1
Propositional logic

The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally. Reasoning about situations means constructing arguments about them; we want to do this formally, so that the arguments are valid and can be defended rigorously, or executed on a machine.

Consider the following argument:

Example 1.1 If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. *Therefore*, there were taxis at the station.

Intuitively, the argument is valid, since if we put the *first* sentence and the *third* sentence together, they tell us that if there are no taxis, then John will be late. The second sentence tells us that he was not late, so it must be the case that there were taxis.

Much of this book will be concerned with arguments that have this structure, namely, that consist of a number of sentences followed by the word 'therefore' and then another sentence. The argument is valid if the sentence after the 'therefore' logically follows from the sentences before it. Exactly what we mean by 'follows from' is the subject of this chapter and the next one.

Consider another example:

Example 1.2 If it is raining and Jane does not have her umbrella with her, then she will get wet. Jane is not wet. It is raining. *Therefore*, Jane has her umbrella with her.

This is also a valid argument. Closer examination reveals that it actually has the same structure as the argument of the previous example! All we have

done is substituted some sentence fragments for others:

Example 1.1	Example 1.2
the train is late	it is raining
there are taxis at the station	Jane has her umbrella with her
John is late for his meeting	Jane gets wet.

The argument in each example could be stated without talking about trains and rain, as follows:

If p and not q, then r. Not r. p. *Therefore*, q.

In developing logics, we are not concerned with what the sentences really mean, but only in their logical structure. Of course, when we *apply* such reasoning, as done above, such meaning will be of great interest.

1.1 Declarative sentences

In order to make arguments rigorous, we need to develop a language in which we can express sentences in such a way that brings out their logical structure. The language we begin with is the language of propositional logic. It is based on *propositions*, or *declarative sentences* which one can, in principle, argue as being true or false. Examples of declarative sentences are:

(1) The sum of the numbers 3 and 5 equals 8.
(2) Jane reacted violently to Jack's accusations.
(3) Every even natural number >2 is the sum of two prime numbers.
(4) All Martians like pepperoni on their pizza.
(5) Albert Camus était un écrivain français.
(6) Die Würde des Menschen ist unantastbar.

These sentences are all declarative, because they are in principle capable of being declared 'true', or 'false'. Sentence (1) can be tested by appealing to basic facts about arithmetic (and by tacitly assuming an Arabic, decimal representation of natural numbers). Sentence (2) is a bit more problematic. In order to give it a truth value, we need to know who Jane and Jack are and perhaps to have a reliable account from someone who witnessed the situation described. In principle, e.g., if we had been at the scene, we feel that we would have been able to detect Jane's *violent* reaction, provided that it indeed occurred in that way. Sentence (3), known as Goldbach's conjecture, seems straightforward on the face of it. Clearly, a fact about *all* even numbers >2 is either true or false. But to this day nobody knows whether sentence (3) expresses a truth or not. It is even not clear whether this could be shown by some finite means, even if it were true. However, in

this text we will be content with sentences as soon as they can, in principle, attain some truth value regardless of whether this truth value reflects the actual state of affairs suggested by the sentence in question. Sentence (4) seems a bit silly, although we could say that *if* Martians exist and eat pizza, then all of them will either like pepperoni on it or not. (We have to introduce predicate logic in Chapter 2 to see that this sentence is also declarative if *no* Martians exist; it is then true.) Again, for the purposes of this text sentence (4) will do. Et alors, qu'est-ce qu'on pense des phrases (5) et (6)? Sentences (5) and (6) are fine if you happen to read French and German a bit. Thus, declarative statements can be made in any natural, or artificial, language.

The kind of sentences we *won't* consider here are non-declarative ones, like

- Could you please pass me the salt?
- Ready, steady, go!
- May fortune come your way.

Primarily, we are interested in precise declarative sentences, or *statements* about the behaviour of computer systems, or programs. Not only do we want to specify such statements but we also want to *check* whether a given program, or system, fulfils a specification at hand. Thus, we need to develop a calculus of reasoning which allows us to draw conclusions from given assumptions, like initialised variables, which are reliable in the sense that they preserve truth: if all our assumptions are true, then our conclusion ought to be true as well. A much more difficult question is whether, given any true property of a computer program, we can find an argument in our calculus that has this property as its conclusion. The declarative sentence (3) above might illuminate the problematic aspect of such questions in the context of number theory.

The logics we intend to design are *symbolic* in nature. We translate a certain sufficiently large subset of all English declarative sentences into strings of symbols. This gives us a compressed but still complete encoding of declarative sentences and allows us to concentrate on the mere mechanics of our argumentation. This is important since specifications of systems or software are sequences of such declarative sentences. It further opens up the possibility of automatic manipulation of such specifications, a job that computers just love to do[1]. Our strategy is to consider certain declarative sentences as

[1] There is a certain, slightly bitter, circularity in such endeavours: in proving that a certain computer program `P` satisfies a given property, we might let some other computer program `Q` try to find a proof that `P` satisfies the property; but who guarantees us that `Q` satisfies the property of producing only correct proofs? We seem to run into an infinite regress.

being *atomic*, or *indecomposable*, like the sentence

'The number 5 is even.'

We assign certain distinct symbols $p, q, r, \ldots$, or sometimes $p_1, p_2, p_3, \ldots$ to each of these atomic sentences and we can then code up more complex sentences in a *compositional* way. For example, given the atomic sentences

p: 'I won the lottery last week.'
q: 'I purchased a lottery ticket.'
r: 'I won last week's sweepstakes.'

we can form more complex sentences according to the rules below:

$\neg$: The *negation* of p is denoted by $\neg p$ and expresses 'I did **not** win the lottery last week,' or equivalently 'It is **not** true that I won the lottery last week.'

$\vee$: Given p and r we may wish to state that *at least one of them* is true: 'I won the lottery last week, **or** I won last week's sweepstakes;' we denote this declarative sentence by $p \vee r$ and call it the *disjunction* of p and r[2].

$\wedge$: Dually, the formula $p \wedge r$ denotes the rather fortunate *conjunction* of p and r: 'Last week I won the lottery **and** the sweepstakes.'

$\rightarrow$: Last, but definitely not least, the sentence '**If** I won the lottery last week, **then** I purchased a lottery ticket.' expresses an *implication* between p and q, suggesting that q is a logical consequence of p. We write $p \rightarrow q$ for that[3]. We call p the *assumption* of $p \rightarrow q$ and q its *conclusion*.

Of course, we are entitled to use these rules of constructing propositions repeatedly. For example, we are now in a position to form the proposition

$$p \wedge q \rightarrow \neg r \vee q$$

which means that '**if** p **and** q **then not** r **or** q'. You might have noticed a potential ambiguity in this reading. One could have argued that this sentence has the structure 'p is the case **and if** q **then** ...' A computer would require the insertion of brackets, as in

$$(p \wedge q) \rightarrow ((\neg r) \vee q)$$

[2] Its meaning should not be confused with the often implicit meaning of **or** in natural language discourse as **either** ... **or**. In this text **or** always means *at least one of them* and should not be confounded with *exclusive or* which states that *exactly one* of the two statements holds.

[3] The natural language meaning of '**if** ... **then** ...' often implicitly assumes a *causal role* of the assumption somehow enabling its conclusion. The logical meaning of implication is a bit different, though, in the sense that it states the *preservation of truth* which might happen without any causal relationship. For example, 'If all birds can fly, then Bob Dole was never president of the United States of America.' is a true statement, but there is no known causal connection between the flying skills of penguins and effective campaigning.

to disambiguate this assertion. However, we humans get annoyed by a proliferation of such brackets which is why we adopt certain conventions about the *binding priorities* of these symbols.

Convention 1.3 $\neg$ binds more tightly than $\vee$ and $\wedge$, and the latter two bind more tightly than $\rightarrow$. Implication $\rightarrow$ is *right-associative*: expressions of the form $p \rightarrow q \rightarrow r$ denote $p \rightarrow (q \rightarrow r)$.

1.2 Natural deduction

How do we go about constructing a calculus for reasoning about propositions, so that we can establish the validity of Examples 1.1 and 1.2? Clearly, we would like to have a set of rules each of which allows us to draw a conclusion given a certain arrangement of premises.

In natural deduction, we have such a collection of *proof rules*. They allow us to *infer* formulas from other formulas. By applying these rules in succession, we may infer a conclusion from a set of premises.

Let's see how this works. Suppose we have a set of formulas[4] ϕ_1, ϕ_2, ϕ_3, $\ldots$, ϕ_n, which we will call *premises*, and another formula, ψ, which we will call a *conclusion*. By applying proof rules to the premises, we hope to get some more formulas, and by applying more proof rules to those, to eventually obtain the conclusion. This intention we denote by

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi.$$

This expression is called a *sequent*; it is *valid* if a proof for it can be found. The sequent for Examples 1.1 and 1.2 is $p \wedge \neg q \rightarrow r, \neg r, p \vdash q$. Constructing such a proof is a creative exercise, a bit like programming. It is not necessarily obvious which rules to apply, and in what order, to obtain the desired conclusion. Additionally, our proof rules should be carefully chosen; otherwise, we might be able to 'prove' invalid patterns of argumentation. For

[4] It is traditional in logic to use Greek letters. Lower-case letters are used to stand for formulas and upper-case letters are used for sets of formulas. Here are some of the more commonly used Greek letters, together with their pronunciation:

Lower-case		Upper-case	
ϕ	phi	Φ	Phi
ψ	psi	Ψ	Psi
χ	chi	Γ	Gamma
η	eta	Δ	Delta
α	alpha		
β	beta		
γ	gamma		

example, we expect that we won't be able to show the sequent $p, q \vdash p \wedge \neg q$. For example, if p stands for 'Gold is a metal.' and q for 'Silver is a metal,' then knowing these two facts should not allow us to infer that 'Gold is a metal whereas silver isn't.'

Let's now look at our proof rules. We present about fifteen of them in total; we will go through them in turn and then summarise at the end of this section.

1.2.1 Rules for natural deduction

The rules for conjunction Our first rule is called the rule for conjunction ($\wedge$): and-introduction. It allows us to conclude $\phi \wedge \psi$, given that we have already concluded ϕ and ψ separately. We write this rule as

$$\frac{\phi \qquad \psi}{\phi \wedge \psi} \wedge\text{i}.$$

Above the line are the two premises of the rule. Below the line goes the conclusion. (It might not yet be the final conclusion of our argument; we might have to apply more rules to get there.) To the right of the line, we write the name of the rule; $\wedge$i is read 'and-introduction'. Notice that we have introduced a $\wedge$ (in the conclusion) where there was none before (in the premises).

For each of the connectives, there is one or more rules to introduce it and one or more rules to eliminate it. The rules for and-elimination are these two:

$$\frac{\phi \wedge \psi}{\phi} \wedge\text{e}_1 \qquad \frac{\phi \wedge \psi}{\psi} \wedge\text{e}_2. \tag{1.1}$$

The rule $\wedge\text{e}_1$ says: if you have a proof of $\phi \wedge \psi$, then by applying this rule you can get a proof of ϕ. The rule $\wedge\text{e}_2$ says the same thing, but allows you to conclude ψ instead. Observe the dependences of these rules: in the first rule of (1.1), the conclusion ϕ has to match the first conjunct of the premise, whereas the exact nature of the second conjunct ψ is irrelevant. In the second rule it is just the other way around: the conclusion ψ has to match the second conjunct ψ and ϕ can be any formula. It is important to engage in this kind of *pattern matching* before the application of proof rules.

Example 1.4 Let's use these rules to prove that $p \wedge q, r \vdash q \wedge r$ is valid. We start by writing down the premises; then we leave a gap and write the

conclusion:

$$p \wedge q$$
$$r$$

$$q \wedge r$$

The task of constructing the proof is to fill the gap between the premises and the conclusion by applying a suitable sequence of proof rules. In this case, we apply $\wedge e_2$ to the first premise, giving us q. Then we apply $\wedge i$ to this q and to the second premise, r, giving us $q \wedge r$. That's it! We also usually number all the lines, and write in the justification for each line, producing this:

1	$p \wedge q$	premise
2	r	premise
3	q	$\wedge e_2$ 1
4	$q \wedge r$	$\wedge i$ 3, 2

Demonstrate to yourself that you've understood this by trying to show on your own that $(p \wedge q) \wedge r,\ s \wedge t \vdash q \wedge s$ is valid. Notice that the ϕ and ψ can be instantiated not just to atomic sentences, like p and q in the example we just gave, but also to compound sentences. Thus, from $(p \wedge q) \wedge r$ we can deduce $p \wedge q$ by applying $\wedge e_1$, instantiating ϕ to $p \wedge q$ and ψ to r.

If we applied these proof rules literally, then the proof above would actually be a tree with root $q \wedge r$ and leaves $p \wedge q$ and r, like this:

$$\dfrac{\dfrac{p \wedge q}{q}\,\wedge e_2 \qquad r}{q \wedge r}\,\wedge i$$

However, we flattened this tree into a linear presentation which necessitates the use of pointers as seen in lines 3 and 4 above. These pointers allow us to recreate the actual proof tree. Throughout this text, we will use the flattened version of presenting proofs. That way you have to concentrate only on finding a proof, not on how to fit a growing tree onto a sheet of paper.

If a sequent is valid, there may be many different ways of proving it. So if you compare your solution to these exercises with those of others, they need not coincide. The important thing to realise, though, is that any putative proof can be *checked* for correctness by checking each individual line, starting at the top, for the valid application of its proof rule.

The rules of double negation Intuitively, there is no difference between a formula ϕ and its *double negation* $\neg\neg\phi$, which expresses no more and nothing less than ϕ itself. The sentence

'It is **not** true that it does **not** rain.'

is just a more contrived way of saying

'It rains.'

Conversely, knowing 'It rains,' we are free to state this fact in this more complicated manner if we wish. Thus, we obtain rules of elimination and introduction for double negation:

$$\frac{\neg\neg\phi}{\phi}\,\neg\neg\text{e} \qquad \frac{\phi}{\neg\neg\phi}\,\neg\neg\text{i}.$$

(There are rules for single negation on its own, too, which we will see later.)

Example 1.5 The proof of the sequent $p, \neg\neg(q \wedge r) \vdash \neg\neg p \wedge r$ below uses most of the proof rules discussed so far:

1	p	premise
2	$\neg\neg(q \wedge r)$	premise
3	$\neg\neg p$	$\neg\neg$i 1
4	$q \wedge r$	$\neg\neg$e 2
5	r	$\wedge\text{e}_2$ 4
6	$\neg\neg p \wedge r$	$\wedge$i 3, 5

Example 1.6 We now prove the sequent $(p \wedge q) \wedge r,\ s \wedge t \vdash q \wedge s$ which you were invited to prove by yourself in the last section. Please compare the proof below with your solution:

1	$(p \wedge q) \wedge r$	premise
2	$s \wedge t$	premise
3	$p \wedge q$	$\wedge\text{e}_1$ 1
4	q	$\wedge\text{e}_2$ 3
5	s	$\wedge\text{e}_1$ 2
6	$q \wedge s$	$\wedge$i 4, 5

The rule for eliminating implication There is one rule to introduce $\rightarrow$ and one to eliminate it. The latter is one of the best known rules of propositional logic and is often referred to by its Latin name *modus ponens*. We will usually call it by its modern name, implies-elimination (sometimes also referred to as arrow-elimination). This rule states that, given ϕ and knowing that ϕ implies ψ, we may rightfully conclude ψ. In our calculus, we write this as

$$\frac{\phi \quad \phi \rightarrow \psi}{\psi} \rightarrow\text{e}.$$

Let us justify this rule by spelling out instances of some declarative sentences p and q. Suppose that

$$\begin{aligned} p &: \text{It rained.} \\ p \rightarrow q &: \text{If it rained, then the street is wet.} \end{aligned}$$

so q is just 'The street is wet.' Now, *if* we know that it rained and *if* we know that the street is wet in the case that it rained, then we may combine these two pieces of information to conclude that the street is indeed wet. Thus, the justification of the $\rightarrow$e rule is a mere application of common sense. Another example from programming is:

$$\begin{aligned} p &: \text{The value of the program's input is an integer.} \\ p \rightarrow q &: \text{If the program's input is an integer, then the program outputs a boolean.} \end{aligned}$$

Again, we may put all this together to conclude that our program outputs a boolean value if supplied with an integer input. However, it is important to realise that the presence of p is absolutely essential for the inference to happen. For example, our program might well satisfy $p \rightarrow q$, but if it doesn't satisfy p – e.g. if its input is a surname – then we will not be able to derive q.

As we saw before, the formal parameters ϕ and the ψ for $\rightarrow$e can be instantiated to any sentence, including compound ones:

1	$\neg p \wedge q$	premise
2	$\neg p \wedge q \rightarrow r \vee \neg p$	premise
3	$r \vee \neg p$	$\rightarrow$e 2, 1

Of course, we may use any of these rules as often as we wish. For example, given p, $p \to q$ and $p \to (q \to r)$, we may infer r:

1	$p \to (q \to r)$	premise
2	$p \to q$	premise
3	p	premise
4	$q \to r$	$\to$e 1, 3
5	q	$\to$e 2, 3
6	r	$\to$e 4, 5

Before turning to implies-introduction, let's look at a hybrid rule which has the Latin name *modus tollens*. It is like the $\to$e rule in that it eliminates an implication. Suppose that $p \to q$ *and* $\neg q$ are the case. Then, *if* p holds we can use $\to$e to conclude that q holds. Thus, we then have that q *and* $\neg q$ hold, which is impossible. Therefore, we may infer that p must be false. But this can only mean that $\neg p$ is true. We summarise this reasoning into the rule *modus tollens*, or MT for short:[5]

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\ \text{MT}.$$

Again, let us see an example of this rule in the natural language setting:

> *'If Abraham Lincoln was Ethiopian, then he was African. Abraham Lincoln was not African; therefore he was not Ethiopian.'*

Example 1.7 In the following proof of

$$p \to (q \to r),\ p,\ \neg r \vdash \neg q$$

we use several of the rules introduced so far:

1	$p \to (q \to r)$	premise
2	p	premise
3	$\neg r$	premise
4	$q \to r$	$\to$e 1, 2
5	$\neg q$	MT 4, 3

[5] We will be able to *derive* this rule from other ones later on, but we introduce it here because it allows us already to do some pretty slick proofs. You may think of this rule as one on a higher level insofar as it does not mention the lower-level rules upon which it depends.

Examples 1.8 Here are two example proofs which combine the rule MT with either ¬¬e or ¬¬i:

1	$\neg p \to q$	premise
2	$\neg q$	premise
3	$\neg\neg p$	MT 1, 2
4	p	¬¬e 3

proves that the sequent $\neg p \to q,\ \neg q \vdash p$ is valid; and

1	$p \to \neg q$	premise
2	q	premise
3	$\neg\neg q$	¬¬i 2
4	$\neg p$	MT 1, 3

shows the validity of the sequent $p \to \neg q,\ q \vdash \neg p$.

Note that the order of applying double negation rules and MT is different in these examples; this order is driven by the structure of the particular sequent whose validity one is trying to show.

The rule implies introduction The rule MT made it possible for us to show that $p \to q,\ \neg q \vdash \neg p$ is valid. But the validity of the sequent $p \to q \vdash \neg q \to \neg p$ seems just as plausible. That sequent is, in a certain sense, saying the same thing. Yet, so far we have no rule which *builds* implications that do not already occur as premises in our proofs. The mechanics of such a rule are more involved than what we have seen so far. So let us proceed with care. Let us suppose that $p \to q$ is the case. If we *temporarily* assume that $\neg q$ holds, we can use MT to infer $\neg p$. Thus, assuming $p \to q$ we can show that $\neg q$ **implies** $\neg p$; but the latter we express *symbolically* as $\neg q \to \neg p$. To summarise, we have found an argumentation for $p \to q \vdash \neg q \to \neg p$:

1	$p \to q$	premise
2	$\neg q$	assumption
3	$\neg p$	MT 1, 2
4	$\neg q \to \neg p$	→i 2–3

The box in this proof serves to demarcate the scope of the temporary assumption $\neg q$. What we are saying is: let's make the assumption of $\neg q$. To

do this, we open a box and put $\neg q$ at the top. Then we continue applying other rules as normal, for example to obtain $\neg p$. But this still depends on the assumption of $\neg q$, so it goes inside the box. Finally, we are ready to apply $\rightarrow$i. It allows us to conclude $\neg q \rightarrow \neg p$, but that conclusion no longer *depends* on the assumption $\neg q$. Compare this with saying that 'If you are French, then you are European.' The truth of this sentence does not depend on whether anybody is French or not. Therefore, we write the conclusion $\neg q \rightarrow \neg p$ outside the box.

This works also as one would expect if we think of $p \rightarrow q$ as a *type* of a procedure. For example, p could say that the procedure expects an integer value x as input and q might say that the procedure returns a boolean value y as output. The validity of $p \rightarrow q$ amounts now to an assume-guarantee assertion: if the input is an integer, then the output is a boolean. This assertion can be true about a procedure while that same procedure could compute strange things or crash in the case that the input is not an integer. Showing $p \rightarrow q$ using the rule $\rightarrow$i is now called *type checking*, an important topic in the construction of compilers for typed programming languages.

