5

Representing Defaults

The closed world assumption introduced in the previous chapter is an example of a **default** – a statement of natural language containing words such as "*normally*," "*typically*," or "*as a rule*." Defaults are very useful to humans because, in the absence of complete information, they allow us to draw conclusions based on knowledge of what is common or typical. However, these conclusions are tentative, and we may be forced to withdraw them when new information becomes available. In fact, a large part of our education seems to consist of learning various defaults, their exceptions, and the skill of reasoning with them. Defaults do not occur in the language of mathematics and, therefore, were not studied by classical mathematical logic. However, they play a very important role in everyday commonsense reasoning and present a considerable challenge to AI researchers. In this chapter we show how defaults and various forms of exceptions to them are represented in ASP and how this general representation can be used for reasoning in a variety of simple domains. After that, we show how defaults can be used to reason about knowledge bases with incomplete information, which is represented by so-called null values. Next, we demonstrate how defaults can be prioritized so that, in some cases, one default is preferred over another. Finally, we discuss the use of defaults when representing hierarchies of classes and the inheritance of class properties by subclasses and members.

5.1 A General Strategy for Representing Defaults

In this section we present the strategy for representing defaults and their exceptions in ASP.

5.1.1 Uncaring John

To illustrate the general idea of defaults, let us go back to our family example from Chapter 1. Suppose you are Sam's teacher and you strongly

believe that Sam needs some extra help to pass the class. You convey this information to Sam's father, John, and expect some action on his part. Your reasoning probably goes along the following lines:

1. John is Sam's parent.
2. *Normally*, parents care about their children.
3. Therefore, John cares about Sam and will help him study.

The second statement is a typical example of a default.

To model this reasoning we introduce relation

$$cares(X, Y) - \text{“}X \text{ cares for } Y\text{.”}$$

The first inclination may be to ignore the word *normally* and simply expand the program

```
1  person(john).
2  person(sam).
3  person(alice).
4
5  father(john,sam).
6  mother(alice,sam).
7
8  parent(X,Y) :- father(X,Y).
9  parent(X,Y) :- mother(X,Y).
10
11 child(X,Y) :-  parent(Y,X).
```

by the new rule

```
% strict rule
cares(X,Y) :- parent(X,Y).
```

Program Π_1, consisting of lines 1–11 and the new strict rule, derives $cares(john, sam)$.

Assume now that in addition to the default “normally parents care about their children,” you learn that “John is an exception to this rule. He does not care about his children.” This new information can be represented by a rule:

```
12 -cares(john,X) :- child(X,john).
```

Unfortunately, the addition of this rule to Π_1 makes the program inconsistent, although in everyday reasoning this new information does not cause inconsistency. We simply withdraw our previous conclusion,

$cares(john, sam)$, and replace it by the new one, $\neg cares(john, sam)$. Representation of the default by a strict rule does not allow such nonmonotonic reasoning. We need another representation. This does not cause a difficulty. In ASP a default, d, stated as "Normally elements of class C have property P," is often represented by a rule:

$$\begin{aligned} p(X) \leftarrow\ & c(X), \\ & not\ ab(d(X)), \\ & not\ \neg p(X). \end{aligned}$$

Here, $ab(d(X))$ is read "X is abnormal with respect to d" or "a default d is not applicable to X" and $not\ \neg p(X)$ is read "$p(X)$ *may* be true."

The same technique can be used if X is a list of variables. For instance, the default d_{cares} "normally parents care about their children" can be represented as follows:

```
13 % default rule
14 cares(X,Y) :- parent(X,Y),
15               not ab(d_cares(X,Y)),
16               not -cares(X,Y).
```

Let Π_2 be the program given by lines 1–16. The new program is consistent and entails $\neg cares(john, sam)$ and $cares(alice, sam)$. As expected, the new information about John forces the program to withdraw one of its previous conclusions and replace it by the new one.

Defaults can have two types of exceptions – weak and strong. **Weak exceptions** render the default inapplicable (i.e., make our agent unable to use the default to come to a hasty conclusion). **Strong exceptions** refute the default's conclusion (i.e., they allow the agent to derive the opposite of the default). A weak exception $e(X)$ to a default d can be encoded by the so-called **cancellation axiom**

$$ab(d(X)) \leftarrow\ not\ \neg e(X). \tag{5.1}$$

which says that d is not applicable to X if X *may be* a weak exception to d. If e is a strong exception we need one more rule,

$$\neg p(X) \leftarrow e(X) \tag{5.2}$$

which allows us to defeat d's conclusion.

To illustrate the notion of weak exception, let us emulate a cautious reasoner who does not want to apply default d_{cares} to a parent of a child if that parent has never been seen at the school. We assume that the latter information is incomplete and is represented in our knowledge base by a collection

of literals of the form $absent(p)$ or $\neg absent(p)$ where p is a person. An absent parent can be viewed as a weak exception to the default. Let's consider what we would expect in the three possible cases of the availability of knowledge about Alice's absence. If $\neg absent(alice)$ were in the knowledge base, then the reasoner would use the default and conclude that Alice cares about Sam. If the knowledge base were to contain $absent(alice)$, then the reasoner would not wish to apply d_{cares} to Alice. In this case her feelings about Sam would be unknown. Last, if the knowledge base had no information about Alice's absence (i.e., if it contained neither $absent(alice)$ nor $\neg absent(alice)$), then being cautious, the reasoner would not wish to apply the default either.

