2

Predicate logic

2.1 The need for a richer language

In the first chapter, we developed propositional logic by examining it from three different angles: its proof theory (the natural deduction calculus), its syntax (the tree-like nature of formulas) and its semantics (what these formulas actually mean). From the outset, this enterprise was guided by the study of declarative sentences, statements about the world which can, for every valuation or model, be given a truth value.

We begin this second chapter by pointing out the limitations of propositional logic with respect to encoding declarative sentences. Propositional logic dealt quite satisfactorily with sentence components like *not*, *and*, *or* and *if ... then*, but the logical aspects of natural and artificial languages are much richer than that. What can we do with modifiers like *there exists ...*, *all ...*, *among ...* and *only ...*? Here, propositional logic shows clear limitations and the desire to express more subtle declarative sentences led to the design of *predicate logic*, which is also called *first-order logic*.

Let us consider the declarative sentence

Every student is younger than some instructor. (2.1)

In propositional logic, we could identify this assertion with a propositional atom p. However, that fails to reflect the finer logical structure of this sentence. What is this statement about? Well, it is about *being a student*, *being an instructor* and *being younger than somebody else*. These are all properties of some sort, so we would like to have a mechanism for expressing them together with their logical relationships and dependences.

We now use *predicates* for that purpose. For example, we could write $S(andy)$ to denote that Andy is a student and $I(paul)$ to say that Paul is an instructor. Likewise, $Y(andy, paul)$ could mean that Andy is younger than

Paul. The symbols S, I and Y are called predicates. Of course, we have to be clear about their meaning. The predicate Y could have meant that the second person is younger than the first one, so we need to specify exactly what these symbols refer to.

Having such predicates at our disposal, we still need to formalise those parts of the sentence above which speak of *every* and *some*. Obviously, this sentence refers to the individuals that make up some academic community (left implicit by the sentence), like Kansas State University or the University of Birmingham, and it says that for each student among them there is an instructor among them such that the student is younger than the instructor.

These predicates are not yet enough to allow us to express the sentence in (2.1). We don't really want to write down all instances of $S(\cdot)$ where $\cdot$ is replaced by every student's name in turn. Similarly, when trying to codify a sentence having to do with the execution of a program, it would be rather laborious to have to write down every state of the computer. Therefore, we employ the concept of a *variable*. Variables are written $u, v, w, x, y, z, \ldots$ or $x_1, y_3, u_5, \ldots$ and can be thought of as *place holders* for concrete values (like a student, or a program state). Using variables, we can now specify the meanings of S, I and Y more formally:

$$\begin{aligned} S(x): &\quad x \text{ is a student} \\ I(x): &\quad x \text{ is an instructor} \\ Y(x,y): &\quad x \text{ is younger than } y. \end{aligned}$$

Note that the names of the variables are not important, provided that we use them consistently. We can state the intended meaning of I by writing

$$I(y): \quad y \text{ is an instructor}$$

or, equivalently, by writing

$$I(z): \quad z \text{ is an instructor.}$$

Variables are mere place holders for objects. The availability of variables is still not sufficient for capturing the essence of the example sentence above. We need to convey the meaning of '**Every** *student* x *is younger than* **some** *instructor* y*.*' This is where we need to introduce *quantifiers* $\forall$ (read: 'for all') and $\exists$ (read: 'there exists' or 'for some') which always come attached to a variable, as in $\forall x$ ('for all x') or in $\exists z$ ('there exists z', or 'there is some z'). Now we can write the example sentence in an entirely symbolic way as

$$\forall x\,(S(x) \to (\exists y\,(I(y) \wedge Y(x,y)))).$$

Actually, this encoding is rather a paraphrase of the original sentence. In our example, the re-translation results in

> *For every x, if x is a student, then there is some y which is an instructor such that x is younger than y.*

Different predicates can have a different number of arguments. The predicates S and I have just one (they are called *unary predicates*), but predicate Y requires two arguments (it is called a *binary predicate*). Predicates with any finite number of arguments are possible in predicate logic.

Another example is the sentence

Not all birds can fly.

For that we choose the predicates B and F which have one argument expressing

$$\begin{aligned} B(x): &\quad x \text{ is a bird} \\ F(x): &\quad x \text{ can fly.} \end{aligned}$$

The sentence 'Not all birds can fly' can now be coded as

$$\neg(\forall x\,(B(x) \rightarrow F(x)))$$

saying: 'It is not the case that all things which are birds can fly.' Alternatively, we could code this as

$$\exists x\,(B(x) \wedge \neg F(x))$$

meaning: 'There is some x which is a bird and cannot fly.' Note that the first version is closer to the linguistic structure of the sentence above. These two formulas should evaluate to `T` in the world we currently live in since, for example, penguins are birds which cannot fly. Shortly, we address how such formulas can be given their meaning in general. We will also explain why formulas like the two above are indeed equivalent *semantically*.

Coding up complex facts expressed in English sentences as logical formulas in predicate logic is important – e.g. in software design with UML or in formal specification of safety-critical systems – and much more care must be taken than in the case of propositional logic. However, once this translation has been accomplished our main objective is to reason symbolically ($\vdash$) or semantically ($\models$) about the information expressed in those formulas.

In Section 2.3, we extend our natural deduction calculus of propositional logic so that it covers logical formulas of predicate logic as well. In this way we are able to prove the validity of sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ in a similar way to that in the first chapter.

In Section 2.4, we generalize the valuations of Chapter 1 to a proper notion of models, real or artificial worlds in which formulas of predicate logic can be true or false, which allows us to define semantic entailment $\phi_1, \phi_2, \ldots, \phi_n \models \psi$.

The latter expresses that, given *any* such model in which all $\phi_1, \phi_2, \ldots, \phi_n$ hold, it is the case that ψ holds in that model as well. In that case, one also says that ψ is *semantically entailed* by $\phi_1, \phi_2, \ldots, \phi_n$. Although this definition of semantic entailment closely matches the one for propositional logic in Definition 1.34, the process of *evaluating a predicate formula* differs from the computation of truth values for propositional logic in the treatment of predicates (and functions). We discuss it in detail in Section 2.4.

It is outside the scope of this book to show that the natural deduction calculus for predicate logic is sound and complete with respect to semantic entailment; but it is indeed the case that

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi \quad \text{iff} \quad \phi_1, \phi_2, \ldots, \phi_n \models \psi$$

for formulas of the predicate calculus. The first proof of this was done by the mathematician K. Gödel.

What kind of reasoning must predicate logic be able to support? To get a feel for that, let us consider the following argument:

> *No books are gaseous. Dictionaries are books. Therefore, no dictionary is gaseous.*

The predicates we choose are

$$\begin{aligned} B(x): &\quad x \text{ is a book} \\ G(x): &\quad x \text{ is gaseous} \\ D(x): &\quad x \text{ is a dictionary.} \end{aligned}$$

Evidently, we need to build a proof theory and semantics that allow us to derive the validity and semantic entailment, respectively, of

$$\neg\exists x\,(B(x) \wedge G(x)),\ \forall x\,(D(x) \rightarrow B(x)) \vdash \neg\exists x\,(D(x) \wedge G(x))$$

$$\neg\exists x\,(B(x) \wedge G(x)),\ \forall x\,(D(x) \rightarrow B(x)) \models \neg\exists x\,(D(x) \wedge G(x)).$$

Verify that these sequents express the argument above in a symbolic form. Predicate logic extends propositional logic not only with quantifiers but with one more concept, that of *function symbols*. Consider the declarative sentence

> *Every child is younger than its mother.*

Using predicates, we could express this sentence as

$$\forall x \, \forall y \, (C(x) \wedge M(y, x) \rightarrow Y(x, y))$$

where $C(x)$ means that x is a child, $M(x, y)$ means that x is y's mother and $Y(x, y)$ means that x is younger than y. (Note that we actually used $M(y, x)$ (y is x's mother), not $M(x, y)$.) As we have coded it, the sentence says that, for all children x and any mother y of theirs, x is younger than y. It is not very elegant to say 'any of x's mothers', since we know that every individual has one and only one mother[1]. The inelegance of coding 'mother' as a predicate is even more apparent if we consider the sentence

Andy and Paul have the same maternal grandmother.

which, using 'variables' a and p for Andy and Paul and a binary predicate M for mother as before, becomes

$$\forall x \, \forall y \, \forall u \, \forall v \, (M(x, y) \wedge M(y, a) \wedge M(u, v) \wedge M(v, p) \rightarrow x = u).$$

This formula says that, if y and v are Andy's and Paul's mothers, respectively, and x and u are *their* mothers (i.e. Andy's and Paul's maternal grandmothers, respectively), then x and u are the same person. Notice that we used a special predicate in predicate logic, *equality*; it is a binary predicate, i.e. it takes two arguments, and is written $=$. Unlike other predicates, it is usually written in between its arguments rather than before them; that is, we write $x = y$ instead of $= (x, y)$ to say that x and y are equal.

The function symbols of predicate logic give us a way of avoiding this ugly encoding, for they allow us to represent y's mother in a more direct way. Instead of writing $M(x, y)$ to mean that x is y's mother, we simply write $m(y)$ to mean y's mother. The symbol m is a function symbol: it takes one argument and returns the mother of that argument. Using m, the two sentences above have simpler encodings than they had using M:

$$\forall x \, (C(x) \rightarrow Y(x, m(x)))$$

now expresses that every child is younger than its mother. Note that we need only one variable rather than two. Representing that Andy and Paul have the same maternal grandmother is even simpler; it is written

$$m(m(a)) = m(m(p))$$

quite directly saying that Andy's maternal grandmother is the same person as Paul's maternal grandmother.

[1] We assume that we are talking about genetic mothers, not adopted mothers, step mothers etc.

One can always do without function symbols, by using a predicate symbol instead. However, it is usually neater to use function symbols whenever possible, because we get more compact encodings. However, function symbols can be used only in situations in which we want to denote a single object. Above, we rely on the fact that every individual has a uniquely defined mother, so that we can talk about x's mother without risking any ambiguity (for example, if x had no mother, or two mothers). For this reason, we cannot have a function symbol $b(\cdot)$ for 'brother'. It might not make sense to talk about x's brother, for x might not have any brothers, or he might have several. 'Brother' must be coded as a binary predicate.

To exemplify this point further, if Mary has several brothers, then the claim that 'Ann likes Mary's brother' is ambiguous. It might be that Ann likes one of Mary's brothers, which we would write as

$$\exists x\,(B(x,m) \wedge L(a,x))$$

where B and L mean 'is brother of' and 'likes,' and a and m mean Ann and Mary. This sentence says that there exists an x which is a brother of Mary and is liked by Ann. Alternatively, if Ann likes all of Mary's brothers, we write it as

$$\forall x\,(B(x,m) \rightarrow L(a,x))$$

saying that any x which is a brother of Mary is liked by Ann. Predicates should be used if a 'function' such as 'your youngest brother' does not always have a value.

Different function symbols may take different numbers of arguments. Functions may take zero arguments and are then called *constants*: a and p above are constants for Andy and Paul, respectively. In a domain involving students and the grades they get in different courses, one might have the binary function symbol $g(\cdot,\cdot)$ taking two arguments: $g(x,y)$ refers to the grade obtained by student x in course y.

2.2 Predicate logic as a formal language

The discussion of the preceding section was intended to give an impression of how we code up sentences as formulas of predicate logic. In this section, we will be more precise about it, giving syntactic rules for the formation of predicate logic formulas. Because of the power of predicate logic, the language is much more complex than that of propositional logic.

The first thing to note is that there are two *sorts* of things involved in a predicate logic formula. The first sort denotes the objects that we are

talking about: individuals such as a and p (referring to Andy and Paul) are examples, as are variables such as x and v. Function symbols also allow us to refer to objects: thus, $m(a)$ and $g(x, y)$ are also objects. Expressions in predicate logic which denote objects are called *terms*.

The other sort of things in predicate logic denotes truth values; expressions of this kind are *formulas*: $Y(x, m(x))$ is a formula, though x and $m(x)$ are terms.

A predicate vocabulary consists of three sets: a set of predicate symbols $\mathcal{P}$, a set of function symbols $\mathcal{F}$ and a set of constant symbols $\mathcal{C}$. Each predicate symbol and each function symbol comes with an arity, the number of arguments it expects. In fact, constants can be thought of as functions which don't take any arguments (and we even drop the argument brackets) – therefore, constants live in the set $\mathcal{F}$ together with the 'true' functions which do take arguments. From now on, we will drop the set $\mathcal{C}$, since it is convenient to do so, and stipulate that constants are 0-arity, so-called *nullary*, functions.

2.2.1 Terms

The terms of our language are made up of variables, constant symbols and functions applied to those. Functions may be nested, as in $m(m(x))$ or $g(m(a), c)$: the grade obtained by Andy's mother in the course c.

Definition 2.1 Terms are defined as follows.

- Any variable is a term.
- If $c \in \mathcal{F}$ is a nullary function, then c is a term.
- If $t_1, t_2, \ldots, t_n$ are terms and $f \in \mathcal{F}$ has arity $n > 0$, then $f(t_1, t_2, \ldots, t_n)$ is a term.
- Nothing else is a term.

In Backus Naur form we may write

$$t ::= x \mid c \mid f(t, \ldots, t)$$

where x ranges over a set of variables var, c over nullary function symbols in $\mathcal{F}$, and f over those elements of $\mathcal{F}$ with arity $n > 0$.

It is important to note that

- the first building blocks of terms are *constants* (nullary functions) and *variables*;
- more complex terms are built from function symbols using as many previously built terms as required by such function symbols; and
- the notion of terms is dependent on the set $\mathcal{F}$. If you change it, you change the set of terms.

Example 2.2 Suppose n, f and g are function symbols, respectively nullary, unary and binary. Then $g(f(n), n)$ and $f(g(n, f(n)))$ are terms, but $g(n)$ and $f(f(n), n)$ are not (they violate the arities). Suppose $0, 1, \ldots$ are nullary, s is unary, and $+$, $-$, and $*$ are binary. Then $*(-(2, +(s(x), y)), x)$ is a term, whose parse tree is illustrated in Figure 2.14 (page 159). Usually, the binary symbols are written infix rather than prefix; thus, the term is usually written $(2 - (s(x) + y)) * x$.

2.2.2 Formulas

The choice of sets $\mathcal{P}$ and $\mathcal{F}$ for predicate and function symbols, respectively, is driven by what we intend to describe. For example, if we work on a database representing relations between our kin we might want to consider $\mathcal{P} = \{M, F, S, D\}$, referring to *being male*, *being female*, *being a son of* ... and *being a daughter of* .... Naturally, F and M are unary predicates (they take one argument) whereas D and S are binary (taking two). Similarly, we may define $\mathcal{F} = \{mother\text{-}of, father\text{-}of\}$.

We already know what the terms over $\mathcal{F}$ are. Given that knowledge, we can now proceed to define the formulas of predicate logic.

Definition 2.3 We define the set of formulas over $(\mathcal{F}, \mathcal{P})$ inductively, using the already defined set of terms over $\mathcal{F}$:

- If $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, and if $t_1, t_2, \ldots, t_n$ are terms over $\mathcal{F}$, then $P(t_1, t_2, \ldots, t_n)$ is a formula.
- If ϕ is a formula, then so is $(\neg\phi)$.
- If ϕ and ψ are formulas, then so are $(\phi \wedge \psi)$, $(\phi \vee \psi)$ and $(\phi \to \psi)$.
- If ϕ is a formula and x is a variable, then $(\forall x\, \phi)$ and $(\exists x\, \phi)$ are formulas.
- Nothing else is a formula.

Note how the arguments given to predicates are always terms. This can also be seen in the Backus Naur form (BNF) for predicate logic:

$$\phi ::= P(t_1, t_2, \ldots, t_n) \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid (\forall x\, \phi) \mid (\exists x\, \phi) \tag{2.2}$$

where $P \in \mathcal{P}$ is a predicate symbol of arity $n \geq 1$, t_i are terms over $\mathcal{F}$ and x is a variable. Recall that each occurrence of ϕ on the right-hand side of the $::=$ stands for any formula already constructed by these rules. (What role could predicate symbols of arity 0 play?)

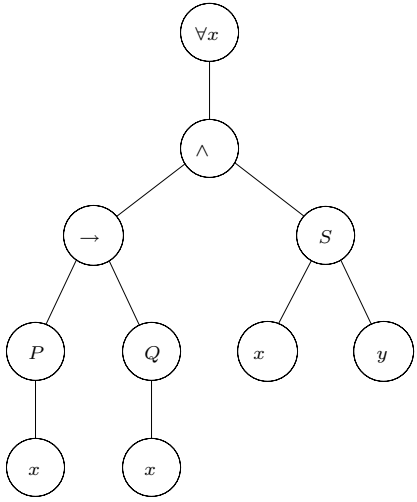

Figure 2.1. A parse tree of a predicate logic formula.

Convention 2.4 For convenience, we retain the usual binding priorities agreed upon in Convention 1.3 and add that $\forall y$ and $\exists y$ bind like $\neg$. Thus, the order is:

- $\neg$, $\forall y$ and $\exists y$ bind most tightly;
- then $\vee$ and $\wedge$;
- then $\rightarrow$, which is right-associative.

We also often omit brackets around quantifiers, provided that doing so introduces no ambiguities.

Predicate logic formulas can be represented by parse trees. For example, the parse tree in Figure 2.1 represents the formula $\forall x\,((P(x) \rightarrow Q(x)) \wedge S(x, y))$.

Example 2.5 Consider translating the sentence

> Every son of my father is my brother.

into predicate logic. As before, the design choice is whether we represent 'father' as a predicate or as a function symbol.

1. As a predicate. We choose a constant m for 'me' or 'I,' so m is a term, and we choose further $\{S, F, B\}$ as the set of predicates with meanings

$$\begin{aligned} S(x,y): &\quad x \text{ is a son of } y \\ F(x,y): &\quad x \text{ is the father of } y \\ B(x,y): &\quad x \text{ is a brother of } y. \end{aligned}$$

Then the symbolic encoding of the sentence above is

$$\forall x \forall y \, (F(x,m) \wedge S(y,x) \rightarrow B(y,m)) \tag{2.3}$$

saying: 'For all x and all y, if x is a father of m and if y is a son of x, then y is a brother of m.'

2. As a function. We keep m, S and B as above and write f for the function which, given an argument, returns the corresponding father. Note that this works only because fathers are unique and always defined, so f really is a function as opposed to a mere relation.
   The symbolic encoding of the sentence above is now

$$\forall x \, (S(x, f(m)) \rightarrow B(x,m)) \tag{2.4}$$

meaning: 'For all x, if x is a son of the father of m, then x is a brother of m;' it is less complex because it involves only one quantifier.

Formal specifications require *domain-specific knowledge*. Domain-experts often don't make some of this knowledge explicit, so a specifier may miss important constraints for a model or implementation. For example, the specification in (2.3) and (2.4) may seem right, but what about the case when the values of x and m are equal? If the domain of kinship is not common knowledge, then a specifier may not realize that a man cannot be his own brother. Thus, (2.3) and (2.4) are not completely correct!

2.2.3 Free and bound variables

The introduction of variables and quantifiers allows us to express the notions of *all ...* and *some ...* Intuitively, to verify that $\forall x \, Q(x)$ is true amounts to replacing x by any of its possible values and checking that Q holds for each one of them. There are two important and different senses in which such formulas can be 'true.' First, if we give concrete meanings to all predicate and function symbols involved we have a *model* and can *check* whether a formula is true for this particular model. For example, if a formula encodes a required behaviour of a hardware circuit, then we would want to know whether it is true for the model of the circuit. Second, one sometimes would like to ensure that certain formulas are true *for all models*. Consider $P(c) \wedge \forall y(P(y) \rightarrow Q(y)) \rightarrow Q(c)$ for a constant c; clearly, this formula should be true no matter what model we are looking at. It is this second kind of truth which is the primary focus of Section 2.3.

Unfortunately, things are more complicated if we want to define formally what it means for a formula to be true in a given model. Ideally, we seek a definition that we could use to write a computer program verifying that a formula holds in a given model. To begin with, we need to understand that variables occur in different ways. Consider the formula

$$\forall x\,((P(x) \to Q(x)) \wedge S(x, y)).$$

We draw its parse tree in the same way as for propositional formulas, but with two additional sorts of nodes:

- The quantifiers $\forall x$ and $\exists y$ form nodes and have, like negation, just one subtree.
- Predicate expressions, which are generally of the form $P(t_1, t_2, \ldots, t_n)$, have the symbol P as a node, but now P has n many subtrees, namely the parse trees of the terms $t_1, t_2, \ldots, t_n$.

So in our particular case above we arrive at the parse tree in Figure 2.1. You can see that variables occur at two different sorts of places. First, they appear next to quantifiers $\forall$ and $\exists$ in nodes like $\forall x$ and $\exists z$; such nodes always have one subtree, subsuming their scope to which the respective quantifier applies.

The other sort of occurrence of variables is *leaf nodes containing variables*. If variables are leaf nodes, then they stand for values that still have to be made concrete. There are two principal such occurrences:

1. In our example in Figure 2.1, we have three leaf nodes x. If we walk up the tree beginning at any one of these x leaves, we run into the quantifier $\forall x$. This means that those occurrences of x are actually *bound* to $\forall x$ so they represent, or stand for, *any possible value of* x.
2. In walking upwards, the only quantifier that the leaf node y runs into is $\forall x$ but that x has nothing to do with y; x and y are different place holders. So y is *free* in this formula. This means that its value has to be specified by some additional information, for example, the contents of a location in memory.

Definition 2.6 Let ϕ be a formula in predicate logic. An occurrence of x in ϕ is free in ϕ if it is a leaf node in the parse tree of ϕ such that there is no path upwards from that node x to a node $\forall x$ or $\exists x$. Otherwise, that occurrence of x is called bound. For $\forall x\,\phi$, or $\exists x\,\phi$, we say that ϕ – minus any of ϕ's subformulas $\exists x\,\psi$, or $\forall x\,\psi$ – is the scope of $\forall x$, respectively $\exists x$.

Thus, if x occurs in ϕ, then it is bound if, and only if, it is in the scope of some $\exists x$ or some $\forall x$; otherwise it is free. In terms of parse trees, the scope of a quantifier is just its subtree, minus any subtrees which re-introduce a

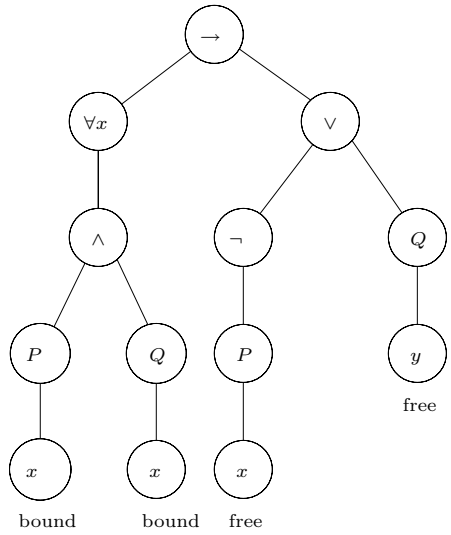

Figure 2.2. A parse tree of a predicate logic formula illustrating free and bound occurrences of variables.

quantifier for x; e.g. the scope of $\forall x$ in $\forall x\,(P(x) \to \exists x\,Q(x))$ is $P(x)$. It is quite possible, and common, that a variable is bound and free in a formula. Consider the formula

$$(\forall x\,(P(x) \wedge Q(x))) \to (\neg P(x) \vee Q(y))$$

and its parse tree in Figure 2.2. The two x leaves in the subtree of $\forall x$ are bound since they are in the scope of $\forall x$, but the leaf x in the right subtree of $\to$ is free since it is *not* in the scope of any quantifier $\forall x$ or $\exists x$. Note, however, that a single leaf either is under the scope of a quantifier, or it isn't. Hence *individual* occurrences of variables are either free or bound, never both at the same time.

