CHAPTER 6

Relational Logic

6.1 Introduction

Propositional Logic does a good job of allowing us to talk about relationships among individual propositions, and it gives us the machinery to derive logical conclusions based on these relationships. Suppose, for example, we believe that, if Jack knows Jill, then Jill knows Jack. Suppose we also believe that Jack knows Jill. From these two facts, we can conclude that Jill knows Jack using a simple application of Modus Ponens.

Unfortunately, when we want to say things more generally, we find that Propositional Logic is inadequate. Suppose, for example, that we wanted to say that, in general, if one person knows a second person, then the second person knows the first. Suppose, as before, that we believe that Jack knows Jill. How do we express the general fact in a way that allows us to conclude that Jill knows Jack? Here, Propositional Logic is inadequate; it gives us no way of encoding this more general belief in a form that captures its full meaning and allows us to derive such conclusions.

Relational Logic is an extension of Propositional Logic that solves this problem. The trick is to augment our language with two new linguistic features, viz. *variables* and *quantifiers*. With these new features, we can express information about multiple objects without enumerating those objects; and we can express the existence of objects that satisfy specified conditions without saying which objects they are.

In this chapter, we proceed through the same stages as in the introduction to Propositional Logic. We start with syntax and semantics. We then discuss evaluation and satisfaction. Finally, we move on to logical entailment. As with Propositional Logic, we leave the discussion of proofs to later chapters.

6.2 Syntax

In Propositional Logic, sentences are constructed from a basic vocabulary of propositional constants. In Relational Logic, there are no propositional constants; instead we have *object constants*, *function constants*, *relation constants*, and *variables*.

In our examples here, we write both variables and constants as strings of letters, digits, and a few non-alphanumeric characters (e.g. "_"). By convention, variables begin with letters from the end of the alphabet (viz. *u*, *v*, *w*, *x*, *y*, *z*). Examples include *x*, *ya*, and *z*_2. By convention, all constants begin with either alphabetic letters (other than *u*, *v*, *w*, *x*, *y*, *z*) or digits. Examples include *a*, *b*, 123, *comp*225, and *barack_obama*.

Note that there is no distinction in spelling between object constants, function constants, and relation constants. The type of each such word is determined by its usage or, in some cases, in an explicit specification.

As we shall see, function constants and relation constants are used in forming complex expressions by combining them with an appropriate number of arguments. Accordingly, each function constant and relation constant has an associated *arity*, i.e. the number of arguments with which that function constant or relation constant can be combined. A function constant or relation constant that can combined with a single argument is said to be *unary*; one that can be combined with two arguments is said to be *binary*; one that can be combined with three arguments is said to be *ternary*; more generally, a function or relation constant that can be combined with n arguments is said to be n-ary.

A *signature* consists of a set of object constants, a set of function constants, a set of relation constants, and an assignment of arities for each of the function constants and relation constants in the signature. (Note that this definition here is slightly non-traditional. In many textbooks, a signature includes a specification of function constants and relation constants but not object constants, whereas our definition here includes are three types of constants.)

A *term* is defined to be a variable, an object constant, or a functional expression (as defined below). Terms typically denote objects presumed or hypothesized to exist in the world; and, as such, they are analogous to noun phrases in natural language, e.g. *Joe* or *the car's left front wheel*.

A *functional expression*, or *functional term*, is an expression formed from an n-ary function constant and n terms enclosed in parentheses and separated by commas. For example, if f is a binary function constant and if a and y are terms, then $f(a,y)$ is a functional expression, as are $f(a,a)$ and $f(y,y)$.

Note that functional expressions can be nested within other functional expressions. For example, if g is a unary function constant and if a is a term, $g(a)$ and $g(g(a))$ and $g(g(g(a)))$ are all functional expressions.

There are three types of *sentences* in Relational Logic, viz. relational sentences (the analog of propositions in Propositional Logic), logical sentences (analogous to the logical sentences in Propositional Logic), and quantified sentences (which have no analog in Propositional Logic).

A *relational sentence* is an expression formed from an n-ary relation constant and n terms. For example, if q is a relation constant with arity 2 and if a and y are terms, then the expression shown below is a syntactically legal relational sentence.

$$q(a, y)$$

Logical sentences are defined as in Propositional Logic. There are negations, conjunctions, disjunctions, implications, and equivalences. The syntax is exactly the same, except that the elementary components are relational sentences and equations rather than proposition constants.

Quantified sentences are 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.

$$(\forall x.(p(x) \Rightarrow q(x,x)))$$

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.

$$(\exists x.(p(x) \land q(x,x)))$$

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.

$$(\forall x.p(x)) \lor (\exists x.q(x,x))$$
$$(\forall x.(\exists x.q(x,y)))$$

As with Propositional Logic, we can drop unneeded parentheses in Relational Logic, relying on precedence to disambiguate the structure of unparenthesized sentences. In Relational Logic, the precedence relations of the logical operators are the same as in Propositional Logic, and quantifiers have higher precedence than logical operators.

The following examples show how to parenthesize sentences with both quantifiers and logical operators. The sentences on the right are partially parenthesized versions of the sentences on the left. (To be fully parenthesized, we would need to add parentheses around each of the sentences as a whole.)

$\forall x.p(x) \Rightarrow q(x)$	$(\forall x.p(x)) \Rightarrow q(x)$
$\exists x.p(x) \land q(x)$	$(\exists x.p(x)) \land q(x)$

Notice that, in each of these examples, the quantifier does *not* apply to the second relational sentence, even though, in each case, that sentence contains an occurrence of the variable being quantified. If we want to apply the quantifier to a logical sentence, we must enclose that sentence in parentheses, as in the following examples.

$$\forall x.(p(x) \Rightarrow q(x))$$
$$\exists x.(p(x) \land q(x))$$

An expression in the language of Relational Logic is *ground* if and only if it contains no variables. For example, the sentence $p(a)$ is ground, whereas the sentence $\forall x.p(x)$ is not.

An occurrence of a variable is *free* if and only if it is not in the scope of a quantifier of that variable. Otherwise, it is *bound*. For example, *y* is free and *x* is bound in the following sentence.

$$\exists x.q(x,y)$$

A sentence is *open* if and only if it has free variables. Otherwise, it is *closed*. For example, the first sentence below is open and the second is closed.

$$p(y) \Rightarrow \exists x.q(x,y)$$
$$\forall y.(p(y) \Rightarrow \exists x.q(x,y))$$

6.3 Semantics

The *Herbrand base* for a relational signature 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.

For a signature with object constants `a` and `b`, no function constants, and relation constants `p` and `q` where `p` has arity 1 and `q` has arity 2, the Herbrand base is shown below.

$$\{p(a), p(b), q(a,a), q(a,b), q(b,a), q(b,b)\}$$

It is worthwhile to note that, for a given relation constant and a finite set of terms, there is an upper bound on the number of ground relational sentences that can be formed using that relation constant. In particular, for a set of terms of size *b*, there are b^n distinct *n*-tuples of object constants; and hence there are b^n ground relational sentences for each *n*-ary relation constant. Since the number of relation constants in a signature is finite, this means that the Herbrand base is also finite.

Of course, not all Herbrand bases are finite. In the presence of function constants, the number of ground terms is infinite; and so the Herbrand base is infinite. For example, in a language with a single object constant *a* and a single unary function constant *f* and a single unary relation constant *p*, the Herbrand base consists of the sentences shown below.

$$\{p(a), p(f(a)), p(f(f(a))), ...\}$$

A *truth assignment* for a Relational Logic language is a function that maps each ground relational sentence in the Herbrand base to a truth value. As in Propositional Logic, we use the digit 1 as a synonym for true and 0 as a synonym for false; and we refer to the value assigned to a ground relational sentence by writing the relational sentence with the name of the truth assignment as a superscript. For example, the truth assignment *i* defined below is an example for the case of the language mentioned a few paragraphs above.

$$\begin{aligned}&p(a)^i = 1\\&p(b)^i = 0\\&q(a,a)^i = 1\\&q(a,b)^i = 0\\&q(b,a)^i = 1\\&q(b,b)^i = 0\end{aligned}$$

As with Propositional Logic, 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.