To model this reasoning we expand our program by the following cancellation axiom:

```
17 ab(d_cares(P,C)) :- person(P), person(C),
18                     not -absent(P).
```

The new program, Π_3 on lines 1–18, has no information on Alice's absence and, as expected, answers *no* to query $cares(john, sam)$ and *unknown* to query $cares(alice, sam)$. Similar behavior would be exhibited if the program contained

```
absent(alice).
```

If the knowledge

```
-absent(alice).
```

were available, then $cares(alice, sam)$ would be concluded. (Note that if definite information $cares(alice, sam)$ or $\neg cares(alice, sam)$ existed in our knowledge base, then the default would have no effect, regardless of Alice's absence.)

Now consider uncaring John, a strong exception to default d_{cares}. This exception was represented in the program by the rule on line 12. However, according to our general methodology, the fact that John does not care for his children should have been translated into ASP by two rules:

```
-cares(john,X) :- parent(john,X).
ab(d_cares(john,X)) :- person(X),
                       not -parent(john,X).
```

It is easy to see, however, that addition of the second rule is unnecessary. (Indeed if John is a parent of X, then the first rule applies and defeats the default. If no information about $parent(john, X)$ is available, the default

is not applicable anyway.) The existence of unnecessary rules is, of course, not surprising. The most general methodology does not necessarily lead to the simplest representation. As in any type of writing, additional editing may substantially improve the result.

To better understand the need for the cancellation axiom for strong exceptions, let's consider another strong exception to d_{cares}. Assume the existence of a mythical country, u, whose inhabitants do not care for their children. This exception is represented by the rule

```
19 -cares(P,X) :- parent(P,X),
20                born_in(P,u).
```

and the cancellation axiom

```
21 ab(d_cares(P,X)) :- person(P), person(X),
22                     not -born_in(P,u).
```

Suppose we have an extension of our family knowledge base that contains information about the national origin of most (but not all) recorded people. Assume, for instance, that, according to our records, Pit and Kathy are the father and mother of Jim. Kathy was born in Moldova, but the national origin of Pit is unknown. He could have been born in u. This can be represented by the following rules:

```
23 person(pit).
24 person(kathy).
25 person(jim).
26 father(pit,jim).
27 mother(kathy,jim).
28 country(moldova).
29 country(u).
30 born_in(kathy,moldova).
31 %% A person can only be born in one country
32 -born_in(P,X) :- country(X),
33                  born_in(P,Y),
34                  X != Y.
```

To simplify the discussion, we assume that both parents have been seen at the school:

```
35 -absent(pit).
36 -absent(kathy).
```

Let Π_4 be the program given by lines 1–36. The fact that neither Pit nor Kathy is absent allows us to ignore the cancellation axiom on lines 17–18

and concentrate on the effects of the cancellation axiom that we just introduced on lines 21–22. We can see that, as expected, Π_4 answers queries $cares(kathy, jim)$ and $cares(pit, jim)$ with *yes* and *unknown*, respectively. Without the new cancellation axiom, the program would have derived that Pit did care about Jim, even though his origin was unknown. If later we were to learn that Pit were indeed from u, then the second answer would be replaced by a definite *no*.

5.1.2 Cowardly Students

Let us consider another example that has both weak and strong exceptions. Consider the following information:

1. Normally, students are afraid of math.
2. Mary is not.
3. Students in the math department are not afraid of math.
4. Those in CS may or may not be afraid of math.

The first statement corresponds to a default, say d. The next two can be viewed as strong exceptions to it. The fourth is a weak exception.

We assume that we are given two sorts of objects, $student$ and $dept$, containing names of all students and departments of the domain.

```
1 student(dave).student(mary).student(bob).student(pat).
2 dept(english).  dept(cs).  dept(math).
```

Suppose also that a (possibly incomplete) list

```
3 in(dave,english).
4 in(mary,cs).
5 in(bob,cs).
6 in(pat,math).
```

relates students to their unique departments. This uniqueness can be defined by rule

```
7 -in(S,D1)  :- dept(D1),
8               in(S,D2),
9               D1 != D2.
```

The default (statement 1 from the specification) is translated by the rule:

```
10 afraid(S,math) :- student(S),
11                   not ab(d(S)),
12                   not -afraid(S,math).
```

According to statement 2 from the specification, Mary is a strong exception to this default. According to our methodology, it can be represented as

```
13 -afraid(mary,math).
14 ab(d(mary)).
```

Note that, in this case, $ab(d(mary))$ is not necessary and can be removed.

The strong exception for math students (statement 3) is expressed as follows:

```
15 -afraid(S,math) :- in(S,math).
16 ab(d(S)) :- student(S),
17              not -in(S,math).
```

The following cancellation rule for CS students (statement 4) allow us to express the weak exception:

```
18 ab(d(S)) :- student(S),
19              not -in(S,cs).
```

It is easy to check that the program gives the following answers:

? $afraid(dave, math)$	Yes
? $afraid(mary, math)$	No
? $afraid(pat, math)$	No
? $afraid(bob, math)$	Unknown

Now consider another student, Jake, whose department affiliation is unknown. Given $person(jake)$, an agent using this program will correctly answer that it does not know whether Jake is afraid of math. Notice that this can only be done because of the cancellation axioms, which allow the expression of weak exceptions. However, if it was known that Jake was neither in the computer science nor in the math department, then the agent would correctly derive that Jake was afraid of math.

5.1.3 A Special Case

Let d be a default "Elements of class C normally have property P" and e be a set of exceptions to this default. If our information about membership in e is complete, then its representation can be substantially simplified. If e is a weak exception to d then cancellation axiom (5.1) can be written as

$$ab(d(X)) \leftarrow e(X). \tag{5.3}$$

If e is a strong exception then axiom (5.1) can be simply omitted.