2.2.4 Substitution

Variables are place holders so we must have some means of *replacing* them with more concrete information. On the syntactic side, we often need to replace a leaf node x by the parse tree of an entire term t. Recall from the definition of formulas that any replacement of x may only be a term; it could not be a predicate expression, or a more complex formula, for x serves as a term to a predicate symbol one step higher up in the parse tree (see Definition 2.1 and the grammar in (2.2)). In substituting t for x we have to

leave untouched the *bound* leaves x since they are in the scope of some $\exists x$ or $\forall x$, i.e. they stand for *some unspecified* or *all* values respectively.

Definition 2.7 Given a variable x, a term t and a formula ϕ we define $\phi[t/x]$ to be the formula obtained by replacing each free occurrence of variable x in ϕ with t.

Substitutions are easily understood by looking at some examples. Let f be a function symbol with two arguments and ϕ the formula with the parse tree in Figure 2.1. Then $f(x, y)$ is a term and $\phi[f(x, y)/x]$ is just ϕ again. This is true because *all* occurrences of x are bound in ϕ, so *none* of them gets substituted.

Now consider ϕ to be the formula with the parse tree in Figure 2.2. Here we have one free occurrence of x in ϕ, so we substitute the parse tree of $f(x, y)$ for that free leaf node x and obtain the parse tree in Figure 2.3. Note that the bound x leaves are unaffected by this operation. You can see that the process of substitution is straightforward, but requires that it be applied *only to the free occurrences* of the variable to be substituted.

A word on notation: in writing $\phi[t/x]$, we really mean this to be the formula *obtained* by performing the operation $[t/x]$ on ϕ. Strictly speaking, the chain of symbols $\phi[t/x]$ is *not* a logical formula, but its *result* will be a formula, provided that ϕ was one in the first place.

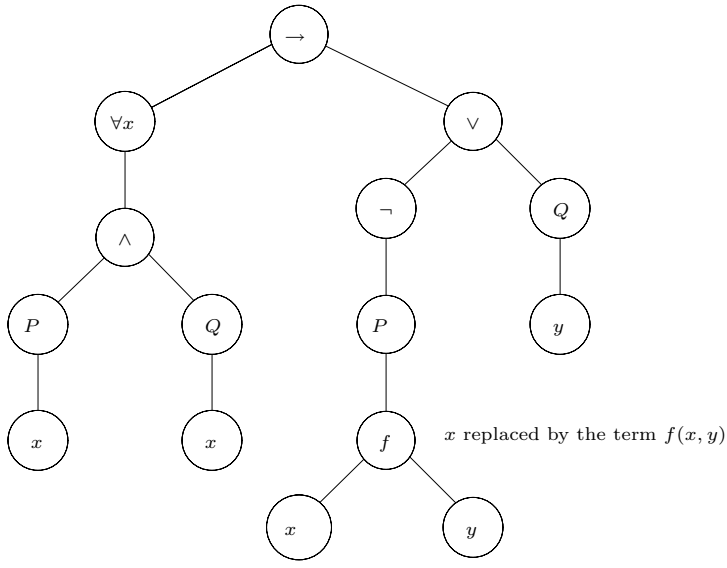

Figure 2.3. A parse tree of a formula resulting from substitution.

Unfortunately, substitutions can give rise to undesired side effects. In performing a substitution $\phi[t/x]$, the term t may contain a variable y, where free occurrences of x in ϕ are under the scope of $\exists y$ or $\forall y$ in ϕ. By carrying out this substitution $\phi[t/x]$, the value y, which might have been fixed by a concrete context, gets caught in the scope of $\exists y$ or $\forall y$. This binding capture overrides the context specification of the concrete value of y, for it will now stand for '*some unspecified*' or '*all*,' respectively. Such undesired variable captures are to be avoided at all costs.

Definition 2.8 Given a term t, a variable x and a formula ϕ, we say that t is free for x in ϕ if no free x leaf in ϕ occurs in the scope of $\forall y$ or $\exists y$ for any variable y occurring in t.

This definition is maybe hard to swallow. Let us think of it in terms of parse trees. Given the parse tree of ϕ and the parse tree of t, we can perform the substitution $[t/x]$ on ϕ to obtain the formula $\phi[t/x]$. The latter has a parse tree where all free x leaves of the parse tree of ϕ are replaced by the parse tree of t. What 't is free for x in ϕ' means is that the variable leaves of the parse tree of t won't become bound if placed into the bigger parse tree of $\phi[t/x]$. For example, if we consider x, t and ϕ in Figure 2.3, then t is free for x in ϕ since the *new* leaf variables x and y of t are not under the scope of any quantifiers involving x or y.

Example 2.9 Consider the ϕ with parse tree in Figure 2.4 and let t be $f(y, y)$. All two occurrences of x in ϕ are free. The leftmost occurrence of x could be substituted since it is not in the scope of any quantifier, but substituting the rightmost x leaf introduces a new variable y in t which becomes bound by $\forall y$. Therefore, $f(y, y)$ is not free for x in ϕ.

What if there are no free occurrences of x in ϕ? Inspecting the definition of 't is free for x in ϕ,' we see that *every* term t is free for x in ϕ in that case, since no free variable x of ϕ is below some quantifier in the parse tree of ϕ. So the problematic situation of variable capture in performing $\phi[t/x]$ cannot occur. Of course, in that case $\phi[t/x]$ is just ϕ again.

It might be helpful to compare 't is free for x in ϕ' with a precondition of calling a procedure for substitution. If you are asked to compute $\phi[t/x]$ in your exercises or exams, then that is what you should do; but any reasonable implementation of substitution used in a theorem prover would have to check whether t is free for x in ϕ and, if not, rename some variables with fresh ones to avoid the undesirable capture of variables.

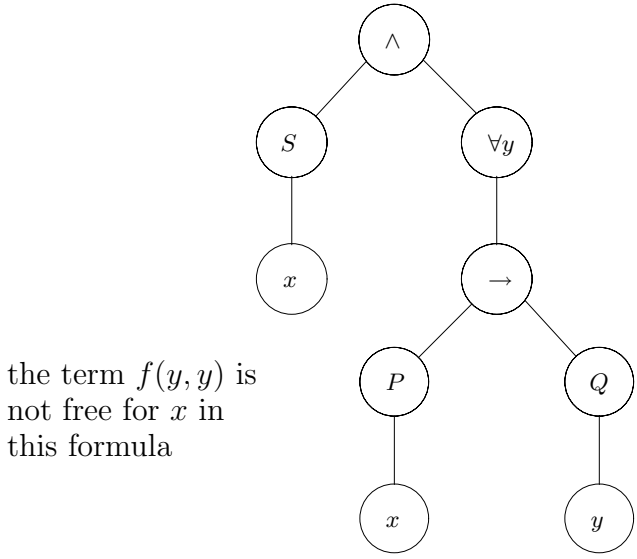

Figure 2.4. A parse tree for which a substitution has dire consequences.

2.3 Proof theory of predicate logic

2.3.1 Natural deduction rules

Proofs in the natural deduction calculus for predicate logic are similar to those for propositional logic in Chapter 1, except that we have new proof rules for dealing with the quantifiers and with the equality symbol. Strictly speaking, we are *overloading* the previously established proof rules for the propositional connectives $\wedge$, $\vee$ etc. That simply means that any proof rule of Chapter 1 is still valid for logical formulas of predicate logic (we originally defined those rules for logical formulas of propositional logic). As in the natural deduction calculus for propositional logic, the additional rules for the quantifiers and equality will come in two flavours: introduction and elimination rules.

The proof rules for equality First, let us state the proof rules for equality. Here equality does not mean syntactic, or intensional, equality, but equality in terms of computation results. In either of these senses, any term t has to be equal to itself. This is expressed by the introduction rule for equality:

$$\frac{}{t = t}\,{=}\mathrm{i} \tag{2.5}$$

which is an axiom (as it does not depend on any premises). Notice that it

may be invoked only if t is a term, our language doesn't permit us to talk about equality between formulas.

This rule is quite evidently sound, but it is not very useful on its own. What we need is a principle that allows us to substitute equals for equals repeatedly. For example, suppose that $y * (w + 2)$ equals $y * w + y * 2$; then it certainly must be the case that $z \geq y * (w + 2)$ implies $z \geq y * w + y * 2$ and vice versa. We may now express this substitution principle as the rule =e:

$$\frac{t_1 = t_2 \quad \phi[t_1/x]}{\phi[t_2/x]} =\text{e}.$$

Note that t_1 and t_2 have to be free for x in ϕ, whenever we want to apply the rule =e; this is an example of a *side condition* of a proof rule.

Convention 2.10 Throughout this section, when we write a substitution in the form $\phi[t/x]$, we implicitly assume that t is free for x in ϕ; for, as we saw in the last section, a substitution doesn't make sense otherwise.

We obtain proof

1	$(x + 1) = (1 + x)$	premise
2	$(x + 1 > 1) \to (x + 1 > 0)$	premise
3	$(1 + x > 1) \to (1 + x > 0)$	=e 1, 2

establishing the validity of the sequent

$$x + 1 = 1 + x,\ (x + 1 > 1) \to (x + 1 > 0) \ \vdash\ (1 + x) > 1 \to (1 + x) > 0.$$

In this particular proof t_1 is $(x + 1)$, t_2 is $(1 + x)$ and ϕ is $(x > 1) \to (x > 0)$. We used the name =e since it reflects what this rule is doing to data: it eliminates the equality in $t_1 = t_2$ by replacing all t_1 in $\phi[t_1/x]$ with t_2. This is a sound substitution principle, since the assumption that t_1 equals t_2 guarantees that the logical meanings of $\phi[t_1/x]$ and $\phi[t_2/x]$ match.

The principle of substitution, in the guise of the rule =e, is quite powerful. Together with the rule =i, it allows us to show the sequents

$$t_1 = t_2 \vdash t_2 = t_1 \tag{2.6}$$

$$t_1 = t_2,\ t_2 = t_3 \vdash t_1 = t_3. \tag{2.7}$$

A proof for (2.6) is:

1	$t_1 = t_2$	premise
2	$t_1 = t_1$	=i
3	$t_2 = t_1$	=e 1, 2

where ϕ is $x = t_1$. A proof for (2.7) is:

1	$t_2 = t_3$	premise
2	$t_1 = t_2$	premise
3	$t_1 = t_3$	=e 1, 2

where ϕ is $t_1 = x$, so in line 2 we have $\phi[t_2/x]$ and in line 3 we obtain $\phi[t_3/x]$, as given by the rule =e applied to lines 1 and 2. Notice how we applied the scheme =e with several different instantiations.

Our discussion of the rules =i and =e has shown that they force equality to be *reflexive* (2.5), *symmetric* (2.6) and *transitive* (2.7). These are minimal and necessary requirements for any sane concept of (extensional) equality. We leave the topic of equality for now to move on to the proof rules for quantifiers.

The proof rules for universal quantification The rule for eliminating $\forall$ is the following:

$$\frac{\forall x\, \phi}{\phi[t/x]}\ \forall x\,\mathrm{e}.$$

It says: If $\forall x\, \phi$ is true, then you could replace the x in ϕ by any term t (given, as usual, the side condition that t be free for x in ϕ) and conclude that $\phi[t/x]$ is true as well. The intuitive soundness of this rule is self-evident.

Recall that $\phi[t/x]$ is obtained by replacing all free occurrences of x in ϕ by t. You may think of the term t as a more concrete *instance* of x. Since ϕ is assumed to be true for all x, that should also be the case for any term t.

Example 2.11 To see the necessity of the proviso that t be free for x in ϕ, consider the case that ϕ is $\exists y\,(x < y)$ and the term to be substituted for x is y. Let's suppose we are reasoning about numbers with the usual 'smaller than' relation. The statement $\forall x\, \phi$ then says that for all numbers n there is some bigger number m, which is indeed true of integers or real numbers. However, $\phi[y/x]$ is the formula $\exists y\,(y < y)$ saying that there is a number which is bigger than itself. This is wrong; and we must not allow a proof rule which derives semantically wrong things from semantically valid

ones. Clearly, what went wrong was that y became bound in the process of substitution; y is not free for x in ϕ. Thus, in going from $\forall x\,\phi$ to $\phi[t/x]$, we have to enforce the side condition that t be free for x in ϕ: use a fresh variable for y to change ϕ to, say, $\exists z\,(x < z)$ and then apply $[y/x]$ to that formula, rendering $\exists z\,(y < z)$.

The rule $\forall x\,\mathrm{i}$ is a bit more complicated. It employs a proof box similar to those we have already seen in natural deduction for propositional logic, but this time the box is to stipulate the scope of the 'dummy variable' x_0 rather than the scope of an assumption. The rule $\forall x\,\mathrm{i}$ is written

$$\dfrac{\boxed{\begin{array}{l} x_0 \\ \quad\vdots \\ \quad \phi[x_0/x] \end{array}}}{\forall x\,\phi}\ \forall x\,\mathrm{i}.$$

It says: If, starting with a 'fresh' variable x_0, you are able to prove some formula $\phi[x_0/x]$ with x_0 in it, then (*because* x_0 *is fresh*) you can derive $\forall x\,\phi$. The important point is that x_0 is a new variable which doesn't occur *anywhere outside its box*; we think of it as an *arbitrary* term. Since we assumed nothing about this x_0, anything would work in its place; hence the conclusion $\forall x\,\phi$.

It takes a while to understand this rule, since it seems to be going from the particular case of ϕ to the general case $\forall x\,\phi$. The side condition, that x_0 does not occur outside the box, is what allows us to get away with this.

To understand this, think of the following analogy. If you want to prove to someone that you can, say, split a tennis ball in your hand by squashing it, you might say 'OK, give me a tennis ball and I'll split it.' So we give you one and you do it. But how can we be sure that you could split *any* tennis ball in this way? Of course, we can't give you *all of them*, so how could we be sure that you could split any one? Well, we assume that the one you did split was an arbitrary, or 'random,' one, i.e. that it wasn't special in any way – like a ball which you may have 'prepared' beforehand; and that is enough to convince us that you could split *any* tennis ball. Our rule says that if you can prove ϕ about an x_0 that isn't special in any way, then you could prove it for any x whatsoever.

To put it another way, the step from ϕ to $\forall x\,\phi$ is legitimate only if we have arrived at ϕ in such a way that none of its assumptions contain x as a free variable. Any assumption which has a free occurrence of x puts constraints

on such an x. For example, the assumption $\mathrm{bird}(x)$ confines x to the realm of birds and anything we can prove about x using this formula will have to be a statement restricted to birds and not about anything else we might have had in mind.

It is time we looked at an example of these proof rules at work. Here is a proof of the sequent $\forall x\,(P(x) \to Q(x)),\ \forall x\,P(x) \ \vdash\ \forall x\,Q(x)$:

1		$\forall x\,(P(x) \to Q(x))$	premise
2		$\forall x\,P(x)$	premise
3	x_0	$P(x_0) \to Q(x_0)$	$\forall x$ e 1
4		$P(x_0)$	$\forall x$ e 2
5		$Q(x_0)$	$\to$e 3, 4
6		$\forall x\,Q(x)$	$\forall x$ i 3–5

The structure of this proof is guided by the fact that the conclusion is a $\forall$ formula. To arrive at this, we will need an application of $\forall x$ i, so we set up the box controlling the scope of x_0. The rest is now mechanical: we prove $\forall x\,Q(x)$ by proving $Q(x_0)$; but the latter we can prove as soon as we can prove $P(x_0)$ and $P(x_0) \to Q(x_0)$, which themselves are instances of the premises (obtained by $\forall$e with the term x_0). Note that we wrote the name of the dummy variable to the left of the first proof line in its scope box.

Here is a simpler example which uses only $\forall x$ e: we show the validity of the sequent $P(t),\ \forall x\,(P(x) \to \neg Q(x)) \ \vdash\ \neg Q(t)$ for any term t:

1	$P(t)$	premise
2	$\forall x\,(P(x) \to \neg Q(x))$	premise
3	$P(t) \to \neg Q(t)$	$\forall x$ e 2
4	$\neg Q(t)$	$\to$e 3, 1

Note that we invoked $\forall x$ e with the same instance t as in the assumption $P(t)$. If we had invoked $\forall x$ e with y, say, and obtained $P(y) \to \neg Q(y)$, then that would have been valid, but it would not have been helpful in the case that y was different from t. Thus, $\forall x$ e is really a *scheme* of rules, one for each term t (free for x in ϕ), and we should make our choice on the basis of consistent pattern matching. Further, note that we have rules $\forall x$ i and $\forall x$ e *for each variable* x. In particular, there are rules $\forall y$ i, $\forall y$ e and so on. We

will write ∀i and ∀e when we speak about such rules without concern for the actual quantifier variable.

Notice also that, although the square brackets representing substitution appear in the rules ∀i and ∀e, they do not appear when we use those rules. The reason for this is that we actually carry out the substitution that is asked for. In the rules, the expression $\phi[t/x]$ means: 'ϕ, but with free occurrences of x replaced by t.' Thus, if ϕ is $P(x, y) \to Q(y, z)$ and the rule refers to $\phi[a/y]$, we carry out the substitution and write $P(x, a) \to Q(a, z)$ in the proof.

A helpful way of understanding the universal quantifier rules is to compare the rules for ∀ with those for ∧. The rules for ∀ are in some sense generalisations of those for ∧; whereas ∧ has just two conjuncts, ∀ acts like it conjoins lots of formulas (one for each substitution instance of its variable). Thus, whereas ∧i has two premises, $\forall x\,\mathrm{i}$ has a premise $\phi[x_0/x]$ for each possible 'value' of x_0. Similarly, where and-elimination allows you to deduce from $\phi \wedge \psi$ whichever of ϕ and ψ you like, forall-elimination allows you to deduce $\phi[t/x]$ from $\forall x\,\phi$, for whichever t you (and the side condition) like. To say the same thing another way: think of $\forall x\,\mathrm{i}$ as saying: to prove $\forall x\,\phi$, you have to prove $\phi[x_0/x]$ for every possible value x_0; while ∧i says that to prove $\phi_1 \wedge \phi_2$ you have to prove ϕ_i for every $i = 1, 2$.

The proof rules for existential quantification The analogy between ∀ and ∧ extends also to ∃ and ∨; and you could even try to guess the rules for ∃ by starting from the rules for ∨ and applying the same ideas as those that related ∧ to ∀. For example, we saw that the rules for or-introduction were a sort of dual of those for and-elimination; to emphasise this point, we could write them as

$$\frac{\phi_1 \wedge \phi_2}{\phi_k}\,{\wedge\mathrm{e_k}} \qquad \frac{\phi_k}{\phi_1 \vee \phi_2}\,{\vee\mathrm{i_k}}$$

where k can be chosen to be either 1 or 2. Therefore, given the form of forall-elimination, we can infer that exists-introduction must be simply

$$\frac{\phi[t/x]}{\exists x \phi}\,\exists x\,\mathrm{i}.$$

Indeed, this is correct: it simply says that we can deduce $\exists x\,\phi$ whenever we have $\phi[t/x]$ for some term t (naturally, we impose the side condition that t be free for x in ϕ).

In the rule ∃i, we see that the formula $\phi[t/x]$ contains, from a computational point of view, more information than $\exists x\,\phi$. The latter merely says

that ϕ holds for some, unspecified, value of x; whereas $\phi[t/x]$ has a witness t at its disposal. Recall that the square-bracket notation asks us actually to carry out the substitution. However, the notation $\phi[t/x]$ is somewhat misleading since it suggests not only the right witness t but also the formula ϕ itself. For example, consider the situation in which t equals y such that $\phi[y/x]$ is $y = y$. Then you can check for yourself that ϕ could be a number of things, like $x = x$ or $x = y$. Thus, $\exists x\, \phi$ will depend on which of these ϕ you were thinking of.

Extending the analogy between $\exists$ and $\vee$, the rule $\vee$e leads us to the following formulation of $\exists$e:

$$\dfrac{\exists x\, \phi \qquad \boxed{\begin{array}{c} x_0\ \phi[x_0/x] \\ \vdots \\ \chi \end{array}}}{\chi}\ \exists\mathrm{e}.$$

Like $\vee$e, it involves a case analysis. The reasoning goes: We know $\exists x\, \phi$ is true, so ϕ is true for at least one 'value' of x. So we do a case analysis over all those possible values, writing x_0 as a generic value representing them all. If assuming $\phi[x_0/x]$ allows us to prove some χ which doesn't mention x_0, then this χ must be true whichever x_0 makes $\phi[x_0/x]$ true. And that's precisely what the rule $\exists$e allows us to deduce. Of course, we impose the side condition that x_0 can't occur outside its box (therefore, in particular, it cannot occur in χ). The box is controlling two things: the scope of x_0 and also the scope of the assumption $\phi[x_0/x]$.

Just as $\vee$e says that to use $\phi_1 \vee \phi_2$, you have to be prepared for either of the ϕ_i, so $\exists$e says that to use $\exists x\, \phi$ you have to be prepared for any possible $\phi[x_0/x]$. Another way of thinking about $\exists$e goes like this: If you know $\exists x\, \phi$ and you can derive some χ from $\phi[x_0/x]$, i.e. by giving a name to the thing you know exists, then you can derive χ even without giving that thing a name (provided that χ does not refer to the name x_0).

The rule $\exists x\, \mathrm{e}$ is also similar to $\vee$e in the sense that both of them are elimination rules which don't have to conclude a *subformula* of the formula they are about to eliminate. Please verify that all other elimination rules introduced so far have this *subformula property*.[2] This property is computationally very pleasant, for it allows us to narrow down the search space for a proof dramatically. Unfortunately, $\exists x\, \mathrm{e}$, like its cousin $\vee$e, is not of that computationally benign kind.

[2] For $\forall x\, \mathrm{e}$ we perform a substitution $[t/x]$, but it preserves the logical structure of ϕ.

Let us practice these rules on a couple of examples. Certainly, we should be able to prove the validity of the sequent $\forall x\,\phi \vdash \exists x\,\phi$. The proof

1		$\forall x\,\phi$	premise
2		$\phi[x/x]$	$\forall x$ e 1
3		$\exists x\,\phi$	$\exists x$ i 2

demonstrates that, where we chose t to be x with respect to both $\forall x$ e and to $\exists x$ i (and note that x is free for x in ϕ and that $\phi[x/x]$ is simply ϕ again).

Proving the validity of the sequent $\forall x\,(P(x) \to Q(x)),\ \exists x\,P(x) \vdash \exists x\,Q(x)$ is more complicated:

1		$\forall x\,(P(x) \to Q(x))$	premise
2		$\exists x\,P(x)$	premise
3	x_0	$P(x_0)$	assumption
4		$P(x_0) \to Q(x_0)$	$\forall x$ e 1
5		$Q(x_0)$	$\to$e 4, 3
6		$\exists x\,Q(x)$	$\exists x$ i 5
7		$\exists x\,Q(x)$	$\exists x$ e 2, 3–6

The motivation for introducing the box in line 3 of this proof is the existential quantifier in the premise $\exists x\,P(x)$ which has to be eliminated. Notice that the $\exists$ in the conclusion has to be introduced *within the box* and observe the nesting of these two steps. The formula $\exists x\,Q(x)$ in line 6 is the instantiation of χ in the rule $\exists$e and does not contain an occurrence of x_0, so it is allowed to leave the box to line 7. The almost identical 'proof'

1		$\forall x\,(P(x) \to Q(x))$	premise
2		$\exists x\,P(x)$	premise
3	x_0	$P(x_0)$	assumption
4		$P(x_0) \to Q(x_0)$	$\forall x$ e 1
5		$Q(x_0)$	$\to$e 4, 3
6		$Q(x_0)$	$\exists x$ e 2, 3–5
7		$\exists x\,Q(x)$	$\exists x$ i 6

is illegal! Line 6 allows the fresh parameter x_0 to escape the scope of the box which declares it. This is not permissible and we will see on page 116 an example where such illicit use of proof rules results in unsound arguments.