We thus formulate the rule $\rightarrow$i as follows:

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}}{\phi \rightarrow \psi}\ \rightarrow\text{i}.$$

It says: in order to prove $\phi \rightarrow \psi$, make a temporary assumption of ϕ and then prove ψ. In your proof of ψ, you can use ϕ and any of the other formulas such as premises and provisional conclusions that you have made so far. Proofs may nest boxes or open new boxes after old ones have been closed. There are rules about which formulas can be used at which points in the proof. Generally, we can only use a formula ϕ in a proof at a given point if that formula occurs *prior* to that point and if no box which encloses that occurrence of ϕ has been closed already.

The line immediately following a closed box has to match the pattern of the conclusion of the rule that uses the box. For implies-introduction, this means that we have to continue after the box with $\phi \rightarrow \psi$, where ϕ was the first and ψ the last formula of that box. We will encounter two more proof rules involving proof boxes and they will require similar pattern matching.

Example 1.9 Here is another example of a proof using →i:

1	$\neg q \to \neg p$	premise
2	p	assumption
3	$\neg\neg p$	¬¬i 2
4	$\neg\neg q$	MT 1, 3
5	$p \to \neg\neg q$	→i 2–4

which verifies the validity of the sequent $\neg q \to \neg p \vdash p \to \neg\neg q$. Notice that we could apply the rule MT to formulas occurring in or above the box: at line 4, no box has been closed that would enclose line 1 or 3.

At this point it is instructive to consider the one-line argument

1	p	premise

which demonstrates $p \vdash p$. The rule →i (with conclusion $\phi \to \psi$) does not prohibit the possibility that ϕ and ψ coincide. They could both be instantiated to p. Therefore we may extend the proof above to

1	p	assumption
2	$p \to p$	→i 1 – 1

We write $\vdash p \to p$ to express that the argumentation for $p \to p$ does not depend on any premises at all.

Definition 1.10 Logical formulas ϕ with valid sequent $\vdash \phi$ are *theorems*.

Example 1.11 Here is an example of a theorem whose proof utilises most of the rules introduced so far:

1	$q \to r$	assumption
2	$\neg q \to \neg p$	assumption
3	p	assumption
4	$\neg\neg p$	¬¬i 3
5	$\neg\neg q$	MT 2, 4
6	q	¬¬e 5
7	r	→e 1, 6
8	$p \to r$	→i 3–7
9	$(\neg q \to \neg p) \to (p \to r)$	→i 2–8
10	$(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$	→i 1–9

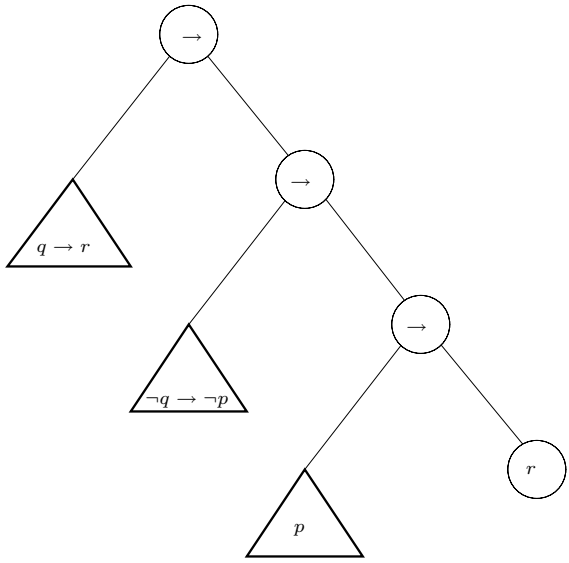


Figure 1.1. Part of the structure of the formula $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ to show how it determines the proof structure.

Therefore the sequent $\vdash (q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ is valid, showing that $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ is another theorem.

Remark 1.12 Indeed, this example indicates that we may transform any proof of $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ in such a way into a proof of the theorem

$$\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\cdots \to (\phi_n \to \psi) \ldots)))$$

by 'augmenting' the previous proof with n lines of the rule $\to$i applied to ϕ_n, ϕ_{n-1},..., ϕ_1 in that order.

The nested boxes in the proof of Example 1.11 reveal a pattern of using elimination rules first, to deconstruct assumptions we have made, and then introduction rules to construct our final conclusion. More difficult proofs may involve several such phases.

Let us dwell on this important topic for a while. How did we come up with the proof above? Parts of it are *determined* by the structure of the formulas we have, while other parts require us to be *creative*. Consider the logical structure of $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ schematically depicted in Figure 1.1. The formula is overall an implication since $\to$ is the root of the tree in Figure 1.1. But the only way to build an implication is by means

of the rule →i. Thus, we need to state the assumption of that implication as such (line 1) and have to show its conclusion (line 9). If we managed to do that, then we know how to end the proof in line 10. In fact, as we already remarked, this is the only way we could have ended it. So essentially lines 1, 9 and 10 are completely determined by the structure of the formula; further, we have reduced the problem to filling the gaps in between lines 1 and 9. But again, the formula in line 9 is an implication, so we have only one way of showing it: assuming its premise in line 2 and trying to show its conclusion in line 8; as before, line 9 is obtained by →i. The formula $p \to r$ in line 8 is yet another implication. Therefore, we have to assume p in line 3 and hope to show r in line 7, then →i produces the desired result in line 8.

The remaining question now is this: how can we show r, using the three assumptions in lines 1–3? This, and only this, is the creative part of this proof. We see the implication $q \to r$ in line 1 and know how to get r (using →e) if only we had q. So how could we get q? Well, lines 2 and 3 almost look like a pattern for the MT rule, which would give us $\neg\neg q$ in line 5; the latter is quickly changed to q in line 6 via ¬¬e. However, the pattern for MT does not match right away, since it requires $\neg\neg p$ instead of p. But this is easily accomplished via ¬¬i in line 4.

The moral of this discussion is that the logical structure of the formula to be shown tells you a lot about the structure of a possible proof and it is definitely worth your while to exploit that information in trying to prove sequents. Before ending this section on the rules for implication, let's look at some more examples (this time also involving the rules for conjunction).

Example 1.13 Using the rule ∧i, we can prove the validity of the sequent $p \wedge q \to r \vdash p \to (q \to r)$:

1	$p \wedge q \to r$	premise
2	p	assumption
3	q	assumption
4	$p \wedge q$	∧i 2, 3
5	r	→e 1, 4
6	$q \to r$	→i 3–5
7	$p \to (q \to r)$	→i 2–6

Example 1.14 Using the two elimination rules $\wedge e_1$ and $\wedge e_2$, we can show that the 'converse' of the sequent above is valid, too:

1	$p \to (q \to r)$	premise
2	$p \wedge q$	assumption
3	p	$\wedge e_1$ 2
4	q	$\wedge e_2$ 2
5	$q \to r$	$\to e$ 1, 3
6	r	$\to e$ 5, 4
7	$p \wedge q \to r$	$\to i$ 2–6

The validity of $p \to (q \to r) \vdash p \wedge q \to r$ and $p \wedge q \to r \vdash p \to (q \to r)$ means that these two formulas are equivalent in the sense that we can prove one from the other. We denote this by

$$p \wedge q \to r \dashv\vdash p \to (q \to r).$$

Since there can be only one formula to the right of $\vdash$, we observe that each instance of $\dashv\vdash$ can only relate *two* formulas to each other.

Example 1.15 Here is an example of a proof that uses introduction *and* elimination rules for conjunction; it shows the validity of the sequent $p \to q \vdash p \wedge r \to q \wedge r$:

1	$p \to q$	premise
2	$p \wedge r$	assumption
3	p	$\wedge e_1$ 2
4	r	$\wedge e_2$ 2
5	q	$\to e$ 1, 3
6	$q \wedge r$	$\wedge i$ 5, 4
7	$p \wedge r \to q \wedge r$	$\to i$ 2–6

The rules for disjunction The rules for disjunction are different in spirit from those for conjunction. The case for conjunction was concise and clear: proofs of $\phi \wedge \psi$ are essentially nothing but a concatenation of a proof of ϕ and a proof of ψ, plus an additional line invoking $\wedge i$. In the case of disjunctions, however, it turns out that the *introduction* of disjunctions is by far easier to grasp than their elimination. So we begin with the rules $\vee i_1$ and $\vee i_2$. From the premise ϕ we can infer that 'ϕ **or** ψ' holds, for we already know

that ϕ holds. Note that this inference is valid for any choice of ψ. By the same token, we may conclude 'ϕ **or** ψ' if we already have ψ. Similarly, that inference works for any choice of ϕ. Thus, we arrive at the proof rules

$$\frac{\phi}{\phi \vee \psi}\,\vee\mathrm{i}_1 \qquad \frac{\psi}{\phi \vee \psi}\,\vee\mathrm{i}_2.$$

So if p stands for 'Agassi won a gold medal in 1996.' and q denotes the sentence 'Agassi won Wimbledon in 1996.' then $p \vee q$ is the case because p is true, regardless of the fact that q is false. Naturally, the constructed disjunction depends upon the assumptions needed in establishing its respective disjunct p or q.

Now let's consider or-elimination. How can we use a formula of the form $\phi \vee \psi$ in a proof? Again, our guiding principle is to *disassemble* assumptions into their basic constituents so that the latter may be used in our argumentation such that they render our desired conclusion. Let us imagine that we want to show some proposition χ by assuming $\phi \vee \psi$. Since we don't know which of ϕ and ψ is true, we have to give *two* separate proofs which we need to combine into one argument:

1. First, we assume ϕ is true and have to come up with a proof of χ.
2. Next, we assume ψ is true and need to give a proof of χ as well.
3. Given these two proofs, we can infer χ from the truth of $\phi \vee \psi$, since our case analysis above is exhaustive.

Therefore, we write the rule $\vee$e as follows:

$$\frac{\phi \vee \psi \qquad \begin{array}{|c|}\hline \phi \\ \vdots \\ \chi \\ \hline\end{array} \qquad \begin{array}{|c|}\hline \psi \\ \vdots \\ \chi \\ \hline\end{array}}{\chi}\,\vee\mathrm{e}.$$

It is saying that: if $\phi \vee \psi$ is true and – no matter whether we assume ϕ or we assume ψ – we can get a proof of χ, then we are entitled to deduce χ anyway. Let's look at a proof that $p \vee q \vdash q \vee p$ is valid:

1		$p \vee q$	premise
2	┌	p	assumption
3	└	$q \vee p$	$\vee\mathrm{i}_2$ 2
4	┌	q	assumption
5	└	$q \vee p$	$\vee\mathrm{i}_1$ 4
6		$q \vee p$	$\vee$e 1, 2−3, 4−5

Here are some points you need to remember about applying the $\vee$e rule.

- For it to be a sound argument we have to make sure that the conclusions in each of the two cases (the χ in the rule) are actually the same formula.
- The work done by the rule $\vee$e is the combining of the arguments of the two cases into one.
- In each case you may not use the temporary assumption of the other case, unless it is something that has already been shown before those case boxes began.
- The invocation of rule $\vee$e in line 6 lists three things: the line in which the disjunction appears (1), and the location of the two boxes for the two cases (2–3 and 4–5).

If we use $\phi \vee \psi$ in an argument where it occurs only as an assumption or a premise, then we are missing a certain amount of information: we know ϕ, or ψ, but we don't know which one of the two it is. Thus, we have to make a solid case for each of the two possibilities ϕ or ψ; this resembles the behaviour of a `CASE` or `IF` statement found in most programming languages.

Example 1.16 Here is a more complex example illustrating these points. We prove that the sequent $q \to r \vdash p \vee q \to p \vee r$ is valid:

1	$q \to r$	premise
2	$p \vee q$	assumption
3	p	assumption
4	$p \vee r$	$\vee\text{i}_1$ 3
5	q	assumption
6	r	$\to$e 1, 5
7	$p \vee r$	$\vee\text{i}_2$ 6
8	$p \vee r$	$\vee$e 2, 3–4, 5–7
9	$p \vee q \to p \vee r$	$\to$i 2–8

Note that the propositions in lines 4, 7 and 8 coincide, so the application of $\vee$e is legitimate.

We give some more example proofs which use the rules $\vee$e, $\vee\text{i}_1$ and $\vee\text{i}_2$.

Example 1.17 Proving the validity of the sequent $(p \vee q) \vee r \vdash p \vee (q \vee r)$ is surprisingly long and seemingly complex. But this is to be expected, since

the elimination rules break $(p \vee q) \vee r$ up into its atomic constituents p, q and r, whereas the introduction rules then built up the formula $p \vee (q \vee r)$.

1	$(p \vee q) \vee r$	premise
2	$(p \vee q)$	assumption
3	p	assumption
4	$p \vee (q \vee r)$	$\vee$i$_1$ 3
5	q	assumption
6	$q \vee r$	$\vee$i$_1$ 5
7	$p \vee (q \vee r)$	$\vee$i$_2$ 6
8	$p \vee (q \vee r)$	$\vee$e $2, 3{-}4, 5{-}7$
9	r	assumption
10	$q \vee r$	$\vee$i$_2$ 9
11	$p \vee (q \vee r)$	$\vee$i$_2$ 10
12	$p \vee (q \vee r)$	$\vee$e $1, 2{-}8, 9{-}11$

Example 1.18 From boolean algebra, or circuit theory, you may know that disjunctions distribute over conjunctions. We are now able to prove this in natural deduction. The following proof:

1	$p \wedge (q \vee r)$	premise
2	p	$\wedge$e$_1$ 1
3	$q \vee r$	$\wedge$e$_2$ 1
4	q	assumption
5	$p \wedge q$	$\wedge$i $2, 4$
6	$(p \wedge q) \vee (p \wedge r)$	$\vee$i$_1$ 5
7	r	assumption
8	$p \wedge r$	$\wedge$i $2, 7$
9	$(p \wedge q) \vee (p \wedge r)$	$\vee$i$_2$ 8
10	$(p \wedge q) \vee (p \wedge r)$	$\vee$e $3, 4{-}6, 7{-}9$

verifies the validity of the sequent $p \wedge (q \vee r) \vdash (p \wedge q) \vee (p \wedge r)$ and you are encouraged to show the validity of the 'converse' $(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)$ yourself.

A final rule is required in order to allow us to conclude a box with a formula which has already appeared earlier in the proof. Consider the sequent $\vdash p \to (q \to p)$, whose validity may be proved as follows:

1	p	assumption
2	q	assumption
3	p	copy 1
4	$q \to p$	$\to$i 2–3
5	$p \to (q \to p)$	$\to$i 1–4

The rule 'copy' allows us to repeat something that we know already. We need to do this in this example, because the rule $\to$i requires that we end the inner box with p. The copy rule entitles us to copy formulas that appeared before, unless they depend on temporary assumptions whose box has already been closed. Though a little inelegant, this additional rule is a small price to pay for the freedom of being able to use premises, or any other 'visible' formulas, more than once.

The rules for negation We have seen the rules $\neg\neg$i and $\neg\neg$e, but we haven't seen any rules that introduce or eliminate single negations. These rules involve the notion of *contradiction*. This detour is to be expected since our reasoning is concerned about the inference, and therefore the preservation, of truth. Hence, there cannot be a direct way of inferring $\neg\phi$, given ϕ.

Definition 1.19 Contradictions are expressions of the form $\phi \wedge \neg\phi$ or $\neg\phi \wedge \phi$, where ϕ is any formula.

Examples of such contradictions are $r \wedge \neg r$, $(p \to q) \wedge \neg(p \to q)$ and $\neg(r \vee s \to q) \wedge (r \vee s \to q)$. Contradictions are a very important notion in logic. As far as truth is concerned, they are all equivalent; that means we should be able to prove the validity of

$$\neg(r \vee s \to q) \wedge (r \vee s \to q) \dashv\vdash (p \to q) \wedge \neg(p \to q) \tag{1.2}$$

since both sides are contradictions. We'll be able to prove this later, when we have introduced the rules for negation.

Indeed, it's not just that contradictions can be derived from contradictions; actually, *any* formula can be derived from a contradiction. This can be

confusing when you first encounter it; why should we endorse the argument $p \wedge \neg p \vdash q$, where

$$p\colon \text{The moon is made of green cheese.}$$
$$q\colon \text{I like pepperoni on my pizza.}$$

considering that our taste in pizza doesn't have anything to do with the constitution of the moon? On the face of it, such an endorsement may seem absurd. Nevertheless, natural deduction does have this feature that any formula can be derived from a contradiction and therefore it makes this argument valid. The reason it takes this stance is that $\vdash$ tells us all the things we may infer, provided that we can assume the formulas to the left of it. This process does not care whether such premises make any sense. This has at least the advantage that we can match $\vdash$ to checks based on semantic intuitions which we formalise later by using truth tables: if all the premises compute to 'true', then the conclusion must compute 'true' as well. In particular, this is not a constraint in the case that one of the premises is (always) false.

The fact that $\bot$ can prove anything is encoded in our calculus by the proof rule bottom-elimination:

$$\frac{\bot}{\phi}\ \bot\text{e}.$$

The fact that $\bot$ itself represents a contradiction is encoded by the proof rule not-elimination:

$$\frac{\phi \quad \neg\phi}{\bot}\ \neg\text{e}.$$

Example 1.20 We apply these rules to show that $\neg p \vee q \vdash p \to q$ is valid:

1	$\neg p \vee q$				
2	$\neg p$	premise	q	premise	
3	p	assumption	p	assumption	
4	$\bot$	$\neg$e 3, 2	q	copy 2	
5	q	$\bot$e 4	$p \to q$	$\to$i 3–4	
6	$p \to q$	$\to$i 3–5			
7	$p \to q$				$\vee$e 1, 2–6

Notice how, in this example, the proof boxes for $\vee$e are drawn side by side instead of on top of each other. It doesn't matter which way you do it.

What about introducing negations? Well, suppose we make an assumption which gets us into a contradictory state of affairs, i.e. gets us $\bot$. Then our assumption cannot be true; so it must be false. This intuition is the basis for the proof rule $\neg$i:

$$\frac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}\ \neg\text{i}.$$

Example 1.21 We put these rules in action, demonstrating that the sequent $p \to q,\ p \to \neg q \vdash \neg p$ is valid:

1	$p \to q$	premise
2	$p \to \neg q$	premise
3	p	assumption
4	q	$\to$e 1, 3
5	$\neg q$	$\to$e 2, 3
6	$\bot$	$\neg$e 4, 5
7	$\neg p$	$\neg$i 3–6

Lines 3–6 contain all the work of the $\neg$i rule. Here is a second example, showing the validity of a sequent, $p \to \neg p \vdash \neg p$, with a contradictory formula as sole premise:

1	$p \to \neg p$	premise
2	p	assumption
3	$\neg p$	$\to$e 1, 2
4	$\bot$	$\neg$e 2, 3
5	$\neg p$	$\neg$i 2–4

Example 1.22 We prove that the sequent $p \to (q \to r),\ p,\ \neg r \vdash \neg q$ is valid,

without using the MT rule:

1	$p \to (q \to r)$	premise
2	p	premise
3	$\neg r$	premise
4	q	assumption
5	$q \to r$	$\to$e 1, 2
6	r	$\to$e 5, 4
7	$\bot$	$\neg$e 6, 3
8	$\neg q$	$\neg$i 4–7

Example 1.23 Finally, we return to the argument of Examples 1.1 and 1.2, which can be coded up by the sequent $p \wedge \neg q \to r$, $\neg r$, $p \vdash q$ whose validity we now prove:

1	$p \wedge \neg q \to r$	premise
2	$\neg r$	premise
3	p	premise
4	$\neg q$	assumption
5	$p \wedge \neg q$	$\wedge$i 3, 4
6	r	$\to$e 1, 5
7	$\bot$	$\neg$e 6, 2
8	$\neg\neg q$	$\neg$i 4–7
9	q	$\neg\neg$e 8

1.2.2 Derived rules

When describing the proof rule *modus tollens* (MT), we mentioned that it is not a primitive rule of natural deduction, but can be derived from some of the other rules. Here is the derivation of

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\ \text{MT}$$

from $\rightarrow$e, $\neg$e and $\neg$i:

1	$\phi \rightarrow \psi$	premise
2	$\neg\psi$	premise
3	ϕ	assumption
4	ψ	$\rightarrow$e 1, 3
5	$\bot$	$\neg$e 4, 2
6	$\neg\phi$	$\neg$i 3–5

We could now go back through the proofs in this chapter and replace applications of MT by this combination of $\rightarrow$e, $\neg$e and $\neg$i. However, it is convenient to think of MT as a shorthand (or a macro).