The rules for logical sentences in Relational Logic are the same as those for logical sentences in Propositional Logic. A truth assignment *i* satisfies a negation $\neg\phi$ if and only if *i* does not satisfy ϕ. Truth assignment *i* satisfies a conjunction $(\phi_1 \wedge ... \wedge \phi_n)$ if and only if *i* satisfies every ϕ_i. Truth assignment *i* satisfies a disjunction $(\phi_1 \vee ... \vee \phi_n)$ if and only if *i* satisfies at least one ϕ_i. Truth assignment *i* satisfies an implication $(\phi \Rightarrow \psi)$ if and only if *i* does not satisfy ϕ or does satisfy ψ. Truth assignment *i* satisfies an equivalence $(\phi \Leftrightarrow \psi)$ if and only if *i* 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 $\forall x.(p(x) \Rightarrow q(x,x))$. What is the truth value under the truth assignment shown above? According to our definition, a universally quantified sentence is true if and only every instance of its scope is true. For this language, there are just two instances. See below.

$$p(a) \Rightarrow q(a,a)$$
$$p(b) \Rightarrow q(b,b)$$

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 $\forall x.\exists y.q(x,y)$ 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.

$$\exists y.q(a,y)$$
$$\exists y.q(b,y)$$

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.

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

6.4 Example - Sorority World

Consider once again the Sorority World example introduced in Chapter 1. Recall that this world focusses on the interpersonal relations of a small sorority. There are just four members - Abby, Bess, Cody, and Dana. Our goal is to represent information about who likes whom.

In order to encode this information in Relational Logic, we adopt a signature with four object constants (*abby*, *bess*, *cody*, *dana*) and one binary relation constant (*likes*).

If we had complete information about the likes and dislikes of the girls, we could completely characterize the state of affairs as a set of ground relational sentences or negations of ground relational sentences, like the ones shown below, with one sentence for each member of the Herbrand base. (In our example here, we have written the positive literals in black and the negative literals in grey in order to distinguish the two more easily.)

$\neg likes(abby,abby)$	$\neg likes(abby,bess)$	$likes(abby,cody)$	$\neg likes(abby,dana)$
$\neg likes(bess,abby)$	$\neg likes(bess,bess)$	$likes(bess,cody)$	$\neg likes(bess,dana)$
$likes(cody,abby)$	$likes(cody,bess)$	$\neg likes(cody,cody)$	$likes(cody,dana)$
$\neg likes(dana,abby)$	$\neg likes(dana,bess)$	$likes(dana,cody)$	$\neg likes(dana,dana)$

To make things more interesting, let's assume that we do *not* have complete information, only fragments of information about the girls likes and dislikes. Let's see how we can encode such fragments in the language of Relational Logic.

Let's start with a simple disjunction. *Bess likes Cody or Dana*. Encoding a sentence with a disjunctive noun phrase (such as *Cody or Dana*) is facilitated by first rewriting the sentence as a disjunction of simple sentences. *Bess likes Cody or Bess likes Dana*. In Relational Logic, we can express this fact as a simple disjunction with the two possibilities as disjuncts.

$$likes(bess,cody) \vee likes(bess,dana)$$

Abby likes everyone Bess likes. Again, paraphrasing helps translate. *If Bess likes a girl, then Abby also likes her*. Since this is a fact about everyone, we use a universal quantifier.

$$\forall y.(likes(bess,y) \Rightarrow likes(abby,y))$$

Cody likes everyone who likes her. In other words, *if some girl likes Cody, then Cody likes that girl*. Again, we use a universal quantifier.

$$\forall y.(likes(y,cody) \Rightarrow likes(cody,y))$$

Bess likes somebody who likes her. The word *somebody* here is a tip-off that we need to use an existential quantifier.

$$\exists y.(likes(bess,y) \wedge likes(y,bess))$$

Nobody likes herself. The use of the word *nobody* here suggests a negation. A good technique in such cases is to rewrite the English sentence as the negation of a positive version of the sentence before translating to Relational Logic.

$$\neg \exists x.likes(x,x)$$

Everybody likes somebody. Here we have a case requiring two quantifiers, one universal and one existential. The key to this case is getting the quantifiers in the right order. Reversing them leads to a very different statement.

$$\forall x.\exists y.likes(x,y)$$

There is someone everyone likes. The preceding sentence tells us that everyone likes someone, but that someone can be different for different people. This sentence tells us that everybody likes the same person.

$$\exists x.\forall y.likes(x,y)$$

6.5 Example - Blocks World

The Blocks World is a popular application area for illustrating ideas in the field of Artificial Intelligence. A typical Blocks World scene is shown in Figure 1.

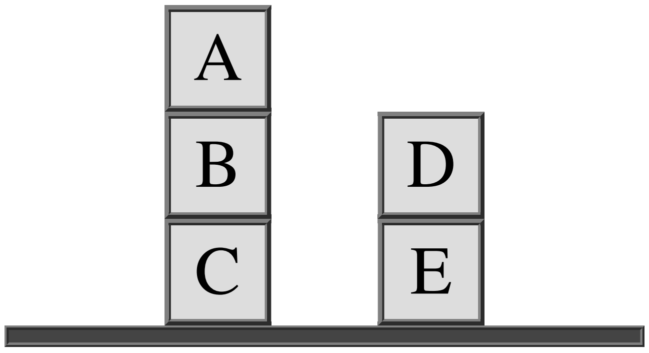

Figure 1 - One state of Blocks World.

Most people looking at this figure interpret it as a configuration of five toy blocks. Some people conceptualize the table on which the blocks are resting as an object as well; but, for simplicity, we ignore it here.

In order to describe this scene, we adopt a signature with five object constants, as shown below, with one object constant for each of the five blocks in the scene. The intent here is for each of these object constants to represent the block marked with the corresponding capital letter in the scene.

$$\{a, b, c, d, e\}$$

In a spatial conceptualization of the Blocks World, there are numerous meaningful relations. For example, it makes sense to think about the the relation that holds between two blocks if and only if one is resting on the other. In what follows, we use the relation constant *on* to refer to this relation. We might also think about the relation that holds between two blocks if and only if one is anywhere above the other, i.e. the first is resting on the second or is resting on a block that is resting on the second, and so forth. In what follows, we use the relation constant *above* to talk about this relation. There is the relation that holds of three blocks that are stacked one on top of the other. We use the relation constant *stack* as a name for this relation. We use the relation constant *clear* to denote the relation that holds of a block if and only if there is no block on top of it. We use the relation constant *table* to denote the relation that holds of a block if and only if that block is resting on the table. The signature corresponding to this conceptualization is shown below.

$$\{on, above, stack, clear, table\}$$

The arities of these relation constants are determined by their intended use. Since *on* is intended to denote a relation between two blocks, it has arity 2. Similarly, *above* has arity 2. The *stack* relation constant has arity 3. Relation constants *clear* and *table* each have arity 1.

Given this signature, we can describe the scene in Figure 1 by writing ground literals that state which relations hold of which objects or groups of objects. Let's start with *on*. The following sentences tell us directly for each ground relational sentence whether it is true or false. (Once again, we have written the positive literals in black and the negative literals in grey in order to distinguish the two more easily.)

$\neg on(a,a)$	$on(a,b)$	$\neg on(a,c)$	$\neg on(a,d)$	$\neg on(a,e)$
$\neg on(b,a)$	$\neg on(b,b)$	$on(b,c)$	$\neg on(b,d)$	$\neg on(b,e)$
$\neg on(c,a)$	$\neg on(c,b)$	$\neg on(c,c)$	$\neg on(c,d)$	$\neg on(c,e)$
$\neg on(d,a)$	$\neg on(d,b)$	$\neg on(d,c)$	$\neg on(d,d)$	$on(d,e)$
$\neg on(e,a)$	$\neg on(e,b)$	$\neg on(e,c)$	$\neg on(e,d)$	$\neg on(e,e)$

We can do the same for the other relations. However, there is an easier way. Each of the remaining relations can be defined in terms of *on*. These definitions together with our facts about the *on* relation logically entail every other ground relational sentence or its negation. Hence, given these definitions, we do not need to write out any additional data.