A sequent with a slightly more complex proof is

$$\forall x\,(Q(x) \to R(x)),\ \exists x\,(P(x) \wedge Q(x)) \ \vdash\ \exists x\,(P(x) \wedge R(x))$$

and could model some argument such as

> *If all quakers are reformists and if there is a protestant who is also a quaker, then there must be a protestant who is also a reformist.*

One possible proof strategy is to assume $P(x_0) \wedge Q(x_0)$, get the instance $Q(x_0) \to R(x_0)$ from $\forall x\,(Q(x) \to R(x))$ and use $\wedge\mathrm{e}_2$ to get our hands on $Q(x_0)$, which gives us $R(x_0)$ via $\to$e …:

1		$\forall x\,(Q(x) \to R(x))$	premise
2		$\exists x\,(P(x) \wedge Q(x))$	premise
3	x_0	$P(x_0) \wedge Q(x_0)$	assumption
4		$Q(x_0) \to R(x_0)$	$\forall x$ e 1
5		$Q(x_0)$	$\wedge\mathrm{e}_2$ 3
6		$R(x_0)$	$\to$e 4, 5
7		$P(x_0)$	$\wedge\mathrm{e}_1$ 3
8		$P(x_0) \wedge R(x_0)$	$\wedge$i 7, 6
9		$\exists x\,(P(x) \wedge R(x))$	$\exists x$ i 8
10		$\exists x\,(P(x) \wedge R(x))$	$\exists x$ e 2, 3–9

Note the strategy of this proof: We list the two premises. The second premise is of use here only if we apply $\exists x$ e to it. This sets up the proof box in lines 3–9 as well as the fresh parameter name x_0. Since we want to prove $\exists x\,(P(x) \wedge R(x))$, this formula has to be the last one in the box (our goal) and the rest involves $\forall x$ e and $\exists x$ i.

The rules $\forall$i and $\exists$e both have the side condition that the dummy variable cannot occur outside the box in the rule. Of course, these rules may still be nested, by choosing another fresh name (e.g. y_0) for the dummy variable. For example, consider the sequent $\exists x\,P(x), \forall x\,\forall y\,(P(x) \to Q(y)) \ \vdash\ \forall y\,Q(y)$. (Look how strong the second premise is, by the way: given any x, y, if $P(x)$, then $Q(y)$. This means that, if there is any object with the property P, then all objects shall have the property Q.) Its proof goes as follows: We take an arbitrary y_0 and prove $Q(y_0)$; this we do by observing that, since some x

satisfies P, so by the second premise any y satisfies Q:

1		$\exists x\, P(x)$	premise
2		$\forall x \forall y\, (P(x) \to Q(y))$	premise
3	y_0		
4	x_0	$P(x_0)$	assumption
5		$\forall y\, (P(x_0) \to Q(y))$	$\forall x$ e 2
6		$P(x_0) \to Q(y_0)$	$\forall y$ e 5
7		$Q(y_0)$	$\to$e 6, 4
8		$Q(y_0)$	$\exists x$ e 1, 4–7
9		$\forall y\, Q(y)$	$\forall y$ i 3–8

There is no special reason for picking x_0 as a name for the dummy variable we use for $\forall x$ and $\exists x$ and y_0 as a name for $\forall y$ and $\exists y$. We do this only because it makes it easier for us humans. Again, study the strategy of this proof. We ultimately have to show a $\forall y$ formula which requires us to use $\forall y$ i, i.e. we need to open up a proof box (lines 3–8) whose subgoal is to prove a generic instance $Q(y_0)$. Within that box we want to make use of the premise $\exists x\, P(x)$ which results in the proof box set-up of lines 4–7. Notice that, in line 8, we may well move $Q(y_0)$ out of the box controlled by x_0.

We have repeatedly emphasised the point that the dummy variables in the rules $\exists$e and $\forall$i must not occur outside their boxes. Here is an example which shows how things would go wrong if we didn't have this side condition. Consider the invalid sequent $\exists x\, P(x), \forall x\, (P(x) \to Q(x)) \vdash \forall y\, Q(y)$. (Compare it with the previous sequent; the second premise is now much weaker, allowing us to conclude Q only for those objects for which we know P.) Here is an alleged 'proof' of its validity:

1		$\exists x\, P(x)$	premise
2		$\forall x\, (P(x) \to Q(x))$	premise
3	x_0		
4	x_0	$P(x_0)$	assumption
5		$P(x_0) \to Q(x_0)$	$\forall x$ e 2
6		$Q(x_0)$	$\to$e 5, 4
7		$Q(x_0)$	$\exists x$ e 1, 4–6
8		$\forall y\, Q(y)$	$\forall y$ i 3–7

The last step introducing $\forall y$ is *not* the bad one; that step is fine. The bad one is the second from last one, concluding $Q(x_0)$ by $\exists x\,\mathrm{e}$ and violating the side condition that x_0 may not leave the scope of its box. You can try a few other ways of 'proving' this sequent, but none of them should work (assuming that our proof system is sound with respect to semantic entailment, which we define in the next section). Without this side condition, we would also be able to prove that 'all x satisfy the property P as soon as one of them does so,' a semantic disaster of biblical proportions!

2.3.2 Quantifier equivalences

We have already hinted at semantic equivalences between certain forms of quantification. Now we want to provide formal proofs for some of the most commonly used quantifier equivalences. Quite a few of them involve several quantifications over more than just one variable. Thus, this topic is also good practice for using the proof rules for quantifiers in a nested fashion.

For example, the formula $\forall x\,\forall y\,\phi$ should be equivalent to $\forall y\,\forall x\,\phi$ since both say that ϕ should hold for all values of x and y. What about $(\forall x\,\phi) \wedge (\forall x\,\psi)$ versus $\forall x\,(\phi \wedge \psi)$? A moment's thought reveals that they should have the same meaning as well. But what if the second conjunct does not start with $\forall x$? So what if we are looking at $(\forall x\,\phi) \wedge \psi$ in general and want to compare it with $\forall x\,(\phi \wedge \psi)$? Here we need to be careful, since x might be free in ψ and would then become bound in the formula $\forall x\,(\phi \wedge \psi)$.

Example 2.12 We may specify 'Not all birds can fly.' as $\neg\forall x\,(B(x) \rightarrow F(x))$ or as $\exists x\,(B(x) \wedge \neg F(x))$. The former formal specification is closer to the structure of the English specification, but the latter is logically equivalent to the former. Quantifier equivalences help us in establishing that specifications that 'look' different are really saying the same thing.

Here are some quantifier equivalences which you should become familiar with. As in Chapter 1, we write $\phi_1 \dashv\vdash \phi_2$ as an abbreviation for the validity of $\phi_1 \vdash \phi_2$ and $\phi_2 \vdash \phi_1$.

Theorem 2.13 Let ϕ and ψ be formulas of predicate logic. Then we have the following equivalences:

1. (a) $\neg\forall x\,\phi \dashv\vdash \exists x\,\neg\phi$
   (b) $\neg\exists x\,\phi \dashv\vdash \forall x\,\neg\phi$.
2. Assuming that x is not free in ψ:

(a) $\forall x\,\phi \wedge \psi \dashv\vdash \forall x\,(\phi \wedge \psi)$[3]
(b) $\forall x\,\phi \vee \psi \dashv\vdash \forall x\,(\phi \vee \psi)$
(c) $\exists x\,\phi \wedge \psi \dashv\vdash \exists x\,(\phi \wedge \psi)$
(d) $\exists x\,\phi \vee \psi \dashv\vdash \exists x\,(\phi \vee \psi)$
(e) $\forall x\,(\psi \to \phi) \dashv\vdash \psi \to \forall x\,\phi$
(f) $\exists x\,(\phi \to \psi) \dashv\vdash \forall x\,\phi \to \psi$
(g) $\forall x\,(\phi \to \psi) \dashv\vdash \exists x\,\phi \to \psi$
(h) $\exists x\,(\psi \to \phi) \dashv\vdash \psi \to \exists x\,\phi$.

3. (a) $\forall x\,\phi \wedge \forall x\,\psi \dashv\vdash \forall x\,(\phi \wedge \psi)$
   (b) $\exists x\,\phi \vee \exists x\,\psi \dashv\vdash \exists x\,(\phi \vee \psi)$.
4. (a) $\forall x\,\forall y\,\phi \dashv\vdash \forall y\,\forall x\,\phi$
   (b) $\exists x\,\exists y\,\phi \dashv\vdash \exists y\,\exists x\,\phi$.

PROOF: We will prove most of these sequents; the proofs for the remaining ones are straightforward adaptations and are left as exercises. Recall that we sometimes write $\bot$ to denote any contradiction.

1. (a) We will lead up to this by proving the validity of two simpler sequents first: $\neg(p_1 \wedge p_2) \vdash \neg p_1 \vee \neg p_2$ and then $\neg\forall x\,P(x) \vdash \exists x\,\neg P(x)$. The reason for proving the first of these is to illustrate the close relationship between $\wedge$ and $\vee$ on the one hand and $\forall$ and $\exists$ on the other – think of a model with just two elements 1 and 2 such that p_i $(i = 1, 2)$ stands for $P(x)$ evaluated at i. The idea is that proving this propositional sequent should give us inspiration for proving the second one of predicate logic. The reason for proving the latter sequent is that it is a special case (in which ϕ equals $P(x)$) of the one we are really after, so again it should be simpler while providing some inspiration. So, let's go.

1	$\neg(p_1 \wedge p_2)$			premise
2	$\neg(\neg p_1 \vee \neg p_2)$			assumption
3	$\neg p_1$	assumption	$\neg p_2$	assumption
4	$\neg p_1 \vee \neg p_2$	$\vee\mathrm{i}_1$ 3	$\neg p_1 \vee \neg p_2$	$\vee\mathrm{i}_2$ 3
5	$\bot$	$\neg$e 4, 2	$\bot$	$\neg$e 4, 2
6	p_1	PBC 3–5	p_2	PBC 3–5
7	$p_1 \wedge p_2$			$\wedge$i 6, 6
8	$\bot$			$\neg$e 7, 1
9	$\neg p_1 \vee \neg p_2$			PBC 2–8

[3] Remember that $\forall x\,\phi \wedge \psi$ is implicitly bracketed as $(\forall x\,\phi) \wedge \psi$, by virtue of the binding priorities.

You have seen this sort of proof before, in Chapter 1. It is an example of something which requires proof by contradiction, or ¬¬e, or LEM (meaning that it simply cannot be proved in the reduced natural deduction system which discards these three rules) – in fact, the proof above used the rule PBC three times.

Now we prove the validity of $\neg\forall x\, P(x) \vdash \exists x\, \neg P(x)$ similarly, except that where the rules for $\wedge$ and $\vee$ were used we now use those for $\forall$ and $\exists$:

1		$\neg\forall x\, P(x)$	premise
2		$\neg\exists x\, \neg P(x)$	assumption
3	x_0		
4		$\neg P(x_0)$	assumption
5		$\exists x \neg P(x)$	$\exists x$ i 4
6		$\bot$	¬e 5, 2
7		$P(x_0)$	PBC 4–6
8		$\forall x\, P(x)$	$\forall x$ i 3–7
9		$\bot$	¬e 8, 1
10		$\exists x\, \neg P(x)$	PBC 2–9

You will really benefit by spending time understanding the way this proof mimics the one above it. This insight is very useful for constructing predicate logic proofs: you first construct a similar propositional proof and then mimic it.

Next we prove that $\neg\forall x\, \phi \vdash \exists x\, \neg\phi$ is valid:

1		$\neg\forall x\, \phi$	premise
2		$\neg\exists x\, \neg\phi$	assumption
3	x_0		
4		$\neg\phi[x_0/x]$	assumption
5		$\exists x \neg\phi$	$\exists x$ i 4
6		$\bot$	¬e 5, 2
7		$\phi[x_0/x]$	PBC 4–6
8		$\forall x\, \phi$	$\forall x$ i 3–7
9		$\bot$	¬e 8, 1
10		$\exists x\, \neg\phi$	PBC 2–9

Proving that the reverse $\exists x\, \neg\phi \vdash \neg\forall x\, \phi$ is valid is more straightforward, for it does not involve proof by contradiction, $\neg\neg$e, or LEM. Unlike its converse, it has a constructive proof which the intuitionists do accept. We could again prove the corresponding propositional sequent, but we leave that as an exercise.

1		$\exists x\, \neg\phi$	assumption
2		$\forall x\, \phi$	assumption
3	x_0		
4		$\neg\phi[x_0/x]$	assumption
5		$\phi[x_0/x]$	$\forall x$ e 2
6		$\bot$	$\neg$e 5, 4
7		$\bot$	$\exists x$ e 1, 3–6
8		$\neg\forall x\, \phi$	$\neg$i 2–7

2. (a) Validity of $\forall x\, \phi \wedge \psi \vdash \forall x\, (\phi \wedge \psi)$ can be proved thus:

1		$(\forall x\, \phi) \wedge \psi$	premise
2		$\forall x\, \phi$	$\wedge\text{e}_1$ 1
3		ψ	$\wedge\text{e}_2$ 1
4	x_0		
5		$\phi[x_0/x]$	$\forall x$ e 2
6		$\phi[x_0/x] \wedge \psi$	$\wedge$i 5, 3
7		$(\phi \wedge \psi)[x_0/x]$	identical to 6, since x not free in ψ
8		$\forall x\, (\phi \wedge \psi)$	$\forall x$ i 4–7

The argument for the reverse validity can go like this:

1		$\forall x\, (\phi \wedge \psi)$	premise
2	x_0		
3		$(\phi \wedge \psi)[x_0/x]$	$\forall x$ e 1
4		$\phi[x_0/x] \wedge \psi$	identical to 3, since x not free in ψ
5		ψ	$\wedge\text{e}_2$ 3
6		$\phi[x_0/x]$	$\wedge\text{e}_1$ 3
7		$\forall x\, \phi$	$\forall x$ i 2–6
8		$(\forall x\, \phi) \wedge \psi$	$\wedge$i 7, 5

Notice that the use of $\wedge$i in the last line is permissible, because ψ was obtained for any instantiation of the formula in line 1; although a formal tool for proof support may complain about such practice.

3. (b) The sequent $(\exists x\,\phi) \vee (\exists x\,\psi) \vdash \exists x\,(\phi \vee \psi)$ is proved valid using the rule $\vee$e; so we have two principal cases, each of which requires the rule $\exists x$ i:

1		$(\exists x\,\phi) \vee (\exists x\,\psi)$			premise
2		$\exists x\,\phi$		$\exists x\,\psi$	assumpt.
3	x_0	$\phi[x_0/x]$	x_0	$\psi[x_0/x]$	assumpt.
4		$\phi[x_0/x] \vee \psi[x_0/x]$		$\phi[x_0/x] \vee \psi[x_0/x]$	$\vee$i 3
5		$(\phi \vee \psi)[x_0/x]$		$(\phi \vee \psi)[x_0/x]$	identical
6		$\exists x\,(\phi \vee \psi)$		$\exists x\,(\phi \vee \psi)$	$\exists x$ i 5
7		$\exists x\,(\phi \vee \psi)$		$\exists x\,(\phi \vee \psi)$	$\exists x$ e 2, 3–6
8		$\exists x\,(\phi \vee \psi)$			$\vee$e 1, 2–7

The converse sequent has $\exists x\,(\phi \vee \psi)$ as premise, so its proof has to use $\exists x$ e as its last rule; for that rule, we need $\phi \vee \psi$ as a temporary assumption and need to conclude $(\exists x\,\phi) \vee (\exists x\,\psi)$ from those data; of course, the assumption $\phi \vee \psi$ requires the usual case analysis:

1		$\exists x\,(\phi \vee \psi)$			premise
2	x_0	$(\phi \vee \psi)[x_0/x]$			assumption
3		$\phi[x_0/x] \vee \psi[x_0/x]$			identical
4		$\phi[x_0/x]$		$\psi[x_0/x]$	assumption
5		$\exists x\,\phi$		$\exists x\,\psi$	$\exists x$ i 4
6		$\exists x\,\phi \vee \exists x\,\psi$		$\exists x\,\phi \vee \exists x\,\psi$	$\vee$i 5
7		$\exists x\,\phi \vee \exists x\,\psi$			$\vee$e 3, 4–6
8		$\exists x\,\phi \vee \exists x\,\psi$			$\exists x$ e 1, 2–7

4. (b) Given the premise $\exists x\,\exists y\,\phi$, we have to nest $\exists x$ e and $\exists y$ e to conclude $\exists y\,\exists x\,\phi$. Of course, we have to obey the format of these elimination rules as done below:

1		$\exists x\, \exists y\, \phi$	premise
2	x_0	$(\exists y\, \phi)[x_0/x]$	assumption
3		$\exists y\, (\phi[x_0/x])$	identical, since x, y different variables
4	y_0	$\phi[x_0/x][y_0/y]$	assumption
5		$\phi[y_0/y][x_0/x]$	identical, since x, y, x_0, y_0 different variables
6		$\exists x\, \phi[y_0/y]$	$\forall x$ i 5
7		$\exists y\, \exists x\, \phi$	$\forall y$ i 6
8		$\exists y\, \exists x\, \phi$	$\exists y$ e3, 4–7
9		$\exists y\, \exists x\, \phi$	$\exists x$ e1, 2–8

The validity of the converse sequent is proved in the same way by swapping the roles of x and y. □

2.4 Semantics of predicate logic

Having seen how natural deduction of propositional logic can be extended to predicate logic, let's now look at how the semantics of predicate logic works. Just like in the propositional case, the semantics should provide a separate, but ultimately equivalent, characterisation of the logic. By 'separate,' we mean that the meaning of the connectives is defined in a different way; in proof theory, they were defined by proof rules providing an *operative* explanation. In semantics, we expect something like truth tables. By 'equivalent,' we mean that we should be able to prove soundness and completeness, as we did for propositional logic – although a fully fledged proof of soundness and completeness for predicate logic is beyond the scope of this book.

Before we begin describing the semantics of predicate logic, let us look more closely at the real difference between a semantic and a proof-theoretic account. In proof theory, the basic object which is constructed is a proof. Let us write Γ as a shorthand for lists of formulas $\phi_1, \phi_2, \ldots, \phi_n$. Thus, to show that $\Gamma \vdash \psi$ is valid, we need to provide a proof of ψ from Γ. Yet, how can we show that ψ is not a consequence of Γ? Intuitively, this is harder; how can you possibly show that *there is no proof* of something? You would have to consider every 'candidate' proof and show it is not one. Thus, proof theory gives a 'positive' characterisation of the logic; it provides convincing evidence for assertions like '$\Gamma \vdash \psi$ is valid,' but it is not very useful for establishing evidence for assertions of the form '$\Gamma \vdash \phi$ is not valid.'

Semantics, on the other hand, works in the opposite way. To show that ψ is *not* a consequence of Γ is the 'easy' bit: find a model in which all ϕ_i are true, but ψ isn't. Showing that ψ is a consequence of Γ, on the other hand, is harder in principle. For propositional logic, you need to show that every valuation (an assignment of truth values to all atoms involved) that makes all ϕ_i true also makes ψ true. If there is a small number of valuations, this is not so bad. However, when we look at predicate logic, we will find that there are infinitely many valuations, called *models* from hereon, to consider. Thus, in semantics we have a 'negative' characterisation of the logic. We find establishing assertions of the form '$\Gamma \not\models \psi$' (ψ is not a semantic entailment of all formulas in Γ) easier than establishing '$\Gamma \models \psi$' (ψ is a semantic entailment of Γ), for in the former case we need only talk about one model, whereas in the latter we potentially have to talk about infinitely many.

All this goes to show that it is important to study *both* proof theory *and* semantics. For example, if you are trying to show that ψ is not a consequence of Γ and you have a hard time doing that, you might want to change your strategy for a while by trying to prove the validity of $\Gamma \vdash \psi$. If you find a proof, you know for sure that ψ is a consequence of Γ. If you can't find a proof, then your attempts at proving it often provide insights which lead you to the construction of a counter example. The fact that proof theory and semantics for predicate logic are equivalent is amazing, but it does not stop them having separate roles in logic, each meriting close study.

2.4.1 Models

Recall how we evaluated formulas in propositional logic. For example, the formula $(p \vee \neg q) \to (q \to p)$ is evaluated by computing a truth value (`T` or `F`) for it, based on a given valuation (assumed truth values for p and q). This activity is essentially the construction of one line in the truth table of $(p \vee \neg q) \to (q \to p)$. How can we evaluate formulas in predicate logic, e.g.

$$\forall x\, \exists y\, ((P(x) \vee \neg Q(y)) \to (Q(x) \to P(y)))$$

which 'enriches' the formula of propositional logic above? Could we simply assume truth values for $P(x)$, $Q(y)$, $Q(x)$ and $P(y)$ and compute a truth value as before? Not quite, since we have to reflect the meaning of the quantifiers $\forall x$ and $\exists y$, their *dependences* and the actual parameters of P and Q – a formula $\forall x\, \exists y\, R(x, y)$ generally means something else other than $\exists y\, \forall x\, R(x, y)$; why? The problem is that variables are place holders for any, or some, unspecified concrete values. Such values can be of almost any kind: students, birds, numbers, data structures, programs and so on.

Thus, if we encounter a formula $\exists y\, \psi$, we try to find some instance of y (some concrete value) such that ψ holds for that particular instance of y. If this succeeds (i.e. there is such a value of y for which ψ holds), then $\exists y\, \psi$ evaluates to `T`; otherwise (i.e. there is *no* concrete value of y which realises ψ) it returns `F`. Dually, evaluating $\forall x\, \psi$ amounts to showing that ψ evaluates to `T` for *all* possible values of x; if this is successful, we know that $\forall x\, \psi$ evaluates to `T`; otherwise (i.e. there is *some* value of x such that ψ computes `F`) it returns `F`. Of course, such evaluations of formulas require a fixed universe of concrete values, the things we are, so to speak, talking about. Thus, the truth value of a formula in predicate logic depends on, and varies with, the actual choice of values and the meaning of the predicate and function symbols involved.

If variables can take on only finitely many values, we can write a program that evaluates formulas in a compositional way. If the root node of ϕ is $\wedge$, $\vee$, $\rightarrow$ or $\neg$, we can compute the truth value of ϕ by using the truth table of the respective logical connective and by computing the truth values of the subtree(s) of that root, as discussed in Chapter 1. If the root is a quantifier, we have sketched above how to proceed. This leaves us with the case of the root node being a predicate symbol P (in propositional logic this was an atom and we were done already). Such a predicate requires n arguments which have to be terms $t_1, t_2, \ldots, t_n$. Therefore, we need to be able to assign truth values to formulas of the form $P(t_1, t_2, \ldots, t_n)$.

For formulas $P(t_1, t_2, \ldots, t_n)$, there is more going on than in the case of propositional logic. For $n = 2$, the predicate P could stand for something like 'the number computed by t_1 is less than, or equal to, the number computed by t_2.' Therefore, we cannot just assign truth values to P directly without knowing the meaning of terms. We require a *model* of all function and predicate symbols involved. For example, terms could denote *real numbers* and P could denote the relation 'less than or equal to' on the set of real numbers.

