view $\mathcal{I}[P]$ as an n-ary function to $\{0, 1\}$:

$$\mathcal{I}[P] \in [\mathcal{D} \times \cdots \times \mathcal{D} \to \{0, 1\}].$$

The relationship between the two specifications is that a tuple of objects is considered to be in the relation over $\mathcal{D}$ if and only if the characteristic function over those

[1]Here and subsequently, mathematical functions are taken to be total.

objects has value 1. This characteristic function also allows us to see more clearly how predicates of arity 0 (i.e., the propositional symbols) are handled. In this case, $\mathcal{I}[P]$ will be either 0 or 1. We can think of the first one as meaning "false" and the second "true." For the propositional subset of FOL, we can ignore $\mathcal{D}$ completely, and think of an interpretation as simply being a mapping $\mathcal{I}$ from the propositional symbols to either 0 or 1.

2.3.2 Denotation

Given an interpretation $\Im = \langle\mathcal{D},\mathcal{I}\rangle$, we can specify which elements of $\mathcal{D}$ are denoted by any variable-free term of FOL. For example, to find the object denoted by the term "bestFriend(johnSmith)" in $\Im$, we use $\mathcal{I}$ to get hold of the function denoted by "bestFriend", and then we apply that function to the element of $\mathcal{D}$ denoted by "johnSmith," producing some other element of $\mathcal{D}$. To deal with terms including variables, we also need to start with a *variable assignment over* $\mathcal{D}$, that is, a mapping from the variables of FOL to the elements of $\mathcal{D}$. So if μ is a variable assignment, $\mu[x]$ will be some element of the domain.

More formally, given an interpretation $\Im$ and variable assignment μ, the *denotation* of term t, written $||t||_{\Im,\mu}$, is defined by these rules:

1. if x is a variable, then $||x||_{\Im,\mu} = \mu[x]$;
2. if $t_1, \ldots, t_n$ are terms, and f is a function symbol of arity n, then

$$||f(t_1, \ldots, t_n)||_{\Im,\mu} = \mathcal{F}(d_1, \ldots, d_n)$$

where $\mathcal{F} = \mathcal{I}[f]$, and $d_i = ||t_i||_{\Im,\mu}$.

Observe that according to these recursive rules, $||t||_{\Im,\mu}$ is always an element of $\mathcal{D}$.

2.3.3 Satisfaction and models

Given an interpretation $\Im = \langle\mathcal{D},\mathcal{I}\rangle$, and the $||\cdot||_{\Im,\mu}$ relation defined above, we can now specify which sentences of FOL are true and which false according to this interpretation. For example, "Dog(bestFriend(johnSmith))" would be true in $\Im$ iff the following holds: we use $\mathcal{I}$ to get hold of the subset of $\mathcal{D}$ denoted by "Dog" and the object denoted by "bestFriend(johnSmith)", and then we say that the sentence is true when that object is in the set. To deal with formulas containing free variables, we again use a variable assignment, as above.

More formally, given an interpretation $\Im$ and variable assignment μ, we say that the formula α is *satisfied* in $\Im$, written $\Im, \mu \models \alpha$, according to these rules:

Assume that $t_1, \ldots, t_n$ are terms, P is a predicate of arity n, α and β are formulas, and x is a variable.

1. $\Im, \mu \models P(t_1, \ldots, t_n)$ iff $\langle d_1, \ldots, d_n \rangle \in \mathcal{P}$, where $\mathcal{P} = \mathcal{I}[P]$, and $d_i = \|t_i\|_{\Im,\mu}$;
2. $\Im, \mu \models t_1 = t_2$ iff d_1 and d_2 are the same elements of $\mathcal{D}$, where $d_i = \|t_i\|_{\Im,\mu}$;
3. $\Im, \mu \models \neg\alpha$ iff it is not the case that $\Im, \mu \models \alpha$;
4. $\Im, \mu \models (\alpha \wedge \beta)$ iff $\Im, \mu \models \alpha$ and $\Im, \mu \models \beta$;
5. $\Im, \mu \models (\alpha \vee \beta)$ iff $\Im, \mu \models \alpha$ or $\Im, \mu \models \beta$ (or both);
6. $\Im, \mu \models \exists x.\alpha$ iff $\Im, \mu' \models \alpha$, for some variable assignment μ' that differs from μ on at most x;
7. $\Im, \mu \models \forall x.\alpha$ iff $\Im, \mu' \models \alpha$, for every variable assignment μ' that differs from μ on at most x.

When the formula α is a sentence, it is easy to see that satisfaction does not depend on the given variable assignment (recall that sentences do not have free variables). In this case, we write $\Im \models \alpha$ and say that α *is true* in the interpretation $\Im$, or that α *is false* otherwise. In the case of the propositional subset of FOL, it is sometimes convenient to write $\mathcal{I}[\alpha] = 1$ or $\mathcal{I}[\alpha] = 0$ according to whether $\mathcal{I} \models \alpha$ or not. We will also use the notation $\Im \models S$, where S is a set of sentences, to mean that all of the sentences in S are true in $\Im$. We say in this case that $\Im$ is a logical *model* of S.

2.4 The pragmatics

The semantic rules of interpretation above tell us how to understand precisely the meaning of any term or formula of FOL in terms of a domain and an interpretation for the non-logical symbols over that domain. What is less clear, perhaps, is why anyone interested in Knowledge Representation should care about this. How are we supposed to use this language to represent knowledge? How is a knowledge-based system supposed to reason about concepts like "DemocraticCountry" or even "Dog" unless it is somehow given the intended interpretation to start with? And how could we possibly "give" a system an interpretation, which could involve (perhaps infinite) sets of honest-to-goodness objects like countries or animals?

2.4.1 Logical consequence

To answer these questions, we first turn to the notion of logical consequence. Observe that although the semantic rules of interpretation above depend on the interpretation of the non-logical symbols, there are connections among sentences of FOL that do not depend on the meaning of those symbols.

For example, let α and β be any two sentences of FOL, and let γ be the sentence $\neg(\beta \wedge \neg\alpha)$. Now suppose that $\Im$ is any interpretation where α is true. Then, by using the rules above, we can see that γ must be also true under this interpretation. This does not depend on how we understand any of the non-logical symbols in α or β. As long as α comes out true, γ will as well. In a sense, the truth of γ is implicit in the truth of α. We say in this case, that γ is a logical consequence of α.

More precisely, let S be a set of sentences, and α any sentence. We say that α is a *logical consequence* of S, or that S logically *entails* α, which we write $S \models \alpha$, iff for *every* interpretation $\Im$, if $\Im \models S$ then $\Im \models \alpha$. In other words, every model of S satisfies α. Yet another way of saying this is that there is no interpretation $\Im$ where $\Im \models S \cup \{\neg\alpha\}$. We say, in this case, that the set $S \cup \{\neg\alpha\}$ is *unsatisfiable*.

As a special case of this definition, we say that a sentence α is logically *valid*, which we write $\models \alpha$, when it is a logical consequence of the empty set. In other words, α is valid if and only if, for every interpretation $\Im$, we have that $\Im \models \alpha$ or, in still other words, iff the set $\{\neg\alpha\}$ is unsatisfiable.

It is not too hard to see that not only is validity a special case of entailment, but finite entailment is also a special case of validity. That is, if $S = \{\alpha_1, \ldots, \alpha_n\}$, then $S \models \alpha$ iff the sentence $[(\alpha_1 \wedge \cdots \wedge \alpha_n) \supset \alpha]$ is valid.

2.4.2 Why we care

Now let us re-examine the connection between knowledge-based systems and entailment, since this is at the root of Knowledge Representation.

A knowledge-based system will not and cannot have access to the user-intended interpretation of the non-logical symbols. As we noted, this could involve infinite sets of real objects quite outside the reach of any computer system. So a knowledge-based system will not be able to decide what to believe by using the rules above to evaluate the truth or falsity of sentences in this intended interpretation. Nor can it simply be "given" the set of sentences true in that interpretation as beliefs, since, among other things, there will be infinitely many such sentences.

However, suppose a set of sentences S entails a sentence α. Then we do know that whatever the intended interpretation is, if S happens to be true in that interpretation, then so must be α. If the user imagines the world satisfying S according to her understanding of the non-logical symbols, then it satisfies α as well. Other sentences not entailed may also be true, but a knowledge-based system can safely conclude that the entailed sentences definitely are.

By itself, this is a weak form of reasoning. It would allow a system that was told that "Dog(fido)" is true to conclude that "¬¬Dog(fido)" and that "Dog(fido) ∨ Happy(john)," are both true. But who cares? These are logically unassailable conclusions, of course, but not the sort of reasoning we would likely be interested in. In a sense, logical entailment gets us nowhere, since all we are doing is finding sentences that are already implied by the given one.

What we really want is a system that can go from "Dog(fido)" to other non-entailed facts that are true in the intended interpretation (where "Dog" is taken to mean the concept of a dog). For example, we want to be able to conclude "Mammal(fido)", and go from there to various properties like bearing its young alive, being an animal, and so on. This is no longer logical entailment because there are interpretations where "Dog(fido)" is true and "Mammal(fido)" is false. For example, let $\Im = \langle \mathcal{D}, \mathcal{I} \rangle$ be an interpretation where for some dog d, $\mathcal{D} = \{d\}$, for every predicate P other than "Dog", $\mathcal{I}[P] = \{\}$, where $\mathcal{I}[\text{Dog}] = \{d\}$, and where for every function symbol f, $\mathcal{I}[f](d, \ldots, d) = d$. This is an interpretation where the one and only dog is not a mammal. So the connection between the two sentences is not a strictly logical one.

The key idea of knowledge representation is this: to get the desired connection between dogs and mammals, we need to include within the set of sentences S a statement connecting the non-logical symbols involved. In this case, we add the sentence

$$\forall x.\text{Dog}(x) \supset \text{Mammal}(x)$$

to S. With this universal and "Dog(fido)" in S, we then get "Mammal(fido)" as a logical consequence. We will examine claims of logical consequence like this one in more detail later. But for now, note that by including this universal as one of the premises in S, we rule out interpretations like the one above where the set of dogs is not a subset of the set of mammals. If we then continue to add more and more sentences like this to S, we will rule out more and more unintended interpretations, and in the end, logical consequence itself will start to behave much more like "truth in the intended interpretation."

This, then, is the fundamental tenet of knowledge representation:

> Reasoning based on logical consequence alone is weak, and only allows safe, logically guaranteed conclusions to be drawn. However, by starting with a rich collection of sentences as given premises, including not only facts about particulars of the intended application, but those expressing connections among the non-logical symbols involved, the set of entailed conclusions becomes a much richer set, closer to the set of sentences true in the intended interpretation. Calculating these entailments thus becomes more like the form of reasoning we would expect of someone who understood the meaning of the terms involved.

In a sense, this is all there is to knowledge representation and reasoning; the rest is just details.

2.5 Explicit and implicit belief

The collection of sentences given as premises mentioned above is what we called a *knowledge base* or KB in the previous chapter: in our case, a finite set of sentences in the language of FOL. The role of a knowledge representation system, as discussed before, is to calculate entailments of this KB. We can think of the KB itself as the beliefs of the system that are *explicitly* given, and the entailments of that KB as the beliefs that are only *implicitly* given.

Just because we are imagining a "rich" collection of sentences in the KB, including the intended connections among the non-logical symbols, we should not be misled into thinking that we have done all the work, and that there is no real reasoning left to do. As we will see in an example below, it is often non-trivial to move from explicit to implicit beliefs.

2.5.1 An example

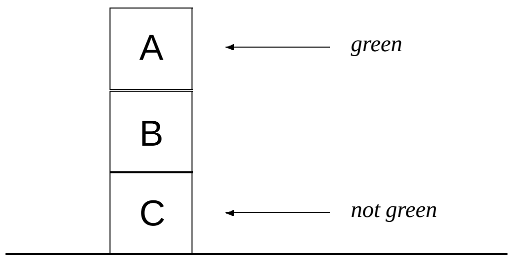
Consider the following example, illustrated in Figure 2.1. Suppose we have three coloured blocks stacked on a table, where the top one is green, the bottom one is not green, and the colour of the middle block is not known. The question to consider is whether there is a green block directly on top of a non-green one. The thing to observe about this question is that the answer (which happens to be *yes*) is not immediately obvious without some thinking.

We can formalize this problem in FOL, using a, b, and c, as the names of the blocks, and predicate symbols G and O to stand for "green" and "on". Then the facts we have in S are

$$\{O(a,b), O(b,c), G(a), \neg G(c)\}$$

and this is all we need. The claim we make here is that these four facts *entail* that there is indeed a green block on top of a non-green one, that is, that $S \models \alpha$, where

Figure 2.1: A stack of three blocks



α is

$$\exists x \exists y. G(x) \wedge \neg G(y) \wedge O(x, y).$$

To see this, we need to show that any interpretation that satisfies S also satisfies α. So let $\Im$ be any interpretation, and assume that $\Im \models S$. There are two cases to consider:

1. Suppose $\Im \models G(b)$. Then because $\neg G(c)$ and $O(b, c)$ are in S, we have that

   $$\Im \models G(b) \wedge \neg G(c) \wedge O(b, c).$$

   It follows from this that

   $$\Im \models \exists x \exists y. G(x) \wedge \neg G(y) \wedge O(x, y).$$

2. Suppose on the other hand that it is not the case that $\Im \models G(b)$. Then we have that $\Im \models \neg G(b)$, and because $G(a)$ and $O(a, b)$ are in S, we have that

   $$\Im \models G(a) \wedge \neg G(b) \wedge O(a, b).$$

   It follows from this that

   $$\Im \models \exists x \exists y. G(x) \wedge \neg G(y) \wedge O(x, y).$$

So either way, we have that $\Im \models \alpha$. Thus, α is a logical consequence of S.

Even though this is a very simple example, we can see that calculating what is implicit in a given collection of facts will sometimes involve subtle forms of reasoning. Indeed, it is well known that for FOL, the problem of determining whether

one sentence is a logical consequence of others is in general *unsolvable*: no automated procedure can decide validity, and so no automated procedure can tell us in all cases whether or not a sentence is entailed.

2.5.2 Knowledge-based systems

To recap, we imagine that for Knowledge Representation, we will start with a (large) KB representing what is explicitly known by a knowledge-based system. This KB could be the result of what the system is told, or perhaps what the system found out for itself through perception or learning. Our goal is to influence the behaviour of the overall system based on what is *implicit* in this KB, or as close as possible.

In general, this will require reasoning. By *deductive inference*, we mean the process of calculating the entailments of a KB, that is, given the KB, and any sentence α, determining whether or not KB $\models \alpha$.

We consider a reasoning process to be logically *sound* if whenever it produces α, then α is guaranteed to be a logical consequence. This rules out the possibility of producing plausible assumptions that may very well be true in the intended interpretation, but are not strictly entailed.

We consider a reasoning process to be logically *complete* if it is guaranteed to produce α whenever α is entailed. This rules out the possibility of missing some entailments, for example, when their status is too difficult to determine.

As we noted above, no automated reasoning process for FOL can be both sound and complete in general. However, the relative simplicity of FOL makes it a natural first step in the study of reasoning. The computational difficulty of FOL is one of the factors that will lead us to consider various other options in subsequent chapters.

2.6 Bibliographic notes

2.7 Exercises

Chapter 3

Expressing Knowledge

The stage is now set for a somewhat more detailed exploration of the process of creating a knowledge base (KB). Recall that knowledge involves taking the world to satisfy some property, as expressed by a declarative sentence. A KB will thus comprise a collection of such sentences, and we take the propositions expressed by these sentences to be beliefs of our putative agent.

Much of this book is an exploration of different languages that can be used to represent the knowledge of an agent in symbolic form, with different consequences, especially regarding reasoning. As we suggested in the previous chapter, first-order logic (FOL), while by no means the only language for representing knowledge, is a convenient choice for getting started with the KR enterprise.

3.1 Knowledge engineering

Having outlined the basic principles of knowledge representation and decided on an initial representation language, we might be tempted to dive right in and begin the implementation of a set of programs that could reason over a specific KB of interest. But before doing so, there are key questions about the knowledge of the agent that need to be considered in the abstract. In the same way that a programmer who is thinking ahead would first outline an *architecture* for her planned system, it is essential that we consider the overall architecture of the system we are about to create. We must think ahead to what it is we ultimately want (or want our artificial agent) to compute. We need to make some commitments to the reasons and times that inference will be necessary in our system's behavior. And finally, we need to stake out what is sometimes called an "ontology"—the kinds of *objects* that will be important to the agent and the *properties* those objects will be thought to have—before we can start populating our agent's KB. This general process, which addresses the KB at the knowledge level, is often called *knowledge engineering*.

This chapter, then, will be an introductory exercise in knowledge engineering, intended to be specific enough to make vivid the import of the previous two chapters. There are any number of example domains that we might use to illustrate how to use a KR language to build a KB. Here we pick a common and commonsensical world to illustrate the process, with people and places and relationships that are representative of many of the types of domains that AI systems will address. Given the complexity of human relations and the kind of behaviors that regular people have, we can think of this example domain as a "soap opera" world. Think of a small town in the midst of a number of scandals and contorted relationships. This little world will include people, places, companies, marriages (and divorces), crimes, death, 'hanky-panky,' and of course, money.

Our task is to create a KB that has appropriate entailments, and the first things we need to consider are what vocabulary to use and what facts to represent.

3.2 Vocabulary

In creating a KB, it is a good idea to start with the set of domain-dependent predicates and functions that provide the basis for the statement of facts about the KB's domain. What sorts of objects will there be in our soap-opera world?

The most obvious place to start is with the *named individuals* that are the actors in our human drama. In FOL, these would be represented by constant symbols, like maryJones, johnQSmith, *etc*. We might need to allow multiple identifiers that could ultimately be found to refer to the same individual: at some point in the process our system might know about a "john," without knowing whether he is johnQSmith or johnPJones, or even the former joannaSmith. Beyond the human players on our stage, we could of course have animals, robots, ghosts, and other sentient entities.

Another class of named individuals would be the legal entities that have their own identities, such as corporations (faultyInsuranceCompany), governments (evilvilleTownCouncil), and restaurants (theRackAndRollRestaurant). Key places need also be identified: tomsHouse, theAbandonedRailwayCar, norasJacuzzi, *etc*. Finally, other important objects need to be scoped out: earring35, butcherknife1, laurasMortgage (note that it is common to use the equivalent of numeric subscripts to distinguish among individuals that do not have uniquely referring names).

After capturing the set of individuals that will be central to the agent's world, it is next essential to circumscribe the basic *types* of objects that those individuals are. This is usually done with one-place predicates in FOL, such as Person(x).

Among the types of unary predicates we will want in our current domain we find Man, Woman, Place, Company, Jewelry, Knife, Contract, *etc*. If we expect to be reasoning about certain places based on what type of entities they are, such as a restaurant as a place to eat that is importantly different than someone's living room (for example), then object types like Restaurant, Bar, House, and SwimmingPool will be useful.

Another set of one-place predicates that is crucial for our domain representation is the set of *attributes* that our objects can have. So we need a vocabulary of properties that can hold of individuals, such as Rich, Beautiful, Unscrupulous, Bankrupt, ClosedForRepairs, Bloody, and Foreclosed. The syntax of FOL is limited in that it does not allow us to distinguish between such properties and the object-types we suggested a moment ago, such as Man and Knife. This usually does not present a problem, although if it were important for the system to distinguish between such types, the language could be extended to do so.[1]

The next key predicates to consider are n-ary predicates that express *relationships* (obviously of crucial interest in any soap-opera world). We can start with the obvious ones, like MarriedTo and DaughterOf, and related ones like LivesAt and HasCEO. We of course can branch out to more esoteric relationships like HairDresserOf, Blackmails, and HadAnAffairWith. And we cannot forget relationships of higher arity than 2, as in LoveTriangle, NamedInWillOf, ConspiresWith, and OccursInTimeInterval.

Finally, we need to capture the important *functions* of the domain. These can take more than one argument, but are most often unary, as in fatherOf, bestFriendOf, and ceoOf.

3.3 Basic facts

Now that we have our basic vocabulary in place, it is appropriate to start representing the simple core facts of our soap-opera world. Such facts are usually represented by atomic sentences and negations of atomic sentences. For example, we can use our type predicates, applied to individuals in the domain, to represent some basic truths: Man(john), Woman(jane), Company(faultyInsuranceCompany), Knife(butcherknife1). Such type predications would define the basic ontology of

[1]FOL does not distinguish because in our semantic account, as presented in the previous chapter, both sorts of predicates will be interpreted as sets of individuals of which the descriptions hold.

this world.[2]

Once we have set down the types of each of our objects, we can capture some of the properties of the objects. These properties will be the chief currency in talking about our domain, since we most often want to see what properties (and relationships) are implied by a set of facts or conjectures. In our sample domain, some useful property assertions might be Rich(john), ¬HappilyMarried(jim), WorksFor(jim,fic), Bloody(butcherknife1), and ClosedForRepairs(marTDiner).

Basic facts like the above yield what amounts to a simple database. These facts could indeed be stored in relational tables. For example, each type predicate could be a table with the table's entries being identifiers for all of the known satisfiers of that predicate. Of course, the details of such a storage strategy would be a symbol-level, not a knowledge-level issue.

Another set of simple facts that are useful in domain representation are those dealing with equality. To express the fact that John is the CEO of Faulty Insurance Company, we could use an equality and a one-place function: john = ceoOf(fic). Similarly, bestFriendOf(jim) = john would capture the fact that John is Jim's best friend. Another use of equalities would be for naming convenience, as when an individual has more than one name, *e.g.*, fic = faultyInsuranceCompany.

3.4 Complex facts

Many of the facts we would like to express about a domain are more complex than can be captured using atomic sentences. Thus we need to use more complex constructions, with quantifiers and other connectives, to express various beliefs about the domain.

In the soap-opera domain, we might want to express the fact that all the rich men in our world love Jane. To do so, we would use universal quantification, ranging over all of the rich individuals in our world, and over all of the men:

$$\forall y[\mathsf{Rich}(y) \wedge \mathsf{Man}(y) \supset \mathsf{Loves}(y, \mathsf{jane})].$$

Note that "rich man" here is captured by a conjunction of predicates. Similarly, we might want to express the fact that in this world, all the women with the possible exception of Jane love John. To do so, we would use a universal ranging over all of the women, and negate an equality to exclude Jane:

$$\forall y[\mathsf{Woman}(y) \wedge y \neq \mathsf{jane} \supset \mathsf{Loves}(y, \mathsf{john})].$$

[2]Note, by the way, that suggestive names are not a form of knowledge representation since they do not support logical inference. Just using "butcherknife1" as a symbol does not give the system any substantive information about the object. This is done using predicates, not orthography.

Universals are also useful for expressing very general facts, not even involving any known individuals. For example,

$$\forall x \forall y[\mathsf{Loves}(x, y) \supset \neg \mathsf{Blackmails}(x, y)]$$

expresses the fact that no one who loves someone will blackmail the one he or she loves.

Note that the universal quantifications above could each be expressed without quantifiers, if all of the individuals in the soap-opera world were enumerated. It would be tedious if the world were at all large, so the universally quantified sentences are handy abbreviations. Further, as new individuals are born or otherwise introduced into our soap-opera world, the universals will cover them as well.

Another type of fact that needs a complex sentence to express it is one that expresses *incomplete knowledge* about our world. For example, if we know that Jane loves one of John or Jim, but not which, we would need to use a disjunction to capture that belief:

$$\mathsf{Loves}(\mathsf{jane}, \mathsf{john}) \vee \mathsf{Loves}(\mathsf{jane}, \mathsf{jim}).$$

Similarly, if we knew that someone (an adult) was blackmailing John, but not who it was, we would use an existential quantifier to posit that unknown person:

$$\exists x[\mathsf{Adult}(x) \wedge \mathsf{Blackmails}(x, \mathsf{john})].$$

This kind of fact would be quite prevalent in a soap-opera world story, although one would expect many such unknowns to be resolved over time.

In contrast to the prior use of universals, the above cases of incomplete knowledge are not merely abbreviations. We cannot write a more complete version of the information in another form—it just isn't known.

Another useful type of complex statement about our soap-opera domain is what we might call a *closure* sentence, used to limit the domain of discourse. So, for example, we could enumerate if necessary all of the people in our world:

$$\forall x[\mathsf{Person}(x) \supset x = \mathsf{jane} \vee x = \mathsf{john} \vee x = \mathsf{jim} \vee \ldots].$$

In a similar fashion, we could circumscribe the set of all married couples:

$$\forall x \forall y[\mathsf{MarriedTo}(x, y) \supset (x = \mathsf{ethel} \wedge y = \mathsf{fred}) \vee \ldots].$$

Anyone not mentioned in this closure sentence could then be assumed to be unmarried. In an even more general way, we can carve out the full set of individuals in the domain of discourse:

$$\forall x[x = \mathsf{fic} \vee x = \mathsf{jane} \vee x = \mathsf{jim} \vee x = \mathsf{marTDiner} \vee \ldots].$$

This ensures that a reasoner would not postulate a new, hitherto unknown object in the course of its reasoning.

It might also be useful to distinguish formally between all known individuals, with a set of sentences like jane $\neq$ john. This would prevent the accidental postulation that two people were the same, for example, in trying to solve a crime.

3.5 Terminological facts

The kinds of facts we have represented so far are sufficient to capture the basic circumstances in a domain, and give us enough grist for the reasoning mill. However, when thinking about domains like the soap-opera world, we would typically also think in terms of relationships among the predicate and function symbols we have exploited above. For example, we would consider it quite "obvious" in this domain that if it were asserted that john were a Man, then we should answer "no" to the query, Woman(john). Or we would easily accede to the fact that MarriedTo(jr,sueEllen) was true if it were already stated that MarriedTo(sueEllen,jr) was. But there is nothing in our current KB that would actually sanction such inferences. In order to support such common and useful inferences, we need to provide a set of facts about the *terminology* we are using.

Terminological facts come in many varieties. Here we look at a sample:

- *Disjointness:* often two predicates are disjoint, and the assertion of one implies the negation of the other, as in

$$\forall x[\mathsf{Man}(x) \supset \neg\mathsf{Woman}(x)]$$

- *Subtypes:* there are many predicates that imply a form of specialization, wherein one type is subsumed by another. For example, since a Senator is a kind of legislator, we would want to capture the subtype relationship:

$$\forall x[\mathsf{Senator}(x) \supset \mathsf{Legislator}(x)]$$

  This way, we should be able to infer the reasonable consequence that anything true of legislators is true of Senators (but not *vice versa*).

- *Exhaustiveness:* this is the converse of the subtype assertion, where two or more subtypes completely account for a supertype, as in

$$\forall x[\mathsf{Adult}(x) \supset (\mathsf{Man}(x) \vee \mathsf{Woman}(x))]$$

- *Symmetry:* as in the case of the MarriedTo predicate, some relationships are symmetric:

$$\forall x, y[\mathsf{MarriedTo}(x, y) \supset \mathsf{MarriedTo}(y, x)]$$

- *Inverses:* some relationships are the opposite of others:

$$\forall x, y[\mathsf{ChildOf}(x, y) \supset \mathsf{ParentOf}(y, x)]$$

- *Type restrictions:* part of the meaning of some predicates is the fact that their arguments must be of certain types. For example, we might want to capture the fact that the definition of marriage entails that the partners are persons and (in most places) of opposite genders:

$$\forall x, y[\mathsf{MarriedTo}(x, y) \supset \mathsf{Person}(x) \wedge \mathsf{Person}(y) \wedge \mathsf{OppositeSex}(x, y)]$$

- *Full definitions:* in some cases, we want to create compound predicates that are completely defined by a logical combination of other predicates. We can use a biconditional to capture such definitions:

$$\forall x[\mathsf{RichMan}(x) \equiv \mathsf{Rich}(x) \wedge \mathsf{Man}(x)]$$

As can be seen from these examples, terminological facts are typically captured in a logical language as universally quantified conditionals or biconditionals.

3.6 Entailments

Now that we have captured the basic structure of our soap-opera domain, it is time to turn to the reason that we have done this representation in the first place: deriving implicit conclusions from our explicitly represented KB. Here we briefly explore this in an intuitive fashion. This will give us a feel for the consequences of a particular characterization of a domain. In the next chapter, we will consider how entailments can be computed in a more mechanical way.

Let us consider all of the basic and complex facts proposed so far in this chapter to be a knowledge base, called KB. Besides asking simple questions of KB like, "is John married to Jane?", we will want to explore more complex and important ones, such as, "is there a company whose CEO loves Jane?" Such a question would look like this in FOL:

$$\exists x[\mathsf{Company}(x) \wedge \mathsf{Loves}(\mathsf{ceoOf}(x), \mathsf{jane})]?$$

What we want to do is find out if the truth of this sentence is implicit in what we already know. In other words, we want to see if the sentence is entailed by KB.

To answer the question, we need to determine whether every logical interpretation that satisfies KB also satisfies the sentence. So let us imagine an interpretation $\Im$, and suppose that $\Im \models$ KB. It follows then that $\Im$ satisfies $\mathsf{Rich(john)}$, $\mathsf{Man(john)}$, and $\forall y[\mathsf{Rich}(y) \land \mathsf{Man}(y) \supset \mathsf{Loves}(y, \mathsf{jane})]$, since these are all in KB. As a result, $\Im \models \mathsf{Loves(john,jane)}$. Now since $(\mathsf{john} = \mathsf{ceoOf(fic)})$ is also in KB, we get that

$$\Im \models \mathsf{Loves(ceoOf(fic), jane)}.$$

Finally, since

$$\mathsf{Company(faultyInsuranceCompany)}$$

and

$$(\mathsf{fic} = \mathsf{faultyInsuranceCompany})$$

are both in KB, we have that

$$\Im \models \mathsf{Company(fic)} \land \mathsf{Loves(ceoOf(fic), jane)},$$

from which it follows that

$$\Im \models \exists x[\mathsf{Company}(x) \land \mathsf{Loves}(\mathsf{ceoOf}(x), \mathsf{jane})].$$

Since this argument goes through for any interpretation $\Im$, we know that the sentence is indeed entailed by KB.

Observe that by looking at the argument we have made, we can determine not only that there is a company whose CEO loves Jane, but also what that company is. In many applications, we will be interested in finding out not only whether something is true or not, but also which individuals satisfy a property of interest. In other words, we need answers not only to yes-no questions, but to wh-questions as well (who? what? where? when? how? why?).[3]

Let us consider a second example, which involves a hypothetical. Consider the question, "If no man is blackmailing John, then is he being blackmailed by someone he loves?" In logical terms, this question would be formulated this way:

$$\begin{array}{l}\forall x[\mathsf{Man}(x) \supset \neg\mathsf{Blackmails}(x, \mathsf{john})] \ \supset \\ \exists y[\mathsf{Loves}(\mathsf{john}, y) \land \mathsf{Blackmails}(y, \mathsf{john})]?\end{array}$$

[3]In the next chapter we will propose a general mechanism for extracting answers from existential questions like the above.

Again we need to determine whether or not the sentence is entailed by KB. Here we use the easily verified fact that KB $\models (\alpha \supset \beta)$ *iff* KB $\cup \{\alpha\} \models \beta$. So let us imagine that we have an interpretation $\Im$ such that $\Im \models$ KB, and that

$$\Im \models \forall x[\mathsf{Man}(x) \supset \neg\mathsf{Blackmails}(x, \mathsf{john})].$$

We must show that we then have it that

$$\Im \models \exists y[\mathsf{Loves}(\mathsf{john}, y) \land \mathsf{Blackmails}(y, \mathsf{john})].$$

To get to this conclusion, there are a number of steps. First of all, we know that someone is blackmailing John,

$$\Im \models \exists x[\mathsf{Adult}(x) \land \mathsf{Blackmails}(x, \mathsf{john})],$$

since this fact is in KB. Also, we have in KB that adults are either men or women,

$$\Im \models \forall x[\mathsf{Adult}(x) \supset (\mathsf{Man}(x) \lor \mathsf{Woman}(x))],$$

and since by hypothesis no man is blackmailing John, we derive the fact that a woman is blackmailing him:

$$\Im \models \exists x[\mathsf{Woman}(x) \land \mathsf{Blackmails}(x, \mathsf{john})].$$

Next, as seen in the previous example, we have it that

$$\Im \models \mathsf{Loves(john,jane)}.$$

So, we have it that some woman is blackmailing John and that John loves Jane. Could she be the blackmailer? Recall that all the women except possibly Jane love John,

$$\Im \models \forall y[\mathsf{Woman}(y) \land y \neq \mathsf{jane} \supset \mathsf{Loves}(y, \mathsf{john})],$$

and that no one who loves someone will blackmail them,

$$\Im \models \forall x \forall y[\mathsf{Loves}(x, y) \supset \neg\mathsf{Blackmails}(x, y)].$$

We can put these two conditionals together and conclude that no woman other than Jane is blackmailing John:

$$\Im \models \forall y[\mathsf{Woman}(y) \land y \neq \mathsf{jane} \supset \neg\mathsf{Blackmails}(y, \mathsf{john})].$$

Since we know that a woman is in fact blackmailing John, we are forced to conclude that it is Jane:

$$\Im \models \mathsf{Blackmails(jane, john)}.$$

Thus, in the end, we have concluded that John loves Jane and she is blackmailing him,

$$\Im \models [\text{Loves(john, jane)} \land \text{Blackmails(jane, john)}],$$

and so

$$\Im \models \exists y[\text{Loves(john}, y) \land \text{Blackmails}(y, \text{john})],$$

as desired.

Here we have illustrated in intuitive form how a *proof* can be thought of as a sequence of FOL sentences, starting with those known to be true in the KB (or surmised as part of the assumptions dictated by the query), that proceeds logically using other facts in the KB and the rules of logic, until a suitable conclusion is reached. In the next chapter, we will examine a different style of proof based on negating the desired conclusion, and showing that this leads to a contradiction.

To conclude this section, let us consider what is involved with an entailment question when the answer is *no*. In the previous example, we made the assumption that no man was blackmailing John. Now let us consider if this was necessary: is it already implicit in what we have in the KB that someone John loves is blackmailing him? In other words, we wish to determine whether or not KB entails

$$\exists y[\text{Loves(john}, y) \land \text{Blackmails}(y, \text{john})].$$

To show that it does *not*, we must show an interpretation that satisfies KB but falsifies the above sentence. That is, we must produce a specific interpretation $\Im = \langle \mathcal{D}, \mathcal{I} \rangle$, and argue that it satisfies every sentence in the KB, as well as the negation of the above sentence. For the number of sentences we have in KB, this is a big job since all of them must be verified, but the essence of the argument is that without contradicting anything already in KB, we can arrange $\Im$ in such a way that John only loves women, and that there is only one person in $\mathcal{D}$ who is blackmailing John, and it is a man. Thus it is not already implicit in KB that someone John loves is blackmailing him.

3.7 Abstract individuals

The FOL language gives us the basic tools for representing facts in a domain, but in many cases, there is a great deal of flexibility that can be exercised in mapping objects in that domain onto predicates and functions. There is also considerable flexibility in what we consider to be the individuals in the domain. In this section, we will see that it is sometimes useful to introduce new abstract individuals that might not have been considered in a first analysis. This is called *reification* and

is typical, as we shall see in later chapters, of systems like description logics and frame languages.

To see why reification might be useful, consider how we might say that John purchased a bike:

Purchases(john,bike) *vs.*
Purchases(john,sears,bike) *vs.*
Purchases(john,sears,bike,feb14) *vs.*
Purchases(john,sears,bike,feb14,$200) *vs.* ...

The problem here is that it seems that the arity of the Purchases predicate depends on how much detail we will want to express, which we may not be able to predict in advance.

A better approach is to take the purchase itself to be an abstract individual, call it p23. To describe this purchase at any level of detail we find appropriate, we need only use 1-place predicates and functions:

$$\begin{aligned}&\text{Purchase(p23)} \land \text{agent(p23)} = \text{john} \land \text{object(p23)} = \text{bike}\\ &\quad \land \text{source(p23)} = \text{sears} \land \text{amount(p23)} = \text{\$200} \land \ldots\end{aligned}$$

For less detail, we simply leave out some of the conjuncts; for more, we include others. The big advantage is that the arity of the predicate and function symbols involved can be determined in advance.

In a similar way we can capture in a reasonable fashion complex relationships of the sort that are common in our soap-opera world. For example, we might initially consider representing marriage relationships this way:

$$\text{MarriedTo}(x, y)$$

but we might also need to consider

$$\text{PreviouslyMarriedTo}(x, y)$$

and

$$\text{ReMarriedTo}(x, y).$$

Rather than create a potentially endless supply of marriage and remarriage (and divorce and annulment and ...) predicates, we can reify marriages and divorces as abstract individuals, and determine anyone's current marital status and complete marital history directly from them:

$$\begin{aligned}&\text{Marriage(m17)} \land \text{husband(m17)} = x \land \text{wife(m17)} = y\\ &\quad \land \text{date(m17)} = \ldots \land \text{witness(m17)} = \ldots \land \ldots\end{aligned}$$

It is now possible to *define* the above predicates (PreviouslyMarriedTo, *etc.*) in terms of the existence (and chronological order) of appropriate marriage and divorce events.

In representing commonsense information like the above, we also find that we need individuals for numbers, dates, times, addresses, *etc*. Basically, any "object" about which we can ask a *wh*-question should have an individual standing for it in the KB, so it can be returned as the result of a query.

The idea of reifying abstract individuals leads to some interesting choices concerning the representation of *quantities*. For example, an obvious representation for ages would be something like this:

$$\mathsf{ageInYears(suzy)} = 14.$$

If a finer-grained notion of age is needed in an application, we might prefer to represent a person's age in months (this is particularly common when talking about young children):

$$\mathsf{ageInMonths(suzy)} = 172.$$[4]

Of course, there is a relationship between ageInYears and ageInMonths. However, we have exactly the same relationship between quantities like durationInYears and durationInMonths, and between expectedLifeInYears and expectedLifeInMonths.

To capture all these regularities, it might be better to introduce an abstract individual to stand for a time duration, independent of any units. So we might take age(suzy) to denote an abstract quantity of time, quite apart from Suzy and 14, and assert that

$$\mathsf{years(age(suzy))} = 14$$

as a way of saying what this quantity would be if measured in years. Now we can write very general facts about such quantities such as

$$\mathsf{months}(x) = 12 * \mathsf{years}(x)$$

to relate the two units of measurement. Similarly, we would have

$$\mathsf{centimeters}(x) = 100 * \mathsf{meters}(x).$$

We could continue in this vein with locations and times. For example, instead of

$$\mathsf{time(m17)} = \text{“Jan 5 1992 4:47:03EST”}$$

[4]For some purposes a more qualitative view of age might be in order, as in age(suzy)=teenager, or age(suzy)=minor.

where we are forced to decide on a fixed granularity, we could use

$$\mathsf{time(m17)} = \mathsf{t41} \wedge \mathsf{year(t41)} = 1992 \wedge \mathsf{month(t41)} = \mathsf{Jan} \wedge \ldots$$

where we have reified time points. This type of representation of abstract individuals for quantities, times, locations, *etc*., is a common technique similar to the reification of events illustrated above.

3.8 Other sorts of facts

With the apparatus described so far, we have seen how to represent the basic facts and individuals of a commonsense domain like our soap-opera world. Before moving on to a look at the variations in different knowledge representation systems and their associated inference machinery, it is important to point out that there are a number of other types of facts about domains that we may want to capture. Each of these is problematical for a straightforward application of first-order logic, but as we shall see in the remainder of the book, they may be represented with extensions of FOL or with other KR languages. The choice of the language to use in a system or analysis will ultimately depend on what types of facts and conclusions are most important for the application.

Among the many types of facts in the soap-opera world that we have not captured are

- *statistical and probabilistic facts*. These include those that involve portions of the sets of individuals satisfying a predicate, in some cases exact subsets and in other cases less exactly quantifiable:
  - Half of the companies are located on the East Side.
  - Most of the employees are restless.
  - Almost none of the employees are completely trustworthy.
- *default and prototypical facts*. These cite characteristics that are usually true, or reasonable to assume true unless told otherwise:
  - Company presidents typically have secretaries intercepting their phone calls.
  - Cars have four wheels.
  - Companies generally do not allow employees that work together to be married.

  - Birds fly.

- *intentional facts*. These express people's mental attitudes and intentions. That is, they can reflect the reality of people's beliefs but not necessarily the "real" world itself:
  - John believes that Henry is trying to blackmail him.
  - Jane does not want Jim to know that she loves him.
  - Tom wants Frank to believe that the shot came from the grassy knoll.

This is not the end of what we would like to be able to express in a KB, of course. In later chapters, we will want to talk about the effects of actions and will end up reifying both actions and states of the world. Ultimately, a knowledge-based system should be able to express and reason with anything that can be expressed by a sentence of English, indeed anything that we can imagine as being either true or false. Here we have only looked at simple forms that are easily expressible in FOL. In subsequent chapters, we will examine other representation languages with different strengths and weaknesses. First, however, we turn to how me might compute entailments of a KB in FOL.

3.9 Bibliographic notes

3.10 Exercises

Chapter 4

Resolution

In the previous chapter, we examined how FOL could be used to represent knowledge about a simple application domain. We also showed how logical reasoning could be used to discover facts that were only implicit in a given knowledge base. All of our deductive reasoning, however, was done by hand, and relatively informally. In this chapter, we will examine in detail how to automate a deductive reasoning procedure.

At the knowledge level, the specification for an idealized deductive procedure is clear: given a knowledge base KB, and a sentence α, we would like a procedure that can determine whether or not KB $\models \alpha$; also, if $\beta[x_1, \ldots, x_n]$ is a formula with free variables among the x_i, we want a procedure that can find ground terms t_i, if they exist, such that KB $\models \beta[t_1, \ldots, t_n]$. Of course, as we discussed in Chapter 1, this is idealized; *no* computational procedure can fully satisfy this specification. What we are really after, in the end, is a procedure that does deductive reasoning in as sound and complete a manner as possible, and in a language as close as possible to that of full FOL.

One observation about this specification is that if we take the KB to be a finite set of sentences $\{\alpha_1, \ldots, \alpha_n\}$, then there are several equivalent ways of formulating the deductive reasoning task:

$$\begin{array}{ll} & \text{KB} \models \alpha \\ \text{iff} & \models [(\alpha_1 \wedge \cdots \wedge \alpha_n) \supset \alpha] \\ \text{iff} & \text{KB} \cup \{\neg\alpha\} \text{ is not satisfiable} \\ \text{iff} & \text{KB} \cup \{\neg\alpha\} \models \neg\text{TRUE} \end{array}$$

where TRUE is any valid sentence, such as $\forall x(x = x)$. What this means is that if we have a procedure for testing the validity of sentences, or for testing the satisfiability of sentences, or for determining whether or not ¬TRUE is entailed, then that procedure can also be used to find the entailments of a finite KB. This is significant since the Resolution procedure which we will consider in this chapter is in fact a procedure for determining whether certain sets of formulas are satisfiable.

In the next section, we begin by looking at a propositional version of Resolution, the clausal representation it depends on, and how it can be used to compute entailments. In Section 4.2, we generalize this account to deal with variables and quantifiers, and show how answer predicates can be used to find bindings for variables in queries. Finally, in Section 4.3, we review the computational difficulties inherent in Resolution, and show some of the refinements to Resolution that are used in practice to deal with them.

4.1 The propositional case

The reasoning procedure we will consider in this chapter works on logical formulas in a special restricted form. It is not hard to see that every formula α of propositional logic can be converted into another formula α' such that $\models (\alpha \equiv \alpha')$, and where α' is a conjunction of disjunctions of literals, where a *literal* is either an atom or its negation. We say that α and α' are *logically equivalent*, and that α' is in *conjunctive normal form*, or CNF. In the propositional case, CNF formulas look like this:

$$(p \vee \neg q) \;\wedge\; (q \vee r \vee \neg s \vee p) \;\wedge\; (\neg r \vee q)$$

The procedure to do the conversion to CNF from any propositional formula is as follows:

1. eliminate $\supset$ and $\equiv$, using the fact that these are abbreviations for formulas using only $\wedge$, $\vee$ and $\neg$;
2. move $\neg$ inwards so that it appears only in front of an atom, using for example, the fact that $\neg(\alpha \wedge \beta)$ is equivalent to $(\neg\alpha \vee \neg\beta)$;
3. distribute $\wedge$ over $\vee$, using for example, the fact that $((\alpha \wedge \beta) \vee \gamma)$ is equivalent to $((\alpha \vee \gamma) \wedge (\beta \vee \gamma))$;
4. collect terms, using for example, the fact that $(\alpha \vee \alpha)$ is equivalent to α.

The end result of this procedure is a logically equivalent CNF formula.[1] For example, for $((p \supset q) \supset r)$, by applying rule (1) above, we get $(\neg(\neg p \vee q) \vee r)$; applying

[1] An analogous procedure also exists to convert a formula into a disjunction of conjunction of literals, which is called *disjunctive normal form*, or DNF.

rule (2), we get $((p \wedge \neg q) \vee r)$; and with rule (3), we get $((p \vee r) \wedge (\neg q \vee r))$, which is in CNF. In this chapter, we will mainly deal with formulas in this CNF.

It is convenient to use a shorthand representation for CNF. A *clausal formula* is a finite set of clauses, where a *clause* is a finite set of literals. The interpretation of clausal formulas is precisely as formulas in CNF: a clausal formula is understood as the conjunction of its clauses, where each clause is understood as the disjunction of its literals, and literals are understood normally. In representing clauses here, we will use the following notation:

- if ρ is a literal then $\overline{\rho}$ is its *complement*, defined by $\overline{p} = \neg p$ and $\overline{\neg p} = p$, for any atom p;
- to distinguish clauses from clausal formulas, we will use "[" and "]" as delimiters for clauses, but "{" and "}" for formulas.

For example, $[p, \neg q, r]$ is the clause consisting of three literals, and understood as the disjunction of the three literals, while $\{[p, \neg q, r], [q]\}$ is the clausal formula consisting of two clauses, and understood as their conjunction. A clause like $[\neg p]$ with a single literal is called a *unit clause*.

Note that the empty clausal formula $\{\}$ is not the same as $\{[]\}$, the formula containing just the empty clause. The empty clause [] is understood as a representation of $\neg$TRUE (the disjunction of no possibilities), and so $\{[]\}$ also stands for $\neg$TRUE. However, the empty clausal formula $\{\}$ (the conjunction of no constraints) is a representation of TRUE.

For convenience, we will move freely back and forth between ordinary formulas in CNF and their representation as sets of clauses.

Putting the comments made at the start of the chapter together with what we have seen about CNF, we get that as far as deductive reasoning is concerned, to determine whether or not KB $\models \alpha$ it will be sufficient to do the following:

1. put the sentences in KB and $\neg\alpha$ into CNF;
2. determine whether or not the resulting set of clauses is satisfiable.

In other words, any question about entailment can be reduced to a question about the satisfiability of a set of clauses.

4.1.1 Resolution derivations

To answer the question about whether or not a set of clauses is satisfiable, we use a rule of inference called *Resolution*:

> Given a clause of the form $c_1 \cup \{\rho\}$ containing some literal ρ, and a clause of the form $c_2 \cup \{\overline{\rho}\}$ containing the complement of ρ, infer the clause $c_1 \cup c_2$ consisting of those literals in the first clause other than ρ and those in the second other than $\overline{\rho}$.[2]

We say in this case that $c_1 \cup c_2$ is a *resolvent* of the two input clauses with respect to ρ. For example, from clauses $[w, p, q]$ and $[s, w, \neg p]$, we have the clause $[w, s, q]$ as a resolvent with respect to p. The clauses $[p, q]$ and $[\neg p, \neg q]$ have two resolvents: $[q, \neg q]$ with respect to p, and $[p, \neg p]$ with respect to q. Note that [] is not a resolvent of these two clauses. The only way to get the empty clause is to resolve two complementary unit clauses like $[\neg p]$ and $[p]$.

A Resolution *derivation* of a clause c from a set of clauses S is a sequence of clauses $c_1, \ldots, c_n$, where the last clause, c_n is c, and where each c_i is either an element of S or a resolvent of two earlier clauses in the derivation. We write $S \vdash c$ if there is a derivation of c from S.

Why do we care about Resolution derivations? The main point is that this purely symbol-level operation on finite sets of literals has a direct connection to knowledge-level logical interpretations.

Observe first of all that a resolvent is always entailed by the two input clauses. Suppose we have two clauses $c_1 \cup \{p\}$ and $c_2 \cup \{\neg p\}$. We claim that

$$\{c_1 \cup \{p\},\ c_2 \cup \{\neg p\}\} \models c_1 \cup c_2.$$

To see why, let $\Im$ be any interpretation, and suppose that $\Im \models c_1 \cup \{p\}$ and $\Im \models c_2 \cup \{\neg p\}$. There are two cases: if $\Im \models p$, then $\Im \not\models \neg p$, but since $\Im \models c_2 \cup \{\neg p\}$, we must have that $\Im \models c_2$, and so $\Im \models c_1 \cup c_2$; similarly, if $\Im \not\models p$, then since $\Im \models c_1 \cup \{p\}$, we must have that $\Im \models c_1$, and so again $\Im \models c_1 \cup c_2$. Either way, we get that $\Im \models c_1 \cup c_2$.