For instance in our cowardly students example, if we had a complete list of students in the CS department, we could replace our original cancellation axiom for CS students by

```
ab(d(X)) :- in(X,cs).
```

If, in addition, we knew that our information about student membership in the math department is complete, then the cancellation axiom for math students could simply be dropped.

Similarly, if in the uncaring John example the place of birth for all parents were known, then the cancellation axiom on lines 21–22 could also be dropped. If information about which parents were never seen at school were complete, the axiom on lines 17–18 could be simplified to

```
ab(d_cares(P,C)) :- person(C),
                    absent(P).
```

5.2 Knowledge Bases with Null Values

Now let us look at the use of defaults for representing and reasoning about incomplete information in the presence of *null values* – constants used to indicate that the value of a certain variable or function is unknown. This technique is commonly used in relational databases, but due to the multiple and not always clearly specified meanings of these constants, it often leads to ambiguity and confusion. We now show how the use of defaults can alleviate this problem.

5.2.1 Course Catalog

Consider a database table representing a tentative summer schedule of a computer science department.

Professor	**Course**
mike	pascal
john	c
staff	prolog

Here "staff" is a null value that stands for an unknown professor. It expresses the fact that Prolog will be taught by *some* professor (possibly different from Mike or John).

To represent this information we assume that we are given two sorts, professors and courses

```
prof(mike).  prof(john).
course(pascal).  course(c).  course(prolog).
```

and introduce a relation $teaches(P, C)$ that says that professor P teaches a course C; the relation $teaches(staff, C)$ states that some professor teaches class C. The positive information from the table can be represented by a collection of facts:

```
teaches(mike,pascal).
teaches(john,c).
teaches(staff,prolog).
```

To represent negative information, we use default d: Normally, P teaches C only if this is listed in the schedule. Notice that d is not applicable to Prolog (or any other course taught by "staff"). This can be represented as follows:

```
-teaches(P,C) :- prof(P), course(C),
                 not ab(d(P,C)),
                 not teaches(P,C).
ab(d(P,C)) :- teaches(staff,C).
```

Check that the resulting program produces correct answers (*no* and *unknown*) to queries $teaches(mike, c)$ and $teaches(mike, prolog)$, respectively.

There can be yet another type of incompleteness in database tables.

Professor	**Course**
mike	pascal
john	c
{mike, john}	prolog

Here {mike, john} represents the second type of nulls in which the value is unknown, but is one of a specified finite set. To represent this information we simply expand our program by

```
teaches(mike,prolog) | teaches(john,prolog).
```

With the new program, our agent's answers to queries $teaches(mike, c)$, $teaches(mike, prolog)$, and $teaches(mike, prolog) \wedge teaches(john, prolog)$ are *No*, *Unknown*, and *No*, respectively.

5.3 Simple Priorities between Defaults

In this section we examine a way to represent simple priorities between defaults. We start with assuming that the agent's knowledge base contains a database of records similar to that used in the orphan story from Section 4.1. To make the example slightly more realistic, we remove the assumptions that a child's record contains complete information about that child's parents. We still assume that the deaths of all people whose records are kept in our knowledge base are properly recorded, as is information about being a child.

In addition to these records let us supply our agent with knowledge about some fictitious legal regulations. The first regulation says that *orphans are entitled to assistance according to special government program 1*, and the second says that all *children are entitled to program 0.* The rules also say that *program* 1 *is preferable to program* 0 (i.e., *a child qualified for receiving assistance from program* 1 *shall not receive assistance from program* 0), and that *no one can receive assistance from more than one program.*

Let us first represent these regulations and then define the records of the agent's knowledge base. Legal regulations usually come with exceptions and hence can be viewed as defaults. What follows is a complete representation. We use relation $record\_for(P)$ to indicate that the agent's knowledge base contains a record for person P. First we list the assistance programs; next we represent the first two regulations by standard default rules and by a rule prohibiting assistance from more than one program:

```
1  program(0).
2  program(1).
3
4  entitled(X,1) :- record_for(X),
5                   orphan(X),
6                   not ab(d1(X)),
7                   not -entitled(X,1).
8  entitled(X,0) :- record_for(X),
9                   child(X),
10                  not ab(d2(X)),
11                  not -entitled(X,0).
12 -entitled(X,N2) :- program(N1), program(N2),
13                     record_for(X),
14                     entitled(X,N1),
15                     N1 != N2.
```

The next two rules express preference for program 1 over program 0. Essentially we treat orphans (who are entitled to assistance from program 1) as strong exceptions to the second default. (In other circumstances preference can be expressed by weak exceptions.) The double negation in the last rule is needed because we may not know if a child is an orphan due to incompleteness of our record.

```
16 -entitled(X,0) :- record_for(X),
17                   orphan(X).
18 ab(d2(X)) :- record_for(X),
19              not -orphan(X).
```

There are also the following strong exceptions to both defaults:

```
20 -entitled(X,N) :- record_for(X),
21                   dead(X),
22                   program(N).
23 -entitled(X,N) :- record_for(X),
24                   -child(X),
25                   program(N).
```

(Since information about *dead* and *child* is complete, we do not need cancellation axioms for these strong exceptions.) Assuming that some person, say Joe, is alive, the rules guarantee that if Joe is an orphan, he will receive assistance from program 1. If Joe is a child who is not an orphan, he will be assisted by program 0. However, if Joe is a child and it is not known whether he is an orphan or not, Joe will receive no benefits at all. Of course this is not right. Something should be done about this case of insufficient documentation. The problem can be detected by the following rule:

```
26 check_status(X) :- record_for(X),
27                    not -orphan(X),
28                    not orphan(X).
```

In other words, the person in charge of financial assistance will need to go to some extra trouble to check person X's status if the system does not know whether or not X is an orphan.