Definition 2.14 Let $\mathcal{F}$ be a set of function symbols and $\mathcal{P}$ a set of predicate symbols, each symbol with a fixed number of required arguments. A model $\mathcal{M}$ of the pair $(\mathcal{F}, \mathcal{P})$ consists of the following set of data:

1. A non-empty set A, the universe of concrete values;
2. for each nullary function symbol $f \in \mathcal{F}$, a concrete element $f^{\mathcal{M}}$ of A
3. for each $f \in \mathcal{F}$ with arity $n > 0$, a concrete function $f^{\mathcal{M}} \colon A^n \rightarrow A$ from A^n, the set of n-tuples over A, to A; and
4. for each $P \in \mathcal{P}$ with arity $n > 0$, a subset $P^{\mathcal{M}} \subseteq A^n$ of n-tuples over A.

The distinction between f and $f^{\mathcal{M}}$ and between P and $P^{\mathcal{M}}$ is most important. The symbols f and P are just that: symbols, whereas $f^{\mathcal{M}}$ and $P^{\mathcal{M}}$ denote a concrete function (or element) and relation in a model $\mathcal{M}$, respectively.

Example 2.15 Let $\mathcal{F} \stackrel{\text{def}}{=} \{i\}$ and $\mathcal{P} \stackrel{\text{def}}{=} \{R, F\}$; where i is a constant, F a predicate symbol with one argument and R a predicate symbol with two arguments. A model $\mathcal{M}$ contains a set of concrete elements A – which may be a set of states of a computer program. The interpretations $i^{\mathcal{M}}$, $R^{\mathcal{M}}$, and $F^{\mathcal{M}}$ may then be a designated initial state, a state transition relation, and a set of final (accepting) states, respectively. For example, let $A \stackrel{\text{def}}{=} \{a, b, c\}$, $i^{\mathcal{M}} \stackrel{\text{def}}{=} a$, $R^{\mathcal{M}} \stackrel{\text{def}}{=} \{(a, a), (a, b), (a, c), (b, c), (c, c)\}$, and $F^{\mathcal{M}} \stackrel{\text{def}}{=} \{b, c\}$. We informally check some formulas of predicate logic for this model:

1. The formula

   $$\exists y\, R(i, y)$$

   says that there is a transition from the initial state to some state; this is true in our model, as there are transitions from the initial state a to a, b, and c.
2. The formula

   $$\neg F(i)$$

   states that the initial state is not a final, accepting state. This is true in our model as b and c are the only final states and a is the intitial one.
3. The formula

   $$\forall x \forall y \forall z\, (R(x, y) \wedge R(x, z) \rightarrow y = z)$$

   makes use of the equality predicate and states that the transition relation is deterministic: all transitions from any state can go to at most one state (there may be no transitions from a state as well). This is false in our model since state a has transitions to b and c.
4. The formula

   $$\forall x \exists y\, R(x, y)$$

   states that the model is free of states that deadlock: all states have a transition to some state. This is true in our model: a can move to a, b or c; and b and c can move to c.

Example 2.16 Let $\mathcal{F} \stackrel{\text{def}}{=} \{e, \cdot\}$ and $\mathcal{P} \stackrel{\text{def}}{=} \{\leq\}$, where e is a constant, $\cdot$ is a function of two arguments and $\leq$ is a predicate in need of two arguments as well. Again, we write $\cdot$ and $\leq$ in infix notation as in $(t_1 \cdot t_2) \leq (t \cdot t)$.

The model $\mathcal{M}$ we have in mind has as set A all binary strings, finite words over the alphabet $\{0, 1\}$, including the empty string denoted by ϵ. The interpretation $e^{\mathcal{M}}$ of e is just the empty word ϵ. The interpretation $\cdot^{\mathcal{M}}$ of $\cdot$ is the concatenation of words. For example, $0110 \cdot^{\mathcal{M}} 1110$ equals 01101110. In general, if $a_1a_2 \ldots a_k$ and $b_1b_2 \ldots b_n$ are such words with $a_i, b_j \in \{0, 1\}$, then $a_1a_2 \ldots a_k \cdot^{\mathcal{M}} b_1b_2 \ldots b_n$ equals $a_1a_2 \ldots a_kb_1b_2 \ldots b_n$. Finally, we interpret $\leq$ as the prefix ordering of words. We say that s_1 is a prefix of s_2 if there is a binary word s_3 such that $s_1 \cdot^{\mathcal{M}} s_3$ equals s_2. For example, 011 is a prefix of 011001 and of 011, but 010 is neither. Thus, $\leq^{\mathcal{M}}$ is the set $\{(s_1, s_2) \mid s_1 \text{ is a prefix of } s_2\}$. Here are again some informal model checks:

1. In our model, the formula
$$\forall x\,((x \leq x \cdot e) \wedge (x \cdot e \leq x))$$
says that every word is a prefix of itself concatenated with the empty word and conversely. Clearly, this holds in our model, for $s \cdot^{\mathcal{M}} \epsilon$ is just s and every word is a prefix of itself.
2. In our model, the formula
$$\exists y\,\forall x\,(y \leq x)$$
says that there exists a word s that is a prefix of every other word. This is true, for we may chose ϵ as such a word (there is no other choice in this case).
3. In our model, the formula
$$\forall x\,\exists y\,(y \leq x)$$
says that every word has a prefix. This is clearly the case and there are in general multiple choices for y, which are dependent on x.
4. In our model, the formula $\forall x\,\forall y\,\forall z\,((x \leq y) \rightarrow (x \cdot z \leq y \cdot z))$ says that whenever a word s_1 is a prefix of s_2, then s_1s has to be a prefix of s_2s for every word s. This is clearly not the case. For example, take s_1 as 01, s_2 as 011 and s to be 0.
5. In our model, the formula
$$\neg\exists x\,\forall y\,((x \leq y) \rightarrow (y \leq x))$$
says that there is no word s such that whenever s is a prefix of some other word s_1, it is the case that s_1 is a prefix of s as well. This is true since there cannot be such an s. Assume, for the sake of argument, that there were such a word s. Then s is clearly a prefix of $s0$, but $s0$ cannot be a prefix of s since $s0$ contains one more bit than s.

It is crucial to realise that the notion of a model is extremely liberal and open-ended. All it takes is to choose a non-empty set A, whose elements

model real-world objects, and a set of concrete functions and relations, one for each function, respectively predicate, symbol. The only mild requirement imposed on all of this is that the concrete functions and relations on A have the same number of arguments as their syntactic counterparts.

However, you, as a designer or implementor of such a model, have the responsibility of choosing your model wisely. Your model should be a sufficiently accurate picture of whatever it is you want to model, but at the same time it should abstract away (= ignore) aspects of the world which are irrelevant from the perspective of your task at hand.

For example, if you build a database of family relationships, then it would be foolish to interpret *father-of*(x, y) by something like 'x is the daughter of y.' By the same token, you probably would not want to have a predicate for 'is taller than,' since your focus in this model is merely on relationships defined by birth. Of course, there are circumstances in which you may want to add additional features to your database.

Given a model $\mathcal{M}$ for a pair $(\mathcal{F}, \mathcal{P})$ of function and predicate symbols, we are now almost in a position to formally compute a truth value for all formulas in predicate logic which involve only function and predicate symbols from $(\mathcal{F}, \mathcal{P})$. There is still one thing, though, that we need to discuss. Given a formula $\forall x\, \phi$ or $\exists x\, \phi$, we intend to check whether ϕ holds for all, respectively some, value a in our model. While this is intuitive, we have no way of expressing this in our syntax: the formula ϕ usually has x as a free variable; $\phi[a/x]$ is well-intended, but ill-formed since $\phi[a/x]$ is *not* a logical formula, for a is not a term but an element of our model.

Therefore we are forced to interpret formulas *relative to an environment*. You may think of environments in a variety of ways. Essentially, they are look-up tables for all variables; such a table l associates with every variable x a value $l(x)$ of the model. So you can also say that environments are functions $l\colon \mathsf{var} \to A$ from the set of variables var to the universe of values A of the underlying model. Given such a look-up table, we can assign truth values to all formulas. However, for some of these computations we need *updated* look-up tables.

Definition 2.17 A look-up table or environment for a universe A of concrete values is a function $l\colon \mathsf{var} \to A$ from the set of variables var to A. For such an l, we denote by $l[x \mapsto a]$ the look-up table which maps x to a and any other variable y to $l(y)$.

Finally, we are able to give a semantics to formulas of predicate logic. For propositional logic, we did this by computing a truth value. Clearly, it suffices to know in which cases this value is $\mathtt{T}$.

Definition 2.18 Given a model $\mathcal{M}$ for a pair $(\mathcal{F}, \mathcal{P})$ and given an environment l, we define the satisfaction relation $\mathcal{M} \vDash_l \phi$ for each logical formula ϕ over the pair $(\mathcal{F}, \mathcal{P})$ and look-up table l by structural induction on ϕ. If $\mathcal{M} \vDash_l \phi$ holds, we say that ϕ computes to $\mathtt{T}$ in the model $\mathcal{M}$ with respect to the environment l.

P: If ϕ is of the form $P(t_1, t_2, \ldots, t_n)$, then we interpret the terms $t_1, t_2, \ldots, t_n$ in our set A by replacing all variables with their values according to l. In this way we compute concrete values $a_1, a_2, \ldots, a_n$ of A for each of these terms, where we interpret any function symbol $f \in \mathcal{F}$ by $f^{\mathcal{M}}$. Now $\mathcal{M} \vDash_l P(t_1, t_2, \ldots, t_n)$ holds iff $(a_1, a_2, \ldots, a_n)$ is in the set $P^{\mathcal{M}}$.

$\forall x$: The relation $\mathcal{M} \vDash_l \forall x\, \psi$ holds iff $\mathcal{M} \vDash_{l[x \mapsto a]} \psi$ holds for all $a \in A$.

$\exists x$: Dually, $\mathcal{M} \vDash_l \exists x\, \psi$ holds iff $\mathcal{M} \vDash_{l[x \mapsto a]} \psi$ holds for some $a \in A$.

$\neg$: The relation $\mathcal{M} \vDash_l \neg\psi$ holds iff it is not the case that $\mathcal{M} \vDash_l \psi$ holds.

$\vee$: The relation $\mathcal{M} \vDash_l \psi_1 \vee \psi_2$ holds iff $\mathcal{M} \vDash_l \psi_1$ or $\mathcal{M} \vDash_l \psi_2$ holds.

$\wedge$: The relation $\mathcal{M} \vDash_l \psi_1 \wedge \psi_2$ holds iff $\mathcal{M} \vDash_l \psi_1$ and $\mathcal{M} \vDash_l \psi_2$ hold.

$\rightarrow$: The relation $\mathcal{M} \vDash_l \psi_1 \rightarrow \psi_2$ holds iff $\mathcal{M} \vDash_l \psi_2$ holds whenever $\mathcal{M} \vDash_l \psi_1$ holds.

We sometimes write $\mathcal{M} \nvDash_l \phi$ to denote that $\mathcal{M} \vDash_l \phi$ does not hold.

There is a straightforward inductive argument on the height of the parse tree of a formula which says that $\mathcal{M} \vDash_l \phi$ holds iff $\mathcal{M} \vDash_{l'} \phi$ holds, whenever l and l' are two environments which are identical on the set of free variables of ϕ. In particular, if ϕ has *no* free variables at all, we then call ϕ a *sentence*; we conclude that $\mathcal{M} \vDash_l \phi$ holds, or does not hold, regardless of the choice of l. Thus, for sentences ϕ we often elide l and write $\mathcal{M} \vDash \phi$ since the choice of an environment l is then irrelevant.

Example 2.19 Let us illustrate the definitions above by means of another simple example. Let $\mathcal{F} \stackrel{\text{def}}{=} \{\mathsf{alma}\}$ and $\mathcal{P} \stackrel{\text{def}}{=} \{\mathsf{loves}\}$ where alma is a constant and loves a predicate with two arguments. The model $\mathcal{M}$ we choose here consists of the privacy-respecting set $A \stackrel{\text{def}}{=} \{a, b, c\}$, the constant function $\mathsf{alma}^{\mathcal{M}} \stackrel{\text{def}}{=} a$ and the predicate $\mathsf{loves}^{\mathcal{M}} \stackrel{\text{def}}{=} \{(a, a), (b, a), (c, a)\}$, which has two arguments as required. We want to check whether the model $\mathcal{M}$ satisfies

> None of Alma's lovers' lovers love her.

First, we need to express the, morally worrying, sentence in predicate logic. Here is such an encoding (as we already discussed, different but logically equivalent encodings are possible):

$$\forall x\, \forall y\, (\mathsf{loves}(x, \mathsf{alma}) \wedge \mathsf{loves}(y, x) \rightarrow \neg\mathsf{loves}(y, \mathsf{alma}))\ . \tag{2.8}$$

Does the model $\mathcal{M}$ satisfy this formula? Well, it does not; for we may choose a for x and b for y. Since (a, a) is in the set $\mathsf{loves}^{\mathcal{M}}$ and (b, a) is in the set $\mathsf{loves}^{\mathcal{M}}$, we would need that the latter does not hold since it is the interpretation of $\mathsf{loves}(y, \mathsf{alma})$; this cannot be.

And what changes if we modify $\mathcal{M}$ to $\mathcal{M}'$, where we keep A and $\mathsf{alma}^{\mathcal{M}}$, but redefine the interpretation of loves as $\mathsf{loves}^{\mathcal{M}'} \stackrel{\text{def}}{=} \{(b, a), (c, b)\}$? Well, now there is exactly one lover of Alma's lovers, namely c; but c is not one of Alma's lovers. Thus, the formula in (2.8) holds in the model $\mathcal{M}'$.

2.4.2 Semantic entailment

In propositional logic, the semantic entailment $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds iff: whenever all $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to $\mathtt{T}$, the formula ψ evaluates to $\mathtt{T}$ as well. How can we define such a notion for formulas in predicate logic, considering that $\mathcal{M} \models_l \phi$ is indexed with an environment?

Definition 2.20 Let Γ be a (possibly infinite) set of formulas in predicate logic and ψ a formula of predicate logic.

1. Semantic entailment $\Gamma \models \psi$ holds iff for all models $\mathcal{M}$ and look-up tables l, whenever $\mathcal{M} \models_l \phi$ holds for all $\phi \in \Gamma$, then $\mathcal{M} \models_l \psi$ holds as well.
2. Formula ψ is satisfiable iff there is some model $\mathcal{M}$ and some environment l such that $\mathcal{M} \models_l \psi$ holds.
3. Formula ψ is valid iff $\mathcal{M} \models_l \psi$ holds for all models $\mathcal{M}$ and environments l in which we can check ψ.
4. The set Γ is consistent or satisfiable iff there is a model $\mathcal{M}$ and a look-up table l such that $\mathcal{M} \models_l \phi$ holds for all $\phi \in \Gamma$.

In predicate logic, the symbol $\models$ is overloaded: it denotes model checks '$\mathcal{M} \models \phi$' and semantic entailment '$\phi_1, \phi_2, \ldots, \phi_n \models \psi$.' Computationally, each of these notions means trouble. First, establishing $\mathcal{M} \models \phi$ will cause problems, if done on a machine, as soon as the universe of values A of $\mathcal{M}$ is infinite. In that case, checking the sentence $\forall x\, \psi$, where x is free in ψ, amounts to verifying $\mathcal{M} \models_{[x \mapsto a]} \psi$ for infinitely many elements a.

Second, and much more seriously, in trying to verify that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds, we have to check things out for *all possible models*, all models which are equipped with the right structure (i.e. they have functions and predicates with the matching number of arguments). This task is impossible to perform mechanically. This should be contrasted to the situation in propositional logic, where the computation of the truth tables for the propositions involved was the basis for computing this relationship successfully.

However, we can sometimes reason that certain semantic entailments are valid. We do this by providing an argument that does not depend on the actual model at hand. Of course, this works only for a very limited number of cases. The most prominent ones are the *quantifier equivalences* which we already encountered in the section on natural deduction. Let us look at a couple of examples of semantic entailment.

Example 2.21 The justification of the semantic entailment

$$\forall x\,(P(x) \to Q(x)) \vDash \forall x\,P(x) \to \forall x\,Q(x)$$

is as follows. Let $\mathcal{M}$ be a model satisfying $\forall x\,(P(x) \to Q(x))$. We need to show that $\mathcal{M}$ satisfies $\forall x\,P(x) \to \forall x\,Q(x)$ as well. On inspecting the definition of $\mathcal{M} \vDash \psi_1 \to \psi_2$, we see that we are done if not every element of our model satisfies P. Otherwise, every element does satisfy P. But since $\mathcal{M}$ satisfies $\forall x\,(P(x) \to Q(x))$, the latter fact forces every element of our model to satisfy Q as well. By combining these two cases (i.e. either all elements of $\mathcal{M}$ satisfy P, or not) we have shown that $\mathcal{M}$ satisfies $\forall x\,P(x) \to \forall x\,Q(x)$.

What about the converse of the above? Is

$$\forall x\,P(x) \to \forall x\,Q(x) \vDash \forall x\,(P(x) \to Q(x))$$

valid as well? Hardly! Suppose that $\mathcal{M}'$ is a model satisfying $\forall x\,P(x) \to \forall x\,Q(x)$. If A' is its underlying set and $P^{\mathcal{M}'}$ and $Q^{\mathcal{M}'}$ are the corresponding interpretations of P and Q, then $\mathcal{M}' \vDash \forall x\,P(x) \to \forall x\,Q(x)$ simply says that, if $P^{\mathcal{M}'}$ equals A', then $Q^{\mathcal{M}'}$ must equal A' as well. However, if $P^{\mathcal{M}'}$ does not equal A', then this implication is vacuously true (remember that $\mathtt{F} \to \cdot = \mathtt{T}$ no matter what $\cdot$ actually is). In this case we do not get any additional constraints on our model $\mathcal{M}'$. After these observations, it is now easy to construct a counter-example model. Let $A' \stackrel{\text{def}}{=} \{a, b\}$, $P^{\mathcal{M}'} \stackrel{\text{def}}{=} \{a\}$ and $Q^{\mathcal{M}'} \stackrel{\text{def}}{=} \{b\}$. Then $\mathcal{M}' \vDash \forall x\,P(x) \to \forall x\,Q(x)$ holds, but $\mathcal{M}' \vDash \forall x\,(P(x) \to Q(x))$ does not.

2.4.3 The semantics of equality

We have already pointed out the open-ended nature of the semantics of predicate logic. Given a predicate logic over a set of function symbols $\mathcal{F}$ and a set of predicate symbols $\mathcal{P}$, we need only a non-empty set A equipped with concrete functions or elements $f^{\mathcal{M}}$ (for $f \in \mathcal{F}$) and concrete predicates $P^{\mathcal{M}}$ (for $P \in \mathcal{P}$) in A which have the right arities agreed upon in our specification. Of course, we also stressed that most models have natural interpretations of

functions and predicates, but central notions like that of semantic entailment ($\phi_1, \phi_2, \ldots, \phi_n \models \psi$) really depend on *all possible models*, even the ones that don't seem to make any sense.

Apparently there is no way out of this peculiarity. For example, where would you draw the line between a model that makes sense and one that doesn't? And would any such choice, or set of criteria, not be *subjective*? Such constraints could also forbid a modification of your model if this alteration were caused by a slight adjustment of the problem domain you intended to model. You see that there are a lot of good reasons for maintaining such a liberal stance towards the notion of models in predicate logic.

However, there is one famous exception. Often one presents predicate logic such that there is always a special predicate $=$ available to denote equality (recall Section 2.3.1); it has two arguments and $t_1 = t_2$ has the intended meaning that the terms t_1 and t_2 compute the same thing. We discussed its proof rule in natural deduction already in Section 2.3.1.

Semantically, one recognises the special role of equality by imposing on an interpretation function $=^{\mathcal{M}}$ to be actual equality on the set A of $\mathcal{M}$. Thus, (a, b) is in the set $=^{\mathcal{M}}$ iff a and b are the same elements in the set A. For example, given $A \stackrel{\text{def}}{=} \{a, b, c\}$, the interpretation $=^{\mathcal{M}}$ of equality is forced to be $\{(a, a), (b, b), (c, c)\}$. Hence the semantics of equality is easy, for it is always modelled *extensionally*.

2.5 Undecidability of predicate logic

We continue our introduction to predicate logic with some negative results. Given a formula ϕ in *propositional logic* we can, at least in principle, determine whether $\models \phi$ holds: if ϕ has n propositional atoms, then the truth table of ϕ contains 2^n lines; and $\models \phi$ holds if, and only if, the column for ϕ (of length 2^n) contains only `T` entries.

The bad news is that such a mechanical procedure, working for all formulas ϕ, cannot be provided in *predicate logic*. We will give a formal proof of this negative result, though we rely on an informal (yet intuitive) notion of computability.

The problem of determining whether a predicate logic formula is valid is known as a *decision problem*. A solution to a decision problem is a program (written in Java, C, or any other common language) that takes problem instances as input and *always* terminates, producing a correct 'yes' or 'no' output. In the case of the decision problem for predicate logic, the input to the program is an arbitrary formula ϕ of predicate logic and the program

is correct if it produces 'yes' whenever the input formula is valid and 'no' whenever it is not. Note that the program which solves a decision problem must terminate for all well-formed input: a program which goes on thinking about it for ever is not allowed. The decision problem at hand is this:

Validity in predicate logic. *Given a logical formula ϕ in predicate logic, does $\vDash \phi$ hold, yes or no?*

We now show that this problem is not solvable; we cannot write a correct C or Java program that works for *all* ϕ. It is important to be clear about exactly what we are stating. Naturally, there are some ϕ which can easily be seen to be valid; and others which can easily be seen to be invalid. However, there are also some ϕ for which it is not easy. Every ϕ can, in principle, be discovered to be valid or not, if you are prepared to work arbitrarily hard at it; but there is no *uniform* mechanical procedure for determining whether ϕ is valid which will work for *all* ϕ.

We prove this by a well-known technique called *problem reduction*. That is, we take some other problem, of which we already know that it is not solvable, and we then show that the solvability of *our* problem entails the solvability of the other one. This is a beautiful application of the proof rules ¬i and ¬e, since we can then infer that our own problem cannot be solvable as well.

The problem that is known not to be solvable, the *Post correspondence problem*, is interesting in its own right and, upon first reflection, does not seem to have a lot to do with predicate logic.

The Post correspondence problem. *Given a finite sequence of pairs $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ such that all s_i and t_i are binary strings of positive length, is there a sequence of indices $i_1, i_2, \ldots, i_n$ with $n \geq 1$ such that the concatenation of strings $s_{i_1} s_{i_2} \ldots s_{i_n}$ equals $t_{i_1} t_{i_2} \ldots t_{i_n}$?*

Here is an *instance* of the problem which we can solve successfully: the concrete correspondence problem instance C is given by a sequence of three pairs $C \stackrel{\text{def}}{=} ((1, 101), (10, 00), (011, 11))$ so

$$\begin{array}{lll} s_1 \stackrel{\text{def}}{=} 1 & s_2 \stackrel{\text{def}}{=} 10 & s_3 \stackrel{\text{def}}{=} 011 \\ t_1 \stackrel{\text{def}}{=} 101 & t_2 \stackrel{\text{def}}{=} 00 & t_3 \stackrel{\text{def}}{=} 11. \end{array}$$

A solution to the problem is the sequence of indices $(1, 3, 2, 3)$ since $s_1 s_3 s_2 s_3$ and $t_1 t_3 t_2 t_3$ both equal 101110011. Maybe you think that this problem must surely be solvable; but remember that a computational solution would have

to be a program that solves *all* such problem instances. Things get a bit tougher already if we look at this (solvable) problem:

$$\begin{array}{cccc} s_1 \stackrel{\text{def}}{=} 001 & s_2 \stackrel{\text{def}}{=} 01 & s_3 \stackrel{\text{def}}{=} 01 & s_4 \stackrel{\text{def}}{=} 10 \\ t_1 \stackrel{\text{def}}{=} 0 & t_2 \stackrel{\text{def}}{=} 011 & t_3 \stackrel{\text{def}}{=} 101 & t_4 \stackrel{\text{def}}{=} 001 \end{array}$$

which you are invited to solve by hand, or by writing a program for this specific instance.