We can extend this argument to prove that any clause derivable by Resolution from S is entailed by S, that is, if $S \vdash c$, then $S \models c$. We show by induction on the length of the derivation that for every c_i, $S \models c_i$: this is clearly true if $c_i \in S$, and otherwise, c_i is a resolvent of two earlier clauses, and so is entailed by them, as argued above, and hence by S.

The converse, however, does *not* hold: we can have $S \models c$ without having $S \vdash c$. For example, let S consist of the single clause $[\neg p]$ and let c be $[\neg q, q]$. Then S clearly entails c even though it has no resolvents. In other words, as a form of reasoning, finding Resolution derivations is sound but not complete.

[2] Either c_1 or c_2 or both can be empty. In the case that c_1 is empty, $c_1 \cup \{\rho\}$ would be the unit clause $[\rho]$.

Figure 4.1: A Resolution procedure

Input: a finite set S of propositional clauses
Output: satisfiable or unsatisfiable

1. Check if $[] \in S$; if so, return unsatisfiable.
2. Otherwise, check if there are two clauses in S, such that they resolve to produce another clause not already in S; if not, return satisfiable.
3. Otherwise, add the new resolvent clause to S, and go back to step 1.

Despite this incompleteness, however, Resolution does have a property that allows it to be used without loss of generality to calculate entailments: Resolution is both sound and complete *when c is the empty clause*. In other words, there is a theorem that states that $S \vdash []$ iff $S \models []$.[3] This means that S is unsatisfiable iff $S \vdash []$. This provides us with a way of determining the satisfiability of any set of clauses, since all we need to do is search for a derivation of the empty clause. Since this works for any set S of clauses, we sometimes say that Resolution is *refutation complete*.

4.1.2 An entailment procedure

We are now ready to consider a symbol-level procedure for determining if KB $\models \alpha$. The idea is to put both KB and $\neg\alpha$ into CNF, as discussed before, and then to check if the resulting set S of clauses (for both) is unsatisfiable by searching for a derivation of the empty clause. As discussed above, S is unsatisfiable iff KB$\cup\{\neg\alpha\}$ is unsatisfiable iff KB $\models \alpha$. This can be done using the nondeterministic procedure in Figure 4.1. What the procedure does is to repeatedly add resolvents to the input clauses S until either the empty clause is added (in which case there is a derivation of the empty clause) or no new clauses can be added (in which case there is no such derivation). If we were interested in returning or printing out a derivation, we would store with each derived clause pointers to its input clauses.

The procedure does not distinguish between clauses that come from the KB, and those that come from the negation of α, which we will call the *query*. Observe that if we have a number of queries we want to ask for the same KB, we need only

[3] This theorem will also carry over to quantified clauses later.

convert the KB to CNF once and then add clauses for the negation of each query. Moreover, if we want to add a new fact α to the KB, we can do so by adding the clauses for α to those already calculated for KB. Thus, to use this type of entailment procedure, it makes good sense to keep KB in CNF, adding and removing clauses as necessary.

Let us now consider some simple examples of this procedure in action. We start with the following KB:

Toddler
Toddler ⊃ Child
Child ∧ Male ⊃ Boy
Infant ⊃ Child
Child ∧ Female ⊃ Girl
Female

We can read these sentences as if they were talking about a particular person: the person is a toddler; if the person is a toddler then the person is a child; if the person is a child and male, then the person is a boy; if the person is an infant, then the person is a child; if the person is a child and female, then the person is a girl; the person is female. In Figure 4.2, we graphically display a Resolution derivation showing that the person is a girl, by showing that KB $\models$ Girl. Observe that in this diagram we use a dashed line to separate the clauses that come directly from the KB or the negation of the query from those that result from applying Resolution. There are six clauses from the KB, one from the negation of the query (i.e., ¬Girl), and four new ones generated by Resolution. Each resolvent in the diagram has two solid lines pointing up to its input clauses. The resulting graph will never have cycles, because input clauses must always appear earlier in the derivation. Note that there are two clauses in the KB that are not used in the derivation and could be left out of the diagram.

A second example uses the following KB:

Sun ⊃ Mail
(Rain ∨ Sleet) ⊃ Mail
Rain ∨ Sun

These formulas can be understood as talking about the weather and the mail service on a particular day. In Figure 4.3, we have a Resolution derivation showing that KB $\models$ Mail. Note that the formula ((Rain∨Sleet) ⊃ Mail) results in two clauses on conversion to CNF. If we wanted to show that KB $\not\models$ Rain, for the same KB, we could do so by displaying a similar graph that contains the clause [¬Rain] and every possible resolvent, but does not contain the empty clause.

Figure 4.2: A first example Resolution derivation

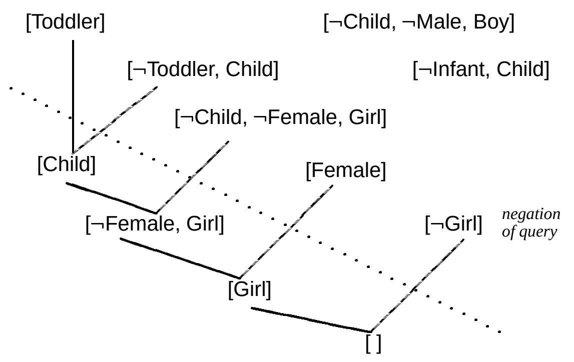
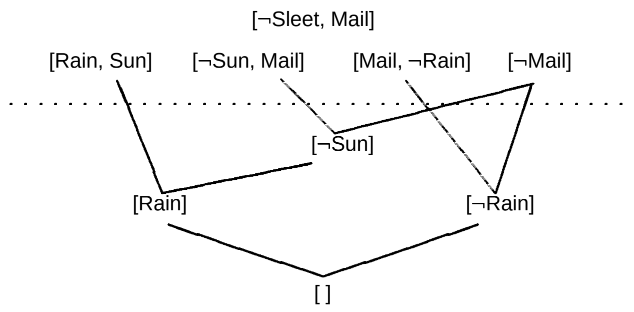


Figure 4.3: A second example Resolution derivation



4.2 Handling variables and quantifiers

Having seen how to do Resolution for the propositional case, we now consider reasoning with variables, terms, and quantifiers. Again, we will want to convert formulas into an equivalent clausal form. For simplicity, we begin by assuming that no existential quantifiers remain once negations have been moved inwards.[4]

1. eliminate $\supset$ and $\equiv$, as before;
2. move $\neg$ inwards so that it appears only in front of an atom, using for example, the fact that $\neg\exists x.\alpha$ is equivalent to $\forall x.\neg\alpha$;
3. standardize variables, that is, ensure that each quantifier is over a distinct variable by renaming them as necessary;
4. eliminate all remaining existentials (discussed later);
5. move universals to the front using for example the fact that $(\forall x\alpha \wedge \beta)$ is equivalent to $\forall x(\alpha \wedge \beta)$ when β does not contain x;
6. distribute $\wedge$ over $\vee$, as before;
7. collect terms as before.

The end result of this procedure is a quantified version of CNF, a universally quantified conjunction of disjunctions of literals, that is once again logically equivalent to the original formula.

Again it is convenient to use a *clausal form* of CNF. We simply drop the quantifiers (since they are all universal anyway), and we are left with a set of clauses, each of which is a set of literals, each of which is either an atom or its negation. An atom now is of the form $P(t_1, \ldots, t_n)$, where the terms t_i may contain variables, constants, and function symbols.[5] Clauses are understood exactly as they were before, except that variables appearing in them are interpreted universally. So for example, the clausal formula

$$\{[P(x), \neg R(a, f(b,x))],\ [Q(x,y)]\}$$

stands for the CNF formula

$$\forall x \forall y\, \{[P(x) \vee \neg R(a, f(b,x))] \ \wedge\ \ Q(x,y)\}.$$

It is convenient to introduce special notation and terminology for *substitutions*. A *substitution* θ is a finite set of pairs $\{x_1/t_1, \ldots, x_n/t_n\}$ where the x_i are distinct variables and the t_i are arbitrary terms. If θ is a substitution and ρ is a literal, then $\rho\theta$ is the literal that results from simultaneously replacing each x_i in ρ by

[4]We will see how to handle existentials in Section 4.2.3.

[5]For now, we ignore atoms involving equality.

t_i. For example, if $\theta = \{x/a, y/g(x,b,z)\}$, and $\rho = P(x,z,f(x,y))$, then $\rho\theta = P(a,z,f(a,g(x,b,z)))$. Similarly, if c is a clause, $c\theta$ is the clause that results from performing the substitution on each literal. We say that a literal (or clause) is *ground* if it contains no variables. We say that a literal ρ is an *instance* of ρ' if for some θ, $\rho = \rho'\theta$.

4.2.1 First-order Resolution

We now consider the Resolution rule as applied to clauses with variables. The main idea is that since clauses with variables are implicitly universally quantified, we want to allow Resolution inferences that can be made from any of their instances.

For example, suppose we have clauses

$$[P(x,a), \neg Q(x)] \ \text{ and } \ [\neg P(b,y), \neg R(b, f(y))].$$

Then implicitly at least, we also have clauses

$$[P(b,a), \neg Q(b)] \ \text{ and } \ [\neg P(b,a), \neg R(b, f(a))],$$

which resolve to $[\neg Q(b), \neg R(b, f(a))]$. We will define the rule of Resolution so that this clause is a resolvent of the two original ones.

So the general rule of Resolution is as follows:

> Suppose we are given a clause of the form $c_1 \cup \{\rho_1\}$ containing some literal ρ_1, and a clause of the form $c_2 \cup \{\overline{\rho_2}\}$ containing the complement of a literal ρ_2. Suppose we rename the variables in the two clauses so that each clause has distinct variables, and that there is a substitution θ such that $\rho_1\theta = \rho_2\theta$. Then, we can infer the clause $(c_1 \cup c_2)\theta$ consisting of those literals in the first clause other than ρ_1 and those in the second other than $\overline{\rho_2}$, after applying θ.

We say in this case that θ *unifies* ρ_1 and ρ_2, and that θ is a *unifier* of the two literals.

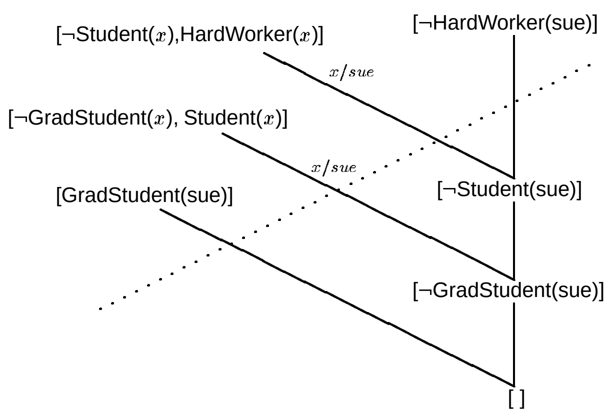
With this new general rule of Resolution, the definition of a derivation stays the same, and ignoring equality, we get as before that $S \vdash []$ iff $S \models []$.

We will use the same conventions as before to show Resolution derivations in diagrams, except that we will now show the unifying substitution as a label near one of the solid lines.[6]

As an example, consider the following KB:

[6]Since it is sometimes not obvious which literals in the input clauses are being resolved, for clarity, we point to them in the input clauses.

Figure 4.4: An example Resolution derivation with variables



$\forall x$.GradStudent(x) $\supset$ Student(x)
$\forall x$.Student(x) $\supset$ HardWorker(x)
GradStudent(sue)

In Figure 4.4, we show that KB $\models$ HardWorker(sue). Note that the conversion of this KB to CNF did not require either existentials or equality.

A slightly more complex derivation is presented in Figure 4.5. This is a Resolution derivation corresponding to the three-block problem first presented in Chapter 1: if there are three stacked blocks where the top one is green, and the bottom one is not green, is there a green block directly on top of a non-green block? The KB here is

On(a,b), On(b,c), Green(a), ¬Green(c)

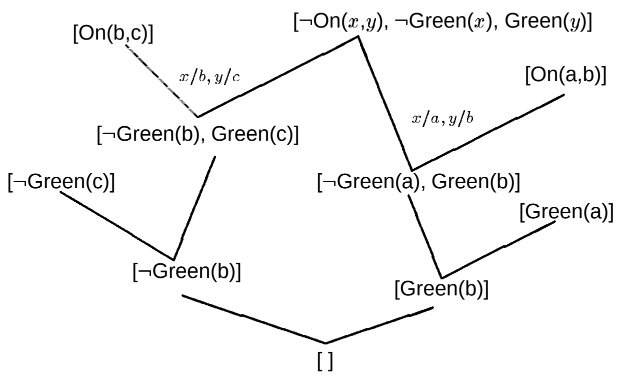
where the three blocks are a, b, and c. Note that this KB is already in CNF. The query is

$\exists x \exists y$.On(x, y) $\wedge$ Green(x) $\wedge$ ¬Green(y)

whose negation contains no existentials or equalities.

Using a Resolution derivation, it is possible to get answers to queries that we might think of as requiring computation. To do arithmetic, for example, we can

Figure 4.5: The 3 block problem



use the constant zero to stand for 0, and succ to stand for the successor function. Every number can then be written as a ground term using these two symbols. For instance, the term

succ(succ(succ(succ(succ(zero)))))

stands for 5. We can use the predicate Plus(x, y, z) to stand for the relation $x+y = z$, and start with a KB that formalizes the properties of addition as follows:

$\forall x$.Plus(zero, x, x)
$\forall x \forall y \forall z$.Plus($x, y, z$) $\supset$ Plus(succ(x), y, succ(z)).

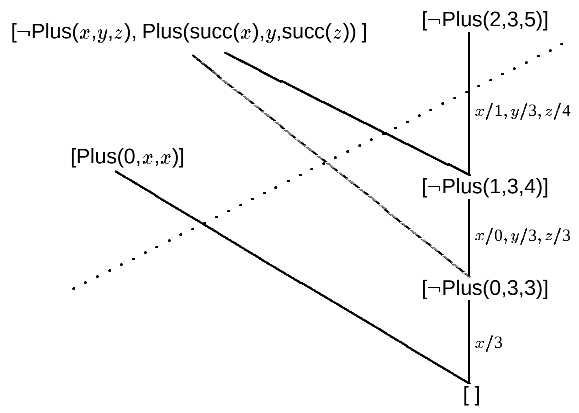
All the expected relations among triples of numbers are entailed by this KB. For example, in Figure 4.6, we show that $2 + 3 = 5$ follows from this KB.[7] A derivation for an entailed existential formula like

$\exists u$.Plus(2, 3, u),

is similar, as shown in Figure 4.7. Here, we need to be careful to rename variables (using v and w) to ensure that the variables in the input clauses are distinct. Observe that by examining the bindings for the variables, we can locate the value of u: it is bound to succ(v), where v is bound to succ(w), and w to 3. In other words,

[7]For readability, instead of using terms like succ(succ(zero)), we write the decimal equivalent, 2.

Figure 4.6: Arithmetic in FOL



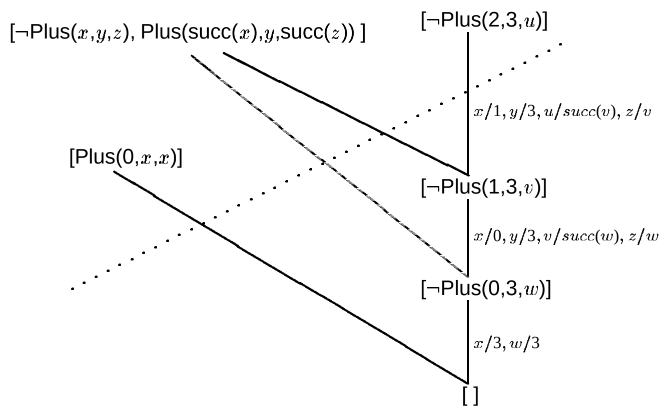
the answer for the addition is correctly determined to be 5. As we will see later in Chapter 5, this form of computation, including locating the answers in a derivation of an existential, is what underlies the Prolog programming language.

4.2.2 Answer extraction

While it is often possible to get answers to questions by looking at the bindings of variables in a derivation of an existential, in full FOL, the situation is more complicated. Specifically, it can happen that a KB entails some $\exists x.P(x)$, without entailing $P(t)$ for any specific t. For example, in the three-block problem from Figure 4.5, the KB entails that *some* block must be green and on top of a non-green block, but not which.

One general method that has been proposed for dealing with answers to queries even in cases like these is the *answer-extraction* process. Here is the idea: we replace a query such as $\exists x.P(x)$ (where x is the variable we are interested in) by $\exists x.P(x) \wedge \neg A(x)$ where A is a new predicate symbol occurring nowhere else, called, the *answer predicate*. Since A appears nowhere else, it will normally not be possible to derive the empty clause from the modified query. Instead, we terminate

Figure 4.7: An existential arithmetic query


[¬Plus(x,y,z), Plus(succ(x),y,succ(z))]
[¬Plus(2,3,u)]
$x/1, y/3, u/succ(v), z/v$
[Plus(0,x,x)]
[¬Plus(1,3,v)]
$x/0, y/3, v/succ(w), z/w$
[¬Plus(0,3,w)]
$x/3, w/3$
[ ]

the derivation as soon as we produce a clause containing *only* the answer predicate.

To see this in action, we begin with an example having a definite answer. Suppose the KB is

Student(john)
Student(jane)
Happy(john)

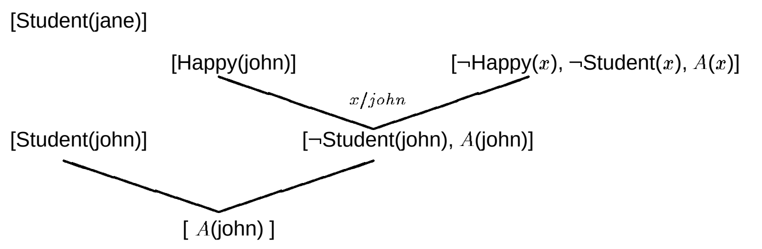
and we wish to show that some student is happy. The query then is

$$\exists x.\text{Student}(x) \wedge \text{Happy}(x).$$

In Figure 4.8, we show a derivation augmented with an answer predicate to derive who that happy student is. The final clause can be interpreted as saying that "An answer is John." A normal derivation of the empty clause can be easily produced from this one by eliminating all occurrences of the answer predicate.

Observe that in this example, we say that *an* answer is produced by the process. There can be many such answers, but each derivation only deals with one. For example, if the KB had been

Figure 4.8: Answer predicate with a definite answer



Student(john)
Student(jane)
Happy(john)
Happy(jane)

then, in one derivation we might extract the answer jane, and in another, john.

Where the answer extraction process especially pays off is in cases involving indefinite answers. Suppose, for example, our KB had been

Student(john)
Student(jane)
Happy(john) ∨ Happy(jane)

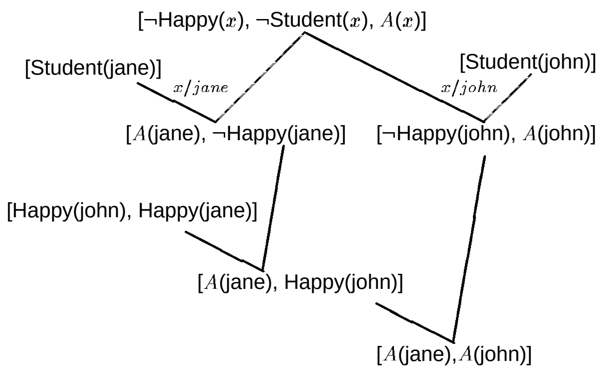
Then we can still see that there is a student who is happy, although we cannot say who. If we use the same query and answer extraction process, we get the derivation in Figure 4.9. In this case, the final clause can be interpreted as saying that "An answer is either Jane or John", which is as specific as the KB allows.

Finally, it is worth noting that the answer extraction process can result in clauses containing variables. For example, if our KB had been

$\forall w$.Student($f(a, w)$)
$\forall y \forall z$.Happy($f(y, g(z))$)

we get a derivation whose final clause is $[A(f(a, g(z)))]$, which can be interpreted as saying that "An answer is any instance of the term $f(a, g(z))$."

Figure 4.9: Answer predicate with an indefinite answer


[¬Happy(x), ¬Student(x), $A(x)$]
[Student(jane)]
$x/jane$
[Student(john)]
$x/john$
[A(jane), ¬Happy(jane)]
[¬Happy(john), A(john)]
[Happy(john), Happy(jane)]
[A(jane), Happy(john)]
[A(jane),A(john)]

4.2.3 Skolemization

So far, in converting formulas to CNF, we have ignored existentials. For example, we could not handle facts in a KB like $\exists x \forall y \exists z. P(x, y, z)$, since we had no way to put them into CNF.

To handle existentials and represent such facts, we use the following idea: since some individuals are claimed to exist, we introduce names for them (called *Skolem constants* and *Skolem functions*, for the logician who first introduced them) and represent facts like the above using those names. If we are careful not to use the names anywhere else, then what will be entailed will be precisely what was entailed by the original existential. For the above formula, for example, an x is claimed to exist, so call it a; moreover, for each y, a z is claimed to exist, call it $f(y)$. So instead of reasoning with $\exists x \forall y \exists z. P(x, y, z)$, we use $\forall y. P(a, y, f(y))$, where a and f are Skolem symbols appearing nowhere else. Informally, if we think of the conclusions we can draw from this formula, they will be the same as those we can draw from the original existential (as long as they do not mention a or f).

In general, then, in our conversion to CNF, we eliminate all existentials (at step 4) by what is called *Skolemization*: repeatedly replace an existential variable by a new function symbol with as many arguments as there are universal variables dominating the existential. In other words, if we start with

$$\forall x_1(\ldots \forall x_2(\ldots \forall x_3(\ldots \exists y[\ldots\ y\ \ldots] \ldots) \ldots) \ldots),$$

where existentially quantified y appears in the scope of universally quantified x_1, x_2, x_3, and only these, we end up with

$$\forall x_1(\ldots \forall x_2(\ldots \forall x_3(\ldots [\ldots\ f(x_1,x_2,x_3)\ \ldots] \ldots) \ldots) \ldots),$$

where f appears nowhere else.

If α is our original formula, and α' is the result of converting it to CNF including Skolemization, then we no longer have that $\models (\alpha \equiv \alpha')$ as we had before. For example, $\exists x. P(x)$ is not logically equivalent to $P(a)$, its Skolemized version. What can be shown, however, is that α is satisfiable iff α' is satisfiable, and this is really all we need for Resolution.[8]

Note that Skolemization depends crucially on the universal variables that dominate the existential. A formula like $\exists x \forall y R(x,y)$ entails $\forall y \exists x R(x,y)$, but the converse does not hold. To show that the former holds using Resolution, we show that

$$\{\exists x \forall y R(x,y), \neg\forall y \exists x R(x,y)\}$$

is unsatisfiable. After conversion to CNF, we get the clauses

$$\{[R(a,y)],\ [\neg R(x,b)]\}$$

where a and b are Skolem constants, which resolve to the empty clause in one step. If we were to try the same with the converse, we would need to show that

$$\{\neg\exists x \forall y R(x,y), \forall y \exists x R(x,y)\}$$

was unsatisfiable. After conversion to CNF, we get

$$\{[\neg R(x,g(x))],\ [R(f(y),y)]\}$$

where f and g are Skolem functions. In this case, there is no derivation of the empty clause (nor should there be) because the two literals $R(x,g(x))$ and $R(f(y),y)$ cannot be unified.[9] So for logical correctness, it is important to get the dependence of variables right. In one case, we had $R(a,y)$ where the value of the existential x did not depend on universal y (i.e., in $\exists x \forall y R(x,y)$); in the other case, we had the much weaker $R(f(y),y)$ where the value of the existential x could depend on the universal (i.e., in $\forall y \exists x R(x,y)$).

[8]We do need to be careful, however, with answer extraction, not to confuse real constants (that have meaning in the application domain) with Skolem constants that are generated only to avoid existentials.

[9]To see this, note that if x is replaced by t_1 and y by t_2, then t_1 would have to be $f(t_2)$ and t_2 would have to be $g(t_1)$. So t_1 would have to be $f(g(t_1))$ which is impossible.

4.2.4 Equality

So far, we have ignored formulas containing equality. If we were to simply treat = as a normal predicate, we would miss many unsatisfiable sets of clauses, for example, $\{a=b, b=c, a\neq c\}$. To handle these, it is necessary to augment the set of clauses to ensure that all the special properties of equality are taken into account. What we require are the clausal versions of the *axioms of equality*:

reflexitivity: $\forall x.\ x=x$;

symmetry: $\forall x \forall y.\ x=y \supset y=x$;

transitivity: $\forall x \forall y \forall z.\ x=y \wedge y=z \supset x=z$;

substitution for functions: for every function symbol f of arity n, an axiom

$$\forall x_1 \forall y_1 \cdots \forall x_n \forall y_n.\ x_1=y_1 \wedge \cdots \wedge x_n=y_n \supset f(x_1,\ldots,x_n)=f(y_1,\ldots,y_n);$$

substitution for predicates: for every predicate symbol P of arity n, an axiom

$$\forall x_1 \forall y_1 \cdots \forall x_n \forall y_n.\ x_1=y_1 \wedge \cdots \wedge x_n=y_n \supset P(x_1,\ldots,x_n)\equiv P(y_1,\ldots,y_n).$$

It can be shown that with the addition of these axioms, equality can be treated as a binary predicate, and soundness and completeness of Resolution for the empty clause will be preserved.

A simple example of the use of the axioms of equality can be found in Figure 4.10. In this example, the KB is

$\forall x$.Married(father(x), mother(x))
father(john)=bill

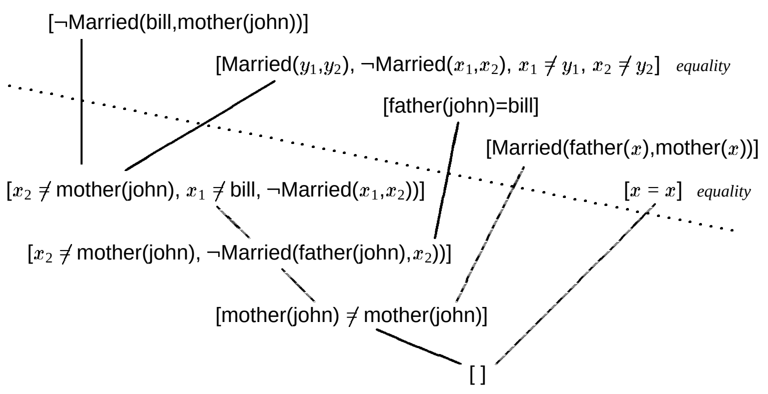
and the query to derive is

Married(bill,mother(john)).

Note that the derivation uses two of the axioms: reflexitivity, and substitution for predicates.

Although the axioms of equality are sufficient for Resolution, they do result in a very large number of resolvents, and their use can easily come to dominate Resolution derivations. A more efficient treatment of equality is discussed below.

Figure 4.10: Using the axioms of equality


[¬Married(bill,mother(john))]
[Married(y_1,y_2), ¬Married(x_1,x_2), $x_1 \neq y_1$, $x_2 \neq y_2$] equality
[father(john)=bill]
[Married(father(x),mother(x))]
[$x_2 \neq$ mother(john), $x_1 \neq$ bill, ¬Married(x_1,x_2))]
[$x = x$] equality
[$x_2 \neq$ mother(john), ¬Married(father(john),x_2))]
[mother(john) $\neq$ mother(john)]
[ ]

4.3 Dealing with computational intractability

The success we have had using Resolution derivations should not mislead us into thinking that Resolution provides a general effective solution to the reasoning problem.

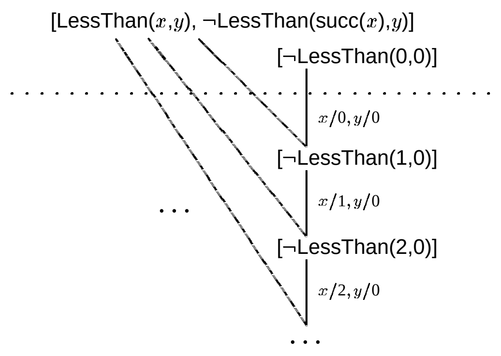
4.3.1 The first-order case

Consider, for example, the KB consisting of a single formula (again in the domain of arithmetic):

$$\text{LessThan}(\text{succ}(x), y) \supset \text{LessThan}(x, y).$$

Suppose our query is LessThan(zero,zero). Obviously, this should fail since the KB does not entail the query (nor its negation). The problem is that if we pose it to Resolution, we get derivations like the one shown in Figure 4.11. Although we never generate the empty clause, we might generate an *infinite* sequence looking for it. Among other things, this suggests that we cannot simply use a depth-first procedure to search for the empty clause, since we run the risk of getting stuck on such an infinite branch.

Figure 4.11: An infinite Resolution branch



We might ask if there is any way to detect when we are on such a branch, so that we can give it up and look elsewhere. The answer unfortunately is *no*. The FOL language is very powerful and can be used as a full programming language. Just as there is no way to detect when a program is looping, there is no way to detect if a branch will continue indefinitely.

This is quite problematic from a KR point of view since it means that there can be no procedure which, given a set of clauses, returns satisfiable when the clauses are satisfiable, and unsatisfiable otherwise.[10] However, we do know that Resolution is refutation complete: if the set of clauses is unsatisfiable, some branch will contain the empty clause (even if some branches may be infinite). So a breadth-first search is guaranteed to report unsatisfiable when the clauses are unsatisfiable. When the clauses are satisfiable, the search may or may not terminate.

In this section, we examine what we can do about this issue.

4.3.2 The Herbrand Theorem

We saw in Section 4.1 that in the propositional case, we can run Resolution to completion, and so we never have the non-termination problem. An interesting fact about Resolution in FOL is that it sometimes reduces to this propositional case. Given a set S of clauses, the *Herbrand universe* of S (named after the logician who

[10]We will see in Chapter 5 that this is also true for the much simpler case of Horn clauses.

first introduced it) is the set of all ground terms formed using just the constants and function symbols in S.[11] For example if S mentions just constants a and b and unary function symbol f, then the Herbrand universe is the set

$$\{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), \ldots\}$$

The *Herbrand base* of S is the set of all ground clauses $c\theta$ where $c \in S$ and θ assigns the variables in c to terms in the Herbrand universe.

Herbrand's theorem is that a set of clauses is satisfiable iff its Herbrand base is.[12] The reason this is significant is that the Herbrand base is a set of clauses without variables, and so is essentially propositional. To reason with the Herbrand base it is not necessary to use unifiers and so on, and we have a sound and complete reasoning procedure that is guaranteed to terminate.

The catch in this approach (and there must be a catch since no procedure can decide the satisfiability of arbitrary sets of clauses) is that the Herbrand base will typically be an *infinite* set of propositional clauses. It will however, be finite when the Herbrand universe is finite (no function symbols and only finitely many constants appear in S). Moreover, sometimes we can keep the universe finite by considering the "type" of the arguments and values of functions, and include a term like $f(t)$ only if the type of t is appropriate for the function f. For example, if our function is birthday (taking a person as argument and producing a date), we may be able to avoid meaningless term like birthday(birthday(john)) in the Herbrand universe.

4.3.3 The propositional case

If we can get a finite set of propositional clauses, we know that the Resolution procedure in Figure 4.1 will terminate. But this does not make it practical. The procedure may terminate, but how long will it take? We might think that this depends on how good our procedure is at finding derivations. However, in 1985, Armin Haken proved that there are unsatisfiable propositional clauses $c_1, \ldots, c_n$ such that the *shortest* derivation of the empty clause had on the order of 2^n steps. This answers the question definitively: no matter how clever we are at finding derivations, and even if we avoid all needless searching, any Resolution procedure will still take *exponential* time on such clauses since it takes that long to get to the end of the derivation.

We might then wonder if this is just a problem with Resolution: might there not be a better way to determine whether a set of propositional clauses is satisfiable? As it turns out, this question is one of the deepest ones in all of Computer

[11] In case S mentions no constant or function symbols, we use a single constant, say a.

[12] This applies to Horn clauses too, as discussed in Chapter 5.

Science and still has no definite answer. In 1972, Steven Cook proved that the satisfiability problem was *NP-complete*: roughly, any search problem where what is being searched for can be verified in polynomial time can be recast as a propositional satisfiability problem. The importance of this result is that many problems of practical interest (in areas such as scheduling, routing, and packing) can be formulated as search problems of this form.[13] Thus a good algorithm for satisfiability (which Haken proved Resolution is not) would imply a good algorithm for all of these tasks. Since so many people have been unable to find good algorithms for any of them, it is strongly believed that propositional satisfiability cannot be solved at all in polynomial time. Proofs, like Haken's for Resolution, however, have been very hard to obtain.

4.3.4 The implications

So what are the implications of these negative results? At the very least, they tell us that Resolution is not a panacea. For KR purposes, we would like to be able to produce entailments of a KB for immediate action, but determining the satisfiability of clauses may simply be too difficult computationally for this purpose.

We may need to consider some other options. One is to give more control over the reasoning process to the user. This is a theme that will show up in the procedural representations in Chapters 5 and 6 and others. Another option is to consider the possibility of using representation languages that are less expressive than full FOL or even full propositional logic. This is a theme that will show up in Chapters 5 and 9, among others. Much of the research in Knowledge Representation and Reasoning can be seen as attempts to deal with this issue, and we will return to it in detail in Chapter 16.

On the other hand, it is worth observing that in some applications of Resolution, it is reasonable to wait for answers, even for a very long time. Using Resolution to do *mathematical theorem proving*, for example to determine whether or not Goldbach's conjecture or its negation follows from the axioms of number theory, is quite different from using Resolution to determine whether or not an umbrella is needed when it looks like rain. In the former case, we might be willing to wait for months or even years for an answer. There is an area of AI called *automated theorem-proving* whose subject matter is precisely the development of procedures for such mathematical applications.

[13] An example is the so-called Traveling Salesman Problem: given a graph with nodes standing for cities, and edges with numbers on them standing for direct routes between cities that many kilometers apart, determine if there is a way to visit all the cities in the graph in less than some given number k of kilometers.

The best we can hope for in such applications of Resolution is not a guarantee of efficiency or even of termination, but a way to search for derivations that eliminates unnecessary steps as much as possible. In the rest of this section, we will consider strategies that can be used to improve the search in this sense.

4.3.5 Most general unifiers

The most important way of avoiding needless search in a derivation is to keep the search as general as possible. Consider, for example two clauses c_1 and c_2, where c_1 contains the literal $P(g(x), f(x), z)$ and c_2 contains $\neg P(y, f(w), a)$. These two literals are unified by the substitution

$$\theta_1 = \{x/b, y/g(b), z/a, w/b\}$$

and also by

$$\theta_2 = \{x/f(z), y/g(f(z)), z/a, w/f(z)\}.$$

We may very well be able to derive the empty clause using θ_1; but if we cannot, we will need to consider other substitutions like θ_2, and so on.

The trouble is that both of these substitutions are overly specific. We can see that any unifier must give w the same value as x, and y the same as $g(x)$, but we do not need to commit yet to a value for x. The substitution

$$\theta_3 = \{y/g(x), z/a, w/x\}$$

unifies the two literals without making an arbitrary choice that might preclude a path to the empty clause. It is a most general unifier.

More precisely, a *most general unifier* (MGU) θ of literals ρ_1 and ρ_2 is a unifier that has the property that for any other unifier θ', there is a further substitution θ^* such that $\theta' = \theta \cdot \theta^*$.[14] So starting with θ you can always get to any other unifier by applying additional substitutions. For example, given θ_3, we can get to θ_1 by further applying x/b, and to θ_2 by applying $x/f(z)$. Note that an MGU need not be unique, in that

$$\theta_4 = \{y/g(w), z/a, x/w\}$$

is also one for c_1 and c_2.

The key fact about MGUs is that (with certain restrictions that need not concern us here) we can limit the Resolution rule to MGUs without loss of completeness. This helps immensely in the search since it dramatically reduces the number of

[14]By $\theta \cdot \theta^*$ we mean the substitution such that for any literal ρ, $\rho(\theta \cdot \theta^*) = (\rho\theta)\theta^*$, that is, we apply θ to ρ and then apply θ^* to the result.

resolvents that can be inferred from two input clauses. Moreover, an MGU of a pair of literals ρ_1 and ρ_2 can be calculated efficiently, by the following procedure:

1. start with $\theta = \{\}$;
2. exit if $\rho_1\theta = \rho_2\theta$;
3. otherwise get the disagreement set, DS, which is the pair of terms at the first place where the two literals disagree;

   i.e., if $\rho_1\theta = P(a, f(a, g(z), \ldots))$ and $\rho_2\theta = P(a, f(a, u, \ldots))$, then $DS = \{u, g(z)\}$;
4. find a variable $v \in DS$, and a term $t \in DS$ not containing v; if not, fail;
5. otherwise, set θ to $\theta \cdot \{v/t\}$, and go to step 2.

This procedure runs in something like $O(n^2)$ time on the length of the terms, and an even better but more complex *linear* time algorithm exists.

Because MGUs greatly reduce the search and can be calculated efficiently, all Resolution-based systems implemented to date use them.

4.3.6 Other refinements

A number of other refinements to Resolution have been proposed to help improve the search.

Clause elimination The idea is to keep the number of clauses generated as small as possible, without giving up completeness, by using the fact that if there is a derivation to the empty clause at all, then there is one that does not use the clause in question. Some examples are:

- *pure clauses*: these are clauses that contain some literal ρ such that $\overline{\rho}$ does not appear anywhere;
- *tautologies*: these are clauses that contain both ρ and $\overline{\rho}$, and can be bypassed in any derivation;
- *subsumed clauses*: these are clauses for which there already exists another clause with a subset of the literals (perhaps after a substitution).

Ordering strategies The idea is prefer to perform Resolution steps in a fixed order, trying to maximize the chance of deriving the empty clause. The best strategy found to date (but not the only one) is *unit preference*, that is, to use unit clauses first. This is because using a unit clause together with a clause of length k always produces a clause of length $k-1$. By going for shorter and shorter clauses, the hope is to arrive at the empty clause more quickly.

Set of support In a KR application, even if the KB and the negation of a query are unsatisfiable, we still expect the KB by itself to be satisfiable. It therefore makes sense not to perform Resolution steps involving only clauses from the KB. The set of support strategy says that we are only allowed to perform Resolution if at least one of the input clauses has an ancestor in the negation of the query. Under the right conditions, this can be done without loss of completeness.

Special treatment of equality We examined above one way to handle equality using the axioms of equality explicitly. Because these can generate so many resolvents, a better way is to introduce a second rule of inference in addition to Resolution, called *Paramodulation*:

> Suppose we are given a clause $c_1 \cup \{t = s\}$ where t and s are terms, and a clause $c_2 \cup \{\rho[t']\}$ containing some term t'. Suppose we rename the variables in the two clauses so that each clause has distinct variables, and that there is a substitution θ such that $t\theta = t'\theta$. Then, we can infer the clause $(\{c_1 \cup c_2 \cup \rho[s]\})\theta$ which eliminates the equality atom, replaces t' by s, and then performs the θ substitution.

With this rule, it is no longer necessary to include the axioms of equality, and what would have required many steps of Resolution involving those axioms, can be done in a single step. Using the previous example above, it is not hard to see that from

$$[\mathsf{father(john) = bill}] \text{ and } [\mathsf{Married(father}(x), \mathsf{mother}(x))],$$

we can derive the clause [Married(bill,mother(john))] in a singe Paramodulation step.

Sorted logic The idea here is to associate sorts with all terms. For example a variable x might be of sort Male, and the function mother might be of sort [Person $\rightarrow$ Female]. We might also want to keep a taxonomy of sorts, for example, that Woman is a subsort of Person. With this information in place, we can refuse to unify $P(s)$ with $P(t)$ if the sorts of s and t are incompatible. The assumption here is that only meaningful (with respect to sorts) unifications can ever lead to the empty clause.

Connection graph In the connection graph method, given a set of clauses, we pre-compute a graph with edges between any two unifiable literals of opposite polarity, and labeled with the MGU of the two literals. In other words, we start by pre-computing all possible unifications. The Resolution procedure, then, involves selecting a link, computing a resolvent clause, and inheriting links for the new clause from its input clauses after substitution. No unification is done at "run time." With this, Resolution can be seen as a kind of state-space search problem — find a sequence of links that ultimately produces the empty clause — and any technique for improving a state-space search (such as using a heuristic function) can be applied to Resolution.

Directional connectives A clause like $[\neg p, q]$, representing "if p then q", can be used in a derivation in two ways: in the forward direction, if we derive a clause containing p, we then derive the clause with q; in the backward direction, if we derive a clause containing $\neg q$, we then derive the clause with $\neg p$. The idea with directional connectives is to mark clauses to be used in one or the other direction only. For example, given a fact in a KB like

$$\forall x.\mathsf{BattleShip}(x) \supset \mathsf{Gray}(x)$$

we may wish to use this only in the forward direction, since it is probably a bad idea to work on deriving that something is gray by trying to derive that it is a battleship. Similarly, a fact like

$$\forall x.\mathsf{Person}(x) \supset \mathsf{Has}(x, \mathsf{spleen})$$

might be used only in the backward direction since it is probably a bad idea to derive having a spleen for every individual derived to be a person. This form of control over how facts are used is the basis for the procedural representation languages which will be discussed extensively in Chapter 6.

4.4 Bibliographic notes

4.5 Exercises

Chapter 5

Horn Logic

In the previous chapter, we saw how a Resolution procedure could in principle be used to calculate entailments of any first-order logical KB. But we also saw that in its most general form, Resolution ran into serious computational difficulties. Although refinements to Resolution help to deal with this issue, if we want to use all of first-order logic, these difficulties can never be completely eliminated. This is a consequence of the fundamental computational intractability of first-order entailment.

In this chapter, we will explore the idea of limiting ourselves to only a certain interesting subset of first-order logic, where the Resolution procedure becomes much more manageable. We will also see that from a representation standpoint, the subset in question is still sufficiently expressive for many purposes.

5.1 Horn clauses

In a Resolution-based system, clauses end up being used for two different purposes. First, they are used to express ordinary disjunctions like

$$[\mathsf{Rain}, \mathsf{Sleet}, \mathsf{Snow}].$$

This is the sort of clause we might use to express incomplete knowledge: there is rain or sleet or snow outside, but we don't know which. But consider a clause like

$$[\neg\mathsf{Child}, \neg\mathsf{Male}, \mathsf{Boy}].$$

While this can certainly be read as a disjunction, namely, "either someone is not a child, or is not male, or is a boy," it is much more naturally understood as a *conditional*: "if someone is a child and is male then that someone is a boy." It is this second reading of clauses that will be our focus in this chapter.

We call a clause like the above—containing at most one positive literal—a *Horn clause*. When there is exactly one positive literal, the clause is called a *positive* (or *definite*) Horn clause. When there are no positive literals, the clause is called a *negative* Horn clause. In either case, there can be zero negative literals, and so the empty clause is a negative Horn clause. Observe that a positive Horn clause $[\neg p_1, \ldots, \neg p_n, q]$ can be read as "if p_1 and ... and p_n, then q." We will sometimes write a clause like this as

$$p_1 \wedge \ldots \wedge p_n \;\Rightarrow\; q$$

to emphasize this conditional, "if-then" reading.

Our focus in this chapter will be on using Resolution to reason with if-then statements (which are sometimes called "rules"). Full first-order logic is concerned with disjunction and incomplete knowledge in a more general form that we are putting aside for the purposes of this chapter.

5.1.1 Resolution derivations with Horn clauses

Given a Resolution derivation over Horn clauses, observe that two negative clauses can never be resolved together, since all of their literals are of the same polarity. If we are able to resolve a negative and a positive clause together, we are guaranteed to produce a negative clause: the two clauses must be resolved with respect to the one positive literal in the positive clause, and so it will not appear in the resolvent. Similarly, if we resolve two positive clauses together, we are guaranteed to produce a positive clause: the two clauses must be resolved with respect to one (and only one) of the positive literals, so the other positive literal will appear in the resolvent. In other words, Resolution over Horn clauses must always involve a positive clause, and if the second clause is negative, the resolvent is negative; if the second clause is positive, the resolvent is positive.

Less obvious, perhaps, is the following fact: suppose S is a set of Horn clauses and $S \vdash c$, where c is a negative clause. Then there is guaranteed to be a derivation of c where all the new clauses in the derivation (i.e., clauses not in S) are negative. The proof is somewhat laborious, but the main idea is this: suppose we have a derivation with some new positive clauses. Take the last one of these, and call it c'. Since c' is the last positive clause in the derivation, all of the Resolution steps after c' produce negative clauses. We now change the derivation so that instead of generating negative clauses using c', we generate these negative clauses using the positive parents of c' (which is where all of the literals in c' come from—c'

must have only positive parents, since it is a positive clause). We know we can do this because in order to get to the negative successor(s) of c', we must have a clause somewhere that can resolve with it to eliminate the one positive literal in c' (call that clause d and the literal p). That p must be present in one of the (positive) parents of c'; so we just use clause d to resolve against the parent of c', thereby eliminating p earlier in the derivation, and producing the negative clauses without producing c'. The derivation still generates c, but this time without c'. If we repeat this for every new positive clause introduced, we eliminate all of them.

We can go further: suppose S is a set of Horn clauses and $S \vdash c$, where c is again a negative clause. Then there is guaranteed to be a derivation of c where each new clause derived is not only negative, but is a resolvent of the previous one in the derivation and an original clause in S. The reason is this: by the above argument, we can assume that each new clause in the derivation is negative. This means that it has one positive and one negative parent. Clearly, the positive parent must be from the original set (since all the new ones are negative). Each new clause then has exactly one negative parent. So starting with c, we can work our way back through its negative ancestors, and end up with a negative clause that is in S. Then by discarding all the clauses that are not on this chain from c to S, we end up with a derivation of the required form.

These observations lead us to the following conclusion:

> There is a derivation of a negative clause (including the empty clause) from a set of Horn clauses S iff there is one where each new clause in the derivation is a negative resolvent of the previous clause in the derivation and some element of S.

We will look at derivations of this form in more detail in the next section.

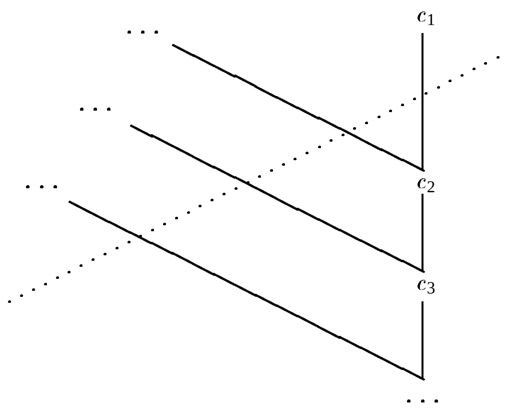
5.2 SLD Resolution

The observations of the previous section lead us to consider a very restricted form of Resolution that is sufficient for Horn clauses. This is a form of Resolution where each new clause introduced is a resolvent of the previous clause and a clause from the original set. This pattern showed up repeatedly in the examples of the previous chapter,[1] and is illustrated schematically in Figure 5.1.

Let us be a little more formal about this. For any set S of clauses (Horn or not), an *SLD derivation* of a clause c from S is a sequence of clauses c_1, c_2, ..., c_n, such that $c_n = c$, $c_1 \in S$, and c_{i+1} is a resolvent of c_i and some clause of S. We

[1]The pattern appears in Figure 4.4 of the previous chapter, but not Figure 4.5

Figure 5.1: The SLD Resolution pattern



write $S \vdash_{\text{SLD}} c$ if there is an SLD derivation of c from S. Notationally, because of its structure, an SLD derivation is simply a type of Resolution derivation where we do not explicitly mention the elements of S except for c_1.[2] We know that at each step of the way, the obvious positive parent from S can be identified, so we can leave it out of our description of the derivation, and just show the chain of negative clauses from c_1 to c.

In the general case, it should be clear that if $S \vdash_{\text{SLD}} []$ then $S \vdash []$. The converse, however, is not true in general. For example, let S be the set of clauses $[p, q]$, $[\neg p, q]$, $[p, \neg q]$, and $[\neg p, \neg q]$. A quick glance at these clauses should convince us that S is unsatisfiable (whatever values we pick for p and q, we cannot make all four clauses true at the same time). Therefore, $S \vdash []$. However, to generate [] by Resolution, the last step must involve two complementary unit clauses $[\rho]$ and $[\overline{\rho}]$, for some atom ρ. Since S contains no unit clauses, it will not be possible to use an element of S for this last step. Consequently there is no SLD derivation of [] from S, even though $S \vdash []$.

In the previous section we argued that for Horn clauses, we could get by with Resolution derivations of a certain shape, wherein each new clause in the derivation was a negative resolvent of the previous clause in the derivation and some element

[2]The name SLD stands for *S*elected literals, *L*inear pattern, over *D*efinite clauses.

of S; we have now called such derivations SLD derivations. So while not the case for Resolution in general, it is the case that if S is a set of Horn clauses, then $S \vdash []$ iff $S \vdash_{\text{SLD}} []$. So if S is Horn, then it is unsatisfiable iff $S \vdash_{\text{SLD}} []$. Moreover, we know that each of the new clauses $c_2, \ldots, c_n$ can be assumed to be negative. So c_2 has a negative and a positive parent, and thus $c_1 \in S$ can be taken to be negative as well. Thus in the Horn case, SLD derivations of the empty clause must begin with a negative clause in the original set.