The same holds for the rule

$$\frac{\phi}{\neg\neg\phi}\ \neg\neg\text{i}.$$

It can be derived from the rules $\neg$i and $\neg$e, as follows:

1	ϕ	premise
2	$\neg\phi$	assumption
3	$\bot$	$\neg$e 1, 2
4	$\neg\neg\phi$	$\neg$i 2–3

There are (unboundedly) many such derived rules which we could write down. However, there is no point in making our calculus fat and unwieldy; and some purists would say that we should stick to a minimum set of rules, all of which are independent of each other. We don't take such a purist view. Indeed, the two derived rules we now introduce are extremely useful. You will find that they crop up frequently when doing exercises in natural deduction, so it is worth giving them names as derived rules. In the case of the second one, its derivation from the primitive proof rules is not very obvious.

The first one has the Latin name *reductio ad absurdum*. It means 'reduction to absurdity' and we will simply call it *proof by contradiction* (PBC for short). The rule says: if from $\neg\phi$ we obtain a contradiction, then we are entitled to deduce ϕ:

$$\frac{\begin{array}{|c|}\hline \neg\phi \\ \vdots \\ \bot \\ \hline\end{array}}{\phi}\ \text{PBC}.$$

This rule looks rather similar to ¬i, except that the negation is in a different place. This is the clue to how to derive PBC from our basic proof rules. Suppose we have a proof of $\bot$ from $\neg\phi$. By $\to$i, we can transform this into a proof of $\neg\phi \to \bot$ and proceed as follows:

1	$\neg\phi \to \bot$	given
2	$\neg\phi$	assumption
3	$\bot$	$\to$e 1, 2
4	$\neg\neg\phi$	¬i 2–3
5	ϕ	¬¬e 4

This shows that PBC can be derived from $\to$i, ¬i, $\to$e and ¬¬e.

The final derived rule we consider in this section is arguably the most useful to use in proofs, because its derivation is rather long and complicated, so its usage often saves time and effort. It also has a Latin name, *tertium non datur*; the English name is the law of the excluded middle, or LEM for short. It simply says that $\phi \vee \neg\phi$ is true: whatever ϕ is, it must be either true or false; in the latter case, $\neg\phi$ is true. There is no third possibility (hence *excluded middle*): the sequent $\vdash \phi \vee \neg\phi$ is valid. Its validity is implicit, for example, whenever you write an if-statement in a programming language: '`if` B $\{C_1\}$ `else` $\{C_2\}$' relies on the fact that $B \vee \neg B$ is always true (and that B and $\neg B$ can never be true at the same time). Here is a proof in natural deduction that derives the law of the excluded middle from basic proof rules:

1	$\neg(\phi \vee \neg\phi)$	assumption
2	ϕ	assumption
3	$\phi \vee \neg\phi$	$\vee i_1$ 2
4	$\bot$	¬e 3, 1
5	$\neg\phi$	¬i 2–4
6	$\phi \vee \neg\phi$	$\vee i_2$ 5
7	$\bot$	¬e 6, 1
8	$\neg\neg(\phi \vee \neg\phi)$	¬i 1–7
9	$\phi \vee \neg\phi$	¬¬e 8

Example 1.24 Using LEM, we show that $p \to q \vdash \neg p \vee q$ is valid:

1	$p \to q$	premise
2	$\neg p \vee p$	LEM
3	$\neg p$	assumption
4	$\neg p \vee q$	$\vee\text{i}_1$ 3
5	p	assumption
6	q	$\to$e 1, 5
7	$\neg p \vee q$	$\vee\text{i}_2$ 6
8	$\neg p \vee q$	$\vee$e 2, 3−4, 5−7

It can be difficult to decide which instance of LEM would benefit the progress of a proof. Can you re-do the example above with $q \vee \neg q$ as LEM?

1.2.3 Natural deduction in summary

The proof rules for natural deduction are summarised in Figure 1.2. The explanation of the rules we have given so far in this chapter is *declarative*; we have presented each rule and justified it in terms of our intuition about the logical connectives. However, when you try to use the rules yourself, you'll find yourself looking for a more *procedural* interpretation; what does a rule do and how do you use it? For example,

- $\wedge$i says: to prove $\phi \wedge \psi$, you must first prove ϕ and ψ separately and then use the rule $\wedge$i.
- $\wedge\text{e}_1$ says: to prove ϕ, try proving $\phi \wedge \psi$ and then use the rule $\wedge\text{e}_1$. Actually, this doesn't sound like very good advice because probably proving $\phi \wedge \psi$ will be harder than proving ϕ alone. However, you might find that you *already have* $\phi \wedge \psi$ lying around, so that's when this rule is useful. Compare this with the example sequent in Example 1.15.
- $\vee\text{i}_1$ says: to prove $\phi \vee \psi$, try proving ϕ. Again, in general it is harder to prove ϕ than it is to prove $\phi \vee \psi$, so this will usually be useful only if you've already managed to prove ϕ. For example, if you want to prove $q \vdash p \vee q$, you certainly won't be able simply to use the rule $\vee\text{i}_1$, but $\vee\text{i}_2$ will work.
- $\vee$e has an excellent procedural interpretation. It says: if you have $\phi \vee \psi$, and you want to prove some χ, then try to prove χ from ϕ and from ψ in turn. (In those subproofs, of course you can use the other prevailing premises as well.)
- Similarly, $\to$i says, if you want to prove $\phi \to \psi$, try proving ψ from ϕ (and the other prevailing premises).
- $\neg$i says: to prove $\neg\phi$, prove $\bot$ from ϕ (and the other prevailing premises).

The basic rules of natural deduction:

	introduction	*elimination*
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\,\wedge\mathrm{i}$	$\dfrac{\phi \wedge \psi}{\phi}\,\wedge\mathrm{e}_1 \qquad \dfrac{\phi \wedge \psi}{\psi}\,\wedge\mathrm{e}_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\,\vee\mathrm{i}_1 \qquad \dfrac{\psi}{\phi \vee \psi}\,\vee\mathrm{i}_2$	$\dfrac{\phi \vee \psi \quad \begin{array}{|c|}\hline \phi \\ \vdots \\ \chi \\ \hline\end{array} \quad \begin{array}{|c|}\hline \psi \\ \vdots \\ \chi \\ \hline\end{array}}{\chi}\,\vee\mathrm{e}$
$\rightarrow$	$\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \psi \\ \hline\end{array}}{\phi \rightarrow \psi}\,\rightarrow\mathrm{i}$	$\dfrac{\phi \quad \phi \rightarrow \psi}{\psi}\,\rightarrow\mathrm{e}$
$\neg$	$\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \bot \\ \hline\end{array}}{\neg\phi}\,\neg\mathrm{i}$	$\dfrac{\phi \quad \neg\phi}{\bot}\,\neg\mathrm{e}$
$\bot$	(no introduction rule for $\bot$)	$\dfrac{\bot}{\phi}\,\bot\mathrm{e}$
$\neg\neg$		$\dfrac{\neg\neg\phi}{\phi}\,\neg\neg\mathrm{e}$

Some useful derived rules:

$$\frac{\phi \rightarrow \psi \quad \neg\psi}{\neg\phi}\,\mathrm{MT} \qquad \frac{\phi}{\neg\neg\phi}\,\neg\neg\mathrm{i}$$

$$\frac{\begin{array}{|c|}\hline \neg\phi \\ \vdots \\ \bot \\ \hline\end{array}}{\phi}\,\mathrm{PBC} \qquad \frac{}{\phi \vee \neg\phi}\,\mathrm{LEM}$$

Figure 1.2. Natural deduction rules for propositional logic.

At any stage of a proof, it is permitted to introduce any formula as assumption, by choosing a proof rule that opens a box. As we saw, natural deduction employs boxes to control the scope of assumptions. When an assumption is introduced, a box is opened. Discharging assumptions is achieved by closing a box according to the pattern of its particular proof rule. It's useful to make assumptions by opening boxes. *But don't forget you have to close them in the manner prescribed by their proof rule.*

OK, but how do we actually go about constructing a proof? Given a sequent, you write its premises at the top of your page and its conclusion at the bottom. Now, you're trying to fill in the gap, which involves working simultaneously on the premises (to bring them towards the conclusion) and on the conclusion (to massage it towards the premises).

Look first at the conclusion. If it is of the form $\phi \to \psi$, then apply[6] the rule $\to$i. This means drawing a box with ϕ at the top and ψ at the bottom. So your proof, which started out like this:

$$\begin{array}{c} \vdots \\ \text{premises} \\ \vdots \\ \phi \to \psi \end{array}$$

now looks like this:

$$\begin{array}{l} \vdots \\ \text{premises} \\ \vdots \\ \boxed{\begin{array}{ll} \phi & \text{assumption} \\ \\ \\ \psi & \end{array}} \\ \phi \to \psi \qquad \to\text{i} \end{array}$$

You still have to find a way of filling in the gap between the ϕ and the ψ. But you now have an extra formula to work with and you have simplified the conclusion you are trying to reach.

[6] Except in situations such as $p \to (q \to \neg r), p \vdash q \to \neg r$ where $\to$e produces a simpler proof.

The proof rule ¬i is very similar to →i and has the same beneficial effect on your proof attempt. It gives you an extra premise to work with and simplifies your conclusion.

At any stage of a proof, several rules are likely to be applicable. Before applying any of them, list the applicable ones and think about which one is likely to improve the situation for your proof. You'll find that →i and ¬i most often improve it, so always use them whenever you can. There is no easy recipe for when to use the other rules; often you have to make judicious choices.

1.2.4 Provable equivalence

Definition 1.25 Let ϕ and ψ be formulas of propositional logic. We say that ϕ and ψ are *provably equivalent* iff (we write 'iff' for 'if, and only if' in the sequel) the sequents $\phi \vdash \psi$ and $\psi \vdash \phi$ are valid; that is, there is a proof of ψ from ϕ and another one going the other way around. As seen earlier, we denote that ϕ and ψ are provably equivalent by $\phi \dashv\vdash \psi$.

Note that, by Remark 1.12, we could just as well have defined $\phi \dashv\vdash \psi$ to mean that the sequent $\vdash (\phi \to \psi) \wedge (\psi \to \phi)$ is valid; it defines the same concept. Examples of provably equivalent formulas are

$$\begin{array}{ll} \neg(p \wedge q) \dashv\vdash \neg q \vee \neg p & \neg(p \vee q) \dashv\vdash \neg q \wedge \neg p \\ p \to q \dashv\vdash \neg q \to \neg p & p \to q \dashv\vdash \neg p \vee q \\ p \wedge q \to p \dashv\vdash r \vee \neg r & p \wedge q \to r \dashv\vdash p \to (q \to r). \end{array}$$

The reader should prove all of these six equivalences in natural deduction.

1.2.5 An aside: proof by contradiction

Sometimes we can't prove something *directly* in the sense of taking apart given assumptions and reasoning with their constituents in a constructive way. Indeed, the proof system of natural deduction, summarised in Figure 1.2, specifically allows for *indirect* proofs that lack a constructive quality: for example, the rule

$$\dfrac{\begin{array}{|c|}\hline \neg\phi \\ \vdots \\ \bot \\ \hline \end{array}}{\phi}\ \text{PBC}$$

allows us to prove ϕ by showing that $\neg\phi$ leads to a contradiction. Although 'classical logicians' argue that this is valid, logicians of another kind, called 'intuitionistic logicians,' argue that to prove ϕ you should do it directly, rather than by arguing merely that $\neg\phi$ is impossible. The two other rules on which classical and intuitionistic logicians disagree are

$$\frac{}{\phi \vee \neg\phi}\ \text{LEM} \qquad \frac{\neg\neg\phi}{\phi}\ \neg\neg\text{e}.$$

Intuitionistic logicians argue that, to show $\phi \vee \neg\phi$, you have to show ϕ, or $\neg\phi$. If neither of these can be shown, then the putative truth of the disjunction has no justification. Intuitionists reject $\neg\neg$e since we have already used this rule to prove LEM and PBC from rules which the intuitionists do accept. In the exercises, you are asked to show why the intuitionists also reject PBC.

Let us look at a proof that shows up this difference, involving real numbers. Real numbers are floating point numbers like 23.54721, only some of them might actually be infinitely long such as 23.138592748500123950734..., with no periodic behaviour after the decimal point.

Given a positive real number a and a *natural* (whole) number b, we can calculate a^b: it is just a times itself, b times, so $2^2 = 2 \cdot 2 = 4$, $2^3 = 2 \cdot 2 \cdot 2 = 8$ and so on. When b is a *real* number, we can also define a^b, as follows. We say that $a^0 \stackrel{\text{def}}{=} 1$ and, for a non-zero rational number k/n, where $n \neq 0$, we let $a^{k/n} \stackrel{\text{def}}{=} \sqrt[n]{a^k}$ where $\sqrt[n]{x}$ is the real number y such that $y^n = x$. From real analysis one knows that any real number b can be approximated by a sequence of rational numbers k_0/n_0, k_1/n_1, ... Then we define a^b to be the real number approximated by the sequence a^{k_0/n_0}, a^{k_1/n_1}, ... (In calculus, one can show that this 'limit' a^b is unique and independent of the choice of approximating sequence.) Also, one calls a real number *irrational* if it can't be written in the form k/n for some integers k and $n \neq 0$. In the exercises you will be asked to find a semi-formal proof showing that $\sqrt{2}$ is irrational.

We now present a proof of a fact about real numbers in the informal style used by mathematicians (this proof can be formalised as a natural deduction proof in the logic presented in Chapter 2). The fact we prove is:

Theorem 1.26 *There exist irrational numbers a and b such that a^b is rational.*

PROOF: We choose b to be $\sqrt{2}$ and proceed by a case analysis. Either b^b is irrational, or it is not. (Thus, our proof uses $\vee$e on an instance of LEM.)

(i) Assume that b^b is rational. Then this proof is easy since we can choose irrational numbers a and b to be $\sqrt{2}$ and see that a^b is just b^b which was assumed to be rational.

(ii) Assume that b^b is *ir*rational. Then we change our strategy slightly and choose a to be $\sqrt{2}^{\sqrt{2}}$. Clearly, a is irrational by the assumption of case (ii). But we know that b is irrational (this was known by the ancient Greeks; see the proof outline in the exercises). So a and b are both irrational numbers and

$$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = (\sqrt{2})^2 = 2$$

is rational, where we used the law $(x^y)^z = x^{(y\cdot z)}$.

Since the two cases above are exhaustive (*either* b^b is irrational, *or* it isn't) we have proven the theorem. □

This proof is perfectly legitimate and mathematicians use arguments like that all the time. The exhaustive nature of the case analysis above rests on the use of the rule LEM, which we use to prove that either b is rational or it is not. Yet, there is something puzzling about it. Surely, we have secured the fact that there are irrational numbers a and b such that a^b is rational, but are we in a position to specify an actual pair of such numbers satisfying this theorem? More precisely, which of the pairs (a, b) above fulfils the assertion of the theorem, the pair $(\sqrt{2}, \sqrt{2})$, or the pair $(\sqrt{2}^{\sqrt{2}}, \sqrt{2})$? Our proof tells us nothing about *which* of them is the right choice; it just says that at least one of them works.

Thus, the intuitionists favour a calculus containing the introduction and elimination rules shown in Figure 1.2 and excluding the rule ¬¬e and the derived rules. Intuitionistic logic turns out to have some specialised applications in computer science, such as modelling type-inference systems used in compilers or the staged execution of program code; but in this text we stick to the full so-called classical logic which includes all the rules.

1.3 Propositional logic as a formal language

In the previous section we learned about propositional atoms and how they can be used to build more complex logical formulas. We were deliberately informal about that, for our main focus was on trying to understand the precise mechanics of the natural deduction rules. However, it should have been clear that the rules we stated are valid for *any* formulas we can form, as long as they match the pattern required by the respective rule. For example,

the application of the proof rule →e in

1	$p \to q$	premise
2	p	premise
3	q	→e 1, 2

is equally valid if we substitute p with $p \vee \neg r$ and q with $r \to p$:

1	$p \vee \neg r \to (r \to p)$	premise
2	$p \vee \neg r$	premise
3	$r \to p$	→e 1, 2

This is why we expressed such rules as schemes with Greek symbols standing for generic formulas. Yet, it is time that we make precise the notion of 'any formula we may form.' Because this text concerns various logics, we will introduce in (1.3) an easy formalism for specifying well-formed formulas. In general, we need an *unbounded* supply of propositional atoms $p, q, r, \ldots$, or $p_1, p_2, p_3, \ldots$ You should not be too worried about the need for infinitely many such symbols. Although we may only need *finitely many* of these propositions to describe a property of a computer program successfully, we cannot specify how many such atomic propositions we will need in any concrete situation, so having infinitely many symbols at our disposal is a cheap way out. This can be compared with the potentially infinite nature of English: the number of grammatically correct English sentences is infinite, but finitely many such sentences will do in whatever situation you might be in (writing a book, attending a lecture, listening to the radio, having a dinner date, ...).

Formulas in our propositional logic should certainly be strings over the alphabet $\{p, q, r, \ldots\} \cup \{p_1, p_2, p_3, \ldots\} \cup \{\neg, \wedge, \vee, \to, (,)\}$. This is a trivial observation and as such is not good enough for what we are trying to capture. For example, the string $(\neg)() \vee pq \to$ is a word over that alphabet, yet, it does not seem to make a lot of sense as far as propositional logic is concerned. So what we have to define are those strings which we want to call formulas. We call such formulas *well-formed*.

Definition 1.27 The well-formed formulas of propositional logic are those which we obtain by using the construction rules below, and only those, finitely many times:

atom: Every propositional atom $p, q, r, \ldots$ and $p_1, p_2, p_3, \ldots$ is a well-formed formula.
¬: If ϕ is a well-formed formula, then so is $(\neg\phi)$.
∧: If ϕ and ψ are well-formed formulas, then so is $(\phi \wedge \psi)$.
∨: If ϕ and ψ are well-formed formulas, then so is $(\phi \vee \psi)$.
→: If ϕ and ψ are well-formed formulas, then so is $(\phi \to \psi)$.

It is most crucial to realize that this definition is the one a computer would expect and that we did not make use of the binding priorities agreed upon in the previous section.

Convention. In this section we act as if we are a rigorous computer and we call formulas well-formed iff they can be deduced to be so using the definition above.

Further, note that the condition 'and only those' in the definition above rules out the possibility of any other means of establishing that formulas are well-formed. Inductive definitions, like the one of well-formed propositional logic formulas above, are so frequent that they are often given by a defining grammar in Backus Naur form (BNF). In that form, the above definition reads more compactly as

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \tag{1.3}$$

where p stands for any atomic proposition and each occurrence of ϕ to the right of $::=$ stands for any already constructed formula.

So how can we show that a string is a well-formed formula? For example, how do we answer this for ϕ being

$$(((\neg p) \wedge q) \to (p \wedge (q \vee (\neg r)))) \ ? \tag{1.4}$$

Such reasoning is greatly facilitated by the fact that the grammar in (1.3) satisfies the *inversion principle*, which means that we can invert the process of building formulas: although the grammar rules allow for five different ways of constructing more complex formulas – the five clauses in (1.3) – there is always a unique clause which was used last. For the formula above, this last operation was an application of the fifth clause, for ϕ is an implication with the assumption $((\neg p) \wedge q)$ and conclusion $(p \wedge (q \vee (\neg r)))$. By applying the inversion principle to the assumption, we see that it is a conjunction of $(\neg p)$ and q. The former has been constructed using the second clause and is well-formed since p is well-formed by the first clause in (1.3). The latter is well-formed for the same reason. Similarly, we can apply the inversion

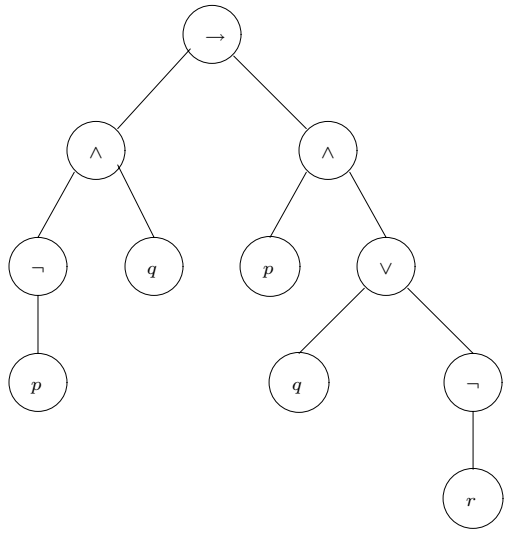


Figure 1.3. A parse tree representing a well-formed formula.

principle to the conclusion $(p \wedge (q \vee (\neg r)))$, inferring that it is indeed well-formed. In summary, the formula in (1.4) is well-formed.

For us humans, dealing with brackets is a tedious task. The reason we need them is that formulas really have a tree-like structure, although we prefer to represent them in a linear way. In Figure 1.3 you can see the parse tree[7] of the well-formed formula ϕ in (1.4). Note how brackets become unnecessary in this parse tree since the paths and the branching structure of this tree remove any possible ambiguity in interpreting ϕ. In representing ϕ as a linear string, the branching structure of the tree is retained by the insertion of brackets as done in the definition of well-formed formulas.