A block satisfies the *clear* relation if and only if there is nothing on it.

$$\forall y.(clear(y) \Leftrightarrow \neg\exists x.on(x,y))$$

A block satisfies the *table* relation if and only if it is not on some block.

$$\forall x.(table(x) \Leftrightarrow \neg\exists y.on(x,y))$$

Three blocks satisfy the *stack* relation if and only if the first is on the second and the second is on the third.

$$\forall x.\forall y.\forall z.(stack(x,y,z) \Leftrightarrow on(x,y) \wedge on(y,z))$$

The *above* relation is a bit tricky to define correctly. One block is above another block if and only if the first block is on the second block or it is on another block that is above the second black. Also, no block can be above itself. Given a complete definition for the *on* relation, these two axioms determine a unique *above* relation.

$$\forall x.\forall z.(above(x,z) \Leftrightarrow on(x,z) \vee \exists y.(on(x,y) \wedge above(y,z))$$

$$\neg\forall x.above(x,x)$$

One advantage to defining relations in terms of other relations is economy. If we record *on* information for every object and encode the relationship between the *on* relation and the these other relations, there is no need to record any ground relational sentences for those relations.

Another advantage is that these general sentences apply to Blocks World scenes other than the one pictured here. It is possible to create a Blocks World scene in which none of the specific sentences we have listed is true, but these definitions are still correct.

6.6 Example - Modular Arithmetic

In this example, we show how to characterize Modular Arithmetic in the language of Relational Logic. In Modular Arithmetic, there are only finitely many objects. For example, in Modular Arithmetic with modulus 4, we would have just four integers - 0, 1, 2, 3 - and that's all. Our goal here to define the addition relation. Admittedly, this is a modest goal; but, once we see how to do this; we can use the same approach to define other arithmetic relations.

Let's start with the *same* relation, which is true of every number and itself and is false for numbers that are different. We can completely characterize the *same* relation by writing ground relational sentences, one positive sentence for each number and itself and negative sentences for all of the other cases.

same(*0*,*0*)	¬*same*(*0*,*1*)	¬*same*(*0*,*2*)	¬*same*(*0*,*3*)
¬*same*(*1*,*0*)	*same*(*1*,*1*)	¬*same*(*1*,*2*)	¬*same*(*1*,*3*)
¬*same*(*2*,*0*)	¬*same*(*2*,*1*)	*same*(*2*,*2*)	¬*same*(*2*,*3*)
¬*same*(*3*,*0*)	¬*same*(*3*,*1*)	¬*same*(*3*,*2*)	*same*(*3*,*3*)

Now, let's axiomatize the *next* relation, which, for each number, gives the next larger number, wrapping back to 0 after we reach 3.

next(0,1)
next(1,2)
next(2,3)
next(3,0)

Properly, we should write out the negative literals as well. However, we can save that work by writing a single axiom asserting that *next* is a functional relation, i.e., for each member of the Herbrand base, there is just one successor.

$$\forall x.\forall y.\forall z.(next(x,y) \wedge next(x,z) \Rightarrow same(y,z)$$

In order to see why this saves us the work of writing out the negative literals, we can write this axiom in the equivalent form shown below.

$$\forall x.\forall y.\forall z.(next(x,y) \wedge \neg same(y,z) \Rightarrow \neg next(x,z))$$

The addition table for Modular Arithmetic is the usual addition table for arbitrary numbers except that we wrap around whenever we get past 3. For such a small arithmetic, it is easy to write out the ground relational sentences for addition, as shown below.

plus(0,0,0)	*plus*(1,0,1)	*plus*(2,0,2)	*plus*(3,0,3)
plus(0,1,1)	*plus*(1,1,2)	*plus*(2,1,3)	*plus*(3,1,0)
plus(0,2,2)	*plus*(1,2,3)	*plus*(2,2,0)	*plus*(3,2,1)
plus(0,3,3)	*plus*(1,3,0)	*plus*(2,3,1)	*plus*(3,3,2)

As with *next*, we avoid writing out the negative literals by writing a suitable functionality axiom for *plus*.

$$\forall x.\forall y.\forall z.\forall w.(plus(x,y,z) \land \neg same(z,w) \Rightarrow \neg plus(x,y,w))$$

That's one way to do things, but we can do better. Rather than writing out all of those relational sentences, we can use the language of Relational Logic to define *plus* in terms of *same* and *next* and use that axiomatization to deduce the ground relational sentences. The definition is shown below. First, we have an identity axiom. Adding 0 to any number results in the same number. Second we have a successor axiom. If *z* is the sum of *x* and *y*, then the sum of the successor of *x* and *y* is the successor of *z*. Finally, we have our functionality axiom once again.

$$\forall y.plus(0,y,y)$$
$$\forall x.\forall y.\forall z.\forall x2.\forall z2.(plus(x,y,z) \land next(x,x2) \land next(z,z2) \Rightarrow plus(x2,y,z2))$$
$$\forall x.\forall y.\forall z.\forall w.(plus(x,y,z) \land \neg same(z,w) \Rightarrow \neg plus(x,y,w))$$

One advantage of doing things this way is economy. With these sentences, we do not need the ground relational sentences about *plus* given above. They are all logically entailed by our sentences about *next* and the definitional sentences. A second advantage is versatility. Our sentences define *plus* in terms of *next* for arithmetic with any modulus, not just modulus 4.

6.7 Example - Peano Arithmetic

Now, let's look at the problem of encoding arithmetic for all of the natural numbers. Since there are infinitely many natural numbers, we need infinitely many terms.

One way to get infinitely many terms is to have a signature with infinitely many object constants. While there is nothing wrong with this in principle, it makes the job of axiomatizing arithmetic effectively impossible, as we would have to write out infinitely many literals to capture our successor relation.

An alternative approach is to represent numbers using a single object constant (e.g. 0) and a single unary function constant (e.g. *s*). We can then represent every number *n* by applying the function constant to 0 exactly *n* times. In this encoding, *s*(0) represents 1; *s*(*s*(0)) represents 2; and so forth. With this encoding, we automatically get an infinite universe of terms, and we can write axioms defining addition and multiplication as simple variations on the axioms of Modular Arithmetic.

Unfortunately, even with this representation, axiomatizing Peano Arithmetic is more challenging than axiomatizing Modular Arithmetic. We cannot just write out ground relational sentences to characterize our relations, because there are infinitely many cases to consider. For Peano Arithmetic, we must rely on logical sentences and quantified sentences, not just because they are more economical but because they are the only way we can characterize our relations in finite space.