To see an example of an SLD derivation, consider the first example of the previous chapter. We start with a KB containing the following positive Horn clauses:

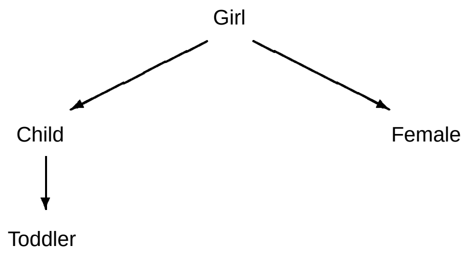
Toddler
Toddler ⊃ Child
Child ∧ Male ⊃ Boy
Infant ⊃ Child
Child ∧ Female ⊃ Girl
Female

and wish to show that KB ⊨ Girl, that is, that there is an SLD derivation of [] from KB together with the negative Horn clause [¬Girl]. Since this is the only negative clause, it must be the c_1 in the derivation. By resolving it with the fifth clause in the KB, we get [¬Child, ¬Female] as c_2. Resolving this with the sixth clause, we get [¬Child] as c_3. Resolving this with the second clause, we get [¬Toddler] as c_4. And finally, resolving this with the first clause, we get [] as the final clause. Observe that all the clauses in the derivation are negative. To display this derivation, we could continue to use Resolution diagrams from the previous chapter. However, for SLD derivations, it is convenient to use a special-purpose terminology and format.

5.2.1 Goal trees

All the literals in all the clauses in a Horn SLD derivation of the empty clause are negative. We are looking for positive clauses in the KB to "eliminate" these negative literals to produce the empty clause. Sometimes, there is a unit clause in the KB that eliminates the literal directly. For example, if a clause like [¬Toddler] appears in the derivation, then the derivation is finished, since there is a positive clause in the KB that resolves with it to produce the empty clause. We say in this case that the *goal* Toddler is *solved*. Sometimes there is a positive clause that eliminates the literal but introduces other negative literals. For example, with a clause like [¬Child] in the derivation, we continue with the clause [¬Toddler], having resolved it against the second clause in our knowledge base ([¬Toddler, Child]). We say in this case that the goal Child *reduces to* the subgoal Toddler. Similarly, the goal Girl reduces

Figure 5.2: An example goal tree



to two subgoals: Child and Female, since two negative literals are introduced when it is resolved against the fifth clause in the KB.

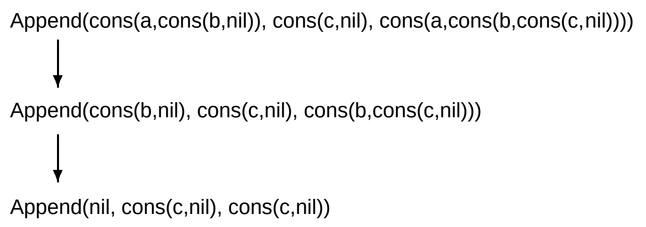
So a restatement of the SLD derivation is as follows: we start with the goal Girl. This reduces to two subgoals, Child and Female. The goal Female is solved, and Child reduces to Toddler. Finally, Toddler is solved.

We can display this derivation using what is called a *goal tree*. We draw the original goal (or goals) at the top, and point from there to the subgoals. For a complete SLD derivation, the leaves of the tree (at the bottom) will be the goals that are solved (see Figure 5.2). This allows us to easily see the form of the argument: we want to show that Girl is entailed by the KB. Reading from the bottom up, we know that Toddler is entailed since it appears in the KB. This means that Child is entailed. Furthermore, Female is also entailed (since it appears in the KB), so we conclude that Girl is entailed.

This way of looking at Horn clauses and SLD derivations, when generalized to deal with variables in the obvious way, forms the basis of the programming language Prolog. We already saw an example of a Prolog style-definition of addition in the previous chapter. Let us consider an additional example involving lists. For our purposes, list terms will either be variables, the constant nil, or a term of the form cons(t_1, t_2) where t_1 is any term and t_2 is a list term. We will write clauses defining the Append(x, y, z) relation, intended to hold when list z is the result of appending list y to list x:

Append(nil, y, y)
Append(x, y, z) $\Rightarrow$ Append(cons(w, x), y, cons(w, z))

Figure 5.3: A goal tree for append


Append(cons(a,cons(b,nil)), cons(c,nil), cons(a,cons(b,cons(c,nil))))
Append(cons(b,nil), cons(c,nil), cons(b,cons(c,nil)))
Append(nil, cons(c,nil), cons(c,nil))

If we wish to show that this entails

Append(cons(a,cons(b,nil)), cons(c,nil), cons(a,cons(b,cons(c,nil))))

we get the goal tree in Figure 5.3. We can also use a variable in the goal and show that the definition entails $\exists u$.Append(cons(a,cons(b,nil)), cons(c,nil), u). The answer u = cons(a,cons(b,cons(c,nil))) can be extracted from the derivation directly. Unlike with general Resolution, it is not necessary to use answer predicates with SLD derivations. This is because if S is a set of Horn clauses, then $S \models \exists x.\alpha$ iff for some term t, $S \models \alpha^x_t$.

5.3 Computing SLD derivations

We now turn our attention to procedures for reasoning with Horn clauses. The idea is that we are given a KB containing a set of positive Horn clauses representing if-then sentences, and we wish to know whether or not some atom (or set of atoms) is entailed. Equivalently, we wish to know whether or not the KB together with the clause consisting of one or more negative literals is unsatisfiable. Thus the typical case, and the one we will consider here, involves determining the satisfiability of a set of Horn clauses containing exactly one negative clause.[3]

[3]It is not hard to generalize the procedures presented here to deal with more than one negative clause. Similarly, the procedures can be generalized to answer entailment questions where the query is an arbitrary (non-Horn) formula in CNF.

Figure 5.4: A recursive back-chaining SLD procedure

Input: a finite list of atomic sentences, $q_1, \ldots, q_n$
Output: yes or no according to whether a given KB entails all of the q_i

```
SOLVE[$q_1, \ldots, q_n$] =
    If $n = 0$ then return yes
    For each clause $c \in$ KB, do
        If $c = [q_1, \neg p_1, \ldots, \neg p_m]$
            and SOLVE[$p_1, \ldots, p_m, q_2, \ldots, q_n$]
        then return yes
    end for
    Return no
```

5.3.1 Back-chaining

A procedure for determining the satisfiability of a set of Horn clauses with exactly one negative clause is presented in Figure 5.4. This procedure starts with a set of goals as input (corresponding to the atoms in the single negative clause) and attempts to solve them. If there are no goals, then it is done. Otherwise, it takes the first goal q_1 and looks for a clause in KB whose positive literal is q_1. Using the negative literals in that clause as subgoals, it then calls itself recursively with these subgoals together with the rest of the original goals. If this is successful, it is done; otherwise it must consider other clauses in the KB whose positive literal is q_1. If none can be found, the procedure returns no, meaning the atoms are not entailed.

This procedure is called *back-chaining* since it works backwards from goals to facts in the KB. It is also called *depth-first* since it attempts to solve the new goals p_i before tackling the old goals q_i. Finally, it is called *left-to-right* since it attempts the goals q_i in order $1, 2, 3$, *etc*. This depth-first left-to-right back-chaining procedure is the one normally used by Prolog implementations to solve goals, although the first-order case obviously requires unification, substitution of variables and so on.

This back-chaining procedure also has a number of drawbacks. First, observe that even in the propositional case it can go into an infinite loop. Suppose we have the tautologous $[p, \neg p]$ in the KB.[4] In this case, a goal of p can reduce to a subgoal of p, and so on, indefinitely.

Even if it does terminate, the back-chaining algorithm can be quite inefficient,

[4]This corresponds to the Prolog program "`p :- p.`"

and do a considerable amount of redundant searching. For example, imagine that we have $2n$ atoms $p_0, \ldots p_{n-1}$ and $q_0, \ldots, q_{n-1}$, and the following $4n - 4$ clauses: for $0 < i < n$,

$$\begin{aligned} p_{i-1} &\Rightarrow p_i \\ p_{i-1} &\Rightarrow q_i \\ q_{i-1} &\Rightarrow p_i \\ q_{i-1} &\Rightarrow q_i \end{aligned}$$

For any i, both SOLVE[p_i] and SOLVE[q_i] will eventually fail, but only after at least 2^i steps. The proof is a simple induction argument.[5] This means that even for a reasonably sized KB (say 396 clauses when $n = 100$), an impossibly large amount of work may be required (over 2^{100} steps).

Given this exponential behaviour, we might wonder if this is a problem with the back-chaining procedure, or another instance of what we saw in the last chapter where the entailment problem itself was simply too hard in its most general form. As it turns out, this time it is the procedure that is to blame.

5.3.2 Forward-chaining

In the propositional case, there is a much more efficient procedure to determine if a Horn KB entails a set of atoms, given in Figure 5.5. This is a *forward-chaining* procedure since it works from the facts in the KB towards the goals. The idea is to mark atoms as "solved" as soon as we have determined that they are entailed by the KB.

Suppose, for example, we start with the Girl example of above. At the outset Girl is not marked as solved, so we go to step 2. At this point, we look for a clause satisfying the given criteria. The clause [Toddler] is one such since all of its negative literals (of which there are none) are marked as solved. So we mark Toddler as solved, and try again. This time we might find the clause [Child, ¬Toddler], and so we can mark Child as solved, and try again. Continuing in this way, we mark Female and finally Girl as solved and we are done.

While this procedure appears to take about the same effort as the back-chaining one, it has much better overall behaviour. Note, in particular, that each time through the iteration we need to find a clause in the KB with an atom that has not been marked. Thus, we will iterate at most as many times as there are clauses in the KB. Each such iteration step may require us to scan the entire KB, but the overall result

[5]The claim is clearly true for $i = 0$. For the goal p_k, where $k > 0$, we need to try to solve both p_{k-1} and q_{k-1}. By induction, each of these take at least 2^{k-1} steps, for a total of 2^k steps. The case for q_k is identical.

Figure 5.5: A forward-chaining SLD procedure

Input: a finite list of atomic sentences, $q_1, \ldots, q_n$
Output: yes or no according to whether a given KB entails all of the q_i

1. if all of the goals q_i are marked as solved, then return yes
2. check if there is a clause $[p, \neg p_1, \ldots, \neg p_n]$ in KB, such that all of its negative atoms $p_1, \ldots, p_n$ are marked as solved, and such that the positive atom p is not marked as solved
3. if there is such a clause, mark p as solved and go to step 1
4. otherwise, return no

will never be exponential. In fact, with a bit of care in the use of data structures, a forward-chaining procedure like this can be made to run in time that is *linear* in the size of the KB.

5.3.3 The first-order case

Thus, in the propositional case at least, we can determine if a Horn KB entails an atom in a linear number of steps. But what about the first-order case? Unfortunately, even with Horn clauses, we still have the possibility of a procedure that runs forever. The example in Figure 4.11 of the previous chapter where an infinite branch of resolvents was generated only required Horn clauses. While it might seem that a forward-chaining procedure could deal with first-order examples like these, avoiding the infinite loops, this cannot be: the problem of determining whether a set of first-order Horn clauses entails an atom remains undecidable. So no procedure can be guaranteed to always work, despite the fact that the propositional case is so easy. This is not too surprising since Prolog is a full programming language, and being able to decide if an atom is entailed would imply being able to decide if a Prolog program would halt.

As with non-Horn clauses, the best that can be expected is to give control of the reasoning to the user to help avoid redundancies and infinite branches. Unlike the non-Horn case, Horn clauses are much easier to structure and control in this way. In the next chapter, we will see some examples of how this is done.

5.4 Bibliographic notes

5.5 Exercises

Chapter 6

Procedural Control of Reasoning

Theorem-proving methods, like Resolution, are general, domain-independent ways of reasoning. A user can express facts in full FOL without having to know *how* this knowledge will ultimately be used for inference by an automated theorem-proving procedure. The ATP mechanism will try all logically permissible uses of everything in the knowledge base in looking for an answer to a query.

This is a double-edged sword, however. Sometimes, it is not computationally feasible to try all logically possible ways of using what is known. Furthermore, we often do have an idea about how knowledge should be used or how to go about searching for a derivation. When we understand the structure of a domain or a problem, we may want to avoid using facts in every possible way or in every possible order. In cases like these, we would like to communicate *guidance* to an automatic theorem-proving procedure based on properties of the domain. This may be in the form of specific methods to use, or perhaps merely what to avoid in trying to answer a query.

For example, consider a variant on a logical language where some of the connectives are to be used only in one direction, as suggested at the end of Chapter 4. Instead of a simple implication symbol, for example, we might have a forward implication symbol that suggests only going from antecedent to consequent, but not the reverse. If we used the symbol, "$\rightarrow$," to represent this one-way implication, then the sentence, $(\mathsf{Battleship}(x) \rightarrow \mathsf{Gray}(x))$, would allow a system to conclude in the forward direction for any specific battleship that it was gray, but would prevent it from trying to show that something was gray by trying to show that it was a battleship (an unlikely prospect for most gray things).

More generally, there are many cases in knowledge representation where we as users will want to *control the reasoning process* in various domain-specific ways.

As noted in Chapter 4, this is is often the best we can do to deal with an otherwise computationally intractable reasoning task. In this chapter, we will examine how knowledge can be expressed to provide control for the simple case of the back-chaining reasoning procedure we examined in Chapter 5.

6.1 Facts and rules

In a clausal representation scheme like those we considered in the chapter on Horn logic, we can often separate the clauses in a KB into two components: a database of *facts*, and a collection of *rules*. The facts are used to cover the basic truths of the domain, and are usually ground atomic wffs; the rules are used to extend the vocabulary, expressing new relations in terms of basic facts, and are usually universally quantified conditionals. Both the basic facts and the (conclusions of) rules can be retrieved by the sort of unification matching we have studied.

For example, we might have the following simple knowledge base fragment:

Mother(jane, billy)
Father(john, billy)
Father(sam, john)
...
Parent$(x, y) \Leftarrow$ Mother(x, y)
Parent$(x, y) \Leftarrow$ Father(x, y)
Child$(x, y) \Leftarrow$ Parent(y, x)
...

We can read the latter sentence, for example, as "x is a child of y if y is a parent of x." In this case, if we ask the knowledge base if John is the father of Billy, we would find the answer by matching the base fact, Father(john, billy), directly. If we ask if John is a parent of Billy, then we would need to chain backward and ask the KB if John was either the mother of Billy or the father of Billy (the latter would of course succeed). If we were to ask whether Billy is a child of John, then we would have to check whether John was a parent of Billy, and then proceed to the mother and father checks.

Because rules involve chaining, and the possible invocation of other rules which can in turn cause more chaining, the key control issue we need to think about is how to make the most effective use of the rules in a knowledge base.

6.2 Rule formation and search strategy

Let's consider defining the notion of Ancestor in terms of the predicate Parent. Here are three logically equivalent ways to express the relationship between the two predicates:

1. Ancestor$(x, y) \Leftarrow$ Parent(x, y)
   Ancestor$(x, y) \Leftarrow$ Parent$(x, z) \wedge$ Ancestor(z, y)

2. Ancestor$(x, y) \Leftarrow$ Parent(x, y)
   Ancestor$(x, y) \Leftarrow$ Parent$(z, y) \wedge$ Ancestor(x, z)

3. Ancestor$(x, y) \Leftarrow$ Parent(x, y)
   Ancestor$(x, y) \Leftarrow$ Ancestor$(x, z) \wedge$ Ancestor(z, y)

In the first case, we see that someone x is an ancestor of someone else y if x is a parent of y, or if there is a third person z who is a child of x and an ancestor of y. So, for example, if Sam is the father of Bill, and Bill is the great-grandfather (an ancestor) of Sue, then Sam is an ancestor of Sue. The second case looks at the situation where Sam might be the great-grandfather of Fred, who is a parent of Sue, and therefore Sam is an ancestor of Sue. In the third case, we observe that if Sam is the great-grandfather of George who is in turn a grandfather of Sue, then again Sam is an ancestor of Sue. While their forms are different, a close look reveals that all three of these yield the same results on all questions.

If we are trying to determine whether or not someone is an ancestor of someone else, in all three cases we would use back-chaining from an initial Ancestor goal, such as Ancestor(sam,sue), which would ultimately reduce to a set of Parent goals. But depending on which version we use, the rules could lead to substantially different amounts of computation. Consider the three cases:

1. the first version of Ancestor above suggests that we start from Sam and look "downward" in the family tree; in other words (assuming that Sam is not Sue's parent), to find out whether or not Ancestor(sam, sue) is true, we first look for a z that is Sam's child: Parent(sam, z). We then check to see if that z is an ancestor of Sue: Ancestor(z, sue).

2. the second option (again, assuming that Sam is not Sue's parent) suggests that we start searching "upward" in the family tree from Sue, looking for some z that is Sue's parent: Parent(z, sue). Once we find one, we then check to see if Sam is an ancestor of that parent: Ancestor(sam, z).

3. the third option suggests a search in both directions, looking at individual Parent relationships both up and down at the same time.

The three search strategies implied by these (logically equivalent) representations are not equivalent, at least in terms of the computational resources needed to answer the query. For example, suppose that people have on average 1 child, but 2 parents. With the first option, as we fan out from Sam, we search a tree downward that has about d nodes where d is the depth of the search; with the second option, as we fan out from Sue, we search a tree upward that has 2^d nodes where d is the depth. So as d gets larger, we can see that the first option would require much less searching. If, on the other hand, people had more than 2 children on average, the second option would be better. Thus we can see how the structure of a particular domain, or even a particular problem, can make logically equivalent characterizations of the rules quite different in their computational impact for a back-chaining derivation procedure.

6.3 Algorithm Design

The same kind of thinking about the structure of rules plays a significant role in a wide variety of problems. For example, familiar numerical relations can be expressed in forms that are logically equivalent, but with substantially different computational properties.

Consider the Fibonacci integer series, wherein each Fibonacci number is the sum of the previous two numbers in the series. Assuming that the first two Fibonacci numbers are 1 and 1, the series looks like this:

1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

One direct and obvious way to characterize this series is with the following two base facts and a rule, using a two-place predicate, $\mathsf{Fibo}(n, v)$, intended to hold when v is the n^{th} Fibonacci number:

$$\begin{array}{l}\mathsf{Fibo}(0,1)\\ \mathsf{Fibo}(1,1)\\ \mathsf{Fibo}(\mathsf{s}(\mathsf{s}(n)),v) \Leftarrow \mathsf{Fibo}(n,y) \wedge \mathsf{Fibo}(\mathsf{s}(n),z) \wedge \mathsf{Plus}(y,z,v)\end{array}$$

This says explicitly that the zeroth and first Fibonacci numbers are both 1, and by the rule, that the $(n+2)^{\text{nd}}$ Fibonacci number is the sum of the $(n+1)^{\text{st}}$ Fibonacci number z and the n^{th} Fibonacci number y. Note that we use a three-place relation for addition: $\mathsf{Plus}(y, z, v)$ means $v = y + z$.

This simple, elegant characterization has significant computational drawbacks if used by an unguided back-chaining theorem prover. In particular, it generates an *exponential* number of Plus subgoals. This is because each application of the rule calls Fibo twice, once each on the previous two numbers in the series. Most of this effort is redundant since the call on the previous number makes a further call on the number before that—which has already been pursued in a different part of the proof tree by the former step. That is, $\mathsf{Fibo}(12, -)$ invokes $\mathsf{Fibo}(11, -)$ and $\mathsf{Fibo}(10, -)$; the call to $\mathsf{Fibo}(11, -)$ then calls $\mathsf{Fibo}(10, -)$ again. The resulting exponential behaviour makes it virtually impossible to calculate the 100th Fibonacci number using these clauses.

An alternative (but still recursive) view of the Fibonacci series uses a four-place intermediate predicate, F. The definition is this:

$$\begin{array}{l}\mathsf{Fibo}(n,v) \Leftarrow \mathsf{F}(n,1,0,v)\\ \mathsf{F}(0,y,z,y)\\ \mathsf{F}(\mathsf{s}(n),y,z,v) \Leftarrow \mathsf{Plus}(y,z,s) \wedge \mathsf{F}(n,s,y,v)\end{array}$$

Here, $\mathsf{F}(n, y, z, v)$ will count down from n using y to keep track of the current Fibonacci number, and z to keep track of the previous one before that. Each time we reduce n by 1, we get a new current number (the sum of the current and previous Fibonacci numbers), and we get a new previous number (which was the current one). At the end, when n is 0, the final result v is the current Fibonacci number y.[1] The important point about this equivalent characterization is that it avoids the redundancy of the previous version and requires only a *linear* number of Plus subgoals. Calculating the 100th Fibonacci number in this case is quite straightforward.

So in a sense, looking for computationally feasible ways of expressing definitions of predicates using rules is not so different from looking for efficient algorithms for computational tasks.

6.4 Ordering Goals

When using rules to do backchaining, we can try to solve subgoals in any order; all orderings of subgoals are logically permissible. But as we have seen in the previous sections, the computational consequences of logically equivalent representations can be significant.

Consider this simple example:

[1]To prove that $\mathrm{F}(n, 1, 0, v)$ holds when v is the n^{th} Fibonacci number, we show by induction on n that $\mathrm{F}(n, y, z, v)$ holds iff v is the sum of y times the n^{th} Fibonacci number and z times the $(n-1)^{\text{st}}$ Fibonacci number.

$$\text{AmericanCousin}(x, y) \Leftarrow \text{American}(x) \wedge \text{Cousin}(x, y)$$

If we are trying to ascertain the truth of AmericanCousin(fred, sally), there is not much difference between choosing to solve the first subgoal (American(fred)) or the second subgoal (Cousin(fred, sally)) first. However, there is a *big* difference if we are looking for an American cousin of Sally: AmericanCousin(x, sally). Our two options are then,

1. find an American and then check to see if she is a cousin of Sally; or

2. find a cousin of Sally and then check to see if she is an American.

Unless Sally has a *lot* of cousins (more than several hundred million), the second method will be much better than the first.

This illustrates the potential importance of ordering goals. We might think of the two parts of the definition above as suggesting that when we want to generate Sally's American cousins, what we want to do is to *generate* Sally's cousins one at a time, and *test* to see if each is an American. Languages like PROLOG, which are used for programming and not just general theorem-proving, take ordering seriously, both of clauses and of the literals within them. In PROLOG notation,

```
G :- G₁, G₂, ..., Gₙ.
```

stands for

$$G \Leftarrow G_1 \wedge G_2 \wedge \ldots \wedge G_n$$

but goals are attempted exactly in the presented order.

6.5 Committing to proof methods

An appropriate PROLOG rendition of our American cousin case would take care of the inefficiency problem we pointed out above:

```
americanCousin(X,Y) :- cousin(X,Y), american(X).
```

In a construct like this, we need to allow for goal backtracking, since for a goal of, say, AmericanCousin(x, sally), we may need to try American(x) for various values of x. In other words, we may need to generate many cousin candidates before we find one that is American.

But sometimes, given a clause of the form

```
G :- T, S.
```

goal T is needed only as a *test* for the applicability of subgoal S, and not as a generator of possibilities for subgoal S to test further. In other words, if T succeeds, then we want to *commit* to S as the appropriate way of achieving goal G. So, if S were then to fail, we would consider goal G as having failed. A consequence of this is that we would not look for other ways of solving T, nor would we look for other clauses with G as the head.

In PROLOG, this type of test/fail control is specified with the "cut" symbol, '`!`'. Notationally, we would have a PROLOG clause that looks like this:

```
G:- T₁, T₂, ..., Tₘ, !, G₁, G₂, ..., Gₙ.
```

which would tell the interpreter to try each of the goals in this exact order, but if all the T_i succeed, to commit to the G_i as the only way of solving G.

A clear application of this construct is in the if-then-else construct of traditional programming languages. Consider, for example, defining a predicate Expt(a, n, v) intended to hold when $v = a^n$. The obvious way of calculating a^n (or reasoning about Expt goals) requires n multiplications. However, there is a much more efficient recursive method that only requires about $\log_2(n)$ multiplications: if n is even, we continue recursively with a^2 and $n/2$ replacing a and n, respectively; otherwise, if n is odd, we continue recursively with a^2 and $(n-1)/2$ and then multiply the result by a. In other words, we are imagining a recursive procedure with an if-then-else of the form

```
**if** $n$ is even
    **then** do one thing
    **else** do another
```

The details need not concern us, except to note the form of the clauses we would use to define the predicate:

$$\text{Expt}(a, 0, 1)$$
$$\text{Expt}(a, n, v) \Leftarrow n > 0 \wedge \text{Even}(n) \wedge \text{Expt}(a^2, n/2, v)$$
$$\text{Expt}(a, n, v) \Leftarrow n > 0 \wedge \neg\text{Even}(n) \wedge \text{Expt}(a^2, (n-1)/2, v') \wedge v = av'$$

The point of this example is that we need to use slightly different methods based on whether n is even or odd. However, we would much prefer to test whether n is even only once: we should attempt the goal Even(n) and if it succeeds do one thing, and if it fails do another. The goal ¬Even(n) should in reality never be considered. A related but less serious consideration is the test for $n = 0$: if $n = 0$ we should commit to the first clause; we should not have to confirm that $n > 0$ in the other two clauses.

In PROLOG both of these concerns can be handled with the cut operator. We would end up with a PROLOG definition like this:

```
expt(A,0,V) :- !, V=1.
expt(A,N,V) :- even(N), !, ...what to do when n is even.
expt(A,N,V) :- ...what to do when n is odd.
```

Note that we *commit* to the first clause when $n = 0$ regardless of the value of a or v, but we only *succeed* when $v = 1$. Thus, while

```
expt(A,N,1) :- N=0, !, V=1.
```

is correct and equivalent to the first clause,

```
expt(A,0,1) :- !.
```

would be incorrect. In general, we can see that something like

```
G :- P, !, R.
G :- S.
```

is logically equivalent to "if P holds then R implies G, and if $\neg P$ holds then S implies G," but that it only considers the P once.

A less algorithmic example of the use of the cut operator might be to define a NumberOfParents predicate: for Adam and Eve, the number of parents is 0, but for everyone else, it is 2:

```
numberOfParents(adam,V) :- !, V=0.
numberOfParents(eve,V) :- !, V=0.
numberOfParents(P,2).
```

In this case, we do not need to confirm in the third clause that the person in question is not Adam or Eve.

6.6 Controlling Backtracking

Another application of the PROLOG cut operator involves control of backtracking on failure. At certain points in a proof, we can have an idea of which steps might be fruitful and which steps will come to nothing and waste resources in the process.

Imagine for example, that we are trying to show that Jane is an American cousin of Billy. Two individuals can be considered to be (first) cousins if they share a grandparent but are not siblings:

$$\text{Cousin}(x,y) \Leftarrow (x \neq y) \wedge \neg\text{Sibling}(x,y) \wedge \\ \text{GParent}(z,x) \wedge \text{GParent}(z,y)$$

Suppose that in trying to show that Jane is an American cousin of Billy, we find that Henry is a grandparent of both of them, but that Jane is not American. The question is what happens now. If it turns out that Elizabeth is also a grandparent of both Jane and Billy, we will find this second z on backtracking, and end up testing whether Jane is American a second time. This will of course fail once more since nothing has changed.

What this example shows is that on failure, we need to avoid trying to redo a goal that was not part of the reason we are failing. It was not the choice of grandparent that caused the trouble here, so there is no point in reconsidering it. Yet this is precisely what PROLOG backtracking would do.[2] To get the effect we want in PROLOG, we would need to represent our goal as

```
cousin(jane,billy), !, american(jane)
```

In other words, once we have found a way to show that Jane is a cousin of Billy (no matter how), we should commit to whatever result comes out of checking that she is American.

As a second example of controlling backtracking, consider the following definition of membership in a list:

$$\text{Member}(x,l) \Leftarrow \text{FirstElement}(x,l)$$
$$\text{Member}(x,l) \Leftarrow \text{RemainingElements}(l,l') \wedge \text{Member}(x,l')$$

with the auxiliary predicates FirstElement and RemainingElements defined in the obvious way. Now imagine that we are trying to establish that some object a is an element of some (large) list c and has property Q. That is, we have the goal

$$\text{Member}(a,c) \wedge Q(a).$$

If the Member(a,c) subgoal were to succeed but $Q(a)$ fail, it would be silly to reconsider Member(a,c) to see if a also occurs later in the list. In PROLOG, we can control this by using the goal

```
member(a,C), !, q(a).
```

More generally, if we know that the Member predicate will only be used to test for membership in a list (and not to generate elements of a list), we can use a PROLOG definition like this:

[2]A more careful but time-consuming version of backtracking (called *dependency-directed* backtracking) avoids the redundant steps here automatically.

```
member(X,L) :- firstElement(X,L), !.
member(X,L) :- remainingElements(L,L1),
                  member(X,L1).
```

This guarantees that once a membership goal succeeds (in the first clause) by finding a sublist whose first element is the item in question, the second clause, which looks farther down the list, will never be reconsidered on failure of a later goal. For example, if we had a list of our friends and some goal needed to check that someone (e.g., George) was both a friend and rich, we could simply write

```
member(george,Friends), rich(george)
```

without having to worry about including a cut. The definition of `member` assures us that once an element is found in the list, if a subsequent test like `rich` fails, we won't go back to see if that element occurs somewhere later in the list and try that failed test again.

6.7 Negation as Failure

Procedurally, we can distinguish between two types of "negative" situations with respect to a goal G:

- being able to solve the goal $\neg G$; or
- being unable to solve the goal G.

In the latter case, we may not be able to find a fact or rule in the KB asserting that G is false, but we run out of options in trying to show that G is true.

There are a number of situations where it is useful to build constructively on the failure to prove a goal. We begin by introducing a new type of goal **not**(G), which is understood to succeed when the goal G fails, and to fail when the goal G succeeds (quite apart from the status of $\neg G$). In PROLOG, **not** behaves as if it were defined roughly like this:

```
not(G) :- G, !, fail.        % fail if G succeeds
not(G).                      % otherwise succeed
```

This type of *negation as failure* is only useful when failure is *finite*. If attempting to prove G results in an infinite branch with an infinite set of resolvents to try, then we cannot expect a goal of **not**(G) to terminate either. However, if there are no more resolvents to try in a proof, then **not**(G) will succeed.

Negation as failure is especially useful in situations where the the collection of facts and the rules express complete knowledge about some predicate. If, for example, we have an *entire* family represented in a KB, we could define in PROLOG

```
noChildren(X) :- not(parent(X,Y)).
```

We know that someone has no children if we cannot find any in the database. With incomplete knowledge, on the other hand, we could fail to find any children in the database simply because we have not yet been told of any.

Another situation where negation as failure is useful is when we have a complete method for computing the complement of a predicate we care about. For example, if we have a rule for determining if a number is prime, we would not need to construct another one to show that a number is not prime; instead we can use negation as failure:

```
composite(N) :- N > 1, not(primeNumber(N)).
```

In this case, failure to prove that a number greater than 1 is prime is sufficient to conclude that the number is composite.

Declaratively, **not** has the same reading as conventional negation, except when new variables appear in the goal. For example, the PROLOG clause for Composite above can be read as saying that

> for every number n, if $n > 1$ and n is not a prime number,
> then n is composite.

However, the clause for NoChildren before that should *not* be read as saying that

> for every x and y, if x is not a parent of y, then x has no children.

For example, suppose that the goal Parent(sue, jim) succeeds, but that the goal Parent(sue, george) fails. Although we do want to conclude that Sue is not a parent of George, we do not want to conclude that she has no children. Logically, the rule needs to be read as

> for every x, if for every y, x is not a parent of y, then x has no children.

Note that the quantifier for the new variable y in the goal has moved inside the scope of the "if."

6.8 Dynamic databases

In this chapter we have considered a KB consisting of a collection of ground atomic facts about the world and universally quantified rules defining new predicates. Because our most basic knowledge is expressed by the elementary facts, we can think of them as a *database* representing a snapshot of the world. It is natural, then, as properties of the world change over time, to think of reflecting these changes with additions and deletions in the database. The removed facts are a reflection of things that are no longer true, and the added facts are a reflection of things that have newly become true.

With this more dynamic view of the database, it is useful to consider three different procedural interpretations for a basic rule like $\mathsf{Parent}(x, y) \Leftarrow \mathsf{Mother}(x, y)$:

1. *if-needed:* whenever we have a goal matching $\mathsf{Parent}(x, y)$, we can solve it by solving $\mathsf{Mother}(x, y)$. This is ordinary back chaining. Procedurally, we wait to make the connection between mothers and parents until we need to prove something about parents.

2. *if-added:* whenever a fact matching $\mathsf{Mother}(x, y)$ is added to the database, we also add $\mathsf{Parent}(x, y)$ to the database. This is forward-chaining. In this case, the connection between mothers and parents is made as soon as we learn about a new mother relationship. A proof of a parent relationship would then be more immediate, but at the cost of the space needed to store facts that may never be used.

3. *if-removed:* whenever something matching $\mathsf{Parent}(x, y)$ is removed from the database, we should also remove $\mathsf{Mother}(x, y)$. This is the dual of the *if-added* case. But there is a more subtle issue here. If the *only* reason we have a parent relationship in the database is because of the mother relationship, then if we remove that mother relationship, we should remove the parent one as well. To do this properly, we would need to keep track of *dependencies* in the database.

Interpretation (1) above is of course the mainstay of PROLOG; interpretations (2) and (3) above suggest the use of "demons," which are procedures that actively monitor the database and trigger—or "fire"—when certain conditions are met. There can be more than one such demon matching a given change to the database, and each demon may end up further changing the database, causing still more demons to fire, in a pattern of spreading activation.

6.9 The PLANNER Language

The practical implications of giving the user more direct control over the reasoning process have led over the years to the development of a set of programming languages based on ideas like the ones we have covered above. The PROLOG language is of course well known, but only covers some of these possibilities. A LISP-based language called PLANNER was invented at about the same time as PROLOG, and was designed specifically to give the user fine-grained control of a theorem-proving process.

The main ideas in PLANNER relevant to our discussion here are these:[3]

- The knowledge base of a PLANNER application is a database of simple facts, like (Mother susan john) and (Person john).

- The rules of the system are formulated as a collection of *if-needed*, *if-added*, and *if-removed* procedures, each consisting of a *pattern* for invocation (e.g., (Mother x y)) and a *body*, which is a program statement to execute once the invocation pattern is matched.

- Each program statement can succeed or fail:
  - (**goal** p), (**assert** p), and (**erase** p) specify, respectively, that a goal should be established (proven or made true), that a new fact should be added to the database, and that an old fact should be removed from the database;
  - (**and** $s_1 \ldots s_n$), where the s_i are program statements, is considered to succeed if all the s_i succeed, allowing for backtracking among them;
  - (**not** s) is negation as failure;
  - (**for** p s), perform program statement s for every way goal p succeeds;
  - (**finalize** s), similar to the PROLOG cut operator;
  - a lot more, including all of LISP.

Here is a simple PLANNER example:

```
(proc if-needed (clearTable)
     (for  (on x table)
          (and (erase (on x table))  (goal (putaway x)))))

(proc  if-removed (on x y)  (print  x "is no longer on" y))
```

[3]We are simplifying the original syntax somewhat.

The first procedure is invoked whenever the goal clearTable needs to be true, that is, in the blocks world of this example, the table needs to be free of objects. To solve this goal, for each item found on the table, we remove the statement in the database that reflects its being on the table, and solve the goal of putting that item away somewhere. We do not here show how those goals are solved, but presumably they could trigger an action by a robot arm to put the item somewhere not on the table, and subsequently to assert the new location in the database. The second procedure just alerts the user to a change in the database, printing that the item is no longer on the surface it was removed from.

The type of program considered in PLANNER suggests an interesting shift in perspective on knowledge representation and reasoning. Instead of thinking of solving a goal as proving that a condition is logically entailed by a collection of facts and rules, we think of it as *making conditions hold*, using some combination of forward and backward chaining. This is the first harbinger of the use of a representation scheme to support the execution of *plans*; hence the name of the language.[4] We also see a shift away from rules with a clear logical interpretation (as universally quantified conditionals) towards arbitrary procedures, and specifically, arbitrary operations over a database of facts. We will see this dynamic view of rules most clearly in the representation for production systems in the next chapter.

6.10 Bibliographic notes

6.11 Exercises

[4]We will reconsider the issue of planning from a logical perspective in Chapter 15.

Chapter 7

Rules in Production Systems

We have seen from our work on Horn clauses and procedural systems in previous chapters that the concept of an if-then conditional or *rule* — if P is true then Q is true — is central to knowledge representation. While the semantics of the logical formula $(P \supset Q)$ is simple and clear, it would suggest that a rule of this sort is no more than a form of disjunction: either P is false or Q is true. However, as we saw in the previous chapter, from a reasoning point of view, we can look at these rules in different ways. In particular, a rule can be understood procedurally as either

- transforming assertions of P into assertions of Q, or
- transforming goals of Q into goals of P.

We can think of these two cases this way:

$$(\textbf{assert}\ P) \Rightarrow (\textbf{assert}\ Q)$$

$$(\textbf{goal}\ Q) \Rightarrow (\textbf{goal}\ P)$$

While both of these arise from the same connection between P and Q, they emphasize the difference between focusing on assertion of facts and seeking the satisfaction of goals. We usually call the two types of reasoning that they suggest,

- *data-directed reasoning*, i.e., reasoning from P to Q, and
- *goal-directed reasoning*, i.e., reasoning from Q to P.

Data-directed reasoning might be most appropriate in a database-like setting, when assertions are made and it is important to follow the implications of those assertions. Goal-directed reasoning might be most appropriate in a problem-solving situation, where a desired result is clear, and the means to achieve that result—the logical foundations for a conclusion—are sought.

Quite separately, we can also distinguish the mechanical direction of the computation. Forward-chaining computations follow the "⇒" in the forward direction, independent of the emphasis on assertion or goal. Backward-chaining reasoning goes in the other direction. While the latter is almost always oriented toward goal-directed reasoning and the former toward data-directed reasoning, these associations are not exclusive. For example, using the notation of the previous chapter, we might imagine procedures of the following sort:

- (**proc if-added** (myGoal Q) ... (**assert** (myGoal P)) ...)
- (**proc if-needed** (myAssert P) ... (**goal** (myAssert Q)) ...)

In the former case, we use forward chaining to do a form of goal-directed reasoning: (myGoal Q) is a formula to be read as saying that Q is a goal; if this is ever asserted (that is, if we ever find out that Q is indeed a goal), we might then assert that P is also a goal. In a complementary way, the latter case illustrates a way to use backward chaining to do a form of data-directed reasoning: (myAssert P) is a formula to be read as saying that P is an assertion in the database; if this is ever a goal (that is, if we ever want to assert P in the database), we might then also have the goal of asserting Q in the database. This latter example suggests how it is possible, for example, to do data-directed reasoning in PROLOG, which is a backward-chaining system.

In the rest of this chapter, we examine a new formalism, *production systems*, used extensively in practical applications, and which emphasizes forward chaining over rules as a way of reasoning. We will see examples, however, where the reasoning is data-directed, and others where it is is goal-directed. Applications built using production systems are often called *rule-based systems* as a way of highlighting this emphasis on rules in the underlying knowledge representation.

7.1 Production Systems — Basic Operation

A *production system* is a forward-chaining reasoning system that uses rules of a certain form called *production rules* (or simply, *productions*) as its representation of general knowledge. A production system keeps an ongoing memory of assertions in what is called its *working memory* (or WM). The WM is like a database, but more volatile; it is constantly changing during the operation of the system.

A *production rule* is a two-part structure comprising an *antecedent* set of *conditions*, which if true, causes a *consequent* set of *actions* to be taken. We usually

write a rule in this form:

IF *conditions* THEN *actions*

The antecedent conditions are tests to be applied to the current state of the WM. The consequent actions are a set of actions that modify the WM.

The basic operation of a production system is a cycle of three steps that repeats until no more rules are applicable to the WM, at which point the system halts. The three parts of the cycle are as follows:

1. *recognize*: find which rules are applicable, *i.e.* those rules whose antecedent conditions are satisfied by the current working memory;
2. *resolve conflict*: among the rules found in the first step (called a *conflict set*), choose which of the rules should "fire," *i.e.* get a chance to execute;
3. *act*: change the working memory by performing the consequent actions of all the rules selected in the second step.

As stated, this cycle repeats until no more rules can fire.

7.2 Working Memory

Working memory is composed of a set of *working memory elements* (WMEs). Each WME is a tuple of the form,

$$(\textit{type}\ \ \textit{attribute}_1\!:\textit{value}_1\ \ \ldots\ \ \textit{attribute}_n\!:\textit{value}_n),$$

where *type*, $\textit{attribute}_i$, and $\textit{value}_i$ are all atoms. Here are some examples of WMEs:

- (person age: 27 home: toronto)
- (goal task: putDown importance: 5 urgency: 1)
- (student name: john department: computerScience)

Declaratively, we understand each WME as an existential sentence:

$$\exists x\,[\ \textit{type}(x)\ \wedge\ \textit{attribute}_1(x) = \textit{value}_1\ \wedge\ \textit{attribute}_2(x) = \textit{value}_2\ \wedge\ \ldots$$
$$\wedge\ \textit{attribute}_n(x) = \textit{value}_n\ ]$$

Note that the individual about whom the assertion is made is not explicitly identified in a WME. If we choose to do so, we can identify individuals by using an attribute that is expected to be unique for the individual. For example, we might use a WME of the form (person identifier: 777-55-1234 name: janeDoe . . .). Note also that the order of attributes in a WME is not significant.

The example WMEs above represent objects in an obvious way. Relationships among objects can be handled by reification.[1] For example, something like

(basicFact relation: olderThan firstArg: john secondArg: mary)

might be used to say that John is older than Mary.

7.3 Production Rules

As we mentioned, the antecedent of a production rule is a set of conditions. If there is more than one condition, they are understood conjunctively, that is, they all have to be true for the rule to be applicable. Each condition can be positive or negative (negative conditions will be expressed as $-$*cond*), and the body of each is a tuple of this form:

$$(\textit{type}\ \ \textit{attribute}_1\text{: }\textit{specification}_1\ \ \ldots\ \ \textit{attribute}_k\text{: }\textit{specification}_k),$$

where each specification is one of the following:

- an atom
- a variable
- an evaluable expression, within "[ ]"
- a test, within "{ }"
- the conjunction ($\wedge$), disjunction ($\vee$), or negation ($\neg$) of a specification.

Here are two examples of rule conditions:

(person age: $[n + 4]$ occupation: x)

This condition is satisfied if there is a WME whose type is person and whose age attribute is exactly $n + 4$ (where n is specified elsewhere). The result binds the occupation value to x, if x is not already bound; if x is already bound, then the occupation value in the WME needs to be the same as the value of x.

[1]The technique of encoding n-ary relationships using reified objects and a collection of unary functions was discussed in Section 3.7 of Chapter 3.

$-$(person age: $\{< 23 \wedge > 6\}$)

This condition is satisfied if there is *no* WME in the WM whose type is person and whose age value is between 6 and 23.

Now, to be more precise about the applicability of rules, a rule is considered applicable if there are values for all the variables in the rule such that all the antecedent conditions are satisfied by the current WM. A positive condition is satisfied if there is a matching WME in the WM; a negative condition is satisfied if there no matching WME. A WME matches a condition if the types are identical and for each attribute/specification pair mentioned in the condition, there is a corresponding attribute/value pair in the WME, where the value matches the specification (under the given assignment of variables) in the obvious way. Of course, the matching WME may have attributes that are not mentioned in the condition.

Note that for a negated condition, there must be no element in the entire WM that matches it. This interpretation is negation as failure, as in PROLOG-type systems (see Chapter 5). We do not need to prove that such a WME could never exist in WM—it just has to be the case that no matching WME can be found at the time the rule is checked for applicability.

The consequent sides of production rules are treated a little differently. They have a strictly procedural interpretation, and each action in the action set is to be executed in sequence, and can be one of the following:

- `ADD` *pattern*: this means that a new WME specified by *pattern* is added directly to the WM.
- `REMOVE` i: i is an integer, and this means to remove (completely) from WM the WME that matched the i-th condition in the antecedent of the rule. This construct is not applicable if that condition was negative.
- `MODIFY` i (*attribute specification*): this means to modify the WME that matched the i-th condition in the antecedent, by replacing its current value for *attribute* by *specification*. `MODIFY` is also not applicable to negative conditions.

Note that in the `ADD` and `MODIFY` actions, any variables that appear refer to the values obtained when matching the antecedent of the rule. For example, the following rule might be used in an ordinary logical reasoning situation:

`IF` (student name: x) `THEN` `ADD` (person name: x)

In other words, if there is a WME of type student, with any name (and bind that name to x), then add to WM an element of type person, with the same name. This is a production rule version of the conditional $\forall x$ (Student(x) $\supset$ Person(x)), here used in a data-directed way. This conditional could also be handled in a very different way with a rule like this:

```
IF   (assertion  predicate: student)
THEN MODIFY 1 (predicate person)
```

In this case, we lose the original fact stated in terms of student, and replace it with one using the predicate, person.

The following example implements a simple database update. It assumes that some rule has added a WME of type birthday to the WM at the right time:

```
IF   (person  age: $x$  name: $n$)  (birthday  who: $n$)
THEN MODIFY 1 (age [$x$ + 1])
     REMOVE 2
```

Note that when the WME with the person's age is changed, the birthday WME is removed, so that the rule will not fire a second time.

The REMOVE action is also used on occasion to deal with control information. We might use a WME of type control to indicate what phase of a computation we are in. This can be initialized in the following way:

```
IF   (starting)
THEN REMOVE 1
     ADD  (control  phase: 1)
```

We could subsequently change phases of control with something like this:

```
IF   (control  phase: $x$)  ... other appropriate conditions ...
THEN  MODIFY 1 (phase [$x$ + 1])
```

7.4 A First Example

In order to illustrate a production system in action, consider the following task. We have three bricks, each of different size, sitting in a heap. We have three identifiable positions in which we want to place the bricks with a robotic "hand"; call these positions 1, 2, and 3. Our goal is to place the bricks in those positions in order of their size, with the largest in position 1 and the smallest in position 3.

Assume that when we begin, working memory has the following elements:

```
(counter  value: 1)
(brick  name: A  size: 10  position: heap)
(brick  name: B  size: 30  position: heap)
(brick  name: C  size: 20  position: heap)
```

In this case, the desired outcome is brick B in position 1, brick C in position 2, and brick A in position 3.

We can achieve our goal with two production rules that work with any number of bricks. The first one will place the largest currently available brick in the hand, and the other one will place the brick currently in the hand into the next position, going through the positions sequentially:

1.
```
IF   (brick  position: heap  name: $n$  size: $s$)
       −(brick  position: heap  size: {> $s$})
       −(brick  position: hand)
THEN MODIFY 1 (position hand)
```

   In other words, if there is a brick in the heap, and there is no bigger brick in the heap, and there is nothing currently in the hand, put the brick in the hand.

2.
```
IF (brick  position: hand)
   (counter  value: $i$)
THEN  MODIFY 1 (position $i$)
      MODIFY 2 (value [$i$ + 1])
```

   When there is a brick in the hand, this rule places it in the next position in sequence given by the counter, and increments the counter.

In this example, no conflict resolution is necessary, since only one rule can fire at a time: the second rule requires there to be a brick in the hand, and the first rule requires there to be none.

It is fairly simple to trace the series of rule firings and actions in this example. Recall that when we start, all bricks are in the heap, and none are in the hand. The counter is initially set to 1.

1. Rule 2 is not applicable, since no brick is in the hand. Rule 1 attempts to match each of the three WMEs of type brick in WM, but only succeeds for brick B, since it is the only one for which no larger brick exists in the heap. When Rule 1 matches, n is bound to B and s to 30. The result of this rule's firing, then, is the modification of the brick B WME to be:

   (brick name: B size: 30 position: hand)

2. Now that there is a brick in the hand, Rule 1 cannot fire. Rule 2 is applicable, with *i* being set to 1. Rule 2's firing results in two modifications, one to the brick B WME (position now becomes 1) and one to the counter WME:

   (brick name: B size: 30 position: 1)
   (counter value: 2)

3. Brick B no longer has its position as the heap, so now Rule 1 matches on brick C, whose position is modified as a result:

   (brick name: C size: 20 position: hand)

4. In a step similar to step 2 above, Rule 2 causes brick C to now be in position 2 and the counter to be reset to 3:

   (brick name: C size: 20 position: 2)
   (counter value: 3)

5. Now A is the only brick left in the heap, so Rule 1 matches its WME, and moves it to the hand:

   (brick name: A size: 10 position: hand)

6. Rule 2 fires again, this time moving brick A to position 3:

   (brick name: A size: 10 position: 3)
   (counter value: 4)

7. Now that there are no bricks in either the heap or the hand, neither Rule 1 nor Rule 2 is applicable. The system halts, with the final configuration of WM as follows:

   (counter value: 4)
   (brick name: A size: 10 position: 3)
   (brick name: B size: 30 position: 1)
   (brick name: C size: 20 position: 2)

7.5 A Second Example

Next we look at an example of a slightly more complex computation that is easy to do with production systems; we present a set of rules that computes how many days there are in any given year. In this example, working memory will have two

simple control elements in it: (wantDays year: n) will be our starting point and express the fact that our goal is to calculate the number of days in the year n. The WME (hasDays days: m) will express the result when the computation is finished. Finally, we will use a WME of type year to break the year down into its value *mod* 4, *mod* 100, and *mod* 400. Here are the five rules that capture the problem:

```
1. IF (wantDays year: n)
   THEN REMOVE 1
        ADD (year mod4: [n%4] mod100: [n%100] mod400: [n%400])

2. IF (year mod400: 0)
   THEN REMOVE 1
        ADD (hasDays days: 366)

3. IF (year mod100: 0 mod400: {≠ 0})
   THEN REMOVE 1
        ADD (hasDays days: 365)

4. IF (year mod4: 0 mod100: {≠ 0})
   THEN REMOVE 1
        ADD (hasDays days: 366)

5. IF (year mod4: {≠ 0})
   THEN REMOVE 1
        ADD (hasDays days: 365)
```

This rule set is structured in a typical way for goal-directed reasoning. The first rule initializes WM with the key values for a year that will lead to the calculation of the length of the year in days. Once it fires, it removes the wantDays WME and is never applicable again. Each of the other four rules check for their applicable conditions, and once one of them fires, it removes the year WME, so that the entire system halts. Each antecedent expresses a condition that only it can match, so again no conflict resolution is needed.