Let us now describe records from the agent's knowledge base. We use a slight modification of the database from the previous family example. As before, we assume that there is a sort *person* satisfied by any reasonable name we decide to use. But the rest of the information is more structured. For testing purposes let us assume that there are records for the following people:

```
29 record_for(bob).
30 father(rich,bob).
31 mother(patty,bob).
32 child(bob).
33
34 record_for(rich).
35 father(charles,rich).
36 mother(susan,rich).
37 dead(rich).
38
39 record_for(patty).
40 dead(patty).
41
42 record_for(mary).
43 child(mary).
44 mother(patty,mary).
```

To express the assumption that the deaths of all people whose records are kept in the knowledge base are properly recorded, we expand the above records by CWA for *dead*:

```
45 -dead(P) :- record_for(P),
46              not dead(P).
```

We do so similarly for the children.

```
47 -child(X) :- record_for(X),
48               not child(X).
```

Notice that our closed world assumptions only apply to people who have records in the agent's knowledge base. This is simply because we are not going to ask any questions about those who are not in it. If, by accident, such a question is asked, the answer will be *unknown*.

Now we are ready to define a notion of orphan. The positive part of the definition remains the same as before:

```
49 orphan(P) :- child(P),
50              parents_dead(P).
```

The negative part, however, undergoes a substantial change. The closed world assumption for orphans used in the previous example is replaced by a weaker statement:

```
51 -orphan(P) :- record_for(P),
52                not may_be_orphan(P).
```

where may_be_orphan is defined as follows:

```
53 may_be_orphan(P) :- record_for(P),
54                     child(P),
55                     not -parents_dead(P).
56
57 parent(X,P) :- father(X,P).
58 parent(X,P) :- mother(X,P).
59
60 parents_dead(P) :- father(X,P),
61                    dead(X),
62                    mother(Y,P),
63                    dead(Y).
64
65 -parents_dead(P) :- parent(X,P),
66                     -dead(X).
```

It is easy to check that the program entails $entitled(bob, 1)$, $\neg entitled(bob, 0)$, and $check\_status(mary)$.

Suppose that after checking the status of Mary, the administrator discovered that Mary has a father, Mike, who is alive. In that case, the record for Mary should be expanded by

```
67 father(mike,mary).
```

We also need a new record for Mike:

```
68 record_for(mike).
```

Now the program entails $entitled(bob, 1)$, $\neg entitled(bob, 0)$, $entitled(mary, 0)$, and $\neg entitled(mary, 1)$.

Of course the program also derives that no other person whose record is stored in the database is entitled to any of the assistance programs. (Note, however, that the system is not able to give a definite answer about entitlements for Charles and Susan, who have no such records. Moreover, the records for these people cannot be created because we do not know if they are dead or alive. The addition would violate our assumption about completeness of the death records.) This is one more example of the usefulness of discriminating between falsity and the mere absence of information. The distinction, together with the simple expression of preferences between defaults, allows our program to produce correct conclusions based on such absence.

5.4 Inheritance Hierarchies with Defaults

Whenever humans have a hierarchical organization of information, we make assumptions about certain properties that members of classes share with each other. For example, if we find out that something is an animal, we can assume that it eats, breathes, and so on. However, there can always be exceptions to such rules, and it is foolish to cling too tightly to conclusions we make based on class membership. Default reasoning is essential for making commonsense assumptions, but taking exceptions into account is essential to true intelligence.

5.4.1 Submarines Revisited

Now that we have the power to represent defaults, let's return to the hierarchical representation of our *Narwhal* example from Section 4.3. Instead of saying that all submarines are black, we can say that *normally* submarines are black. In accordance with our general methodology we simply replace

```
has_color(X,black) :- member(X,sub).
```

by

```
has_color(X,black) :- member(X,sub),
                      not ab(dc(X)),
                      not -has_color(X,black).
```

Suppose that we learned about a submarine named *Blue Deep*. In accordance with its name, this submarine is blue. (After all, it makes sense to use blue for better camouflage in case the screen door does not deter some of the fish.) The new information can be added as follows:

```
object(blue_deep).
color(blue).
is_a(blue_deep,sub).
has_color(blue_deep,blue).
```

Since colors of submarines are unique (see the rule on lines 29–31 from Section 4.3), the new blue submarine is a strong exception to our default. As expected the new program is consistent. It allows us to conclude that the *Blue Deep* is blue while retaining our ability to conclude that the *Narwhal* is black by default. If later we learn that the *Narwhal* is also blue, this would cause no contradiction.

5.4.2 Membership Revisited

Another lovely consequence of being able to add the word "normally" to our statements is that we can weaken our assumption that leaf classes of hierarchies are disjoint. The positive part of the definition of $member$ and the definition of $sibling$ remain unchanged:

```
member(X,C) :- is_a(X,C).
member(X,C) :- is_a(X,C0),
               subclass(C0,C).
siblings(C1,C2) :- is_subclass(C1,C),
                   is_subclass(C2,C),
                   C1 != C2.
```

The only change needed is the addition of a new line at the end of the old definition of $\neg member(X, C)$. Here is the new rule:

```
-member(X,C2) :- member(X,C1),
                 siblings(C1,C2),
                 C1 != C2,
                 not member(X,C2).
```

Now we can introduce an amphibious vehicle called *Darling* owned by some great man of action. It belongs both to the car and the submarine class. This can be recorded by

```
object(darling).
is_a(darling, car).
is_a(darling, sub).
```

The resulting program allows our agent to deduce that the *Darling* is a member of both subclasses, but that the *Narwhal* is a sub and not a car.