Note that the same number can occur in the sequence of indices, as happened in the first example in which 3 occurs twice. This means that the search space we are dealing with is infinite, which should give us some indication that the problem is unsolvable. However, we do not formally prove it in this book. The proof of the following theorem is due to the mathematician A. Church.

Theorem 2.22 The decision problem of validity in predicate logic is undecidable: no program exists which, given any ϕ, decides whether $\models \phi$.

Proof: As said before, we pretend that validity is decidable for predicate logic and thereby solve the (insoluble) Post correspondence problem. Given a correspondence problem instance C:

$$\begin{array}{c} s_1\ s_2\ \ldots\ s_k \\ t_1\ t_2\ \ldots\ t_k \end{array}$$

we need to be able to construct, within finite space and time and uniformly so for all instances, some formula ϕ of predicate logic such that $\models \phi$ holds iff the correspondence problem instance C above has a solution.

As function symbols, we choose a constant e and two function symbols f_0 and f_1 each of which requires one argument. We think of e as the empty string, or word, and f_0 and f_1 symbolically stand for concatenation with 0, respectively 1. So if $b_1 b_2 \ldots b_l$ is a binary string of bits, we can code that up as the term $f_{b_l}(f_{b_{l-1}} \ldots (f_{b_2}(f_{b_1}(e))) \ldots)$. Note that this coding spells that word *backwards*. To facilitate reading those formulas, we abbreviate terms like $f_{b_l}(f_{b_{l-1}} \ldots (f_{b_2}(f_{b_1}(t))) \ldots)$ by $f_{b_1 b_2 \ldots b_l}(t)$.

We also require a predicate symbol P which expects two arguments. The intended meaning of $P(s,t)$ is that there is some sequence of indices $(i_1, i_2, \ldots, i_m)$ such that s is the term representing $s_{i_1} s_{i_2} \ldots s_{i_m}$ and t represents $t_{i_1} t_{i_2} \ldots t_{i_m}$. Thus, s constructs a string using the same sequence of indices as does t; only s uses the s_i whereas t uses the t_i.

Our sentence ϕ has the coarse structure $\phi_1 \wedge \phi_2 \rightarrow \phi_3$ where we set

$$\phi_1 \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} P(f_{s_i}(e), f_{t_i}(e))$$

$$\phi_2 \stackrel{\text{def}}{=} \forall v \, \forall w \left(P(v, w) \rightarrow \bigwedge_{i=1}^{k} P(f_{s_i}(v), f_{t_i}(w)) \right)$$

$$\phi_3 \stackrel{\text{def}}{=} \exists z \, P(z, z) \; .$$

Our claim is $\vDash \phi$ holds iff the Post correspondence problem C has a solution.

First, let us assume that $\vDash \phi$ holds. Our strategy is to find a model for ϕ which tells us there is a solution to the correspondence problem C simply by inspecting what it means for ϕ to satisfy that particular model. The universe of concrete values A of that model is the set of all finite, binary strings (including the empty string denoted by ϵ).

The interpretation $e^{\mathcal{M}}$ of the constant e is just that empty string ϵ. The interpretation of f_0 is the unary function $f_0^{\mathcal{M}}$ which appends a 0 to a given string, $f_0^{\mathcal{M}}(s) \stackrel{\text{def}}{=} s0$; similarly, $f_1^{\mathcal{M}}(s) \stackrel{\text{def}}{=} s1$ appends a 1 to a given string. The interpretation of P on $\mathcal{M}$ is just what we expect it to be:

$$P^{\mathcal{M}} \stackrel{\text{def}}{=} \{(s, t) \mid \text{there is a sequence of indices } (i_1, i_2, \ldots, i_m) \text{ such that } s \text{ equals } s_{i_1} s_{i_2} \ldots s_{i_m} \text{ and } t \text{ equals } t_{i_1} t_{i_2} \ldots t_{i_m}\}$$

where s and t are binary strings and the s_i and t_i are the data of the correspondence problem C. A pair of strings (s, t) lies in $P^{\mathcal{M}}$ iff, using the same sequence of indices $(i_1, i_2, \ldots, i_m)$, s is built using the corresponding s_i and t is built using the respective t_i.

Since $\vDash \phi$ holds we infer that $\mathcal{M} \vDash \phi$ holds, too. We claim that $\mathcal{M} \vDash \phi_2$ holds as well, which says that whenever the pair (s, t) is in $P^{\mathcal{M}}$, then the pair $(s\, s_i,\, t\, t_i)$ is also in $P^{\mathcal{M}}$ for $i = 1, 2, \ldots, k$ (you can verify that is says this by inspecting the definition of $P^{\mathcal{M}}$). Now $(s, t) \in P^{\mathcal{M}}$ implies that there is some sequence $(i_1, i_2, \ldots, i_m)$ such that s equals $s_{i_1} s_{i_2} \ldots s_{i_m}$ and t equals $t_{i_1} t_{i_2} \ldots t_{i_m}$. We simply choose the new sequence $(i_1, i_2, \ldots, i_m, i)$ and observe that $s\, s_i$ equals $s_{i_1} s_{i_2} \ldots s_{i_m} s_i$ and $t\, t_i$ equals $t_{i_1} t_{i_2} \ldots t_{i_m} t_i$ and so $\mathcal{M} \vDash \phi_2$ holds as claimed. (Why does $\mathcal{M} \vDash \phi_1$ hold?)

Since $\mathcal{M} \vDash \phi_1 \wedge \phi_2 \rightarrow \phi_3$ and $\mathcal{M} \vDash \phi_1 \wedge \phi_2$ hold, it follows that $\mathcal{M} \vDash \phi_3$ holds as well. By definition of ϕ_3 and $P^{\mathcal{M}}$, this tells us there is a solution to C.

Conversely, let us assume that the Post correspondence problem C has some solution, namely the sequence of indices $(i_1, i_2, \ldots, i_n)$. Now we have to show that, if $\mathcal{M}'$ is *any* model having a constant $e^{\mathcal{M}'}$, two unary functions,

$f_0^{\mathcal{M}'}$ and $f_1^{\mathcal{M}'}$, and a binary predicate $P^{\mathcal{M}'}$, then that model has to satisfy ϕ. Notice that the root of the parse tree of ϕ is an implication, so this is the crucial clause for the definition of $\mathcal{M}' \vDash \phi$. By that very definition, we are already done if $\mathcal{M}' \nvDash \phi_1$, or if $\mathcal{M}' \nvDash \phi_2$. The harder part is therefore the one where $\mathcal{M}' \vDash \phi_1 \wedge \phi_2$, for in that case we need to verify $\mathcal{M}' \vDash \phi_3$ as well. The way we proceed here is by *interpreting* finite, binary strings in the domain of values A' of the model $\mathcal{M}'$. This is not unlike the coding of an interpreter for one programming language in another. The interpretation is done by a function $\mathsf{interpret}$ which is defined inductively on the data structure of finite, binary strings:

$$\begin{aligned}
\mathsf{interpret}(\epsilon) &\stackrel{\text{def}}{=} e^{\mathcal{M}'} \\
\mathsf{interpret}(s0) &\stackrel{\text{def}}{=} f_0^{\mathcal{M}'}(\mathsf{interpret}(s)) \\
\mathsf{interpret}(s1) &\stackrel{\text{def}}{=} f_1^{\mathcal{M}'}(\mathsf{interpret}(s)) \ .
\end{aligned}$$

Note that $\mathsf{interpret}(s)$ is defined inductively on the length of s. This interpretation is, like the coding above, backwards; for example, the string 0100110 gets interpreted as $f_0^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_0^{\mathcal{M}'}(f_0^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_0^{\mathcal{M}'}(e^{\mathcal{M}'}))))))))$. Note that $\mathsf{interpret}(b_1 b_2 \ldots b_l) = f_{b_l}^{\mathcal{M}'}(f_{b_{l-1}}^{\mathcal{M}'}(\ldots(f_{b_1}(e^{\mathcal{M}'})\ldots)))$ is just the meaning of $f_s(e)$ in A', where s equals $b_1 b_2 \ldots b_l$. Using that and the fact that $\mathcal{M}' \vDash \phi_1$, we conclude that $(\mathsf{interpret}(s_i), \mathsf{interpret}(t_i)) \in P^{\mathcal{M}'}$ for $i = 1, 2, \ldots, k$. Similarly, since $\mathcal{M}' \vDash \phi_2$, we know that for all $(s,t) \in P^{\mathcal{M}'}$ we have that $(\mathsf{interpret}(ss_i), \mathsf{interpret}(tt_i)) \in P^{\mathcal{M}'}$ for $i = 1, 2, \ldots, k$. Using these two facts, starting with $(s,t) = (s_{i_1}, t_{i_1})$, we repeatedly use the latter observation to obtain

$$(\mathsf{interpret}(s_{i_1} s_{i_2} \ldots s_{i_n}), \mathsf{interpret}(t_{i_1} t_{i_2} \ldots t_{i_n})) \in P^{\mathcal{M}'}. \tag{2.9}$$

Since $s_{i_1} s_{i_2} \ldots s_{i_n}$ and $t_{i_1} t_{i_2} \ldots t_{i_n}$ together form a solution of C, they are equal; and therefore $\mathsf{interpret}(s_{i_1} s_{i_2} \ldots s_{i_n})$ and $\mathsf{interpret}(t_{i_1} t_{i_2} \ldots t_{i_n})$ are the same elements in A', for interpreting the same thing gets you the same result. Hence (2.9) verifies $\exists z\, P(z,z)$ in $\mathcal{M}'$ and thus $\mathcal{M}' \vDash \phi_3$. □

There are two more negative results which we now get quite easily. Recall that a formula ϕ is *satisfiable* if there is some model $\mathcal{M}$ and some environment l such that $\mathcal{M} \vDash_l \phi$ holds. This property is not to be taken for granted; the formula $\exists x\,(P(x) \wedge \neg P(x))$ is clearly unsatisfiable. More interesting is the observation that ϕ is unsatisfiable if, and only if, $\neg\phi$ is valid, i.e. holds in *all* models. This is an immediate consequence of the definitional clause $\mathcal{M} \vDash_l \neg\phi$ for negation. Since we can't compute validity, it follows that we cannot compute satisfiability either.

The other undecidability result comes from the soundness and completeness of predicate logic which, in special form for sentences, reads as

$$\models \phi \quad \text{iff} \quad \vdash \phi \tag{2.10}$$

which we do not prove in this text. Since we can't decide validity, we cannot decide *provability* either, on the basis of (2.10). One might reflect on that last negative result a bit. It means bad news if one wants to implement perfect theorem provers which can mechanically produce a proof of a given formula, or refute it. It means good news, though, if we like the thought that machines still need a little bit of human help. Creativity seems to have limits if we leave it to machines alone.

2.6 Expressiveness of predicate logic

Predicate logic is much more expressive than propositional logic, having predicate and function symbols, as well as quantifiers. This expressivess comes at the cost of making validity, satisfiability and provability undecidable. The good news, though, is that checking formulas on models is practical; SQL queries over relational databases or XQueries over XML documents are examples of this in practice.

Software models, design standards, and execution models of hardware or programs often are described in terms of directed graphs. Such models $\mathcal{M}$ are interpretations of a two-argument predicate symbol R over a concrete set A of 'states.'

Example 2.23 Given a set of states $A = \{s_0, s_1, s_2, s_3\}$, let $R^{\mathcal{M}}$ be the set $\{(s_0, s_1), (s_1, s_0), (s_1, s_1), (s_1, s_2), (s_2, s_0), (s_3, s_0), (s_3, s_2)\}$. We may depict this model as a directed graph in Figure 2.5, where an edge (a transition) leads from a node s to a node s' iff $(s, s') \in R^{\mathcal{M}}$. In that case, we often denote this as $s \to s'$.

The validation of many applications requires to show that a 'bad' state cannot be reached from a 'good' state. What 'good' and 'bad' mean will depend on the context. For example, a good state may be one in which an integer expression, say $x * (y - 1)$, evaluates to a value that serves as a safe index into an array `a` of length 10. A bad state would then be one in which this integer expression evaluates to an unsafe value, say 11, causing an 'out-of-bounds exception.' In its essence, deciding whether from a good state one can reach a bad state is the *reachability* problem in directed graphs.

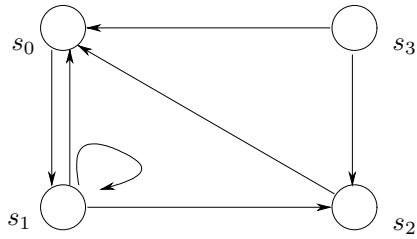

Figure 2.5. A directed graph, which is a model $\mathcal{M}$ for a predicate symbol R with two arguments. A pair of nodes (n, n') is in the interpretation $R^{\mathcal{M}}$ of R iff there is a transition (an edge) from node n to node n' in that graph.

Reachability: Given nodes n and n' in a directed graph, is there a finite path of transitions from n to n'?

In Figure 2.5, state s_2 is reachable from state s_0, e.g. through the path $s_0 \to s_1 \to s_2$. By convention, every state reaches itself by a path of length 0. State s_3, however, is not reachable from s_0; only states s_0, s_1, and s_2 are reachable from s_0. Given the evident importance of this concept, can we express reachability in predicate logic – which is, after all, so expressive that it is undecidable? To put this question more precisely: can we find a predicate-logic formula ϕ with u and v as its only free variables and R as its only predicate symbol (of arity 2) such that ϕ holds in directed graphs iff there is a path in that graph from the node associated to u to the node associated to v? For example, we might try to write:

$$u = v \vee \exists x(R(u, x) \wedge R(x, v)) \vee \exists x_1 \exists x_2(R(u, x_1) \wedge R(x_1, x_2) \wedge R(x_2, v)) \vee \ldots$$

This is infinite, so it's not a well-formed formula. The question is: can we find a well-formed formula with the same meaning?

Surprisingly, this is not the case. To show this we need to record an important consequence of the completeness of natural deduction for predicate logic.

Theorem 2.24 (Compactness Theorem) Let Γ be a set of sentences of predicate logic. If all finite subsets of Γ are satisfiable, then so is Γ.

Proof: We use proof by contradiction: Assume that Γ is not satisfiable. Then the semantic entailment $\Gamma \vDash \bot$ holds as there is no model in which all $\phi \in \Gamma$ are true. By completeness, this means that the sequent $\Gamma \vdash \bot$ is valid. (Note that this uses a slightly more general notion of sequent in which we may have infinitely many premises at our disposal. Soundness and

completeness remain true for that reading.) Thus, this sequent has a proof in natural deduction; this proof – being a finite piece of text – can use only finitely many premises Δ from Γ. But then $\Delta \vdash \bot$ is valid, too, and so $\Delta \models \bot$ follows by soundness. But the latter contradicts the fact that all finite subsets of Γ are consistent. □

From this theorem one may derive a number of useful techniques. We mention a technique for ensuring the existence of models of infinite size.

Theorem 2.25 (Löwenheim-Skolem Theorem) Let ψ be a sentence of predicate logic such for any natural number $n \geq 1$ there is a model of ψ with at least n elements. Then ψ has a model with infinitely many elements.

Proof: The formula $\phi_n \stackrel{\text{def}}{=} \exists x_1 \exists x_2 \ldots \exists x_n \bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j)$ specifies that there are at least n elements. Consider the set of sentences $\Gamma \stackrel{\text{def}}{=} \{\psi\} \cup \{\phi_n \mid n \geq 1\}$ and let Δ be any if its finite subsets. Let $k \geq 1$ be such that $n \leq k$ for all n with $\phi_n \in \Delta$. Since the latter set is finite, such a k has to exist. By assumption, $\{\psi, \phi_k\}$ is satisfiable; but $\phi_k \rightarrow \phi_n$ is valid for all $n \leq k$ (why?). Therefore, Δ is satisfiable as well. The compactness theorem then implies that Γ is satisfiable by some model $\mathcal{M}$; in particular, $\mathcal{M} \models \psi$ holds. Since $\mathcal{M}$ satisfies ϕ_n for all $n \geq 1$, it cannot have finitely many elements. □

We can now show that reachability is not expressible in predicate logic.

Theorem 2.26 Reachability is not expressible in predicate logic: there is no predicate-logic formula ϕ with u and v as its only free variables and R as its only predicate symbol (of arity 2) such that ϕ holds in directed graphs iff there is a path in that graph from the node associated to u to the node associated to v.

Proof: Suppose there is a formula ϕ expressing the existence of a path from the node associated to u to the node associated to v. Let c and c' be constants. Let ϕ_n be the formula expressing that there is a path of length n from c to c': we define ϕ_0 as $c = c'$, ϕ_1 as $R(c, c')$ and, for $n > 1$,

$$\phi_n \stackrel{\text{def}}{=} \exists x_1 \ldots \exists x_{n-1}(R(c, x_1) \wedge R(x_1, x_2) \wedge \cdots \wedge R(x_{n-1}, c')).$$

Let $\Delta = \{\neg\phi_i \mid i \geq 0\} \cup \{\phi[c/u][c'/v]\}$. All formulas in Δ are sentences and Δ is unsatisfiable, since the 'conjunction' of all sentences in Δ says that there is no path of length 0, no path of length 1, etc. from the node denoted by c to the node denoted by c', but there is a finite path from c to c' as $\phi[c/u][c'/v]$ is true.

However, every finite subset of Δ is satisfiable since there are paths of any finite length. Therefore, by the Compactness Theorem, Δ itself is satisfiable. This is a contradiction. Therefore, there cannot be such a formula ϕ. □

2.6.1 Existential second-order logic

If predicate logic cannot express reachability in graphs, then what can, and at what cost? We seek an extension of predicate logic that can specify such important properties, rather than inventing an entirely new syntax, semantics and proof theory from scratch. This can be realized by applying quantifiers not only to variables, but also to predicate symbols. For a predicate symbol P with $n \geq 1$ arguments, consider formulas of the form

$$\exists\, P\, \phi \tag{2.11}$$

where ϕ is a formula of predicate logic in which P occurs. Formulas of that form are the ones of *existential second-order logic*. An example of arity 2 is

$$\exists P\, \forall x \forall y \forall z\, (C_1 \wedge C_2 \wedge C_3 \wedge C_4) \tag{2.12}$$

where each C_i is a Horn clause[4]

$$\begin{aligned} C_1 &\stackrel{\text{def}}{=} P(x,x) \\ C_2 &\stackrel{\text{def}}{=} P(x,y) \wedge P(y,z) \rightarrow P(x,z) \\ C_3 &\stackrel{\text{def}}{=} P(u,v) \rightarrow \bot \\ C_4 &\stackrel{\text{def}}{=} R(x,y) \rightarrow P(x,y). \end{aligned}$$

If we think of R and P as *two* transition relations on a set of states, then C_4 says that any R-edge is also a P-edge, C_1 states that P is reflexive, C_2 specifies that P is transitive, and C_3 ensures that there is no P-path from the node associated to u to the node associated to v.

Given a model $\mathcal{M}$ with interpretations for all function and predicate symbols of ϕ in (2.11), *except* P, let $\mathcal{M}_T$ be that same model augmented with an interpretation $T \subseteq A \times A$ of P, i.e. $P^{\mathcal{M}_T} = T$. For any look-up table l, the semantics of $\exists P\, \phi$ is then

$$\mathcal{M} \vDash_l \exists P\, \phi \qquad \text{iff} \qquad \text{for some } T \subseteq A \times A,\ \mathcal{M}_T \vDash_l \phi. \tag{2.13}$$

[4] Meaning, a Horn clause after all atomic subformulas are replaced with propositional atoms.

Example 2.27 Let $\exists P\,\phi$ be the formula in (2.12) and consider the model $\mathcal{M}$ of Example 2.23 and Figure 2.5. Let l be a look-up table with $l(u) = s_0$ and $l(v) = s_3$. Does $\mathcal{M} \vDash_l \exists P\,\phi$ hold? For that, we need an interpretation $T \subseteq A \times A$ of P such that $\mathcal{M}_T \vDash_l \forall x \forall y \forall x\,(C_1 \wedge C_2 \wedge C_3 \wedge C_4)$ holds. That is, we need to find a reflexive and transitive relation $T \subseteq A \times A$ that contains $R^{\mathcal{M}}$ but not (s_0, s_3). Please verify that $T \stackrel{\text{def}}{=} \{(s, s') \in A \times A \mid s' \neq s_3\} \cup \{(s_3, s_3)\}$ is such a T. Therefore, $\mathcal{M} \vDash_l \exists P\,\phi$ holds.

In the exercises you are asked to show that the formula in (2.12) holds in a directed graph iff there isn't a finite path from node $l(u)$ to node $l(v)$ in that graph. Therefore, this formula specifies *un*reachability.

2.6.2 Universal second-order logic

Of course, we can negate (2.12) and obtain

$$\forall P\,\exists x \exists y \exists z\,(\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4) \tag{2.14}$$

by relying on the familiar de Morgan laws. This is a formula of *universal second-order logic*. This formula expresses reachability.

Theorem 2.28 Let $\mathcal{M} = (A, R^{\mathcal{M}})$ be any model. Then the formula in (2.14) holds under look-up table l in $\mathcal{M}$ iff $l(v)$ is R-reachable from $l(u)$ in $\mathcal{M}$.

Proof:

1. First, assume that $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\,(\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds for all interpretations T of P. Then it also holds for the interpretation which is the reflexive, transitive closure of $R^{\mathcal{M}}$. But for that T, $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\,(\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ can hold only if $\mathcal{M}_T \vDash_l \neg C_3$ holds, as all other clauses C_i $(i \neq 3)$ are false. But this means that $\mathcal{M}_T \vDash_l P(u, v)$ has to hold. So $(l(u), l(v)) \in T$ follows, meaning that there is a finite path from $l(u)$ to $l(v)$.
2. Conversely, let $l(v)$ be R-reachable from $l(u)$ in $\mathcal{M}$.
   - For any interpretation T of P which is not reflexive, not transitive or does not contain $R^{\mathcal{M}}$ the relation $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\,(\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds, since T makes one of the clauses $\neg C_1$, $\neg C_2$ or $\neg C_4$ true.
   - The other possibility is that T be a reflexive, transitive relation containing $R^{\mathcal{M}}$. Then T contains the reflexive, transitive closure of $R^{\mathcal{M}}$. But $(l(u), l(v))$ is in that closure by assumption. Therefore, $\neg C_3$ is made true in the interpretation T under look-up table l, and so $\mathcal{M}_T \vDash_l \exists x \exists y \exists z\,(\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds.