It is easy to see how this rule set works. If the input is 2000, then we start with (wantDays year: 2000) in WM. The first rule fires, which then adds to WM the WME (year mod4: 0 mod100: 0 mod400: 0). This matches only Rule 2, yielding (hasDays days: 366) at the end. If the input is 1900, the first rule adds the WME (year mod4: 0 mod100: 0 mod400: 300), which then matches only Rule 3, for a value of 365. If the input is 1996, we get (year mod4: 0 mod100: 96 mod400: 396), which matches only Rule 4, for a value of 366.

7.6 Conflict Resolution

Depending on whether we are doing data-directed reasoning or goal-directed reasoning, we may want to fire different numbers of rules, in case more than one rule is applicable. In a data-directed context, we may want to fire *all* rules that are applicable, to get all consequences of a sentence added to working memory; in a goal-directed context, we may prefer to pursue only a single method at a time, and thus wish to fire only one rule.

In cases where we do want to eliminate some applicable rules, there are many strategies for resolving rule conflicts and arriving at the most appropriate rule(s) to fire. The most obvious one is to choose an applicable rule at random. Here are some other common approaches:

- *order:* pick the first applicable rule in order of presentation. This is the type of strategy that PROLOG uses and is one of the most common ones. Production system programmers would take this strategy into account when formulating rule sets.
- *specificity:* select the applicable rule whose conditions are most specific. One set of conditions is said to be more specific than another if the set of WMs that satisfy it is a subset of those that satisfy the other. For example, consider the three rules

  ```
  IF (bird) THEN ADD (canFly)
  IF (bird weight: {>100}) THEN ADD (cannotFly)
  IF (bird) (penguin) THEN ADD (cannotFly)
  ```

  Here the second and third rules are both more specific than the first. If we have a bird that is heavy or that is a penguin, then the first rule applies, but the others should take precedence. (Note that if the bird is a penguin *and* heavy, another conflict resolution criteria might still have to come into play to help decide between the second and third rules.)
- *recency:* select an applicable rule based on how recently it has been used. There are different versions of this strategy, ranging from firing the rule that matches on the most recently created (or modified) WME to firing the rule that has been least recently used. The former could be used to make sure a problem solver stays focussed on what it was just doing; the latter would ensure that every rule gets a fair chance to influence the outcome.
- *refractoriness:* do not select a rule that has just been applied with the same values of its variables. This prevents the looping behaviour that results from firing a rule repeatedly because of the same WME. A variant forbids re-using a given rule-WME pair. Either the refractoriness can disappear automatically after a few cycles, or an explicit "refresh" mechanism can be used.

As implied in our penguin example above, non-trivial rule systems often need to use more than one conflict resolution criterion. For example, the OPS5 production rule system uses the following criteria for selecting the rule to fire amongst those that are found to be applicable:

1. discard any rule that has just been used for that value of variables;
2. order the remaining instances in terms of recency of WME matching the first condition, and then the second condition, *etc.*;
3. order the remaining rules by number of conditions;
4. if there is still a conflict, select arbitrarily among the remaining candidates.

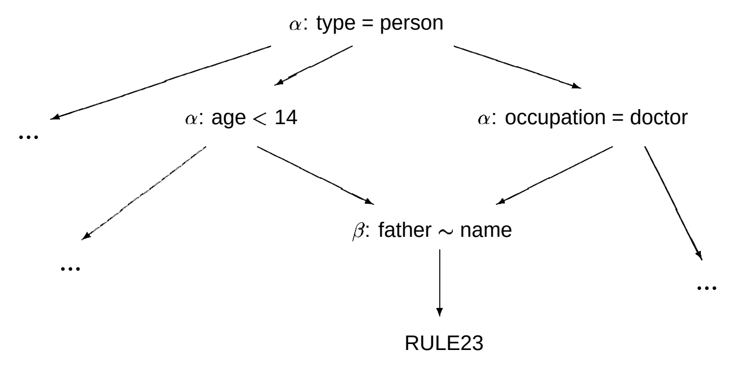
One interesting approach to conflict resolution is provided by the SOAR system. This system is a general problem solver that attempts to find a path from a start state to a goal state by applying productions. It treats selecting which rule to fire as deciding what the system should do next. Thus, if unable to decide on which rule to fire at some point, SOAR sets up a new *meta*-goal to solve, namely the goal of selecting which rule to use, and the process iterates. When this meta-goal is solved (which could in principle involve meta-meta-goals *etc.*), the system has made a decision about which base goal to pursue, and therefore the conflict is resolved.

7.7 Making Production Systems More Efficient

Early production systems, implemented in a straightforward way, ended up spending inordinate amounts of time (as much as 90%) in rule matching. Surprisingly, this remained true even when the matching was implemented using sophisticated indexing and hashing.

But two key observations led to an implementation breakthrough: first, that the WM was modified only very slightly on each rule-firing cycle, and second, that many rules shared conditions. The idea behind what came to be called the RETE algorithm was to create a network from the rule antecedents. Since the rules in a production system don't change during its operation, this network could be computed in advance. During operation of the production system, "tokens" representing new or changed WMEs are passed incrementally through the network of tests. Tokens that make it all the way through the network on any given cycle are considered

Figure 7.1: A sample RETE network



to satisfy all of the conditions of a rule. At each cycle, a new conflict set can then be calculated from the previous one and any incremental changes made to WM. This way, only a very small part of the WM is re-matched against any rule conditions, drastically reducing the time needed to calculate the conflict set.

A simple example will serve to illustrate. Consider a rule like the following:

```
IF (person name: x  age: {< 14}  father: y)
   (person name: y  occupation: doctor)
THEN  ...
```

This rule would cause the RETE network of figure 7.1 to be created. The network has two types of nodes: *"alpha"* nodes, which represent simple, self-contained tests, and *"beta"* nodes, which take into account the fact that variables create constraints between different parts of an antecedent. Tokens for all new WMEs whose type was person would enter the network at the topmost (alpha) node. If the age of the person was not known to be less than 14, or the person was not known to be a doctor, there the token would sit until one of the relevant attributes was modified by a rule. A person WME whose age was known to be less than 14 would pass down to the age alpha node; one whose occupation was doctor would pass to the other alpha node in the figure. In the case where a pair of WMEs residing at those alpha nodes also shared a common value between their respective father and name attributes, a token would pass through the lower beta node expressing the constraint, indicating that this rule was now applicable. For tokens left sitting in the network at the end of a cycle, any modifications to the corresponding WMEs would cause a reassessment, to see if they could pass further down the network, or combine with other WMEs at a beta node. Thus the work at each step is quite small and incremental.

7.8 Applications and Advantages

Production systems are a general computational framework, but one based originally on the observation that human experts appear to reason from "rules of thumb" in carrying out tasks. The production system architecture was the first reasoning system to attempt to model explicitly not only the knowledge that people have, but also the reasoning method they use when performing mental tasks. Here, for example, is a production rule that suggests one step in the procedure a person might use in carrying out a subtraction:

```
IF (goal is: getUnitDigit)
   (minuend unit: d)
   (subtrahend unit: {> d})
THEN REMOVE 1
     ADD (goal is: borrowFromTens)
```

What was especially interesting to researchers in this area of psychology was the possibility of modeling the errors or misconceptions people might have in symbolic procedures of this sort.

Subsequently, what was originally a descriptive framework for psychological modeling was taken up in a more prescriptive fashion in what became known as *expert systems*. Expert systems, now a core technology in the field, use rules as a representation of knowledge for problems that ordinarily take human expertise to solve. But because human experts seem to reason from symptoms to causes (and similarly in other diagnosis and reasoning problems) in a heuristic fashion, production rules seem to be able to handle significant problems of great consequence, ranging from medical diagnosis, to checking for credit-worthiness, to configuration of complex products. We will look briefly at some of these rule-based systems in the next section below.

There are many advantages claimed for production systems when applied to practical complex problems. Among the key advantages, these are usually cited:

- *modularity:* in a production rule framework, each rule works independently of the others. This allows new rules to be added or old rules to be removed incrementally in a relatively easy fashion.
- *fine-grained control:* production systems have a very simple control structure. There are no complex goal or control stacks hidden in the implementation, among other things.
- *transparency:* because rules are usually derived from expert knowledge or observation of expert behaviour, they tend to use terminology that humans can resonate with. In contrast to formalisms like neural networks, the reasoning behavior of the system can be traced and explained in natural language.

In reality—especially when the systems get large and are used to solve complex problems—these advantages tend to wither. With hundreds or even thousands of rules, it is deceptive to think that rules can be added or removed with impunity. Often, more complex control structures than one might suppose are embedded in the elements of WM (remember attributes like phase and counter from above) and in very complex rule antecedents. But production rules have been used successfully on a very wide variety of practical problems, and are an essential element of every AI researcher's toolkit.

7.9 Some Significant Production Rule Systems

Given the many years that they have been used and the many problems to which they have been applied, there are many variants on the production system theme. While it is impossible to survey here even all of the important developments in the area, one or two significant contributions are worth mentioning. Among other systems, work on MYCIN and XCON has influenced virtually all subsequent work in the area.

MYCIN was developed at Stanford in the 1970's to aid physicians in the diagnosis of bacterial infections. After working with infectious disease specialists, the MYCIN team built a system with approximately 500 production rules for recognizing roughly 100 causes of infection. While the system operated in the typical forward-chaining manner of production systems (using the recognize/resolve/act cycle we studied above), it performed its reasoning in a goal-directed fashion. Rules looked for symptoms in WM and used those symptoms to build evidence for certain hypotheses.

Here is a simplified version of a typical MYCIN rule:

```
IF
    the type of $x$ is primary bacteremia
    the suspected entry point of $x$ is the gastrointestinal tract
    the site of the culture of $x$ is one of the sterile sites
THEN
    there is evidence (0.8) that $x$ is bacteroides
```

MYCIN also introduced the use of other static data structures (not in WM) to augment the reasoning mechanism; these included things like lists of organisms and clinical parameters. But perhaps the most significant development was the introduction of a level of *certainty* in the accumulation of evidence and confidence in hypotheses. Since in medical diagnosis not all conclusions are obvious, and many diseases can produce the same symptoms, MYCIN worked by accumulating evidence and trying to ascertain what was the most likely hypothesis, given that evidence. The technical means for doing this was what were called "*certainty factors*," which were numbers from 0 to 1 attached to the conclusions of rules; these allowed the rank ordering of alternative hypotheses. Since rules could introduce these numeric measures into working memory, and newly considered evidence could change the confidence in various outcomes, MYCIN had to specify a set of combination rules for certainty factors. For example, the conjunction of two conclusions might take the minimum of the two certainty factors involved, and their disjunction might imply the maximum of the two.[2]

In a very different line of thinking, researchers at Carnegie-Mellon produced an important rule-based system called XCON (originally called R1). The system has been in use for many years at what was the Digital Equipment Corporation for configuring computers, starting with its VAX line of products. The most recent versions of the system had over 10,000 rules, covering hundreds of types of components. This system was the main stimulus for widespread commercial interest in rule-based expert systems. Substantial commercial development, including the formation of several new companies, has subsequently gone into the business of configuring complex systems, using the kind of technology pioneered by XCON.

Here is a simplified version of a typical XCON rule:

```
IF
    the context is doing layout and assigning a power supply
    an sbi module of any type has been put in a cabinet
    there is space available for the power supply
```

[2]We address certainty factors and their relationship to other numerical means of combining evidence in Chapter 11.

```
        there is no available power supply
        the voltage and frequency of the components are known
    THEN
        add an appropriate power supply
```

XCON was the first rule-based system to segment a complex task into sections, or "contexts," to allow subsets of the very large rule base to work completely independently of one another. It broke the configuration task down into a number of major phases, each of which could proceed sequentially. Each rule would typically include a condition like (control phase: 6) to ensure that it was applicable to just one phase of the task. Then special context switching rules, like the kind we saw at the end of Section 7.3, would be used to move from one phase of the computation to another. This type of framework allowed for more explicit emulation of standard control structures, although again, one should note that this type of architecture is not ideal for complex control scenarios.

While grouping rules into contexts is a useful way of managing the complexity of large knowledge bases, we now turn our attention to an even more powerful organizational principle, object orientation.

7.10 Bibliographic notes

7.11 Exercises

Chapter 8

Object-Oriented Representation

One property shared by all of the representation methods we have considered so far is that they are *flat*: each piece of representation is self-contained and can be understood independently of any other. Recall that when we discussed logical representations in Chapter 3, we observed that information about a given object we might care about could be scattered amongst any number of seemingly unrelated sentences. With production system rules and the procedures in procedural systems, we have the corresponding problem: knowledge about a given object or type of object could be scattered around the knowledge base.

As the number of sentences or procedures in a KB grows, it becomes critical to organize them in some way. As we have seen, in a production system, rule sets can be organized by their context of application. But this is primarily a control structure convenience for grouping items by when they might execute. A more representationally motivated approach would be to group facts or rules in terms of the kinds of *objects* they pertain to. Indeed it is very natural to think of knowledge itself not as a mere collection of sentences, but rather as structured and organized in terms of what the knowledge is *about*, the objects of knowledge. In this chapter, we will examine a procedural knowledge representation formalism that is object-oriented in this way.

8.1 Objects and frames

The objects that we care about range far and wide, from physical objects like houses and people, to more conceptual objects like courses and trips, and even to reified abstractions like events and relations. Each of these types of object has its own *parts*, some physical (roof, doors, rooms, fixtures, etc.; arms, torso, head, etc.), and some more abstract (course title, teacher, students, meeting time, etc.; destination, conveyance, departure date, etc.). The parts are constrained in various ways: the roof has to be connected to the walls in a certain way, the departure date and the first leg of a trip have to be related, and so on. The constraints between the parts might be expressed procedurally, such as by the registration procedure that connects a student to a course, or the procedure for reserving an airline seat that connects the second leg of a trip to the first. And some types of objects might have procedures of other sorts that are crucial to our understanding of them: procedures for recognizing bathrooms in houses, for reserving hotel rooms on trips, and so on. In general, in a procedural object-oriented representation system, we consider the kinds of reasoning operations that are relevant for the various types of objects in our application, and we design procedures to deal with them.

In one of the more seminal papers in the history of Knowledge Representation, Marvin Minsky in 1975 suggested the idea of using object-oriented groups of procedures to recognize and deal with new situations. Minsky used the term *frame* for the data structure used to represent these situations. While the original intended application of frames as a knowledge representation was for recognition, the idea of grouping related procedures in this way for reasoning has much wider applicability. Among its more natural applications we might find the kind of relationship recognition common in story understanding, data monitoring in which we look for key situations to arise, and propagation and enforcement of constraints in planning tasks.

8.2 A basic frame formalism

To examine the way frames can be used for reasoning, it will help us to have a formal representation language to express their structure. For the sake of discussion, we will keep the language simple, although extremely elaborate frame languages have been developed.

8.2.1 Generic and individual frames

For our purposes, there are two type of frames: *individual* frames, used to represent single objects, and *generic* frames, used to represent categories or classes of objects. An individual frame is a named list of "buckets" into which values can be dropped. The buckets are called *slots*, and the items that go into them are called *fillers*. Individual frames are similar to the working memory elements of production systems seen in the previous chapter. Schematically, an individual frame looks

like this:

```
(Frame-name
    <slot-name1 filler1>
    <slot-name2 filler2>
    ...)
```

The frame and slot names are atomic symbols; the fillers are either atomic values (like numbers or strings) or the names of other individual frames.

Notationally, the names of generic frames appear here capitalized, while individual frames will be in lower case. Slot names will be capitalized and prefixed with a ":". For example, we might have the following frames:

```
(tripLeg123
    <:INSTANCE-OF TripLeg>
    <:Destination toronto> ...)
(toronto
    <:INSTANCE-OF CanadianCity>
    <:Province ontario>
    <:Population 4.5M> ...)
```

Individual frames also have a special distinguished slot called **:INSTANCE-OF**, whose filler is the name of a generic frame indicating the category of the object being represented. We say that the individual frame is an *instance* of the generic one, so, in the above, toronto is an instance of CanadianCity.

Generic frames, in their simplest form, have a syntax that is similar to individual frames:

```
(CanadianCity
    <:IS-A City>
    <:Province CanadianProvince>
    <:Country canada>)
```

In this case, slot fillers are the names of either generic frames (like CanadianProvince) or individual ones (like canada). Instead of an **:INSTANCE-OF** slot, generic frames can have a distinguished slot called **:IS-A**, whose filler is the name of a more general generic frame. We say that the generic frame is a *specialization* of the more general one, e.g., CanadianCity is a specialization of City.

Slots of generic frames can also have *procedures* associated with them. In the simple case we consider here, there are two types of "attached" procedures, **IF-ADDED** and **IF-NEEDED**, which are object-oriented versions of the if-added and if-needed procedures from Chapter 6. The syntax is illustrated in these examples:

```
(Table
    <:Clearance [IF-NEEDED ComputeClearanceFromLegs]> ...)
(Lecture
    <:DayOfWeek WeekDay>
    <:Date [IF-ADDED ComputeDayOfWeek]> ...)
```

Note that a slot can have both a filler and an attached procedure in the same frame.

8.2.2 Inheritance

As we will see below, much of the reasoning that is done with a frame system involves creating individual instances of generic frames, filling some of the slots with values, and inferring some other values. The **:INSTANCE-OF** and **:IS-A** slots have a special role to play in this process. In particular, the generic frames can be used to fill in values that are not mentioned explicitly in the creation of the instance, and they can also trigger additional actions when slot fillers are provided.

For example, if we ask for the :Country of the toronto frame above, we can determine that it is canada by using the **:INSTANCE-OF** slot, which points to CanadianCity, where that value is given. The process of passing information from generic frames down to their instances is called *inheritance* (the "child" frames inherit properties from their "parents"), and we say that toronto inherits the :Country property from CanadianCity. If we had not provided a filler for the :Province of toronto, we would still know by inheritance that we were looking for an instance of CanadianProvince (which could be useful in a recognition task). Similarly, if we had not provided a filler for :Population, but we also had the following frame,

```
(City
    <:Population NonNegativeNumber> ...)
```

then by using both the **:INSTANCE-OF** slot of toronto and the **:IS-A** slot of CanadianCity, we would know by inheritance that we were looking for an instance of NonNegativeNumber.

The inheritance of attached procedures works analogously. If we create an instance of Table above, and we need to find the filler of the :Clearance slot for that instance, we can use the attached **IF-NEEDED** procedure to compute the clearance of that table from the height of its legs. This procedure would also be used through inheritance if we created an instance of the frame MahoganyCoffeeTable, where we had the following:

```
(CoffeeTable
    <:IS-A Table> ...)
(MahoganyCoffeeTable
    <:IS-A CoffeeTable> ...)
```

Similarly, if we create an instance of the Lecture frame from above with a lecture date specified explicitly, the attached **IF-ADDED** procedure would fire immediately to calculate the day of the week for the lecture, filling the slot :DayOfWeek. If we later changed the :Date slot, the :DayOfWeek slot would again be changed by the same procedure.

One of the distinguishing features of the inheritance of properties in frame systems is that it is *defeasible*. By this we mean that we use an inherited value only if we cannot find a filler otherwise. So a slot filler in a generic frame can be overridden explicitly in its instances and in its specializations. For example, if we have a generic frame like

```
(Elephant
    <:IS-A Mammal>
    <:EarSize large>
    <:Color gray> ...)
```

we are saying that instances of Elephant have a certain :EarSize and :Color property *by default*. We might have the following other frames:

```
(raja
    <:INSTANCE-OF Elephant>
    <:EarSize small> ...)
(RoyalElephant
    <:IS-A Elephant>
    <:Color white> ...)
(clyde
    <:INSTANCE-OF RoyalElephant> ...)
```

In this case, raja inherits the gray color of elephants, but has small ears; clyde inherits the large ears from Elephant via RoyalElephant, but inherits the white color from RoyalElephant, overriding the default from Elephant.

Normally in frame systems, all values are understood as default values, and nothing is done automatically to check the validity of an explicitly provided filler. So, for example, nothing stops us from creating an individual frame like

```
(city135
    <:INSTANCE-OF CanadianCity>
    <:Country holland>)
```

It is also worth mentioning that in many frame systems, individual frames are allowed to be instances of (and generic frames are allowed to be specializations of) more than one generic frame. For example, we might want to say that

```
(AfricanElephant
    <:IS-A Elephant>
    <:IS-A AfricanAnimal> . . . )
```

with properties inherited from both generic frames. This of course complicates inheritance considerably since the values from Elephant may conflict with those from AfricanAnimal. We will further examine this more general form of inheritance in Chapter 10.

8.2.3 Reasoning with frames

The procedures attached to frames give us a flexible, organized framework for computation. Reasoning within a frame system usually starts with the system's "recognizing" an object as an instance of a generic frame, and then applying procedures triggered by that recognition. Such procedure invocations can then produce more data or changes in the knowledge base that can cascade to other procedure calls. When no more procedures are applicable, the system halts.

More specifically, the basic reasoning loop in a frame system has these three steps:

1. a user or external system using the frame system as its knowledge representation declares that an object or situation exists, thereby instantiating some generic frame;

2. any slot fillers that are not provided explicitly but can be inherited by the new frame instance are inherited;

3. for each slot with a filler, any **IF-ADDED** procedure that can be inherited is run, possibly causing new slots to be filled, or new frames to be instantiated, and the cycle repeats.

If the user, the external system, or an attached procedure requires the filler of a slot, then we get the following behavior:

1. if there is a filler stored in the slot, then that value is returned;

2. otherwise, any **IF-NEEDED** procedure that can be inherited is run, calculating the filler for the slot, but potentially also causing other slots to be filled, or new frames to be instantiated, as above.

If neither of these produce a result, then the value of the slot is considered to be unknown. Note that in this account, the inheritance of property values is done at the time the individual frame is created, but **IF-NEEDED** procedures, which calculate property values, are only invoked as required. Other schemes are possible.

The above comprises the local reasoning involving a single frame. When constructing a frame knowledge base, one would also think about the global structure of the KB and how computation should produce the desired overall reasoning. Typically, generic frames are created for any major object-type or situation-type required in the problem-solving. Any constraints between slots are expressed by the attached **IF-ADDED** and **IF-NEEDED** procedures. As in the procedural systems of Chapter 6, it is up to the designer to decide whether reasoning should be done in a data-directed or goal-directed fashion.

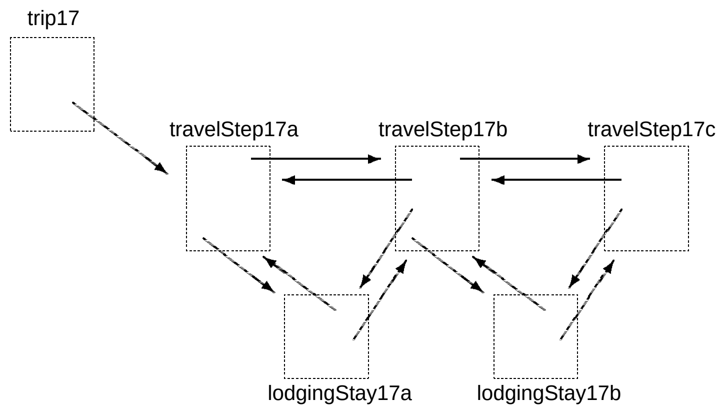
In the above account, default values are filled in whenever they are available on slots. It is worth noting that in the original, psychological view that first gave rise to frames, defaults were considered to play a major role in scene, situation, and object recognition; it was felt that people were prone to generalize from situations they had seen before, and that they would assume that objects and situations were "typical"—had key aspects taking on their normal default values—unless specific features in the individual case were noticed to be exceptional.

Overall, given the constraints between slots that are enforced by attached procedures, we can think of a frame knowledge base as a symbolic "spreadsheet," with constraints between the objects we care about being propagated by attached procedures. But the procedures in a frame KB can do a lot more, including invoking complex actions by the system.

8.3 An example: using frames to plan a trip

We now turn our attention to developing an example frame system, to see how these representations work in practice. The example will be part of a scheme for planning trips. We will see how the "symbolic spreadsheet" style of reasoning in frame systems is used. This might be particularly useful in supporting the documentation one often uses in a company for reporting expenses.

Figure 8.1: Sketch of structure of a trip



The basic structure of our representation involves two main types of frames: Trip and TravelStep. A Trip will have a sequence of TravelSteps, linked together by appropriate slots. A TravelStep will usually terminate in a LodgingStay, except when there are two travel legs in a single day, and for the last leg of a trip.

In order to make the correspondences work out correctly (and be able to keep track of what is related to what), a LodgingStay will use slots to point to its arriving TravelStep and its departing TravelStep. Similarly, TravelSteps will indicate the LodgingStays at their origin and destination. Graphically, for a trip with three legs (instances of TravelStep), we might sketch the relationships as in Figure 8.1.

Using the obvious slot names, a Trip in general will look like this:

```
(Trip
     <:FirstStep  TravelStep>
     <:Traveler  Person>
     <:BeginDate  Date>
     <:EndDate  Date>
     <:TotalCost  Price>
     ...)
```

A specific Trip, say, trip17, might look like this:

```
(trip17
     <:FirstStep  travelStep17a>
     <:Traveler  ronB>
     <:BeginDate  11/13/98>
     <:EndDate  11/18/98>
     <:TotalCost  $1752.45>
     ...)
```

In general, instances of TravelStep and LodgingStay will share some properties (e.g., each has a beginning date, an end date, a cost, and a payment method), so for representational conciseness, we might posit a more general category, TripPart, of which the two other frames would be specializations:

```
(TripPart
     <:BeginDate  Date>
     <:EndDate  Date>
     <:Cost  Price>
     <:PaymentMethod  FormOfPayment>
     ...)
(LodgingStay
     <:IS-A  TripPart>
     <:Place  City>
     <:LodgingPlace  LodgingPlace>
     <:ArrivingTravelStep  TravelStep>
     <:DepartingTravelStep  TravelStep>
     ...)
(TravelStep
     <:IS-A  TripPart>
     <:Origin  City>
     <:Destination  City>
     <:OriginLodgingStay  LodgingStay>
     <:DestinationLodgingStay  LodgingStay>
     <:Means  FormOfTransportation>
     <:DepartTime  Time>
     <:ArrivalTime  Time>
     <:NextStep  TravelStep>
     <:PreviousStep  TravelStep>
     ...)
```

This gives us our basic overall structure for a trip. Next we embellish the frame structure with various defaults, and procedures that will help us enforce constraints. For example, our trips might most often be made by air, in which case the default filler for the :Means slot of a TravelStep should be airplane:

```
(TravelStep
    <:Means  airplane> ...)
```

We might also make a habit of paying for parts of trips with a Visa credit card:

```
(TripPart
    <:PaymentMethod  visaCard> ...)
```

However, perhaps because of the insurance provided by a certain credit card, we may prefer American Express for travel steps, overriding this default:

```
(TravelStep
    <:PaymentMethod  americanExpressCard> ...)
```

As indicated earlier, not all inherited fillers of slots will necessarily be specified as fixed values; it may be more appropriate to compute them from the current circumstances. For example, it would be appropriate to automatically set up the origin of a travel step as our home airport, say Newark, as long as there was no previous travel step—in other words, Newark is the default airport for the beginning of a trip. To do this we introduce two pieces of notation:

- if x refers to an individual frame and y to a slot, then xy refers to the filler of the slot for the frame;
- SELF will be a way to refer to the frame currently being processed.

Thus, our travel step description would look like this:

```
(TravelStep
    <:Origin
        [IF-NEEDED
            {if no SELF:PreviousStep
                    then newark
                    else SELF:PreviousStep:Destination}]> ...)
```

This attached procedure says that for any TravelStep, if we want its origin city, use the destination of the previous TravelStep. If there is no previous TravelStep, use newark.

Another useful thing to do with a travel planning symbolic spreadsheet would be to compute the total cost of a trip from the costs of each of its parts:

```
(Trip
    <:TotalCost
        [IF-NEEDED
            {let result ← 0;
             let x ← SELF:FirstStep;
             repeat
                {if x:NextStep
                   then
                      {result ← result + x:Cost
                       if x:DestinationLodgingStay then
                          result ← result + x:DestinationLodgingStay:Cost;
                       x ← x:NextStep}
                   else return result + x:Cost}}]> ...)
```

This **IF-NEEDED** procedure iterates through the travel steps, starting at the trip's :FirstStep. At each TravelStep, it adds the cost of the travel step itself (x:Cost) to the previous result, and if there is a subsequent step, the cost of the lodging stay between those two steps, if any (x:DestinationLodgingStay:Cost). Note that this procedure is written in a suggestive pseudo-code rather than in any particular programming language.

Another useful thing to expect an automatic travel documentation system to do would be to create a skeletal lodging stay instance each time a new travel leg was added. The following **IF-ADDED** procedure does a basic form of this:

```
(TravelStep
    <:NextStep
        [IF-ADDED
            {if SELF:EndDate ≠ SELF:NextStep:BeginDate
               then
                 SELF:DestinationLodgingStay ←
                  SELF:NextStep:OriginLodgingStay ←
                   create new LodgingStay
                      with :BeginDate = SELF:EndDate
                      and with :EndDate = SELF:NextStep:BeginDate
                      and with :ArrivingTravelStep = SELF
                      and with :DepartingTravelStep = SELF:NextStep
               ...}]> ...)
```

Note that the first thing done is to confirm that the next travel leg begins on a different day than the one we are starting with ends; presumably no lodging stay is needed if the two travel legs join on the same day.

Note also that the default :Place of a LodgingStay (and other fillers) could also be calculated as another piece of automatic processing:

```
(LodgingStay
    <:Place [IF-NEEDED
              {SELF:ArrivingTravelStep:Destination}]> ...)
```

This might be a fairly weak default, however, and its utility would depend on the particular application. It is quite possible that a traveller's preferred default city for lodging is different than the destination city for the arriving leg of the trip (e.g., flights may arrive in San Francisco, but I may prefer as a default to stay in Palo Alto).

8.3.1 Using the example frames

We now consider how the various frame fragments we have created might work together in specifying a trip. Imagine that we propose a trip to Toronto on December 21, 2006, returning home the following day. First, we create an individual frame for the overall trip (call it trip18), and one for the first leg of the trip:

```
(trip18
    <:INSTANCE-OF Trip>
    <:FirstStep travelStep18a>)
(travelStep18a
    <:INSTANCE-OF TravelStep>
    <:Destination toronto>
    <:BeginDate 12/21/06>
    <:EndDate 12/21/06>)
```

Since we know we are to return home the next day, we create the second leg of the trip:

```
(travelStep18b
    <:INSTANCE-OF TravelStep>
    <:Origin toronto>
    <:BeginDate 12/22/06>
    <:PreviousStep travelStep18a>)
```

To complete the initial setup, travelStep18a will need its :NextStep slot filled with travelStep18b.

As a consequence of the initial setup of the two instances of TravelStep—in particular, the assignment of travelStep18b as the :NextStep of travelStep18a—a default LodgingStay is automatically created to represent the overnight stay between those two legs of the trip (using the **IF-ADDED** procedure on the :NextStep slot):

```
(lodgingStay18a
    <:INSTANCE-OF LodgingStay>
    <:BeginDate 12/21/06>
    <:EndDate 12/22/06>
    <:ArrivingTravelStep travelStep18a>
    <:DepartingTravelStep travelStep18b>)
```

Note that the **IF-NEEDED** procedure for the :Place slot of LodgingStay would infer a default filler of toronto for lodgingStay18a, if required. Once we have established the initial structure, we can see how the :Means slot of either step would be filled by default, and a query about the :Origin slot of the first step would produce an appropriate default value, as in Figure 8.2.

As a final illustration, imagine that we have over the course of our trip filled in the :Cost slots for each of the instances of TripPart. If we ask for the :TotalCost of the entire trip, the **IF-NEEDED** procedure defined above will come into play (assuming the totalCost slot has not already been filled manually). Given the final state of the trip as expressed in Figure 8.2, the calculation proceeds as follows:

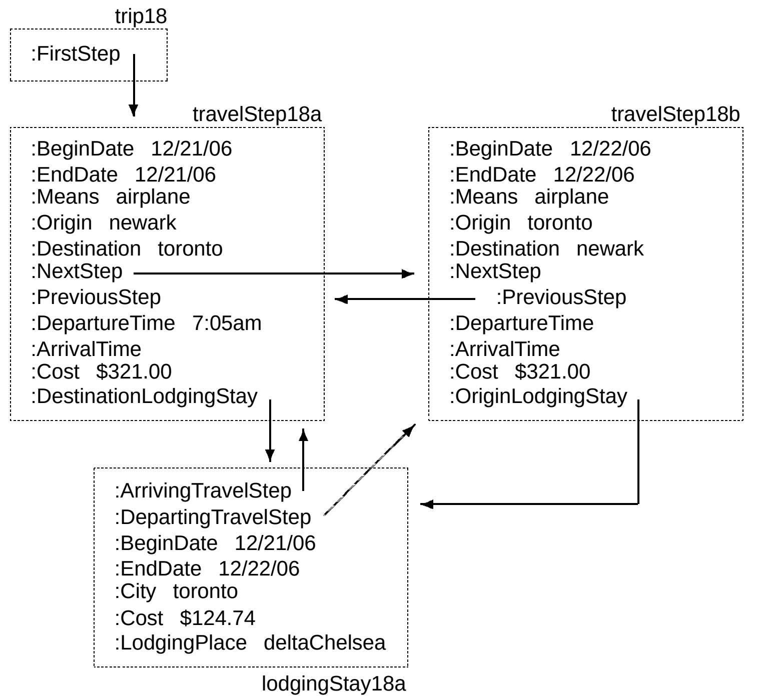
- *result* is initialized to 0, and x is initialized to travelStep18a, which makes x:NextStep be travelStep18b;
- the first time through the `repeat` loop, *result* is set to the sum of *result* (0), the cost of x ($321.00), and the cost of the :DestinationLodgingStay of the current step (lodgingStay18a) ($124.75); x is then set to travelStep18b;
- the next time through, since x (travelStep18b) has no following step, the loop is broken and the sum of *result* ($445.75) and the cost of x ($321.00) is returned.

So a grand total of $766.75 is taken to be the :TotalCost of trip18.

8.4 Beyond the basics

The trip planning example considered above is typical of how frame systems have been used: start with a sketchy description of some circumstance and embellish it

Figure 8.2: The travel example with lodging stay



with defaults and implied values. The **IF-ADDED** procedures can make updates easier and help to maintain consistency; the **IF-NEEDED** procedures allow values to be computed only when they are needed. There is a tradeoff here, of course, and which type of procedure to use in an application will depend on the potential value to the user of seeing implied values computed up front, versus the value of waiting to do computation only as required.

8.4.1 Other uses of frames

There are other types of applications for frame systems. One would be to use a frame system to provide a structured, knowledge-based monitoring function over a database. By hooking the frames to items in a database, changes in values and newly-added values could be detected by the frame system, and new frame instances or implied slot values could be computed and added to the database, without having to modify the DBMS itself to handle rules. In some ways, this combination would act like an expert system. But database monitors are probably more naturally thought of as object-centered (generic frames could line up with relations in the schema, for example), in which case a frame representation is a better fit than a flat production system.

Other uses of frame systems come closer to the original thinking about psychologically-oriented recognition processes espoused by Minsky in 1975. These include, for example, structuring views of typical activities of characters in stories. The frame structures for such activities have been called *scripts*, and have been used to recognize the motivations of characters in the stories, and to set up expectations for their later behavior. More general commonsense reasoning of the sort that Minsky envisioned would use local cues from a situation to suggest potentially relevant frames, which in turn would set up further expectations that could drive investigation procedures.

Consider for example, a situation where many people in a room were holding what appeared to be wrapped packages, and balloons and cake were in evidence. This would suggest a birthday party, and prompt us to look for the focal person at the party (a key slot of the birthday party frame), and to interpret the meaning of lit candles in a certain way. Expectations set up by the suggested frames could be used to confirm the current hypothesis (that this is a birthday party). If they were subsequently violated, then an appropriately represented "differential diagnosis" attached to the frame could lead the system to suggest other candidate frames, taking the reasoning in a different direction. For example, no candles on the cake could suggest a retirement or anniversary party.

8.4.2 Extensions to the frame formalism

As with other knowledge representation formalisms, frame systems have been subject to many extensions to handle ever more complex applications. Here we briefly review some of these extensions.

Other procedures: An obvious way to increase the expressiveness and utility of the frame mechanism is to include other types of procedures. The whole point of object-oriented reasoning is to determine the sort of questions appropriate for a type of object, and to design procedures to answer them. For trips, for example, we have only considered two forms of questions, exemplified by "What is the total cost of a trip?" (handled by an **IF-NEEDED** procedure) and "What should I do if I find out about a new leg of a trip?" (handled by an **IF-ADDED** procedure). But other questions that do not fit these two patterns are certainly possible, such as "What should I do if I cancel a leg of a trip?" (requiring some sort of "if-removed" procedure), or "How do I recognize an overly expensive trip?" (along the lines of the birthday party recognition example above), or "What do I need to look out for in an overseas trip?" and so on.

Multiple slot fillers: In addition to extending the repertoire of procedures attached to a frame knowledge base, we can also expand the types of slots used to express parts and features of objects. One obvious extension is to allow *sets* of frames to fill slots. Procedures attached to the slot could then operate on the entire set of fillers, and constraints on the cardinality of these sets could be used in reasoning, as we will see in the description logics of Chapter 9. One complication this raises concerns inheritance: with multiple slot fillers, we need to know whether the fillers of a slot given explictly should or should not be augmented by other fillers through inheritance.

Other slot facets: So far, we have seen that both default fillers and procedures can be associated with a slot. We can imagine dealing with other aspects of the relationship between a slot and a frame. For example, we might want to be able to *insist* that instances of a generic frame provide a filler of a certain type (or perhaps check the validity of the provided filler with a procedure), rather than being merely a default. Another possibility is to state *preferences* we might have regarding the filler of a slot. Preferences could be used to help select a filler among a number of competing inherited values.

Meta-frames: Generic frames can sometimes usefully be considered to be instances of higher-level meta-frames. For example, generic frames like CanadianCity and NewJerseyCity represent a type of city defined by a geographic region. So we might think of them as being instances (not specializations) of a meta-frame like GeographicalCityType. We might have something like

```
(GeographicalCityType
    <:IS-A CityType>
    <:DefiningRegion GeographicalRegion>
    <:AveragePopulation NonNegativeNumber> ...)
```

An instance of this frame, like CanadianCity, would have a particular value for the :DefiningRegion slot, namely canada. The filler for the :AveragePopulation slot

for CanadianCity could be calculated by an **IF-NEEDED** procedure, by iterating through all the Canadian cities. Observe that individual cities themselves do not have a defining region or an average population. So we need to ensure that frames like toronto do not inherit these slots from CanadianCity. The usual way this is done is to distinguish the "member" slots of a generic frame, which apply to instances (members) of the frame (like the :Country of a CanadianCity), from the "own" slots of the frame, which apply to the frame itself (like the AveragePopulation of CanadianCity).

8.4.3 Object-driven programming with frames

Frame-structured knowledge bases are the first instance we have seen of an object-oriented representation. Careful attention to the mapping of generic frames to categories of objects in a domain of interest can yield a simple declarative knowledge base, emphasizing taxonomies of objects and their structural relationships. However, as we have seen, attached procedures can be a useful adjunct to a pure object-oriented representation structure, and in practice, we are encouraged to take advantage of their power to build a complex, highly procedural knowledge base. In this case, what is known about the connections among the various symbols used is expressed through the attached procedures, just as it was in the procedural and production systems of previous chapters. While there is nothing intrinsically wrong with this, it does mean moving away from the original declarative view of knowledge—taking the world to be one way and not another—presented in the first chapter.

The shift to a more procedural view of frames moves us close to conventional object-oriented programming (OOP). Indeed frame-based representation languages and OOP systems were developed concurrently, and share many of the same intuitions and techniques. A procedural frame system shares the advantages of a conventional OOP system: definition is done primarily by specialization of more general classes, control is localized, methods can be inherited, encapsulation of abstract procedures is possible, etc. The main difference is that frame systems tend to have a centralized, conventional control regime, whereas OOP systems have objects acting as small, independent agents sending each other messages. Frame systems tend to work in a cycle: instantiate a frame and declare some slot fillers, inherit values from more general frames, trigger appropriate forward-chaining procedures, and then, when quiescent, stop and wait for the next input; OOP systems tend to be more decentralized and less patterned. As a result, there can be some applications for which a frame-based system can provide some advantages over a more generic OOP system, for example, in the style of applications that we touched on above. But if the primary use of a frame system is as an organizing method for procedures, this contrast should be examined carefully to be sure that the system is best suited to the task.

In Chapter 9 we will continue our investigation of object-oriented knowledge representation, but now without procedures, in a more logical and declarative form.

8.5 Bibliographic notes

8.6 Exercises

Chapter 9

Structured Descriptions

In Chapter 8, we looked at knowledge organization inspired by our natural tendency to think in terms of categories of *objects*. However, the frame representation seen there focused more on the organization and invocation of *procedures* than on inferences about the objects and categories themselves. Reasoning about objects in everyday thinking goes well beyond the simple cascaded computations seen in that chapter, and is based on considerations like the following:

- objects naturally fall into classes (e.g., my pet is a dog; my wife is a physician), but are very often thought of as being members of multiple classes (I am an author, an employee and a father);
- classes can be more general or more specific than others (e.g., Collie and Schnauzer are types of dogs; a rheumatologist is a type of physician; a father is a type of parent);
- in addition to generalization being common for classes with simple atomic names, it is also natural for those with more complex descriptions (e.g., a part-time employee is an employee; a family with at least 1 child is a family; a family with 3 children is a family that is not childless);
- objects have parts, sometimes in multiples (e.g., books have (single) titles, tables have at least 3 legs, automobiles have 4 wheels);
- the configuration of an object's parts is essential to its being considered a member of a class (e.g., a stack of bricks is not the same as a pile of the very same bricks).

In this chapter we will delve into representation techniques that look more directly at these aspects of objects and classes than frames did. In focusing on the more declarative aspects of an object-oriented representation, our analysis will take us back to concepts like predicates and entailment from FOL. But as we shall see, what matters about these predicates and the kind of entailments we will consider here will be quite different.

9.1 Descriptions

Before we look at the details of a formal knowledge representation language in the next section, one useful way to get our bearings is to think in terms of the expressions of a natural language like English. In our discussion of knowledge in Chapter 1, and in our presentation of FOL, we focussed mainly on *sentences*, since it is sentences, after all, that express what is known. Here, we want to talk about *noun phrases*. Like sentences, noun phrases can be simple or complex, and they give us a nice window onto our thinking about objects.

9.1.1 Noun phrases

Recall that in our introduction to expressing knowledge in FOL-like languages (Chapter 3), we represented categories of objects with one-place predicates using common nouns like Company(x), Knife(x), Contract(x). But there is more to noun phrases than just nouns. To capture more interesting types of nominal constructions, such as "a hunter-gatherer" or "a father whose children are all doctors," we would need predicates with internal structure.

For example, if we had a truly compound predicate like Hunter&Gatherer(x),[1] then we would expect that for any x for which Hunter&Gatherer(x) was true, both Hunter(x) and Gatherer(x) would also be true. Most importantly, this connection among the three predicates would hold not by virtue of some fact about the world, but by *definition* of what we meant by the compound predicate.

Similarly, we would expect that if Child(x,y) and FatherOfOnlyDoctors(x) were both true, y would have to be a doctor, again (somehow), by definition. Note that this would be so even if we had a simple name that served as an abbreviation for a concept like this, which is very often the case in natural language (e.g., Teenager is synonymous with PersonWithAgeBetween13and19).

[1]We are using the "&" and complex predicate names suggestively here; we will introduce formal machinery shortly.

Traditional first-order logic does not provide any tools for dealing with compound predicates of this sort. In a sense, the only noun phrases in FOL are the nouns. But given the prominence and naturalness of such constructs in natural language, it is worthwhile considering KR machinery that does provide such tools. Since a logic that would allow us to manipulate complex predicates would be working mainly with *descriptions*, we call a logical system based on these ideas a *description logic* or DL.[2]

9.1.2 Concepts, roles, and constants

Looking at the examples above, we can already see that two sorts of nouns are involved: there are category nouns like Hunter, Teenager, and Doctor describing basic classes of objects, and there are relational nouns like Child and Age that describe objects that are parts or attributes or properties of other objects.[3] We saw a similar a distinction in Chapter 8 between a frame and a slot. In a description logic, we refer to the first type of description as a *concept* and the second type as a *role*.

As with frames, we will think of concepts as being organized into a generalization hierarchy where, for example, Hunter&Gatherer is a specialization of Hunter. However, we will see that much of the generalization hierarchy in a description logic follows logically from the meaning of the compound concepts involved, quite unlike the case with frames where hierarchies were stipulated by the user. And, as we will see, much of the reasoning performed by a description logic system centers around automatically computing this generalization relation.

For simplicity, we will not consider roles to be organized hierarchically in this way except briefly in Section 9.6. In contrast to the slots in frame systems, however, roles will be allowed to have multiple fillers. This way we can naturally describe a person with several children, a function with multiple arguments, or a wine made from more than one type of grape.

Finally, although much of the reasoning we perform in a description logic concerns generic categories, we will want to know how these descriptions apply to individuals as well. Consequently, we will also include *constants* like johnSmith in our description logic language below.

[2]Other names used in the literature include "terminological logics," "term subsumption systems," "taxonomic logics," or even "KL-One-like systems," because of their origin in early work on a system called KL-One.

[3]Many nouns can be used both ways. For example, "child" can mean a relation (the inverse of parent) or a category (a person of a young age).

9.2 A description language

We begin here with the syntax of a very simple but illustrative description logic language that we call $\mathcal{DL}$. Like FOL, $\mathcal{DL}$ has two types of symbols: logical symbols, which have a fixed meaning or use, and non-logical symbols, which are application-dependent. There are four sorts of logical symbols in $\mathcal{DL}$:

1. *punctuation*: "[", "]", "(", ")";
2. *positive integers*: 1,2, 3, *etc.*;
3. *concept-forming operators*: "**ALL**", "**EXISTS**", "**FILLS**", "**AND**";
4. *connectives*: "$\sqsubseteq$", "$\doteq$", "$\rightarrow$".

We distinguish three sorts of non-logical symbols in $\mathcal{DL}$:

1. *atomic concepts*, written in capitalized mixed case, e.g., Person, WhiteWine, FatherOfOnlyDaughters; $\mathcal{DL}$ also has a special atomic concept, Thing;
2. *roles*, written like atomic concepts, but preceded by ":", e.g., :Child, :Height, :Employer, :Arm;
3. *constants*, written in uncapitalized mixed case, e.g., table13, maryAnnJones.

There are four types of legal syntactic expressions in $\mathcal{DL}$: *constants*, *roles* (both seen above), *concepts* and *sentences*. We use c and r to range over constants and roles respectively, d and e to range over concepts, and a to range over atomic concepts. The set of concepts of $\mathcal{DL}$ is the least set satisfying the following:

- every atomic concept is a concept;
- if r is a role and d is a concept, then [**ALL** r d] is a concept;
- if r is a role and n is a positive integer, then [**EXISTS** n r] is a concept;
- if r is a role and c is a constant, then [**FILLS** r c] is a concept;
- if $d_1 \ldots d_n$ are concepts, then [**AND** $d_1 \ldots d_n$] is a concept.

Finally, there are three types of sentences in $\mathcal{DL}$:

- if d_1 and d_2 are concepts then $(d_1 \sqsubseteq d_2)$ is a sentence;
- if d_1 and d_2 are concepts, then $(d_1 \doteq d_2)$ is a sentence;
- if c is a constant and d is a concept, then $(c \rightarrow d)$ is a sentence.

A KB in a description logic like $\mathcal{DL}$ is considered to be any collection of sentences of this form.

What are these syntactic expressions supposed to mean? Constants are intended to stand for individuals in some application domain as they did in FOL; atomic concepts (and indeed all concepts in general) are intended to stand for categories or classes of individuals; and roles are intended to stand for binary relations over those individuals.