So how would you go about showing that a string of symbols ψ is *not* well-formed? At first sight, this is a bit trickier since we somehow have to make sure that ψ could not have been obtained by *any* sequence of construction rules. Let us look at the formula $(\neg)() \vee pq \rightarrow$ from above. We can decide this matter by being very observant. The string $(\neg)() \vee pq \rightarrow$ contains $\neg)$ and $\neg$ cannot be the rightmost symbol of a well-formed formula (check all the rules to verify this claim!); but the only time we can put a ')' to the right of something is if that something is a well-formed formula (again, check all the rules to see that this is so). Thus, $(\neg)() \vee pq \rightarrow$ is *not* well-formed.

Probably the easiest way to verify whether some formula ϕ is well-formed is by trying to draw its parse tree. In this way, you can verify that the

[7] We will use this name without explaining it any further and are confident that you will understand its meaning through the examples.

formula in (1.4) is well-formed. In Figure 1.3 we see that its parse tree has $\rightarrow$ as its root, expressing that the formula is, at its top level, an implication. Using the grammar clause for implication, it suffices to show that the left and right subtrees of this root node are well-formed. That is, we proceed in a top-down fashion and, in this case, successfully. Note that the parse trees of well-formed formulas have either an atom as root (and then this is all there is in the tree), or the root contains $\neg$, $\vee$, $\wedge$ or $\rightarrow$. In the case of $\neg$ there is only *one* subtree coming out of the root. In the cases $\wedge$, $\vee$ or $\rightarrow$ we must have *two* subtrees, each of which must behave as just described; this is another example of an *inductive* definition.

Thinking in terms of trees will help you understand standard notions in logic, for example, the concept of a *subformula*. Given the well-formed formula ϕ above, its subformulas are just the ones that correspond to the subtrees of its parse tree in Figure 1.3. So we can list all its leaves p, q (occurring twice), and r, then $(\neg p)$ and $((\neg p) \wedge q)$ on the left subtree of $\rightarrow$ and $(\neg r)$, $(q \vee (\neg r))$ and $((p \wedge (q \vee (\neg p))))$ on the right subtree of $\rightarrow$. The whole tree is a subtree of itself as well. So we can list all nine subformulas of ϕ as

$$
\begin{array}{l}
p \\
q \\
r \\
(\neg p) \\
((\neg p) \wedge q) \\
(\neg r) \\
(q \vee (\neg r)) \\
((p \wedge (q \vee (\neg r)))) \\
(((\neg p) \wedge q) \rightarrow (p \wedge (q \vee (\neg r)))).
\end{array}
$$

Let us consider the tree in Figure 1.4. Why does it represent a well-formed formula? All its leaves are propositional atoms (p twice, q and r), all branching nodes are logical connectives ($\neg$ twice, $\wedge$, $\vee$ and $\rightarrow$) and the numbers of subtrees are correct in all those cases (one subtree for a $\neg$ node and two subtrees for all other non-leaf nodes). How do we obtain the linear representation of this formula? If we ignore brackets, then we are seeking nothing but the *in-order* representation of this tree as a list[8]. The resulting well-formed formula is $((\neg(p \vee (q \rightarrow (\neg p)))) \wedge r)$.

[8] The other common ways of flattening trees to lists are *preordering* and *postordering*. See any text on binary trees as data structures for further details.

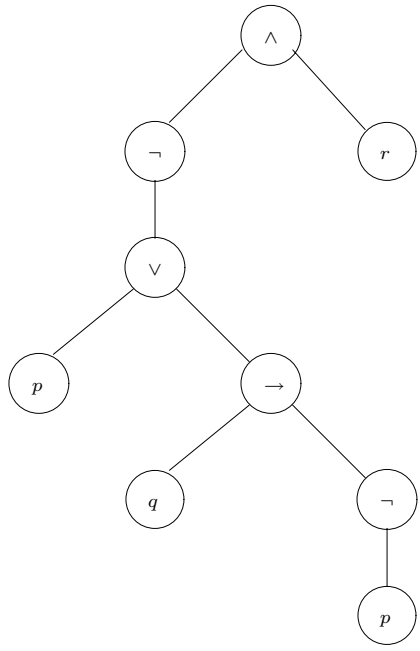


Figure 1.4. Given: a tree; wanted: its linear representation as a logical formula.

The tree in Figure 1.21 on page 82, however, does *not* represent a well-formed formula for two reasons. First, the leaf $\wedge$ (and a similar argument applies to the leaf $\neg$), the left subtree of the node $\rightarrow$, is not a propositional atom. This could be fixed by saying that we decided to leave the left and right subtree of that node unspecified and that we are willing to provide those now. However, the second reason is fatal. The p node is not a leaf since it has a subtree, the node $\neg$. This cannot make sense if we think of the entire tree as some logical formula. So this tree does not represent a well-formed logical formula.

1.4 Semantics of propositional logic

1.4.1 The meaning of logical connectives

In the second section of this chapter, we developed a calculus of reasoning which could verify that sequents of the form $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ are valid, which means: from the premises ϕ_1, ϕ_2, …, ϕ_n, we may conclude ψ.

In this section we give another account of this relationship between the premises ϕ_1, ϕ_2, …, ϕ_n and the conclusion ψ. To contrast with the sequent

above, we define a new relationship, written

$$\phi_1, \phi_2, \ldots, \phi_n \vDash \psi.$$

This account is based on looking at the 'truth values' of the atomic formulas in the premises and the conclusion; and at how the logical connectives manipulate these truth values. What is the truth value of a declarative sentence, like sentence (3) 'Every even natural number > 2 is the sum of two prime numbers'? Well, declarative sentences express a fact about the real world, the physical world we live in, or more abstract ones such as computer models, or our thoughts and feelings. Such factual statements either match reality (they are *true*), or they don't (they are *false*).

If we combine declarative sentences p and q with a logical connective, say $\wedge$, then the truth value of $p \wedge q$ is determined by three things: the truth value of p, the truth value of q and the meaning of $\wedge$. The meaning of $\wedge$ is captured by the observation that $p \wedge q$ is true iff p *and* q are both true; otherwise $p \wedge q$ is false. Thus, as far as $\wedge$ is concerned, it needs only to know whether p and q are true, it does *not* need to know what p and q are actually saying about the world out there. This is also the case for all the other logical connectives and is the reason why we can compute the truth value of a formula just by knowing the truth values of the atomic propositions occurring in it.

Definition 1.28 1. The set of truth values contains two elements `T` and `F`, where `T` represents 'true' and `F` represents 'false'.
2. A *valuation* or *model* of a formula ϕ is an assignment of each propositional atom in ϕ to a truth value.

Example 1.29 The map which assigns `T` to q and `F` to p is a valuation for $p \vee \neg q$. Please list the remaining three valuations for this formula.

We can think of the meaning of $\wedge$ as a function of two arguments; each argument is a truth value and the result is again such a truth value. We specify this function in a table, called the *truth table for conjunction*, which you can see in Figure 1.5. In the first column, labelled ϕ, we list all possible

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

Figure 1.5. The truth table for conjunction, the logical connective $\wedge$.

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

ϕ	ψ	$\phi \vee \psi$
T	T	T
T	F	T
F	T	T
F	F	F

ϕ	ψ	$\phi \to \psi$
T	T	T
T	F	F
F	T	T
F	F	T

ϕ	$\neg\phi$
T	F
F	T

$\top$
T

$\bot$
F

Figure 1.6. The truth tables for all the logical connectives discussed so far.

truth values of ϕ. Actually we list them *twice* since we also have to deal with another formula ψ, so the possible number of combinations of truth values for ϕ and ψ equals $2 \cdot 2 = 4$. Notice that the four pairs of ϕ and ψ values in the first two columns really exhaust all those possibilities (TT, TF, FT and FF). In the third column, we list the result of $\phi \wedge \psi$ according to the truth values of ϕ and ψ. So in the first line, where ϕ and ψ have value T, the result is T again. In all other lines, the result is F since at least one of the propositions ϕ or ψ has value F.

In Figure 1.6 you find the truth tables for all logical connectives of propositional logic. Note that $\neg$ turns T into F and vice versa. Disjunction is the mirror image of conjunction if we swap T and F, namely, a disjunction returns F iff both arguments are equal to F, otherwise (= at least one of the arguments equals T) it returns T. The behaviour of implication is not quite as intuitive. Think of the meaning of $\to$ as checking whether *truth is being preserved*. Clearly, this is not the case when we have $\mathtt{T} \to \mathtt{F}$, since we infer something that is false from something that is true. So the second entry in the column $\phi \to \psi$ equals F. On the other hand, $\mathtt{T} \to \mathtt{T}$ obviously preserves truth, but so do the cases $\mathtt{F} \to \mathtt{T}$ and $\mathtt{F} \to \mathtt{F}$, because there is no truth to be preserved in the first place as the assumption of the implication is false.

If you feel slightly uncomfortable with the semantics (= the meaning) of $\to$, then it might be good to think of $\phi \to \psi$ as an abbreviation of the formula $\neg\phi \vee \psi$ *as far as meaning is concerned*; these two formulas are very different syntactically and natural deduction treats them differently as well. But using the truth tables for $\neg$ and $\vee$ you can check that $\phi \to \psi$ evaluates

to T iff $\neg\phi \vee \psi$ does so. This means that $\phi \to \psi$ and $\neg\phi \vee \psi$ are *semantically equivalent*; more on that in Section 1.5.

Given a formula ϕ which contains the propositional atoms $p_1, p_2, \ldots, p_n$, we can construct a truth table for ϕ, at least in principle. The caveat is that this truth table has 2^n many lines, each line listing a possible combination of truth values for $p_1, p_2, \ldots, p_n$; and for large n this task is impossible to complete. Our aim is thus to compute the value of ϕ for each of these 2^n cases for moderately small values of n. Let us consider the example ϕ in Figure 1.3. It involves three propositional atoms ($n = 3$) so we have $2^3 = 8$ cases to consider.

We illustrate how things go for one particular case, namely for the valuation in which q evaluates to F; and p and r evaluate to T. What does $\neg p \wedge q \to p \wedge (q \vee \neg r)$ evaluate to? Well, the beauty of our semantics is that it is *compositional*. If we know the meaning of the subformulas $\neg p \wedge q$ and $p \wedge (q \vee \neg r)$, then we just have to look up the appropriate line of the $\to$ truth table to find the value of ϕ, for ϕ is an implication of these two subformulas. Therefore, we can do the calculation by traversing the parse tree of ϕ in a bottom-up fashion. We know what its leaves evaluate to since we stated what the atoms p, q and r evaluated to. Because the meaning of p is T, we see that $\neg p$ computes to F. Now q is assumed to represent F and the conjunction of F and F is F. Thus, the left subtree of the node $\to$ evaluates to F. As for the right subtree of $\to$, r stands for T so $\neg r$ computes to F and q means F, so the disjunction of F and F is still F. We have to take that result, F, and compute its conjunction with the meaning of p which is T. Since the conjunction of T and F is F, we get F as the meaning of the right subtree of $\to$. Finally, to evaluate the meaning of ϕ, we compute $\mathtt{F} \to \mathtt{F}$ which is T. Figure 1.7 shows how the truth values propagate upwards to reach the root whose associated truth value is the truth value of ϕ given the meanings of p, q and r above.

It should now be quite clear how to build a truth table for more complex formulas. Figure 1.8 contains a truth table for the formula $(p \to \neg q) \to (q \vee \neg p)$. To be more precise, the first two columns list all possible combinations of values for p and q. The next two columns compute the corresponding values for $\neg p$ and $\neg q$. Using these four columns, we may compute the column for $p \to \neg q$ and $q \vee \neg p$. To do so we think of the first and fourth columns as the data for the $\to$ truth table and compute the column of $p \to \neg q$ accordingly. For example, in the first line p is T and $\neg q$ is F so the entry for $p \to \neg q$ is $\mathtt{T} \to \mathtt{F} = \mathtt{F}$ by definition of the meaning of $\to$. In this fashion, we can fill out the rest of the fifth column. Column 6 works similarly, only we now need to look up the truth table for $\vee$ with columns 2 and 3 as input.

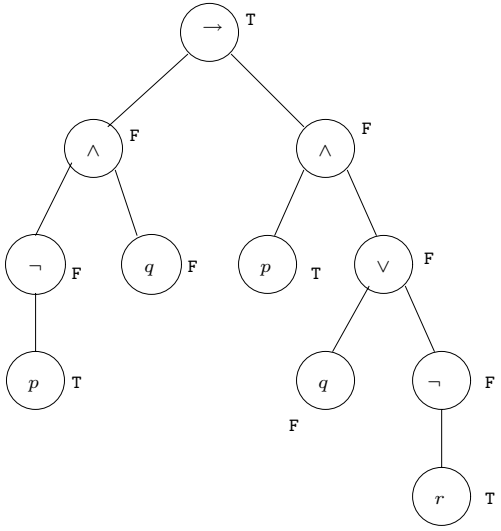


Figure 1.7. The evaluation of a logical formula under a given valuation.

p	q	$\neg p$	$\neg q$	$p \to \neg q$	$q \vee \neg p$	$(p \to \neg q) \to (q \vee \neg p)$
T	T	F	F	F	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	T
F	F	T	T	T	T	T

Figure 1.8. An example of a truth table for a more complex logical formula.

Finally, column 7 results from applying the truth table of $\to$ to columns 5 and 6.

1.4.2 Mathematical induction

Here is a little anecdote about the German mathematician Gauss who, as a pupil at age 8, did not pay attention in class (can you imagine?), with the result that his teacher made him sum up all natural numbers from 1 to 100. The story has it that Gauss came up with the correct answer 5050 within seconds, which infuriated his teacher. How did Gauss do it? Well, possibly he knew that

$$1 + 2 + 3 + 4 + \cdots + n = \frac{n \cdot (n+1)}{2} \tag{1.5}$$

for all natural numbers n.[9] Thus, taking $n = 100$, Gauss could easily calculate:

$$1 + 2 + 3 + 4 + \cdots + 100 = \frac{100 \cdot 101}{2} = 5050.$$

Mathematical induction allows us to prove equations, such as the one in (1.5), for arbitrary n. More generally, it allows us to show that *every* natural number satisfies a certain property. Suppose we have a property M which we think is true of all natural numbers. We write $M(5)$ to say that the property is true of 5, etc. Suppose that we know the following two things about the property M:

1. **Base case:** The natural number 1 has property M, i.e. we have a proof of $M(1)$.
2. **Inductive step:** If n is a natural number which *we assume* to have property $M(n)$, then *we can show* that $n + 1$ has property $M(n + 1)$; i.e. we have a proof of $M(n) \to M(n + 1)$.

Definition 1.30 The principle of mathematical induction says that, on the grounds of these two pieces of information above, every natural number n has property $M(n)$. The assumption of $M(n)$ in the inductive step is called the *induction hypothesis*.

Why does this principle make sense? Well, take *any* natural number k. If k equals 1, then k has property $M(1)$ using the base case and so we are done. Otherwise, we can use the inductive step, applied to $n = 1$, to infer that $2 = 1 + 1$ has property $M(2)$. We can do that using $\to$e, for we know that 1 has the property in question. Now we use that same inductive step on $n = 2$ to infer that 3 has property $M(3)$ and we repeat this until we reach $n = k$ (see Figure 1.9). Therefore, we should have no objections about using the principle of mathematical induction for natural numbers.

Returning to Gauss' example we claim that the sum $1 + 2 + 3 + 4 + \cdots + n$ equals $n \cdot (n + 1)/2$ for all natural numbers n.

Theorem 1.31 *The sum* $1 + 2 + 3 + 4 + \cdots + n$ *equals* $n \cdot (n + 1)/2$ *for all natural numbers* n.

[9] There is another way of finding the sum $1 + 2 + \cdots + 100$, which works like this: write the sum backwards, as $100 + 99 + \cdots + 1$. Now add the forwards and backwards versions, obtaining $101 + 101 + \cdots + 101$ (100 times), which is 10100. Since we added the sum to itself, we now divide by two to get the answer 5050. Gauss probably used this method; but the method of mathematical induction that we explore in this section is much more powerful and can be applied in a wide variety of situations.

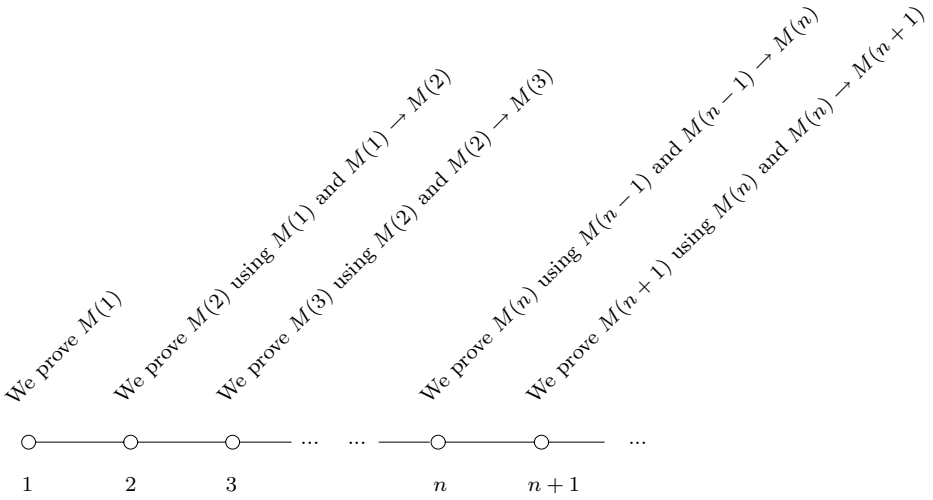


Figure 1.9. How the principle of mathematical induction works. By proving just two facts, $M(1)$ and $M(n) \to M(n+1)$ for a formal (and unconstrained) parameter n, we are able to deduce $M(k)$ for each natural number k.

Proof: We use mathematical induction. In order to reveal the fine structure of our proof we write LHS_n for the expression $1+2+3+4+\cdots+n$ and RHS_n for $n \cdot (n+1)/2$. Thus, we need to show $\mathrm{LHS}_n = \mathrm{RHS}_n$ for all $n \geq 1$.

Base case: If n equals 1, then LHS_1 is just 1 (there is only one summand), which happens to equal $\mathrm{RHS}_1 = 1 \cdot (1+1)/2$.

Inductive step: Let us assume that $\mathrm{LHS}_n = \mathrm{RHS}_n$. Recall that this assumption is called the induction hypothesis; it is the driving force of our argument. We need to show $\mathrm{LHS}_{n+1} = \mathrm{RHS}_{n+1}$, i.e. that the longer sum $1+2+3+4+\cdots+(n+1)$ equals $(n+1) \cdot ((n+1)+1)/2$. The key observation is that the sum $1+2+3+4+\cdots+(n+1)$ is nothing but the sum $(1+2+3+4+\cdots+n)+(n+1)$ of two summands, where the first one is the sum of our induction hypothesis. The latter says that $1+2+3+4+\cdots+n$ equals $n \cdot (n+1)/2$, and we are certainly entitled to substitute equals for equals in our reasoning. Thus, we compute

$$\begin{aligned} &\mathrm{LHS}_{n+1} \\ &\quad = 1+2+3+4+\cdots+(n+1) \\ &\quad = \mathrm{LHS}_n + (n+1) \quad \text{regrouping the sum} \end{aligned}$$

$$
\begin{aligned}
&= \mathrm{RHS}_n + (n+1) \quad \text{by our induction hypothesis} \\
&= \tfrac{n\cdot(n+1)}{2} + (n+1) \\
&= \tfrac{n\cdot(n+1)}{2} + \tfrac{2\cdot(n+1)}{2} \quad \text{arithmetic} \\
&= \tfrac{(n+2)\cdot(n+1)}{2} \quad \text{arithmetic} \\
&= \tfrac{((n+1)+1)\cdot(n+1)}{2} \quad \text{arithmetic} \\
&= \mathrm{RHS}_{n+1}.
\end{aligned}
$$

Since we successfully showed the base case and the inductive step, we can use mathematical induction to infer that all natural numbers n have the property stated in the theorem above. □

Actually, there are numerous variations of this principle. For example, we can think of a version in which the base case is $n = 0$, which would then cover all natural numbers including 0. Some statements hold only for all natural numbers, say, greater than 3. So you would have to deal with a base case 4, but keep the version of the inductive step (see the exercises for such an example). The use of mathematical induction typically suceeds on properties $M(n)$ that involve inductive definitions (e.g. the definition of k^l with $l \geq 0$). Sentence (3) on page 2 suggests there may be true properties $M(n)$ for which mathematical induction won't work.

Course-of-values induction. There is a variant of mathematical induction in which the induction hypothesis for proving $M(n+1)$ is not just $M(n)$, but the conjunction $M(1) \wedge M(2) \wedge \cdots \wedge M(n)$. In that variant, called *course-of-values* induction, there doesn't have to be an explicit base case at all – everything can be done in the inductive step.