Let's look at the same relation first. The axioms shown here define the *same* relation in terms of 0 and *s*.

$$\forall x.same(x,x)$$
$$\forall x.(\neg same(0,x) \land \neg same(x,0))$$
$$\forall x.\forall y.(same(x,y) \Rightarrow same(s(x),s(y)))$$

It is easy to see that these axioms completely characterize the *same* relation. By the first axiom, the same relation holds of every term and itself. The other two axioms tell us what is not true.

$$\begin{array}{l} \mathit{same}(0,0) \\ \mathit{same}(s(0),s(0)) \\ \mathit{same}(s(s(0)),s(s(0))) \\ \quad\quad\ldots \end{array}$$

The second axiom tells us that 0 is not the same as any composite term. The same hold true with the arguments reversed.

$$\begin{array}{ll} \neg\mathit{same}(0,s(0)) & \neg\mathit{same}(s(0),0) \\ \neg\mathit{same}(0,s(s(0))) & \neg\mathit{same}(s(s(0)),0) \\ \neg\mathit{same}(0,s(s(s(0)))) & \neg\mathit{same}(s(s(s(0))),0) \\ \quad\quad\ldots & \quad\quad\ldots \end{array}$$

The third axiom builds on these results to show that non-identical composite terms of arbitrary complexity do not satisfy the same relation. Viewed the other way around, to see that two non-identical terms are not the same, we just strip away occurrences of s from each term till one of the two terms becomes 0 and the other one is not 0. By the second axiom, these are not the same, and so the original terms are not the same.

$$\begin{array}{ll} \mathit{same}(s(0),s(s(0))) & \neg\mathit{same}(s(s(0)),s(0)) \\ \mathit{same}(s(0),s(s(s(0)))) & \neg\mathit{same}(s(s(s(0))),s(0)) \\ \mathit{same}(s(0),s(s(s(s(0))))) & \neg\mathit{same}(s(s(s(s(0)))),s(0)) \\ \quad\quad\ldots & \quad\quad\ldots \end{array}$$

Once we have the *same* relation, we can define the other relations in our arithmetic. The following axioms define the plus relation in terms of 0, s, and *same*. Adding 0 to any number results in that number. If adding a number x to a number y produces a number z, then adding the successor of x to y produces the successor of z. Finally, we have a functionality axiom for *plus*.

$$\begin{array}{l} \forall y.\mathit{plus}(0,y,y) \\ \forall x.\forall y.\forall z.(\mathit{plus}(x,y,z) \Rightarrow \mathit{plus}(s(x),y,s(z))) \\ \forall x.\forall y.\forall z.\forall w.(\mathit{plus}(x,y,z) \wedge \neg\mathit{same}(z,w) \Rightarrow \neg\mathit{plus}(x,y,w)) \end{array}$$

The axiomatization of multiplication is analogous. Multiplying any number by 0 produces 0. If a number z is the product of x and y and w is the sum of y and z, then w is the product of the successor of x and y. As before, we have a functionality axiom.

$$\begin{array}{l} \forall y.\mathit{times}(0,y,0) \\ \forall x.\forall y.\forall z.\forall w.(\mathit{times}(x,y,z) \wedge \mathit{plus}(y,z,w) \Rightarrow \mathit{times}(s(x),y,w)) \\ \forall x.\forall y.\forall z.\forall w.(\mathit{times}(x,y,z) \wedge \neg\mathit{same}(zw) \Rightarrow \neg\mathit{times}(x,y,w)) \end{array}$$

That's all we need - just three axioms for *same* and three axioms for each arithmetic function.

Before we leave our discussion of Peano aritmetic, it is worthwhile to look at the concept of Diophantine equations. A *polynomial equation* is a sentence composed using only addition, multiplication, and exponentiation with fixed exponents (that is numbers not variables). For example, the expression shown below in traditional math notation is a polynomial equation.

$$x^2 + 2y = 4z$$

A *natural Diophantine equation* is a polynomial equation in which the variables are restricted to the natural numbers. For example, the polynomial equation here is also a Diophantine equation and happens to have a solution in the natural numbers, viz. x=4 and y=8 and z=8.

Diophantine equations can be readily expressed as sentences In Peano Arithmetic. For example, we can represent the Diophantine equation above with the sentence shown below.

$$\forall x.\forall y.\forall z.\forall u.\forall v.\forall w.(times(x,x,u) \wedge times(2,y,v) \wedge plus(u,v,w) \Rightarrow times(4,z,w))$$

This is a little messy, but it is doable. And we can always clean things up by adding a little syntactic sugar to our language to make it look like traditional math notation.

Once this mapping is done, we can use the tools of logic to work with these sentences. In some cases, we can find solutions; and, in some cases, we can prove that solutions do not exist. This has practical value in some situations, but it also has significant theoretical value in establishing important properties of Relational Logic, a topic that we discuss in a later section.

6.8 Example - Linked Lists

A list is finite sequence of objects. Lists can be flat, e.g. [a, b, c]. Lists can also be nested within other lists, e.g. [a, [b, c], d].

A linked list is a way of representing nested lists of variable length and depth. Each element is represented by a cell containing a value and a pointer to the remainder of the list. Our goal in this example is to formalize linked lists and define some useful relations.

To talk about lists of arbitrary length and depth, we use the binary function constant *cons*, and we use the object constant *nil* to refer to the empty list. In particular, a term of the form $cons(\tau_1, \tau_2)$ designates a sequence in which τ_1 denotes the first element and τ_2 denotes the rest of the list. With this function constant, we can encode the list [a, b, c] as follows.

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

The advantage of this representation is that it allows us to describe functions and relations on lists without regard to length or depth.

As an example, consider the definition of the binary relation *member*, which holds of an object and a list if the object is a top-level member of the list. Using the function constant *cons*, we can characterize the *member* relation as shown below. Obviously, an object is a member of a list if it is the first element; however, it is also a member if it is member of the rest of the list.

$$\forall x.\forall y.member(x, cons(x, y))$$
$$\forall x.\forall y.\forall z.(member(x, z) \Rightarrow member(x, cons(y, z)))$$

We also can define functions to manipulate lists in different ways. For example, the following axioms define a relation called *append*. The value of *append* (its last argument) is a list consisting of the elements in the list supplied as its first argument followed by the elements in the list supplied as its second. For example, we would have $append(cons(a,nil), cons(b, cons(c, nil)), cons(a, cons(b, cons(c, nil))))$.

$$\forall z.append(nil, z, z)$$
$$\forall x.\forall y.\forall z.(append(y, z, w) \Rightarrow append(cons(x, y), z, cons(x,w)))$$

We can also define relations that depend on the structure of the elements of a list. For example, the *among* relation is true of an object and a list if the object is a member of the list, if it is a member of a list that is itself a member of the list, and so on.

$$\forall x.among(x, x)$$
$$\forall x.\forall y.\forall z.(among(x, y) \vee among(x, z) \Rightarrow among(x, cons(y, z)))$$

Lists are an extremely versatile representational device, and the reader is encouraged to become as familiar as possible with the techniques of writing definitions for functions and relations on lists. As is true of many tasks, practice is the best approach to gaining skill.

6.9 Example - Pseudo English

Pseudo English is a formal language that is intended to approximate the syntax of the English language. One way to define the syntax of Pseudo English is to write grammatical rules in Backus Naur Form (BNF). The rules shown below illustrate this approach for a small subset of Pseudo English. A sentence is a noun phrase followed by a verb phrase. A noun phrase is either a noun or two nouns separated by the word *and*. A verb phrase is a verb followed by a noun phrase. A noun is either the word *Mary* or the word *Pat* or the word *Quincy*. A verb is either *like* or *likes*.

```
<sentence> ::= <np> <vp>
<np> ::= <noun>
<np> ::= <noun> "and" <noun>
<vp> ::= <verb> <np>
<noun> ::= "mary" | "pat" | "quincy"
<verb> ::= "like" | "likes"
```

Alternatively, we can use Relational Logic to formalize the syntax of Pseudo English. The sentences shown below express the grammar described in the BNF rules above. (We have dropped the universal quantifiers here to make the rules a little more readable.) Here, we are using the *append* relation defined in the section of lists.

$$np(x) \wedge vp(y) \wedge append(x,y,z) \Rightarrow sentence(z)$$
$$noun(x) \Rightarrow np(x)$$
$$noun(x) \wedge noun(y) \wedge append(x,\text{and},z) \wedge append(z,y,w) \Rightarrow np(w)$$
$$verb(x) \wedge np(y) \wedge append(x,y,z) \Rightarrow vp(z)$$
$$noun(mary)$$
$$noun(pat)$$
$$noun(quincy)$$
$$verb(like)$$

$verb(likes)$

Using these sentences, we can test whether a given sequence of words is a syntactically legal sentence in Pseudo English and we can use our logical entailment procedures to enumerate syntactically legal sentences, like those shown below.

mary likes pat
pat and quincy like mary
mary likes pat and quincy

One weakness of our BNF and the corresponding axiomatization is that there is no concern for agreement in number between subjects and verbs. Hence, with these rules, we can get the following expressions, which in Natural English are ungrammatical.