As for the complex concepts, their meanings are derived from the meanings of their parts the way the meanings of noun phrases are. Imagine that we have a role r standing for some binary relation. Then the concept [**EXISTS** n r] stands for the class of individuals in the domain that are related by relation r to at least n other individuals. So the concept [**EXISTS** 1 :Child] could represent someone who was not childless. Next, imagine that constant c stands for some individual; then the concept [**FILLS** r c] stands for those individuals that are r-related to that individual. So [**FILLS** :Cousin vinny] would represent someone one of whose cousins was Vinny. Next, imagine that concept d stands for some class of individuals; then the concept [**ALL** r d] stands for those individuals who are r-related only to elements of that class. So [**ALL** :Employee UnionMember] describes something whose employees, if any, are all union members. Finally, the concept [**AND** $d_1 \ldots d_n$] stands for anything that is described by d_1 and $\ldots d_n$.

Turning now to sentences, these expressions are intended to be true or false in the domain, as they would be in FOL. Imagine that we have two concepts d_1 and d_2, standing for two classes of individuals, and a constant c, standing for some individual. Then $(d_1 \sqsubseteq d_2)$ says that concept d_1 is *subsumed* by concept d_2, i.e., all individuals that satisfy d_1 also satisfy d_2. For example, (Surgeon $\sqsubseteq$ Doctor) says that any surgeon is also a doctor (among other things). Similarly, $(d_1 \doteq d_2)$ will mean that the two concepts are *equivalent*, i.e., the individuals that satisfy d_1 are precisely those that satisfy d_2. This is just a convenient way of saying that both $(d_1 \sqsubseteq d_2)$ and $(d_2 \sqsubseteq d_1)$ are true. Finally, $(c \rightarrow d)$ says that the individual denoted by c satisfies the description expressed by concept d.

While the sentences of $\mathcal{DL}$ are all atomic, it is easy to create complex concepts. For example,

```
[AND Wine
     [FILLS :Color red]
     [EXISTS 2 :GrapeType]]
```

would represent the category of a blended red wine (literally, a wine one of whose colors is red and which has at least two types of grape in it).

A typical sentence in a description logic KB is one that assigns a name to a complex concept:

```
(ProgressiveCompany ≐ [AND Company
                          [EXISTS 7 :Director]
                          [ALL :Manager [AND Woman
                                             [FILLS :Degree phD]]]
                          [FILLS :MinSalary $24.00/hour]])
```

The concept on the right-hand side represents the notion of a company with at least seven directors, and all of whose managers are women with a Ph.D., and whose minimum salary is $24.00/hour. The sentence as a whole says that ProgressiveCompany, as a concept, is equivalent to the one on the right. If this sentence is in a KB, we consider ProgressiveCompany to be fully *defined* in the KB, that is, we have a set of necessary and sufficient conditions for being a ProgressiveCompany, exactly expressed by the right-hand side. If we used the $\sqsubseteq$ connective instead, the sentence would say only that ProgressiveCompany as a concept was subsumed by the one on the right. Without a $\doteq$ sentence in the KB defining it, we consider ProgressiveCompany to be a *primitive* concept in that we only have necessary conditions it must satisfy. As a result, while we could draw conclusions about an individual ProgressiveCompany once we were told it was one, we would not have a way to recognize an individual as a ProgressiveCompany.

9.3 Meaning and Entailment

As we saw in the previous section, there are four different sorts of syntactic expressions in a description logic—constants, roles, concepts, and sentences—with different intended uses. In this section, we will explain precisely what these expressions are supposed to mean, and under what circumstances a collection of sentences in this logic entails another. As in ordinary FOL, it is this entailment relation that a description logic reasoner will be required to calculate.

9.3.1 Interpretations

The starting point for the semantics of description logics is the *interpretation*, just as it was for FOL. An interpretation $\Im$ for $\mathcal{DL}$ is a pair $\langle \mathcal{D}, \mathcal{I} \rangle$ as before, where $\mathcal{D}$ is any set of objects called the *domain* of the interpretation, and $\mathcal{I}$ is a mapping called the *interpretation mapping* from the non-logical symbols of $\mathcal{DL}$ to elements and relations over $\mathcal{D}$, where

1. for every constant symbol c, $\mathcal{I}[c] \in \mathcal{D}$;
2. for every atomic concept a, $\mathcal{I}[a] \subseteq \mathcal{D}$;
3. for every role symbol r, $\mathcal{I}[r] \subseteq \mathcal{D} \times \mathcal{D}$.

Comparing this to FOL, we can see that constants have the same meaning as they would as terms in FOL, that atomic concepts are understood as unary predicates, and that roles are understood as binary predicates. The set $\mathcal{I}[d]$ associated with a concept d in an interpretation is called its *extension*.

As we have emphasized, a distinguishing feature of description logics is the existence of non-atomic concepts whose meanings are completely determined by the meanings of their parts. For example, the extension of [**AND** Doctor Female] is required to be the intersection of the extension of Doctor and that of Female. More generally, we can extend the definition of $\mathcal{I}$ to all concepts as follows:

- for the distinguished concept Thing, $\mathcal{I}$[Thing] = $\mathcal{D}$;
- $\mathcal{I}$[[**ALL** r d]] = $\{x \in D \mid$ for any y, if $\langle x, y\rangle \in \mathcal{I}[r]$, then $y \in \mathcal{I}[d]\}$;
- $\mathcal{I}$[[**EXISTS** n r]] =
  $\{x \in D \mid$ there are at least n distinct y such that $\langle x, y\rangle \in \mathcal{I}[r]\}$;
- $\mathcal{I}$[[**FILLS** r c]] = $\{x \in D \mid \langle x, \mathcal{I}[c]\rangle \in \mathcal{I}[r]\}$;
- $\mathcal{I}$[[**AND** $d_1 \ldots d_n$]] = $\mathcal{I}[d_1] \cap \ldots \cap \mathcal{I}[d_n]$.

So if we are given an interpretation $\Im$, with an interpretation mapping for constants, atomic concepts, and roles, these rules tell us how to find the extension of any concept.

9.3.2 Truth in an interpretation

Given an interpretation, we can now specify which sentences of $\mathcal{DL}$ are true and which are false according to that interpretation. A sentence $(c \rightarrow d)$ will be true when the object denoted by c is in the extension of d; a sentence $(d \sqsubseteq d')$ will be true when the extension of d is a subset of the extension of d'; a sentence $(d \doteq d')$ will be true when the extension of d is the same as that of d'. More formally, given an interpretation $\Im = \langle \mathcal{D}, \mathcal{I} \rangle$, we say that α is *true* in $\Im$, written $\Im \models \alpha$, according to these rules:

Assume that d and d' are concepts, and that c is a constant.

1. $\Im \models (c \rightarrow d)$ iff $\mathcal{I}[c] \in \mathcal{I}[d]$;

2. $\Im \models (d \sqsubseteq d')$ iff $\mathcal{I}[d] \subseteq \mathcal{I}[d']$;
3. $\Im \models (d \doteq d')$ iff $\mathcal{I}[d] = \mathcal{I}[d']$.

As in FOL, we will also use the notation $\Im \models S$, where S is a set of sentences, to mean that all the sentences in S are true in $\Im$.

9.3.3 Entailment

The definition of entailment in $\mathcal{DL}$ is exactly like it is in FOL. Let S be a set of sentences, and α any individual sentence. We say that S logically *entails* α, which we write $S \models \alpha$, if and only if for every interpretation $\Im$, if $\Im \models S$ then $\Im \models \alpha$. As a special case of this definition, we say that a sentence α is logically *valid*, which we write $\models \alpha$, when it is logically entailed by the empty set.

There are two basic sorts of reasoning we will be concerned with in description logics:[4] determining whether or not some constant c satisfies a certain concept d, and determining whether or not a concept d is subsumed by another concept d'. Both of these involve calculating entailments of a KB: in the first case, we need to determine if the KB entails $(c \rightarrow d)$, and in the second case, if the KB entails $(d \sqsubseteq d')$. So, as in FOL, reasoning in a description logic means calculating entailments.

Note that in some cases, the entailment relationship will hold because the sentences themselves are valid. For example, consider the sentence

([**AND** Doctor Female] $\sqsubseteq$ Doctor]).

This sentence is valid according to the definition above: the sentence must be true in every interpretation $\Im$ because no matter what extension it assigns to Doctor and Female, the extension of the **AND** concept (which is the intersection of the two sets) will always be a subset of the extension of Doctor. Consequently, for any KB, the first concept is subsumed by the second—in other words, a female doctor is always a doctor. Similarly, the sentence

(john $\rightarrow$ Thing)

is valid: the sentence must be true in every interpretation $\Im$ because no matter what extension it assigns to john, it must be an element of $\mathcal{D}$, which is the extension of Thing. Consequently, for any KB, the constant satisfies that concept—in other words, the individual John is always something.

[4]We will see in Section 9.6 that other useful varieties of reasoning reduce to these two.

In more typical cases, the entailment relationship will depend on the sentences in the KB. For example, if a knowledge base, KB, contains the sentence

(Surgeon $\sqsubseteq$ Doctor),

then we get the following entailment:

KB $\models$ ([**AND** Surgeon Female] $\sqsubseteq$ Doctor).

To see why, consider any interpretation $\Im$, and suppose that $\Im \models$ KB. Then for this interpretation, the extension of Surgeon is a subset of that of Doctor, and so the extension of the **AND** concept (that is, the intersection of the extensions of Surgeon and Female) must also be a subset of that of Doctor. So for this KB, the first concept is subsumed by the second—if a surgeon is a doctor (among other things), then a female surgeon is also a doctor. This conclusion would also follow if instead of (Surgeon $\sqsubseteq$ Doctor), the KB were to contain

(Surgeon $\doteq$ [**AND** Doctor [**FILLS** :Specialty surgery]]).

In this case we are defining a surgeon to be a certain kind of doctor, which again requires the extension of Surgeon to be a subset of that of Doctor. With the empty KB on the other hand, there would be no subsumption relation since we can find an $\Im$ where the extension of the first concept is not a subset of the second: let $\mathcal{D}$ be the set of all integers, and let $\mathcal{I}$ assign Doctor to the empty set, and both Surgeon and Female to the set of all integers.

9.4 Computing entailments

As stated above, there are two major types of reasoning that we care about with a description logic: given a knowledge base, KB, we want to be able to determine if KB $\models \alpha$, for sentences α of the form,[5]

- $(c \rightarrow d)$, where c is a constant and d is a concept; and
- $(d \sqsubseteq e)$, where d and e are both concepts.

In fact, the first of these is easy to handle once we deal with the second, and so we begin by considering how to compute subsumption. As with Resolution for FOL, the key fact about this symbol-level computation we are about to present is that it is correct relative to the knowledge-level definition of entailment given above.

[5]As we have mentioned, KB $\models (d \doteq e)$ iff KB $\models (d \sqsubseteq e)$ and KB $\models (e \sqsubseteq d)$.

9.4.1 Simplifying the knowledge base

Observe first of all that subsumption entailments are unaffected by the presence of sentences of the form $(c \rightarrow d)$ in the KB. In other words, if KB$'$ is just like KB except that all the $(c \rightarrow d)$ sentences have been removed, then it can be shown that KB $\models (d \sqsubseteq e)$ if and only if KB$' \models (d \sqsubseteq e)$. So we can assume that for subsumption questions, the KB in question contains no $(c \rightarrow d)$ sentences.

For pragmatic purposes, it is useful to make a further restriction: we insist that the left-hand sides of the $\sqsubseteq$ and $\doteq$ sentences in the KB be atomic concepts other than Thing and that each atom appears on the left-hand side of a sentence exactly once in the KB. We can think of such sentences as providing either a definition of the atomic concept (in the case of $\doteq$) or its necessary conditions (in the case of $\sqsubseteq$). We will, however, still be able to compute KB $\models \alpha$ for sentences α of the more general form above (e.g., subsumption between two complex concepts).

Finally, we assume that the $\sqsubseteq$ and $\doteq$ sentences in the KB are *acyclic*. Informally we want to rule out a KB like

$$\{\ (d_1 \doteq [\textbf{AND}\ d_2\ \ldots]),\ \ (d_2 \sqsubseteq [\textbf{ALL}\ r\ d_3]),\ \ (d_3 \sqsubseteq d_1)\ \}$$

which has a cycle (d_1, d_2, d_3, d_1). While this type of cycle is meaningful in our semantics, it complicates the calculation of subsumption.

With these restrictions in place, to determine whether or not KB $\models (d \sqsubseteq e)$ it will be sufficient to do the following:

1. using the definitional declarations ($\doteq$) in KB, put d and e into a special normalized form;
2. using the subsumption declarations ($\sqsubseteq$) in KB, determine whether each part of the normalized e is implied by some part of the normalized d.

So subsumption in a description logic KB reduces to a question about a structural relationship between two normalized concepts.[6]

9.4.2 Normalization

Normalization in description logics is similar in spirit to the derivation of normal forms like CNF in FOL. During this phase, we draw some inferences, but only small, obvious ones. This pre-processing then makes the subsequent structure-matching step to follow straightforward.

[6]There are other ways of computing subsumption; this is probably the most common and direct way that takes concept structure into account.

Normalization applies to one concept at a time, and involves a small number of steps. Here we outline the steps and then review the whole process on a larger expression.

1. **expand definitions:** Any atomic concept that appears as the left-hand side of a $\doteq$ sentence in the KB is replaced by its definition. For example, if we have the following sentence in KB,

   $$(\text{Surgeon} \doteq [\textbf{AND}\ \text{Doctor}\ [\textbf{FILLS}\ \text{:Specialty surgery}]])$$

   then the concept [**AND** ... Surgeon ...] expands to

   $$[\textbf{AND}\ \ldots\ [\textbf{AND}\ \text{Doctor}\ [\textbf{FILLS}\ \text{:Specialty surgery}]]\ \ldots].$$

2. **flatten the AND operators:** Any subconcept of the form

   $$[\textbf{AND}\ldots\ [\textbf{AND}\ d_1\ \ldots\ d_n]\ldots]$$

   can be simplified to $[\textbf{AND}\ldots\ d_1\ \ldots\ d_n \ldots]$.

3. **combine the ALL operators:** Any subconcept of the form

   $$[\textbf{AND}\ldots[\textbf{ALL}\ r\ d_1]\ldots[\textbf{ALL}\ r\ d_2]\ldots],$$

   can be simplified to $[\textbf{AND}\ldots[\textbf{ALL}\ r\ [\textbf{AND}\ d_1\ d_2]]\ \ldots]$.

4. **combine EXISTS operators:** Any subconcept of the form

   $$[\textbf{AND}\ldots[\textbf{EXISTS}\ n_1\ r]\ldots[\textbf{EXISTS}\ n_2\ r]\ldots]$$

   can be simplified to the concept $[\textbf{AND}\ldots[\textbf{EXISTS}\ n\ r]\ \ldots]$, where n is the maximum of n_1 and n_2.

5. **deal with Thing:** Certain concepts are vacuous and should be removed as an argument to **AND**: Thing, [**ALL** r Thing], and **AND** with no arguments. In the end, the concept Thing should only appear if this is what the entire expression simplifies to.

6. **remove redundant expressions:** Eliminate any expression that is an exact duplicate of another within the same **AND** expression.

To normalize a concept, these operations can be applied repeatedly in any order, and at any level of embedding within **ALL** and **AND** operators. However, the process only terminates when no further steps are applicable.

In the end, the result of a normalization is either Thing or a concept of the following form:

[**AND** $a_1 \ldots a_m$
  [**FILLS** $r_1\ c_1$] ... [**FILLS** $r_{m'}\ c_{m'}$]
  [**EXISTS** $n_1\ s_1$] ... [**EXISTS** $n_{m''}\ s_{m''}$]
  [**ALL** $t_1\ e_1$] ... [**ALL** $t_{m'''}\ e_{m'''}$]]

where the a_i are primitive atomic concepts other than Thing, the r_i, s_i and t_i are roles, the n_i are positive integers, and the e_i are themselves normalized concepts. In fact, we can think of Thing itself as the same as [**AND**]. We call the arguments to **AND** in a normalized concept the *components* of the concept.

To illustrate the normalization process, we consider an example. Assume that KB has the following definitions:

WellRoundedCo $\doteq$
  [**AND** Company [**ALL** :Manager [**AND** B-SchoolGrad
    [**EXISTS** 1 :TechnicalDegree]]]]
HighTechCo $\doteq$
  [**AND** Company [**FILLS** :Exchange nasdaq] [**ALL** :Manager Techie]]

Techie $\doteq$ [**EXISTS** 2 :TechnicalDegree]

These definitions amount to a WellRoundedCo being a company whose managers are business school graduates who each have at least one technical degree, a HighTechCo being a company listed on the NASDAQ whose managers are all Techies, and a Techie being someone with at least two technical degrees.

Given these definitions, let us examine how we would normalize the expression

[**AND** WellRoundedCo HighTechCo].

First, we would expand the definitions of WellRoundedCo and HighTechCo, and then, Techie, yielding this:

[**AND** [**AND** Company
    [**ALL** :Manager [**AND** B-SchoolGrad
      [**EXISTS** 1 :TechnicalDegree]]]]
  [**AND** Company
    [**FILLS** :Exchange nasdaq]
    [**ALL** :Manager [**EXISTS** 2 :TechnicalDegree]]]]

Next, we flatten the **AND** operators at the top level and then combine the **ALL** operators over :Manager:

[**AND** Company
  [**ALL** :Manager [**AND** B-SchoolGrad
    [**EXISTS** 1 :TechnicalDegree]
    [**EXISTS** 2 :TechnicalDegree]]]
  Company
  [**FILLS** :Exchange nasdaq]]

Finally, we remove the redundant Company concept and combine the **EXISTS** operators over :TechnicalDegree, yielding the following:

[**AND** Company
  [**ALL** :Manager [**AND** B-SchoolGrad [**EXISTS** 2 :TechnicalDegree]]]
  [**FILLS** :Exchange nasdaq]]

This is the concept of a company listed on the NASDAQ exchange whose managers are business school graduates with at least two technical degrees.

9.4.3 Structure matching

In order to compute whether KB $\models (d \sqsubseteq e)$, we need to compare the normalized versions of d and e. The idea behind structure-matching is that for d to be subsumed by e, the normalized d must account for each component of the normalized e in some way. For example, if e has the component [**FILLS** :Color red], then d must contain this component too. If e has the component [**EXISTS** 3 :Child], then d must have a component [**EXISTS** n :Child], and we must have $n \geq 3$. If e contains the component [**ALL** $r\ e'$], then d must contain some [**ALL** $r\ d'$], where d' is subsumed by e'. Finally, if e contains some atomic concept a, there are two cases: either d contains a itself, or d contains some a' such that $(a' \sqsubseteq a)$ is derivable using the $\sqsubseteq$ sentences in the KB. The full procedure for structure matching is shown in Figure 9.1.

To illustrate briefly the structure-matching algorithm, consider the concept, d,

[**AND** LegalEntity [**ALL** :Manager B-SchoolGrad]].

Assume that the declaration, (Company $\sqsubseteq$ LegalEntity), exists in KB. In this case, d can be seen to subsume the one that resulted from the normalization procedure above (call it d') by looking at each of d's two components, and seeing that there exists in d' a matching component:

Figure 9.1: A procedure for structure matching

Input: Two normalized concepts d and e where
d is [**AND** $d_1 \ldots d_m$] and e is [**AND** $e_1 \ldots e_{m'}$]
Output: yes or no, according to whether KB $\models (d \sqsubseteq e)$

Return yes iff for each component e_j, for $1 \leq j \leq m'$, there exists a component d_i where $1 \leq i \leq m$, such that d_i *matches* e_j, as follows:

1. if e_j is an atomic concept, then either d_i is identical to e_j, or there is a sentence of the form $(d_i \sqsubseteq d')$ in the KB, where recursively some component of d' matches e_j;
2. if e_j is of the form [**FILLS** r c], then d_i must be identical to it;
3. if e_j is of the form [**EXISTS** n r], then the corresponding d_i must be of the form [**EXISTS** n' r], for some $n' \geq n$; in the case where n = 1, the matching d_i can be of the form [**FILLS** r c], for any constant c;
4. if e_j is of the form [**ALL** r e'], then the corresponding d_i must be of the form [**ALL** r d'], where recursively d' is subsumed by e'.

- LegalEntity is an atomic concept; there is a component in d' (Company), where there is an appropriate sentence in the KB that satisfies the requirement in Step 1 of the algorithm (namely, (Company $\sqsubseteq$ LegalEntity)).
- For the **ALL** component of d, whose restriction is B-SchoolGrad, there is an **ALL** component of d' such that the restriction on that **ALL** component is subsumed by B-SchoolGrad (namely the conjunction, [**AND** B-SchoolGrad [**EXISTS** 2 :TechnicalDegree]]).

9.4.4 Computing satisfaction

Computing whether an individual denoted by a constant satisfies a concept is very similar to computing subsumption between two concepts. The main difference is that we need to take the $\rightarrow$ sentences in the KB into account. More precisely, it can be shown that KB $\models (c \rightarrow e)$ if and only if KB $\models (d \sqsubseteq e)$, where d is the **AND** of every concept d_i such that $(c \rightarrow d_i)$ is in the KB. What this means is that concept satisfaction can be computed directly using the procedure presented above for concept subsumption, once we gather together all the relevant $\rightarrow$ sentences in the KB.

9.4.5 The correctness of the subsumption computation

We conclude this section by claiming correctness for the procedure presented here: KB $\models (d \sqsubseteq e)$ (according to the definition in terms of interpretations) if and only if d normalizes to some d', e normalizes to some e', and for every component of e', there is a corresponding matching component of d' as above. We will not present a full proof since it is quite involved, but merely sketch the argument.

The first observation is that given a KB in the simplified form discussed in Section 9.4.1, every concept can be put into normal form, and moreover, each step of the normalization preserves concept equivalence. It follows that KB $\models (d \sqsubseteq e)$ if and only if KB $\models (d' \sqsubseteq e')$.

The next part of the proof is to show that if the procedure returns yes given d' and e', then KB $\models (d' \sqsubseteq e')$. So suppose that each component of e' has a corresponding component in d'. To show subsumption, imagine that we have some interpretation $\Im = \langle \mathcal{D}, \mathcal{I} \rangle$ and some $x \in \mathcal{D}$ such that $x \in \mathcal{I}[d']$. To prove that $x \in \mathcal{I}[e']$ (and consequently that d' is subsumed by e'), we look at the various components e_j of e' case by case, and show that $x \in \mathcal{I}[e_j]$ because there is a matching d_i in d' and $x \in \mathcal{I}[d_i]$.

The final part of the proof is the trickiest. We must show that if the procedure returns no, then it is not the case that KB $\models (d' \sqsubseteq e')$. To do so, we need to construct an interpretation where for some $x \in \mathcal{D}$, $x \in \mathcal{I}[d']$ but $x \notin \mathcal{I}[e']$.

Here is how to do so in the simplest case where there are no $\sqsubseteq$ sentences in the KB, and no **EXISTS** concepts involved. Let the domain $\mathcal{D}$ be the set of all constants together with the set of *role chains* defined to be all sequences of roles (including the empty sequence). Then for every constant c, let $\mathcal{I}[c]$ be c; for every atomic concept a, let $\mathcal{I}[a]$ be all constants and all role chains σ where $\sigma = r_1 \cdots r_k$ where $k \geq 0$ and such that d' is of the form

$$[\textbf{AND} \ldots [\textbf{ALL}\ r_1 \ldots [\textbf{AND} \ldots [\textbf{ALL}\ r_k\ a] \ldots] \ldots] \ldots];$$

finally, for every role r, let $\mathcal{I}[r]$ be every pair of constants, together with every pair $(\sigma, \sigma{\cdot}r)$ where σ is a role chain, together with every pair (σ, c) where c is a constant, $\sigma = r_1 \cdots r_k$ where $k \geq 0$, and such that d' is of the form

$$[\textbf{AND} \ldots [\textbf{ALL}\ r_1 \ldots [\textbf{AND} \ldots [\textbf{ALL}\ r_k\ [\textbf{FILLS}\ r\ c]] \ldots] \ldots] \ldots].$$

Assuming the procedure returns no, it can be shown for this interpretation that the empty role chain is in the extension of d', but not in the extension of e', and consequently that d' does not subsume e'. We omit all further details.

9.5 Taxonomies and classification

In practice, there are a small number of key questions that would typically be asked of a description logic KB. Since these KBs resemble databases, where the concepts correspond roughly to elements of a schema and constants correspond to records, it is common to ask for all of the instances of a concept:

given some query concept, q, find all c in KB such that KB $\models (c \rightarrow q)$.

On the other hand, since these KB's resemble frame systems in some ways, it is common to ask for all of the known categories that an individual satisfies, in order, for example, to trigger procedures associated with those classes:

given a constant c, find all atomic concepts a such that KB $\models (c \rightarrow a)$.

While the logic and computational methods we have presented so far are adequate for finding the answers to these questions, a naive approach might consider doing a full scan of the KB, requiring time that grows *linearly* with the number of sentences in the KB. However, one of the key reasons for using a description logic in the first place is to exploit the fact that concepts are naturally thought of as organized hierarchically, with the most general ones at the top, and the more specialized ones further down. In this section, we will consider a special tree-like data structure that we call a *taxonomy* for representing sentences in a description logic KB. This taxonomy will allow us to answer queries like the above much more efficiently, requiring time that in many cases grows only *logarithmically* with the number of sentences in the KB. The net result: it becomes practical to consider extremely large knowledge bases, with thousands or even millions of concepts and constants.

9.5.1 A taxonomy of atomic concepts and constants

The key observation is that subsumption is a partial order, and a taxonomy naturally falls out of any given set of concepts. Assume that $a_1, \ldots, a_n$ are all the atomic concepts that occur on the left-hand sides of $\doteq$ or $\sqsubseteq$ sentences in KB. The resultant taxonomy will have nodes for each of the a_i, and edges from a_i up to a_j, whenever a_i is less general that a_j, but not less general than anything more specific than a_j. This will produce a directed acyclic graph. The graph will have no redundant links

in it, and the transitivity of the links will capture all of the subsumption relationships implied by the declarations defining a_i. If we add to this the requirement that each constant c in KB be linked only to the most specific a_i such that KB $\models (c \rightarrow a_i)$, we have a hierarchical representation of KB that makes our key questions easier to answer.[7]

Once we have a taxonomy of concepts corresponding to some KB, we can consider adding a sentence to the KB for some new atomic concept or constant. This will involve creating some links from the new concept or constant to existing ones in the taxonomy, and perhaps redirecting some existing links. This process is called *classification*. Because classification itself exploits the structure of the taxonomy, the process requires time that is typically logarithmic in the size of the KB. Furthermore, we can think of building the entire taxonomy by classification: we start with a single concept Thing in the taxonomy, and then add new atomic concepts and constants to it incrementally.

9.5.2 Computing classification

We begin by considering how to add a sentence $(a_{new} \doteq d)$ to a taxonomy where a_{new} is an atomic concept not appearing anywhere in the KB and d is any concept:

1. We first calculate S, the *most specific subsumers* of d, that is, the set of atomic concepts a in the taxonomy such that KB $\models (d \sqsubseteq a)$, but such that there is no a' other than a such that KB $\models (d \sqsubseteq a')$ and KB $\models (a' \sqsubseteq a)$. We will see how to do this efficiently below.

2. We next calculate G, the *most general subsumees* of d, that is, the set of atomic concepts a in the taxonomy such that KB $\models (a \sqsubseteq d)$, but such that there is no a' other than a such that KB $\models (a' \sqsubseteq d)$ and KB $\models (a \sqsubseteq a')$. We will also see how to do this efficiently.

3. If there is a concept a in $S \cap G$, then the new concept a_{new} is already present in the taxonomy under a different name (namely, a), and we have handled this case.

4. Otherwise, if there are any links from concepts in G up to concepts in S, we remove them, since we will be putting a_{new} between the two groups.

5. We add links from a_{new} up to each concept in S, and links from each concept in G up to a_{new}.

[7]We assume that with each node in the taxonomy, we also store the concept making up the right-hand side of the sentence it appeared in.

6. Finally we handle constants: we calculate C, the set of constants c in the taxonomy such that for every $a \in S$, KB $\models (c \rightarrow a)$, but such that there is no $a' \in G$ such that KB $\models (c \rightarrow a')$. (This is done by doing intersections and set differences on the sets of constants below concepts in the obvious way.) Then, for each $c \in C$, we test if KB $\models (c \rightarrow d)$, and if so, we remove the links from c to the concepts in S, and add a single link from c up to a_{new}.

To add a sentence $(a_{new} \sqsubseteq d)$ to a taxonomy, the procedure is similar, but simpler. Because a_{new} is a new primitive, there will be no concepts or constants below it in the taxonomy. So we need only link a_{new} up to the most specific subsumers of d. Similarly, to add a sentence $(c_{new} \rightarrow d)$, we again link c_{new} up to the most specific subsumers of d.

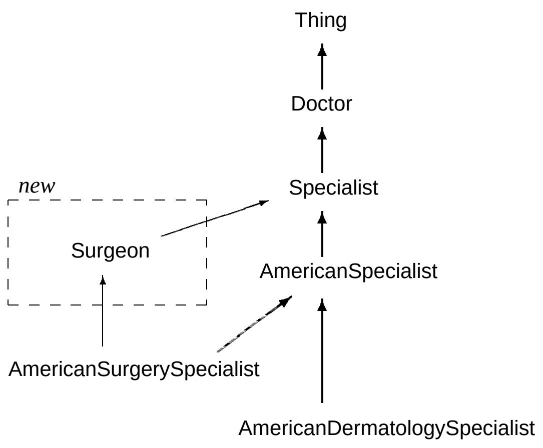
Now, to calculate the most specific subsumers of a concept d, we begin at the very top of the taxonomy with the set $\{$Thing$\}$ as our first S. Assume we have a list S of subsumers of d. Suppose that some $a \in S$ has at least one child a' immediately below it in the taxonomy such that KB $\models (d \sqsubseteq a')$. Then we remove a from S and replace it with all those children a'. We keep doing this until no element of S has a child that subsumes d.

Observe that if we have an atomic concept a' below $a \in S$ that does *not* subsume d, then we will not use any other concept below this a' during the classification. If a' is high enough in the taxonomy, like just below Thing, an entire subtree can be safely ignored. This is the sense in which the structure of the taxonomy allows us to do classification efficiently even for very large knowledge bases.

Finally, to calculate the most general subsumees of a concept d, we start with the most specific subsumers S as our first G (again, other distant parts of the taxonomy will not be used). Now suppose that for some $a \in G$, it is not the case that KB $\models (a \sqsubseteq d)$; then we replace this a in G by its children in the taxonomy (and if a had no children, we simply delete it). We keep doing this, working our way down the taxonomy, until every element of G is subsumed by d.

Following this procedure, Figure 9.2 shows how a new concept, Surgeon, defined by the sentence (Surgeon $\doteq$ [**AND** Doctor [**FILLS** :Specialty surgery]]), can be classified, given a taxonomy that already includes appropriate definitions for concepts like Doctor, AmericanSpecialist, *etc.* First, we calculate the most specific subsumers of Surgeon, S. We start with S = $\{$Thing$\}$. Assume that none of the direct subsumees of Thing except for Doctor subsume Surgeon. Given that, and the fact that (Surgeon $\sqsubseteq$ Doctor), we replace Thing in the set S by Doctor. The concept Specialist is immediately below Doctor, and (Surgeon $\sqsubseteq$ Specialist), so we then replace Doctor in S with Specialist. Finally, we see that no child of Specialist subsumes Surgeon (*i.e.*, not all surgeons are American specialists), so we have

Figure 9.2: Classifying a new concept in a taxonomy



computed the set of most specific subsumers, S = $\{$Specialist$\}$.

Now we turn our attention to the most general subsumees. We start with G = S = $\{$Specialist$\}$. It is not the case that (Specialist $\sqsubseteq$ Surgeon), so we replace Specialist in G with its one child in the taxonomy; now G = $\{$AmericanSpecialist$\}$. Similarly, it is not the case that (AmericanSpecialist $\sqsubseteq$ Surgeon), so we replace that concept in G with its children, resulting in G = $\{$AmericanDermatologySpecialist, AmericanSurgerySpecialist$\}$. Then, since AmericanDermatologySpecialist is not subsumed by Surgeon, and that concept has no children, it is deleted from G. Finally, we see that it is the case that (AmericanSurgerySpecialist $\sqsubseteq$ Surgeon), and we are done, with G = $\{$AmericanSurgerySpecialist$\}$. As a result of this classification process, the new concept, Surgeon, is placed between the two concepts Specialist and AmericanSurgerySpecialist.

9.5.3 Answering the questions

If we construct, in the above manner, a taxonomy corresponding to a knowledge base, we are left in a position to answer the key description logic questions quite easily. To find all of the constants that satisfy a query concept, q, we simply classify

q, and then collect all constants at the fringe of the tree below q. This would involve a simple tree walk in only the part of the taxonomy subtended by q. Similarly, to find all atomic concepts that are satisfied by a constant c, we start at c and walk up the tree, collecting all concept nodes that can be reached by following the links representing subsumption.

9.5.4 Taxonomies vs. frame hierarchies

The taxonomies we derive by classification in a description logic KB look a lot like the hierarchies of frames we encountered in the preceding chapter. In the case of frames, the KB designer could create the hierarchy in any arbitrary way desired, simply by adding whatever :IS-A and :INSTANCE-OF slot-fillers seemed appropriate. However, with DL's, the logic of concepts strictly dictates what each concept means, as well as what must be above or below it in the resulting taxonomy. As a result, we cannot just throw labeled nodes together in a hierarchy, or arbitrarily change a taxonomy—we must honor the relationships implicit in the structures of the concepts. A concept of the form [**AND** Fish [**FILLS** :Size large]. . .] *must* appear in a taxonomy below Fish, even if we originally constructed it to be the referent of Whale. If we at some point realized that that was an inaccurate rendition of Whale, what would have to be changed is the association of the symbol Whale with the expression, changing it to perhaps [**AND** Mammal [**FILLS** :Size large]. . .]. But the compound concept with Fish in it could not possibly go anywhere in the taxonomy but under Fish.

9.5.5 Inheritance and propagation

Recall that in our Frames chapter (Chapter 8) we introduced the notion of *inheritance*, whereby individual frames were taken to have values (and attached procedures) represented in parent frames somewhere up the generalization hierarchy. The same phenomenon can be seen here with description logic taxonomies: a constant in the taxonomy should be taken as having all properties (as expressed by **FILLS**, **ALL**, and **EXISTS**) that appear both on it locally (as part of the right-hand side of the sentence where it was first introduced) as well as on any parent concept further up the taxonomy.

Inheritance here tends to be much simpler than inheritance found in most frame systems, since it is *strict*—there are no exceptions permitted by the logic of the concept-forming operators. It is important to note, though, that these inferences are sanctioned by the logic, and issues of how to compute them using the taxonomy are purely implementation considerations. We will return to a much richer notion of

inheritance in the next chapter.

Another important inference in practical description logic systems involves the *propagation* of properties to an individual caused by an assertion. We are imagining, in other words, that we can add a sentence ($c \rightarrow d$) to the KB even if we had already previously classified c. This can then cause other constants to be reclassified. For example, suppose we introduce Lauren with the sentence (lauren $\rightarrow$ [**FILLS** :Child rebecca]), and we define ParentOfDocs by

$$(\text{ParentOfDocs} \doteq [\textbf{ALL}\ \text{:Child Doctor}]).$$

Then as soon as it is asserted that (lauren $\rightarrow$ ParentOfDocs), we are forced to conclude that Rebecca is a doctor. If we also knew that (rebecca $\rightarrow$ Woman), and we had the atomic concept FemaleDoc defined as [**AND** Woman Doctor], then the assertion about Lauren should result in Rebecca being reclassified as a FemaleDoc.

This kind of cascaded inference is interesting in applications where membership in classes is monitored, and changes in class membership are considered significant (e.g., imagine we are monitoring the stock market and have classes representing stocks whose values are changing in significant ways). It is also reminiscent of the kind of cascaded computation we saw with frame systems, except that here again the computations are dictated by the logic.

9.6 Beyond the basics

In this final section, we examine briefly how we can move beyond the simple picture of description logics presented so far.

9.6.1 Extensions to the language

First, we consider some extensions to $\mathcal{DL}$ that would make it more useful. Each of the extensions ends up having serious consequences for computing subsumption. In many cases, it is no longer possible to use normalization and structure matching to do the job; in some cases, subsumption can even be shown to be undecidable.[8]

Bounds on the number of role fillers: The $\mathcal{DL}$ construct **EXISTS** is used to say that a role has a minimum number of fillers. We can think of the dual operator **AT-MOST** where [**AT-MOST** n r] describes individuals related by role r to *at most* n individuals. This seemingly small addition to $\mathcal{DL}$ in fact allows a wide range of new inferences. First of all, we have descriptions like

$$[\textbf{AND}\ [\textbf{EXISTS}\ 4\ r]\ [\textbf{AT-MOST}\ r\ 3]]$$

[8] We will revisit this issue again in detail in Chapter 16.

which are *inconsistent* in that their extension is guaranteed to be the empty set. Moreover, a simple concept like [**ALL** r d] now subsumes one like

[**AND** [**FILLS** r c] [**AT-MOST** r 1] [**ALL** s d] [**FILLS** s c]]

even though there is no obvious structure to match.

We should also note that as soon as inconsistency is allowed into the language, computation gets complex. Besides the difficulties with structure-matching noted above, normalization suffers also. For example, if we have found d to be inconsistent, then although [**ALL** r d] is not inconsistent by itself, the result of conjoining it with [**EXISTS** 1 r] is inconsistent, and this would need to be detected during normalization.

Sets of individuals: Another important construct would package up a set of individuals into a set concept, which could then be used, for example, in restricting the values of roles. [**ONE-OF** c_1 c_2 $\ldots$ c_n] would be a concept that could only be satisfied by the c_i. In an **ALL** restriction, we might find such a set:

[**ALL** :BandMember [**ONE-OF** john paul george ringo]]

would represent the concept of something whose band members could only be taken from the specified set. Note that such a combination would have consequences for the cardinality of the BandMember role, implying [**AT-MOST** 4 BandMember], although it would imply nothing about the minimum number of band members.

Relating the roles: While we have discussed classes of objects with internal structure (via its roles), we have ignored a key ingredient of complex terms—how the role fillers actually interrelate. A simple case of this is when fillers for two roles are required to be identical. Consider a construct [**SAME-AS** r_1 r_2], which equates the fillers of r_1 and r_2. [**AND** Company [**SAME-AS** :CEO :President]] would thus mean a company whose CEO was identical to its President. Despite its apparent simplicity, without some restrictions, **SAME-AS** makes subsumption very difficult to compute. This is especially true if we allow a very natural extension to the **SAME-AS** construct—allowing it to take as arguments *chains* of roles, rather than single roles. In that case, [**SAME-AS** (:Mother :Sister)(:Father :Partner :Lawyer)] would represent something whose mother's sister is its father's partner's lawyer. Computation can be simplified by restricting **SAME-AS** to chains of "features" or "attributes"—roles that have exactly one filler.

Qualified number restrictions: Another natural extension to $\mathcal{DL}$ is what has been called a "qualified number restriction." [**EXISTS** n r d] would allow us to represent something that is r-related to n individuals who are also instances of d. For example, [**EXISTS** 2 :Child Female] would represent someone with at least

two daughters. This is a very natural and useful construct, but causes surprising computational difficulties, even if the rest of the language is kept very simple.

Complex roles: So far we have taken roles to be primitive atomic constructs. It is plausible to consider a logic of roles reminiscent of the logic of concepts. For example, some description logics have role-forming operators that construct *conjunctive roles* (much like **AND** over concepts). This would imply a role taxonomy akin to the concept taxonomy. Another extension that has been explored is that of *role inverses*. If we have introduced a role like :Parent, it is quite natural to think of introducing :Child to be defined as its inverse.

Rules: In $\mathcal{DL}$, there is no way to *assert* that all instances of one concept are also instances of another. Consider, for example, the concept of a red Bordeaux wine, which we might define as follows:

```
(RedBordeauxWine ≐ [AND Wine
                        [FILLS :Color red]
                        [FILLS :Region bordeaux]]).
```

We might also have the following concept:

```
(DryRedBordeauxWine ≐ [AND Wine
                           [FILLS :Color red]
                           [FILLS :Region bordeaux]
                           [FILLS :SugarContent dry]]),
```

These two concepts are clearly not equivalent. But suppose that we want to assert that all red Bordeaux wines are in fact dry. If we were to try to do this by using the second concept above as the definition of RedBordeauxWine, we would be saying in effect that red Bordeaux wines are dry *by definition*. In this case, the status of the first concept would be unclear: should the subsumption relation be changed somehow so that the two concepts end up being equivalent? To avoid this difficulty, we can keep the original definition of RedBordeauxWine, but extend $\mathcal{DL}$ with a simple form of *rules*, which capture universal assertions. A rule will have an atomic concept as its antecedent, and an arbitrary concept as its consequent:

(**if** RedBordeauxWine **then** [**FILLS** :SugarContent dry])

Rules of this sort give us a new and quite useful form of propagation: a constant gets classified, then inherits rules from concepts that it satisfies, which then are applied and yield new properties for the constant (and possibly other constants), which can then cause a new round of classification. This is reminiscent of the triggering of **IF-ADDED** procedures in frame systems, except that the classification is done automatically.

9.6.2 Applications of description logics

We now turn our attention to how description logic systems can be utilized in practical applications.

Assertion and query: One mode of use is the exploration of the consequences of axiomatizing a domain by describing it in a concept hierarchy. In this scenario, we generate a taxonomy of useful general categories, and then describe individuals in terms of those categories. The system then classifies the individuals according to the general scheme, and propagates to related individuals any new properties that they should accrue. We might then ask if a given individual satisfies a certain concept, or we might ask for the entire set of individuals satisfying a concept.

This would be appealing in a situation where a catalogue of products was described in terms of a complex domain model. The system may be able to determine that a product falls into some categories unanticipated by the user.

Another situation in which this style of interaction is important involves configuration of complex structured items. Asserting that a certain board goes in a certain slot of a computer hardware assembly could cause the propagation of constraints to other boards, power supplies, software, *etc*. The domain theory then acts as a kind of object-oriented constraint propagator. One could also ask questions about properties of an incrementally evolving configuration, or even "what if" questions.

Contradiction detection in configuration: Configuration-style applications can also make good use of contradiction-detection facilities for those DLs that have enough power to express them. In particular, as an incremental picture of the configured assembly evolves, it is useful to detect when a proposed part or subassembly violates some constraint expressed in the knowledge base. This keeps us from making invalid configurations. It is also possible to design explanation mechanisms so that the reasons for the violation can be outlined to the user.

Classification and contradiction detection in knowledge acquisition: In a similar way, some of the inferential properties of a description logic system can be used as partial validation during knowledge acquisition. As we add more concepts or constants to a DL knowledge base, a DL system will notice if any inconsistencies are introduced. This can alert us to mistakes. Because of its classification property, a DL can alert us to certain failures of domain modeling in a way that frame systems cannot, for example, the unintended merger of two concepts that look different on the surface but which mutually subsume one another, or the unintended classification of a new item below one that the user had not expected.

Assertion and classification in monitoring scenarios: In some applications, it is normal to build the description of an individual incrementally over time. This might be the case in a diagnosis scenario, where information about a suspected fault is gathered in pieces, or in a situation with a hardware device sending a stream of status and error reports. Such an incremental setting leads one to expect the refinement of classifications of individuals over time. If we are on the lookout for members of certain classes (e.g., Class1CriticalError), we can alert a user when new members for those classes are generated by new data. We can also imagine actions (external procedures) being triggered automatically when such class members are found. While this begins to sound like the sort of operation done with a procedural system, in the case of a DL, the detection of interesting situations is handled automatically once the situation is described as a concept.

Working memory for a production system: The above scenario is somewhat reminiscent of a common use of production systems; in situations where the description logic language is expressive enough, a DL could in fact be used entirely to take the place of a production system. In other cases, it may be useful to preserve the power and style of a production system, but a DL might provide some very useful added value. In particular, if the domain of interest has a natural object-oriented, hierarchical structure, as so many do, a true picture of the domain can only be achieved in a pure production system if there are explicit rules capturing the inheritance relationships, part-whole relationships, etc. An alternative would be to use a DL as the working memory. The DL would encode the hierarchical domain theory, and take care of classification and inheritance automatically. The production system could then restrict its attention to complex pattern detection and action—where it belongs—with its rules represented at just the right, natural level (the antecedents could refer to classes at any level of a DL generalization hierarchy), avoiding any *ad hoc* attempts to encode inheritance or classification procedurally.

Using concepts as queries and access to databases: It is possible to think of a concept as a query asking for all of its instances. Imagine we have "raw" data stored in a relational database system. We can then develop an object-oriented model of the world in our DL, and specify a mapping from that model to the schema used in the conventional DBMS. This would then allow us to ask questions of a relational database mediated by an object-oriented domain model. One could implement such a hybrid system either by pre-classifying in the KB all objects from the DB and using classification of a DL query to find answers, or leaving the data in the DB and dynamically translating a DL query into a DB query language like SQL.

9.7 Bibliographic notes

9.8 Exercises

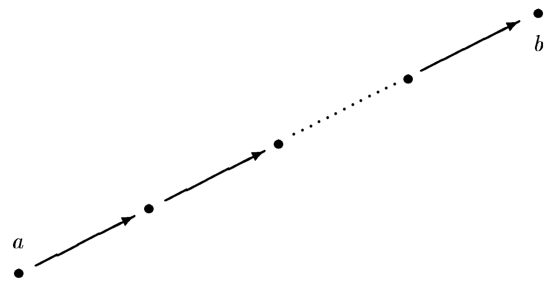
Chapter 10

Inheritance

As we saw in previous chapters on frames and description logics, when we think about the world in an object-centered way, we inevitably end up thinking in terms of *hierarchies*. This reflects the importance of abstraction, classification, and generalization in the enterprise of knowledge representation. Groups of things in the world naturally share properties, and we talk about them most concisely using words for abstractions like "furniture" or "situation comedy" or "seafood." Further, hierarchies allow us to avoid repeating representations—it is sufficient to say that "elephants are mammals" to immediately know a great deal about them. Taxonomies of kinds of objects are so fundamental to our thinking about the world that they are found everywhere, especially when it comes to organizing knowledge in a comprehensible form for human consumption, in encyclopedias, dictionaries, scientific classifications, and so on.

The centrality of taxonomy means that the idea of *property inheritance* that we saw with frames and description logics is also fundamental to knowledge representation. In the kind of classification networks we built using description logics, inheritance was just a way of doing logical reasoning in a graphically-oriented form: if we have a network where the concept PianoConcerto is directly below Concerto, which is directly below MusicalWork, then PianoConcerto inherits properties from MusicalWork because logically all instances of PianoConcerto are instances of Concerto and all instances of Concerto are instances of MusicalWork. Similar considerations apply in the case of frames, although the reasoning there is not strict: if the **IS-A** slot of frame AdultHighSchoolStudent points to HighSchoolStudent and HighSchoolStudent points to Teenager, then AdultHighSchoolStudent may inherit properties from HighSchoolStudent and HighSchoolStudent in turn from Teenager, but we are no longer justified in concluding that an instance of AdultHighSchool-

Figure 10.1: Inheritance reasoning is path reasoning



Student must be an instance of Teenager. In both cases, however, "can a inherit properties from b?" involves asking if b is in the transitive closure of some sort of generalization relation from a. As illustrated in Figure 10.1, this amounts to asking if there is a *path* of connections from a to b.

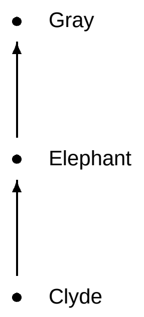
Many interesting considerations arise even when we just focus our attention on where the information comes from in a network representation like this. In order to highlight the richness of path-based reasoning in networks, in this chapter we are going to concentrate just on inheritance and transitivity relations among nodes in a network. While the networks we will use will suppress a great deal of representational detail, it is important to keep in mind that they are merely the backbones of inheritance hierarchies expressing generalization relationships among frames or concepts. Because the nodes in these networks stand for richly structured frames or concepts, inheritance reasoning complements the other forms of reasoning we have covered in previous chapters.

10.1 Inheritance networks

In this chapter, we reduce the frames and descriptions of previous chapters to simple *nodes* that appear in *inheritance networks*, like the one expressed in the graph in Figure 10.2. We will use the following concepts in our discussion:

- *edges* in the network, connecting one node directly to another. In the figure, Clyde · Elephant and Elephant · Gray are the two edges. These represent **IS-A**

Figure 10.2: A simple inheritance network



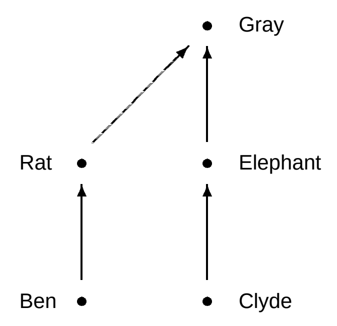
or subsumption relations.