How can this work without a base case? The answer is that the base case is implicitly included in the inductive step. Consider the case $n = 3$: the inductive-step instance is $M(1) \wedge M(2) \wedge M(3) \rightarrow M(4)$. Now consider $n = 1$: the inductive-step instance is $M(1) \rightarrow M(2)$. What about the case when n equals 0? In this case, there are zero formulas on the left of the $\rightarrow$, so we have to prove $M(1)$ from nothing at all. The inductive-step instance is simply the obligation to show $M(1)$. You might find it useful to modify Figure 1.9 for course-of-values induction.

Having said that the base case is implicit in course-of-values induction, it frequently turns out that it still demands special attention when you get inside trying to prove the inductive case. We will see precisely this in the two applications of course-of-values induction in the following pages.

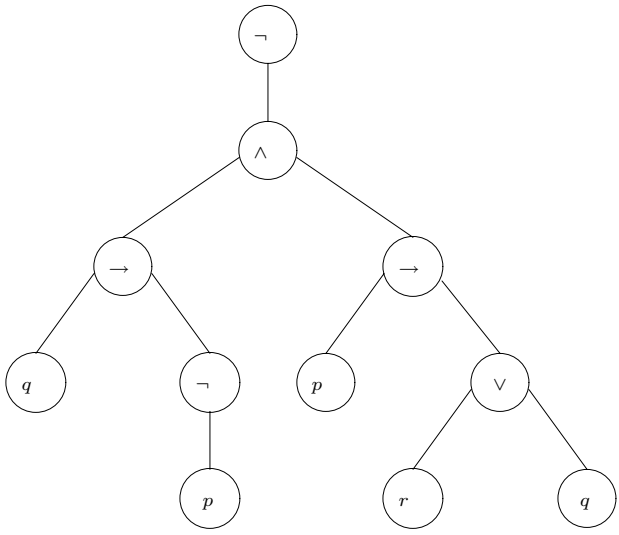


Figure 1.10. A parse tree with height 5.

In computer science, we often deal with finite structures of some kind, data structures, programs, files etc. Often we need to show that *every* instance of such a structure has a certain property. For example, the well-formed formulas of Definition 1.27 have the property that the number of '(' brackets in a particular formula equals its number of ')' brackets. We can use mathematical induction on the domain of natural numbers to prove this. In order to succeed, we somehow need to connect well-formed formulas to natural numbers.

Definition 1.32 Given a well-formed formula ϕ, we define its *height* to be 1 plus the length of the longest path of its parse tree.

For example, consider the well-formed formulas in Figures 1.3, 1.4 and 1.10. Their heights are 5, 6 and 5, respectively. In Figure 1.3, the longest path goes from $\rightarrow$ to $\wedge$ to $\vee$ to $\neg$ to r, a path of length 4, so the height is $4 + 1 = 5$. Note that the height of atoms is $1 + 0 = 1$. Since every well-formed formula has finite height, we can show statements about all well-formed formulas by mathematical induction on their height. This trick is most often called *structural induction*, an important reasoning technique in computer science. Using the notion of the height of a parse tree, we realise that structural induction is just a special case of course-of-values induction.

Theorem 1.33 *For every well-formed propositional logic formula, the number of left brackets is equal to the number of right brackets.*

PROOF: We proceed by course-of-values induction on the height of well-formed formulas ϕ. Let $M(n)$ mean 'All formulas of height n have the same number of left and right brackets.' We assume $M(k)$ for each $k < n$ and try to prove $M(n)$. Take a formula ϕ of height n.

- **Base case:** Then $n = 1$. This means that ϕ is just a propositional atom. So there are no left or right brackets, 0 equals 0.
- **Course-of-values inductive step:** Then $n > 1$ and so the root of the parse tree of ϕ must be $\neg$, $\rightarrow$, $\vee$ or $\wedge$, for ϕ is well-formed. We assume that it is $\rightarrow$, the other three cases are argued in a similar way. Then ϕ equals $(\phi_1 \rightarrow \phi_2)$ for some well-formed formulas ϕ_1 and ϕ_2 (of course, they are just the left, respectively right, linear representations of the root's two subtrees). It is clear that the heights of ϕ_1 and ϕ_2 are strictly smaller than n. Using the induction hypothesis, we therefore conclude that ϕ_1 has the same number of left and right brackets and that the same is true for ϕ_2. But in $(\phi_1 \rightarrow \phi_2)$ we added just two more brackets, one '(' and one ')'. Thus, the number of occurrences of '(' and ')' in ϕ is the same. □

The formula $(p \rightarrow (q \wedge \neg r))$ illustrates why we could not prove the above directly with mathematical induction on the height of formulas. While this formula has height 4, its two subtrees have heights 1 and 3, respectively. Thus, an induction hypothesis for height 3 would have worked for the right subtree but failed for the left subtree.

1.4.3 Soundness of propositional logic

The natural deduction rules make it possible for us to develop rigorous threads of argumentation, in the course of which we arrive at a conclusion ψ assuming certain other propositions $\phi_1, \phi_2, \ldots, \phi_n$. In that case, we said that the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid. Do we have any evidence that these rules are all *correct* in the sense that valid sequents all 'preserve truth' computed by our truth-table semantics?

Given a proof of $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$, is it conceivable that there is a valuation in which ψ above is false although all propositions $\phi_1, \phi_2, \ldots, \phi_n$ are true? Fortunately, this is not the case and in this subsection we demonstrate why this is so. Let us suppose that some proof in our natural deduction calculus has established that the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid. We need to show: for all valuations in which all propositions $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to `T`, ψ evaluates to `T` as well.

Definition 1.34 If, for all valuations in which all $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to T, ψ evaluates to T as well, we say that

$$\phi_1, \phi_2, \ldots, \phi_n \models \psi$$

holds and call $\models$ the *semantic entailment* relation.

Let us look at some examples of this notion.

1. Does $p \wedge q \models p$ hold? Well, we have to inspect all assignments of truth values to p and q; there are four of these. Whenever such an assignment computes T for $p \wedge q$ we need to make sure that p is true as well. But $p \wedge q$ computes T only if p and q are true, so $p \wedge q \models p$ is indeed the case.
2. What about the relationship $p \vee q \models p$? There are three assignments for which $p \vee q$ computes T, so p would have to be true for all of these. However, if we assign T to q and F to p, then $p \vee q$ computes T, but p is false. Thus, $p \vee q \models p$ does not hold.
3. What if we modify the above to $\neg q, p \vee q \models p$? Notice that we have to be concerned only about valuations in which $\neg q$ *and* $p \vee q$ evaluate to T. This forces q to be false, which in turn forces p to be true. Hence $\neg q, p \vee q \models p$ is the case.
4. Note that $p \models q \vee \neg q$ holds, despite the fact that no atomic proposition on the right of $\models$ occurs on the left of $\models$.

From the discussion above we realize that a soundness argument has to show: if $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds.

Theorem 1.35 (Soundness) *Let $\phi_1, \phi_2, \ldots, \phi_n$ and ψ be propositional logic formulas. If $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds.*

PROOF: Since $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid we know there is a proof of ψ from the premises $\phi_1, \phi_2, \ldots, \phi_n$. We now do a pretty slick thing, namely, we reason by *mathematical induction on the length of this proof!* The length of a proof is just the number of lines it involves. So let us be perfectly clear about what it is we mean to show. We intend to show the assertion $M(k)$:

> *'For all sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ ($n \geq 0$) which have a proof of length k, it is the case that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds.'*

by course-of-values induction on the natural number k. This idea requires

some work, though. The sequent $p \wedge q \to r \vdash p \to (q \to r)$ has a proof

1	$p \wedge q \to r$	premise
2	p	assumption
3	q	assumption
4	$p \wedge q$	$\wedge$i 2, 3
5	r	$\to$e 1, 4
6	$q \to r$	$\to$i 3–5
7	$p \to (q \to r)$	$\to$i 2–6

but if we remove the last line or several of the last lines, we no longer have a proof as the outermost box does not get closed. We get a complete proof, though, by removing the last line and re-writing the assumption of the outermost box as a premise:

1	$p \wedge q \to r$	premise
2	p	premise
3	q	assumption
4	$p \wedge q$	$\wedge$i 2, 3
5	r	$\to$e 1, 4
6	$q \to r$	$\to$i 3–5

This is a proof of the sequent $p \wedge q \to r,\ p \vdash p \to r$. The induction hypothesis then ensures that $p \wedge q \to r,\ p \vDash p \to r$ holds. But then we can also reason that $p \wedge q \to r \vDash p \to (q \to r)$ holds as well – why?

Let's proceed with our proof by induction. We assume $M(k')$ for each $k' < k$ and we try to prove $M(k)$.

Base case: a one-line proof. If the proof has length 1 ($k = 1$), then it must be of the form

1	ϕ	premise

since all other rules involve more than one line. This is the case when $n = 1$ and ϕ_1 and ψ equal ϕ, i.e. we are dealing with the sequent $\phi \vdash \phi$. Of course, since ϕ evaluates to `T` so does ϕ. Thus, $\phi \vDash \phi$ holds as claimed.

Course-of-values inductive step: Let us assume that the proof of the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ has length k and that the statement we want to prove is true for all numbers less than k. Our proof has the following structure:

$$\begin{array}{ll} 1 & \phi_1 \text{ premise} \\ 2 & \phi_2 \text{ premise} \\ & \vdots \\ n & \phi_n \text{ premise} \\ & \vdots \\ k & \psi \text{ justification} \end{array}$$

There are two things we don't know at this point. First, what is happening in between those dots? Second, what was the last rule applied, i.e. what is the justification of the last line? The first uncertainty is of no concern; this is where mathematical induction demonstrates its power. The second lack of knowledge is where all the work sits. In this generality, there is simply no way of knowing which rule was applied last, so we need to consider all such rules in turn.

1. Let us suppose that this last rule is $\wedge$i. Then we know that ψ is of the form $\psi_1 \wedge \psi_2$ and the justification in line k refers to two lines further up which have ψ_1, respectively ψ_2, as their conclusions. Suppose that these lines are k_1 and k_2. Since k_1 and k_2 are smaller than k, we see that there exist proofs of the sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_1$ and $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_2$ with length *less than* k – just take the first k_1, respectively k_2, lines of our original proof. Using the induction hypothesis, we conclude that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_1$ and $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_2$ holds. But these two relations imply that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_1 \wedge \psi_2$ holds as well – why?
2. If ψ has been shown using the rule $\vee$e, then we must have proved, assumed or given as a premise some formula $\eta_1 \vee \eta_2$ in some line k' with $k' < k$, which was referred to via $\vee$e in the justification of line k. Thus, we have a shorter proof of the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \eta_1 \vee \eta_2$ within that proof, obtained by turning all assumptions of boxes that are open at line k' into premises. In a similar way we obtain proofs of the sequents $\phi_1, \phi_2, \ldots, \phi_n, \eta_1 \vdash \psi$ and $\phi_1, \phi_2, \ldots, \phi_n, \eta_2 \vdash \psi$ from the case analysis of $\vee$e. By our induction hypothesis, we conclude that the relations $\phi_1, \phi_2, \ldots, \phi_n \vDash \eta_1 \vee \eta_2$, $\phi_1, \phi_2, \ldots, \phi_n, \eta_1 \vDash \psi$ and $\phi_1, \phi_2, \ldots, \phi_n, \eta_2 \vDash \psi$ hold. But together these three relations then force that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds as well – why?
3. You can guess by now that the rest of the argument checks each possible proof rule in turn and ultimately boils down to verifying that our natural deduction

rules behave semantically in the same way as their corresponding truth tables evaluate. We leave the details as an exercise. □

The soundness of propositional logic is useful in ensuring the *non-existence* of a proof for a given sequent. Let's say you try to prove that $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ is valid, but that your best efforts won't succeed. How could you be sure that no such proof can be found? After all, it might just be that you can't find a proof even though there is one. It suffices to find a valuation in which ϕ_i evaluate to `T` whereas ψ evaluates to `F`. Then, by definition of $\vDash$, we don't have $\phi_1, \phi_2, \ldots, \phi_2 \vDash \psi$. Using soundness, this means that $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ cannot be valid. Therefore, this sequent does not have a proof. You will practice this method in the exercises.

1.4.4 Completeness of propositional logic

In this subsection, we hope to convince you that the natural deduction rules of propositional logic are *complete*: whenever $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, then there exists a natural deduction proof for the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$. Combined with the soundness result of the previous subsection, we then obtain

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi \text{ is valid} \quad \text{iff} \quad \phi_1, \phi_2, \ldots, \phi_n \vDash \psi \text{ holds.}$$

This gives you a certain freedom regarding which method you prefer to use. Often it is much easier to show one of these two relationships (although neither of the two is universally better, or easier, to establish). The first method involves a *proof search*, upon which the *logic programming* paradigm is based. The second method typically forces you to compute a truth table which is exponential in the size of occurring propositional atoms. Both methods are intractable in general but particular instances of formulas often respond differently to treatment under these two methods.

The remainder of this section is concerned with an argument saying that if $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, then $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid. Assuming that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, the argument proceeds in three steps:

Step 1: We show that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots (\phi_n \to \psi) \ldots)))$ holds.
Step 2: We show that $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots (\phi_n \to \psi) \ldots)))$ is valid.
Step 3: Finally, we show that $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid.

The first and third steps are quite easy; all the real work is done in the second one.

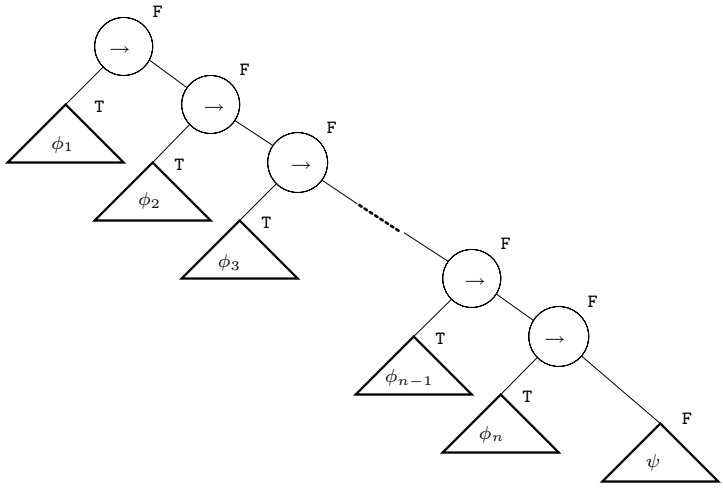


Figure 1.11. The only way this parse tree can evaluate to F. We represent parse trees for ϕ_1, ϕ_2, ..., ϕ_n as triangles as their internal structure does not concern us here.

Step 1:

Definition 1.36 A formula of propositional logic ϕ is called a *tautology* iff it evaluates to T under all its valuations, i.e. iff $\models \phi$.

Supposing that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds, let us verify that $\phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots(\phi_n \to \psi)\ldots)))$ is indeed a tautology. Since the latter formula is a nested implication, it can evaluate to F only if all $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to T *and* ψ evaluates to F; see its parse tree in Figure 1.11. But this contradicts the fact that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds. Thus, $\models \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots(\phi_n \to \psi)\ldots)))$ holds.

Step 2:

Theorem 1.37 *If $\models \eta$ holds, then $\vdash \eta$ is valid. In other words, if η is a tautology, then η is a theorem.*

This step is the hard one. Assume that $\models \eta$ holds. Given that η contains n distinct propositional atoms $p_1, p_2, \ldots, p_n$ we know that η evaluates to T for all 2^n lines in its truth table. (Each line lists a valuation of η.) How can we use this information to construct a proof for η? In some cases this can be done quite easily by taking a very good look at the concrete structure of η. But here we somehow have to come up with a *uniform* way of building such a proof. The key insight is to 'encode' each line in the truth table of η

as a sequent. Then we construct proofs for these 2^n sequents and assemble them into a proof of η.

Proposition 1.38 *Let ϕ be a formula such that $p_1, p_2, \ldots, p_n$ are its only propositional atoms. Let l be any line number in ϕ's truth table. For all $1 \leq i \leq n$ let $\hat{p}_i$ be p_i if the entry in line l of p_i is* `T`*, otherwise $\hat{p}_i$ is $\neg p_i$. Then we have*

1. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi$ is provable if the entry for ϕ in line l is* `T`
2. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi$ is provable if the entry for ϕ in line l is* `F`

Proof: This proof is done by structural induction on the formula ϕ, that is, mathematical induction on the height of the parse tree of ϕ.

1. If ϕ is a propositional atom p, we need to show that $p \vdash p$ and $\neg p \vdash \neg p$. These have one-line proofs.
2. If ϕ is of the form $\neg\phi_1$ we again have two cases to consider. First, assume that ϕ evaluates to `T`. In this case ϕ_1 evaluates to `F`. Note that ϕ_1 has the same atomic propositions as ϕ. We may use the induction hypothesis on ϕ_1 to conclude that $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi_1$; but $\neg\phi_1$ is just ϕ, so we are done.
   Second, if ϕ evaluates to `F`, then ϕ_1 evaluates to `T` and we get $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1$ by induction. Using the rule $\neg\neg$i, we may extend the proof of $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1$ to one for $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\neg\phi_1$; but $\neg\neg\phi_1$ is just $\neg\phi$, so again we are done.

The remaining cases all deal with two subformulas: ϕ equals $\phi_1 \circ \phi_2$, where $\circ$ is $\rightarrow$, $\wedge$ or $\vee$. In all these cases let $q_1, \ldots, q_l$ be the propositional atoms of ϕ_1 and $r_1, \ldots, r_k$ be the propositional atoms of ϕ_2. Then we certainly have $\{q_1, \ldots, q_l\} \cup \{r_1, \ldots, r_k\} = \{p_1, \ldots, p_n\}$. Therefore, whenever $\hat{q}_1, \ldots, \hat{q}_l \vdash \psi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \psi_2$ are valid so is $\hat{p}_1, \ldots, \hat{p}_n \vdash \psi_1 \wedge \psi_2$ using the rule $\wedge$i. In this way, we can use our induction hypothesis and only owe proofs that the conjunctions we conclude allow us to prove the desired conclusion for ϕ or $\neg\phi$ as the case may be.

3. To wit, let ϕ be $\phi_1 \rightarrow \phi_2$. If ϕ evaluates to `F`, then we know that ϕ_1 evaluates to `T` and ϕ_2 to `F`. Using our induction hypothesis, we have $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \neg\phi_2$, so $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ follows. We need to show $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg(\phi_1 \rightarrow \phi_2)$; but using $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$, this amounts to proving the sequent $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \rightarrow \phi_2)$, which we leave as an exercise.
   If ϕ evaluates to `T`, then we have three cases. First, if ϕ_1 evaluates to `F` and ϕ_2 to `F`, then we get, by our induction hypothesis, that $\hat{q}_1, \ldots, \hat{q}_l \vdash \neg\phi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \neg\phi_2$, so $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$ follows. Again, we need only to show the sequent $\neg\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \rightarrow \phi_2$, which we leave as an exercise. Second, if ϕ_1 evaluates to `F` and ϕ_2 to `T`, we use our induction hypothesis to arrive at

$\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ and have to prove $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$, which we leave as an exercise. Third, if ϕ_1 and ϕ_2 evaluate to `T`, we arrive at $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, using our induction hypothesis, and need to prove $\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$, which we leave as an exercise as well.

4. If ϕ is of the form $\phi_1 \wedge \phi_2$, we are again dealing with four cases in total. First, if ϕ_1 and ϕ_2 evaluate to `T`, we get $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \phi_2$ by our induction hypothesis, so $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$ follows. Second, if ϕ_1 evaluates to `F` and ϕ_2 to `T`, then we get $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ using our induction hypothesis and the rule $\wedge$i as above and we need to prove $\neg\phi_1 \wedge \phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise. Third, if ϕ_1 and ϕ_2 evaluate to `F`, then our induction hypothesis and the rule $\wedge$i let us infer that $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$; so we are left with proving $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise. Fourth, if ϕ_1 evaluates to `T` and ϕ_2 to `F`, we obtain $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ by our induction hypothesis and we have to show $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise.
5. Finally, if ϕ is a disjunction $\phi_1 \vee \phi_2$, we again have four cases. First, if ϕ_1 and ϕ_2 evaluate to `F`, then our induction hypothesis and the rule $\wedge$i give us $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$ and we have to show $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \vee \phi_2)$, which we leave as an exercise. Second, if ϕ_1 and ϕ_2 evaluate to `T`, then we obtain $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, by our induction hypothesis, and we need a proof for $\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. Third, if ϕ_1 evaluates to `F` and ϕ_2 to `T`, then we arrive at $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$, using our induction hypothesis, and need to establish $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. Fourth, if ϕ_1 evaluates to `T` and ϕ_2 to `F`, then $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ results from our induction hypothesis and all we need is a proof for $\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. □

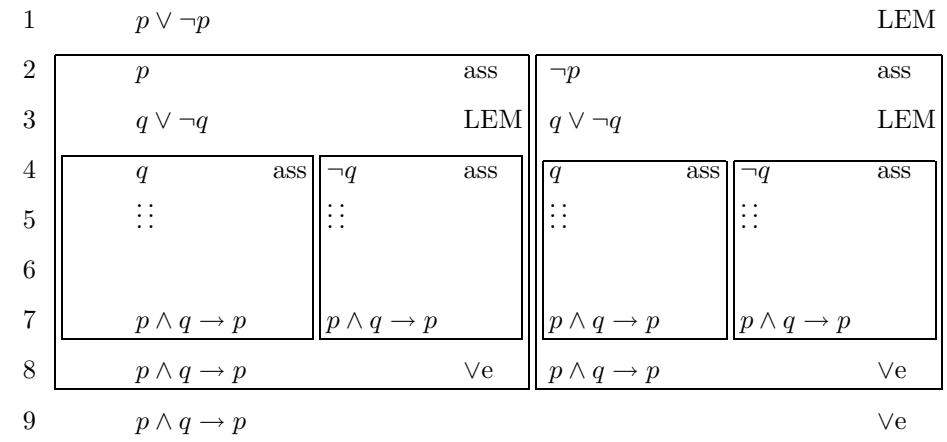
We apply this technique to the formula $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots(\phi_n \to \psi)\ldots)))$. Since it is a tautology it evaluates to `T` in all 2^n lines of its truth table; thus, the proposition above gives us 2^n many proofs of $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \eta$, one for each of the cases that $\hat{p}_i$ is p_i or $\neg p_i$. Our job now is to assemble all these proofs into a single proof for η which does not use any premises. We illustrate how to do this for an example, the tautology $p \wedge q \to p$.