× *mary like pat*
× *pat and quincy likes mary*

Fortunately, we can fix this problem by elaborating our rules just a bit. In particular, we add an argument to some of our relations to indicate whether the expression is singular or plural. Here, 0 means singular, and 1 means plural. We then use this to block sequences of words where the number do not agree.

$np(x,w) \wedge vp(y,w) \wedge append(x,y,z) \Rightarrow sentence(z)$
$noun(x) \Rightarrow np(x,0)$
$noun(x) \wedge noun(y) \wedge append(x,\mathrm{and},z) \wedge append(z,y,w) \Rightarrow np(w,1)$
$verb(x,w) \wedge np(y,v) \wedge append(x,y,z) \Rightarrow vp(z,w)$
$noun(mary)$
$noun(pat)$
$noun(quincy)$
$verb(like,1)$
$verb(likes,0)$

With these rules, the syntactically correct sentences shown above are still guaranteed to be sentences, but the ungrammatical sequences are blocked. Other grammatical features can be formalized in similar fashion, e.g. gender agreement in pronouns (*he* versus *she*), possessive adjectives (*his* versus *her*), reflexives (like *himself* and *herself*), and so forth.

6.10 Example - Metalevel Logic

Throughout this book, we have been writing sentences in English about sentences in Logic, and we have been writing informal proofs in English about formal proofs in Logic. A natural question to ask is whether it is possible formalize Logic within Logic. The answer is yes. The limits of what can be done are very interesting. In this section, we look at a small subset of this problem, viz. using Relational Logic to formalize information about Propositional Logic.

The first step in formalizing Propositional Logic in Relational Logic is to represent the syntactic components of Propositional Logic.

In what follows, we make each proposition constant in our Propositional Logic language an object constant in our Relational Logic formalization. For example, if our Propositional Logic language has relation constants p, q, and r, then p, q, and r are object constants in our formalization.

Next, we introduce function constants to represent constructors of complex sentences. There is one function constant for each logical operator - *not* for $\neg$, *and* for $\wedge$, *or* for $\vee$, *if* for $\Rightarrow$, and *iff* for $\Leftrightarrow$. Using these function constants, we represent Propositional Logic sentences as terms in our language. For example, we use the term $and(p,q)$ to represent the Propositional Logic sentence $(p \wedge q)$; and we use the term $if(and(p,q),r)$ to represent the Propositional Logic sentence $(p \wedge q \Rightarrow r)$.

Finally, we introduce a selection of relation constants to express the types of various expressions in our Propositional Logic language. We use the unary relation constant *proposition* to assert that an expression is a proposition. We use the the unary relation constant *negation* to assert that an expression is a negation. We use the the unary relation constant *conjunction* to assert that an expression is a conjunction. We use the the unary relation constant *disjunction* to assert that an expression is a disjunction. We use the the unary relation constant *implication* to assert that an expression is an implication. We use the the unary relation constant *biconditional* to assert that an expression is a biconditional. And we use the unary relation constant *sentence* to assert that an expression is a proposition.

With this vocabulary, we can characterize the syntax of our language as follows. We start with declarations of our proposition constants.

$$proposition(p)$$
$$proposition(q)$$
$$proposition(r)$$

Next, we define the types of expressions involving our various logical operators.

$$\forall x.(sentence(x) \Rightarrow negation(not(x)))$$
$$\forall x.(sentence(x) \wedge sentence(y) \Rightarrow conjunction(and(x,y)))$$
$$\forall x.(sentence(x) \wedge sentence(y) \Rightarrow disjunction(or(x,y)))$$
$$\forall x.(sentence(x) \wedge sentence(y) \Rightarrow implication(if(x,y)))$$
$$\forall x.(sentence(x) \wedge sentence(y) \Rightarrow biconditional(iff(x,y)))$$

Finally, we define sentences as expressions of these types.

$$\forall x.(proposition(x) \Rightarrow sentence(x))$$
$$\forall x.(negation(x) \Rightarrow sentence(x))$$
$$\forall x.(conjunction(x) \Rightarrow sentence(x))$$
$$\forall x.(disjunction(x) \Rightarrow sentence(x))$$
$$\forall x.(implication(x) \Rightarrow sentence(x))$$
$$\forall x.(biconditional(x) \Rightarrow sentence(x))$$

Note that these sentences constrain the types of various expressions but do not define them completely. For example, we have not said that $not(p)$ is *not* a conjunction. It is possible to make our definitions more complete by writing negative sentences. However, they are a little messy, and we do not need them for the purposes of this section.

With a solid characterization of syntax, we can formalize our rules of inference. We start by representing each rule of inference as a relation constant. For example, we use the ternary relation constant *ai* to represent And Introduction, and we use the binary relation constant *ae* to represent And Elimination. With this vocabulary, we can define these relations as shown below.

$$\forall x.(sentence(x) \wedge sentence(y) \Rightarrow ai(x,y,and(x,y)))$$
$$\forall x.(sentence(x) \wedge sentence(y) \Rightarrow ae(and(x,y),x))$$
$$\forall x.(sentence(x) \wedge sentence(y) \Rightarrow ae(and(x,y),y))$$

In similar fashion, we can define proofs - both linear and structured. We can even define truth assignments, satisfaction, and the properties of validity, satisfiability, and so forth. Having done all of this, we can use the proof methods discussed in the next chapters to prove our metatheorems about Propositional Logic.

We can use a similar approach to formalizing Relational Logic within Relational Logic. However, in that case, we need to be very careful. If done incorrectly, we can write paradoxical sentences, i.e. sentences that are neither true or false. For example, a careless formalization leads to formal versions of sentences like *This sentence is false*, which is self-contradictory, i.e. it cannot be true and cannot be false. Fortunately, with care it is possible to avoid such paradoxes and thereby get useful work done.

6.11 Properties of Relational Sentences

Although the languages of Propositional Logic and Relational Logic are different, many of the key concepts of the two logics are the same. Notably, the concepts of validity, contingency, unsatisfiability, and so forth for Relational Logic have the same definitions in Relational Logic as in Propositional Logic.

A sentence is *satisfiable* if and only if it is satisfied by at least one truth assignment. A sentence is *falsifiable* if and only if there is at least one truth assignment that makes it false. We have already seen several examples of satisfiable and falsifiable sentences. A sentence is *unsatisfiable* if and only if it is not satisfied by any truth assignment, i.e. no matter what truth assignment we take, the sentence is always false. A sentence is *contingent* if and only if it is both satisfiable and falsifiable, i.e. it is neither valid nor unsatisfiable.

Not only are the definitions of these concepts the same; some of the results are the same as well. If we think of ground relational sentences and ground equations as propositions, we get similar results for the two logics - a ground sentence in Relational Logic is valid / contingent / unsatisfiable if and only if the corresponding sentence in Propositional Logic is valid / contingent / unsatisfiable.

Here, for example, are Relational Logic versions of common Propositional Logic validities - the Law of the Excluded Middle, Double Negation, and deMorgan's laws for distributing negation over conjunction and disjunction.

$$p(a) \vee \neg p(a)$$
$$p(a) \Leftrightarrow \neg\neg p(a)$$
$$\neg(p(a) \wedge q(a,b)) \Leftrightarrow (\neg p(a) \vee \neg q(a,b))$$

$$\neg(p(a) \vee q(a,b)) \Leftrightarrow (\neg p(a) \wedge \neg q(a,b))$$

Of course, not all sentences in Relational Logic are ground. There are valid sentences of Relational Logic for which there are no corresponding sentences in Propositional Logic.