In summary, $\mathcal{M}_T \models_l \exists x \exists y \exists z\, (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds for all interpretations $T \subseteq A \times A$. Therefore, $\mathcal{M} \models_l \forall P\, \exists x \exists y \exists z\, (\neg C_1 \vee \neg C_2 \vee \neg C_3 \vee \neg C_4)$ holds. □

It is beyond the scope of this text to show that reachability can also be expressed in *existential* second-order logic, but this is indeed the case. It is an important open problem to determine whether existential second-order logic is closed under negation, i.e. whether for all such formulas $\exists P\, \phi$ there is a formula $\exists Q\, \psi$ of existential second-order logic such that the latter is semantically equivalent to the negation of the former.

If we allow existential and universal quantifiers to apply to predicate symbols in the *same* formula, we arrive at fully-fledged second-order logic, e.g.

$$\exists P \forall Q\, (\forall x \forall y\, (Q(x, y) \to Q(y, x)) \to \forall u \forall v\, (Q(u, v) \to P(u, v))). \tag{2.15}$$

We have $\exists P \forall Q\, (\forall x \forall y\, (Q(x, y) \to Q(y, x)) \to \forall u \forall v\, (Q(u, v) \to P(u, v)))$ iff there is some T such that for all U we have $(\mathcal{M}_T)_U \vDash \forall x \forall y\, (Q(x, y) \to Q(y, x)) \to \forall u \forall v\, (Q(u, v) \to P(u, v))$, the latter being a model check in first-order logic.

If one wants to quantify over relations of relations, one gets third-order logic etc. Higher-order logics require great care in their design. Typical results such as completeness and compactness may quickly fail to hold. Even worse, a naive higher-order logic may be inconsistent at the meta-level. Related problems were discovered in naive set theory, e.g. in the attempt to define the 'set' A that contains as elements those sets X that do not contain themselves as an element:

$$A \stackrel{\text{def}}{=} \{X \mid X \notin X\}. \tag{2.16}$$

We won't study higher-order logics in this text, but remark that many theorem provers or deductive frameworks rely on higher-order logical frameworks.

2.7 Micromodels of software

Two of the central concepts developed so far are

- *model checking*: given a formula ϕ of predicate logic and a matching model $\mathcal{M}$ determine whether $\mathcal{M} \vDash \phi$ holds; and
- *semantic entailment*: given a set of formulas Γ of predicate logic, is $\Gamma \vDash \phi$ valid?

How can we put these concepts to use in the modelling and reasoning about software? In the case of semantic entailment, Γ should contain all the requirements we impose on a software design and ϕ may be a property we think should hold in *any* implementation that meets the requirements Γ. Semantic entailment therefore matches well with software specification and validation; alas, it is undecidable in general. Since model checking is decidable, why not put all the requirements into a model $\mathcal{M}$ and then check $\mathcal{M} \models \phi$? The difficulty with this approach is that, by comitting to a particular model $\mathcal{M}$, we are comitting to a lot of detail which doesn't form part of the requirements. Typically, the model instantiates a number of parameters which were left free in the requirements. From this point of view, semantic entailment is better, because it allows a variety of models with a variety of different values for those parameters.

We seek to combine semantic entailment and model checking in a way which attempts to give us the advantages of both. We will extract from the requirements a relatively small number of small models, and check that they satisfy the property ϕ to be proved. This satisfaction checking has the tractability of model checking, while the fact that we range over a set of models (albeit a small one) allows us to consider different values of parameters which are not set in the requirements.

This approach is implemented in a tool called Alloy, due to D. Jackson. The models we consider are what he calls *'micromodels'* of software.

2.7.1 State machines

We illlustrate this approach by revisiting Example 2.15 from page 125. Its models are *state machines* with $\mathcal{F} = \{i\}$ and $\mathcal{P} = \{R, F\}$, where i is a constant, F a predicate symbol with one argument and R a predicate symbol with two arguments. A (concrete) model $\mathcal{M}$ contains a set of concrete elements A – which may be a set of states of a computer program. The interpretations $i^{\mathcal{M}} \in A$, $R^{\mathcal{M}} \in A \times A$, and $F^{\mathcal{M}} \subseteq A$ are understood to be a designated initial state, a state transition relation, and a set of final (accepting) states, respectively. Model $\mathcal{M}$ is concrete since there is nothing left un-specified and all checks $\mathcal{M} \models \phi$ have definite answers: they either hold or they don't.

In practice not all functional or other requirements of a software system are known in advance, and they are likely to change during its lifecycle. For example, we may not know how many states there will be; and some transitions may be mandatory whereas others may be optional in an implementation. Conceptually, we seek a description $\mathbb{M}$ of all *compliant*

implementations M_i ($i \in I$) of some software system. Given some matching property ψ, we then want to know

- *(assertion checking)* whether ψ holds in all implementations $M_i \in \mathbb{M}$; or
- *(consistency checking)* whether ψ holds in some implementation $M_i \in \mathbb{M}$.

For example, let $\mathbb{M}$ be the set of all concrete models of state machines, as above. A possible assertion check ψ is 'Final states are never initial states.' An example of a consistency check ψ is 'There are state machines that contain a non-final but deadlocked state.'

As remarked earlier, if $\mathbb{M}$ were the set of all state machines, then checking properties would risk being undecidable, and would at least be intractable. If $\mathbb{M}$ consists of a single model, then checking properties would be decidable; but a single model is not general enough. It would comit us to instantiating several parameters which are not part of the requirements of a state machine, such as its size and detailed construction. A better idea is to fix a finite bound on the size of models, and check whether all models of that size that satisfy the requirements also satisfy the property under consideration.

- If we get a positive answer, we are somewhat confident that the property holds in all models. In this case, the answer is not conclusive, because there could be a larger model which fails the property, but nevertheless a positive answer gives us some confidence.
- If we get a negative answer, then we have found a model in $\mathbb{M}$ which violates the property. In that case, we have a conclusive answer, and can inspect the model in question.

D. Jackson's *small scope hypothesis* states that negative answers tend to occur in small models already, boosting the confidence we may have in a positive answer. Here is how one could write the requirements for $\mathbb{M}$ for state machines in Alloy:

```
sig State {}

sig StateMachine {
  A : set State,
  i : A,
  F : set A,
  R : A -> A
}
```

The model specifies two *signatures*. Signature `State` is simple in that it has no internal structure, denoted by {}. Although the states of real systems may

well have internal structure, our Alloy declaration abstracts it away. The second signature `StateMachine` has internal, composite structure, saying that every state machine has a set of states `A`, an initial state `i` from `A`, a set of final states `F` from `A`, and a transition relation `R` of type `A -> A`. If we read `->` as the cartesian product $\times$, we see that this internal structure is simply the structural information needed for models of Example 2.15 (page 125). Concrete models of state machines are *instances* of signature `StateMachine`. It is useful to think of signatures as sets whose elements are the instances of that signature. Elements possess all the structure declared in their signature.

Given these signatures, we can code and check an assertion:

```
assert FinalNotInitial {
  all M : StateMachine | no M.i & M.F
} check FinalNotIntial for 3 but 1 StateMachine
```

declares an assertion named `FinalNotInitial` whose body specifies that for all models `M` of type `StateMachine` the property `no M.i & M.F` is true. Read & for set intersection and `no S` ('there is no S') for 'set `S` is empty.' Alloy identifies elements a with singleton sets $\{a\}$, so this set intersection is well typed. The relational dot operator `.` enables access to the internal components of a state machine: `M.i` is the initial state of `M` and `M.F` is its set of final states etc. Therefore, the expression `no M.i & M.F` states 'No initial state of `M` is also a final state of `M`.' Finally, the `check` directive informs the analyzer of Alloy that it should try to find a counterexample of the assertion `FinalNotInitial` with at most three elements for every signature, except for `StateMachine` which should have at most one.

The results of Alloy's assertion check are shown in Figure 2.7. This visualization has been customized to decorate initial and final states with respective labels `i` and `F`. The transition relation is shown as a labeled graph and there is only one transition (from `State_0` back to `State_0`) in this example. Please verify that this is a counterexample to the claim of the assertion `FinalNotInitial` within the specified scopes. Alloy's GUI lets you search for additional witnesses (here: counterexamples), if they exist.

Similarly, we can check a property of state machines for consistency with our model. Alloy uses the keyword `fun` for consistency checks. e.g.

```
fun AGuidedSimulation(M : StateMachine, s : M.A) {
  no s.(M.R)
  not s in M.F
  # M.A = 3
} run AGiudedSimulation for 3 but 1 StateMachine
```

```
module AboutStateMachines

sig State {}       -- simple states

sig StateMachine { -- composite state machines
  A : set State,   -- set of states of a state machine
  i : A,           -- initial state of a state machine
  F : set A,       -- set of final states of a state machine
  R : A -> A       -- transition relation of a state machine
}

-- Claim that final states are never initial: false.
assert FinalNotInitial {
  all M : StateMachine | no M.i & M.F
} check FinalNotInitial for 3 but 1 StateMachine

-- Is there a three-state machine with a non-final deadlock? True.
fun AGuidedSimulation(M : StateMachine, s : M.A) {
  no s.(M.R)
  not s in M.F
  # M.A = 3
} run AGuidedSimulation for 3 but 1 StateMachine
```

Figure 2.6. The complete Alloy module for models of state machines, with one assertion and one consistency check. The lexeme -- enables comments on the same line.

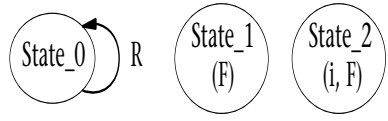

Figure 2.7. Alloy's analyzer finds a state machine model (with one transition only) within the specified scope such that the assertion `FinalNotInitial` is false: the initial state `State_2` is also final.

This consistency check is named `AGuidedSimulation` and followed by an ordered finite list of parameter/type pairs; the first parameter is `M` of type `StateMachine`, the second one is `s` of type `M.A` – i.e. `s` is a state of `M`. The body of a consistency check is a finite list of constraints (here three), which are conjoined implicitly. In this case, we want to find a model with instances of the parameters `M` and `s` such that `s` is a non-final state of `M`, the second constraint `not s in M.F` plus the type information `s : M.A`; and there is no transition out of `s`, the first constraint `no s.(M.R)`.

The latter requires further explanation. The keyword `no` denotes 'there is no;' here it is applied to the set `s.(M.R)`, expressing that there are no

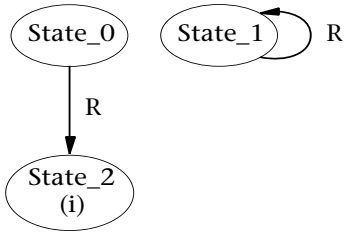

Figure 2.8. Alloy's analyzer finds a state machine model within the specified scope such that the consistency check `AGuidedSimulation` is true: there is a non-final deadlocked state, here `State_2`.

elements in `s.(M.R)`. Since `M.R` is the transition relation of `M`, we need to understand how `s.(M.R)` constructs a set. Well, `s` is an element of `M.A` and `M.R` has type `M.A -> M.A`. Therefore, we may form the set of all elements `s'` such that there is a `M.R`-transition from `s` to `s'`; this is the set `s.(M.R)`. The third constraint states that `M` has exactly three states: in Alloy, `# S = k` declares that the set `S` has exactly k elements.

The `run` directive instructs to check the consistency of `AGuidedSimulation` for at most one state machine and at most three states; the constraint analyzer of Alloy returns the witness (here: an example) of Figure 2.8. Please check that this witness satisfies all constraints of the consistency check and that it is within the specified scopes.

The complete model of state machines with these two checks is depicted in Figure 2.6. The keyword plus name `module AboutStateMachines` identify this under-specified model $\mathbb{M}$, rightly suggesting that Alloy is a modular specification and analysis platform.

2.7.2 Alma – re-visited

Recall Example 2.19 from page 128. Its model had three elements and did not satisfy the formula in (2.8). We can now write a module in Alloy which checks whether all *smaller* models have to satisfy (2.8). The code is given in Figure 2.9. It names the module `AboutAlma` and defines a simple signature of type `Person`. Then it declares a signature `SoapOpera` which has a `cast` – a set of type `Person` – a designated cast member `alma`, and a relation `loves` of type `cast -> cast`. We check the assertion `OfLovers` in a scope of at most two persons and at most one soap opera. The body of that assertion is the typed version of (2.8) and deserves a closer look:

1. Expressions of the form `all x : T | F` state that formula `F` is true for all instances `x` of type `T`. So the assertion states that `with S` {...} is true for all soap operas `S`.

```
module AboutAlma

sig Person {}

sig SoapOpera {
  cast : set Person,
  alma : cast,
  loves : cast -> cast
}

assert OfLovers {
  all S : SoapOpera |
    with S {
      all x, y : cast |
        alma in x.loves && x in y.loves => not alma in y.loves
    }
}
check OfLovers for 2 but 1 SoapOpera
```

Figure 2.9. In this module, the analysis of `OfLovers` checks whether there is a model of ≤ 2 persons and ≤ 1 soap operas for which the query in (2.8), page 128, is false.

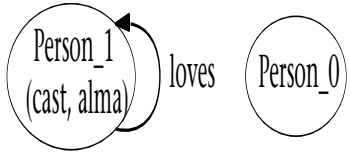

Figure 2.10. Alloy's analyzer finds a counterexample to the formula in (2.8): Alma is the only cast member and loves herself.

2. The expression `with S {...}` is a convenient notation that allows us to write `loves` and `cast` instead of the needed `S.loves` and `S.cast` (respectively) within its curly brackets.
3. Its body `...` states that for all x, and y in the cast of `S`, if `alma` is loved by x and x is loved by y, then – the symbol `=>` expresses implication – `alma` is not loved by y.

Alloy's analysis finds a counterexample to this assertion, shown in Figure 2.10. It is a counterexample since `alma` is her own lover, and therefore also one of her lover's lovers'. Apparently, we have underspecified our model: we implicitly made the domain-specific assumption that self-love makes for

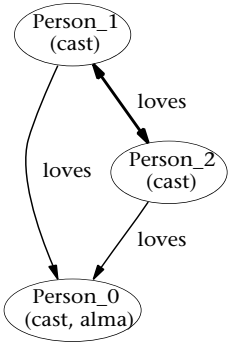

Figure 2.11. Alloy's analyzer finds a counterexample to the formula in (2.8) that meets the constraint of `NoSelfLove` with three cast members. The bidirectional arrow indicates that `Person_1` loves `Person_2` and vice versa.

a poor script of jealousy and intrigue, but *did not rule out* self-love in our Alloy module. To remedy this, we can add a `fact` to the module; facts may have names and restrict the set of possible models: assertions and consistency checks are conducted only over concrete models that satisfy *all* facts of the module. Adding the declaration

```
fact NoSelfLove {
  all S : SoapOpera, p : S.cast | not p in p.(S.loves)
}
```

to the module `AboutAlma` enforces that no member of any soap-opera cast loves him or herself. We re-check the assertion and the analyzer informs us that no solution was found. This suggests that our model from Example 2.19 is indeed a minimal one in the presence of that domain assumption. If we retain that fact, but change the occurrence of 2 in the `check` directive to 3, we get a counterexample, depicted in Figure 2.11. Can you see why it is a counterexample?

2.7.3 A software micromodel

So far we used Alloy to generate instances of models of first-order logic that satisfy certain constraints expressed as formulas of first-order logic. Now we apply Alloy and its constraint analyzer to a more serious task: we model a software system. The intended benefits provided by a system model are

1. it captures formally static and dynamic system structure and behaviour;
2. it can verify consistency of the constrained design space;

3. it is executable, so it allows guided simulations through a potentially very complex design space; and
4. it can boost our confidence into the correctness of claims about static and dynamic aspects of *all* its compliant implementations.

Moreover, formal models attached to software products can be seen as a *reliability contract;* a promise that the software implements the structure and behaviour of the model and is expected to meet all of the assertions certified therein. (However, this may not be very useful for extremely under-specified models.)

We will model a *software package dependency system.* This system is used when software packages are installed or upgraded. The system checks to see if prerequisites in the form of libraries or other packages are present. The requirements on a software package dependency system are not straightforward. As most computer users know, the upgrading process can go wrong in various ways. For example, upgrading a package can involve replacing shared libraries with newer versions. But other packages which rely on the older versions of the shared libraries may then cease to work.

Software package dependency systems are used in several computer systems, such as Red Hat Linux, .NET's Global Assembly Cache and others. Users often have to guess how technical questions get resolved within the dependency system. To the best of our knowledge, there is no publicly available formal and executable model of any particular dependency system to which application programmers could turn if they had such non-trivial technical questions about its inner workings.

In our model, applications are built out of components. Components offer services to other components. A service can be a number of things. Typically, a service is a method (a modular piece of program code), a field entry, or a type – e.g. the type of a class in an object-oriented programming language. Components typically require the import of services from other components. Technically speaking, such import services resolve all un-resolved references within that component, making the component linkable. A component also has a name and may have a special service, called 'main.'

We model components as a signature in Alloy:

```
sig Component {
  name: Name,           -- name of the component
  main: option Service, -- component may have a 'main' service
  export: set Service,  -- services the component exports
  import: set Service,  -- services the component imports
  version: Number       -- version number of the component
}{ no import & export }
```

The signatures `Service` and `Name` won't require any composite structure for our modelling purposes. The signature `Number` will get an ordering later on. A component is an instance of `Component` and therefore has a `name`, a set of services `export` it offers to other components, and a set `import` of services it needs to import from other components. Last but not least, a component has a `version` number. Observe the role of the modifiers `set` and `option` above.

A declaration `i : set S` means that `i` is a subset of set `S`; but a declaration `i : option S` means that `i` is a subset of `S` with *at most one element*. Thus, `option` enables us to model an element that may (non-empty, singleton set) or may not (empty set) be present; a very useful ability indeed. Finally, a declaration `i : S` states that `i` is a subset of `S` containing *exactly one element*; this really specifies a scalar/element of type `S` since Alloy identifies elements a with sets $\{a\}$.

We can constrain all instances of a signature with `C` by adding `{ C }` to its signature declaration. We did this for the signature `Component`, where `C` is the constraint `no import & export`, stating that, in all components, the intersection (`&`) of `import` and `export` is empty (`no`).

A Package Dependency System (PDS) consists of a set of `components`:

```
sig PDS {
  components : set Component
...
}{ components.import in components.export }
```

and other structure that we specify later on. The primary concern in a PDS is that its set of components be *coherent*: at all times, all imports of all of its components can be serviced within that PDS. This requirement is enforced for all instances of `PDS` by adding the constraint `components.import in components.export` to its signature. Here `components` is a *set* of components and Alloy defines the meaning of `components.import` as the *union* of all sets `c.import`, where `c` is an element of `components`. Therefore the requirement states that, for all `c` in `components`, all of `c`'s needed services can be provided by some component in `components` as well. This is exactly the integrity constraint we need for the set of components of a PDS. Observe that this requirement does not specify which component provides which service, which would be an unacceptable imposition on implementation freedom.

Given this integrity constraint we can already model the installation (adding) or removal of a component in a PDS, without having specified the remaining structure of a PDS. This is possible since, in the context of these operations, we may abstract a PDS into its set of `components`. We model

the addition of a component to a PDS as a parametrized `fun`-statement with name `AddComponent` and three parameters

```
fun AddComponent(P, P': PDS, c: Component) {
  not c in P.components
  P'.components = P.components + c
} run AddComponent for 3
```

where `P` is intended to be the PDS *prior* to the execution of that operation, `P'` models the PDS *after* that execution, and `c` models the component that is to be added. This intent interprets the parametric constraint `AddComponent` as an *operation* leading from one 'state' to another (obtained by removing `c` from the PDS `P`). The body of `AddComponent` states two constraints, conjoined implicitly. Thus, this operation applies only if the component `c` is not already in the set of components of the PDS (`not c in P.components`; an example of a *precondition*) and if the PDS adds only `c` and does not lose any other components (`P'.components = P.components + c`; an example of a *postcondition*).

To get a feel for the complexities and vexations of designing software systems, consider our conscious or implicit decision to enforce that all instances of `PDS` have a coherent set of components. This sounds like a very good idea, but what if a 'real' and faulty PDS ever gets to a state in which it is incoherent? We would then be prevented from adding components that may restore its coherence! Therefore, the aspects of our model do not include issues such as repair – which may indeed by an important software management aspect.

The specification for the removal of a component is very similar to the one for `AddComponent`:

```
fun RemoveComponent(P, P': PDS, c: Component) {
  c in P.components
  P'.components = P.components - c
} run RemoveComponent for 3
```

except that the precondition now insists that `c` be in the set of components of the PDS prior to the removal; and the postcondition specifies that the PDS lost component `c` but did not add or lose any other components. The expression `S - T` denotes exactly those 'elements' of `S` that are not in `T`.

It remains to complete the signature for `PDS`. Three additions are made.

1. A relation `schedule` assigns to each PDS component and any of its import services a component in that PDS that provides that service.

```
fact SoundPDSs {
  all P : PDS |
    with P {
    all c : components, s : Service |  --1
      let c' = c.schedule[s] {
      (some c' iff s in c.import) && (some c' => s in c'.export)
     }
    all c : components | c.requires = c.schedule[Service]  --2
    }
}
```

Figure 2.12. A fact that constrains the state and schedulers of all PDSs.

2. Derived from `schedule` we obtain a relation `requires` between components of the PDS that expresses the dependencies between these components based on the `schedule`.
3. Finally, we add constraints that ensure the integrity and correct handling of `schedule` and `requires` for *all instances of* `PDS`.

The complete signature of `PDS` is

```
sig PDS {
  components : set Component,
  schedule   : components -> Service ->? components,
  requires   : components -> components
}
```

For any `P : PDS`, the expression `P.schedule` denotes a relation of type `P.components -> Service ->? P.components`. The `?` is a *multiplicity constraint*, saying that each component of the PDS and each service get related to *at most one* component. This will ensure that the scheduler is deterministic and that it may not schedule anything – e.g. when the service is not needed by the component in the first argument. In Alloy there are also multiplicity markings `!` for 'exactly one' and `+` for 'one or more.' The absence of such markings means 'zero or more.' For example, the declaration of `requires` uses that default reading.

We use a `fact`-statement to constrain even further the structure and behaviour of all PDSs, depicted in Figure 2.12. The fact named `SoundPDSs` quantifies the constraints over all instances of PDSs (`all P : PDS | ...`) and uses `with P {...}` to avoid the use of navigation expressions of the form `P.e`. The body of that fact lists two constraints `--1` and `--2`:

`--1` states two constraints within a `let`-expression of the form `let x = E {...}`. Such a `let`-expression declares all free occurrences of `x` in `{...}` to be equal to `E`. Note that `[]` is a version of the dot operator `.` with lower binding priority, so `c.schedule[s]` is syntactic sugar for `s.(c.schedule)`.

- In the first constraint, component `c` and a service `s` have another component `c'` scheduled (`some c'` is true iff set `c'` is non-empty) if and only if `s` is actually in the import set of `c`. Only needed services are scheduled!
- In the second constraint, if `c'` is scheduled to provide service `s` for `c`, then `s` is in the export set of `c'` – we can only schedule components that can provide the scheduled services!

`--2` defines `requires` in terms of `schedule`: a component `c` requires all those components that are scheduled to provide some service for `c`.

Our complete Alloy model for PDSs is shown in Figure 2.13. Using Alloy's constraint analyzer we validate that all our `fun`-statements, notably the operations of removing and adding components to a PDS, are logically consistent for this design.

The assertion `AddingIsFunctionalForPDSs` claims that the execution of the operation which adds a component to a PDS renders a unique result PDS. Alloy's analyzer finds a counterexample to this claim, where `P` has no components, so nothing is scheduled or required; and `P'` and `P''` have `Component_2` as only component, added to `P`, so this component is required and scheduled in those PDSs.