The formula $p \wedge q \to p$ has two propositional atoms p and q. By the proposition above, we are guaranteed to have a proof for each of the four sequents

$$\begin{aligned} p, q &\vdash p \wedge q \to p \\ \neg p, q &\vdash p \wedge q \to p \\ p, \neg q &\vdash p \wedge q \to p \\ \neg p, \neg q &\vdash p \wedge q \to p. \end{aligned}$$

Ultimately, we want to prove $p \wedge q \to p$ by appealing to the four proofs of the sequents above. Thus, we somehow need to get rid of the premises on

the left-hand sides of these four sequents. This is the place where we rely on the law of the excluded middle which states $r \vee \neg r$, for any r. We use LEM for all propositional atoms (here p and q) and then we separately assume all the four cases, by using $\vee$e. That way we can invoke all four proofs of the sequents above and use the rule $\vee$e repeatedly until we have got rid of all our premises. We spell out the combination of these four phases schematically:



As soon as you understand how this particular example works, you will also realise that it will work for an arbitrary tautology with n distinct atoms. Of course, it seems ridiculous to prove $p \wedge q \to p$ using a proof that is this long. But remember that this illustrates a *uniform* method that constructs a proof for every tautology η, no matter how complicated it is.

Step 3: Finally, we need to find a proof for $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$. Take the proof for $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots (\phi_n \to \psi) \ldots)))$ given by step 2 and augment its proof by introducing $\phi_1, \phi_2, \ldots, \phi_n$ as premises. Then apply $\to$e n times on each of these premises (starting with ϕ_1, continuing with ϕ_2 etc.). Thus, we arrive at the conclusion ψ which gives us a proof for the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$.

Corollary 1.39 (Soundness and Completeness) *Let $\phi_1, \phi_2, \ldots, \phi_n, \psi$ be formulas of propositional logic. Then $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ is holds iff the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid.*

1.5 Normal forms

In the last section, we showed that our proof system for propositional logic is sound and complete for the truth-table semantics of formulas in Figure 1.6.

Soundness means that whatever we prove is going to be a true fact, based on the truth-table semantics. In the exercises, we apply this to show that a sequent does not have a proof: simply show that $\phi_1, \phi_2, \ldots, \phi_2$ does not semantically entail ψ; then soundness implies that the sequent $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ does not have a proof. Completeness comprised a much more powerful statement: no matter what (semantically) valid sequents there are, they all have syntactic proofs in the proof system of natural deduction. This tight correspondence allows us to freely switch between working with the notion of proofs ($\vdash$) and that of semantic entailment ($\models$).

Using natural deduction to decide the validity of instances of $\vdash$ is only one of many possibilities. In Exercise 1.2.6 we sketch a non-linear, tree-like, notion of proofs for sequents. Likewise, checking an instance of $\models$ by applying Definition 1.34 literally is only one of many ways of deciding whether $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds. We now investigate various alternatives for deciding $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ which are based on transforming these formulas syntactically into 'equivalent' ones upon which we can then settle the matter by purely syntactic or algorithmic means. This requires that we first clarify what exactly we mean by equivalent formulas.

1.5.1 Semantic equivalence, satisfiability and validity

Two formulas ϕ and ψ are said to be equivalent if they have the same 'meaning.' This suggestion is vague and needs to be refined. For example, $p \to q$ and $\neg p \vee q$ have the same truth table; all four combinations of `T` and `F` for p and q return the same result. 'Coincidence of truth tables' is not good enough for what we have in mind, for what about the formulas $p \wedge q \to p$ and $r \vee \neg r$? At first glance, they have little in common, having different atomic formulas and different connectives. Moreover, the truth table for $p \wedge q \to p$ is four lines long, whereas the one for $r \vee \neg r$ consists of only two lines. However, both formulas are always true. This suggests that we define the equivalence of formulas ϕ and ψ via $\models$: if ϕ semantically entails ψ and vice versa, then these formulas should be the same as far as our truth-table semantics is concerned.

Definition 1.40 Let ϕ and ψ be formulas of propositional logic. We say that ϕ and ψ are *semantically equivalent* iff $\phi \models \psi$ and $\psi \models \phi$ hold. In that case we write $\phi \equiv \psi$. Further, we call ϕ *valid* if $\models \phi$ holds.

Note that we could also have defined $\phi \equiv \psi$ to mean that $\models (\phi \to \psi) \wedge (\psi \to \phi)$ holds; it amounts to the same concept. Indeed, because of soundness and completeness, semantic equivalence is identical to *provable equivalence*

(Definition 1.25). Examples of equivalent formulas are

$$\begin{aligned} p \to q &\equiv \neg q \to \neg p \\ p \to q &\equiv \neg p \vee q \\ p \wedge q \to p &\equiv r \vee \neg r \\ p \wedge q \to r &\equiv p \to (q \to r). \end{aligned}$$

Recall that a formula η is called a tautology if $\vDash \eta$ holds, so the tautologies are exactly the valid formulas. The following lemma says that any decision procedure for tautologies is in fact a decision procedure for the validity of sequents as well.

Lemma 1.41 *Given formulas $\phi_1, \phi_2, \ldots, \phi_n$ and ψ of propositional logic, $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds iff $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ holds.*

Proof: First, suppose that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ holds. If $\phi_1, \phi_2, \ldots, \phi_n$ are all true under some valuation, then ψ has to be true as well for that same valuation. Otherwise, $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ would not hold (compare this with Figure 1.11). Second, if $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, we have already shown that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ follows in step 1 of our completeness proof. □

For our current purposes, we want to transform formulas into ones which don't contain $\to$ at all and the occurrences of $\wedge$ and $\vee$ are confined to separate layers such that validity checks are easy. This is being done by

1. using the equivalence $\phi \to \psi \equiv \neg\phi \vee \psi$ to remove all occurrences of $\to$ from a formula and
2. by specifying an algorithm that takes a formula without any $\to$ into a *normal form* (still without $\to$) for which checking validity is easy.

Naturally, we have to specify which forms of formulas we think of as being 'normal.' Again, there are many such notions, but in this text we study only two important ones.

Definition 1.42 A *literal* L is either an atom p or the negation of an atom $\neg p$. A formula C is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where each clause D is a disjunction of literals:

$$\begin{aligned} L &::= p \mid \neg p \\ D &::= L \mid L \vee D \\ C &::= D \mid D \wedge C. \end{aligned} \tag{1.6}$$

Examples of formulas in conjunctive normal form are

$$(i) \quad (\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q \qquad (ii) \quad (p \vee r) \wedge (\neg p \vee r) \wedge (p \vee \neg r).$$

In the first case, there are three clauses of type D: $\neg q \vee p \vee r$, $\neg p \vee r$, and q – which is a literal promoted to a clause by the first rule of clauses in (1.6). Notice how we made implicit use of the associativity laws for $\wedge$ and $\vee$, saying that $\phi \vee (\psi \vee \eta) \equiv (\phi \vee \psi) \vee \eta$ and $\phi \wedge (\psi \wedge \eta) \equiv (\phi \wedge \psi) \wedge \eta$, since we omitted some parentheses. The formula $(\neg(q \vee p) \vee r) \wedge (q \vee r)$ is not in CNF since $q \vee p$ is not a literal.

Why do we care at all about formulas ϕ in CNF? One of the reasons for their usefulness is that they allow easy checks of validity which otherwise take times exponential in the number of atoms. For example, consider the formula in CNF from above: $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$. The semantic entailment $\vDash (\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ holds iff all three relations

$$\vDash \neg q \vee p \vee r \qquad \vDash \neg p \vee r \qquad \vDash q$$

hold, by the semantics of $\wedge$. But since all of these formulas are disjunctions of literals, or literals, we can settle the matter as follows.

Lemma 1.43 *A disjunction of literals $L_1 \vee L_2 \vee \cdots \vee L_m$ is valid iff there are $1 \leq i, j \leq m$ such that L_i is $\neg L_j$.*

Proof: If L_i equals $\neg L_j$, then $L_1 \vee L_2 \vee \cdots \vee L_m$ evaluates to `T` for all valuations. For example, the disjunct $p \vee q \vee r \vee \neg q$ can never be made false.

To see that the converse holds as well, assume that no literal L_k has a matching negation in $L_1 \vee L_2 \vee \cdots \vee L_m$. Then, for each k with $1 \leq k \leq n$, we assign `F` to L_k, if L_k is an atom; or `T`, if L_k is the negation of an atom. For example, the disjunct $\neg q \vee p \vee r$ can be made false by assigning `F` to p and r and `T` to q. □

Hence, we have an easy and fast check for the validity of $\vDash \phi$, provided that ϕ is in CNF; inspect all conjuncts ψ_k of ϕ and search for atoms in ψ_k such that ψ_k also contains their negation. If such a match is found for all conjuncts, we have $\vDash \phi$. Otherwise (= some conjunct contains no pair L_i and $\neg L_i$), ϕ is not valid by the lemma above. Thus, the formula $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ above is not valid. Note that the matching literal has to be found in the same conjunct ψ_k. Since there is no free lunch in this universe, we can expect that the computation of a formula ϕ' in CNF, which is equivalent to a given formula ϕ, is a costly worst-case operation.

Before we study how to compute equivalent conjunctive normal forms, we introduce another semantic concept closely related to that of validity.

Definition 1.44 Given a formula ϕ in propositional logic, we say that ϕ is *satisfiable* if it has a valuation in which is evaluates to `T`.

For example, the formula $p \vee q \rightarrow p$ is satisfiable since it computes `T` if we assign `T` to p. Clearly, $p \vee q \rightarrow p$ is not valid. Thus, satisfiability is a weaker concept since every valid formula is by definition also satisfiable but not vice versa. However, these two notions are just mirror images of each other, the mirror being negation.

Proposition 1.45 *Let ϕ be a formula of propositional logic. Then ϕ is satisfiable iff $\neg\phi$ is not valid.*

Proof: First, assume that ϕ is satisfiable. By definition, there exists a valuation of ϕ in which ϕ evaluates to `T`; but that means that $\neg\phi$ evaluates to `F` for that same valuation. Thus, $\neg\phi$ cannot be valid.

Second, assume that $\neg\phi$ is not valid. Then there must be a valuation of $\neg\phi$ in which $\neg\phi$ evaluates to `F`. Thus, ϕ evaluates to `T` and is therefore satisfiable. (Note that the valuations of ϕ are exactly the valuations of $\neg\phi$.) □

This result is extremely useful since it essentially says that we need provide a decision procedure for only one of these concepts. For example, let's say that we have a procedure `P` for deciding whether any ϕ is valid. We obtain a decision procedure for satisfiability simply by asking `P` whether $\neg\phi$ is valid. If it is, ϕ is not satisfiable; otherwise ϕ is satisfiable. Similarly, we may transform any decision procedure for satisfiability into one for validity. We will encounter both kinds of procedures in this text.

There is one scenario in which computing an equivalent formula in CNF is really easy; namely, when someone else has already done the work of writing down a full truth table for ϕ. For example, take the truth table of $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$ in Figure 1.8 (page 40). For each line where $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$ computes `F` we now construct a disjunction of literals. Since there is only one such line, we have only one conjunct ψ_1. That conjunct is now obtained by a disjunction of literals, where we include literals $\neg p$ and q. Note that the literals are just the syntactic opposites of the truth values in that line: here p is `T` and q is `F`. The resulting formula in CNF is thus $\neg p \vee q$ which is readily seen to be in CNF and to be equivalent to $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$.

Why does this always work for any formula ϕ? Well, the constructed formula will be false iff at least one of its conjuncts ψ_i will be false. This means that all the disjuncts in such a ψ_i must be `F`. Using the de Morgan

rule $\neg\phi_1 \vee \neg\phi_2 \vee \cdots \vee \neg\phi_n \equiv \neg(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$, we infer that the conjunction of the syntactic opposites of those literals must be true. Thus, ϕ and the constructed formula have the same truth table.

Consider another example, in which ϕ is given by the truth table:

p	q	r	ϕ
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	T

Note that this table is really just a specification of ϕ; it does not tell us what ϕ looks like syntactically, but it does tells us how it ought to 'behave.' Since this truth table has four entries which compute F, we construct four conjuncts ψ_i $(1 \leq i \leq 4)$. We read the ψ_i off that table by listing the disjunction of all atoms, where we negate those atoms which are true in those lines:

$$\psi_1 \stackrel{\text{def}}{=} \neg p \vee \neg q \vee r \quad \text{(line 2)} \qquad \psi_2 \stackrel{\text{def}}{=} p \vee \neg q \vee \neg r \quad \text{(line 5)}$$
$$\psi_3 \stackrel{\text{def}}{=} p \vee \neg q \vee r \quad \text{etc} \qquad \psi_4 \stackrel{\text{def}}{=} p \vee q \vee \neg r.$$

The resulting ϕ in CNF is therefore

$$(\neg p \vee \neg q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge (p \vee \neg q \vee r) \wedge (p \vee q \vee \neg r).$$

If we don't have a full truth table at our disposal, but do know the structure of ϕ, then we would like to compute a version of ϕ in CNF. It should be clear by now that a full truth table of ϕ and an equivalent formula in CNF are pretty much the same thing as far as questions about validity are concerned – although the formula in CNF may be much more compact.

1.5.2 Conjunctive normal forms and validity

We have already seen the benefits of conjunctive normal forms in that they allow for a fast and easy syntactic test of validity. Therefore, one wonders whether any formula can be transformed into an *equivalent* formula in CNF. We now develop an algorithm achieving just that. Note that, by Definition 1.40, a formula is valid iff any of its equivalent formulas is valid. We reduce the problem of determining whether *any* ϕ is valid to the problem of computing an equivalent $\psi \equiv \phi$ such that ψ is in CNF and checking, via Lemma 1.43, whether ψ is valid.

Before we sketch such a procedure, we make some general remarks about its possibilities and its realisability constraints. First of all, there could be more or less efficient ways of computing such normal forms. But even more so, there could be many possible correct outputs, for $\psi_1 \equiv \phi$ and $\psi_2 \equiv \phi$ do not generally imply that ψ_1 is the same as ψ_2, even if ψ_1 and ψ_2 are in CNF. For example, take $\phi \stackrel{\text{def}}{=} p$, $\psi_1 \stackrel{\text{def}}{=} p$ and $\psi_2 \stackrel{\text{def}}{=} p \wedge (p \vee q)$; then convince yourself that $\phi \equiv \psi_2$ holds. Having this ambiguity of equivalent conjunctive normal forms, the computation of a CNF for ϕ with minimal 'cost' (where 'cost' could for example be the number of conjuncts, or the height of ϕ's parse tree) becomes a very important practical problem, an issue persued in Chapter 6. Right now, we are content with stating a *deterministic* algorithm which always computes the same output CNF for a given input ϕ.

This algorithm, called `CNF`, should satisfy the following requirements:

(1) `CNF` terminates for all formulas of propositional logic as input;
(2) for each such input, `CNF` outputs an equivalent formula; and
(3) all output computed by `CNF` is in CNF.

If a call of `CNF` with a formula ϕ of propositional logic as input terminates, which is enforced by (1), then (2) ensures that $\psi \equiv \phi$ holds for the output ψ. Thus, (3) guarantees that ψ is an equivalent CNF of ϕ. So ϕ is valid iff ψ is valid; and checking the latter is easy relative to the length of ψ.

What kind of strategy should `CNF` employ? It will have to function correctly for all, i.e. infinitely many, formulas of propositional logic. This strongly suggests to write a procedure that computes a CNF by structural induction on the formula ϕ. For example, if ϕ is of the form $\phi_1 \wedge \phi_2$, we may simply compute conjunctive normal forms η_i for ϕ_i $(i = 1, 2)$, whereupon $\eta_1 \wedge \eta_2$ is a conjunctive normal form which is equivalent to ϕ *provided that* $\eta_i \equiv \phi_i$ $(i = 1, 2)$. This strategy also suggests to use proof by structural induction on ϕ to prove that `CNF` meets the requirements (1–3) stated above.

Given a formula ϕ as input, we first do some *preprocessing*. Initially, we translate away all implications in ϕ by replacing all subformulas of the form $\psi \to \eta$ by $\neg\psi \vee \eta$. This is done by a procedure called `IMPL_FREE`. Note that this procedure has to be recursive, for there might be implications in ψ or η as well.

The application of `IMPL_FREE` might introduce double negations into the output formula. More importantly, negations whose scopes are non-atomic formulas might still be present. For example, the formula $p \wedge \neg(p \wedge q)$ has such a negation with $p \wedge q$ as its scope. Essentially, the question is whether one can efficiently compute a CNF for $\neg\phi$ from a CNF for ϕ. Since *nobody* seems to know the answer, we circumvent the question by translating $\neg\phi$

into an equivalent formula that contains only negations of atoms. Formulas which only negate atoms are said to be in *negation normal form* (NNF). We spell out such a procedure, `NNF`, in detail later on. The key to its specification for implication-free formulas lies in the de Morgan rules. The second phase of the preprocessing, therefore, calls `NNF` with the implication-free output of `IMPL_FREE` to obtain an equivalent formula in NNF.

After all this preprocessing, we obtain a formula ϕ' which is the result of the call $\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi))$. Note that $\phi' \equiv \phi$ since both algorithms only transform formulas into equivalent ones. Since ϕ' contains no occurrences of $\rightarrow$ and since only atoms in ϕ' are negated, we may program `CNF` by an analysis of only *three* cases: literals, conjunctions and disjunctions.

- If ϕ is a literal, it is by definition in CNF and so `CNF` outputs ϕ.
- If ϕ equals $\phi_1 \wedge \phi_2$, we call `CNF` recursively on each ϕ_i to get the respective output η_i and return the CNF $\eta_1 \wedge \eta_2$ as output for input ϕ.
- If ϕ equals $\phi_1 \vee \phi_2$, we again call `CNF` recursively on each ϕ_i to get the respective output η_i; but this time we must not simply return $\eta_1 \vee \eta_2$ since that formula is certainly *not* in CNF, unless η_1 and η_2 happen to be literals.

So how can we complete the program in the last case? Well, we may resort to the distributivity laws, which entitle us to translate any disjunction of conjunctions into a conjunction of disjunctions. However, for this to result in a CNF, we need to make certain that those disjunctions generated contain only literals. We apply a strategy for using distributivity based on matching patterns in $\phi_1 \vee \phi_2$. This results in an independent algorithm called `DISTR` which will do all that work for us. Thus, we simply call `DISTR` with the pair (η_1, η_2) as input and pass along its result.

Assuming that we already have written code for `IMPL_FREE`, `NNF` and `DISTR`, we may now write pseudo code for `CNF`:

function `CNF` (ϕ):
/* precondition: ϕ implication free and in NNF */
/* postcondition: `CNF` (ϕ) computes an equivalent CNF for ϕ */
begin function
  case
    ϕ is a literal: **return** ϕ
    ϕ is $\phi_1 \wedge \phi_2$: **return** $\texttt{CNF}\,(\phi_1) \wedge \texttt{CNF}\,(\phi_2)$
    ϕ is $\phi_1 \vee \phi_2$: **return** $\texttt{DISTR}\,(\texttt{CNF}\,(\phi_1), \texttt{CNF}\,(\phi_2))$
  end case
end function

Notice how the calling of DISTR is done with the computed conjunctive normal forms of ϕ_1 and ϕ_2. The routine DISTR has η_1 and η_2 as input parameters and does a case analysis on whether these inputs are conjunctions. What should DISTR do if none of its input formulas is such a conjunction? Well, since we are calling DISTR for inputs η_1 and η_2 which are in CNF, this can only mean that η_1 and η_2 are literals, or disjunctions of literals. Thus, $\eta_1 \vee \eta_2$ is in CNF.