The *Common Quantifier Reversal* tells us that reversing quantifiers of the same type has no effect on truth assignment.

$$\forall x.\forall y.q(x,y) \Leftrightarrow \forall y.\forall x.q(x,y)$$
$$\exists x.\exists y.q(x,y) \Leftrightarrow \exists y.\exists x.q(x,y))$$

Existential Distribution tells us that it is okay to move an existential quantifier inside of a universal quantifier. (Note that the reverse is not valid, as we shall see later.)

$$\exists y.\forall x.q(x,y) \Rightarrow \forall x.\exists y.\neg q(x,y)$$

Finally, *Negation Distribution* tells us that it is okay to distribute negation over quantifiers of either type by flipping the quantifier and negating the scope of the quantified sentence.

$$\neg\forall x.p(x) \Leftrightarrow \exists x.\neg p(x)$$

6.12 Logical Entailment

A set of Relational Logic sentences Δ *logically entails* a sentence ϕ (written $\Delta \models \phi$) if and only if every truth assignment that satisfies Δ also satisfies ϕ.

As with validity and contingency and satisfiabiity, this definition is the same for Relational Logic as for Propositional Logic. As before, if we treat ground relational sentences and ground equations as propositions, we get similar results. In particular, a set of ground premises in Relational Logic logically entails a ground conclusion in Relational Logic if and only if the corresponding set of Propositional Logic premises logically entails the corresponding Propositional Logic conclusion.

For example, we have the following results. The sentence $p(a)$ logically entails $(p(a) \vee p(b))$. The sentence $p(a)$ does *not* logically entail $(p(a) \wedge p(b))$. However, any set of sentences containing both $p(a)$ and $p(b)$ does logically entail $(p(a) \wedge p(b))$.

The presence of variables allows for additional logical entailments. For example, the premise $\exists y.\forall x.q(x,y)$ logically entails the conclusion $\forall x.\exists y.q(x,y)$. If there is *some* object y that is paired with every x, then every x has some object that it pairs with, viz. y.

Here is another example. The premise $\forall x.\forall y.q(x,y)$ logically entails the conclusion $\forall x.\forall y.q(y,x)$. The first sentence says that q is true for all pairs of objects, and the second sentence says the exact same thing. In cases like this, we can interchange variables.

Understanding logical entailment for Relational Logic is complicated by the fact that it is possible to have free variables in Relational Logic sentences. Consider, for example, the premise $q(x,y)$ and the conclusion $q(y,x)$. Does $q(x,y)$ logically entail $q(y,x)$ or not?

Our definition for logical entailment and the semantics of Relational Logic give a clear answer to this question. Logical entailment holds if and only if every truth assignment that satisfies the premise satisfies the conclusion. A truth assignment satisfies a sentence with free variables if and only if it satisfies every instance. In other words, a sentence with free variables is equivalent to the sentence in which all of the free variables are universally quantified. In other words, $q(x,y)$ is satisfied if and only if $\forall x.\forall y.q(x,y)$ is satisfied, and similarly for $q(y,x)$. So, the first sentence here logically entails the second if and only if $\forall x.\forall y.q(x,y)$ logically entails $\forall x.\forall y.q(y,x)$; and, as we just saw, this is, in fact, the case.

6.13 Finite Relational Logic

Finite Relational Logic (FRL) is that subset of Relational Logic in which the Herbrand base is finite. In order to guarantee a finite Herbrand base, we must restrict ourselves to signatures in which there are at most finitely many object constants and no function constants at all.

One interesting feature of FRL is that it is expressively equivalent to Propositional Logic (PL). For any FRL language, we can produce a pairing between the ground relational sentences that language and the proposition constants in a Propositional Logic language. Given this correspondence, for any set of *arbitrary* sentences in our FRL language, there is a corresponding set of sentences in the language of PL such that any FRL truth assignment that satisfies our FRL sentences agrees with the corresponding Propositional Logic truth assignment applied to the Propositional Logic sentences.

The procedure for transforming our FRL sentences to PL has multiple steps, but each step is easy. We first convert our sentences to prenex form, then we ground the result, and we rewrite in Propositional Logic. Let's look at these steps in turn.

A sentence is in *prenex form* if and only if it is closed and all of the quantifiers are on the outside of all logical operators.

Converting a set of FRL sentences to a logically equivalent set in prenex form is simple. First, we rename variables in different quantified sentences to eliminate any duplicates. We then apply quantifier distribution rules in reverse to move quantifiers outside of logical operators. Finally, we universally quantify any free variables in our sentences.

For example, to convert the closed sentence $\forall y.p(x,y) \vee \exists y.q(y)$ to prenex form, we first rename one of the variables. In this case, let's rename the y in the second disjunct to z. This results in the sentence $\forall y.p(x,y) \vee \exists z.q(z)$. We then apply the distribution rules in reverse to produce $\forall y.\exists z.(p(x,y) \vee q(z))$. Finally, we universally quantify the free variable x to produce the prenex form of our original sentence, viz. $\forall x.\forall y.\exists z.(p(x,y) \vee q(z))$

Once we have a set of sentences in prenex form, we compute the grounding. We start with our initial set Δ of sentences and we incrementally build up our grounding Γ. On each step we process a sentence in Δ, using the following described below. The procedure terminates when Δ becomes empty. The set Γ at that point is the grounding of the input.

(1) The first rule covers the case when the sentence ϕ being processed is ground. In this case, we remove the sentence from Delta and add it to Gamma.

$$\Delta_{i+1} = \Delta_i - \{\phi\}$$
$$\Gamma_{i+1} = \Gamma_i \cup \{\phi\}$$

(2) If our sentence is of the form $\forall \nu.\phi[\nu]$, we eliminate the sentence from Δ_i and replace it with

copies of the target, one copy for each object constant τ in our language.

$$\Delta_{i+1} = \Delta_i - \{\forall \nu.\phi[\nu]\} \cup \{\phi[\tau] \mid \tau \text{ an object constant}\}$$
$$\Gamma_{i+1} = \Gamma_i$$

(3) If our sentence of the form $\exists \nu.\phi[\nu]$, we eliminate the sentence from Δ_i and replace it with a disjunction, where each disjunct is a copy of the target in which the quantified variable is replaced by an object constant in our language.

$$\Delta_{i+1} = \Delta_i - \{\exists \nu.\phi[\nu]\} \cup \{\phi[\tau_1] \vee ... \vee \phi[\tau_n]\}$$
$$\Gamma_{i+1} = \Gamma_i$$

The procedure halts when Δ_i becomes empty. The set Γ_i is the grounding of the input. It is easy to see that Γ_i is logically equivalent to the input set.

Here is an example. Suppose we have a language with just two object constants *a* and *b*. And suppose we have the set of sentences shown below. We have one ground sentence, one universally quantified sentence, and one existentially quantified sentence. All are in prenex form.

$$\{p(a), \forall x.(p(x) \Rightarrow q(x)), \exists x.\neg q(x)\}$$

A trace of the procedure is shown below. The first sentence is ground, so we remove it from Δ add it to Γ. The second sentence is universally quantified, so we replace it with a copy for each of our two object constants. The resulting sentences are ground, and so we move them one by one from Δ to Γ. Finally, we ground the existential sentence and add the result to Δ and then move the ground sentence to Γ. At this point, since Δ is empty, Γ is our grounding.

$$\Delta_0 = \{\{p(a), \forall x.(p(x) \Rightarrow q(x)), \exists x.\neg q(x)\}$$
$$\Gamma_0 = \{\}$$

$$\Delta_1 = \{\forall x.(p(x) \Rightarrow q(x)), \exists x.\neg q(x)\}$$
$$\Gamma_1 = \{p(a)\}$$

$$\Delta_2 = \{p(a) \Rightarrow q(a), p(b) \Rightarrow q(b), \exists x.\neg q(x)\}$$
$$\Gamma_2 = \{p(a)\}$$

$$\Delta_3 = \{p(b) \Rightarrow q(b), \exists x.\neg q(x)\}$$
$$\Gamma_3 = \{p(a), p(a) \Rightarrow q(a)\}$$

$$\Delta_4 = \{\exists x.\neg q(x)\}$$
$$\Gamma_4 = \{p(a), p(a) \Rightarrow q(a), p(b) \Rightarrow q(b)\}$$

$$\Delta_5 = \{\neg q(a) \vee \neg q(b)\}$$

$$\Gamma_5 = \{p(a), p(a) \Rightarrow q(a), p(b) \Rightarrow q(b)\}$$

$$\Delta_6 = \{\}$$

$$\Gamma_6 = \{p(a), p(a) \Rightarrow q(a), p(b) \Rightarrow q(b), \neg q(a) \vee \neg q(b)\}$$

Once we have a grounding Γ, we replace each ground relational sentence in Γ by a proposition constant. The resulting sentences are all in Propositional Logic; and the set is equivalent to the sentences in Δ* in that any FRL truth assignment that satisfies our FRL sentences agrees with the corresponding Propositional Logic truth assignment applied to the Propositional Logic sentences.