5.4.3 The Specificity Principle

Let's consider another example that illustrates a classic problem that arose with the study of inheritance hierarchies. "Eagles and penguins are types of birds. Birds are a type of animal. Sam is an eagle, and Tweety is a penguin. Tabby is a cat." We represent this hierarchy exactly as before:

```
1 class(animal).
2 class(bird).
3 class(eagle).
```

```
4  class(penguin).
5  class(cat).
6
7  object(sam).
8  object(tweety).
9  object(tabby).
10
11 is_subclass(eagle,bird).
12 is_subclass(penguin,bird).
13 is_subclass(bird,animal).
14 is_subclass(cat,animal).
15
16 subclass(C1,C2) :- is_subclass(C1,C2).
17 subclass(C1,C2) :- is_subclass(C1,C3),
18                     subclass(C3,C2).
19
20 is_a(sam,eagle).
21 is_a(tweety,penguin).
22 is_a(tabby,cat).
23
24 member(X,C) :- is_a(X,C).
25 member(X,C) :- is_a(X,C0),
26                 subclass(C0,C).
27
28 siblings(C1,C2) :- is_subclass(C1,C),
29                     is_subclass(C2,C),
30                     C1 != C2.
31 -member(X,C2) :- member(X,C1),
32                   siblings(C1,C2),
33                   C1 != C2,
34                   not member(X,C2).
```

Our agent should now be able to answer correctly that Tweety is not an eagle but that Tweety is a penguin, a bird, and an animal. All these queries can be made using the $member$ predicate.

Now we add default properties of classes. For example, *animals normally do not fly, birds normally fly, and penguins normally do not fly*. (The last default may look strange but we need it for illustrative purposes. After all, we may eventually want to consider penguins that fly because they are sprinkled with pixie dust.) What does the new theory allow us to conclude

about Sam's ability to fly? Since Sam is both a bird and an animal, his flying abilities are defined by two contradictory defaults. The program has two answer sets containing $fly(sam)$ and $\neg fly(sam)$, respectively. Our common sense, however, tells us that only the first conclusion is justified. This is apparently the result of a broadly shared, commonsense **specificity principle** that states that *more specific information overrides less specific information.* The principle was first formalized by David S. Touretzky. The default "normally elements of class C_1 have property P" is preferred to the default "normally elements of class C_2 have property $\neg P$" if C_1 is a subclass of C_2. The following rules represent our defaults together with the specificity principle.

```
%% Animals normally do not fly.
-fly(X) :- member(X,animal),
           not ab(d1(X)),
           not fly(X).

%% Birds normally fly.
fly(X) :- member(X,bird),
          not ab(d2(X)),
          not -fly(X).

%% Penguins normally do not fly.
-fly(X) :- member(X,penguin),
           not ab(d3(X)),
           not fly(X).

%% X is abnormal w.r.t d2 if X might be a  penguin.
ab(d2(X)) :- not -member(X,penguin).

%% X is abnormal w.r.t d1 if X might be a  bird.
ab(d1(X)) :- not -member(X,bird).
```

The last rule, which prohibits the application of default d_1 to animals that might possibly be birds, expresses preference for default d_2 over default d_1. The previous rule does the same for defaults d_3 and d_2. We chose to express exceptions this way because our hierarchy allows objects for which the complete characterization of their membership relation is unknown. For example, if there is a bird that cannot be classified further, the agent should

conclude "unknown" about its flying ability. After all, the bird might turn out to be a penguin.

Now our agent has no problem figuring out which animals and birds fly and which do not. Sam flies and Tweety and Tabby do not. And if we wanted to teach the program about baby eagles that do not fly, or penguins that do, we could.

Of course, the specificity principle is applicable to defaults with arbitrary incompatible conclusions, not just those of the form P and $\neg P$. For example, "Tucson is normally sunny" could be overridden by the more specific default that states, "Tucson is normally rainy during the monsoon season." The incompatibility of "sunny" and "rainy" may indirectly follow from other rules of the program.

Note that our formalization does not have a single rule expressing the specificity principle. Instead we need to write a separate rule for each pair of the corresponding contradictory defaults. This requirement is not surprising because to write such a rule we would need to reify defaults. There are commonsense theories in which defaults are reified and the specificity principle is stated as one rule, but they are beyond the scope of this book. You can give it a try in the last exercise.

5.5 (*) Indirect Exceptions to Defaults

In this section we consider yet another type of possible exceptions to defaults, sometimes referred to as **indirect exceptions**. Intuitively, these are rare exceptions that come into play only as a last resort, to restore the consistency of the agent's worldview when all else fails. The representation of indirect exceptions seems to be beyond the power of ASP. This observation led to the development of a simple but powerful extension of ASP called **CR-Prolog** (or ASP with consistency-restoring rules). To illustrate the problem let us look at the following example.

Consider an ASP representation of the default "elements of class c normally have property p":

$$\begin{aligned} p(X) \leftarrow\ & c(X), \\ & not\ ab(d(X)), \\ & not\ \neg p(X). \end{aligned}$$

together with the rule

$$q(X) \leftarrow p(X).$$

and two observations:

$$c(x).$$
$$\neg q(x).$$

where x is a constant denoting a particular object of the domain. It is not difficult to check that this program is inconsistent. No rules allow the reasoner to prove that the default is not applicable to x (i.e., to prove $ab(d(x))$) or that x does not have property p. Hence the default must conclude $p(x)$. The second rule implies $q(x)$, which contradicts the second observation.