Otherwise, at least one of the formulas η_1 and η_2 is a conjunction. Since one conjunction suffices for simplifying the problem, we have to decide which conjunct we want to transform if *both* formulas are conjunctions. That way we maintain that our algorithm CNF is deterministic. So let us suppose that η_1 is of the form $\eta_{11} \wedge \eta_{12}$. Then the distributive law says that $\eta_1 \vee \eta_2 \equiv (\eta_{11} \vee \eta_2) \wedge (\eta_{12} \vee \eta_2)$. Since all participating formulas η_{11}, η_{12} and η_2 are in CNF, we may call DISTR again for the pairs (η_{11}, η_2) and (η_{12}, η_2), and then simply form their conjunction. This is the key insight for writing the function DISTR.

The case when η_2 is a conjunction is symmetric and the structure of the recursive call of DISTR is then dictated by the equivalence $\eta_1 \vee \eta_2 \equiv (\eta_1 \vee \eta_{21}) \wedge (\eta_1 \vee \eta_{22})$, where $\eta_2 = \eta_{21} \wedge \eta_{22}$:

```
function DISTR(η1, η2):
/* precondition: η1 and η2 are in CNF */
/* postcondition: DISTR(η1, η2) computes a CNF for η1 ∨ η2 */
begin function
  case
    η1 is η11 ∧ η12: return DISTR(η11, η2) ∧ DISTR(η12, η2)
    η2 is η21 ∧ η22: return DISTR(η1, η21) ∧ DISTR(η1, η22)
    otherwise (= no conjunctions): return η1 ∨ η2
  end case
end function
```

Notice how the three clauses are exhausting all possibilities. Furthermore, the first and second cases overlap if η_1 and η_2 are both conjunctions. It is then our understanding that this code will inspect the clauses of a case statement from the top to the bottom clause. Thus, the first clause would apply.

Having specified the routines CNF and DISTR, this leaves us with the task of writing the functions IMPL_FREE and NNF. We delegate the design

of IMPL_FREE to the exercises. The function NNF has to transform any implication-free formula into an equivalent one in negation normal form. Four examples of formulas in NNF are

$$\begin{array}{ll} p & \neg p \\ \neg p \wedge (p \wedge q) & \neg p \wedge (p \rightarrow q), \end{array}$$

although we won't have to deal with a formula of the last kind since $\rightarrow$ won't occur. Examples of formulas which are not in NNF are $\neg\neg p$ and $\neg(p \wedge q)$.

Again, we program NNF recursively by a case analysis over the structure of the input formula ϕ. The last two examples already suggest a solution for two of these clauses. In order to compute a NNF of $\neg\neg\phi$, we simply compute a NNF of ϕ. This is a sound strategy since ϕ and $\neg\neg\phi$ are semantically equivalent. If ϕ equals $\neg(\phi_1 \wedge \phi_2)$, we use the de Morgan rule $\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \vee \neg\phi_2$ as a recipe for how NNF should call itself recursively in that case. Dually, the case of ϕ being $\neg(\phi_1 \vee \phi_2)$ appeals to the other de Morgan rule $\neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$ and, if ϕ is a conjunction or disjunction, we simply let NNF pass control to those subformulas. Clearly, all literals are in NNF. The resulting code for NNF is thus

function NNF (ϕ):
/* precondition: ϕ is implication free */
/* postcondition: NNF (ϕ) computes a NNF for ϕ */
begin function
  case
    ϕ is a literal: **return** ϕ
    ϕ is $\neg\neg\phi_1$: **return** NNF (ϕ_1)
    ϕ is $\phi_1 \wedge \phi_2$: **return** NNF $(\phi_1) \wedge$ NNF (ϕ_2)
    ϕ is $\phi_1 \vee \phi_2$: **return** NNF $(\phi_1) \vee$ NNF (ϕ_2)
    ϕ is $\neg(\phi_1 \wedge \phi_2)$: **return** NNF $(\neg\phi_1) \vee$ NNF $(\neg\phi_2)$
    ϕ is $\neg(\phi_1 \vee \phi_2)$: **return** NNF $(\neg\phi_1) \wedge$ NNF $(\neg\phi_2)$
  end case
end function

Notice that these cases are exhaustive due to the algorithm's precondition. Given any formula ϕ of propositional logic, we may now convert it into an

equivalent CNF by calling $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi)))$. In the exercises, you are asked to show that

- all four algorithms terminate on input meeting their preconditions,
- the result of $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi)))$ is in CNF and
- that result is semantically equivalent to ϕ.

We will return to the important issue of formally proving the correctness of programs in Chapter 4.

Let us now illustrate the programs coded above on some concrete examples. We begin by computing $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\neg p \wedge q \to p \wedge (r \to q))))$. We show almost all details of this computation and you should compare this with how you would expect the code above to behave. First, we compute $\texttt{IMPL\_FREE}\ (\phi)$:

$$\begin{aligned}
\texttt{IMPL_FREE}\,(\phi) &= \neg\texttt{IMPL_FREE}\,(\neg p \wedge q) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg((\texttt{IMPL_FREE}\,\neg p) \wedge (\texttt{IMPL_FREE}\,q)) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg((\neg p) \wedge \texttt{IMPL_FREE}\,q) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg(\neg p \wedge q) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg(\neg p \wedge q) \vee ((\texttt{IMPL_FREE}\,p) \wedge \texttt{IMPL_FREE}\,(r \to q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge \texttt{IMPL_FREE}\,(r \to q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg(\texttt{IMPL_FREE}\,r) \vee (\texttt{IMPL_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee (\texttt{IMPL_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}$$

Second, we compute $\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi)$:

$$\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL_FREE}\ \phi) &= \texttt{NNF}\ (\neg(\neg p \wedge q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= \texttt{NNF}\ (\neg(\neg p) \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (\texttt{NNF}\ (\neg\neg p)) \vee (\texttt{NNF}\ (\neg q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee (\texttt{NNF}\ (\neg q))) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee ((\texttt{NNF}\ p) \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge ((\texttt{NNF}\ (\neg r)) \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}$$

Third, we finish it off with

$$\begin{aligned}
\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL_FREE}\,\phi)) &= \texttt{CNF}\,((p \vee \neg q) \vee (p \wedge (\neg r \vee q))) \\
&= \texttt{DISTR}\,(\texttt{CNF}\,(p \vee \neg q), \texttt{CNF}\,(p \wedge (\neg r \vee q))) \\
&= \texttt{DISTR}\,(p \vee \neg q, \texttt{CNF}\,(p \wedge (\neg r \vee q))) \\
&= \texttt{DISTR}\,(p \vee \neg q, p \wedge (\neg r \vee q)) \\
&= \texttt{DISTR}\,(p \vee \neg q, p) \wedge \texttt{DISTR}\,(p \vee \neg q, \neg r \vee q) \\
&= (p \vee \neg q \vee p) \wedge \texttt{DISTR}\,(p \vee \neg q, \neg r \vee q) \\
&= (p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q)\ .
\end{aligned}$$

The formula $(p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q)$ is thus the result of the call $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,\phi))$ and is in conjunctive normal form and equivalent to ϕ. Note that it is satisfiable (choose p to be true) but not valid (choose p to be false and q to be true); it is also equivalent to the simpler conjunctive normal form $p \vee \neg q$. Observe that our algorithm does not do such optimisations so one would need a separate optimiser running on the output. Alternatively, one might change the code of our functions to allow for such optimisations 'on the fly,' a computational overhead which could prove to be counter-productive.

You should realise that we omitted several computation steps in the sub-calls $\texttt{CNF}\,(p \vee \neg q)$ and $\texttt{CNF}\,(p \wedge (\neg r \vee q))$. They return their input as a result since the input is already in conjunctive normal form.

As a second example, consider $\phi \stackrel{\text{def}}{=} r \to (s \to (t \wedge s \to r))$. We compute

$$\begin{aligned}
\texttt{IMPL_FREE}\,(\phi) &= \neg(\texttt{IMPL_FREE}\ r) \vee \texttt{IMPL_FREE}\,(s \to (t \wedge s \to r)) \\
&= \neg r \vee \texttt{IMPL_FREE}\,(s \to (t \wedge s \to r)) \\
&= \neg r \vee (\neg(\texttt{IMPL_FREE}\,s) \vee \texttt{IMPL_FREE}\,(t \wedge s \to r)) \\
&= \neg r \vee (\neg s \vee \texttt{IMPL_FREE}\,(t \wedge s \to r)) \\
&= \neg r \vee (\neg s \vee (\neg(\texttt{IMPL_FREE}\,(t \wedge s)) \vee \texttt{IMPL_FREE}\,r)) \\
&= \neg r \vee (\neg s \vee (\neg((\texttt{IMPL_FREE}\,t) \wedge (\texttt{IMPL_FREE}\,s)) \vee \texttt{IMPL_FREE}\,r)) \\
&= \neg r \vee (\neg s \vee (\neg(t \wedge (\texttt{IMPL_FREE}\,s)) \vee (\texttt{IMPL_FREE}\,r))) \\
&= \neg r \vee (\neg s \vee (\neg(t \wedge s)) \vee (\texttt{IMPL_FREE}\,r)) \\
&= \neg r \vee (\neg s \vee (\neg(t \wedge s)) \vee r)
\end{aligned}$$

$$
\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi) &= \texttt{NNF}\ (\neg r \vee (\neg s \vee \neg(t \wedge s) \vee r)) \\
&= (\texttt{NNF}\ \neg r) \vee \texttt{NNF}\ (\neg s \vee \neg(t \wedge s) \vee r) \\
&= \neg r \vee \texttt{NNF}\ (\neg s \vee \neg(t \wedge s) \vee r) \\
&= \neg r \vee (\texttt{NNF}\ (\neg s) \vee \texttt{NNF}\ (\neg(t \wedge s) \vee r)) \\
&= \neg r \vee (\neg s \vee \texttt{NNF}\ (\neg(t \wedge s) \vee r)) \\
&= \neg r \vee (\neg s \vee (\texttt{NNF}\ (\neg(t \wedge s)) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee (\texttt{NNF}\ (\neg t \vee \neg s)) \vee \texttt{NNF}\ r) \\
&= \neg r \vee (\neg s \vee ((\texttt{NNF}\ (\neg t) \vee \texttt{NNF}\ (\neg s)) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee ((\neg t \vee \texttt{NNF}\ (\neg s)) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee ((\neg t \vee \neg s) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee ((\neg t \vee \neg s) \vee r))
\end{aligned}
$$

where the latter is already in CNF and valid as r has a matching $\neg r$.

1.5.3 Horn clauses and satisfiability

We have already commented on the computational price we pay for transforming a propositional logic formula into an equivalent CNF. The latter class of formulas has an easy syntactic check for validity, but its test for satisfiability is very hard in general. Fortunately, there are practically important subclasses of formulas which have much more efficient ways of deciding their satisfiability. One such example is the class of *Horn formulas*; the name 'Horn' is derived from the logician A. Horn's last name. We shortly define them and give an algorithm for checking their satisfiability.

Recall that the logical constants $\bot$ ('bottom') and $\top$ ('top') denote an unsatisfiable formula, respectively, a tautology.

Definition 1.46 A *Horn formula* is a formula ϕ of propositional logic if it can be generated as an instance of H in this grammar:

$$
\begin{aligned}
P &::= \bot \mid \top \mid p \\
A &::= P \mid P \wedge A \\
C &::= A \rightarrow P \\
H &::= C \mid C \wedge H.
\end{aligned}
\tag{1.7}
$$

We call each instance of C a *Horn clause*.

Horn formulas are conjunctions of Horn clauses. A Horn clause is an implication whose assumption A is a conjunction of propositions of type P and whose conclusion is also of type P. Examples of Horn formulas are

$$(p \wedge q \wedge s \to p) \wedge (q \wedge r \to p) \wedge (p \wedge s \to s)$$
$$(p \wedge q \wedge s \to \bot) \wedge (q \wedge r \to p) \wedge (\top \to s)$$
$$(p_2 \wedge p_3 \wedge p_5 \to p_{13}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \to \bot).$$

Examples of formulas which are *not* Horn formulas are

$$(p \wedge q \wedge s \to \neg p) \wedge (q \wedge r \to p) \wedge (p \wedge s \to s)$$
$$(p \wedge q \wedge s \to \bot) \wedge (\neg q \wedge r \to p) \wedge (\top \to s)$$
$$(p_2 \wedge p_3 \wedge p_5 \to p_{13} \wedge p_{27}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \to \bot)$$
$$(p_2 \wedge p_3 \wedge p_5 \to p_{13} \wedge p_{27}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \vee \bot).$$

The first formula is not a Horn formula since $\neg p$, the conclusion of the implication of the first conjunct, is not of type P. The second formula does not qualify since the premise of the implication of the second conjunct, $\neg q \wedge r$, is not a conjunction of atoms, $\bot$, or $\top$. The third formula is not a Horn formula since the conclusion of the implication of the first conjunct, $p_{13} \wedge p_{27}$, is not of type P. The fourth formula clearly is not a Horn formula since it is not a conjunction of implications.

The algorithm we propose for deciding the satisfiability of a Horn formula ϕ maintains a list of all occurrences of type P in ϕ and proceeds like this:

1. It marks $\top$ if it occurs in that list.
2. If there is a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ such that all P_j with $1 \leq j \leq k_i$ are marked, mark P' as well and go to 2. Otherwise (= there is no conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ such that all P_j are marked) go to 3.
3. If $\bot$ is marked, print out 'The Horn formula ϕ is unsatisfiable.' and stop. Otherwise, go to 4.
4. Print out 'The Horn formula ϕ is satisfiable.' and stop.

In these instructions, the markings of formulas are *shared* by all other occurrences of these formulas in the Horn formula. For example, once we mark p_2 because of one of the criteria above, then all other occurrences of p_2 are marked as well. We use pseudo code to specify this algorithm formally:

function HORN(ϕ):
/* precondition: ϕ is a Horn formula */
/* postcondition: HORN(ϕ) decides the satisfiability for ϕ */
begin function
    mark all occurrences of $\top$ in ϕ;
    while there is a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ
        such that all P_j are marked but P' isn't **do**
        mark P'
    end while
    if $\bot$ is marked **then return** 'unsatisfiable' **else return** 'satisfiable'
end function

We need to make sure that this algorithm terminates on all Horn formulas ϕ as input and that its output (= its decision) is always correct.

Theorem 1.47 *The algorithm* HORN *is correct for the satisfiability decision problem of Horn formulas and has no more than $n+1$ cycles in its while-statement if n is the number of atoms in ϕ. In particular,* HORN *always terminates on correct input.*

Proof: Let us first consider the question of program termination. Notice that entering the body of the while-statement has the effect of marking an unmarked P which is not $\top$. Since this marking applies to all occurrences of P in ϕ, the while-statement can have at most one more cycle than there are atoms in ϕ.

Since we guaranteed termination, it suffices to show that the answers given by the algorithm HORN are always correct. To that end, it helps to reveal the functional role of those markings. Essentially, marking a P means that that P has got to be true if the formula ϕ is ever going to be satisfiable. We use mathematical induction to show that

'All marked P are true for all valuations in which ϕ evaluates to T.' (1.8)

holds after any number of executions of the body of the while-statement above. The base case, zero executions, is when the while-statement has not yet been entered but we already and only marked all occurrences of $\top$. Since $\top$ must be true in all valuations, (1.8) follows.

In the inductive step, we assume that (1.8) holds after k cycles of the while-statement. Then we need to show that same assertion for all marked P after $k+1$ cycles. If we enter the $(k+1)$th cycle, the condition of the while-statement is certainly true. Thus, there exists a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ such that all P_j are marked. Let v be any valuation

in which ϕ is true. By our induction hypothesis, we know that all P_j and therefore $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i}$ have to be true in v as well. The conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow P'$ of ϕ has be to true in v, too, from which we infer that P' has to be true in v.

By mathematical induction, we therefore secured that (1.8) holds no matter how many cycles that while-statement went through.

Finally, we need to make sure that the if-statement above always renders correct replies. First, if $\bot$ is marked, then there has to be some conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow \bot$ of ϕ such that all P_i are marked as well. By (1.8) that conjunct of ϕ evaluates to $\mathtt{T} \rightarrow \mathtt{F} = \mathtt{F}$ whenever ϕ is true. As this is impossible the reply 'unsatisfiable' is correct. Second, if $\bot$ is not marked, we simply assign $\mathtt{T}$ to all marked atoms and $\mathtt{F}$ to all unmarked atoms and use proof by contradiction to show that ϕ has to be true with respect to that valuation.

If ϕ is *not* true under that valuation, it must make one of its principal conjuncts $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow P'$ false. By the semantics of implication this can only mean that all P_j are true and P' is false. By the definition of our valuation, we then infer that all P_j are marked, so $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow P'$ is a conjunct of ϕ that would have been dealt with in one of the cycles of the while-statement and so P' is marked, too. Since $\bot$ is not marked, P' has to be $\top$ or some atom q. In any event, the conjunct is then true by (1.8), a contradiction □

Note that the proof by contradiction employed in the last proof was not really needed. It just made the argument seem more natural to us. The literature is full of such examples where one uses proof by contradiction more out of psychological than proof-theoretical necessity.

1.6 SAT solvers

The marking algorithm for Horn formulas computes marks as constraints on all valuations that can make a formule true. By (1.8), all marked atoms have to be true for any such valuation. We can extend this idea to general formulas ϕ by computing constraints saying which subformulas of ϕ require a certain truth value for all valuations that make ϕ true:

'All marked subformulas evaluate to their mark value
for all valuations in which ϕ evaluates to $\mathtt{T}$.' (1.9)

In that way, marking atomic formulas generalizes to marking subformulas; and 'true' marks generalize into 'true' and 'false' marks. At the same

time, (1.9) serves as a guide for designing an algorithm and as an invariant for proving its correctness.

1.6.1 A linear solver

We will execute this marking algorithm on the parse tree of formulas, except that we will translate formulas into the adequate fragment

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \tag{1.10}$$

and then share common subformulas of the resulting parse tree, making the tree into a directed, acyclic graph (DAG). The inductively defined translation

$$\begin{array}{ll} T(p) = p & T(\neg\phi) = \neg T(\phi) \\ T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2) & T(\phi_1 \vee \phi_2) = \neg(\neg T(\phi_1) \wedge \neg T(\phi_2)) \\ T(\phi_1 \rightarrow \phi_2) = \neg(T(\phi_1) \wedge \neg T(\phi_2)) & \end{array}$$

transforms formulas generated by (1.3) into formulas generated by (1.10) such that ϕ and $T(\phi)$ are semantically equivalent and have the same propositional atoms. Therefore, ϕ is satisfiable iff $T(\phi)$ is satisfiable; and the set of valuations for which ϕ is true equals the set of valuations for which $T(\phi)$ is true. The latter ensures that the diagnostics of a SAT solver, applied to $T(\phi)$, is meaningful for the original formula ϕ. In the exercises, you are asked to prove these claims.

Example 1.48 For the formula ϕ being $p \wedge \neg(q \vee \neg p)$ we compute $T(\phi) = p \wedge \neg\neg(\neg q \wedge \neg\neg p)$. The parse tree and DAG of $T(\phi)$ are depicted in Figure 1.12.

Any valuation that makes $p \wedge \neg\neg(\neg q \wedge \neg\neg p)$ true has to assign `T` to the topmost $\wedge$-node in its DAG of Figure 1.12. But that forces the mark `T` on the p-node and the topmost $\neg$-node. In the same manner, we arrive at a complete set of constraints in Figure 1.13, where the time stamps '1:' etc indicate the order in which we applied our intuitive reasoning about these constraints; this order is generally not unique.

