CHAPTER 19

Incomplete Game Descriptions

19.1 Introduction

In our discussion of General Game Playing thus far, we have assumed that game descriptions are complete. This does not mean that players have full information. For example, in multiple player games, they do not know the moves of other players before they make those moves; and, in the case of imperfect information games, they do not even know the exact state of the world, only what they can perceive of it. However, in our work thus far, we have assumed that the game rules are complete - they completely define the initial state, legality, percepts, update, goals, and termination in terms of states and actions.

Unfortunately, in real world settings, complete information is not always available. In some cases, agents do not know all of the effects of their actions; they may not even know exactly which actions are legal or what their rewards are or when a game is over.

Incomplete Game Description Language (or IGDL) is a variant of GDL designed to facilitate the encoding of incomplete game descriptions. In place of rules that define concepts exactly, IGDL has logical sentences that constrain concepts in more or less detail. (Spoiler for those with background in Logic and Logic Programming - the essential difference between GDL and IGDL is that, in IGDL, there is no negation as failure.)

In this chapter, we define IGDL and illustrate its use in writing incomplete descriptions of various sorts. In the next section, we introduce Relational Logic, which is the logical basis for IGDL. In the section after that, we define IGDL in Relational Logic in much the same way that we defined GDL in terms of Logic Programming. We then show how to use IGDL in the context of some examples. Finally, we talk about game management and game play.

19.2 Relational Logic

The basic elements of Relational Logic are the same as those of Logic Programming. We have the same vocabulary - object constants, function constants, relation constants, and variables. We define terms in exactly the same way - as object constants, variables, and functional terms. And we define atomic sentences and literals in the same way as well.

The main difference between the language of Relational Logic and the language of Logic Programming is that, in Relational Logic, we write (1) logical sentences and (2) quantified sentences instead of rules.

There are five types of *logical sentences* in Relational Logic - negations, conjunctions, disjunctions, implications, and equivalences.

A *negation* consists of the negation operator ~ and a simple or compound sentence, called the *target*. For example, given the sentence `p(a)`, we can form the negation of `p(a)` as shown below.

```
(~p(a))
```

A *conjunction* is a sequence of sentences separated by occurrences of the `&` operator and enclosed in parentheses, as shown below. The constituent sentences are called *conjuncts*. For example, we can form the conjunction of `p(a)` and `q(a)` as follows.

```
(p(a) & q(a))
```

A *disjunction* is a sequence of sentences separated by occurrences of the | operator and enclosed in parentheses. The constituent sentences are called *disjuncts*. For example, we can form the disjunction of `p(a)` and `q(a)` as follows.

```
(p(a) | q(a))
```

An *implication* consists of a pair of sentences separated by the => operator and enclosed in parentheses. The sentence to the left of the operator is called the *antecedent*, and the sentence to the right is called the *consequent*. The implication of `p(a)` and `q(a)` is shown below.

```
(p(a) => q(a))
```

A *biconditional*, is a combination of an implication and a reverse implication. For example, we can express the biconditional of `p(a)` and `q(a)` as shown below.

```
(p(a) <=> q(a))
```

Note that the constituent sentences within any compound sentence can be either simple sentences or compound sentences or a mixture of the two. For example, the following is a legal compound sentence.

```
((p(a) | q(a)) => r(a))
```

A *quantified sentence* in Relational Logic is formed from a *quantifier*, a variable, and an embedded sentence. The embedded sentence is called the *scope* of the quantifier. There are two types of quantified sentences in Relational Logic, viz. universally quantified sentences and existentially quantified sentences.

A *universally quantified sentence* is used to assert that all objects have a certain property. For example, the following expression is a universally quantified sentence asserting that, if `p` holds of an object, then `q` holds of that object and itself.

```
(AX:(p(X) => q(X,X))
```

An *existentially quantified sentence* is used to assert that some object has a certain property. For example, the following expression is an existentially quantified sentence asserting that there is an object that satisfies `p` and, when paired with itself, satisfies `q` as well.

```
(EX:(p(X) & q(X,X))
```

Note that quantified sentences can be nested within other sentences. For example, in the first sentence below, we have quantified sentences inside of a disjunction. In the second sentence, we have a quantified sentence nested inside of another quantified sentence.

```
(AX:p(X)) | (EX:q(X,X))
   (AX:(EY:q(X,Y)))
```

One disadvantage of our notation, as written, is that the parentheses tend to build up and need to be matched correctly. It would be nice if we could dispense with parentheses, e.g. simplifying the preceding sentence to the one shown below.

```
p(a) | q(a) => r(a)
```

Unfortunately, we cannot do without parentheses entirely, since then we would be unable to render certain sentences unambiguously. For example, the sentence shown above could have resulted from dropping parentheses from either of the following sentences.

```
(p(a) | q(a)) => r(a)
 p(a) | (q(a) => r(a))
```