Since `P'` and `P''` seem to be equal, how can this be a counterexample? Well, we ran the analysis in scope 3, so `PDS = {PDS_0, PDS_1, PDS_2}` and Alloy chose `PDS_0` as `P`, `PDS_1` as `P'`, and `PDS_2` as `P''`. Since the set `PDS` contains three elements, Alloy 'thinks' that they are all different from each other. This is the interpretation of equality enforced by predicate logic. Obviously, what is needed here is a *structural equality of types*: we want to ensure that the addition of a component results into a PDS with unique structure. A `fun`-statement can be used to specify structural equality:

```
fun StructurallyEqual(P, P' : PDS) {
  P.components = P'.components
  P.schedule = P'.schedule
  P.requires = P'.requires
} run StructurallyEqual for 2
```

We then simply replace the expression `P' = P''` in `AdditionIsFunctional` with the expression `StructurallyEqual(P',P'')`, increase the scope for

```
module PDS

open std/ord    -- opens specification template for linear order

sig Component {
  name: Name,
  main: option Service,
  export: set Service,
  import: set Service,
  version: Number
}{ no import & export }

sig PDS {
  components: set Component,
  schedule: components -> Service ->? components,
  requires: components -> components
}{ components.import in components.export }

fact SoundPDSs {
  all P : PDS |
    with P {
    all c : components, s : Service |  --1
      let c' = c.schedule[s] {
        (some c' iff s in c.import) && (some c' => s in c'.export) }
    all c : components | c.requires = c.schedule[Service]  }  --2
}

sig Name, Number, Service {}

fun AddComponent(P, P': PDS, c: Component) {
 not c in P.components
 P'.components = P.components + c
} run AddComponent for 3 but 2 PDS

fun RemoveComponent(P, P': PDS, c : Component) {
  c in P.components
  P'.components = P.components - c
} run RemoveComponent for 3 but 2 PDS

fun HighestVersionPolicy(P: PDS) {
  with P {
    all s : Service, c : components, c' : c.schedule[s],
    c'' : components - c' {
       s in c''.export && c''.name = c'.name =>
         c''.version in c'.version.^(Ord[Number].prev) } }
} run HighestVersionPolicy for 3 but 1 PDS

fun AGuidedSimulation(P,P',P'' : PDS, c1, c2 : Component) {
  AddComponent(P,P',c1)    RemoveComponent(P,P'',c2)
  HighestVersionPolicy(P)  HighestVersionPolicy(P')  HighestVersionPolicy(P'')
} run AGuidedSimulation for 3

assert AddingIsFunctionalForPDSs {
  all P, P', P'': PDS, c: Component   {
    AddComponent(P,P',c) &&
    AddComponent(P,P'',c) => P' = P'' }
} check AddingIsFunctionalForPDSs for 3
```

Figure 2.13. Our Alloy model of the PDS.

that assertion to 7, re-built the model, and re-analyze that assertion. Perhaps surprisingly, we find as counterexample a `PDS_0` with two components `Component_0` and `Component_1` such that `Component_0.import = { Service_2 }` and `Component_1.import = { Service_1 }`. Since `Service_2` is contained in `Component_2.export`, we have two structurally different legitimate post states which are obtained by adding `Component_2` but which differ in their scheduler. In `P'` we have the same scheduling instances as in `PDS_0`. Yet `P''` schedules `Component_2` to provide service `Service_2` for `Component_0`; and `Component_0` still provides `Service_1` to `Component_1`. This analysis reveals that the addition of components creates opportunities to reschedule services, for better (e.g. optimizations) or for worse (e.g. security breaches).

The utility of a micromodel of software resides perhaps more in the ability to explore it through guided simulations, as opposed to verifying some of its properties with absolute certainty. We demonstrate this by generating a simulation that shows the removal and the addition of a component to a PDS such that the scheduler always schedules components with the highest version number possible in all PDSs. Therefore we know that such a scheduling policy is consistent for these two operations; it is by no means the only such policy and is not guaranteed to ensure that applications won't break when using scheduled services. The `fun`-statement

```
fun HighestVersionPolicy(P: PDS) {
  with P {
    all s : Service, c : components, c' : c.schedule[s],
    c'' : components - c' {
       s in c''.export && c''.name = c'.name =>
         c''.version in c'.version.^(Ord[Number].prev)
    }
  }
} run HighestVersionPolicy for 3 but 1 PDS
```

specifies that, among those suppliers with identical name, the scheduler chooses one with the highest available version number. The expression

```
c'.version.^(Ord[Number].prev)
```

needs explaining: `c'.version` is the version number of `c'`, an element of type `Number`. The symbol `^` can be applied to a binary relation `r : T -> T` such that `^r` has again type `T -> T` and denotes the *transitive closure* of `r`. In this case, `T` equals `Number` and `r` equals `Ord[Number].prev`.

But what shall me make of the latter expression? It assumes that the module contains a statement `open std/ord` which opens the signature specifications from another module in file `ord.als` of the library `std`. That module contains a signature named `Ord` which has a type variable as a parameter; it is *polymorphic*. The expression `Ord[Number]` instantiates that type variable with the type `Number`, and then invokes the `prev` relation of that signature with that type, where `prev` is constrained in `std/ord` to be a linear order. The net effect is that we create a linear order on `Number` such that `n.prev` is the previous element of `n` with respect to that order. Therefore, `n.^prev` lists all elements that are smaller than `n` in that order. Please reread the body of that `fun`-statement to convince yourself that it states what is intended.

Since `fun`-statements can be invoked with instances of their parameters, we can write the desired simulation based on `HighestVersionPolicy`:

```
fun AGuidedSimulation(P,P',P'' : PDS, c1, c2 : Component) {
  AddComponent(P,P',c1)   RemoveComponent(P,P'',c2)
  HighestVersionPolicy(P)
  HighestVersionPolicy(P') HighestVersionPolicy(P'')
} run AGuidedSimulation for 3
```

Alloy's analyzer generates a scenario for this simulation, which amounts to two different operation snapshots originating in `P` such that all three participating PDSs schedule according to `HighestVersionPolicy`. Can you spot why we had to work with two components `c1` and `c2`?

We conclude this case study by pointing out limitations of Alloy and its analyzer. In order to be able to use a SAT solver for propositional logic as an analysis engine, we can only check or run formulas of existential or universal second-order logic in the bodies of assertions or in the bodies of `fun`-statements (if they are wrapped in existential quantifiers for all parameters). For example, we cannot even check whether there is an instance of `AddComponent` such that for the resulting PDS a certain scheduling policy is *impossible*. For less explicit reasons it also seems unlikely that we can check in Alloy that every coherent set of components is realizable as `P.components` for some PDS `P`. This deficiency is due to the inherent complexity of such problems and theorem provers may have to be used if such properties need to be guaranteed. On the other hand, the expressiveness of Alloy allows for the rapid prototyping of models and the exploration of simulations and possible counterexamples which should enhance once understanding of a design and so improve that design's reliability.

2.8 Exercises

Exercises 2.1

* 1. Use the predicates

$$\begin{aligned} A(x,y):&\quad x \text{ admires } y\\ B(x,y):&\quad x \text{ attended } y\\ P(x):&\quad x \text{ is a professor}\\ S(x):&\quad x \text{ is a student}\\ L(x):&\quad x \text{ is a lecture} \end{aligned}$$

and the nullary function symbol (constant)

$$m:\quad \text{Mary}$$

to translate the following into predicate logic:

(a) Mary admires every professor.
(The answer is not $\forall x\, A(m, P(x))$.)
(b) Some professor admires Mary.
(c) Mary admires herself.
(d) No student attended every lecture.
(e) No lecture was attended by every student.
(f) No lecture was attended by any student.

2. Use the predicate specifications

$$\begin{aligned} B(x,y):&\quad x \text{ beats } y\\ F(x):&\quad x \text{ is an (American) football team}\\ Q(x,y):&\quad x \text{ is quarterback of } y\\ L(x,y):&\quad x \text{ loses to } y \end{aligned}$$

and the constant symbols

$$\begin{aligned} c:&\quad \text{Wildcats}\\ j:&\quad \text{Jayhawks} \end{aligned}$$

to translate the following into predicate logic.

(a) Every football team has a quarterback.
(b) If the Jayhawks beat the Wildcats, then the Jayhawks do not lose to every football team.
(c) The Wildcats beat some team, which beat the Jayhawks.

* 3. Find appropriate predicates and their specification to translate the following into predicate logic:

(a) All red things are in the box.
(b) Only red things are in the box.
(c) No animal is both a cat and a dog.
(d) Every prize was won by a boy.
(e) A boy won every prize.

4. Let $F(x, y)$ mean that x is the father of y; $M(x, y)$ denotes x is the mother of y. Similarly, $H(x, y)$, $S(x, y)$, and $B(x, y)$ say that x is the husband/sister/brother of y, respectively. You may also use constants to denote individuals, like 'Ed' and 'Patsy.' However, you are not allowed to use any predicate symbols other than the above to translate the following sentences into predicate logic:
   (a) Everybody has a mother.
   (b) Everybody has a father and a mother.
   (c) Whoever has a mother has a father.
   (d) Ed is a grandfather.
   (e) All fathers are parents.
   (f) All husbands are spouses.
   (g) No uncle is an aunt.
   (h) All brothers are siblings.
   (i) Nobody's grandmother is anybody's father.
   (j) Ed and Patsy are husband and wife.
   (k) Carl is Monique's brother-in-law.
5. The following sentences are taken from the RFC3157 Internet Taskforce Document 'Securely Available Credentials – Requirements.' Specify each sentence in predicate logic, defining predicate symbols as appropriate:
   (a) An attacker can persuade a server that a successful login has occurred, even if it hasn't.
   (b) An attacker can overwrite someone else's credentials on the server.
   (c) All users enter passwords instead of names.
   (d) Credential transfer both to and from a device MUST be supported.
   (e) Credentials MUST NOT be forced by the protocol to be present in cleartext at any device other than the end user's.
   (f) The protocol MUST support a range of cryptographic algorithms, including syymetric and asymmetric algorithms, hash algorithms, and MAC algorithms.
   (g) Credentials MUST only be downloadable following user authentication or else only downloadable in a format that requires completion of user authentication for deciphering.
   (h) Different end user devices MAY be used to download, upload, or manage the same set of credentials.

Exercises 2.2

1. Let $\mathcal{F}$ be $\{d, f, g\}$, where d is a constant, f a function symbol with two arguments and g a function symbol with three arguments.
   (a) Which of the following strings are terms over $\mathcal{F}$? Draw the parse tree of those strings which are indeed terms:
      i. $g(d, d)$
   * ii. $f(x, g(y, z), d)$

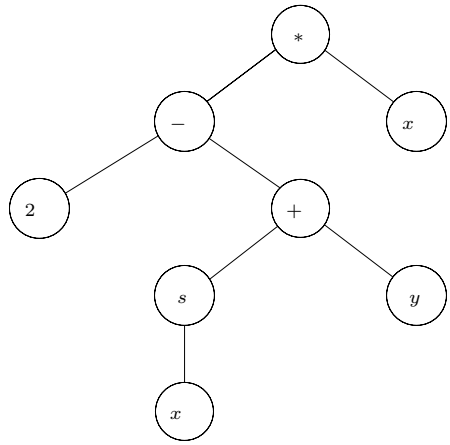

Figure 2.14. A parse tree representing an arithmetic term.

* iii. $g(x, f(y, z), d)$
iv. $g(x, h(y, z), d)$
v. $f(f(g(d, x), f(g(d, x), y, g(y, d)), g(d, d)), g(f(d, d, x), d), z)$

(b) The length of a term over $\mathcal{F}$ is the length of its string representation, where we count all commas and parentheses. For example, the length of $f(x, g(y, z), z)$ is 13. List all variable-free terms over $\mathcal{F}$ of length less than 10.

* (c) The height of a term over $\mathcal{F}$ is defined as 1 plus the length of the longest path in its parse tree, as in Definition 1.32. List all variable-free terms over $\mathcal{F}$ of height less than 4.

2. Draw the parse tree of the term $(2 - s(x)) + (y * x)$, considering that $-$, $+$, and $*$ are used in infix in this term. Compare your solution with the parse tree in Figure 2.14.
3. Which of the following strings are formulas in predicate logic? Specify a reason for failure for strings which aren't, draw parse trees of all strings which are.

* (a) Let m be a constant, f a function symbol with one argument and S and B two predicate symbols, each with two arguments:
i. $S(m, x)$
ii. $B(m, f(m))$
iii. $f(m)$
iv. $B(B(m, x), y)$
v. $S(B(m), z)$
vi. $(B(x, y) \to (\exists z\, S(z, y)))$
vii. $(S(x, y) \to S(y, f(f(x))))$
viii. $(B(x) \to B(B(x)))$.

(b) Let c and d be constants, f a function symbol with one argument, g a function symbol with two arguments and h a function symbol with three arguments. Further, P and Q are predicate symbols with three arguments:

i. $\forall x\, P(f(d), h(g(c,x),d,y))$
ii. $\forall x\, P(f(d), h(P(x,y),d,y))$
iii. $\forall x\, Q(g(h(x,f(d),x),g(x,x)),h(x,x,x),c)$
iv. $\exists z\, (Q(z,z,z) \to P(z))$
v. $\forall x \forall y\, (g(x,y) \to P(x,y,x))$
vi. $Q(c,d,c)$.

4. Let ϕ be $\exists x\, (P(y,z) \wedge (\forall y\, (\neg Q(y,x) \vee P(y,z))))$, where P and Q are predicate symbols with two arguments.
* (a) Draw the parse tree of ϕ.
* (b) Identify all bound and free variable leaves in ϕ.
(c) Is there a variable in ϕ which has free and bound occurrences?
* (d) Consider the terms w (w is a variable), $f(x)$ and $g(y,z)$, where f and g are function symbols with arity 1 and 2, respectively.
i. Compute $\phi[w/x]$, $\phi[w/y]$, $\phi[f(x)/y]$ and $\phi[g(y,z)/z]$.
ii. Which of w, $f(x)$ and $g(y,z)$ are free for x in ϕ?
iii. Which of w, $f(x)$ and $g(y,z)$ are free for y in ϕ?
(e) What is the scope of $\exists x$ in ϕ?
* (f) Suppose that we change ϕ to $\exists x\, (P(y,z) \wedge (\forall x\, (\neg Q(x,x) \vee P(x,z))))$. What is the scope of $\exists x$ now?

5. (a) Let P be a predicate symbol with arity 3. Draw the parse tree of $\psi \stackrel{\text{def}}{=} \neg(\forall x\, ((\exists y\, P(x,y,z)) \wedge (\forall z\, P(x,y,z))))$.
(b) Indicate the free and bound variables in that parse tree.
(c) List all variables which occur free and bound therein.
(d) Compute $\psi[t/x]$, $\psi[t/y]$ and $\psi[t/z]$, where $t \stackrel{\text{def}}{=} g(f(g(y,y)),y)$. Is t free for x in ψ; free for y in ψ; free for z in ψ?

6. Rename the variables for ϕ in Example 2.9 (page 106) such that the resulting formula ψ has the same meaning as ϕ, but $f(y,y)$ is free for x in ψ.

Exercises 2.3

1. Prove the validity of the following sequents using, among others, the rules =i and =e. Make sure that you indicate for each application of =e what the rule instances ϕ, t_1 and t_2 are.
(a) $(y = 0) \wedge (y = x) \vdash 0 = x$
(b) $t_1 = t_2 \vdash (t + t_2) = (t + t_1)$
(c) $(x = 0) \vee ((x + x) > 0) \vdash (y = (x + x)) \to ((y > 0) \vee (y = (0 + x)))$.

2. Recall that we use $=$ to express the equality of elements in our models. Consider the formula $\exists x \exists y\, (\neg(x = y) \wedge (\forall z\, ((z = x) \vee (z = y))))$. Can you say, in plain English, what this formula specifies?

3. Try to write down a sentence of predicate logic which intuitively holds in a model iff the model has (respectively)
* (a) exactly three distinct elements
(b) at most three distinct elements
* (c) only finitely many distinct elements.

What 'limitation' of predicate logic causes problems in finding such a sentence for the last item?

4. (a) Find a (propositional) proof for $\phi \to (q_1 \wedge q_2) \vdash (\phi \to q_1) \wedge (\phi \to q_2)$.
   (b) Find a (predicate) proof for $\phi \to \forall x\, Q(x) \vdash \forall x\, (\phi \to Q(x))$, provided that x is not free in ϕ.
   (Hint: whenever you used $\wedge$ rules in the (propositional) proof of the previous item, use $\forall$ rules in the (predicate) proof.)
   (c) Find a proof for $\forall x\, (P(x) \to Q(x)) \vdash \forall x\, P(x) \to \forall x\, Q(x)$.
   (Hint: try $(p_1 \to q_1) \wedge (p_2 \to q_2) \vdash p_1 \wedge p_2 \to q_1 \wedge q_2$ first.)
5. Find a propositional logic sequent that corresponds to $\exists x\, \neg\phi \vdash \neg\forall x\, \phi$. Prove it.
6. Provide proofs for the following sequents:
   (a) $\forall x\, P(x) \vdash \forall y\, P(y)$; using $\forall x\, P(x)$ as a premise, your proof needs to end with an application of $\forall$i which requires the formula $P(y_0)$.
   (b) $\forall x\, (P(x) \to Q(x)) \vdash (\forall x\, \neg Q(x)) \to (\forall x\, \neg P(x))$
   (c) $\forall x\, (P(x) \to \neg Q(x)) \vdash \neg(\exists x\, (P(x) \wedge Q(x)))$.
7. The sequents below look a bit tedious, but in proving their validity you make sure that you really understand how to nest the proof rules:
   * (a) $\forall x\, \forall y\, P(x, y) \vdash \forall u\, \forall v\, P(u, v)$
   (b) $\exists x\, \exists y\, F(x, y) \vdash \exists u\, \exists v\, F(u, v)$
   * (c) $\exists x\, \forall y\, P(x, y) \vdash \forall y\, \exists x\, P(x, y)$.
8. In this exercise, whenever you use a proof rule for quantifiers, you should mention how its side condition (if applicable) is satisfied.
   (a) Prove 2(b-h) of Theorem 2.13 from page 117.
   (b) Prove one direction of 1(b) of Theorem 2.13: $\neg\exists x\, \phi \vdash \forall x\, \neg\phi$.
   (c) Prove 3(a) of Theorem 2.13: $(\forall x\, \phi) \wedge (\forall x\, \psi) \dashv\vdash \forall x\, (\phi \wedge \psi)$; recall that you have to do two separate proofs.
   (d) Prove both directions of 4(a) of Theorem 2.13: $\forall x\, \forall y\, \phi \dashv\vdash \forall y\, \forall x\, \phi$.
9. Prove the validity of the following sequents in predicate logic, where F, G, P, and Q have arity 1, and S has arity 0 (a 'propositional atom'):
   * (a) $\exists x\, (S \to Q(x)) \vdash S \to \exists x\, Q(x)$
   (b) $S \to \exists x\, Q(x) \vdash \exists x\, (S \to Q(x))$
   (c) $\exists x\, P(x) \to S \vdash \forall x\, (P(x) \to S)$
   * (d) $\forall x\, P(x) \to S \vdash \exists x\, (P(x) \to S)$
   (e) $\forall x\, (P(x) \vee Q(x)) \vdash \forall x\, P(x) \vee \exists x\, Q(x)$
   (f) $\forall x\, \exists y\, (P(x) \vee Q(y)) \vdash \exists y\, \forall x\, (P(x) \vee Q(y))$
   (g) $\forall x\, (\neg P(x) \wedge Q(x)) \vdash \forall x\, (P(x) \to Q(x))$
   (h) $\forall x\, (P(x) \wedge Q(x)) \vdash \forall x\, (P(x) \to Q(x))$
   (i) $\exists x\, (\neg P(x) \wedge \neg Q(x)) \vdash \exists x\, (\neg(P(x) \wedge Q(x)))$
   (j) $\exists x\, (\neg P(x) \vee Q(x)) \vdash \exists x\, (\neg(P(x) \wedge \neg Q(x)))$
   * (k) $\forall x\, (P(x) \wedge Q(x)) \vdash \forall x\, P(x) \wedge \forall x\, Q(x)$.
   * (l) $\forall x\, P(x) \vee \forall x\, Q(x) \vdash \forall x\, (P(x) \vee Q(x))$.
   *(m) $\exists x\, (P(x) \wedge Q(x)) \vdash \exists x\, P(x) \wedge \exists x\, Q(x)$.
   * (n) $\exists x\, F(x) \vee \exists x\, G(x) \vdash \exists x\, (F(x) \vee G(x))$.
   (o) $\forall x\, \forall y\, (S(y) \to F(x)) \vdash \exists y S(y) \to \forall x\, F(x)$.

* (p) $\neg\forall x\,\neg P(x) \vdash \exists x\,P(x)$.
* (q) $\forall x\,\neg P(x) \vdash \neg\exists x\,P(x)$.
* (r) $\neg\exists x\,P(x) \vdash \forall x\,\neg P(x)$.

10. Just like natural deduction proofs for propositional logic, certain things that look easy can be hard to prove for predicate logic. Typically, these involve the $\neg\neg$e rule. The patterns are the same as in propositional logic:
    (a) Proving that $p \vee q \vdash \neg(\neg p \wedge \neg q)$ is valid is quite easy. Try it.
    (b) Show that $\exists x\,P(x) \vdash \neg\forall x\,\neg P(x)$ is valid.
    (c) Proving that $\neg(\neg p \wedge \neg q) \vdash p \vee q$ is valid is hard; you have to try to prove $\neg\neg(p \vee q)$ first and then use the $\neg\neg$e rule. Do it.
    (d) Re-express the sequent from the previous item such that p and q are unary predicates and both formulas are universally quantified. Prove its validity.
11. The proofs of the sequents below combine the proof rules for equality and quantifiers. We write $\phi \leftrightarrow \psi$ as an abbreviation for $(\phi \to \psi) \wedge (\psi \to \phi)$. Find proofs for
    * (a) $P(b) \vdash \forall x\,(x = b \to P(x))$
    (b) $P(b), \forall x\forall y\,(P(x) \wedge P(y) \to x = y) \vdash \forall x\,(P(x) \leftrightarrow x = b)$
    * (c) $\exists x\,\exists y\,(H(x,y) \vee H(y,x)), \neg\exists x\,H(x,x) \vdash \exists x\exists y\,\neg(x = y)$
    (d) $\forall x\,(P(x) \leftrightarrow x = b) \vdash P(b) \wedge \forall x\forall y\,(P(x) \wedge P(y) \to x = y)$.
* 12. Prove the validity of $S \to \forall x\,Q(x) \vdash \forall x\,(S \to Q(x))$, where S has arity 0 (a 'propositional atom').
13. By natural deduction, show the validity of
    * (a) $\forall x\,P(a,x,x),\ \forall x\,\forall y\,\forall z\,(P(x,y,z) \to P(f(x),y,f(z))) \vdash P(f(a),a,f(a))$
    * (b) $\forall x\,P(a,x,x),\ \forall x\,\forall y\,\forall z\,(P(x,y,z) \to P(f(x),y,f(z))) \vdash \exists z\,P(f(a),z,f(f(a)))$
    * (c) $\forall y\,Q(b,y),\ \forall x\,\forall y\,(Q(x,y) \to Q(s(x),s(y))) \vdash \exists z\,(Q(b,z) \wedge Q(z,s(s(b))))$
    (d) $\forall x\,\forall y\,\forall z\,(S(x,y) \wedge S(y,z) \to S(x,z)),\ \forall x\,\neg S(x,x) \vdash \forall x\,\forall y\,(S(x,y) \to \neg S(y,x))$
    (e) $\forall x\,(P(x) \vee Q(x)),\ \exists x\,\neg Q(x),\ \forall x\,(R(x) \to \neg P(x)) \vdash \exists x\,\neg R(x)$
    (f) $\forall x\,(P(x) \to (Q(x) \vee R(x))),\ \neg\exists x\,(P(x) \wedge R(x)) \vdash \forall x\,(P(x) \to Q(x))$
    (g) $\exists x\,\exists y\,(S(x,y) \vee S(y,x)) \vdash \exists x\,\exists y\,S(x,y)$
    (h) $\exists x\,(P(x) \wedge Q(x)),\ \forall y\,(P(x) \to R(x)) \vdash \exists x\,(R(x) \wedge Q(x))$.
14. Translate the following argument into a sequent in predicate logic using a suitable set of predicate symbols:

> If there are any tax payers, then all politicians are tax payers. If there are any philanthropists, then all tax payers are philanthropists. So, if there are any tax-paying philanthropists, then all politicians are philanthropists.