There, however, seems to exists a commonsense argument that may allow a reasoner to avoid inconsistency and to conclude that x is an indirect exception to the default. The argument is based on the **Contingency Axiom** for default $d(X)$ that says, *"Any element of class c can be an exception to the default $d(X)$ above, but such a possibility is very rare and, whenever possible, should be ignored."* One may informally argue that since the application of the default to x leads to a contradiction, the possibility of x being an exception to $d(x)$ cannot be ignored and hence x must satisfy this rare property.

In what follows we give a brief description of CR-Prolog – an extension of ASP capable of encoding and reasoning about such rare events. We start with a description of the syntax and semantics of the language.

A program of CR-Prolog is a four-tuple consisting of

1. A (possibly sorted) signature.
2. A collection of regular rules of ASP.
3. A collection of rules of the form

$$l_0 \stackrel{+}{\leftarrow} l_1, \ldots, l_k, \ not\ l_{k+1}, \ldots, \ not\ l_n \tag{5.4}$$

   where ls are literals. Rules of type (5.4) are called **consistency-restoring rules (cr-rules)**.
4. A partial order, $\leq$, defined on sets of cr-rules. This partial order is often referred to as a **preference relation**.

Intuitively, rule (5.4) says that if the reasoner associated with the program believes the body of the rule, then it "may possibly" believe its head; however, this possibility may be used only if there is no way to obtain a consistent set of beliefs by using only regular rules of the program. The partial order over sets of cr-rules is used to select preferred possible resolutions of the conflict. Currently the inference engine of CR-Prolog supports two such relations. One is based on the set-theoretic inclusion ($R_1 \leq_1 R_2$ holds iff $R_1 \subseteq R_2$). Another is defined by the cardinality of the

corresponding sets ($R_1 \leq_2 R_2$ holds iff $|R_1| \leq |R_2|$). To give the precise semantics we need some terminology and notation.

The set of regular rules of a CR-Prolog program Π is denoted by Π^r; the set of cr-rules of Π is denoted by Π^{cr}. By $\alpha(r)$ we denote a regular rule obtained from a consistency-restoring rule r by replacing $\stackrel{+}{\leftarrow}$ by $\leftarrow$; α is expanded in a standard way to a set R of cr-rules, i.e., $\alpha(R) = \{\alpha(r) : r \in R\}$. As in the case of ASP, the semantics of CR-Prolog is given for ground programs. A rule with variables is viewed as a shorthand for a schema of ground rules.

Definition 5.5.1. *(Abductive Support)*
A minimal (with respect to the preference relation of the program) collection R of cr-rules of Π such that $\Pi^r \cup \alpha(R)$ is consistent (i.e., has an answer set) is called an **abductive support** *of Π.*

Definition 5.5.2. *(Answer Sets of CR-Prolog)*
A set A is called an answer set *of Π if it is an answer set of a regular program $\Pi^r \cup \alpha(R)$ for some abductive support R of Π.*

Consider, for instance, the following CR-Prolog program:

$$p(a) \leftarrow \ not\ q(a).$$
$$\neg p(a).$$
$$q(a) \stackrel{+}{\leftarrow} .$$

It is easy to see that the regular part of this program (consisting of the program's first two rules) is inconsistent. The third rule, however, provides an abductive support that allows resolution of the inconsistency. Hence the program has one answer set $\{q(a), \neg p(a)\}$.

The previous example had only one possible resolution of the conflict, and hence its abductive support did not depend on the preference relation of the program. This is, of course, not always the case. Consider for instance the following collection of rules:

$$p_1 \leftarrow \ not\ \neg p_1.$$
$$\neg p_1 \stackrel{+}{\leftarrow} .$$
$$p_2 \leftarrow \ not\ \neg p_2.$$
$$\neg p_2 \stackrel{+}{\leftarrow} .$$
$$p_3 \leftarrow \ not\ \neg p_3.$$
$$\neg p_3 \stackrel{+}{\leftarrow} .$$

$$r \leftarrow p_1.$$
$$\neg r \leftarrow p_2.$$
$$\neg r \leftarrow p_3.$$

It is not difficult to see that a program consisting of these rules and set inclusion as the preference relation has two answer sets $\{\neg p_1, p_2, p_3, \neg r\}$ and $\{p_1, \neg p_2, \neg p_3, r\}$ corresponding to abductive supports $\{\neg p_1 \stackrel{+}{\leftarrow}\}$ and $\{\neg p_2 \stackrel{+}{\leftarrow}, \ \neg p_3 \stackrel{+}{\leftarrow}\}$. (Note that the collection of all three cr-rules of the program does not provide a minimal conflict resolution.)