The solution to this problem is the use of *operator precedence*. The following table gives a hierarchy of precedences for our operators. The ~ operator has higher precedence than `&`; `&` has higher precedence than |; and | has higher precedence than => and <=>.

```
   ~
   &
   |
=> <=>
```

In unparenthesized sentences, it is often the case that an expression is flanked by operators, one on either side. In interpreting such sentences, the question is whether the expression associates with the operator on its left or the one on its right. We can use precedence to make this determination. In particular, we agree that an operand in such a situation always associates with the operator of higher precedence. When an operand is surrounded by operators of equal precedence, the operand associates to the right.

As with Logic Programming, the *Herbrand base* for Relational Logic language is the set of all ground relational sentences that can be formed from the constants of the language. Said another way, it is the set of all sentences of the form $r(t_1,...,t_n)$, where r is an n-ary relation constant and t_1, ... , t_n are ground terms.

A *truth assignment* for a Relational Logic language is a function that maps each ground relational sentence in its Herbrand base to a truth value. For example, the truth assignment defined below is an example for the case of the language mentioned a few paragraphs above.

```
p(a)   → 1
p(b)   → 0
q(a,a) → 1
q(a,b) → 0
q(b,a) → 1
q(b,b) → 0
```

Once we have a truth assignment for the ground relational sentences of a language, the semantics of our operators prescribes a unique extension of that assignment to the complex sentences of the language. A truth assignment satisfies a negation ~ϕ if and only if it does not satisfy ϕ. A truth assignment satisfies a conjunction (ϕ_1 & ... & ϕ_n) if and only if it satisfies every ϕ_i. A truth assignment satisfies a disjunction (ϕ_1 | ... | ϕ_n) if and only if it satisfies at least one ϕ_i. A truth assignment satisfies an implication (ϕ => ψ) if and only if it does not satisfy ϕ or does satisfy ψ. A truth assignment satisfies an equivalence (ϕ <=> ψ) if and only if it satisfies both ϕ and ψ or it satisfies neither ϕ nor ψ.

In order to define satisfaction of quantified sentences, we need the notion of instances. An *instance* of an expression is an expression in which all variables have been consistently replaced by ground terms. Consistent replacement here means that, if one occurrence of a variable is replaced by a ground term, then all occurrences of that variable are replaced by the same ground term.

A universally quantified sentence is true for a truth assignment if and only if *every* instance of the scope of the quantified sentence is true for that assignment. An existentially quantified sentence is true for a truth assignment if and only if *some* instance of the scope of the quantified sentence is true for that assignment.

As an example of these definitions, consider the sentence `AX:(p(X) => q(X,X))`. What is the truth value under the truth assignment shown above? According to our definition, a universally quantified sentence is true if and only if every instance of its scope is true. For this language, with object constants `a` and `b` and no function constants, there are just two instances. See below.

```
p(a) => q(a,a)
```

```
p(b) => q(b,b)
```

We know that `p(a)` is true and `q(a,a)` is true, so the first instance is true. `q(b,b)` is false, but so is `p(b)` so the second instance is true as well. Since both instances are true, the original quantified sentence is true.

Now let's consider a case with nested quantifiers. Is `AX:EY:q(X,Y)` true or false for the truth assignment shown above? As before, we know that this sentence is true if every instance of its scope is true. The two possible instances are shown below.

```
EY:q(a,Y)
EY:q(b,Y)
```

To determine the truth of the first of these existential sentences, we must find at least one instance of the scope that is true. The possibilities are shown below. Of these, the first is true; and so the first existential sentence is true.

```
q(a,a)
q(a,b)
```

Now, we do the same for the second existentially quantified. The possible instances follow. Of these, again the first is true; and so the second existential sentence is true.

```
q(b,a)
q(b,b)
```

Since both existential sentences are true, the original universally quantified sentence must be true as well.

We say that a truth assignment *satisfies* a sentence with free variables if and only if it satisfies every instance of that sentence. A truth assignment *satisfies* a set of sentences if and only if it satisfies every sentence in the set.

19.3 Incomplete Game Description Language

IGDL is not so much a language as a family of languages. It has multiple dialects - one for each of the various dialects of GDL. There is IGDL corresponding to GDL; there is IGDL-II, corresponding to GDL-II; and there is ISDL, corresponding to SDL.

Given one of these dialects of GDL, the IGDL variant is obtained using the language of Relational Logic in place of the language of Logic Programming. In other words, instead of writing rules, one writes logical sentences. Everything else remains the same.