For example, let's represent the FRL sentence $p(a)$ with the proposition pa; let's represent $p(a)$ with pa; let's represent $a(a)$ with qa; and let's represent $q(b)$ with qb. With this correspondence, we can represent the sentences in our grounding with the Propositional Logic sentences shown below.

$$\{pa, pa \Rightarrow qa, pb \Rightarrow qb, \neg qa \vee \neg qb\}$$

Since the question of unsatisfiability for PL is decidable, then the question of unsatisfiability for FRL is also decidable. Since logical entailment and unsatisfiability are correlated, we also know that the question of logical entailment for FRL is decidable.

Another consequence of this correspondence between FRL and PL is that, like PL, FRL is *compact* - every unsatisfiable set of sentences contains a finite subset that is unsatisfiable. This is important as it assures us that we can demonstrate the unsatisfiability by analyzing just a finite set of sentences; and, as we shall see in the next chapter, logical entailment can be demonstrated with finite proofs.

6.14 Omega Relational Logic

Omega Relational Logic (ORL) is function-free Relational Logic. Like FRL, there are no function constants; but, unlike FRL, we can have infinitely many object constants.

Recall that a logic is *compact* if and only if every unsatisfiable set of sentences has a finite unsatisfiable subset. Suppose that the signature for a language consists of the object constants 1, 2, 3, 4, ... and the unary relation constant p. Take the set of sentences $p(1), p(2), p(3), p(4)$, ... and add in the sentence $\exists x.\neg p(x)$. This set is unsatisfiable. (Our ground atoms say that p is true of everything, but our existentially quantified sentence says that there is something for which p is not true.) However, if we were to remove any one of the sentences, it would be satisfiable; and so every finite subset is satisfiable.

Although it is not compact, ORL is well-behaved in one important way. In particular, the question of unsatisfiability for *finite* sets of sentences is *semidecidable*, i.e. there is a procedure that takes a finite set of sentences in ORL as argument and is guaranteed to halt in finite time with a positive answer if the set is unsatisfiable. Given a satisfiable set of sentences, the procedure *may* halt with a negative answer. However, this is not guaranteed. If the input set is satisfiable, the procedure may run forever.

The first step of the procedure is to convert our sentences to prenex form using the technique described in the section of Finite Relational Logic. We then ground the resulting set. Finally, we examine finite subsets of the grounding for unsatisfiability.

Unfortunately, the grounding technique for FRL does not work for ORL. Since we can have infinitely many object constants in ORL, in dealing with an existentially quantified sentence, we would have to generate an infinitely large disjunction, and sentences of infinite size are not permitted in our language. Fortunately, there is a an alternative form of grounding, called *omega grounding*, which serves our purpose without this defect.

Given a set of sentences in prenex form, we define its omega grounding as follows. Let Δ_0 be our initial set of sentences in prenex form. Let $\Lambda[0]$ be the empty set. And let $\Gamma[0]$ be the empty set. On each step of the definition, we consider one sentences in Δ, taking the sentences in the order in which they were added to Δ; we apply the following rules to define successive versions of Δ and Γ and Lambda; according to the following rules.

(1) If our sentence ϕ is ground, we drop the sentence from Δ and add it to Γ.

$$\begin{aligned}\Delta_{i+1} &= \Delta_i - \{\phi\}\\ \Lambda_{i+1} &= \Lambda_i\\ \Gamma_{i+1} &= \Gamma_i \cup \{\phi\}\end{aligned}$$

(2) If our sentence has the form $\forall\nu.\phi[\nu]$, we remove it from Δ, add it to Λ, and add instances of the target for each constant τ_i that appears in any of our datasets. Note that we do not add copies for every constant in the language, only those used in our sets of sentences.

$$\begin{aligned}\Delta_{i+1} &= \Delta_i - \{\forall\nu.\phi[\nu]\} \cup \{\phi[\tau_1], \ldots, \phi[\tau_k]\}\\ \Lambda_{i+1} &= \Lambda_i \cup \{\forall\nu.\phi[\nu]\}\\ \Gamma_{i+1} &= \Gamma_i\end{aligned}$$

(3) If our sentence has the from $\exists\nu.\phi[\nu]$, we eliminate the sentence from Δ and add in its place an instance of the target for some unused constant τ. There must always be an unused constant, since at each point we have only finitely many sentence while the language has infinitely many constants. We also add to Delta instances of the targets of all universally quantified sentences in Λ using the the new constant τ.

$$\begin{aligned}\Delta_{i+1} &= \Delta_i - \{\exists\nu.\phi[\nu]\} \cup \{\phi[\tau]\} \cup \{\psi[\tau] \mid \forall\mu.\psi[\mu] \in \Lambda_i\}\\ \Lambda_{i+1} &= \Lambda_i\\ \Gamma_{i+1} &= \Gamma_i\end{aligned}$$

If Δ ever becomes empty, the process terminates. In any case, the omega grounding of the initial Δ is the union of all of the Γi sets. It is possible to show that this Γ is satisfiable if and only if the original Δ is satisfiable.

Let's look at an example. Suppose we have a language with object constants 1, 2, 3, ... And suppose we have the set of sentences shown below. We have one ground sentence, one universally quantified sentence, and one existentially quantified sentence. All are in prenex form.

$$\{p(1), \forall x.(p(x) \Rightarrow q(x)), \exists x.\neg q(x)\}$$

The definition goes as follows. The first sentence is ground, so we remove it from Δ add it to Γ. The second sentence is universally quantified, so we add it to Λ; and we replace it on Δ with a copy with 1 substituted for the universally quantified variable. Next, we focus on the existentially quantified sentence. We drop the sentence form Δ and add an instance in its place. In this case, we replace the existential variable by the previously unused object constant 2. We also use 2 in another instance of the sentence in Λ. At this point, everything in Δ is ground, so we just move the sentences, one by one, to Γ. At this point, since we have run out of sentences in Δ, we are done and the current value of Γ is our omega grounding.

$$\Delta_0 = \{p(1), \forall x.(p(x) \Rightarrow q(x)), \exists x.\neg q(x)\}$$
$$\Lambda_0 = \{\}$$
$$\Gamma_0 = \{\}$$

$$\Delta_1 = \{\forall x.(p(x) \Rightarrow q(x)), \exists x.\neg q(x)\}$$
$$\Lambda_1 = \{\}$$
$$\Gamma_1 = \{p(1)\}$$

$$\Delta_2 = \{\exists x.\neg q(x), p(1) \Rightarrow q(1)\}$$
$$\Lambda_2 = \{\forall x.(p(x) \Rightarrow q(x))\}$$
$$\Gamma_2 = \{p(1)\}$$

$$\Delta_3 = \{p(1) \Rightarrow q(1), \neg q(2), p(2) \Rightarrow q(2)\}$$
$$\Lambda_3 = \{\forall x.(p(x) \Rightarrow q(x))\}$$
$$\Gamma_3 = \{p(1)\}$$

$$\Delta_4 = \{\neg q(2), p(2) \Rightarrow q(2)\}$$
$$\Lambda_4 = \{\forall x.(p(x) \Rightarrow q(x))\}$$
$$\Gamma_4 = \{p(1), p(1) \Rightarrow q(1)\}$$

$$\Delta_5 = \{p(2) \Rightarrow q(2)\}$$
$$\Lambda_5 = \{\forall x.(p(x) \Rightarrow q(x))\}$$
$$\Gamma_5 = \{p(1), p(1) \Rightarrow q(1), \neg q(2)\}$$

$$\Delta_6 = \{\}$$
$$\Lambda_6 = \{\forall x.(p(x) \Rightarrow q(x))\}$$
$$\Gamma_6 = \{p(1), p(1) \Rightarrow q(1), \neg q(2), p(2) \Rightarrow q(2)\}$$