- *paths* included in the network; a path is a sequence of one or more edges. In the figure, the edges mentioned above are also paths, as is Clyde · Elephant · Gray.
- *conclusions* supported by the paths. In this figure, these conclusions are supported: Clyde $\rightarrow$ Elephant; Elephant $\rightarrow$ Gray; Clyde $\rightarrow$ Gray. These conclusions are supported because the edges represent **IS-A** or subsumption relations, and these relations are transitive.

Finally, note that for our discussion here we treat object-like concepts, like Elephant, and properties, like Gray, equivalently as nodes. If we wanted to be more precise, we could use terms like GrayThing (for a Thing whose Color role was filled with the individual gray), but for purposes of this exposition that is not really necessary. Also, we normally do not distinguish which nodes at the bottom of the hierarchy stand for individuals like Clyde, and which stand for kinds like Elephant. We will capitalize the names of both.

Before getting into some of the interesting complications with inheritance networks, we should look at some simple configurations of nodes and basic forms of inheritance.

Figure 10.3: Strict inheritance in a tree



10.1.1 Strict inheritance

The simplest form of inheritance is the kind used in description logics and other systems based on classical logic: *strict* inheritance. In a strict inheritance network, conclusions are produced by the complete transitive closures of all paths in the network. Any traversal procedure for computing the transitive closure will do for determining the supported conclusions.

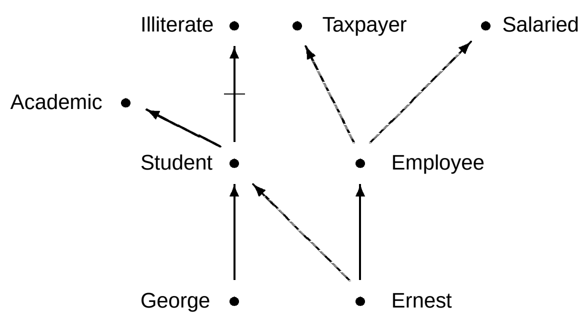
In a tree-structured strict inheritance network, inheritance is very simple. As in Figure 10.3, all nodes reachable from a given node are implied. In this figure, supported conclusions include the fact that Ben is gray, and that Clyde is gray.

In an inheritance network that is a *directed acyclic graph* (DAG), the results are the same as for strict inheritance: all conclusions you can reach by any path are supported. This includes conclusions found by traversing different branches upward from a node in question. Figure 10.4 illustrates a strict DAG. It says that Ernest is both a student and an employee. The network supports the conclusions that Ernest is an academic, as well as a taxpayer, and salaried.

Note that in this figure we introduce a negative edge with a bar through it, between Student and Illiterate, standing roughly for "is-not-a" or "is-not." So edges in these networks have *polarity*—positive or negative. Thus the conclusion that Ernest is not illiterate is supported by the network in the figure.[1]

[1]As we will see more precisely in Section 10.3, when a network contains negative edges, a path is considered to be zero or more *positive* edges followed by a single positive or negative edge.

Figure 10.4: Strict inheritance in a DAG



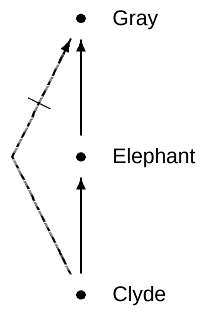
Inheritance in directed acyclic networks is often called "multiple inheritance" when a node has more than one parent node; in such cases, because of the meaning of the edges, the node must inherit from all of its parents.

10.1.2 Defeasible inheritance

In our study of frame systems, we saw numerous illustrations of a non-strict inheritance policy. In these representations, inherited properties do not always hold; they can be *defeated*, or overridden. This is most obviously true in the case of **DEFAULT** facets for slots, such as the default origin of one of my trips. But a closer examination of the logic of frame systems such as those that we covered in Chapter 8 would suggest that in fact virtually *all* properties (and procedures) can be overridden (one exception is the **REQUIRE** facet we discussed briefly). We call the kind of inheritance networks in which properties can be defeated, "*defeasible* inheritance networks."

In a defeasible inheritance scheme, conclusions are determined by searching upward from a *focus node*—the one about which we are trying to draw a conclusion—and selecting the first version of the property being considered. An example will make this clear. In Figure 10.5, there is an edge from Clyde to Elephant, and one from there to Gray. There is also, however, a negative edge from Clyde directly to Gray. This network is intended to capture the knowledge that while elephants in general are gray, Clyde is not. Intuitively, if we were trying to find what conclu-

Figure 10.5: Defeasible inheritance



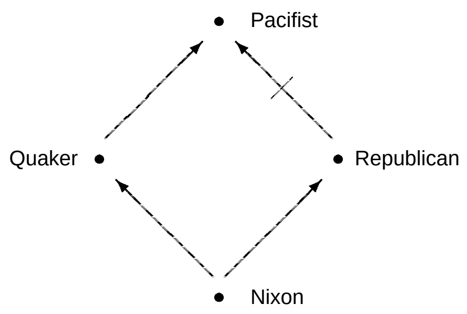
sion this network supported about Clyde's color, we would first find the negative conclusion about Gray, since that is directly asserted of Clyde.

In general, what will complicate defeasible reasoning, and what will occupy us for much of this chapter, is the fact that different paths in a network can support conflicting conclusions, and a reasoning procedure needs to decide which conclusion should prevail, if any. In the above example, there is an argument for Clyde being gray: he is an elephant and elephants are gray; however, there is a "better" argument for concluding that he is not gray, since this has been asserted of him specifically.

Of course, we expect that in some cases we will not be able to say which conclusion is better or worse. In Figure 10.6 there is nothing obvious that tells us how to choose between the positive or negative conclusions about Nixon's pacifism. The network tells us that by virtue of his being a Quaker he is a pacifist; it also tells us that by virtue of his being a Republican, he is not. This type of network is said to be *ambiguous*.

When exploring different accounts for reasoning under this kind of circumstance, we typically see two types of approaches: *credulous* accounts allow us to choose arbitrarily between conclusions that appear equally well supported; *skeptical* accounts are more conservative, often accepting only conclusions that are not contradicted by other paths. In the above case, a credulous account would in essence flip a coin and choose one of Nixon $\rightarrow$ Pacifist or Nixon $\rightarrow$ $\neg$Pacifist, since either conclusion is as good as the other. A skeptical account would draw no conclusion

Figure 10.6: Is Nixon a pacifist, or not?



about Nixon's pacifism.

10.2 Strategies for defeasible inheritance

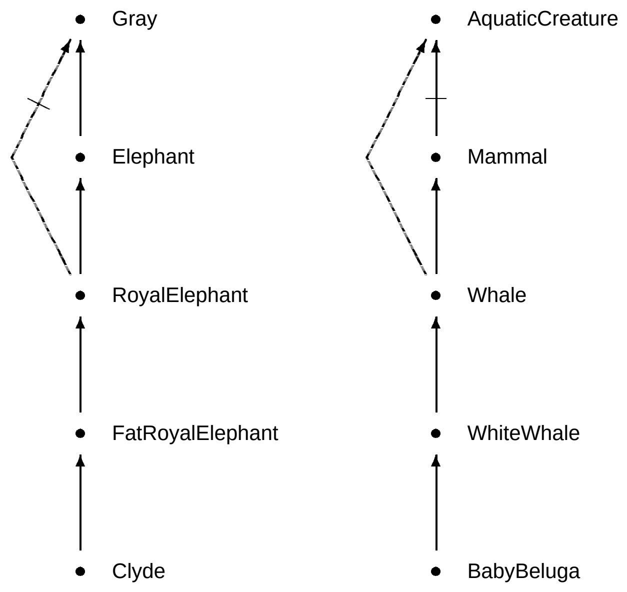
For DAGs with defeasible inheritance, we need a method for deciding which conclusion to choose (if any) when there are contradictory conclusions supported by different paths through the network. In this section, we examine two possible ways of doing this informally, before moving to a precise characterization of inheritance reasoning in the next section.

10.2.1 The shortest path heuristic

Figure 10.7 shows two examples of defeasible inheritance networks that produce intuitively plausible conclusions. In the one on the left, we see that while Royal Elephants are elephants, and elephants are (typically) gray, Royal Elephants are not. Since Clyde is a Royal Elephant, it would be reasonable to assume he is not gray.

To decide this in an automated way, the *shortest path heuristic* says that we should prefer conclusions resulting from shorter paths in the network. Since there are fewer edges in the path from Clyde to Gray that includes the negative edge than in the path that includes the positive edge, the negative conclusion prevails.

Figure 10.7: Shortest path heuristic



In the network on the right, we see the opposite polarity conclusion being supported. Whales are mammals, but mammals are typically not aquatic creatures. Whales are exceptional in that respect, and are directly asserted to be aquatic creatures. We infer using the shortest path heuristic that BabyBeluga is an AquaticCreature.

The intuition behind the shortest path heuristic is that it makes sense to inherit from the most specific subsuming class. If two superclasses up the chain disagree on a property (e.g., Gray vs. ¬Gray), we take the value from the more specific one, since that is likely to be more directly relevant.[2]

[2]A similar consideration arises in probabilistic reasoning in Chapter 11 regarding choosing what is called a "reference class": our degree of belief in an individual having a certain property depends on the most specific class he belongs to for which we have statistics.

Notice then, that in defeasible inheritance networks, not all paths count in generating conclusions. It make sense to think of the paths in the network as *arguments* in support of conclusions. Some arguments are *preempted* by others. Those that are not we might call "admissible." The inheritance problem, then, is "What are the admissible conclusions supported by the network?"

10.2.2 Problems with shortest path

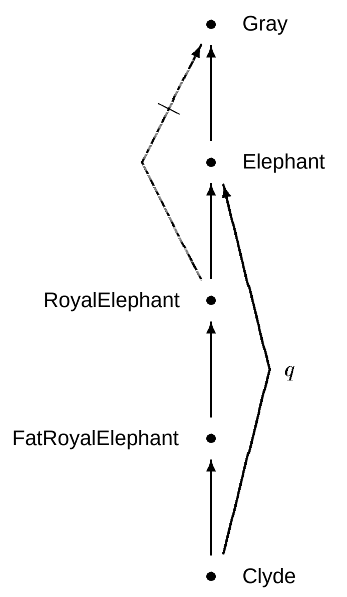
While intuitively plausible, and capable of producing correct conclusions in many cases, the shortest path heuristic has serious flaws. Unfortunately, it can produce incorrect answers in the presence of redundant edges—those that are already implied by the basic network. Look at the network in Figure 10.8. The edge labeled q is simply redundant, in that it is clear from the rest of the network that Clyde is unambiguously an elephant. But by creating an edge directly from Clyde to Elephant we have inadvertently changed the polarity of the conclusion about Clyde's color! The path from Clyde to Gray that goes through edge q is now shorter (length=2) than the one with the negative edge from RoyalElephant to Gray (length=3). So the inclusion of an edge that is already implicitly part of the network undermines the shortest path heuristic.

Another problem with the shortest path heuristic is the fact that the length of a path through the network does not necessarily reflect anything salient about the domain. Depending on the problem or application, some paths may describe object hierarchies in excruciating detail, while others may be very sketchy. There is no reason that just because an inheritance chain makes many fine-grained distinctions there should be a bias against it in drawing conclusions. Figure 10.9 illustrates in a somewhat extreme way how this causes problems. The left-hand path has a very large number of nodes in it, and ends with a positive edge. The right-hand path has just one more edge, and ends with a negative edge. So for this network, the shortest path heuristic supports the positive conclusion. But if we were to add another two edges—anywhere in the path—to the left-hand side, the conclusion would be reversed. This seems rather silly; the network should be considered ambiguous in the same manner as the one in Figure 10.6.

10.2.3 Inferential distance

Shortest path is what is considered to be a *preemption strategy*, which allows us to make admissibility choices among competing paths. It tries to provide a *specificity criterion*, matching our intuition that more specific information about an item is

Figure 10.8: Shortest path in the face of redundant links

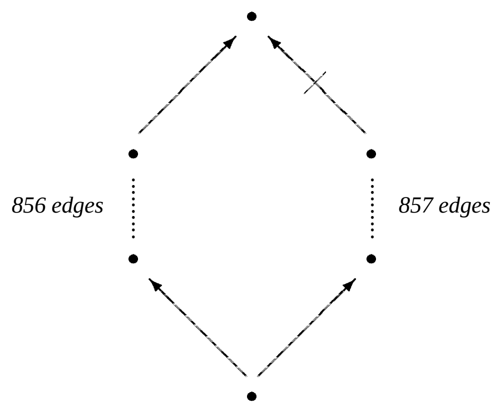


more relevant than information more generally true about a broader class of items of which it is a member.

As we have seen, shortest path has its problems. Fortunately, it is not the only possible specificity criterion. A more plausible strategy would be to use *inferential distance*, which rather than being linear distance-based, is topologically based.

Consider Figure 10.8 once again. Starting at the node for Clyde, we would like to say that RoyalElephant is more specific than Elephant despite the redundant edge q because there is a path to Elephant that passes through RoyalElephant. Because it is more specific, we then prefer the negative edge from RoyalElephant to Gray over the positive one from Elephant to Gray. More generally, a node a is considered nearer to node b than to node c according to inferential distance iff there is a path

Figure 10.9: Very long paths



from a to c through b, regardless of the actual length of any paths from a to b and to c.

This criterion handles the earlier simple cases of inheritance from Figure 10.7. Furthermore, in the case of the ambiguous network of Figure 10.9, inferential distance prefers neither conclusion, as desired.

Unfortunately, inferential distance has its own problems. What should happen, for example, when the path from a through b to c is itself contradicted by another path? Rather than attempt to patch the definition to deal with such problematic cases, we will consider a different formalization of inheritance that incorporates a version of inferential distance as well as other reasonable accounts of defeasible inheritance networks.

10.3 A formal account of inheritance networks

The discussion above was intended to convey some of the intent and issues behind defeasible inheritance networks, but was somewhat informal. The ideas in these networks can be captured and studied in a much more formal way. We here briefly present one of the clearer formal accounts of inheritance networks (there are many that are impenetrable), owing to Lynn Stein.

An *inheritance hierarchy* $\Gamma = \langle V, E\rangle$ is a directed, acyclic graph with positive and negative edges, intended to denote "(normally) is-a" and "(normally) is-not-a," respectively (V are the nodes, or vertices, in the graph; E are the edges). Positive edges will be written as $(a \cdot x)$ and negative edges will be written as $(a \cdot \neg x)$.

A *positive path* is a sequence of one or more positive edges $a \cdot \ldots \cdot x$. A *negative path* is a sequence of zero or more positive edges followed by a single negative edge: $a \cdot \ldots \cdot v \cdot \neg x$. A *path* is either a positive or negative path.

Note that there are no paths with more than one negative edge, although a negative path could have no positive edges (*i.e.*, be just a negative edge).

A path (or *argument*) supports a *conclusion* in the following ways:

- $a \cdot \ldots \cdot x$ supports the conclusion $a \rightarrow x$ (a is an x);
- $a \cdot \ldots \cdot v \cdot \neg x$ supports the conclusion $a \not\rightarrow x$ (a is not an x).

A single conclusion can be supported by many arguments. However, not all arguments are equally believable. We now look at what makes an argument prevail, given other arguments in the network. This stems from a formal definition of admissibility:

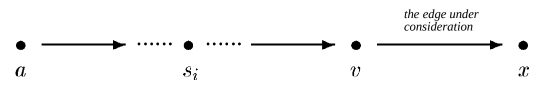
Γ *supports a path* $a \cdot s_1 \cdot \ldots \cdot s_n \cdot (\neg)x$ if the corresponding set of edges are in E, and it is *admissible* according to the definition below. The hierarchy *supports a conclusion* $a \rightarrow x$ (or $a \not\rightarrow x$) if it supports some corresponding path between a and x.

A path is *admissible* if every edge in it is admissible.

An edge $v \cdot (\neg)x$ is *admissible in* Γ *w.r.t.* a if there is a positive path $a \cdot s_1 \cdot \ldots s_n \cdot v$ ($n \geq 0$) in E and

1. each edge in $a \cdot s_1 \cdot \ldots s_n \cdot v$ is admissible in Γ w.r.t. a (recursively);
2. no edge in $a \cdot s_1 \cdot \ldots s_n \cdot v$ is redundant in Γ w.r.t. a (see below);
3. no intermediate node $a, s_1, \ldots, s_n$ is a preemptor of $v \cdot (\neg)x$ w.r.t. a (see below).

Figure 10.10: Basic path situation for formalization



So, an edge is admissible with respect to a if there is a nonredundant, admissible path leading to it from a that contains no preempting intermediaries. This situation is sketched in Figure 10.10.

The definitions of preemption along a path and of redundancy will complete the basic formalization:

A node y along path $a \cdot \ldots \cdot y \cdot \ldots \cdot v$ is a *preemptor of* $v \cdot x$ ($v \cdot \neg x$) *w.r.t.* a if $y \cdot \neg x \in E$ ($y \cdot x \in E$). For example, in Figure 10.11, the node Whale preempts the negative edge from Mammal to AquaticCreature with respect to both Whale and BlueWhale.

A positive edge $b \cdot w$ is *redundant in* Γ *w.r.t. node* a if there is some positive path $b \cdot t_1 \cdot \ldots \cdot t_m \cdot w \in E$ ($m \geq 1$) for which

1. each edge in $b \cdot t_1 \cdot \ldots \cdot t_m$ is admissible in Γ w.r.t. a (i.e., none of the edges are themselves preempted);
2. there are no c and i such that $c \cdot \neg t_i$ is admissible in Γ w.r.t. a;
3. there is no c such that $c \cdot \neg w$ is admissible in Γ w.r.t. a.

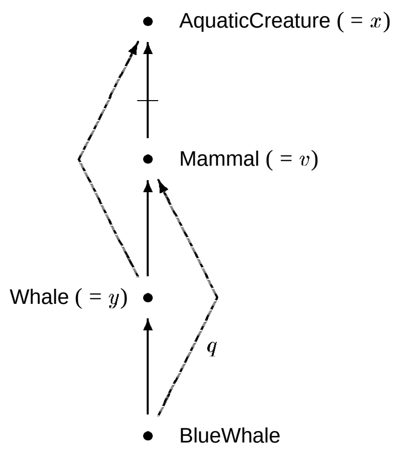
By this definition, the edge labeled q in Figure 10.11 is redundant.

The definition of redundancy for a negative edge $b \cdot \neg w$ is analogous to the above.

10.3.1 Extensions

Now that we have covered the basics of admissibility and preemption, we can finally look at how to calculate what conclusions should believed given an inheritance network. As we noted in Section 10.1.2, we do not expect an ambiguous network to specify a unique set of conclusions. We use the term *extension* to mean a possible set of beliefs supported by the network. Ambiguous networks will have multiple extensions. More formally, we have the following:

Figure 10.11: A preempting node

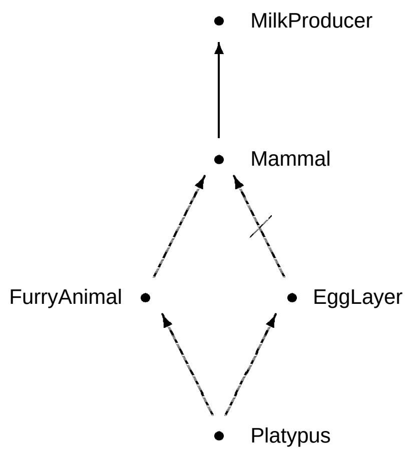


> Γ is *a-connected* iff for every node x in Γ, there is a path from a to x, and for every edge $v \cdot (\neg)x$ in Γ, there is a positive path from a to v. In other words, every node and edge is reachable from a.
>
> Γ is (potentially) *ambiguous w.r.t. node a at x* if there is some node $x \in V$ such that both $a \cdot s_1 \cdot \ldots \cdot s_n \cdot x$ and $a \cdot t_1 \cdot \ldots \cdot t_m \cdot \neg x$ are paths.
>
> A *credulous extension* of an inheritance hierarchy Γ with respect to a node a is a maximal unambiguous a-connected subhierarchy of Γ with respect to a.

So if X is a credulous extension of Γ, then adding an edge of Γ to X makes X either ambiguous or not a-connected.

Figure 10.12 illustrates an ambiguous network, and Figure 10.13 shows its two credulous extensions. Note that adding the edge from Mammal to MilkProducer in the extension on the left would cause that extension to no longer be a-connected (where a is Platypus), because there is no positive path from Platypus to Mammal. Adding the edge from FurryAnimal to Mammal in the extension on the left, or the

Figure 10.12: An ambiguous network

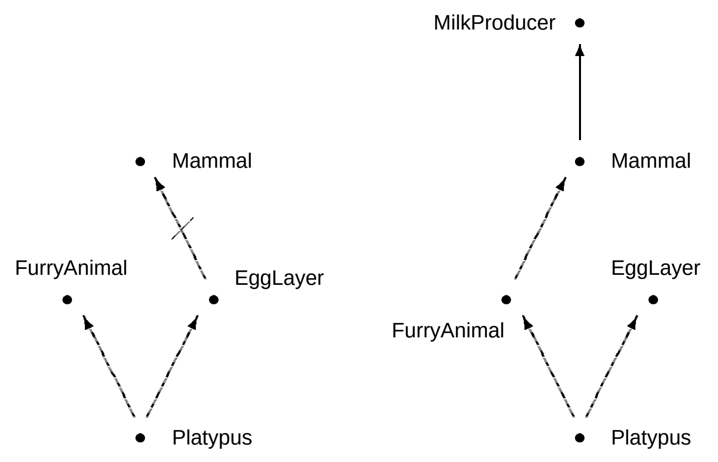


edge from EggLayer to Mammal in the extension on the right, would make the extensions ambiguous. Thus, both extensions in the figure are credulous extensions.

Credulous extensions do not incorporate any notion of admissibility or preemption. For example, the network of Figure 10.5 has two credulous extensions with respect to node Clyde. However, given our earlier discussion and our intuition about reasoning about the natural world, we would like our formalism to rule out one of these extensions. This leads us to a definition of *preferred* extensions:

> Let X and Y be credulous extensions of Γ w.r.t. a node a. X is *preferred* to Y iff there are nodes v and x such that
>
> - X and Y agree on all edges whose endpoints precede x topologically,
> - there is an edge $v \cdot x$ (or $v \cdot \neg x$) that is *inadmissible* in Γ, and
> - this edge is in Y but not in X.
>
> A credulous extension is a *preferred extension* if there is no other credulous extension that is preferred to it.

Figure 10.13: Two credulous extensions



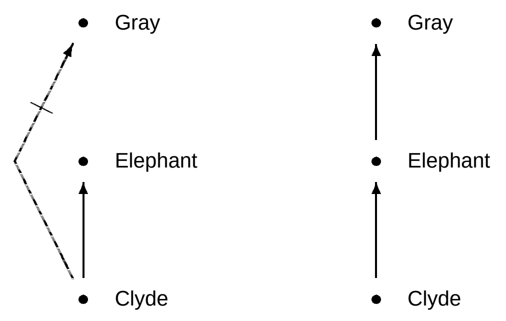
The key part of this definition is that it appeals to the notion of admissibility defined above. So, for example, for the Γ shown in Figure 10.5, the extension on the left in Figure 10.14 is a preferred extension, while the one on the right is not. If we use the assignment a=Clyde, v=Elephant, and x=Gray, we can see that the two extensions agree up to Elephant, but the edge Elephant · Gray is not admissible because it has a preemptor, Clyde, and that edge is in extension on the right but not on the left.

10.3.2 Some subtleties of inheritance reasoning

While we have detailed some reasonable formal definitions that allow us to distinguish between different types of extensions, an agent still needs to make a choice based on such a representation of what actually to believe. The extensions offer sets of consistent conclusions, but one's attitude towards such extensions can vary. Different forms of reasoning have been proposed based on the type of formalization we have presented here:

- *credulous reasoning*: choose a preferred extension, perhaps arbitrarily, and believe all of the conclusions supported by it.

Figure 10.14: A preferred credulous extension



- *skeptical reasoning*: believe the conclusions supported by any path that is present in all preferred extensions.

- *ideally skeptical reasoning*: believe the conclusions that are supported by all preferred extensions. This is subtly different from skeptical reasoning as above, in that these conclusions may be supported by different paths in each extension. One significant consequence of this is that ideally skeptical reasoning cannot be computed in a path-based way.

One final point to note is that our emphasis in this chapter has been on "upwards" reasoning—in each case, we start at a node and see what can be inherited from its ancestor nodes further "up" the tree. There are actually many variations on this definition, and none has emerged as the agreed upon, or "correct" one. One alternative, for example, looks from the top and sees what propagates downward through the network.

In Chapter 12, we will reconsider in more general logical terms the kind of defeasible reasoning seen here in inheritance networks. We will study some very expressive representation languages for this that go well beyond what can be represented in a network. While these languages have a clear logical foundation, we will see that it is quite difficult to get them to emulate in a convincing way the subtle path-based account of reasoning we have investigated here.

10.4 Bibliographic notes

10.5 Exercises

Chapter 11

Numerical Uncertainty

11.1 Bibliographic notes

11.2 Exercises

Chapter 12

Defaults

In Chapter 8 on Frames, the kind of reasoning exemplified by the inheritance of properties was actually a simple form of *default reasoning*, where a slot was assumed to have a certain value unless a different one was provided explicitly. In Chapter 10 on inheritance, we also considered a form of default reasoning in hierarchies. We might know, for example, that elephants are gray, but understand that there could be special kinds of elephants that are not. In this chapter, we look at this form of default reasoning in detail and in logical terms, without tying our analysis either to procedural considerations or to the topology of a network as we did before.

12.1 Introduction

Despite the fact that FOL is an extremely expressive representation language, it is nonetheless restricted in the patterns of reasoning it admits. To see this, imagine that we have a KB in FOL that contains facts about animals of various sorts, and that we would like to find out whether a particular individual, Fido, is a carnivore. Assuming that the KB contains the sentence Dog(fido), there are exactly two ways to get to the conclusion Carnivore(fido):

1. the KB contains other facts that use the constant fido explicitly;
2. the KB entails a universal of the form $\forall x.\mathsf{Dog}(x) \supset \mathsf{Carnivore}(x)$.

It is not too hard to see that if neither of these two conditions are satisfied, the desired conclusion simply cannot be derived: there is a logical interpretation that satisfies the KB but not Carnivore(fido).[1] So it is clear that if we want to deduce

[1]The construction is as follows: take any model $\Im = \langle \mathcal{D}, \mathcal{I} \rangle$ of the KB that does not satisfy the above universal. So there is a dog d in $\mathcal{D}$ that is not a carnivore. Let $\Im' = \langle \mathcal{D}, \mathcal{I}' \rangle$ be just like $\Im$

something about a particular dog that we know nothing else about, the only option available to us in FOL is to use what we know about *each and every* dog. In general, to reason from $P(a)$ to $Q(a)$ in FOL where we know nothing else about a itself, we need to use what is known to hold for all instances of P.

12.1.1 Generics and Universals

So what is the problem? It is this: all along, we have been imagining that we will build a KB that contains facts about a wide variety of topics, somewhat like an encyclopedia. There would be "entries" on turtles, violins, wildflowers, and ferris wheels as in normal encyclopedias, as well as entries on more mundane subjects, like grocery stores, birthday parties, rubber balls, and haircuts. Clearly, what we would like to say about these topics goes beyond facts about particular cases of turtles or violins. The troublesome fact of the matter is that although we may have a great deal to write down about violins, say, almost none of it applies to *all* violins. The problem is how to express what we know about the topics *in general* using FOL, and in particular, using universal quantification.

We might want to state, for example, that

Violins have four strings

to distinguish them from guitars, which have six. But we most assuredly do *not* want to state that

All violins have four strings

since, obviously, this would rule out a violin with a string added or removed. One possible solution is to to attempt to enumerate the conditions under which violins would not have four strings:

All violins that are not P_1 or P_2 or ... or P_n have four strings

where the P_i state the various exceptional cases. The problem is to characterize these cases. We would need to cover at least the following: natural manufacturing (or genetic) varieties, like electric violins; cases in exceptional circumstances, like violins that have been modified or damaged; borderline cases, like miniature toy violins; imagined cases, like multi-player violins (whatever they might be); and so on. Because of the range of possibilities, we are almost reduced to saying

All violins have four strings except those that do not

except that $\mathcal{I}'$[fido] = d. Since KB contains no facts other than Dog(fido) that mention fido, $\Im'$ still satisfies KB, but $\Im'$ satisfies ¬Carnivore(fido).

—a true but quite pointless universal.

This is obviously not just a problem with the topic of violins. When we say that lemons are yellow and tart, that polar bears are white and live in Arctic regions, that birds have wings and fly, that children sing "Happy Birthday" at birthday parties, that banks are closed on Sundays, and on and on, we do not mean to say that such sentences hold of each and every instance of the corresponding class. And yet the facts are true; it would be wrong to say that at birthday parties, children sing "Oh! Susanna," for example.

So we need to distinguish between *universals*, properties that do hold for all instances, easily expressible in FOL, and *generics*, properties that hold "in general." Much of our common-sense knowledge of the world appears to be concerned with generics, so it is quite important to consider formalisms that go beyond FOL in allowing us to handle general, but not truly universal, knowledge.

12.1.2 Default reasoning

Assuming we know that dogs are, generally speaking, carnivores, and that Fido is a dog, under what circumstances is it appropriate to infer that Fido is a carnivore? The answer we will consider in very general terms is this:

> Given that a P is generally a Q, and given $P(a)$, it is reasonable to conclude $Q(a)$ unless there is an explicit reason not to.

This answer is unfortunately somewhat vague: exactly what constitutes a good reason not to conclude something? Different ways of making this precise will be the subject of the rest of the chapter.[2]

One thing to notice, however, is that if absolutely nothing is known about the individual a except that it is an instance of P, then we should be able to conclude that it is an instance of Q, since there can be nothing that would urge us not to. When we happen to know that a polar bear has been rolling in the mud, or swimming in an algae-ridden pool, or playing with paint cans, then we may not be willing to conclude anything about its color; but if *all* we know is that the individual is a polar bear, it seems perfectly reasonable to conclude that it is white.

Note, however, that just because we don't know that the bear has been blackened by soot, for example, doesn't mean that it hasn't been. The conclusion does not have the guarantee of logical soundness; everything else we believe about polar bears could be true without this particular bear being white. It is only a reasonable

[2]In Chapter 11 we consider ways of dealing with this issue numerically. Here our approach is qualitative.

default. That is to say, if we are pressed for some reason to come to some decision about its color, white is a reasonable choice. In general, this form of reasoning, which involves applying some general though not universal fact to a particular individual is called *default reasoning*.

We do not want to suggest, however, that the only source of default reasoning has to do with general properties of kinds like violins, polar bears, or birthday parties. There are a wide variety of reasons for wanting to conclude $Q(a)$ given $P(a)$ even in the absence of true universal quantification. Here are some examples:

General Statements

- *normal:* Under typical circumstances, P's are Q's.
  (People work close to where they live. Children enjoy singing.)
- *prototypical:* The prototypical P is a Q.
  (Apples are red. Owls hunt at night.)
- *statistical:* Most P's are Q's.
  (The people in the waiting room are growing impatient.)

Lack of information to the contrary

- *familiarity:* If a P was not a Q, you would know it.
  (No nation has a political leader more than 7 feet tall.)
- *group confidence:* All the known P's are known (or assumed) to be Q's.
  (Natural languages are easy for children to learn.)

Conventional Uses

- *conversational:* A P is a Q, unless I tell you otherwise.
  (Being told "The closest gas station is two blocks east"—the assumed default: the gas station is open).
- *representational:* A P is a Q, unless otherwise indicated.
  (The speed limit in a city. An open door to an office, meaning that the occupant can be disturbed.)

Persistence

- *inertia:* A P is a Q unless something changes it.
  (Marital status. The position of objects (within limits).)
- *time:* A P is a Q if it used to be a Q.
  (The color of objects. Their sizes.)

These categories are not intended to be exhaustive. But they do suggest the very wide variety of sources of default information. In all cases, our concern in this chapter will be the same: how to characterize precisely when, in the absence of universals, it is appropriate to draw a default conclusion. In so doing, we will only use the simplest of examples, like the default that birds fly, which in FOL would have to be approximated by $\forall x(\mathsf{Bird}(x) \supset \mathsf{Flies}(x))$. But the techniques considered here apply to all the various forms of defaults above, which, as we have argued, cover much of what we know.

12.1.3 Non-monotonicity

In the rest of this chapter, we will consider four approaches to default reasoning: closed-world reasoning, circumscription, default logic, and autoepistemic logic. In all cases, we start with a KB from which we wish to derive a set of implicit beliefs. In the simple case with no default reasoning, implicit beliefs are just the entailments of the KB; but with defaults, we go beyond these by making various assumptions.

Ordinary deductive reasoning is *monotonic*, which is to say that new facts can only produce additional beliefs. In other words, if $\text{KB}_1 \models \alpha$, then $\text{KB}_2 \models \alpha$, for any KB_2 such that $\text{KB}_1 \subseteq \text{KB}_2$. However, default reasoning is *non-monotonic*: new facts will sometimes invalidate previous beliefs. For example, if we are only told that Tweety is bird, we may believe that Tweety flies. However, if we are now told that Tweety is an emu, we may no longer believe that she flies. This is because the belief that Tweety flies was a default based on an *absence* of information to the contrary. When we find out that Tweety is an exceptional bird, we reconsider.

For this reason, default reasoning of the kind we will discuss in this chapter is often called *non-monotonic reasoning*, where the emphasis is not so much on how assumptions are made or where they come from, but on inference relations that are similar to entailment, but which are non-monotonic.

12.2 Closed-world Reasoning

The simplest formalization of default reasoning we will consider was also the first to be developed, and is based on the following observation:

> Imagine representing facts about the world in FOL within a fixed, finite vocabulary of predicates, function and constant symbols. Of the large (but finite) number of atomic sentences that can be formed, only a very small fraction are expected to be *true*. A reasonable representational

convention, then, is to explicitly represent the true atomic sentences, and to assume that any unmentioned atomic sentence is false.

Consider, for example, information sources like an airline flight guide. The kind of information we find in a such a guide might be roughly represented in FOL by sentences like

DirectConnect(cleveland,toronto),
DirectConnect(toronto,northBay),
DirectConnect(cleveland,phoenix),

telling us which cities have flights between them. What we do not expect to find in such a guide are statements about which cities do *not* have flights between them:

¬DirectConnect(northBay,phoenix).

The convention is that if an airline does not list a flight between two cities, then there is none. Similar conventions are used, of course, in encyclopedias, dictionaries, maps, and many other information sources. It is also the assumption used in computerized *databases*, modeled exactly on such information sources.

12.2.1 The closed-world assumption

In general terms, the assumption here, called the *closed-world assumption* or CWA, is the following:

Unless an atomic sentence is known to be true,
it can be assumed to be false.

Note that expressed this way, the CWA can be seen to involve a form of default reasoning. A sentence assumed to be false could later be determined in fact to be true.

Perhaps the easiest way to formalize the reasoning inherent in the CWA is to consider a new form of entailment, $\models_C$, where we say that KB $\models_C \alpha$ iff KB$^+$ $\models \alpha$, where

$$\text{KB}^+ = \text{KB} \cup \{\neg p \mid p \text{ is atomic and KB} \not\models p\}.$$

So $\models_C$ is just like ordinary entailment, except with respect to an augmented KB, namely one that includes all negative atomic facts not explicitly ruled out by the KB.[3] In the airline guide example above, KB$^+$ would include all the appropriate ¬DirectConnect(c_1, c_2) sentences.

[3]This definition applies to the propositional subset of FOL. We will deal with quantifiers below.

12.2.2 Consistency and completeness of knowledge

It is useful to introduce two terms at this point: we say that a KB exhibits *consistent* knowledge if and only if there is no sentence α such that both α and $\neg\alpha$ are known. This is the same as requiring the KB to be satisfiable. We also say that a KB exhibits *complete* knowledge if and only if for every sentence α (within its vocabulary), either α or $\neg\alpha$ is known.

In general, of course, knowledge can be incomplete. For example, suppose KB consists of a single sentence, $(p \vee q)$. Then, KB does not entail either p or $\neg p$, and so exhibits incomplete knowledge. If we consider the CWA as formalized as above, however, for any sentence α, it holds that either KB $\models_C \alpha$ or KB $\models_C \neg\alpha$. (The argument is by induction on the length of α.) So with the CWA, we have completely filled out the entailment relation for the KB. Every sentence is *decided* by KB$^+$, that is, either it or its negation is entailed by KB$^+$.

It is not hard to see that if a KB is complete in this sense (the way KB$^+$ is), it also has the property that if it tells us that one of two sentences is true, then it must also tell us which. In other words, if KB exhibits complete knowledge and KB $\models (\alpha \vee \beta)$, then KB $\models \alpha$ or KB $\models \beta$. Again, note that this is not the case in general, for example, for the KB comprising only $(p \vee q)$ as described above.

The idea behind the CWA then, is to act *as if* the KB represented complete knowledge. Whenever KB $\not\models p$, then either KB $\models \neg p$ directly, or the assumption is that $\neg p$ is what was intended, and it is conceptually added to the KB.

12.2.3 Query evaluation

The fact that every sentence is decided by the CWA allows queries to be handled very directly. The question as to whether KB $\models_C \alpha$ ends up reducing to a collection of questions about the literals in α. We begin with the following general properties of entailment:

1. KB $\models (\alpha \wedge \beta)$ iff KB $\models \alpha$ and KB $\models \beta$.
2. KB $\models \neg\neg\alpha$ iff KB $\models \alpha$.
3. KB $\models \neg(\alpha \vee \beta)$ iff KB $\models \neg\alpha$ and KB $\models \neg\beta$.

Next, as discussed above, because KB$^+$ is complete, we also have the following properties:

4. KB $\models_C (\alpha \vee \beta)$ iff KB $\models_C \alpha$ or KB $\models_C \beta$.
5. KB $\models_C \neg(\alpha \wedge \beta)$ iff KB $\models_C \neg\alpha$ or KB $\models_C \neg\beta$.

Putting all of these together, we can recursively reduce any question about whether KB $\models_C \alpha$ to a set of questions about the literals in α. For example, it is the case that

$$\text{KB} \models_C ((p \wedge q) \vee \neg(r \wedge \neg s)) \quad \text{iff}$$
$$\text{either KB} \models_C p \text{ and KB} \models_C q, \text{ or KB} \models_C \neg r, \text{ or KB} \models_C s.$$

If we further assume that KB$^+$ is consistent (which we discuss below), we get:

6. If KB$^+$ is consistent, KB $\models_C \neg\alpha$ iff KB $\not\models_C \alpha$.

With this extra condition, we can reduce a query to a set of questions about the *atoms* in α. For example, assuming consistency, the sentence $((p \wedge q) \vee \neg(r \wedge \neg s))$ will be entailed under the CWA if and only if either p and q are entailed or r is not entailed or s is entailed. What this suggests, is that for a KB that is consistent and complete, *entailment conditions are just like truth conditions*: a conjunction is entailed if and only if both conjuncts are; a disjunction is entailed if and only if either disjunct is; and a negation is entailed if and only if the negated sentence is not entailed. As long as we have a way of handling atomic queries, all other queries can be handled recursively.[4]

12.2.4 Consistency and a generalized assumption

Just because a KB is consistent does not mean that KB$^+$ will also be consistent. Consider, for example, the consistent KB composed of the single sentence $(p \vee q)$ mentioned above. Since KB $\not\models p$, it is the case that $\neg p \in$ KB$^+$. Similarly, $\neg q \in$ KB$^+$. So KB$^+$ contains $\{(p \vee q), \neg p, \neg q\}$, and thus is inconsistent. In this case, KB $\models_C \alpha$, for *every* sentence α.

On the other hand, it is clear that if a KB consists of just atomic sentences (like the DirectConnect KB from above) and is itself consistent, then KB$^+$ will be consistent. The same is true if the KB contains conjunctions of atomic sentences (or of other conjunctions). It is also true if the KB contains disjunctions of negative literals. But it is not clear what a reasonable closure assumption should be for disjunctions like $(p \vee q)$.

One possibility is to apply the CWA only to atoms that are completely "uncontroversial." For example, in the above case, while we might not apply the CWA to either p or q, since they are both controversial (because we know that one of them is true), we might be willing to apply it to any other atom. This suggests a generalized version of the CWA, which we call the *generalized closed-world assumption*, or GCWA, where KB $\models_{GC} \alpha$ if and only if KB$^\star \models \alpha$, where KB* is defined as follows:

[4]We will explore the implications of this for reasoning procedures in Chapter 16.

$$\text{KB}^\star = \text{KB} \cup \{\neg p \mid \text{ for all collections of atoms } q_1, \ldots, q_n, \\ \text{if KB} \models (p \vee q_1 \vee \ldots \vee q_n), \text{ then KB} \models (q_1 \vee \ldots \vee q_n)\}.$$

So an atom p can be assumed to be false only if it is the case that whenever a disjunction of atoms including that atom is entailed by the KB, then the smaller disjunction without the atom is also entailed. In other words, we will not assume that p is false if there exists an entailed disjunction of atoms including p that cannot be reduced to a smaller entailed disjunction.

For example, suppose that KB is $(p \vee q)$, and consider the atom p. Here we have that KB $\models (p \vee p \vee q)$, and this indeed reduces to KB $\models (p \vee q)$; however, we also have that KB $\models (p \vee q)$, even though KB $\not\models q$. So $\neg p \notin$ KB*. Similarly, $\neg q \notin$ KB*. However, consider an atom r. Here it is the case that $\neg r \in$ KB*, since although KB $\models (r \vee p \vee q)$, we also have the reduced disjunction KB $\models (p \vee q)$.[5]

Note that if we restrict the definition of KB* to the case where $n = 0$, we get

$$\text{KB}^\star = \text{KB} \cup \{\neg p \mid \text{ if KB} \models p, \text{ then KB} \models \Box\}$$

or equivalently,

$$\text{KB}^\star = \text{KB} \cup \{\neg p \mid \text{ if KB} \models p, \text{ then KB is inconsistent}\}.$$

It follows then that for a consistent KB, the GCWA implies the CWA, i.e., KB* would only include $\neg p$ in the case where KB $\not\models p$, which means that KB* would be the same as KB$^+$. But more importantly, it is the case that if KB is consistent, then so must be KB*. The proof is as follows: suppose KB* is inconsistent, and let KB $\cup \{\neg p_1, \ldots, \neg p_n\}$ be an inconsistent subset with a minimal set of $\neg p_i$ literals. It follows from this inconsistency that KB $\models (p_1 \vee \ldots \vee p_n)$. But then, since $\neg p_1 \in$ KB*, KB $\models (p_2 \vee \ldots \vee p_n)$. This means that KB $\cup \{\neg p_2, \ldots, \neg p_n\}$ is also inconsistent, contradicting the minimality assumption. So the GCWA is an extension to the CWA that is always consistent, and implies the CWA when the KB itself is consistent.

12.2.5 Quantifiers and domain closure

So far we have only considered the properties of the CWA in terms of sentences without quantifiers. Unfortunately, its most desirable properties do not immedi-

[5]The intuition behind this is as follows: say that we know that there is a flight from Cleveland either to Dallas or to Houston (but not which one). As a result, we also know that there is a flight from Cleveland to one of Dallas, Houston, or Austin. But since we know that there is definitely a flight to one of the first two, it makes sense, under normal closed-world reasoning, to assume that there is no flight to Austin.

ately generalize to sentences with quantifiers. To see why, consider a simple representation language containing a single predicate DirectConnect as before and constants $c_1, ..., c_n$. If we start with a KB containing only atomic sentences of the form DirectConnect(c_i, c_j), the CWA will add to this a collection of literals of the form ¬DirectConnect(c_i, c_j). In the resulting KB$^+$, for any pair of constants c_i and c_j, either DirectConnect(c_i, c_j) is in KB$^+$ or ¬DirectConnect(c_i, c_j) is in KB$^+$.

Let us suppose that there is a certain constant smallTown that does not appear in the imagined guide, so that for every c_j, ¬DirectConnect(smallTown, c_j) is in KB$^+$. Now consider the query, $\neg\exists x$DirectConnect(smallTown, x). Ideally, by closed-world reasoning, this sentence should be entailed: there is no city directly connected to smallTown. However, even under the CWA, neither this sentence nor its negation is entailed: the CWA precludes smallTown being connected to any of the *named* cities, $c_1, \ldots, c_n$, but it does not preclude smallTown being connected to some other *unnamed* cities. That is, there is a model of KB$^+$ where the domain includes a city not named by any c_i such that it and the denotation of smallTown are in the extension of DirectConnect. So the problem is that the CWA has not gone far enough: not only do we want to assume that smallTown is not connected to the c_i, we want to assume that there are no other possible cities to connect to.

Perhaps the easiest way to achieve this effect is to assume that the named constants are the only individuals of interest, in other words, that every individual is named by one of the c_i. This leads to a stronger form of closed-world reasoning, which is the *closed world assumption with domain closure*, and a new form of entailment: KB $\models_{\text{CD}} \alpha$ iff KB$^\diamond \models \alpha$, where

KB$^\diamond$ = KB$^+ \cup \{\forall x[x = c_1 \vee \ldots \vee x = c_n]\}$,
where $c_1, \ldots, c_n$, are all the constant symbols appearing in KB.

So this is exactly like the CWA, but with the additional assumption that no objects exist apart from the named constants. Returning to the smallTown example, since ¬DirectConnect(smallTown, c_i) is entailed under the CWA for every c_i, it will follow that $\neg\exists x$DirectConnect(smallTown, x) is entailed under the CWA with domain closure.

The main property of this extension to the CWA is the following:

KB $\models_{\text{CD}} \forall x\alpha$ iff KB $\models_{\text{CD}} \alpha^x_c$, for every c appearing in KB
KB $\models_{\text{CD}} \exists x\alpha$ iff KB $\models_{\text{CD}} \alpha^x_c$, for some c appearing in KB.

This means that the correspondence between entailment conditions and truth conditions now generalizes to quantified sentences. With this additional completeness assumption, it is the case that KB $\models_{\text{CD}} \alpha$ or KB $\models_{\text{CD}} \neg\alpha$, for any α even with quantifiers. Similarly, the recursive query operation, which reduces queries to the atomic

case, now works for quantified sentences as well. This property can also be extended to deal with formulas with equality (and hence all of FOL) by including a *unique name assumption*, which adds to KB$^\diamond$ all sentences of the form ($c \neq c'$), for distinct constants c and c'.

Finally there is the issue of consistency. First note that domain closure does not rule out the use of function symbols. If we use sentences like $\forall x P(x) \supset P(f(x))$, then under the CWA with domain closure, we end up assuming that each term $f(t)$ is equal to one of the constants. In other words, even though individuals have a unique constant name, they can have other non-constant names.

However, it is possible to construct a KB that is inconsistent with domain closure in more subtle ways. Consider, for instance, the following:

$$P(c),\ \forall x\neg R(x,x),\ \forall x[P(x)\supset\exists y(R(x,y)\wedge P(y))]$$

This KB is consistent and does not even use equality. However, KB$^\diamond$ is inconsistent. The individual denoted by c cannot be the only instance of P since the other two sentences in effect assert that there must be another one. It is also possible to have a consistent KB that asserts the existence of infinitely many instances of P, guaranteeing that domain closure cannot be used for any finite set of constants. It is worth noting, on the other hand, that such examples are somewhat farfetched; they look more like formulas that might appear in axiomatizations of set theory than in databases. For "normal" applications, domain closure is much less of a problem.

12.3 Circumscription

In general terms, the CWA is the convention that arbitrary atomic sentences are taken to be false by default. Formally, $\models_{\text{C}}$ is defined as the entailments of KB$^+$, which is KB augmented by a set of negative literals. For a sentence α to be believed (under the CWA), it is not necessary for α to be true in all models of the KB, but only those that are also models of KB$^+$. In the first-order case, because of the presence of the negated literals in KB$^+$, we end up looking at models of the KB where the extension of the predicates is made as small as possible. This suggests a natural generalization: consider forms of entailment where the extension of certain predicates (perhaps not all) is as small as possible.

One way to handle default knowledge is to assume that we have a predicate Ab to talk about the exceptional cases where a default should not apply. Instead of saying that all birds fly, we might say:

$$\forall x[\text{Bird}(x) \wedge \neg\text{Ab}(x) \supset \text{Flies}(x)].$$

This can be read as saying that all birds that are not in some way abnormal fly, or more succinctly, that all normal birds fly.[6] Now imagine we have this fact in a KB along with these facts:

Bird(chilly), Bird(tweety), (tweety $\neq$ chilly), $\neg$Flies(chilly).

The intent here is clear: we would like to conclude by default that Tweety flies, whereas Chilly, of course, does not.

Note, however, that KB $\not\models$ Flies(tweety): there are interpretations satisfying the KB where Flies(tweety) is false. However, in these interpretations, the denotation of Tweety is contained in the extension of Ab. This then suggests a strategy for making default conclusions: as with the CWA, we will only consider certain interpretations of the KB, but in this case, only those where the Ab predicate is as small as possible. In other words, the strategy is to *minimize abnormality.* Intuitively, the default conclusions are taken to be those that are true in models of the KB where as few of the individuals as possible are abnormal.