Consider now the program consisting of these rules and the cardinality-based preference relation. Observe that this program has only one abductive support and one answer set, $\{\neg p_1, p_2, p_3, \neg r\}$. Now let us show how CR-Prolog can be used to represent defaults and their indirect exceptions. The CR-Prolog representation of default $d(X)$ may look as follows:

$$\begin{aligned} p(X) \leftarrow\ & c(X), \\ & not\ ab(d(X)), \\ & not\ \neg p(X). \\ \neg p(X) \stackrel{+}{\leftarrow}\ & c(X). \end{aligned}$$

The first rule is the standard ASP representation of the default, whereas the second rule expresses the contingency axiom for default $d(X)$.[1] Consider now a program obtained by combining these two rules with an atom

$$c(a).$$

The program's answer set is $\{c(a), p(a)\}$. Of course this is also the answer set of the regular part of our program. (Since the regular part is consistent, the contingency axiom is ignored.) Let us now expand this program by the rules

$$q(X) \leftarrow p(X).$$
$$\neg q(a).$$

The regular part of the new program is inconsistent. To save the day we need to use the contingency axiom for $d(a)$ to form the abductive support of the program. As a result the new program has the answer set $\{\neg q(a), c(a), \neg p(a)\}$. The new information does not produce inconsistency as in the analogous case of ASP representation. Instead the program

[1] In this form of the contingency axiom, we treat X as a strong exception to the default. Sometimes it may be useful to also allow weak indirect exceptions; this can be achieved by adding the rule: $ab(d(X)) \stackrel{+}{\leftarrow} c(X)$.

withdraws its previous conclusion and recognizes a as a (strong) exception to default $d(a)$.

Here is another small example: Consider a reasoning agent whose knowledge base contains a default that says that "people normally keep their cars in working condition." A (slightly simplified) version of this default can be represented in CR-Prolog as follows:

$$\begin{aligned} \neg broken(X) \leftarrow\ & car(X), \\ & not\ ab(d(X)), \\ & not\ broken(X). \\ broken(X) \stackrel{+}{\leftarrow}\ & car(X). \end{aligned}$$

Suppose also that the agent has some information about the normal operations of cars (e.g., it knows that turning the ignition key starts the car's engine). This knowledge can be represented by the rules:

$$\begin{aligned} starts(X) \leftarrow\ & turn\_key(X), \\ & \neg broken(X). \\ \neg starts(X) \leftarrow\ & turn\_key(X), \\ & broken(X). \end{aligned}$$

(For simplicity we assume here that broken cars do not start.)

Given that the ignition key of car x was turned, i.e., statements

$$car(x).$$

$$turn\_key(x).$$

belong to the agent's knowledge base, the agent will be able to use regular rules to conclude $\neg broken(x)$ and $starts(x)$. If, however, in addition the agent learns

$$\neg starts(x)$$

both of the above conclusions will be withdrawn, and the cr-rule will be used to prove $broken(x)$, which, in turn, will imply $\neg starts(x)$.

The possibility of encoding rare events that may serve as unknown exceptions to defaults has proven to be very useful for various knowledge representation tasks, including planning, diagnostics, and reasoning about the agent's intentions. The later chapters contain a number of examples of such uses.

There are currently two ways to run CR-Prolog programs. One is to use a solver called CRModels, which can be found at `http://www`

`.mbal.tk/crmodels/`. The other is to use SPARC as described in Appendix C.

Summary

In this chapter we discussed the general strategy for representing defaults and their exceptions. The strategy explains how to translate natural-language statements of the form "normally, typically, as a rule," etc., into defeasible ASP rules and how to classify exceptions into strong, weak, and indirect. We showed how the first two types of exceptions could be recorded in ASP and gave a general representation of indirect exceptions in CR-Prolog. The nonmonotonicity of the ASP entailment was crucial for reasoning that allows the retraction of the defaults' conclusions after new information about exceptions to them became available. A number of examples illustrated the use of defaults for various knowledge representation tasks. In particular we showed how defaults could be used to specify negative information in databases containing incomplete knowledge expressed by null values – a task notoriously difficult in weaker languages. Another example dealt with inheritance hierarchies with default properties of classes. It showed how defaults could be blocked from being inherited by subclasses or objects using the general strategy for encoding exceptions. This allowed us to formalize the informal "specificity principle" used by people in their commonsense reasoning. A similar strategy could be used for specifying general preferences between defaults.

The ability to represent and reason about defaults is a very substantial achievement in KR. The design of logics and languages capable of doing this and gaining the ability to understand the correct ways of reasoning with defaults and their exceptions took many years of extensive research. Now, however, it has been honed enough to be included in a textbook.

References and Further Reading

A number of examples and techniques introduced in this chapter (including the representation of defaults with strong and weak exceptions) were first presented in Baral and Gelfond (1994). The "orphans" example was discussed in Gelfond (2002). The treatment of null values follows that of Taylor and Gelfond (1994). A rather general treatment of defaults including preferences between them can be found in Gelfond and Son (1997). In that paper defaults are reified and the preference relation can be defined

by the context-dependent rules of the program. There is also a substantial amount of work on defining preferences between arbitrary logic programming rules as well as between answer sets of a logic program; see, for instance, Sakama and Inoue (2000), Delgrande, Schaub, and Tompits (2003), Delgrande et al. (2004), and Brewka and Eiter (1998). A general account of preferences and their role in nonmonotonic reasoning can be found in Brewka, Niemela, and Truszczynski (2008). The specificity principle is due to David Touretzky (1986). CR-Prolog was first introduced by Marcello Balduccini and Michael Gelfond in Doherty, McCarthy, and Williams (2003) and Balduccini (2007). An interesting application of CR-Prolog to planning can be found in Balduccini (2004); Balduccini (2007) contains the description of a CR-Prolog reasoning algorithm implemented in CRModels.

Exercises

Use the methodology for encoding defaults to represent the knowledge given in the following stories in ASP.