Note that, when there are existentially quantified sentences nested within universally quantified sentences, the result of this definition may be infinite in size. Consider, as an example, the set of sentences shown below.

$$\{p(1), \forall x.\exists y.q(x,y)\}$$

In this case, the procedure goes as follows. The first sentence is ground, so we remove it from our input set add it to Γ. The next sentence is universally quantified, so we add it to Λ and replace it in Δ with a copy for each constant mentioned in our sentences. In this case, there is just one, viz. 1. Next, we ground the existential sentence using a new constant, in this case 2, and add the result to Δ We also add a corresponding instance of the sentence in Λ. Our ground instance is then moved to Γ. Again, we ground our existential sentence, in this case using the object constant 3; and, again, we instantiate the sentence in Λ. As before, we move the ground instance to Γ. The construction goes on repeatedly in this fashion.

$\Delta_0 = \{p(1), \forall x.\exists y.q(x,y)\}$
$\Lambda_0 = \{\}$
$\Gamma_0 = \{\}$

$\Delta_1 = \{\forall x.\exists y.q(x,y)\}$
$\Lambda_1 = \{\}$
$\Gamma_1 = \{p(1)\}$

$\Delta_2 = \{\exists y.q(1,y)\}$
$\Lambda_2 = \{\forall x.\exists y.q(x,y)\}$
$\Gamma_2 = \{p(1)\}$

$\Delta_3 = \{q(1,2), \exists y.q(2,y)\}$
$\Lambda_3 = \{\forall x.(p(x) \Rightarrow q(x))\}$
$\Gamma_3 = \{p(1)\}$

$\Delta_4 = \{\exists y.q(2,y)\}$
$\Lambda_4 = \{\forall x.(p(x) \Rightarrow q(x))\}$
$\Gamma_4 = \{p(1), q(1,2)\}$

$\Delta_5 = \{q(2,3), \exists y.q(3,y)\}$
$\Lambda_5 = \{\forall x.(p(x) \Rightarrow q(x))\}$
$\Gamma_5 = \{p(1), q(1,2)\}$

$\Delta_6 = \{\exists y.q(3,y)\}$
$\Lambda_6 = \{\forall x.(p(x) \Rightarrow q(x))\}$
$\Gamma_6 = \{p(1), q(1,2), q(2,3)\}$

...

The union of this infinite sequence of Γ sets is the omega grounding for Δ. Although the answer in this case in infinitely large, it is okay, because we do not need to compute the entire thing, only finite subsets of increasing size.

Note that the omega grounding for a finite set of ORL sentences is all ground. We know that, if a set of ground sentences is unsatisfiable, then there must be a finite subset that is unsatisfiable. Hence, to determine unsatisfiability, we can just check finite subsets of our omega grounding. If the set is unsatisfiable, we will eventually encounter an unsatisfiable set. Otherwise, the procedure may run forever. In any case, this demonstrates the semidecidability of for finite sets of sentences in ORL.

6.15 General Relational Logic

As we have seen Finite Relational is very well-behaved. The questions of unsatisfiability and logical entailment are decidable. In ORL we can express more; and, although we lose decidability, we retain semidecidability. What happens when we relax our restrictions and look at General Relational Logic (GRL), i.e. Relational Logic without any restrictions whatsoever?

The good news about GRL is that it is highly expressive. We can formalize things in GRL that cannot be formalized (at least in finite form) in the other subsets we have examined thus far. For example, in an earlier section, we showed how to define addition and multiplication in finite form. This is not possible with restricted forms of Relational Logic (such as FRL and ORL) and in other logics (e.g. First-Order Logic).

The bad news is that the questions of unsatisfiability and logical entailment for GRL are not semidecidable. Explaining this in detail is beyond the scope of this course. However, we can give a line of argument that suggests why it is true. The argument reduces a problem that is generally accepted to be non-semideciable to the question of unsatisfiability / logical entailment for General Relational Logic. If our logic were semidecidable, then this other question would be semidecidable as well; and, since it is known not to be semidecidable, the GRL must not be semidecidable either.

As we know, Diophantine equations can be readily expressed as sentences in GRL. For example, we can represent the solvability of Diophantine equation $3x^2=1$ with the sentence shown below.

$$\exists x.\exists y.(times(x, x, y) \wedge times(s(s(s(0))), y, s(0)))$$

We can represent every Diophantine in an analogous way. We can express the unsolvability of a Diophantine equation by negating the corresponding sentence. We can then ask the question of whether the axioms of arithmetic logically entail this negation or, equivalently, whether the axioms of Arithmetic together with the unnegated sentence are unsatisfiable.

The problem is that it is well known that determining whether Diophantine equations are unsolvable is not semidecidable. If we could determine the unsatisfiability of our GRL encoding of a Diophantine equation, we could decide whether it is unsolvable, contradicting the non-semidecidability of that problem.

Note that this does not mean GRL is useless. In fact, it is great for expressing such information; and we can prove many results, even though, in general, we cannot prove everything that follows from arbitrary sets of sentences in Relational Logic. We discuss this issue further in later chapters.

Recap

Relational Logic is an extension of Propositional Logic that includes some linguistic features, viz. constants and variables and quantifiers. In Relational Logic, simple sentences have more structure than in Propositional Logic. Furthermore, using variables and quantifiers, we can express information about multiple objects without enumerating those objects; and we can express the existence of objects that satisfy specified conditions without saying which objects they are. The syntax of Relational Logic begins with object constants, function constants, and relation constants. *Relational sentences* and *equations* are the atomic elements from which more complex sentences are built. *Logical sentences* are formed by combining simpler sentences with logical operators. In the version of Relational Logic used here, there are five types of logical sentences - negations, conjunctions, disjunctions, implications, and equivalences. There are two types of *quantified sentences*, viz. *universal sentences* and *existential sentences*. The *Herbrand base* for a Relational Logic language is the set of all ground relational sentences in the language. A *truth assignment* for a Relational Logic language is a mapping that assigns a truth value to each element of it Herbrand base. The truth or falsity of compound sentences is determined from an truth assignment using rules based on the five logical operators of the language. A truth assignment *i satisfies* a sentence if and only if the sentences is *true* under that truth assignment. A sentence is *valid* if and only if it is satisfied by *every* truth assignment. A sentence is *satisfiable* if and only if it is satisfied by at least one truth assignment. A sentence is *falsifiable* if and only if there is at least one truth assignment that makes it false. A sentence is *unsatisfiable* if and only if it is not satisfied by any truth assignment. A sentence is *contingent* if and only if it is both satisfiable and falsifiable, i.e. it is neither valid nor unsatisfiable. A set of sentences Δ *logically entails* a sentence φ (written Δ |= φ) if and only if every truth assignment that satisfies Δ also satisfies φ. A class of questions is *decidable* if and only if there is an effective procedure that is guaranteed to halt on any question in the class and give the correct answer. A class of questions is *semidecidable* if and only if there is an effective procedure that is guaranteed to halt on any question in the class in case the answer is true but is not guaranteed to halt if the answers is false. The questions of unsatisfiability and logical entailment for Finite Relational Logic is decidable. The questions of unsatisfiability and logical entailment for Omega Relational Logic is semidecidable for finite sets of sentences. The questions of unsatisfiability and logical entailment for General Relational Logic is not even semidecidable. A logic is *compact* if and only if every unsatisfiable set of sentences contains a finite subset that is unsatisfiable. Finite Relational Logic is compact, but Omega Relational Logic and General Relational Logic are not.