In the above example, we already know that Chilly is an abnormal bird, but we do not know one way or another about Tweety. The default assumption we wish to make is that the extension of Ab is only as large as it has to be given what we know; hence it includes Chilly, since it has to because of Chilly's known abnormality, but excludes Tweety, because nothing that we know dictates that Ab must include her. This is called *circumscribing* the predicate Ab, and as a whole, the technique is called *circumscription*.

Note that while Chilly is abnormal in her flying ability, she may be quite normal in having two legs, laying eggs, and so on. This suggests that we do not really want to use a single predicate Ab, and not be able to assume any defaults at all about Chilly, but rather have a family of predicates Ab_i for talking about the various aspects of individuals. Chilly might be in the extension of Ab_1, but not in that of Ab_2, for instance.

12.3.1 Minimal entailment

Circumscription is intended to be a much more fine-grained tool than the CWA, and because of this and the fact that we wish to apply it in much broader settings, the formalization we use does not involve adding negative literals to the KB. Instead, we characterize a new form of entailment directly in terms of properties of interpretations themselves.

Let $\boldsymbol{P}$ be a fixed set of unary predicates, which we will intuitively understand to be the Ab predicates. Let $\Im_1$ and $\Im_2$ be logical interpretations over the same domain

[6]We are not suggesting that this is exactly what is meant by the sentence "Birds fly."

such that every constant and function is interpreted the same. So $\Im_1 = \langle \mathcal{D}, \mathcal{I}_1 \rangle$ and $\Im_2 = \langle \mathcal{D}, \mathcal{I}_2 \rangle$. Then we define the relationship, $\leq$:

$\Im_1 \leq \Im_2$ iff for every $P \in \boldsymbol{P}$, it is the case that $\mathcal{I}_1[P] \subseteq \mathcal{I}_2[P]$.

Also, $\Im_1 < \Im_2$ if and only if $\Im_1 \leq \Im_2$ but $\Im_2 \not\leq \Im_1$. Intuitively, given two interpretations over the same domain, we are saying that one is less than another in this ordering if it makes the extension of all the abnormality predicates smaller. Informally then, we can think of an interpretation that is less than another as *more normal*.

With this idea, we can define a new form of entailment $\models_{\leq}$ (which we call *minimal entailment*) as follows:

KB $\models_{\leq} \alpha$ iff for every interpretation $\Im$ such that $\Im \models$ KB, either $\Im \models \alpha$ or there is an $\Im'$ such that $\Im' < \Im$ and $\Im' \models$ KB.

This is very similar to the definition of entailment itself: we require every interpretation that satisfies KB to satisfy α except that it may be excused when there is another more normal interpretation that also satisfies the KB. Roughly speaking, we do not require α to be true in *all* interpretations satisfying the KB, but only in the minimal or *most normal* ones satisfying the KB.[7]

Consider for example, the KB above with Tweety and Chilly. As noted, KB $\not\models$ Flies(tweety). However, KB $\models_{\leq}$ Flies(tweety). The reason is this: if $\Im \models$ KB but $\Im \not\models$ Flies(tweety), then $\Im \models$ Ab(tweety). So let $\Im'$ be exactly $\Im$ except that we remove the denotation of tweety from the extension of Ab. Then $\Im' < \Im$ (assuming $\boldsymbol{P} = \{\mathsf{Ab}\}$, of course), and $\Im' \models$ KB. Thus, in the minimal models of the KB, Tweety is a normal bird: KB $\models_{\leq} \neg$Ab(tweety), from which we can infer that Tweety flies. We cannot do the same for Chilly, since in all models of the KB, normal or not, Chilly is an abnormal bird. Note that the only default step in this reasoning was to conclude that Tweety was normal; the rest was ordinary deductive reasoning given what we know about normal birds. This then is the circumscription proposal for formalizing default reasoning.

Note that in general, we do not expect the "most normal" models of the KB all to satisfy exactly the same sentences. Suppose for example, a KB contains Bird(c), Bird(d), and ($\neg$Flies(c) $\vee$ $\neg$Flies(d)). Then in any model of the KB, the extension of Ab must contain either the denotation of c or the denotation of d. Any model that contains other abnormal individuals (including ones where the denotations of both

[7]This is a convenient but slightly inaccurate way of putting it. In fact, there may be no "most normal" models; we could have an infinite descending chain of ever more normal models. However, the definition as presented above works even in such situations.

c and d are abnormal) would not be minimal. Because we need to consider what is true in *all* minimal models, we see that KB $\not\models_{\leq}$ Flies(c) and KB $\not\models_{\leq}$ Flies(d). In other words, we cannot conclude by default that c is a normal bird, nor that d is. However, what we can conclude by default is that *one of them* is normal: KB $\models_{\leq}$ Flies(c) $\lor$ Flies(d).

This is very different from the behavior of the CWA. Under similar circumstances, because it is consistent with what is known that c is normal, using the CWA we would add the literal $\neg$Ab(c), and by similar reasoning, $\neg$Ab(d), leading to inconsistency. Thus circumscription is more cautious than the CWA in the assumptions it makes about "controversial" individuals, like those denoted by c and d. However, circumscription is less cautious than the GCWA: the GCWA would not conclude anything about either the denotation of c or d, whereas circumscription is willing to conclude by default that one of them flies.

Another difference between circumscription and the CWA involves quantified sentences. By using interpretations directly rather than adding literals to the KB, circumscription works equally well with unnamed individuals. For example, if the KB contains $\exists x$[Bird(x) $\land$ ($x \neq$ chilly) $\land$ ($x \neq$ tweety) $\land$ InTree(x)], then with circumscription we would conclude by default that this unnamed individual flies:

$$\exists x[\text{Bird}(x) \land (x \neq \text{chilly}) \land (x \neq \text{tweety}) \land \text{InTree}(x) \land \text{Flies}(x)].$$

The reason here is the same as before: in the minimal models there will be a single abnormal individual, Chilly. This also carries over to unnamed abnormal individuals. If our KB contains the assertion that

$$\exists x[\text{Bird}(x) \land (x \neq \text{chilly}) \land (x \neq \text{tweety}) \land \neg\text{Flies}(x)],$$

then a model of the KB will be minimal if and only if there are exactly two abnormal individuals: Chilly, and the unnamed one. Thus, we conclude by default that

$$\exists x \forall y[(\text{Bird}(y) \land \neg\text{Flies}(y)) \equiv (y = \text{chilly} \lor y = x)].$$

So unlike the CWA and the GCWA, we do not need to name exceptions explicitly to avoid inconsistency. Indeed, the issue of consistency for circumscription is considerably more subtle than it was for the CWA, and characterizing it precisely remains an open question.

12.3.2 The circumscription axiom

One of the conceptual advantages of the CWA is that, although it is a form of nonmonotonic reasoning, we can understand its effect in terms of ordinary deductive

reasoning over a KB that has been augmented by certain assumptions. As we saw above, we cannot duplicate the effect of circumscription by simply adding a set of negative literals to a KB.

We *can*, however, view the effect of circumscription in terms of ordinary deductive reasoning from an augmented KB if we are willing to use *second-order logic*. Without going into details, it is worth observing that for any KB, there is a second-order sentence τ such that KB $\models_{\leq} \alpha$ if and only if KB $\cup \{\tau\} \models \alpha$ in second-order logic. What is required here of the sentence τ is that it should restrict interpretations to be minimal in the ordering. That is, if an interpretation $\Im$ is such that $\Im \models$KB, what we need (to get the correspondence with $\models_{\leq}$) is that $\Im \models \tau$ if and only if there does not exist $\Im' < \Im$ such that $\Im' \models$ KB. The idea here (due to John McCarthy) is that instead of talking about another interpretation $\Im'$, we could just as well have said that there must not exist a smaller extension for the Ab predicates that would also satisfy the KB. This requires quantification over the extensions of Ab predicates, and is what makes τ second-order.

12.3.3 Fixed and variable predicates

Although the default assumptions made by circumscription are usually weaker than those of the CWA, there are cases where it appears too strong. Suppose, for example, that we have the following KB:

$$\forall x[\text{Bird}(x) \land \neg\text{Ab}(x) \supset \text{Flies}(x)],$$
$$\text{Bird}(\text{tweety}),$$
$$\forall x[\text{Penguin}(x) \supset (\text{Bird}(x) \land \neg\text{Flies}(x))].$$

It then follows that $\forall x$[Penguin(x) $\supset$ Ab(x)], that is, with respect to flying anyway, penguins are abnormal birds.

The problem is this: to make default assumptions using circumscription, we end up minimizing the set of abnormal individuals. For the above KB, we conclude that there are no abnormal individuals at all:

$$\text{KB} \models_{\leq} \neg\exists x \text{Ab}(x).$$

But this has the effect of also minimizing penguins. In the process of wanting to derive the conclusion that Tweety flies, we end up concluding not only that Tweety is not a penguin, which is perhaps reasonable, but also that *there are no penguins*, which seems unreasonable:

$$\text{KB} \models_{\leq} \neg\exists x \text{Penguin}(x).$$

In our zeal to make things as normal as possible, we have ruled out penguins. What would be much better in this case, it seems, is to be able to conclude by default merely that penguins are the only abnormal birds.

One solution that has been proposed is to redefine $\models_\leq$ so that in looking at more normal worlds, we do not in the process exclude the possibility of exceptional classes like penguins. What we should say is something like this: we can ignore a model of the KB if there is a similar model with fewer abnormal individuals, *but with exactly the same penguins*. That is, in the process of minimizing abnormality, we should not be allowed to also minimize the set of penguins. We say that the extension of Penguin remains *fixed* in the minimization. But it is not as if all predicates other than Ab will remain fixed. In moving from a model $\Im$ to a lesser model $\Im'$ where Ab has a smaller extension, we are willing to change the extension of Flies, and indeed to conclude that Tweety flies. We say that the extension of Flies is *variable* in the minimization.

More formally, we redefine $\leq$ with respect to a set of unary predicates $\boldsymbol{P}$ (understood as the ones to be minimized) and a set of arbitrary predicates $\boldsymbol{Q}$ (understood as the predicates that are fixed in the minimization). Let $\Im_1$ and $\Im_2$ be as before. Then $\Im_1 \leq \Im_2$ if and only if for every $P \in \boldsymbol{P}$, it is the case that $\mathcal{I}_1[P] \subseteq \mathcal{I}_2[P]$, and for every $Q \in \boldsymbol{Q}$, it is the case that $\mathcal{I}_1[Q] = \mathcal{I}_2[Q]$. The rest of the definition of $\models_\leq$ is as before. Taking $\boldsymbol{P} = \{\mathsf{Ab}\}$ and $\boldsymbol{Q} = \{\mathsf{Penguin}\}$ amounts to saying that we want to minimize the instances of Ab holding constant the instances of Penguin. The earlier version of $\models_\leq$ was simply one where $\boldsymbol{Q}$ was empty.

Returning to the example bird KB, there will now be minimal models where there are penguins: KB $\not\models_\leq \neg\exists x \mathsf{Penguin}(x)$. In fact, a model of the KB will be minimal if and only if its abnormal individuals are precisely the penguins: obviously the penguins must be abnormal; conversely, assume to the contrary that in interpretation $\Im$ we have an abnormal individual o who is not one of the penguins. Then construct $\Im'$ by moving o out of the extension of Ab and, if it is in the extension of Bird, into the extension of Flies. Clearly, $\Im'$ satisfies KB and $\Im' < \Im$. So it follows that

$$\text{KB} \models_\leq \forall x[(\mathsf{Bird}(x) \land \neg\mathsf{Flies}(x)) \equiv \mathsf{Penguin}(x)].$$

Unfortunately, this version of circumscription still has some serious problems. For one thing, our method of using circumscription needs to specify not only which predicates to minimize, but also which additional predicates to keep fixed: we need to be able to figure out somehow beforehand that flying should be a variable predicate, for example, and it is far from clear how.

More seriously perhaps, KB $\not\models_\leq$ Flies(tweety). The reason is this: consider a model of the KB where Tweety happens to be a penguin; we can no longer find a

lesser model where Tweety flies since that would mean changing the set of penguins, which must remain fixed. What we do get is that

$$\text{KB} \models_\leq \neg\mathsf{Penguin}(\mathsf{tweety}) \supset \mathsf{Flies}(\mathsf{tweety}).$$

So if we know that Tweety is not a penguin, as in

$$\mathsf{Canary}(\mathsf{tweety}),\ \forall x[\mathsf{Canary}(x) \supset \neg\mathsf{Penguin}(x)],$$

then we get the desired conclusion. But this is not derivable by default. Even if we add something saying that birds are normally not penguins, as in

$$\forall x[\mathsf{Bird}(x) \land \neg\mathsf{Ab}_2(x) \supset \neg\mathsf{Penguin}(x)],$$

Tweety still does not fly, because we cannot change the set of penguins. Various solutions to this problem have been proposed in the literature, but none are completely satisfactory.

In fact, this sort of problem was already there in the background with the earlier version of circumscription. For example, consider the KB we had before with Tweety and Chilly, but this time without (tweety $\neq$ chilly). Then as with the penguins, we lose the assumption that Tweety flies and only get

$$\text{KB} \models_\leq (\mathsf{tweety} \neq \mathsf{chilly}) \supset \mathsf{Flies}(\mathsf{tweety}).$$

The reason is that there is a model of the KB with a minimal number of abnormal birds where Tweety does not fly, namely one where Chilly and Tweety are the same bird.[8] Putting Chilly aside, all it really takes is the existence of a single abnormal bird: if the KB contains $\exists x[\mathsf{Bird}(x) \land \neg\mathsf{Flies}(x)]$, then although we can assume by default that this flightless bird is unique, we have not ruled out the possibility that Tweety is that bird, and we can no longer assume by default that Tweety flies. This means that there is a serious limitation in using circumscription for default reasoning: we must ensure that any abnormal individual is known to be distinct from the other individuals.

12.4 Default logic

In the previous section, we introduced the idea of circumscription as a generalization of the CWA: instead of minimizing all predicates, we minimize abnormality

[8]It would be nice here to be able to somehow conclude *by default* that any two named constants denote distinct individuals. Unfortunately, it can be shown that this cannot be done using a mechanism like circumscription.

predicates. Of course, in the CWA section above, we looked at it differently: we thought of it as deductive reasoning from a KB that had been enlarged by certain default assumptions, the negative literals that are added to form KB$^+$.

A generalization in a different direction then suggests itself: instead of adding to a KB all negative literals that are consistent with the KB, we provide a mechanism for specifying explicitly which sentences should be added to the KB when it is consistent to do so. For example, if $\mathsf{Bird}(t)$ is entailed by the KB, we might want to add the default assumption $\mathsf{Flies}(t)$, if it is consistent to do so. Or perhaps this should only be done in certain contexts.

This is the intuition underlying *default logic*. A KB is now thought of as a *default theory* consisting of two parts, a set $\mathcal{F}$ of first-order sentences as usual, and a set $\mathcal{D}$ of *default rules*, which are specifications of what assumptions can be made and when. The job of a default logic is then to specify what the appropriate set of implicit beliefs should be, somehow incorporating the facts in $\mathcal{F}$, as many default assumptions as we can, given the default rules in $\mathcal{D}$, and the logical entailments of both. As we will see, defining these implicit beliefs is non-trivial: in some cases, there will be more than one candidate set of sentences that could be regarded as a reasonable set of beliefs (just as there could be multiple preferred extensions in Chapter 10); in other cases, no set of sentences seems to work properly.

12.4.1 Default rules

Perhaps the most general form of default rule that has been examined in the literature is due to Reiter: it consists of three sentences, a *prerequisite* α, a *justification* β, and a *conclusion* δ. The informal interpretation of this triple is that δ should be believed if α is believed and it is consistent to believe β. That is, if we have α and we do not have $\neg\beta$, then we can assume δ. We will write such a rule as $\langle\, \alpha; \beta; \delta \,\rangle$.

For example, a rule might be $\langle\, \mathsf{Bird(tweety)}; \mathsf{Flies(tweety)}; \mathsf{Flies(tweety)} \,\rangle$. This says that if we know that Tweety is bird, then we should assume that Tweety flies if it is consistent to assume that Tweety flies. This type of rule, where the justification and conclusion are the same, is called a *normal default rule* and is by far the most common case. We will sometimes write such rules as $\mathsf{Bird(tweety)} \Rightarrow \mathsf{Flies(tweety)}$. We call a default theory all of whose rules are normal a *normal default theory*. As we will see below, there are cases where non-normal defaults are useful.

Note that the rules in the above are particular to Tweety. In general, we would like rules that could apply to any bird. To do so, we allow a default rule to use formulas with free variables. These should be understood as abbreviations for the set of all substitution instances. So, for example, $\langle\, \mathsf{Bird}(x); \mathsf{Flies}(x); \mathsf{Flies}(x) \,\rangle$ stands

for all rules of the form $\langle\, \mathsf{Bird}(t); \mathsf{Flies}(t); \mathsf{Flies}(t) \,\rangle$ where t is any closed term. This will allow us to conclude by default of any bird that it flies, without also forcing us to believe by default that *all* birds fly, a useful distinction.

12.4.2 Default extensions

Given a default theory KB = $(\mathcal{F}, \mathcal{D})$, what sentences ought to be believed? We will call a set of sentences that constitute a reasonable set of beliefs given a default theory an *extension* of the theory. In this subsection, we present a simple and workable definition of extension; in the next, we will argue that sometimes a more complex definition is called for.

For our purposes, a set of sentences $\mathcal{E}$ is an extension of a default theory $(\mathcal{F}, \mathcal{D})$ if and only if for every sentence π,

$$\pi \in \mathcal{E} \quad \text{iff} \quad \mathcal{F} \cup \{\delta \mid \langle\, \alpha; \beta; \delta \,\rangle \in \mathcal{D}, \alpha \in \mathcal{E}, \neg\beta \notin \mathcal{E}\} \models \pi.$$

Thus, a set of sentences is an extension if it is the set of all entailments of $\mathcal{F} \cup \Delta$, where Δ is a suitable set of assumptions. In this respect, the definition of extension is similar to the definition of the CWA: we add default assumptions to a set of basic facts. Here, the assumptions to be added are those that we will call *applicable to the extension* $\mathcal{E}$: an assumption is applicable if and only if it is the conclusion of a default rule whose prerequisite is in the extension and the negation of whose justification is not. Note that we require α to be in $\mathcal{E}$, not in $\mathcal{F}$. This has the effect of allowing the prerequisite to be believed as the result of other default assumptions, and therefore, of allowing default rules to chain. Note also that this definition is not constructive: it does not tell us how to find an $\mathcal{E}$ given $\mathcal{F}$ and $\mathcal{D}$, or even if there is one or more than one to be found. However, given $\mathcal{F}$ and $\mathcal{D}$, the $\mathcal{E}$ is completely characterized by its set of applicable assumptions, Δ.

For example, suppose we have the following normal default theory:

$$\mathcal{F} = \{\mathsf{Bird(tweety)}, \mathsf{Bird(chilly)}, \neg\mathsf{Flies(chilly)}\}$$
$$\mathcal{D} = \{\mathsf{Bird}(x) \Rightarrow \mathsf{Flies}(x)\}.$$

We wish to show that there is a unique extension to this default theory characterized by the assumption $\mathsf{Flies(tweety)}$. To show this, we must first establish that the entailments of $\mathcal{F} \cup \{\mathsf{Flies(tweety)}\}$—call this set $\mathcal{E}$—are indeed an extension according to the above definition. This means showing that $\mathsf{Flies(tweety)}$ is the only assumption applicable to $\mathcal{E}$: it is applicable since $\mathcal{E}$ contains $\mathsf{Bird(tweety)}$ and does not contain $\neg\mathsf{Flies(tweety)}$. Moreover, for no other t is $\mathsf{Flies}(t)$ applicable, since $\mathcal{E}$ contains $\mathsf{Bird}(t)$ only for t = chilly, for which $\mathcal{E}$ also contains $\neg\mathsf{Flies(chilly)}$. So this

$\mathcal{E}$ is indeed an extension. Observe that unlike circumscription, we do not require Tweety and Chilly to be distinct to draw the default conclusion.

But are there other extensions? Assume that some $\mathcal{E}'$ is also an extension for some applicable set of assumptions $\mathsf{Flies}(t_1), \ldots, \mathsf{Flies}(t_n)$. First observe that no matter what Flies assumptions we make, we will never be able to conclude that $\neg\mathsf{Flies}(\mathsf{tweety})$. Thus $\mathsf{Flies}(\mathsf{tweety})$ must be applicable to $\mathcal{E}'$. However, we will not be able to conclude $\mathsf{Bird}(t)$, for any t other that tweety or chilly. So $\mathsf{Flies}(\mathsf{tweety})$ is the only applicable assumption, and therefore $\mathcal{E}'$ must be the entailments of $\mathcal{F} \cup \{\mathsf{Flies}(\mathsf{tweety})\}$, as above.

In arguing above that there was a unique extension, we made statements like "no matter what assumptions we make, we will never be able to conclude α." Of course, if $\mathcal{E}$ is *inconsistent* we can conclude anything we want. For example, if we could somehow add the assumption $\mathsf{Flies}(\mathsf{chilly})$, then we could conclude $\mathsf{Bird}(\mathsf{george})$. It turns out that such contradictory assumptions are never possible: an extension $\mathcal{E}$ of a default theory $(\mathcal{F}, \mathcal{D})$ is inconsistent if and only if $\mathcal{F}$ is inconsistent.

12.4.3 Multiple extensions

Now consider the following default theory:

$$\mathcal{F} = \{\mathsf{Republican}(\mathsf{dick}),\ \mathsf{Quaker}(\mathsf{dick})\}$$
$$\mathcal{D} = \{\mathsf{Republican}(x) \Rightarrow \neg\mathsf{Pacifist}(x),\ \mathsf{Quaker}(x) \Rightarrow \mathsf{Pacifist}(x)\}.$$

Here, there are two defaults that are in conflict for Dick. There are, correspondingly two extensions:

1. $\mathcal{E}_1$ is characterized by the assumption $\mathsf{Pacifist}(\mathsf{dick})$.
2. $\mathcal{E}_2$ is characterized by the assumption $\neg\mathsf{Pacifist}(\mathsf{dick})$.

Both of these are extensions since their assumption is applicable, and no other assumption (for any t other than dick) is. Moreover, there are no other extensions: The empty set of assumptions does not give an extension since both $\mathsf{Pacifist}(\mathsf{dick})$ and $\neg\mathsf{Pacifist}(\mathsf{dick})$ would be applicable; for any other potential extension, assumptions would be of the form $\mathsf{Pacifist}(t)$ or $\neg\mathsf{Pacifist}(t)$ none of which are applicable for any t other than dick, since we will never have the corresponding prerequisite $\mathsf{Quaker}(t)$ or $\mathsf{Republican}(t)$ in $\mathcal{E}$. Thus, $\mathcal{E}_1$ and $\mathcal{E}_2$ are the only extensions.

So what default logic tells us here is that we may choose to assume that Dick is a pacifist or that he is not a pacifist. On the basis of what we have been told, either set of beliefs is reasonable. As in the case of inheritance hierarchies in Chapter 10, there are two immediate possibilities:

1. a *skeptical* reasoner will only believe those sentences that are common to all extensions of the default theory;

2. a *credulous* reasoner will simply choose arbitrarily one of the extensions of the default theory as the set of sentences to believe.

Arguments for and against each type of reasoning have been made. Note, that minimal entailment, in giving us what is true in *all* minimal models is much more like skeptical reasoning.

In some cases, the existence of multiple extensions is merely an indication that we have not said enough to make a reasonable decision. In the above example, we may want to say that the default regarding Quakers should only apply to individuals not known to be politically active. Assuming we have the fact

$$\forall x[\mathsf{Republican}(x) \supset \mathsf{Political}(x)],$$

we can replace the original rule with $\mathsf{Quaker}(x)$ as the prerequisite by a non-normal one like

$$\langle\ \mathsf{Quaker}(x); (\mathsf{Pacifist}(x) \wedge \neg\mathsf{Political}(x)); \mathsf{Pacifist}(x)\ \rangle.$$

Then, for ordinary Republicans and ordinary Quakers, the assumption would be as before; for Quaker Republicans like Dick, we would assume (unequivocally) that they were not pacifists. Note that if we merely say that Republicans are politically active *by default*, we would again be left with two extensions.

This idea of arbitrating among conflicting default rules is crucial when it comes to dealing with concept hierarchies. For example, suppose we have a KB that contains $\forall x[\mathsf{Penguin}(x) \supset \mathsf{Birds}(x)]$ together with two default rules:

$$\mathsf{Bird}(x) \Rightarrow \mathsf{Flies}(x)$$
$$\mathsf{Penguin}(x) \Rightarrow \neg\mathsf{Flies}(x).$$

If we also have $\mathsf{Penguin}(\mathsf{chilly})$, we get two extensions: one where Chilly is assumed to fly and one where Chilly is assumed not to fly. Unlike the Quaker Republican example, however, what ought to have happened here is clear: the default that penguins do not fly should *preempt* the more general default that birds fly. In other words, we only want one extension, where Chilly is assumed not to fly. To get this in default logic, it is necessary to encode the penguin case as part of the justification in a non-normal default for birds:

$$\langle\ \mathsf{Bird}(\mathsf{tweety}); (\mathsf{Flies}(\mathsf{tweety}) \wedge \neg\mathsf{Penguin}(\mathsf{tweety})); \mathsf{Flies}(\mathsf{tweety})\ \rangle.$$

This is not a very satisfactory solution since there may be a very large number of interacting defaults to consider:

$$\langle \text{Bird(tweety)}; [\text{Flies(tweety)} \land \neg\text{Penguin(tweety)} \land \neg\text{Emu(tweety)} \land \neg\text{Ostrich(tweety)} \land \neg\text{Dead(tweety)} \land \ \ldots]; \text{Flies(tweety} \rangle.$$

It is a severe limitation of default logic and indeed of all the default formalisms considered in this chapter that unlike the inheritance formalism of Chapter 10, they do not automatically prefer the most specific defaults in cases like this.

Now consider the following example. Suppose we have a default theory $(\mathcal{F}, \mathcal{D})$ where $\mathcal{F}$ is empty and $\mathcal{D}$ contains a single non-normal default $\langle$ TRUE; $p; \neg p \rangle$, where p is any atomic sentence. This default theory has *no* extensions: if $\mathcal{E}$ were an extension, then $\neg p \in \mathcal{E}$ iff $\neg p$ is an applicable assumption iff $\neg p \notin \mathcal{E}$. This means that with this default rule, there is no reasonable set of beliefs to hold. Having no extension is very different from having a single but inconsistent one, such as when $\mathcal{F}$ is inconsistent. A skeptical believer might go ahead and believe all sentences (since every sentence is trivially common to all the extensions), but a credulous believer is stuck. Fortunately, this situation does not arise with normal defaults, as it can be proven that every normal default theory has at least one extension.

An even more serious problem is shown in the following example. Suppose we have a default theory $(\mathcal{F}, \mathcal{D})$ where $\mathcal{F}$ is empty and $\mathcal{D}$ contains a single non-normal default $\langle p$; TRUE; $p \rangle$. This theory has two extensions, one of which is the set of all valid sentences, and the other of which is the set $\mathcal{E}$ consisting of the entailments of p. (The assumption p is applicable here since $p \in \mathcal{E}$ and $\neg$TRUE $\notin \mathcal{E}$.) However, on intuitive grounds, this second extension is quite inappropriate. The default rule says that p can be assumed if p is believed. This really should not allow us to conclude by default that p is true any more than a fact saying that p is true if p is true would. It would be much better to end up with a single extension consisting of just the valid sentences, since there is no good reason to believe p by default.

One way to resolve this problem is to rule out any extension for which a proper subset is also an extension. This works for this example, but fails on other examples. A more complex definition of extension, due to Reiter, appears to handle all such anomalies: Let $(\mathcal{F}, \mathcal{D})$ be any default theory. For any set S, let $\Delta(S)$ be the least set containing $\mathcal{F}$, closed under entailment, and satisfying the following:

If $\langle \alpha; \beta; \delta \rangle \in \mathcal{D}$, $\alpha \in \Delta(S)$, $\neg\beta \notin S$, then $\delta \in \Delta(S)$.

Then a set $\mathcal{E}$ is a *grounded extension* of $(\mathcal{F}, \mathcal{D})$ if and only if $\mathcal{E} = \Delta(\mathcal{E})$. This definition is considerably more complex to work with than the one we have considered, but does have some desirable properties, including handling the above example correctly, while agreeing with the simpler definition on all of the earlier examples.

We will not pursue this version in any more detail except to observe one simple feature: in the definition of $\Delta(S)$, we test if $\neg\beta \notin S$, rather than $\neg\beta \notin \Delta(S)$. Had we gone with the latter, the definition of $\Delta(S)$ would have been this: the least set containing $\mathcal{F}$, closed under entailment, and containing all of its applicable assumptions. Except for the part about "least set", this is precisely our earlier definition of extension. So this very small change to how justifications are considered ends up making all the difference.

12.5 Autoepistemic logic

One advantage circumscription has over default logic is that defaults end up as ordinary *sentences* in the language (using abnormality predicates). In default logic, although we can reason *with* defaults, we cannot reason *about* them. For instance, suppose we have the default $\langle \alpha; \beta; \delta \rangle$. It would be nice to say that we also implicitly have the defaults $\langle (\alpha \land \alpha'); \beta; \delta \rangle$ and $\langle \alpha; \beta; (\delta \lor \delta') \rangle$. Similarly, we might want to say that we also have the "contrapositive" default $\langle \neg\delta; \beta; \neg\alpha \rangle$. But these questions cannot even be posed in default logic since, despite its name, it is not a logic of defaults at all, as there is no notion of entailment among defaults. On the other hand, default logic deals more directly with what it is consistent to assume, whereas circumscription forces us to handle defaults in terms of abnormalities. The consistency in default logic is, of course, relative to what is currently believed. This suggests another approach to default reasoning where like circumscription, defaults are represented as sentences, but like default logic, these sentences talk about what it is consistent to assume.

Roughly speaking, we will represent the default about birds, for example, by

Any bird that can be consistently believed to fly does fly.

Given that beliefs (as far as we are concerned) are closed under entailment, then a sentence can be consistently believed if and only if its negation is not believed. So we can restate the default as

Any bird not believed to be flightless flies.

To encode defaults like these as sentences in a logic, we extend the FOL language to talk about belief directly. In particular, we will assume that for every formula α, there is another formula $\mathbf{B}\alpha$ to be understood informally as saying "α is believed to be true." The **B** should be thought of as a new unary connective (like negation). Defaults, then, are represented by sentences like

$$\forall x[\text{Bird}(x) \land \neg\mathbf{B}\neg\text{Flies}(x) \supset \text{Flies}(x)].$$

For this to work, it must be the case that saying that a sentence α is true is not the same as saying that $\mathbf{B}\alpha$ is true; just because something is true should not mean that it is known to be true.[9] Since we imagine reasoning using sentences like the above, we will be reasoning about birds of course, but also about *what we believe about birds*. The fact that this is a logic about our own beliefs is why it is called *autoepistemic logic.*

12.5.1 Stable sets and expansions

As usual, our primary concern is to determine a reasonable set of beliefs in the presence of defaults. With autoepistemic logic, the question is: given a KB that contains sentences using the **B** operator, what is a reasonable set of beliefs to hold? To answer this question, we begin by examining some minimal properties we expect any set of beliefs $\mathcal{E}$ to satisfy. We call a set $\mathcal{E}$ *stable* if and only if it satisfies these three properties:

1. Closure under entailment: If $\mathcal{E} \models \alpha$, then $\alpha \in \mathcal{E}$.
2. Positive introspection: If $\alpha \in \mathcal{E}$, then $\mathbf{B}\alpha \in \mathcal{E}$.
3. Negative introspection: If $\alpha \notin \mathcal{E}$, then $\neg\mathbf{B}\alpha \in \mathcal{E}$.

So first, we want $\mathcal{E}$ to be closed under entailment. Since we have not yet defined entailment for a language with **B** operators, we take this simply to mean ordinary logical entailment, where we treat

$$\forall x[\mathsf{Bird}(x)\wedge\neg\mathbf{B}\neg\mathsf{Flies}(x) \supset \mathsf{Flies}(x)]$$

as if it were something like

$$\forall x[\mathsf{Bird}(x)\wedge\neg Q(x) \supset \mathsf{Flies}(x)]$$

where Q is a new predicate symbol.

The other two properties of a stable set deal with the **B** operator. They ensure that if α is believed then so is $\mathbf{B}\alpha$, and if α is not believed then $\neg\mathbf{B}\alpha$ is believed. These are called introspection constraints since they deal with beliefs about beliefs.

Given a KB, there will be many stable sets $\mathcal{E}$ that contain it. In deciding what sentences to believe, we want a stable set that contains the entailments of the KB and the appropriate introspective beliefs, but nothing else. This is called a *stable expansion* of the KB and its formal definition, due to Robert Moore, is this: a set $\mathcal{E}$ is a stable expansion of KB if and only if for every sentence π, it is the case that

[9] As we have been doing throughout the book, we use “know” and “believe” interchangeably. Unless otherwise indicated, “believe” is what is intended, and “know” is used for stylistic variety.

$$\pi \in \mathcal{E} \quad \text{iff} \quad \text{KB} \cup \{\mathbf{B}\alpha \mid \alpha \in \mathcal{E}\} \cup \{\neg\mathbf{B}\alpha \mid \alpha \notin \mathcal{E}\} \models \pi.$$

This is a familiar pattern: the implicit beliefs $\mathcal{E}$ are those sentences that are entailed by KB $\cup\, \Delta$, where Δ is a suitable set of assumptions. In this case, the assumptions are those arising from the introspection constraints.

To see how this leads to default reasoning, suppose we a have a KB that consists of the following:

$$\mathsf{Bird(chilly)}, \mathsf{Bird(tweety)}, (\mathsf{tweety} \neq \mathsf{chilly}), \neg\mathsf{Flies(chilly)},$$
$$\forall x[\mathsf{Bird}(x) \wedge \neg\mathbf{B}\neg\mathsf{Flies}(x) \supset \mathsf{Flies}(x)].$$

Informally, let’s consider the consequences of this KB. First we see that there is no way to conclude ¬Flies(tweety): ¬Flies(tweety) is not explicitly in the knowledge base, and there is no rule that would allow us to conclude it, even by default (the conclusion of our one rule is of the form Flies(x)). This means that if $\mathcal{E}$ is a stable expansion of the KB, it will not include this fact. But because of our negative introspection property, a stable expansion that did not include the fact ¬Flies(tweety) would include the assumption, ¬**B**¬Flies(tweety). Now given this assumption,[10] and the fact that $\forall x[\mathsf{Bird}(x) \wedge \neg\mathbf{B}\neg\mathsf{Flies}(x) \supset \mathsf{Flies}(x)]$ is in the KB, we conclude Flies(tweety) using ordinary logical entailment. So in autoepistemic logic, default assumptions are typically of the form $\neg\mathbf{B}\alpha$, and new default beliefs about the world, like Flies(tweety), are deduced from these assumptions.

12.5.2 Enumerating stable expansions

The above illustrated informally how the notion of a stable expansion of a knowledge base can account for default reasoning of a certain sort. To be more precise about this, and show that the KB above does in fact have a stable expansion containing Flies(tweety), and that it is unique, we will consider the simpler propositional version of the definition and show how to enumerate stable expansions. In the propositional case, we replace the sentence,

$$\forall x[\mathsf{Bird}(x) \wedge \neg\mathbf{B}\neg\mathsf{Flies}(x) \supset \mathsf{Flies}(x)]$$

by all of its instances, as we did with default rules in the previous section.

Let us a call a sentence *objective* if it does not contain any **B** operators. The first thing to observe is that in the propositional case, a stable expansion is completely determined by its objective subset; the non-objective part can be reconstructed using the two introspection constraints and logical entailment.

[10] This really is an assumption, since ¬**B**¬Flies(tweety) does not follow from what is in the KB; the KB does not specify one way or another.

Figure 12.1: A procedure to generate stable expansions

Input: a propositional KB, containing subformulas $\mathbf{B}\alpha_1, \mathbf{B}\alpha_2, \ldots, \mathbf{B}\alpha_n$
Output: the objective part of a stable expansion of the KB.

1. Replace each $\mathbf{B}\alpha_i$ in KB by either TRUE or ¬TRUE.
2. Simplify, and call the resulting objective knowledge base KB°.
3. If $\mathbf{B}\alpha_i$ was replaced by TRUE, confirm that KB° $\models \alpha_i$; if $\mathbf{B}\alpha_i$ was replaced by ¬TRUE, confirm that KB° $\not\models \alpha_i$.
4. If the condition is confirmed for every $\mathbf{B}\alpha_i$, then return KB°, whose entailments form the objective part of a stable expansion.

So imagine we have a KB that contains objective and non-objective sentences, where $\mathbf{B}\alpha_1, \mathbf{B}\alpha_2, \ldots, \mathbf{B}\alpha_n$ are all the **B** formulas mentioned. Assume for simplicity that all the α_i are objective. If we knew which of the $\mathbf{B}\alpha_i$ formulas were true in a stable expansion, we could calculate the objective part of that stable expansion using ordinary logical reasoning. So the procedure we will use is to *guess* nondeterministically which of the $\mathbf{B}\alpha_i$ formulas are true, and then check whether the result makes sense as the objective part of a stable expansion: if we guessed that $\mathbf{B}\alpha_i$ was true, we need to confirm that α_i is entailed; if we guessed that $\mathbf{B}\alpha_i$ was false, we need to confirm that α_i is not entailed. A more precise version of this procedure is shown in Figure 12.1. Observe that using this procedure we can generate at most 2^n stable expansions.

To see this procedure in action, consider a propositional version of the flying bird example. In this case, our KB is

Bird(chilly), Bird(tweety), ¬Flies(chilly),
[Bird(tweety) ∧ ¬**B**¬Flies(tweety) ⊃ Flies(tweety)],
[Bird(chilly) ∧ ¬**B**¬Flies(chilly) ⊃ Flies(chilly)].

There are two subformulas with **B** operators, **B**¬Flies(tweety) and **B**¬Flies(chilly), and so at most $2^2 = 4$ stable expansions. For each constant c, if **B**¬Flies(c) is true, then [Bird(c) ∧ ¬**B**¬Flies(c) ⊃ Flies(c)] simplifies to TRUE; if **B**¬Flies(c) is ¬TRUE then the sentence simplifies to [Bird(c) ⊃ Flies(c)] which will reduce to Flies(c), since the KB contains Bird(c). So, our four cases are these:

1. **B**¬Flies(tweety) true and **B**¬Flies(chilly) true, for which KB° is

   Bird(tweety), Bird(chilly), ¬Flies(chilly).

   This is the case because the two implications each simplify to TRUE. Then, following Step 3, for each of the two $\mathbf{B}\alpha_i$ formulas, which were replaced by TRUE, we need to confirm that the α_i are entailed by KB°. KB° does not entail ¬Flies(tweety). As a result, this is not a stable expansion.

2. **B**¬Flies(tweety) true and **B**¬Flies(chilly) false, for which KB° is

   Bird(tweety), Bird(chilly), ¬Flies(chilly), Flies(chilly).

   Following Step 3, we need to confirm that KB° entails ¬Flies(tweety) and that it does not entail ¬Flies(chilly). Since KB° entails ¬Flies(chilly), this is not a stable expansion (actually, this KB fails on both counts).

3. **B**¬Flies(tweety) false and **B**¬Flies(chilly) true, for which KB° is

   Bird(tweety), Bird(chilly), ¬Flies(chilly), Flies(tweety).

   Step 3 tells us to confirm that KB° entails ¬Flies(chilly) and does not entail ¬Flies(tweety). In this case, we succeed on both counts, and this characterizes a stable expansion.

4. Finally, **B**¬Flies(tweety) false and **B**¬Flies(chilly) false, for which KB° is

   Bird(tweety), Bird(chilly), ¬Flies(chilly), Flies(tweety), Flies(chilly).

   Since KB° entails ¬Flies(chilly), this is not a stable expansion.

Thus, this KB has a unique stable expansion, and in this expansion, Tweety flies.

As another example, we can use the procedure to show that (¬**B**p ⊃ p) has no stable expansion: if **B**p is false, then the KB° is p which entails p; conversely, if **B**p is true, then KB° is TRUE which does not entail p. So there is no stable expansion.

Similarly, we can use the procedure to show that the KB consisting of the sentences (¬**B**p ⊃ q) and (¬**B**q ⊃ p) has exactly two stable expansions: if **B**p is true and **B**q false, the KB° is p which entails p and does not entail q, and so this is the first stable expansion; symmetrically, the other stable expansion is when **B**p is false and **B**q true; if both are true, the KB° is TRUE which neither entails p nor q; and if both are false, the KB° is ($p \wedge q$) which entails both.

It is worth noting that as with default logic, in some cases, this definition of stable expansion may not be strong enough. Consider, for example, a KB consisting of a single sentence, (**B**p ⊃ p). Using the above procedure, we can see that there are two stable expansions: one containing p, and one that does not. But intuitively it seems like the first expansion is inappropriate: the only possible justification for believing p is **B**p itself. As in the default logic case, it seems that the assumption is not properly grounded.

A new definition of stable expansion (due to Konolige) has been proposed to deal with this problem: a set of sentences is a *minimal stable expansion* if and only if it is a stable expansion that is minimal in its objective sentences. In the above example, only the stable expansion not containing p would be a minimal stable expansion. However, further examples suggest that an even stronger definition may be required, for which there is an exact correspondence between stable expansions and the grounded extensions of default logic.

12.6 Conclusion

In this chapter, we have examined four different logical formalisms for default reasoning. While each of them does the job in many cases, they each have drawbacks of one sort or another. Getting a logical account of default reasoning that is simple, broadly applicable, and intuitively correct remains an open problem. In fact, because so much of what we know involves default reasoning, it is perhaps the central open problem in the whole area of knowledge representation. Not surprisingly, much of the theoretical research over the last twenty years has been on this topic.

12.7 Bibliographic notes

12.8 Exercises

Chapter 13

Abductive Reasoning

So far in this book we have concentrated on reasoning that is primarily *deductive* in nature: given a KB representing some explicit beliefs about the world, we try to deduce some α, to determine if it is an implicit belief, or perhaps to find a constant (or constants) c such that α^x_c is an implicit belief. This pattern shows up not only in ordinary logical reasoning, but also in description logics and procedural systems. In fact, a variant even shows up in probabilistic and default reasoning, where extra assumptions might be added to the KB, or degrees of belief might be considered.

In this chapter, we consider a completely different sort of reasoning task. Suppose we are given a KB and an α that we do not believe at all (even with default assumptions). We might ask the following: given what we already know, what would it take for us to believe that α was true? In other words, what else would we have to be told for α to become an implicit belief? One interesting aspect of this question is that the answer we are expecting will not be "yes" or "no" or the names of some individuals; instead, the answer should be a formula of the representation language.[1]

The typical pattern for deductive reasoning is as follows:

given $(p \supset q)$, from p, we can deduce q;

the corresponding pattern for what is called *abductive reasoning* is as follows:

given $(p \supset q)$, from q, we can abduce p;

Abductive reasoning is in some sense the converse of deductive reasoning: instead of looking for sentences entailed by p given what is known, we look for sentences

[1]In the last section of this chapter, we will see that it can be useful to have some deductive tasks that return formulas as well.

that would entail q given what is known.[2]

Another way to look at abduction is as a way of providing an *explanation*. The typical application of these ideas is in reasoning about causes and effects. Imagine that p is a cause (for example, "it is raining") and q is an effect (for example, "the grass is wet"). Deductive reasoning would be used to predict the effects of rain, *i.e.*, wet grass, among others; abductive reasoning would be used to conjecture the cause of wet grass, *i.e.*, rain, among others. In this case, we are trying to find something that would be sufficient to explain a sentence's being true.

13.1 Diagnosis

One case of reasoning about causes and effects where abductive reasoning appears especially useful is *diagnosis*. Imagine that we have a collection of facts in a KB of the form

$$(\mathit{Disease} \wedge \ldots \supset \mathit{Symptoms})$$

where the ellipsis is collection of hedges or qualifications. The goal of diagnosis is to find a disease (or diseases) that best explains a given set of observed symptoms.

Note that in this setting we would not expect to be able to reason deductively using facts of the form

$$(\mathit{Symptoms} \wedge \ldots \supset \mathit{Disease}),$$

because facts like these are much more difficult to obtain. Typically, a disease will have a small number of well-known symptoms, but a symptom can be associated with a large number of potential diseases (e.g., fever can be caused by hundreds of afflictions). It is usually much easier to account for an effect of a given cause than to prescribe a cause of a given effect. So the diagnosis we are looking for will not be an entailment of what is known; rather, it is merely a conjecture.

For example, imagine a KB containing the following (in non-quantified form, to keep things simple):

$$\begin{array}{l}\mathsf{TennisElbow} \supset \mathsf{SoreElbow},\\ \mathsf{TennisElbow} \supset \mathsf{TennisPlayer},\\ \mathsf{Arthritis} \wedge \neg\mathsf{Treated} \supset \mathsf{SoreJoints},\\ \mathsf{SoreJoints} \supset \mathsf{SoreElbow} \wedge \mathsf{SoreHips}.\end{array}$$

[2]The term "abduction" in this sense is due to the philosopher C. S. Peirce, who also discussed a third possible form of reasoning, *induction*, which takes as given (a number of instances of) both p and q, and induces that $(p \supset q)$ is true.

Now suppose we would like to explain an observed symptom: $\mathsf{SoreElbow}$. Informally, what we are after is a diagnosis like $\mathsf{TennisElbow}$ which clearly accounts for the symptom, given what is known. Another equally good diagnosis would be $(\mathsf{Arthritis} \wedge \neg\mathsf{Treated})$ which also explains the symptom. So we are imagining that there will in general be multiple explanations for any given symptom, quite apart from the fact that logically equivalent formulas like $(\neg\mathsf{Treated} \wedge \neg\neg\mathsf{Arthritis})$ would work as well.

13.2 Explanation

In characterizing precisely what we are after in an explanation, it useful to think in terms of four criteria:

Given KB and a formula β to be explained, we are looking for a formula α satisfying the following:

1. α is sufficient to account for β. More precisely, we want to find an α such that $\mathrm{KB} \cup \{\alpha\} \models \beta$, or equivalently, $\mathrm{KB} \models (\alpha \supset \beta)$. Any α that does not satisfy this property would be considered too weak to serve as an explanation for β.
2. α is not ruled out by the KB. More precisely, we want it to be the case that $\mathrm{KB} \cup \{\alpha\}$ is consistent, or equivalently, that $\mathrm{KB} \not\models \neg\alpha$. Without this, a formula like $(p \wedge \neg p)$, which always satisfies the first criterion above, would be a reasonable explanation. Similarly, if $\neg\mathsf{TennisPlayer}$ were a fact in the above KB, then even though $\mathsf{TennisElbow}$ would still entail $\mathsf{SoreElbow}$, it would not be an appropriate diagnosis.
3. α is as simple and parsimonious as possible. By this we mean that α does not mention extraneous conditions. A simple case of the kind of situation we want to avoid is when α is unnecessarily *strong*. In the above example, a formula like

   $$(\mathsf{TennisElbow} \wedge \mathsf{ChickenPox})$$

   satisfies the first two criteria: it implies the symptom and is consistent with the KB. But the part about chicken pox is unnecessary. Similarly (but less obviously), the α can be unnecessarily *weak*. If $\neg\mathsf{Vegetarian}$ were a fact in the above KB, then a formula like

   $$(\mathsf{TennisElbow} \vee \mathsf{Vegetarian})$$

would still satisfy the first two criteria, although the vegetarian part is unnecessary. In general, we want α to use as few terms as possible. In the propositional case, this means as few literals as possible.

4. α is in the appropriate vocabulary. Note, for example, that according to the first three criteria above, SoreElbow is a formula that explains SoreElbow. We might call this the *trivial* explanation. It is also the case that SoreJoints satisfies the first three criteria. For various applications, this may or may not be suitable. Intuitively, however, in this case, since we think of SoreJoints in this KB as being almost just another name for the conjunction of SoreElbow and SoreHips, it would not really be a good explanation. Usually, we have in mind a set $\mathcal{H}$ of possible hypotheses (a set of atomic sentences) in terms of which explanations are to be phrased. In the case of medical diagnoses, for instance, these would be diseases or conditions like ChickenPox or TennisElbow. In that case, SoreJoints would not be a suitable explanation.

We call an α that satisfies the four conditions above an abductive *explanation* of β with respect to KB.

13.2.1 Some simplifications

With this definition of an explanation in hand, we will see that in the propositional case at least, certain simplifications to the task of generating explanations are possible.

First of all, while we have considered explaining an arbitrary formula β, it is sufficient to know how to explain a single literal, or even just an atom. The reason for this is that we can choose a new atom p that appears nowhere else, and get that α is an explanation for β with respect to KB if and only if α is an explanation for p with respect to (KB $\cup \{(p \equiv \beta\}$), as can be verified by considering the definition of explanation above. In other words, according to the criteria in the above definition, anything that is an explanation for p would also be considered an explanation for β, and vice-versa.

Next, while we have considered explanations that could be any sort of formula, it is sufficient to limit our attention to conjunctions of literals. To see why, imagine that some arbitrary formula α is an explanation for β, and assume that when α is converted into DNF, we get $(d_1 \vee \cdots \vee d_n)$, where each d_i is a conjunction of literals. Observe that each d_i entails β, and uses terms of the appropriate vocabulary.