We make this more concrete in the following sections by looking at various descriptions of a single game. First, we look at Buttons and Lights written in GDL. We then look at a complete description of the game written in IGDL-II. And then we look at an incomplete description written in IGDL-II. Finally, we discuss game management and game play with incomplete descriptions.

19.4 Buttons and Lights Revisited

In this section, we return to the game of Buttons and Lights. Recall that, in ordinary Buttons and Lights, there are three base propositions (the lights) and three actions (the buttons). See below. Pushing the first button in each group toggles the first light; pushing the second button in each group interchanges the first and second lights; and pushing the third button in each group interchanges the second and third lights. Initially, the lights are all off. The goal is to turn on all of the lights. The game terminates on step 7 (after 6 moves).

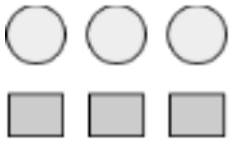

Figure 11.1 - Buttons and Lights

The ordinary GDL for this game is shown below. There is just one role, here called `robot`. There are three base propositions, one percept, three actions, and seven steps (with the usual successor relation). The robot can see proposition `q` whenever it is true. All three actions are legal in all states. The update rules, goal, and termination are as just described.

```
role(robot)

base(p)
base(q)
base(r)

percept(robot,q)

input(robot,a)
input(robot,b)
input(robot,c)

step(1)
step(2)
step(3)
step(4)
step(5)
step(6)
step(7)

successor(1,2)
successor(2,3)
successor(3,4)
successor(4,5)
successor(5,6)
successor(6,7)

sees(q) :- true(q)

legal(robot,a)
legal(robot,b)
legal(robot,c)

next(p) :- does(robot,a) & ~true(p)
next(p) :- does(robot,b) & true(q)
next(p) :- does(robot,c) & true(p)
next(q) :- does(robot,a) & true(q)
next(q) :- does(robot,b) & true(p)
next(q) :- does(robot,c) & true(r)
next(r) :- does(robot,a) & true(r)
next(r) :- does(robot,b) & true(r)
next(r) :- does(robot,c) & true(q)

goal(robot,100) :- true(p) & true(q) & true(r)
goal(robot,0) :- ~true(p)
goal(robot,0) :- ~true(q)
goal(robot,0) :- ~true(r)

terminal :- true(7)
```

In playing games with incomplete descriptions, the game manager starts with a complete description, typically encoded in GDL, like the one shown here. However, the players are given only partial descriptions. These partial descriptions are written in IGDL or IGDL-II or ISDL. In

the next section, we look at a complete description of Buttons and Lights written in IGDL-II; and in the section after that we look at an incomplete description.

19.5 Complete Description of Buttons and Lights

The first step in writing an IGDL-II description is the same as for GDL or GDL-II or SDL - we enumerate the structural components of the game - roles, base propositions, percepts, actions, and steps. In almost all cases, these descriptions are complete, as in this case.

The IGDL-II description of these structural components is shown below. The main difference here is that we have written negative sentences for each relation to tell us what is not true. In Logic Programming, this is not necessary, since anything that is not known to be true there is assumed to be false. In incomplete descriptions, we cannot make this assumption - If we do not know something, it is not necessarily false; we just do not know whether it is true or false. So, we need to be explicit about the things we know to be false.

```
role(robot)
X!=robot => ~role(X)

base(p)
base(q)
base(r)
X!=p & X!=q & X!=r => ~base(X)

percept(robot,q)
X!=robot | Y!=q => ~percept(X,Y)

input(robot,a)
input(robot,b)
input(robot,c)
X!=robot | Y!=a & Y!=b & Y!=c => ~input(X,Y)

step(1)
step(2)
step(3)
step(4)
step(5)
step(6)
step(7)
X!=1 & X!=2 & X!=3 & X!=4 & X!=5 & X!=6 & X!=7 => ~step(X)

successor(1,2)
successor(2,3)
successor(3,4)
successor(4,5)
successor(5,6)
successor(6,7)
(X!=1 | Y!2) & ... & (X!=6 | Y!7) => ~successor(X,Y)
```

The other components of the game description can be formalized in the same way. See below. All lights are off in the initial state. The player can see its single percept. All three actions are legal. The update rules are the same. And the goal and termination rules are the same.

```
~init(X)

true(q) <=> sees(q)
X!=q => ~sees(X)

legal(robot,a)
input(robot,b)
input(robot,c)
X!=robot | Y!=a & Y!=b & Y!=c => ~legal(X,Y)

next(p) <=>
  does(robot,a) & ~true(p) |
```

```
  does(robot,b) & true(q) |
  does(robot,c) & true(p)

next(q) <=>
  does(robot,a) & true(q) |
  does(robot,b) & true(p) |
  does(robot,c) & true(r)

next(r) <=>
  does(robot,a) & true(r) |
  does(robot,b) & true(r) |
  does(robot,c) & true(q)

X!=p & X!=q & X!=r => ~next(X)

goal(robot,100) <=> true(p) & true(q) & true(r)
goal(robot,0) <=> ~true(p) | ~true(q) | ~true(r)
X!=robot | (Y!100 & Y!=0) => ~goal(X,Y)

terminal <=> true(7)
```

The sentences here are a little different from those in the GDL description. However, with a little reflection on the semantics of GDL-II and IGDL-II, it is easy to see that they describe exactly the same game.