Now come up with a proof of that sequent in predicate logic.

15. Discuss in what sense the equivalences of Theorem 2.13 (page 117) form the basis of an algorithm which, given ϕ, pushes quantifiers to the top of the formula's parse tree. If the result is ψ, what can you say about commonalities and differences between ϕ and ψ?

Exercises 2.4

* 1. Consider the formula $\phi \stackrel{\text{def}}{=} \forall x \forall y\, Q(g(x,y), g(y,y), z)$, where Q and g have arity 3 and 2, respectively. Find two models $\mathcal{M}$ and $\mathcal{M}'$ with respective environments l and l' such that $\mathcal{M} \vDash_l \phi$ but $\mathcal{M}' \nvDash_{l'} \phi$.

2. Consider the sentence $\phi \stackrel{\text{def}}{=} \forall x \exists y \exists z\, (P(x,y) \wedge P(z,y) \wedge (P(x,z) \rightarrow P(z,x)))$. Which of the following models satisfies ϕ?
   (a) The model $\mathcal{M}$ consists of the set of natural numbers with $P^{\mathcal{M}} \stackrel{\text{def}}{=} \{(m,n) \mid m < n\}$.
   (b) The model $\mathcal{M}'$ consists of the set of natural numbers with $P^{\mathcal{M}'} \stackrel{\text{def}}{=} \{(m, 2 * m) \mid m \text{ natural number}\}$.
   (c) The model $\mathcal{M}''$ consists of the set of natural numbers with $P^{\mathcal{M}''} \stackrel{\text{def}}{=} \{(m,n) \mid m < n+1\}$.

3. Let P be a predicate with two arguments. Find a model which satisfies the sentence $\forall x \neg P(x,x)$; also find one which doesn't.

4. Consider the sentence $\forall x(\exists y P(x,y) \wedge (\exists z P(z,x) \rightarrow \forall y P(x,y)))$. Please simulate the evaluation of this sentence in a model and look-up table of your choice, focusing on how the initial look-up table l grows and shrinks like a stack when you evaluate its subformulas according to the definition of the satisfaction relation.

5. Let ϕ be the sentence $\forall x \forall y \exists z\, (R(x,y) \rightarrow R(y,z))$, where R is a predicate symbol of two arguments.
   * (a) Let $A \stackrel{\text{def}}{=} \{a,b,c,d\}$ and $R^{\mathcal{M}} \stackrel{\text{def}}{=} \{(b,c),(b,b),(b,a)\}$. Do we have $\mathcal{M} \vDash \phi$? Justify your answer, whatever it is.
   * (b) Let $A' \stackrel{\text{def}}{=} \{a,b,c\}$ and $R^{\mathcal{M}'} \stackrel{\text{def}}{=} \{(b,c),(a,b),(c,b)\}$. Do we have $\mathcal{M}' \vDash \phi$? Justify your answer, whatever it is.

* 6. Consider the three sentences

$$\phi_1 \stackrel{\text{def}}{=} \forall x\, P(x,x)$$
$$\phi_2 \stackrel{\text{def}}{=} \forall x \forall y\, (P(x,y) \rightarrow P(y,x))$$
$$\phi_3 \stackrel{\text{def}}{=} \forall x \forall y \forall z\, ((P(x,y) \wedge P(y,z) \rightarrow P(x,z)))$$

which express that the binary predicate P is reflexive, symmetric and transitive, respectively. Show that none of these sentences is semantically entailed by the other ones by choosing for each pair of sentences above a model which satisfies these two, but not the third sentence – essentially, you are asked to find three binary relations, each satisfying just two of these properties.

7. Show the semantic entailment $\forall x\, \neg\phi \vDash \neg\exists x\, \phi$; for that you have to take any model which satisfies $\forall x\, \neg\phi$ and you have to reason why this model must also satisfy $\neg\exists x\, \phi$. You should do this in a similar way to the examples in Section 2.4.2.

* 8. Show the semantic entailment $\forall x\, P(x) \vee \forall x\, Q(x) \vDash \forall x\, (P(x) \vee Q(x))$.

9. Let ϕ and ψ and η be sentences of predicate logic.
   (a) If ψ is semantically entailed by ϕ, is it necessarily the case that ψ is not semantically entailed by $\neg\phi$?
   * (b) If ψ is semantically entailed by $\phi \wedge \eta$, is it necessarily the case that ψ is semantically entailed by ϕ and semantically entailed by η?
   (c) If ψ is semantically entailed by ϕ or by η, is it necessarily the case that ψ is semantically entailed by $\phi \vee \eta$?
   (d) Explain why ψ is semantically entailed by ϕ iff $\phi \to \psi$ is valid.

10. Is $\forall x\, (P(x) \vee Q(x)) \vDash \forall x\, P(x) \vee \forall x\, Q(x)$ a semantic entailment? Justify your answer.

11. For each set of formulas below show that they are consistent:
   (a) $\forall x\, \neg S(x,x),\ \exists x\, P(x),\ \forall x\, \exists y\, S(x,y),\ \forall x\, (P(x) \to \exists y\, S(y,x))$
   * (b) $\forall x\, \neg S(x,x),\ \forall x\, \exists y\, S(x,y),$
   $\forall x\, \forall y\, \forall z\, ((S(x,y) \wedge S(y,z)) \to S(x,z))$
   (c) $(\forall x\, (P(x) \vee Q(x))) \to \exists y\, R(y),\ \forall x\, (R(x) \to Q(x)),\ \exists y\, (\neg Q(y) \wedge P(y))$
   * (d) $\exists x\, S(x,x),\ \forall x\, \forall y\, (S(x,y) \to (x = y))$.

12. For each of the formulas of predicate logic below, either find a model which does not satisfy it, or prove it is valid:
   (a) $(\forall x\, \forall y\, (S(x,y) \to S(y,x))) \to (\forall x\, \neg S(x,x))$
   * (b) $\exists y\, ((\forall x\, P(x)) \to P(y))$
   (c) $(\forall x\, (P(x) \to \exists y\, Q(y))) \to (\forall x\, \exists y\, (P(x) \to Q(y)))$
   (d) $(\forall x\, \exists y\, (P(x) \to Q(y))) \to (\forall x\, (P(x) \to \exists y\, Q(y)))$
   (e) $\forall x\, \forall y\, (S(x,y) \to (\exists z\, (S(x,z) \wedge S(z,y))))$
   (f) $(\forall x\, \forall y\, (S(x,y) \to (x = y))) \to (\forall z\, \neg S(z,z))$
   * (g) $(\forall x\, \exists y\, (S(x,y) \wedge ((S(x,y) \wedge S(y,x)) \to (x = y)))) \to$
   $(\neg\exists z\, \forall w\, (S(z,w)))$.
   (h) $\forall x\, \forall y\, ((P(x) \to P(y)) \wedge (P(y) \to P(x)))$
   (i) $(\forall x\, ((P(x) \to Q(x)) \wedge (Q(x) \to P(x)))) \to ((\forall x\, P(x)) \to (\forall x\, Q(x)))$
   (j) $((\forall x\, P(x)) \to (\forall x\, Q(x))) \to (\forall x\, ((P(x) \to Q(x)) \wedge (Q(x) \to P(x))))$
   (k) Difficult: $(\forall x\, \exists y\, (P(x) \to Q(y))) \to (\exists y\, \forall x\, (P(x) \to Q(y)))$.

Exercises 2.5

1. Assuming that our proof calculus for predicate logic is sound (see exercise 3 below), show that the validity of the following sequents cannot be proved by finding for each sequent a model such that all formulas to the left of $\vdash$ evaluate to `T` and the sole formula to the right of $\vdash$ evaluates to `F` (explain why this guarantees the non-existence of a proof):

(a) $\forall x\,(P(x) \vee Q(x)) \vdash \forall x\,P(x) \vee \forall x\,Q(x)$
* (b) $\forall x\,(P(x) \to R(x)),\ \forall x\,(Q(x) \to R(x)) \vdash \exists x\,(P(x) \wedge Q(x))$
(c) $(\forall x\,P(x)) \to L \vdash \forall x\,(P(x) \to L)$, where L has arity 0
* (d) $\forall x\,\exists y\,S(x,y) \vdash \exists y\,\forall x\,S(x,y)$
(e) $\exists x\,P(x),\ \exists y\,Q(y) \vdash \exists z\,(P(z) \wedge Q(z))$.
* (f) $\exists x\,(\neg P(x) \wedge Q(x)) \vdash \forall x\,(P(x) \to Q(x))$
* (g) $\exists x\,(\neg P(x) \vee \neg Q(x)) \vdash \forall x\,(P(x) \vee Q(x))$.

2. Assuming that $\vdash$ is sound and complete for $\vDash$ in first-order logic, explain in detail why the undecidability of $\vDash$ implies that satisfiability, validity, and provability are all undecidable for that logic.
3. To show the soundness of our natural deduction rules for predicate logic, it intuitively suffices to show that the conclusion of a proof rule is true provided that all its premises are true. What additional complication arises due to the presence of variables and quantifiers? Can you precisely formalise the necessary induction hypothesis for proving soundness?

Exercises 2.6

1. In Example 2.23, page 136, does $\mathcal{M} \vDash_l \exists P\,\phi$ hold if l satisfies
* (a) $l(u) = s_3$ and $l(v) = s_1$;
(b) $l(u) = s_1$ and $l(v) = s_3$?
Justify your answers.
2. Prove that $\mathcal{M} \vDash_l \exists P\,\forall x \forall y \forall z\,(C_1 \wedge C_2 \wedge C_3 \wedge C_4)$ holds iff state $l(v)$ is not reachable from state $l(u)$ in the model $\mathcal{M}$, where the C_i are the ones of (2.12) on page 139.
3. Does Theorem 2.26 from page 138 apply or remain valid if we allow ϕ to contain function symbols of any finite arity?
* 4. In the directed graph of Figure 2.5 from page 137, how many paths are there that witness the reachability of node s_3 from s_2?
5. Let P and R be predicate symbols of arity 2. Write formulas of existential second-order logic of the form $\exists P\,\psi$ that hold in all models of the form $\mathcal{M} = (A, R^{\mathcal{M}})$ iff
* (a) R contains a reflexive and symmetric relation;
(b) R contains an equivalence relation
(c) there is an R-path that visits each node of the graph exactly once – such a path is called Hamiltonian
(d) R can be extended to an equivalence relation: there is some equivalence relation T with $R^{\mathcal{M}} \subseteq T$
* (e) the relation 'there is an R-path of length 2' is transitive.
* 6. Show informally that (2.16) on page 141 gives rise to Russell's paradox: A has to be, and cannot be, an element of A.
7. The second item in the proof of Theorem 2.28 (page 140) relies on the fact that if a binary relation R is contained in a reflexive, transitive relation T of

the same type, then T also contains the reflexive, transitive closure of R. Prove this.

8. For the model of Example 2.23 and Figure 2.5 (page 137), determine which model checks hold and justify your answer:
   * (a) $\exists P\,(\forall x \forall y\, P(x,y) \rightarrow \neg P(y,x)) \wedge (\forall u \forall v\, R(u,v) \rightarrow P(v,u))$;
   (b) $\forall P\,(\exists x \exists y \exists z\, P(x,y) \wedge P(y,z) \wedge \neg P(x,z)) \rightarrow (\forall u \forall v\, R(u,v) \rightarrow P(u,v))$; and
   (c) $\forall P\,(\forall x\, \neg P(x,x)) \vee (\forall u \forall v\, R(u,v) \rightarrow P(u,v))$.
9. Express the following statements about a binary relation R in predicate logic, universal second-order logic, or existential second-order logic – if at all possible:
   (a) All symmetric, transitive relations either don't contain R or are equivalence relations.
   * (b) All nodes are on at least one R-cycle.
   (c) There is a smallest relation containing R which is symmetric.
   (d) There is a smallest relation containing R which is reflexive.
   * (e) The relation R is a maximal equivalence relation: R is an equivalence relation; and there is no relation contained in R that is an equivalence relation.

Exercises 2.7

1* (a) Explain why the model of Figure 2.11 (page 148) is a counterexample to `OfLovers` in the presence of the fact `NoSelfLove`.
   (b) Can you identify the set $\{a, b, c\}$ from Example 2.19 (page 128) with the model of Figure 2.11 such that these two models are structurally the same? Justify your answer.
   * (c) Explain informally why no model with less than three elements can satisfy (2.8) from page 128 and the fact `NoSelfLove`.
2. Use the following fragment of an Alloy module

```
module AboutGraphs

sig Element {}

sig Graph {
  nodes : set Element,
  edges : nodes -> nodes
}
```

   for these modelling tasks:
   (a) Recall Exercise 6 from page 163 and its three sentences, where $P(x,y)$ specifies that there is an edge from x to y. For each sentence, write a consistency check that attempts to generate a model of a graph in which that sentence is false, but the other two are true. Analyze it within Alloy. What it the smallest scope, if any, in which the analyzer finds a model for this?

* (b) (Recall that the expression `# S = n` specifies that set `S` has n elements.) Use Alloy to generate a graph with seven nodes such that each node can reach exactly five nodes on finite paths (not necessarily the same five nodes).
   (c) A cycle of length n is a set of n nodes and a path through each of them, beginning and ending with the same node. Generate a cycle of length 4.
3. An undirected graph has a set of nodes and a set of edges, except that every edge connects two nodes without any sense of direction.
   (a) Adjust the Alloy module from the previous item – e.g. by adding an appropriate `fact` – to 'simulate' undirected graphs.
   (b) Write some consistency and assertion checks and analyze them to boost the confidence you may have in your Alloy module of undirected graphs.
* 4. A colorable graph consists of a set of nodes, a binary symmetric relation (the edges) between nodes and a function that assigns to each node a color. This function is subject to the constraint that no nodes have the same color if they are related by an edge.
   (a) Write a signature `AboutColoredGraphs` for this structure and these constraints.
   (b) Write a `fun`-statement that generates a graph whose nodes are colored by two colors only. Such a graph is 2-colorable.
   (c) For eack $k = 3, 4$ write a `fun`-statement that generates a graph whose nodes are colored by k colors such that all k colors are being used. Such a graph is k-colorable.
   (d) Test these three functions in a module.
   (e) Try to write a `fun`-statement that generates a graph that is 3-colorable but definitely not 2-colorable. What does Alloy's model builder report? Consider the formula obtained from that `fun`-statement's body by existentially quantifying that body with all its parameters. Determine whether is belongs to predicate logic, existential or universal second-order logic.
* 5. A Kripke model is a state machine with a non-empty set of initial states `init`, a mapping `prop` from states to atomic properties (specifying which properties are true at which states), a state transition relation `next`, and a set of final states `final` (states that don't have a next state). With a module `KripkeModel`:
   (a) Write a signature `StateMachine` and some basic facts that reflect this structure and these constraints.
   (b) Write a `fun`-statement `Reaches` which takes a state machine as first parameter and a set of states as a second parameter such that the second parameter denotes the first parameter's set of states reachable from any initial state. Note: Given the type declaration `r : T -> T`, the expression `*r` has type `T -> T` as well and denotes the reflexive, transitive closure of `r`.
   (c) Write these `fun`-statements and check their consistency:
      i. `DeadlockFree(m: StateMachine)`, among the reachable states of `m` only the `final` ones can deadlock;

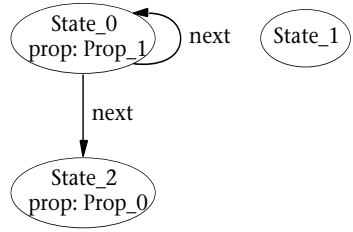

Figure 2.15. A snapshot of a non-deterministic state machine in which no non-final state deadlocks and where states that satisfy the same properties are identical.

ii. `Deterministic(m: StateMachine)`, at all reachable states of `m` the state transition relation is deterministic: each state has at most one outgoing transition;
iii. `Reachability(m: StateMachine, p: Prop)`, some state which has property `p` can be reached in `m`; and
iv. `Liveness(m: StateMachine, p: Prop)`, no matter which state `m` reaches, it can – from that state – reach a state in which `p` holds.

(d) i. Write an assertion `Implies` which says that whenever a state machine satisfies `Liveness` for a property then it also satisfies `Reachability` for that property.
ii. Analyze that assertion in a scope of your choice. What conclusions can you draw from the analysis' findings?

(e) Write an assertion `Converse` which states that `Reachability` of a property implies its `Liveness`. Analyze it in a scope of 3. What do you conclude, based on the analysis' result?

(f) Write a `fun`-statement that, when analyzed, generates a statemachine with two propositions and three states such that it satisfies the statement of the sentence in the caption of Figure 2.15.

* 6. Groups are the bread and butter of cryptography and group operations are applied in the silent background when you use PUTTY, Secure Socket Layers etc. A group is a tuple $(G, \star, 1)$, where $\star\colon G \times G \to G$ is a function and $1 \in G$ such that

G1 for every $x \in G$ there is some $y \in G$ such that $x \star y = y \star x = 1$ (any such y is called an inverse of x);
G2 for all $x, y, z \in G$, we have $x \star (y \star z) = (x \star y) \star z$; and
G3 for all $x \in G$, we have $x \star 1 = 1 \star x = x$.

(a) Specify a signature for groups that realizes this functionality and its constraints.

(b) Write a `fun`-statement `AGroup` that generates a group with three elements.

(c) Write an assertion `Inverse` saying that inverse elements are unique. Check it in the scope of 5. Report your findings. What would the small scope hypothesis suggest?

(d) i. Write an assertion `Commutative` saying that all groups are commutative. A group is commutative iff $x \star y = y \star x$ for all its elements x and y.
ii. Check the assertion `Commutative` in scope 5 and report your findings. What would the small scope hypothesis suggest?
iii. Re-check assertion `Commutative` in scope 6 and record how long the tool takes to find a solution. What lesson(s) do you learn from this?

(e) For the functions and assertions above, is it safe to restrict the scope for groups to 1? And how does one do this in Alloy?

7. In Alloy, one can extend a signature. For example, we may declare

```
sig Program extends PDS {
  m : components   -- initial main of PDS
}
```

This declares instances of `Program` to be of type `PDS`, but to also possess a designated component named `m`. Observe how the occurrence of `components` in `m : components` refers to the set of components of a program, viewed as a PDS[5]. In this exercise, you are asked to modify the Alloy module of Figure 2.13 on page 154.

(a) Include a signature `Program` as above. Add a fact stating that all programs' designated component has a `main` method; and for all programs, their set of `components` is the reflexive, transitive closure of their relation `requires` applied to the designated component `m`. Alloy uses `*r` to denote the reflexive, transitive closure of relation `r`.

(b) Write a guided simulation that, if consistent, produces a model with three PDSs, exactly one of them being a program. The program has four components – including the designated `m` – all of which schedule services from the remaining three components. Use Alloy's analyzer to detemine whether your simulation is consistent and compliant with the specification given in this item.

(c) Let's say that a component of a program is garbage for that program if no service reachable from the `main` service of `m` via `requires` schedules that component. Explain whether, and if so how, the constraints of `AddComponent` and `RemoveComponent` already enforce the presence of 'garbage collection' if the instances of `P` and `P'` are constrained to be programs.

8. Recall our discussion of existential and universal second-order logic from Section 2.6. Then study the structure of the `fun`-statements and assertions in Figure 2.13 on page 154. As you may know, Alloy analyzes such statements by deriving from them a formula for which it tries to find a model within the specified scope: the negation of the body of an assertion; or the body of a `fun`-statement, existentially quantified with all its parameters. For each of these derived formulas,

[5] In most object-oriented languages, e.g. Java, `extends` creates a new type. In Alloy 2.0 and 2.1, it creates a subset of a type and not a new type as such, where the subset has additional structure and may need to satisfy additional constraints.

determine whether they can be expressed in first-order logic, existential second-order logic or universal second-order logic.

9. Recalling the comment on page 142 that Alloy combines model checking $\mathcal{M} \models \phi$ and validity checking $\Gamma \models \phi$, can you discuss to what extent this is so?

2.9 Bibliographic notes

Many design decisions have been taken in the development of predicate logic in the form known today. The Greeks and the medievals had systems in which many of the examples and exercises in this book could be represented, but nothing that we would recognise as predicate logic emerged until the work of Gottlob Frege in 1879, printed in [Fre03]. An account of the contributions of the many other people involved in the development of logic can be found in the first few pages of W. Hodges' chapter in [Hod83].

There are many books covering classical logic and its use in computer science; we give a few incomplete pointers to the literature. The books [SA91], [vD89] and [Gal87] cover more theoretical applications than those in this book, including type theory, logic programming, algebraic specification and term-rewriting systems. An approach focusing on automatic theorem proving is taken by [Fit96]. Books which study the mathematical aspects of predicate logic in greater detail, such as completeness of the proof systems and incompleteness of first-order arithmetic, include [Ham78] and [Hod83].

Most of these books present other proof systems besides natural deduction such as axiomatic systems and tableau systems. Although natural deduction has the advantages of elegance and simplicity over axiomatic methods, there are few expositions of it in logic books aimed at a computer science audience. One exception to this is the book [BEKV94], which is the first one to present the rules for quantifiers in the form we used here. A natural deduction theorem prover called Jape has been developed, in which one can vary the set of available rules and specify new ones[6].

A standard reference for computability theory is [BJ80]. A proof for the undecidability of the Post correspondence problem can be found in the text book [Tay98]. The second instance of a Post correspondence problem is taken from [Sch92]. A text on the fundamentals of databases systems is [EN94]. The discussion of Section 2.6 is largely based on the text [Pap94] which we highly recommend if you mean to find out more about the intimate connections between logic and computational complexity.

[6] `www.comlab.ox.ac.uk/oucl/users/bernard.sufrin/jape.html`

The source code of all complete Alloy modules from this chapter (working under Alloy 2.0 and 2.1) as well as source code compliant with Alloy 3.0 are available under 'ancillary material' at the book's website. The PDS model grew out of a coursework set in the Fall 2002 for *C475 Software Engineering Environments*, co-taught by Susan Eisenbach and the first author; a published model customized for the .NET global assembly cache will appeared in [EJC03]. The modelling language Alloy and its constraint analyzer [JSS01] have been developed by D. Jackson and his Software Design Group at the Laboratory for Computer Science at the Massachusetts Institute of Technology. The tool has a dedicated repository website at `alloy.mit.edu`.

More information on typed higher-order logics and their use in the modelling and verifying of programming frameworks can be found on F. Pfenning's course homepage[7] on Computation and Deduction.

[7] `www-2.cs.cmu.edu/~fp/courses/comp-ded/`