Moreover, at least one of the d_i must be consistent with the KB (since otherwise α would not be). This d_i is also as simple or simpler than α itself. So this single d_i by itself can be used instead of α as an explanation for β.

Because a conjunction of literals is logically equivalent to the negation of a clause, it then follows that to explain a literal ρ, it is sufficient to look for a clause c (in the desired vocabulary) with as few literals as possible that satisfies the following constraints:

1. KB $\models (\neg c \supset \rho)$, or equivalently, KB $\models (c \cup \{\rho\})$, and
2. KB $\not\models c$.

This brings us to the topic of prime implicates.

13.2.2 Prime implicates

A clause c is said to be a *prime implicate* of a KB if and only if

1. KB $\models c$, and
2. for every $c' \subset c$, it is the case that KB $\not\models c'$.

Note that for any clause c, if KB $\models c$, then some subset of c or perhaps c itself must be a prime implicate of KB. For example, if we have a KB consisting of

$$\{(p \wedge q \wedge r \supset g), (\neg p \wedge q \supset g), (\neg q \wedge r \supset g)\}$$

then among the prime implicates are $(p \vee \neg q \vee g)$ and $(\neg r \vee g)$. Each of these clauses is entailed by KB, and no subset of either of them is entailed. In this KB, the tautologies $(p \vee \neg p)$ $(q \vee \neg q)$, $(r \vee \neg r)$, *etc.*, are also prime implicates. In general, note that for any atom ρ, unless KB $\models \rho$ or KB $\models \neg\rho$, the tautology $(\rho \vee \neg\rho)$ will be a prime implicate.

Returning now to explanations for a literal ρ, as we said, we want to find minimal clauses c' such that KB $\models (c' \cup \{\rho\})$ but KB $\not\models c'$. Therefore, it will be sufficient to find prime implicates c containing ρ, in which case, the negation of $(c - \rho)$ will be an explanation for ρ. For the example KB above, if we want to generate the explanations for g, we first generate the prime implicates of KB containing g, which are $(p \vee \neg q \vee g)$, $(\neg r \vee g)$, and $(g \vee \neg g)$, and then we remove the atom g and negate the clauses to obtain three explanations (as conjunctions of literals): $(\neg p \wedge q)$, r, and g itself. Note that tautologous prime implicates will always generate trivial explanations.

13.2.3 Computing explanations

From the above we can derive a procedure to compute explanations for any literal ρ in some vocabulary $\mathcal{H}$:

1. calculate the set of prime implicates of the KB that contain the literal ρ;
2. remove ρ from each of the clauses;
3. return as explanations the negations of the resulting clauses, provided that the literals are in the language $\mathcal{H}$.

The only thing left to consider is how to generate prime implicates.

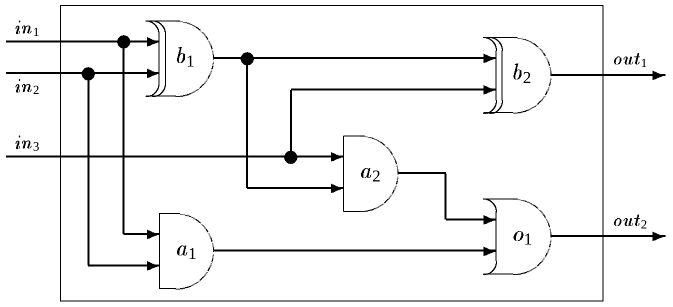
As it turns out, Resolution can be used directly for this: it can be shown that in the propositional case, Resolution is complete for non-tautologous prime implicates. In other words, if KB is a set of clauses, and if KB $\models c$ where c is a non-tautologous prime implicate, then KB $\vdash c$. The completeness of Resolution for the empty clause, used in the Resolution chapter, is just a special case: the empty clause, if entailed, must be a prime implicate. So we can compute all prime implicates of KB containing ρ by running Resolution to completion, generating *all* resolvents, and then keeping only the minimal ones containing ρ. If we want to generate trivial explanations as well, we then need to add the tautologous prime implicates to this set.

This way of handling explanations suggests that it might be a good idea to pre-compute all prime implicates of a KB using Resolution, then to generate explanations for a literal by consulting this set as needed. Unfortunately, this will not work in practice. Even for a KB that is a set of Horn clauses, there can be *exponentially* many prime implicates. For example, consider the following Horn KB over the atoms p_i, q_i, E_i, O_i for $0 \leq i < n$, and E_n and O_n. This example is a version of parity checking; p_i means bit i is on, q_i means off, E_i means the count up to level i is even, O_i means odd:

$$\begin{array}{l}
E_i \wedge p_i \supset O_{i+1} \\
E_i \wedge q_i \supset E_{i+1} \\
O_i \wedge p_i \supset E_{i+1} \\
O_i \wedge q_i \supset O_{i+1} \\
E_0 \\
\neg O_0
\end{array}$$

This KB contains $4n+2$ Horn clauses of size 3 or less. Nonetheless there are 2^{n-1} prime implicates that contain E_n: any clause of the form $[x_0, \ldots x_{n-1}, E_n]$ where x_i is either p_i or q_i and an even number of them are p's will be a prime implicate.

Figure 13.1: A circuit for a full adder



13.3 A circuit example

In this section, we apply the above ideas to a circuit diagnosis problem. Overall, the problem is to determine which component (or components) of a Boolean circuit might have failed given certain inputs and outputs, and a background KB specifying the structure of the circuit, the normal behaviour of logic gates, and perhaps a fault model.

The circuit in question is the full adder shown in Figure 13.1. A full adder takes three bits as input—two addends and a carry bit from a previous adder—and produces two outputs—the sum and the next carry bit. The facts we would expect to have in a KB capturing this circuit are as follows:

- Components, using gate predicates:

  $\forall x.\mathsf{Gate}(x) \equiv \mathsf{AndGate}(x) \vee \mathsf{OrGate}(x) \vee \mathsf{XorGate}(x)$;
  AndGate(a_1), AndGate(a_2),
  XorGate(b_1), XorGate(b_2),
  OrGate(o_1);
  the whole circuit: FullAdder(f).

- Connectivity, using functions in$_i$ for input i, and out$_i$ for output i (where inputs and outputs are numbered from the top down in the diagram):

$\text{in}_1(b_1) = \text{in}_1(f),\ \text{in}_2(b_1) = \text{in}_2(f),$
$\text{in}_1(b_2) = \text{out}(b_1),\ \text{in}_2(b_2) = \text{in}_3(f),$
$\text{in}_1(a_1) = \text{in}_1(f),\ \text{in}_2(a_1) = \text{in}_2(f),$
$\text{in}_1(a_2) = \text{in}_3(f),\ \text{in}_2(a_2) = \text{out}(b_1),$
$\text{in}_1(o_1) = \text{out}(a_2),\ \text{in}_2(o_1) = \text{out}(a_1),$
$\text{out}_1(f) = \text{out}(b_2),\ \text{out}_2(f) = \text{out}(o_1).$

- Truth tables in terms of functions and, or and xor:

  $\text{and}(0,0) = 0$, $\text{and}(0,1) = 0$, *etc.*
  $\text{or}(0,0) = 0$, $\text{or}(0,1) = 1$, *etc.*
  $\text{xor}(0,0) = 0$, $\text{xor}(0,1) = 1$, *etc.*

- The normal behavior of logic gates, using a predicate Ab:[3]

  $\forall x.\text{AndGate}(x) \wedge \neg Ab(x) \supset \text{out}(x) = \text{and}(\text{in}_1(x), \text{in}_2(x)),$
  $\forall x.\text{OrGate}(x) \wedge \neg Ab(x) \supset \text{out}(x) = \text{or}(\text{in}_1(x), \text{in}_2(x)),$
  $\forall x.\text{XorGate}(x) \wedge \neg Ab(x) \supset \text{out}(x) = \text{xor}(\text{in}_1(x), \text{in}_2(x)).$

- Finally, we may or may not wish to include some specification of possible abnormal behaviors of the circuit. This is what is usually called a *fault model*. For example, we might have the following specification:

  short circuit:
  $\forall x.[\text{OrGate}(x) \vee \text{XorGate}(x)] \wedge Ab(x) \supset \text{out}(x) = \text{in}_2(x)$

In this example, nothing is specified regarding the behavior of abnormal and-gates. Of course by leaving out parts of a fault model like this, or by making it too weak, we run the risk that certain abnormal behaviors may be inexplicable, as we will discuss further below. Note also that abnormal behavior can be compatible with normal behavior on certain inputs (the output is the same whether or not the gate is working).

13.3.1 The diagnosis

The abductive diagnosis task is as follows: given a KB as above, and some input *settings* of the circuit, for example,

$\text{in}_1(f) = 1,\ \text{in}_2(f) = 0,\ \text{in}_3(f) = 1,$

[3]Although this predicate was used for minimal entailment in Chapter 12, no default reasoning will be used here.

explain some output *observations* of the circuit, for example,

$\text{out}_1(f) = 1,\ \text{out}_2(f) = 0,$

in the language of Ab. What we are looking for, in other words, is a minimal conjunction α of ground $Ab(c)$ and $\neg Ab(c)$ terms such that

$$\text{KB} \cup \textit{Settings} \cup \{\alpha\} \models \textit{Observations}.$$

To do this computation, we can use the techniques described above, although we first have to "propositionalize" by observing, for example, that the universally quantified x in the above need only range over the five given gates.

To do this by hand, the easiest way is to make a table of all 2^5 possibilities regarding which gates are normal or abnormal, seeing which of them entail the observations, and then looking for commonalities (and thus simplest possible explanations). In Figure 13.2, in each row of the table, the sixth column says whether or not the conjunction of Ab literals (either positive or negative) together with the KB and the input settings entails the output observations. (Ignore the seventh column for now.) For example, in row 5, we see that

$$Ab(b_1) \wedge Ab(b_2) \wedge \neg Ab(a_1) \wedge Ab(a_2) \wedge Ab(o_1)$$

entails the outputs; however, it is not an explanation since

$$Ab(b_1) \wedge \neg Ab(a_1) \wedge Ab(o_1)$$

also entails the outputs (as can be verified by examining the 4 rows of the table with these values) and is simpler. Moreover, no subset of these literals entails the outputs. Continuing in this way, we end up with 3 abductive explanations:

1. $Ab(b_1) \wedge \neg Ab(a_1) \wedge Ab(o_1)$,
   gates b_1 and o_1 are defective, but a_1 is working;

2. $Ab(b_1) \wedge \neg Ab(a_1) \wedge \neg Ab(a_2)$,
   gate b_1 is defective, but a_1 and a_2 are working;

3. $Ab(b_2) \wedge \neg Ab(a_1) \wedge Ab(o_1)$;,
   gates b_2 and o_1 are defective, but a_1 is working.

Observe that not all components are mentioned in these explanations. This is because, given the settings and the fault model, we would get the same results whether or not the components were working normally. Different settings (or different fault

Figure 13.2: Diagnosis of the full adder

b_1	b_2	a_1	a_2	o_1	entailed?	consistent?
$Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	no	yes
$Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	no	yes
$Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	yes	yes
$Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	yes	yes
$Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	yes	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	no	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	no	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	yes	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	yes	yes
$Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	yes	yes
$\neg Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	no	yes
$\neg Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$\neg Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	no	yes
$\neg Ab(b_1)$	$Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	no	no
$\neg Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	yes	yes
$\neg Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	yes
$\neg Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	yes	yes
$\neg Ab(b_1)$	$Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$Ab(a_2)$	$\neg Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$Ab(o_1)$	no	no
$\neg Ab(b_1)$	$\neg Ab(b_2)$	$\neg Ab(a_1)$	$\neg Ab(a_2)$	$\neg Ab(o_1)$	no	no

models) could lead to different diagnoses. In fact, a key principle in this area is what is called *differential diagnosis*, that is, trying to discover tests that would distinguish between competing explanations. In the case of the circuit, this amounts to trying to find different input settings that would provide different outputs depending

on what is or is not working normally. One principle of good engineering design is to make a circuit *testable*, that is, configured in such a way as to facilitate testing its (usually inaccessible) internal components.

13.3.2 Consistency-based diagnosis

One problem with the abductive form of diagnosis presented above is that it relies crucially on the presence of a fault model. Without a specification of how a circuit would behave when it is not working, certain output observations can be inexplicable, and this form of diagnosis can be much less helpful.

In many cases, however, we know how a circuit is supposed to work, but may not be able to characterize its failure modes. We would like to find out which components could be at fault when output observations conflict with this specification. Of course, with no fault model at all, we would be free to conjecture that *all* components were at fault. What we are really after, then, is a *minimal* diagnosis, that is, one that does not assume any unnecessary faults.[4]

This second version of diagnosis can be made precise as follows:

> Assume KB uses the predicate Ab as before. (The KB may or may not include a fault model.) We want to find a set of components D such that the set
> $$\{Ab(c) | c \in D\} \cup \{\neg Ab(c) | c \notin D\}$$
> is *consistent* with the set
> $$\text{KB} \cup \textit{Settings} \cup \textit{Observations}$$
> and no proper subset of D is. Any such D is called a *consistency-based diagnosis* of the circuit.

So for consistency-based diagnosis, we look for (minimal sets of) assumptions of abnormality that are consistent with the settings and observations, rather than (minimal sets of) assumptions of normality and abnormality that entail the observations.

In the case of the circuit example above (with the given fault model), we can look for the diagnoses by hand by again making a table of all 2^5 possibilities regarding which gates are normal or abnormal, seeing which of them are consistent with the settings and observations, and then looking for commonalities (and thus minimal sets of faulty components). Returning to the table in Figure 13.2, in each

[4]Note that in the abductive account, we do not necessarily minimize the set of components assumed to be faulty, in that the literals $Ab(c)$ and $\neg Ab(c)$ have equal status.

row of the table, the seventh column says whether or not the conjunction of Ab literals (either positive or negative) is consistent with the KB together with the input settings and the output observations. (Ignore the sixth column this time.) For example, in row 5, we see that

$$\{Ab(b_1), Ab(b_2), \neg Ab(a_1), Ab(a_2), Ab(o_1)\}$$

is consistent with the inputs and outputs. This does not yet give us a diagnosis since

$$\{Ab(b_1), \neg Ab(b_2), \neg Ab(a_1), \neg Ab(a_2), \neg Ab(o_1)\}$$

is also consistent (row 16), and assumes a smaller set of abnormal components.

Continuing in this way, this time we end up with 3 consistency-based diagnoses: $\{b_1\}$, $\{b_2, a_2\}$, and $\{b_2, o_1\}$. Further testing could then be used to narrow down the possibilities.

While it is difficult to compare the two approaches to diagnosis in general terms, it is worth noting that they do behave quite differently regarding fault models. In the abductive case, with less of a fault model, there are usually fewer diagnoses involving abnormal components, since nothing follows regarding their behaviour; in the consistency-based case, the opposite usually happens, since anything can be assumed regarding their behaviour. For example, one of three possibilities considered in the consistency-based account is that both b_2 and a_2 are abnormal, since it is consistent that a_2 is producing a 0, and then that the output of o_1 is 0. In the abductive case, none of the explanations involve a_2 being abnormal, since there would then be no way to confirm that the output of o_1 is 0. In general, however, it is difficult to give hard and fast rules about which type of diagnosis should be used.

13.4 Beyond the basics

We conclude this chapter by examining some complications to the simple picture of abductive reasoning we have presented, and then finally sketching some non-diagnostic applications of abductive reasoning.

13.4.1 Extensions

There are are a number of ways in which our account of abductive reasoning could be enlarged for more realistic applications.

Variables and quantification: In the first-order case of abductive reasoning, we might need to change, at the very least, our definition of what it means for an explanation to be as simple as possible. It might also be useful to consider

explaining *open formulas*, as a way of answering certain types of WH-questions, in a way that goes beyond answer extraction. Imagine we have a query like $P(x)$. We might return the answer $(x = \mathsf{john})$ using answer extraction, since this is one way of explaining how $P(x)$ could be true. But we might also return something like $Q(x)$ as the answer to the question. For example, if we ask the question "what are yellow song birds that serve as pets?" the answer we are expecting is probably not the names of some individual birds, but rather another predicate like "canaries." Note however that it is not clear how to use Resolution to generate explanations in a first-order setting.

Negative evidence: We have insisted that explanations entail everything to be explained. We might, however, imagine cases where missing observations need to be accounted for. For example, we might be interested in a medical diagnosis that does not entail fever, without necessarily requiring that it entail ¬fever.

Defaults: We have used logical entailment as the relation between an explanation α and what is being explained β. In a more general setting, it might be preferable to require that it be reasonable to believe β given α, where this belief could involve default assumptions. For example, being a bird might explain an animal being able to fly, even though it would not entail it.

Probabilities: We have preferred explanations and diagnoses that are as simple as possible. However, in general, not all simplest ones would be expected to be equally likely. For example, we may have two circuit diagnoses, each involving a single component, but it may be that one of them is much more likely to fail than the other. Perhaps the failure of one component makes it very likely that another will fail as well. Moreover, the "causal laws" we have between (say) diseases and symptoms would typically have a probabilistic component: only a certain percentage of the time would we expect a disease to show a symptom.

13.4.2 Other applications

Finally, let us consider other applications of abductive reasoning.

Object recognition: This is an application where a system is given input from a camera, say, and must determine what is being viewed. At one level, the question is this: what scene would explain the image elements being observed? Abduction is required here since, as with diseases and symptoms, it is presumed to be easier to obtain facts that tell us what would be visible if an object were present, than to obtain facts that tell us what object is present if certain patterns are visible. At a higher level, once certain properties of the object have been determined, another question to consider is this: what object(s) would explain the collection of properties discovered? Both of these tasks can be nicely formulated in abductive terms.

Plan recognition: In this case, the observations are the actions of an agent, and the explanation we seek is one that relates to the high-level goals of the agent. If we observe the agent boiling water, and heating a tomato sauce, we might abduce that a pasta dish is being prepared.

Hypothetical reasoning: As a final application, consider the following. Instead of asking "what would I have to be told to believe that β is true?" as in abductive reasoning, we ask "what would I learn if I were told that α were true?" For example, we might be looking for new symptoms that would be entailed if a disease were present. This is clearly a form of deductive reasoning, but one where we are interested in returning a formula, rather than a yes/no answer or the names of some individuals. In a sense, it is the dual of explanation: we are looking for a formula β that is entailed by α together with the KB, but one that is not already entailed by the KB itself, that is simple and parsimonious, and that is in the correct vocabulary.

Interestingly, there is a precise connection between this form of reasoning and the type of explanation we have already defined: we should learn β on being told α in the above sense if and only if the formula $\neg\beta$ is an abductive explanation for $\neg\alpha$ as already defined. For instance, to go back to the tennis example at the start of the chapter, one of new things we ought to learn on being told

$$(\mathsf{Arthritis} \land \neg\mathsf{SoreElbow})$$

would be Treated (that is, the arthritis is being treated). If we now go back to the definition of explanation, we can verify that ¬Treated is indeed an abductive explanation for

$$\neg(\mathsf{Arthritis} \land \neg\mathsf{SoreElbow}),$$

since ¬Treated entails this sentence, is consistent with the KB, and is as simple as possible. The nice thing about this account is that an existing procedure for abductive reasoning could be used directly for this type of deductive reasoning.

13.5 Bibliographic notes

13.6 Exercises

Chapter 14

Actions

The language of FOL is sometimes criticized as being an overly "static" representation formalism. Sentences of FOL are either true or false in an interpretation and stay that way. Unlike procedural representations or production systems, there is seemingly nothing in FOL corresponding to any sort of *change*.

In fact, there are two sorts of changes that we might want to consider. First, there is the idea of changing what is believed about the world. Suppose α is a sentence saying that birds are the descendants of dinosaurs. At some point, you might come to believe that α is true, perhaps by being told directly. If you had no beliefs about α before, this is a straightforward process that involves adding α to your current KB. If you had previously thought that α was false, however, perhaps having concluded this from a number of other beliefs, dealing with the new information is a much more complicated process. The study of which of your old beliefs to discard is an important area of research known as *belief revision*, but one that is beyond the scope of this book.

The second notion of change to consider is when the beliefs themselves are about a changing world. Instead of merely believing that John is a student, for example, you might believe that John was not a student initially, but that he became a student by enrolling at a university, and that he later graduated, and ceased to be a student. In this case, while the world you are imagining is certainly changing, the beliefs you have about John's history as a whole need not change at all.[1]

In this chapter, we will study how beliefs about a changing world of this sort can in fact be represented in a dialect of FOL called the *situation calculus*. This is not the only way to represent a changing world, of course, but it is a simple and

[1]Of course, we might also have changing beliefs about a changing world, but we will not pursue this here.

powerful way to do so. It also naturally lends itself to various sorts of reasoning, including planning, discussed separately in the next chapter.

14.1 The situation calculus

One way of thinking about change is to imagine being in a certain situation, with actions moving you from one situation to the next. The situation calculus is a dialect of FOL in which such situations and actions are taken to be objects in the domain. In particular, there are two distinguished sorts of first-order terms:

- *actions*: such as jump (the act of jumping), kick(x) (kicking object x), and put(r, x, y) (robot r putting object x on top of object y). The constant and function symbols for actions are completely application-dependent.
- *situations*, which denote possible world histories. A distinguished constant S_0 and function symbol *do* are used. S_0 denotes the initial situation, before any action has been performed; *do*(a, s) denotes the situation that results from performing action a in situation s.

For example, the situation term *do*(pickup(b_2), *do*(pickup(b_1), S_0)) denotes the situation that results from first picking up object b_1 in S_0 and then picking up object b_2. Note that this situation is not the same as *do*(pickup(b_1), *do*(pickup(b_2), S_0)), since they have different histories, even though the resulting states may be indistinguishable.

14.1.1 Fluents

Predicates and functions whose values may vary from situation to situation are called *fluents*, and are used to describe what holds in a situation. By convention, the last argument of a fluent is a situation. For example, the fluent Holding(r, x, s) might stand for the relation of robot r holding object x in situation s. Thus, we can have formulas like

$$\neg\mathsf{Holding}(r,x,s) \land \mathsf{Holding}(r,x,\mathit{do}(\mathsf{pickup}(r,x),s))$$

which says that robot r is not holding x in some situation s, but is holding x in the situation that results from picking it up. Note that in the situation calculus there is no distinguished "current" situation. A single formula like this can talk about many different situations, past, present, or future.

Finally, a distinguished predicate *Poss*(a, s) is used to state that action a can be performed in situation s. For example,

$$\mathit{Poss}(\mathsf{pickup}(r,x),S_0)$$

says that the robot r is able to pick up object x in the initial situation.

This completes the specification of the dialect.

14.1.2 Precondition and effect axioms

To reason about a changing world, it is necessary to have beliefs not only about what is true initially, but also about how the world changes as the result of actions.

Actions typically have *preconditions*, that is, conditions that need to be true for the action to occur. For example, in a robotics setting, we might have the following:

- a robot can pick up an object if and only if it is not holding anything, the object is not too heavy, and the robot is next to the object:[2]

$$\mathit{Poss}(\mathsf{pickup}(r,x),s) \equiv \forall z.\neg\mathsf{Holding}(r,z,s) \land \neg\mathsf{Heavy}(x) \land \mathsf{NextTo}(r,x,s);$$

- it is possible for a robot to repair an object if and only if the object is broken and there is glue available:

$$\mathit{Poss}(\mathsf{repair}(r,x),s) \equiv \mathsf{Broken}(x,s) \land \mathsf{HasGlue}(r,s).$$

Actions typically also have *effects*, that is, fluents that are changed as a result of performing the action. For example,

- dropping a fragile object causes it to break:

$$\mathsf{Fragile}(x) \supset \mathsf{Broken}(x,\mathit{do}(\mathsf{drop}(r,x),s));$$

- repairing an object causes it to be unbroken:

$$\neg\mathsf{Broken}(x,\mathit{do}(\mathsf{repair}(r,x),s)).$$

Formulas like those above are often called *precondition axioms* and *effect axioms* respectively.[3] Effect axioms are called *positive* if they describe when a fluent becomes true, and *negative* otherwise.

[2]In this chapter, free variables should be assumed to be universally quantified from the outside.

[3]These are called "axioms" for historical reasons: a KB can be thought of as the axioms of a logical theory (like number theory or set theory), with the entailed beliefs considered as theorems.

14.1.3 Frame axioms

To fully capture the dynamics of a situation, we need to go beyond the preconditions and effects of actions. So far, if a fluent is not mentioned in an effect axiom for an action a, we would not know anything at all about it in the situation $\mathit{do}(a, s)$. To really know how the world can change, it is also necessary to know what fluents are *unaffected* by performing an action. For example,

- dropping an object does not change its colour:

  $$\mathsf{Colour}(x, c, s) \supset \mathsf{Colour}(x, c, \mathit{do}(\mathsf{drop}(r, x), s));$$

- dropping an object y does not break an object x when $x \neq y$ or x is not fragile:

  $$\neg\mathsf{Broken}(x, s) \land [x \neq y \lor \neg\mathsf{Fragile}(x)] \supset \neg\mathsf{Broken}(x, \mathit{do}(\mathsf{drop}(r, y), s)).$$

Formulas like these are often called *frame axioms*. Observe that we would not normally expect them to be entailed by the precondition or effect axioms for the actions involved.

Frame axioms do present a serious problem, however, sometimes called the *frame problem*. Simply put, the problem is that it will be necessary to know and reason effectively with an extremely large number of frame axioms. Indeed, for any given fluent, we would expect that only a very small number of actions affect the value of that fluent; the rest leave it invariant. For instance, an object's colour is unaffected by picking things up, opening a door, using the phone, making linguini, walking the dog, electing a new Prime Minister of Canada *etc. etc.* All of these will require frame axioms. It seems very counterintuitive that we should need to even think about these $\approx 2 \times \mathcal{A} \times \mathcal{F}$ facts (where $\mathcal{A}$ is the number of actions, and $\mathcal{F}$, the number of fluents) about what does not change when we perform an action.

What counts as a solution to this problem? Suppose the person responsible for building a KB has written down *all* the relevant effect axioms. That is, for each fluent $F(\vec{x}, s)$ and action a that can cause the fluent to change, we have an effect axiom of the form

$$\phi(\vec{x}, s) \supset (\neg)F(\vec{x}, \mathit{do}(a, s)),$$

where $\phi(\vec{x}, s)$ is some condition on situation s. What we would like is a systematic procedure for generating all the frame axioms from these effect axioms. Moreover, if possible, we also want a parsimonious representation for them, since in their simplest form, there are too many.

And why do we want such a solution? There are at least three reasons:

- Frame axioms are necessary beliefs about a dynamic world that are not entailed by other beliefs we may have.
- For the convenience of the KB builder: generating the frame axioms automatically gives us modularity, since only the effect axioms need to be given by hand. This ensures there is no inadvertent omission or error.
- Such a solution is useful for theorizing about actions: we can see what assumptions need to be made to draw conclusions about what does not change.

We will examine a simple solution to the frame problem in Section 14.2.

14.1.4 Using the situation calculus

Given a KB containing facts expressed in the situation calculus as above, there are various sorts of reasoning tasks we can consider. We will see in the next chapter that we can do planning. In Section 14.3, we will see that we can figure out how to execute a high-level action specification. Here we consider two basic reasoning tasks: projection and legality testing.

The *projection task* is the following: given a sequence of actions and some initial situation, determine what would be true if those actions were performed starting in that initial situation. This can be formalized as follows:

> Suppose that $\phi(s)$ is a formula with a single free variable s of the situation sort, and that $\vec{a}$ is a sequence of actions $\langle a_1, \ldots, a_n\rangle$. To find out if $\phi(s)$ would be true after performing $\vec{a}$ starting in the initial situation S_0, we determine whether or not KB $\models \phi(\mathit{do}(\vec{a}, S_0))$, where $\mathit{do}(\vec{a}, S_0)$ is an abbreviation for $\mathit{do}(a_n, \mathit{do}(a_{n-1}, \ldots, \mathit{do}(a_2, \mathit{do}(a_1, S_0)) \ldots))$.

For example, using the above effect and frame axioms, it follows that the fluent $\neg\mathsf{Broken}(b_2, s)$ would hold after the sequence of actions

$$\langle \mathsf{pickup}(b_1), \mathsf{pickup}(b_2), \mathsf{drop}(b_2), \mathsf{repair}(b_2), \mathsf{drop}(b_1)\rangle.$$

In other words, the fluent holds in the situation

$$s = \mathit{do}(\mathsf{drop}(b_1), \mathit{do}(\mathsf{repair}(b_2), \mathit{do}(\mathsf{drop}(b_2), \mathit{do}(\mathsf{pickup}(b_2), \mathit{do}(\mathsf{pickup}(b_1), S_0))))).$$

It is a separate matter to determine whether or not the given sequence of actions could in fact be performed starting in the initial situation. This is called the *legality testing task*. For example, a robot might not be able to pick up more than one object at a time. We call a situation term *legal* if it is either the initial situation, or the result of performing an action whose preconditions are satisfied starting in a legal situation. For example, although the term

$$do(\mathsf{pickup}(b_2), do(\mathsf{pickup}(b_1), S_0))$$

is well formed, it is not a legal situation, since the precondition for picking up b_2 (*e.g* not holding anything) will not be satisfied in a situation where b_1 has already been picked up. So the legality task is determining whether a sequence of actions leads to a legal situation. This can be formalized as follows:

> Suppose that $\vec{a}$ is a sequence of actions $\langle a_1, \ldots, a_n \rangle$. To find out if $\vec{a}$ can be legally performed starting in the initial situation S_0, we determine whether or not KB $\models$ *Poss*$(a_i, do(\langle a_1, \ldots, a_{i-1}\rangle, S_0))$ for every i such that $1 \leq i \leq n$.

Before concluding this section on the situation calculus, it is perhaps worth noting some of the representational limitations of this language:

- *single agent*: there are no unknown or unobserved exogenous actions performed by other agents, and no unnamed events;
- *no time*: we have not talked about how long an action takes, or when it occurs;
- *no concurrency*: if a situation is the result of performing two actions, one of them is performed first and the other afterwards;
- *discrete actions*: there are no continuous actions like pushing an object from one point to another, or a bathtub filling with water;
- *only hypotheticals*: we cannot say that an action *has* occurred in reality, or *will* occur;
- *only primitive actions*: there are no actions that are constructed from other actions as parts, such as iterations or conditionals.

Many of these limitations can be dealt with by refinements and extensions to the dialect of the situation calculus considered here. We will deal with the last of these in Section 14.3 below.

But first we turn to a solution to the frame problem.

14.2 A simple solution to the frame problem

The solution to the frame problem we will consider depends on first putting all effect axioms into a normal form.

Suppose, for example, that there are two positive effect axioms for the fluent Broken:

$$\mathsf{Fragile}(x) \supset \mathsf{Broken}(x, do(\mathsf{drop}(r, x), s))$$
$$\mathsf{NextTo}(b, x, s) \supset \mathsf{Broken}(x, do(\mathsf{explode}(b), s)).$$

So an object is broken if it is fragile and it was dropped, or something next to it exploded. Using a universally quantified action variable a, these can be rewritten as a single formula

$$\exists r\{a = \mathsf{drop}(r, x) \wedge \mathsf{Fragile}(x)\} \vee$$
$$\exists b\{a = \mathsf{explode}(b) \wedge \mathsf{NextTo}(b, x, s)\} \supset$$
$$\mathsf{Broken}(x, do(a, s))$$

Similarly, a negative effect axiom like

$$\neg\mathsf{Broken}(x, do(\mathsf{repair}(r, x), s)),$$

saying that an object is not broken after it is repaired, can be rewritten as

$$\exists r\{a = \mathsf{repair}(r, x)\} \supset \neg\mathsf{Broken}(x, do(a, s)).$$

In general, for any fluent $F(\vec{x}, s)$, we can rewrite all of the positive effect axioms as a single formula of the form

$$\Pi_F(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)), \qquad (1)$$

and all the negative effect axioms as a single formula of the form

$$\mathrm{N}_F(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s)), \qquad (2)$$

where $\Pi_F(\vec{x}, a, s)$ and $\mathrm{N}_F(\vec{x}, a, s)$ are formulas whose free variables are among the x_i, a, and s.

14.2.1 Explanation closure

Now imagine that we make a completeness assumption about the effect axioms we have for a fluent: assume that formulas (1) and (2) above characterize *all* the conditions under which an action a changes the value of fluent F. We can in fact formalize this assumption using what are called *explanation closure axioms* as follows:

$\neg F(\vec{x}, s) \wedge F(\vec{x}, do(a, s)) \supset \Pi_F(\vec{x}, a, s)$ (3)
if F were false, and made true by doing action a, then condition Π_F must have been true;

$F(\vec{x}, s) \wedge \neg F(\vec{x}, do(a, s)) \supset \mathrm{N}_F(\vec{x}, a, s)$ (4)
if F were true, and made false by doing action a, then condition N_F must have been true.

Informally, these axioms add an "only if" component to the normal form effect axioms: (1) says that F is made true if Π_F holds, while (3) says that F is made true only if Π_F holds.[4] In fact, by rewriting them slightly, these explanation closure axioms can be seen to be disguised versions of frame axioms:

$$\neg F(\vec{x}, s) \wedge \neg\Pi_F(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$$
$$F(\vec{x}, s) \wedge \neg\mathrm{N}_F(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)).$$

In other words, F remains false after doing a when Π_F is false, and F remains true after doing a when N_F is false.

14.2.2 Successor state axioms

If we are willing to make two assumptions about our KB, the formulas (1), (2), (3), and (4) can be combined in a particularly simple and elegant way. Specifically, we assume that our KB entails the following:

- integrity of the effect axioms for every fluent F:

$$\neg\exists \vec{x}, a, s.\ \Pi_F(\vec{x}, a, s) \wedge \mathrm{N}_F(\vec{x}, a, s)$$

- unique names for actions:

$$A(\vec{x}) = A(\vec{y}) \supset (x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$$
$A(\vec{x}) \neq B(\vec{y})$, where A and B are distinct action names

The first assumption is merely that no action a satisfies the condition to make the fluent F both true and false. The second assumption is that the only action terms that can be equal are two identical actions with identical arguments.

With these two assumptions, it can be shown that for any fluent F, KB entails that (1), (2), (3), and (4) together are logically equivalent to the following formula:

$$F(\vec{x}, do(a, s)) \equiv \Pi_F(\vec{x}, a, s) \vee (F(\vec{x}, s) \wedge \neg\mathrm{N}_F(\vec{x}, a, s)).$$

[4]Note that in (3) we need to ensure that F was originally false and was made true to be able to conclude that Π_F held, and similarly for (4).

A formula of this form is called a *successor state axiom* for the fluent F because it completely characterizes the value of fluent F in the successor state resulting from performing action a in situation s. Specifically, F is true after doing a if and only if before doing a, Π_F (the positive effect condition for F) was true or both F and $\neg\mathrm{N}_F$ (the negative effect condition for F) were true. For example, for the fluent Broken, we have the following successor state axiom:

$$\begin{aligned}&\mathsf{Broken}(x, do(a, s)) \equiv \\ &\quad \exists r\{a = \mathsf{drop}(r, x) \wedge \mathsf{Fragile}(x)\} \vee \\ &\quad \exists b\{a = \mathsf{explode}(b) \wedge \mathsf{NextTo}(b, x, s)\} \vee \\ &\quad \mathsf{Broken}(x, s) \wedge \forall r\{a \neq \mathsf{repair}(r, x)\}\end{aligned}$$

This says that an object x is broken after doing action a if and only if a is a dropping action and x is fragile, or a is a bomb exploding action when x is near to the bomb, or x was already broken and a is not the action of repairing it.

Note that it follows from this axiom that dropping a fragile object will break it. Moreover, it also follows logically that talking on the phone does not affect whether or not an object is broken (assuming unique names, *i.e.* talking on the phone is distinct from any dropping, exploding, or repairing action). Thus a KB containing this single axiom would entail all the necessary effect and frame axioms for the fluent in question.

14.2.3 Summary

We have, therefore, a simple solution to the frame problem in terms of the following axioms:

- successor state axioms, one per fluent,
- precondition axioms, one per action,
- unique name axioms for actions.

Observe that we do not get a small number of axioms at the expense of prohibitively long ones. The length of a successor state axiom is roughly proportional to the number of actions that affect the value of the fluent, and, as we noted earlier, we do not expect in general that very many of the actions would change the value of any given fluent.

The conciseness and perspicuity of this solution to the frame problem clearly depends on three factors:

1. the ability to quantify over actions, so that only actions changing the fluent need to be mentioned by name;
2. the assumption that relatively few actions affect each fluent, which keeps the successor state axioms short;
3. the completeness assumption for the effects of actions, which allows us to conclude that actions that are not mentioned explicitly in effect axioms leave the fluent invariant.

The solution also depends on being able to put effect axioms in the normal form used above. This would not be possible, for example, if we had actions whose effects were *nondeterministic*. For example, imagine an action flipcoin whose effect is to make either the fluent Heads or the fluent Tails true. An effect axiom like

$$\mathsf{Heads}(do(\mathsf{flipcoin}, s)) \lor \mathsf{Tails}(do(\mathsf{flipcoin}, s))$$

cannot be put into the required normal form. In general, we need to assume that every action a is deterministic in the sense that all the given effect axioms are of the form

$$\phi(\vec{x}, s) \supset (\neg)F(\vec{x}, do(a, s)).$$

How to deal in some way with nondeterministic choice and other complex actions is the topic of the next section.

14.3 Complex actions

So far, in our treatment of the situation calculus, we have assumed that there are only primitive actions, with effects and preconditions independent of each other. We have no way of handling *complex actions*, that is to say, actions that have other actions as components. Examples of these are actions like the following:

- *conditionals*: if the car is in the driveway then drive and otherwise walk;
- *iterations*: while there are blocks on the table, remove one;
- *nondeterministic choice*: pick a red block up off the table and put it on the floor;

and others, as described below. What we would like to do is to *define* such actions in terms of their primitive components in such a way that we can inherit their solution to the frame problem. To do this, we need a compositional treatment of the frame problem for complex actions. This is precisely what we will provide, and we will see that it results in a novel kind of programming language.

14.3.1 The Do formula

To handle complex actions in general, it is sufficient to show that for each complex action A we care about, there is a formula of the situation calculus, which we call $\boldsymbol{Do}(A, s, s')$, that says that action A when started in situation s can terminate legally in situation s'. Because complex actions can be nondeterministic, there may be more than one such s'. Consider, for example, the complex action

$$[\mathsf{pickup}(b_1)\ ;\ \textit{if}\ \mathsf{InRoom}(\mathsf{kitchen})\ \textit{then}\ \mathsf{putaway}(b_1)\ \textit{else}\ \mathsf{goto}(\mathsf{kitchen})].$$

For this action to start in situation s and terminate legally in s', the following sentence must be true:

$$\begin{array}{l} Poss(\mathsf{pickup}(b_1), s) \ \land \\ \quad [\ (\mathsf{InRoom}(\mathsf{kitchen}, do(\mathsf{pickup}(b_1), s)) \\ \qquad \land\ Poss(\mathsf{putaway}(b_1), do(\mathsf{pickup}(b_1), s)) \\ \qquad \land\ s' = do(\mathsf{putaway}(b_1), do(\mathsf{pickup}(b_1), s))) \\ \qquad\qquad \lor \\ \quad\ (\neg\mathsf{InRoom}(\mathsf{kitchen}, do(\mathsf{pickup}(b_1), s)) \\ \qquad \land\ Poss(\mathsf{goto}(\mathsf{kitchen}), do(\mathsf{pickup}(b_1), s)) \\ \qquad \land\ s' = do(\mathsf{goto}(\mathsf{kitchen}), do(\mathsf{pickup}(b_1), s)))\ ] \end{array}$$

In general, we define the formula $\boldsymbol{Do}$ recursively on the structure of the complex action as follows:

1. For any primitive action A, we have

$$\boldsymbol{Do}(A, s, s') \overset{def}{=} Poss(A, s) \land s' = do(A, s).$$

2. For the sequential composition of complex actions A and B, $[A\ ;\ B]$, we have

$$\boldsymbol{Do}([A\ ;\ B], s, s') \overset{def}{=} \exists s''.\ \boldsymbol{Do}(A, s, s'') \land \boldsymbol{Do}(B, s'', s').$$

3. For a conditional involving a test ϕ[5] of the form $[\textit{if}\ \phi\ \textit{then}\ A\ \textit{else}\ B]$, we have

$$\boldsymbol{Do}([\textit{if}\ \phi\ \textit{then}\ A\ \textit{else}\ B], s, s') \overset{def}{=} \\ [\phi(s)\ \land\ \boldsymbol{Do}(A, s, s')]\ \lor\ [\neg\phi(s) \land \boldsymbol{Do}(B, s, s')].$$

[5]If $\phi(s)$ is a formula of the situation calculus with a free variable s, then ϕ is that formula with the situation argument suppressed. For example, in a complex action we would use the test $\mathsf{Broken}(x)$ instead of $\mathsf{Broken}(x, s)$.

4. For a test action, [ϕ?], determining if a condition ϕ currently holds, we have

$$\textbf{\textit{Do}}([\phi?], s, s') \overset{def}{=} \phi(s) \wedge s' = s.$$

5. For a nondeterministic branch to action A or action B, $[A \mid B]$, we have

$$\textbf{\textit{Do}}([A \mid B], s, s') \overset{def}{=} \textbf{\textit{Do}}(A, s, s') \vee \textbf{\textit{Do}}(B, s, s').$$

6. For a nondeterministic choice of a value for variable x, $[\pi x.A]$, we have

$$\textbf{\textit{Do}}([\pi x.A], s, s') \overset{def}{=} \exists x.\textbf{\textit{Do}}(A, s, s').$$

7. For an iteration of the form [*while* ϕ *do* A], we have[6]

$$\textbf{\textit{Do}}([\textit{while } \phi \textit{ do } A], s, s') \overset{def}{=} \forall P\{\cdots \supset P(s, s')\}$$

where the ellipsis is an abbreviation for the conjunction of

$$\forall s_1.\, \neg\phi(s_1) \supset P(s_1, s_1)$$
$$\forall s_1, s_2, s_3.\, \phi(s_1) \wedge \textbf{\textit{Do}}(A, s_1, s_2) \wedge P(s_2, s_3) \supset P(s_1, s_3)$$

Similar rules can be given for recursive procedures, and even constructs involving concurrency and interrupts. The main point is that what it means to perform these complex actions can be fully specified in the language of the situation calculus. What we are giving, in effect, is a purely logical semantics for many of the constructs of traditional programming languages.

14.3.2 GOLOG

What we end up with, then, is a programming language, called GOLOG, that generalizes conventional imperative programming languages.[7] It includes the usual imperative constructs (sequence, iteration, *etc.*), as well as nondeterminism and other features. The main difference, however, is that the primitive statements of GOLOG are not operations on internal states, like assignment statements or pointer updates, but rather primitive actions in the world, such as picking up a block. Moreover, what these primitive actions are supposed to do is not fixed in advance by the language designer, but is specified by the user separately by precondition and successor state axioms.

[6]The rule for iteration involves *second-order quantification*: the P in this formula is a quantified predicate variable. The definition says that an iteration takes you from s to s' iff the smallest relation P satisfying certain conditions does so. The details are not of concern here.

[7]The name comes from "*Algol in logic*," after one of the original and influential programming languages.

Given that the primitive actions are not fixed in advance or executed internally, it is not immediately obvious what it should mean to execute a GOLOG program A. There are two steps:

1. find a sequence of primitive actions $\vec{a}$ such that $\textbf{\textit{Do}}(A, S_0, do(\vec{a}, S_0))$ is entailed by the KB;

2. pass the sequence of actions $\vec{a}$ to a robot or simulator for actual execution in the world.

In other words, to execute a program we must first find a sequence of actions that would take us to a legal terminating situation for the program starting in the initial situation S_0, and then run that sequence.

Note that to find such a sequence, it will be necessary to reason using the given precondition and effect axioms, performing projection and legality testing. For example, suppose we have the program

$$[A \;;\; \textit{if } \mathsf{Holding}(x) \textit{ then } B \textit{ else } C].$$

To decide between B and C, we need to determine whether or not $\mathsf{Holding}(x, s)$ would be true in the situation that results from performing action A.

14.3.3 An example

To see how this would work, consider a simple example in a robotics domain involving three primitive actions, $\mathsf{pickup}(x)$ (picking up a block), $\mathsf{putonfloor}(x)$ (putting a block on the floor), and $\mathsf{putontable}(x)$ (putting a block on the table), and three fluents $\mathsf{Holding}(x, s)$ (the robot is holding a block), $\mathsf{OnFloor}(x, s)$ (a block is on the floor), and $\mathsf{OnTable}(x, s)$ (a block is on the table).

The precondition axioms are the following:

- $\textit{Poss}(\mathsf{pickup}(x), s) \equiv \forall z.\neg\mathsf{Holding}(z, s)$;
- $\textit{Poss}(\mathsf{putonfloor}(x), s) \equiv \mathsf{Holding}(x, s)$;
- $\textit{Poss}(\mathsf{putontable}(x), s) \equiv \mathsf{Holding}(x, s)$.

The successor state axioms are the following:

- $\mathsf{Holding}(x, do(a, s)) \equiv a = \mathsf{pickup}(x) \vee$
  $\mathsf{Holding}(x, s) \wedge a \neq \mathsf{putonfloor}(x) \wedge a \neq \mathsf{putontable}(x)$;
- $\mathsf{OnFloor}(x, do(a, s)) \equiv a = \mathsf{putonfloor}(x) \vee$
  $\mathsf{OnFloor}(x, s) \wedge a \neq \mathsf{pickup}(x)$;
- $\mathsf{OnTable}(x, do(a, s)) \equiv a = \mathsf{putontable}(x) \vee$
  $\mathsf{OnTable}(x, s) \wedge a \neq \mathsf{pickup}(x)$.

We might also have the following facts about the initial situation:

- $\neg\mathsf{Holding}(x, S_0)$;
- $\mathsf{OnTable}(x, S_0) \equiv (x = b_1) \vee (x = b_2)$.

So initially, the robot is not holding anything, and b_1 and b_2 are the only blocks on the table. Finally, we can consider two complex actions, removing a block, and clearing the table:

- *proc* $\mathsf{RemoveBlock}(x)$: $[\mathsf{pickup}(x)\ ;\ \mathsf{putonfloor}(x)]$;
- *proc* ClearTable : *while* $\exists x.\mathsf{OnTable}(x)$ *do*
  $\pi x[\mathsf{OnTable}(x)?\ ;\ \mathsf{RemoveBlock}(x)]$.

This completes the specification of the example.

To execute the GOLOG program ClearTable, it is necessary to first find an appropriate terminating situation, $do(\vec{a}, S_0)$, which determines the actions $\vec{a}$ to perform. To find this situation, we can use Resolution theorem-proving with answer extraction for the query

$$\mathrm{KB} \models \exists s.\ \boldsymbol{Do}(\mathsf{ClearTable}, S_0, s).$$

We omit the details of this derivation, but the result will yield a value for s like

$$s = do(\mathsf{putonfloor}(b_2), do(\mathsf{pickup}(b_2)$$
$$do(\mathsf{putonfloor}(b_1), do(\mathsf{pickup}(b_1), S_0))))$$

from which the desired sequence starting from S_0 is

$$\langle \mathsf{pickup}(b_1), \mathsf{putonfloor}(b_1), \mathsf{pickup}(b_2), \mathsf{putonfloor}(b_2)\rangle.$$

In a more general setting, an answer predicate could be necessary. In fact, in some cases, it may not be possible to obtain a definite sequence of actions. This happens, for example, if what is known about the initial situation is that either block b_1 or block b_2 is on the table.

Observe that if what is known about the initial situation and the actions can be expressed as Horn clauses, the evaluation of GOLOG programs can be done directly in Prolog. Instead of expanding $\boldsymbol{Do}(A, s, s')$ into a long formula of the situation calculus and then using Resolution, we write Prolog clauses such as

```
do(A,S1,S2) :-                          /* for primitive actions */
    prim_action(A), poss(A,S1), S2=do(A,S1).
do(seq(A,B),S1,S2) :-                   /* for sequences */
    do(A,S1,S3), do(B,S3,S2).
do(while(F,A),S1,S2) :-                 /* for while loops (test false) */
    not holds(F,S1), S2=S1.
do(while(F,A),S1,S2) :-                 /* for while loops (test true) */
    holds(F,S1), do(seq(A,while(F,A)),S1,S2).
```

and so on. Then the Prolog goal

```
?- do(clear_table,s0,S).
```

would return the binding for the final situation.

This idea of using Resolution with answer extraction to derive a sequence of actions to perform will be taken up again in the next chapter on planning. When the problem can be reduced to Prolog, we get a convenient and efficient way of generating a sequence of actions. This has proven to be an effective method of providing high-level control for a robot.

14.4 Bibliographic notes

14.5 Exercises

Chapter 15

Planning

15.1 Bibliographic notes

15.2 Exercises

Chapter 16

A Knowledge Representation Tradeoff

16.1 Bibliographic notes

16.2 Exercises

Bibliography

[Brachman et al. (2000)] R. Brachman and H. Levesque, *Knowledge Representation and Reasoning*. in preparation.

Index