19.6 Incomplete Description of Buttons and Lights

As a simple example of an *incomplete* description written in IGDL-II, consider a simple variation of Buttons and Lights in which the players know everything except the initial state of the game.

To be more precise, we take all of the sentences from Section 19.5, with the exception of the sentence asserting that none of the base propositions are true in the initial state. Removing this sentence means that the player starts the game in any one of eight possible states.

Of course, once the player gets a percept, i.e. it sees whether `q` is true or false, this ambiguity is cut down to just four states. Unfortunately, in this situation there is still no guaranteed solution in the allowed number of steps. There are just too many cases to consider.

We can make things a little better by giving the player a bit more information. Let's say we tell the player that proposition `p` and proposition `q` have the same initial value. This can be done by augmenting the description with a sentence like the one shown below.

```
init(p) <=> init(q)
```

With this additional information, the player knows that the game starts in one of four possible states. Once it is given its initial percept, it can cut this down to just two initial states. And, by clever play, it can then solve the problem despite the residual ambiguity, as described in the next section.

19.7 Playing Buttons and Lights With an Incomplete Description

Game management and play with incomplete descriptions is a little different from management and play with complete information.

First of all, the manager and the players typically have different descriptions of the game. The manager has a complete description, like the one in Sections 19.4 and 19.5. This is necessary so that it can simulate the game accurately. However, the players have only partial descriptions, like the one in Section 19.6. In some cases, it is even possible for different players to have different descriptions.

Second, our usual techniques for game play do not necessarily work. For example, with incomplete descriptions, the players may not know the initial state exactly (as is the case in the description of Section 19.6). Or they may know the initial state but not be able to determine a unique successor state, given limited information about the update rules for the game. They may not even know in all cases whether the game is over.

Dealing with limitations of these sorts means that players must keep open multiple options on state (as in playing games in IGDL-II). Also, they must use the description in new and interesting ways to extract as much information from the description as possible, possibly combining from multiple time steps.

To illustrate these ideas, let's consider the sort of computation necessary in the context of the incomplete game description given in Section 19.6.

When the game begins, the player is given its percept and learns that `q` is false. From this and the partial constraint on `init`, it knows that `p` must also be false. It still does know whether `r` is true or false, so there are two possible states to consider. See below.

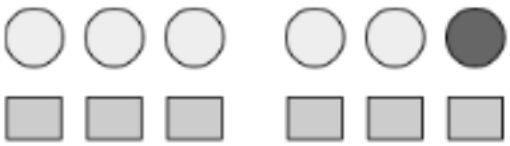

Figure 11.2 - Possible states of Buttons and Lights on step 1

There are several ways to proceed in this situation. In one approach, the player presses the `a` button to make `p` true, and then it presses the `b` button to interchange `p` and `q`. This makes `p` false and `q` true.

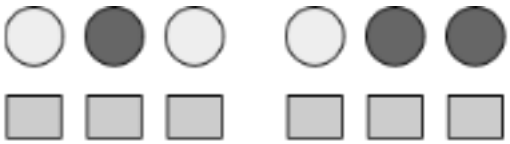

Figure 11.3 - Possible states of Buttons and Lights on step 3

On the third step, it presses the `c` button to interchange `q` and `r`. This makes `r` true and gives `q` the value of `r`.

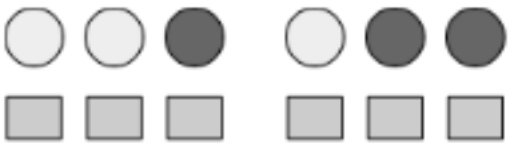

Figure 11.4 - Possible states of Buttons and Lights on step 4

Now, if `q` is false after this operation, then the player knows that `r` was false in the initial state. In this case, it needs to press `a` again, followed by `b`, followed by `a` to get all three lights on.

If `q` is true after interchanging with `r`, then the player just needs to press `a` to make `p` true, at which point all three lights are on. It can then press `b` or `c` a couple of times to get to step 7.

The challenge in building players for IGDL descriptions is making them able to use constraints that can help it to infer as much information about the state as possible from their perceptual inputs and the known constraints. In some cases, like this one, the job is easy. In other cases, especially with ISDL, the constraints may involve reasoning about multiple time steps, and the process can be extremely complex.