1. "Apollo and Helios are lions in a zoo. Normally lions are dangerous. Baby lions are not dangerous. Helios is a baby lion." Assume that the zoo has a complete list of baby lions that it maintains regularly. Your program should be able to deduce that Apollo is dangerous, whereas Helios is not. Make sure that (a) if you add another baby lion to your knowledge base, the program would derive that it is not dangerous, even though that knowledge is not explicit; and (b) if you add an explicit fact that Apollo is not dangerous, there is no contradiction and the program answers intelligently.
2. "John is married to Susan and Bob is married to Mary. Married people normally like each other. However, Bob hates Mary."
   (a) Make sure your program answers *yes* to queries
   $?\ likes(john, susan)$,
   $?\ likes(susan, john)$,
   $?\ likes(mary, bob)$,
   and *no* to $?\ likes(bob, mary)$.
   (b) Add the following knowledge to your program. "Arnold and Kate are also married, but Kate's behavior often does not follow predictable rules." Make sure your program can deduce that Arnold likes Kate, but it is unknown whether Kate likes Arnold. (Note that if you used the methodology properly, you should not need to change the first program but can just add to it.)

3. "American citizens normally live in the United States. American diplomats may or may not live in the United States. John, Miriam, and Caleb are American citizens. John lives in Italy. Miriam is an American diplomat."
   (a) Assume we do *not* have a complete list of American diplomats. (Note that your program should not be able to conclude that Caleb lives in the United States.)
   (b) Now assume we have a *complete* list of American diplomats. Add this information to the program. What does your new program say about Caleb's place of residence?
   (c) Rewrite the program from 3b by using the simplified form of the cancellation axiom.
4. "Adults normally work. Children do not work. Students are adults but they normally do not work. John and Betty are students. John works. Bob and Jim are adults who are not students. Bob does not work. Kate is an adult who may or may not be a student. Mary is a child." Make sure that your program is not only capable of answering questions about who works and does not work but also of who is or is not a child, an adult, and so on.
5. "A field that studies pure ideas does not study the natural world. A field that studies the natural world does not study pure ideas. Mathematics normally studies pure ideas. Science normally studies the natural world. As a computer scientist, Daniela studies both mathematics and science. Both mathematics and science study our place in the world." Make sure your program can deduce that Daniela studies our place in the world.
6. "Cars normally have four seats. Pick-up trucks are exceptions to this rule. They normally have two seats. Pick-up trucks with extended cab are exceptions to this rule. They have four seats." Your program should work correctly in conjunction with complete lists of facts of the form

$$car(a), car(b), car(c), \ldots$$
$$pickup(b), pickup(c), \ldots$$
$$extended\_cab(c), \ldots$$

7. You are given three complete lists of facts of the form

$$course(math), course(graphs), \ldots$$
$$student(john), student(mary), \ldots$$
$$took(john, math), took(mary, graphs), \ldots$$

Students can graduate only if they have taken all the courses in the first list. Write a program that, given the above information, determines which students can graduate. Make sure that, given the following sample knowledge base,

$$student(john).$$
$$student(mary).$$
$$course(math).$$
$$course(graphs).$$
$$took(john, math).$$
$$took(john, graphs).$$
$$took(mary, graphs).$$

your program is able to

$$conclude\ can\_graduate(john).$$
$$\neg can\_graduate(mary).$$

8. Consider the problem presented in Exercise 7. This time, however, the list of courses that the students took may be incomplete. Write a program that determines
   - which students can graduate
   - which students cannot graduate
   - which students' ability to graduate cannot be determined from the knowledge base; in this case, the program must recommend a review of the records of those students.

   Make sure that, given the following sample knowledge base,

$$student(john).$$
$$student(mary).$$
$$student(bob).$$
$$student(rick).$$
$$course(math).$$
$$course(graphs).$$
$$took(john, math).$$
$$took(john, graphs).$$
$$\neg took(mary, math).$$
$$took(mary, graphs).$$
$$\neg took(bob, math).$$

your program is able to conclude

$$can\_graduate(john).$$
$$\neg can\_graduate(mary).$$
$$\neg can\_graduate(bob).$$
$$review\_records(rick).$$

9. Using the notions of hierarchy and defaults as detailed in Section 5.4, write an ASP program to represent the following information. Be as general as you can.

   - A Selmer Mark VI is a saxophone.
   - Jake's saxophone is a Selmer Mark VI.
   - Mo's saxophone is a Selmer Mark VI.
   - Part of a saxophone is a high D key.
   - Part of the high D key is a spring that makes it work.
   - The spring is normally not broken.
   - Mo's spring for his high D key is broken.

   Make sure that your program correctly entails that Jake's saxophone works while Mo's is broken. For simplicity, assume that no one has more than one saxophone, and hence, saxophones can be identified by the name of their owner.

10. Consider the program from Section 5.4.3 in which lines 51 and 54 are replaced by

```
ab(d2(X)) :- member(X,penguin).
ab(d1(X)) :- member(X,bird).
```

    Given a bird, Squeaky, of unknown species, what would the agent answer about its flying ability?

11. (*) Encode the following story using CR-Prolog: Given the following program:

```
day(d1).
day(d2).
% A day is considered to be a schoolday according to
% the school calendar if it is not stated to be
% otherwise:
schoolday(D) :- day(D),
                not -schoolday(D).
```

```
% Schools are normally open on schooldays:
open_school(D) :- schoolday(D),
                  not ab(d(D)),
                  not -open_school(D).

% If school is open on day D, then D is not a
% snowday.
-snowday(D) :- open_school(D).

snowday(d1).
```

Add a cr-rule to resolve the inconsistency. It should express that, in rare cases, schools are closed on school days.

12. (**) We have noted before that to encode the specificity principle explicitly, we would have to reify the notion of default. Rewrite the program from Section 5.4.3 making the defaults objects of the domain and encode the specificity principle as a general rule about defaults.