The formal set of rules for forcing new constraints from old ones is depicted in Figure 1.14. A small circle indicates any node ($\neg$, $\wedge$ or atom). The force laws for negation, $\neg_{\mathrm{t}}$ and $\neg_{\mathrm{f}}$, indicate that a truth constraint on a $\neg$-node forces its dual value at its sub-node and vice versa. The law $\wedge_{\mathrm{te}}$ propagates a `T` constraint on a $\wedge$-node to its two sub-nodes; dually, $\wedge_{\mathrm{ti}}$ forces a `T` mark on a $\wedge$-node if both its children have that mark. The laws $\wedge_{\mathrm{fl}}$ and $\wedge_{\mathrm{fr}}$ force a `F` constraint on a $\wedge$-node if any of its sub-nodes has a `F` value. The laws $\wedge_{\mathrm{fll}}$

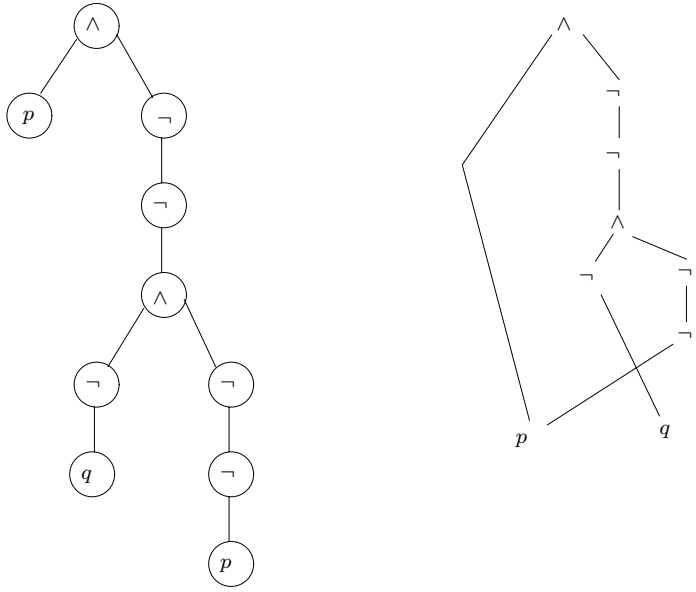


Figure 1.12. Parse tree (left) and directed acyclic graph (right) of the formula from Example 1.48. The p-node is shared on the right.

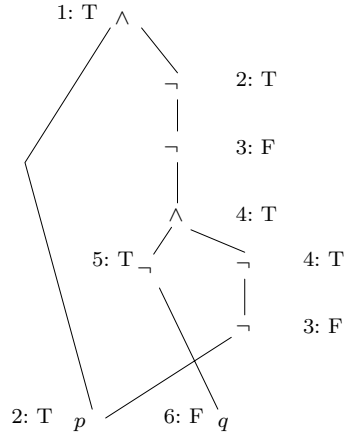


Figure 1.13. A witness to the satisfiability of the formula represented by this DAG.

and $\wedge_{\mathrm{frr}}$ are more complex: if an $\wedge$-node has a `F` constraint and one of its sub-nodes has a `T` constraint, then the *other* sub-node obtains a `F`-constraint. Please check that all constraints depicted in Figure 1.13 are derivable from these rules. The fact that each node in a DAG obtained a forced marking does not yet show that this is a witness to the satisfiability of the formula

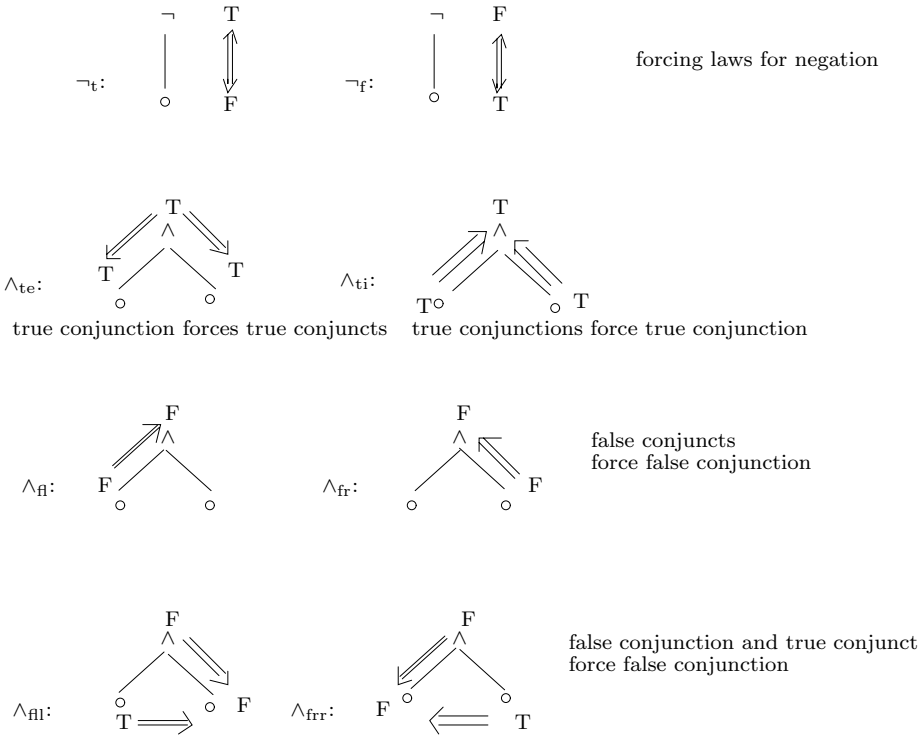


Figure 1.14. Rules for flow of constraints in a formula's DAG. Small circles indicate arbitrary nodes ($\neg$, $\wedge$ or atom). Note that the rules $\wedge_{\mathrm{fll}}$, $\wedge_{\mathrm{frr}}$ and $\wedge_{\mathrm{ti}}$ require that the source constraints of both $\Longrightarrow$ are present.

represented by this DAG. A post-processing phase takes the marks for all atoms and re-computes marks of all other nodes in a bottom-up manner, as done in Section 1.4 on parse trees. Only if the resulting marks match the ones we computed have we found a witness. Please verify that this is the case in Figure 1.13.

We can apply SAT solvers to checking whether sequents are valid. For example, the sequent $p \wedge q \to r \vdash p \to q \to r$ is valid iff $(p \wedge q \to r) \to p \to q \to r$ is a theorem (why?) iff $\phi = \neg((p \wedge q \to r) \to p \to q \to r)$ is *not* satisfiable. The DAG of $T(\phi)$ is depicted in Figure 1.15. The annotations "1" etc indicate which nodes represent which sub-formulas. Notice that such DAGs may be constructed by applying the translation clauses for T to sub-formulas in a bottom-up manner – sharing equal subgraphs were applicable.

The findings of our SAT solver can be seen in Figure 1.16. The solver concludes that the indicated node requires the marks `T` *and* `F` for (1.9) to be met. Such contradictory constraints therefore imply that all formulas $T(\phi)$ whose DAG equals that of this figure are not satisfiable. In particular, all

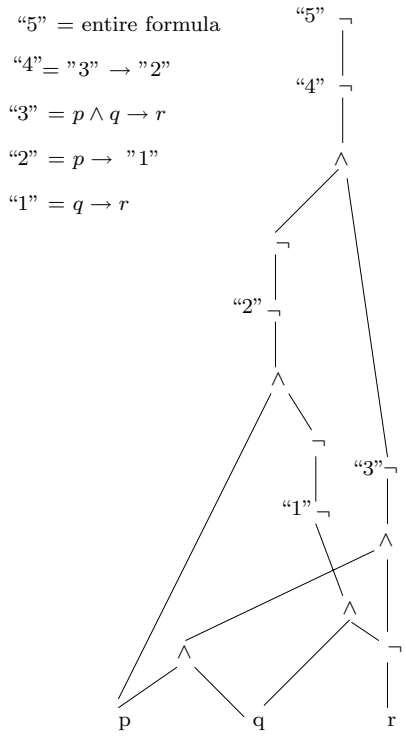


Figure 1.15. The DAG for the translation of $\neg((p \wedge q \to r) \to p \to q \to r)$. Labels "1" etc indicate which nodes represent what subformulas.

such ϕ are unsatisfiable. This SAT solver has a linear running time in the size of the DAG for $T(\phi)$. Since that size is a linear function of the length of ϕ – the translation T causes only a linear blow-up – our SAT solver has a linear running time in the length of the formula. This linearity came with a price: our linear solver fails for all formulas of the form $\neg(\phi_1 \wedge \phi_2)$.

1.6.2 A cubic solver

When we applied our linear SAT solver, we saw two possible outcomes: we either detected contradictory constraints, meaning that no formula represented by the DAG is satisfiable (e.g. Fig. 1.16); or we managed to force consistent constraints on all nodes, in which case all formulas represented by this DAG are satisfiable with those constraints as a witness (e.g. Fig. 1.13). Unfortunately, there is a third possibility: all forced constraints are consistent with each other, but not all nodes are constrained! We already remarked that this occurs for formulas of the form $\neg(\phi_1 \wedge \phi_2)$.

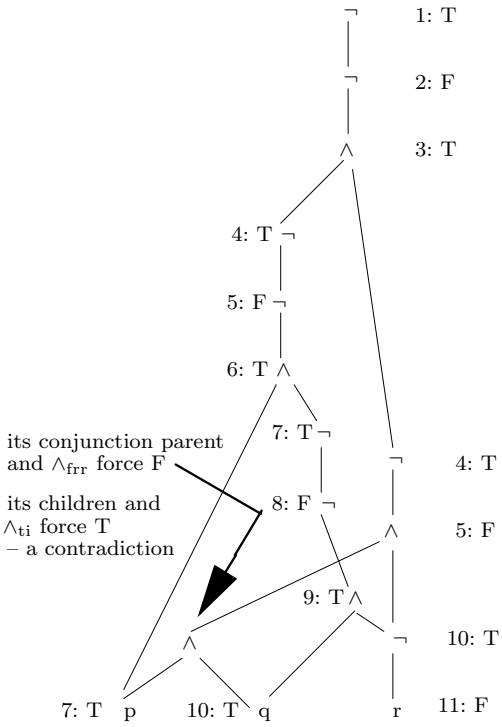


Figure 1.16. The forcing rules, applied to the DAG of Figure 1.15, detect contradictory constraints at the indicated node – implying that the initial constraint '1:T' cannot be realized. Thus, formulas represented by this DAG are not satisfiable.

Recall that checking validity of formulas in CNF is very easy. We already hinted at the fact that checking satisfiability of formulas in CNF is hard. To illustrate, consider the formula

$$((p \vee (q \vee r)) \wedge ((p \vee \neg q) \wedge ((q \vee \neg r) \wedge ((r \vee \neg p) \wedge (\neg p \vee (\neg q \vee \neg r))))))) \tag{1.11}$$

in CNF – based on Example 4.2, page 77, in [Pap94]. Intuitively, this formula should not be satisfiable. The first and last clause in (1.11) 'say' that at least one of p, q, and r are false and true (respectively). The remaining three clauses, in their conjunction, 'say' that p, q, and r all have the same truth value. This cannot be satisfiable, and a good SAT solver should discover this without any user intervention. Unfortunately, our linear SAT solver can neither detect inconsistent constraints nor compute constraints for all nodes. Figure 1.17 depicts the DAG for $T(\phi)$, where ϕ is as in (1.11); and reveals

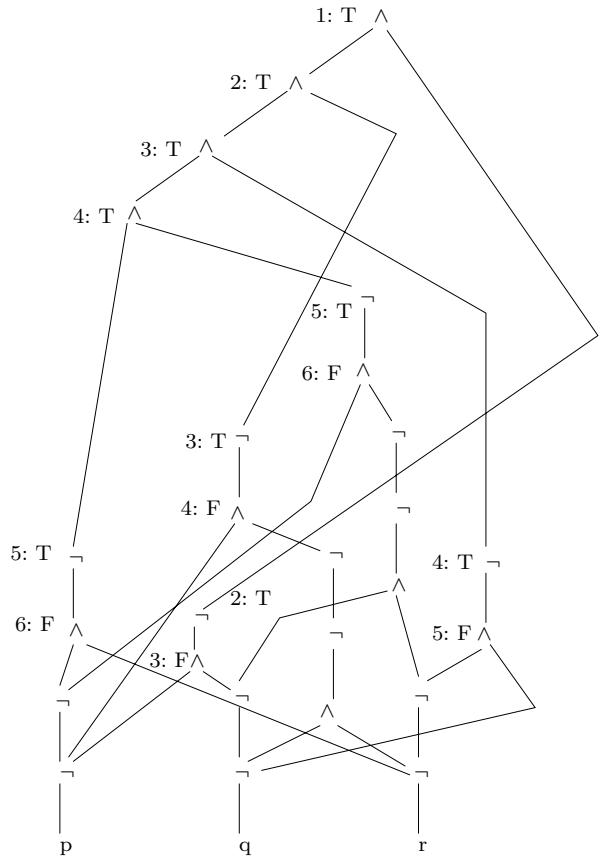


Figure 1.17. The DAG for the translation of the formula in (1.11). It has a $\wedge$-spine of length 4 as it is a conjunction of five clauses. Its linear analysis gets stuck: all forced constraints are consistent with each other but several nodes, including all atoms, are unconstrained.

that our SAT solver got stuck: no inconsistent constraints were found and not all nodes obtained constraints; in particular, no atom received a mark! So how can we improve this analysis? Well, we can mimic the role of LEM to improve the precision of our SAT solver. For the DAG with marks as in Figure 1.17, pick any node n that is not yet marked. Then *test* node n by making two independent computations:

1. determine which *temporary* marks are forced by adding to the marks in Figure 1.17 the `T` mark only to n; and
2. determine which *temporary* marks are forced by adding, again to the marks in Figure 1.17, the `F` mark only to n.

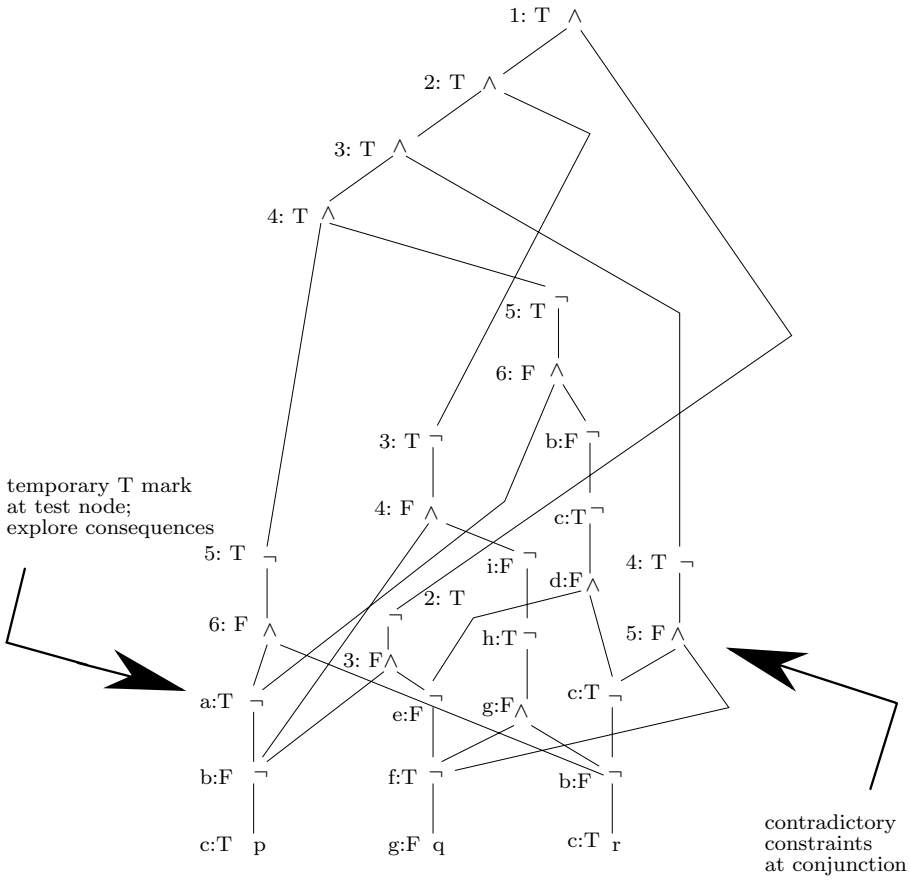


Figure 1.18. Marking an unmarked node with T and exploring what new constraints would follow from this. The analysis shows that this test marking causes contradictory constraints. We use lowercase letters 'a:' etc to denote temporary marks.

If both runs find contradictory constraints, the algorithm stops and reports that $T(\phi)$ is unsatisfiable. Otherwise, all nodes that received the same mark in both of these runs receive that very mark as a *permanent* one; that is, we update the mark state of Figure 1.17 with all such shared marks.

We test any further unmarked nodes in the same manner until we either find contradictory *permanent* marks, a complete witness to satisfiability (all nodes have consistent marks), or we have tested *all* currently unmarked nodes in this manner without detecting any shared marks. Only in the latter case does the analysis terminate without knowing whether the formulas represented by that DAG are satisfiable.

Example 1.49 We revisit our stuck analysis of Figure 1.17. We test a ¬-node and explore the consequences of setting that ¬-node's mark to `T`; Figure 1.18 shows the result of that analysis. Dually, Figure 1.19 tests the consequences of setting that ¬-node's mark to `F`. Since both runs reveal a contradiction, the algorithm terminates, ruling that the formula in (1.11) is not satisfiable.

In the exercises, you are asked to show that the specification of our cubic SAT solver is sound. Its running time is indeed cubic in the size of the DAG (and the length of original formula). One factor stems from the linear SAT solver used in each test run. A second factor is introduced since each unmarked node has to be tested. The third factor is needed since each new permanent mark causes *all* unmarked nodes to be tested again.

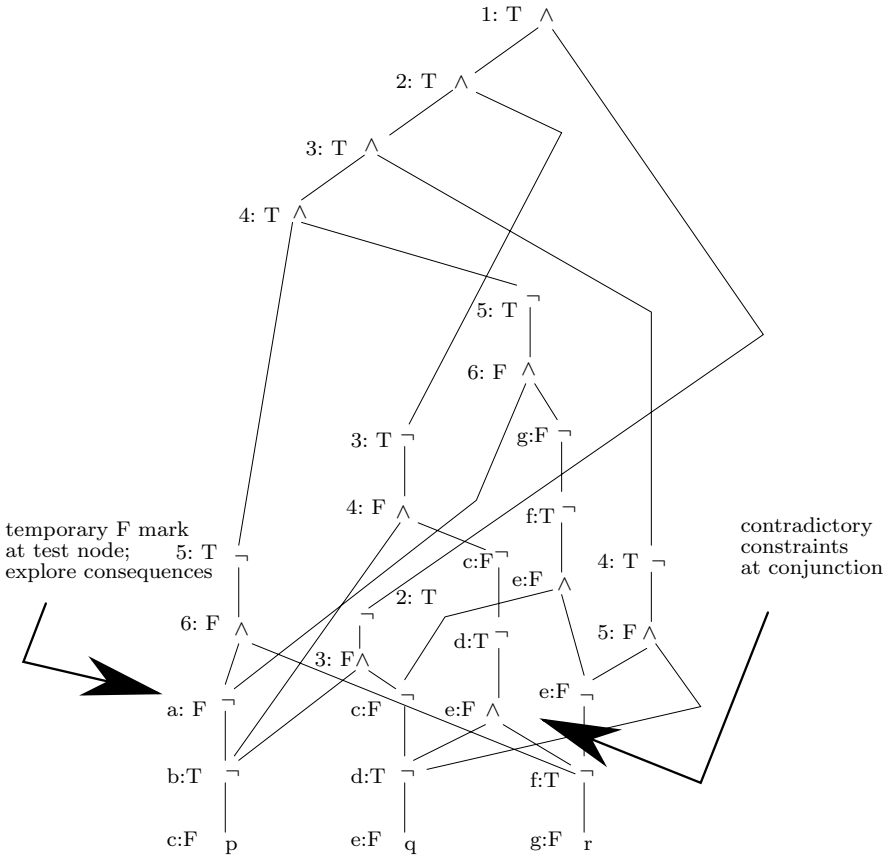


Figure 1.19. Marking the same unmarked node with `F` and exploring what new constraints would follow from this. The analysis shows that this test marking also causes contradictory constraints.

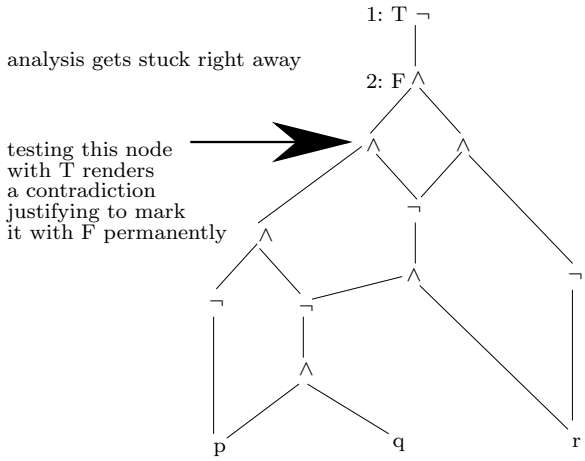


Figure 1.20. Testing the indicated node with T causes contradictory constraints, so we may mark that node with F permanently. However, our algorithm does not seem to be able to decide satisfiability of this DAG even with that optimization.

We deliberately under-specified our cubic SAT solver, but any implementation or optimization decisions need to secure soundness of the analysis. All replies of the form

1. 'The input formula is not satisfiable' and
2. 'The input formula is satisfiable under the following valuation ...'

have to be correct. The third form of reply 'Sorry, I could not figure this one out.' is correct by definition. :-) We briefly discuss two sound modifications to the algorithm that introduce some overhead, but may cause the algorithm to decide many more instances. Consider the state of a DAG right after we have explored consequences of a temporary mark on a test node.

1. If that state – permanent plus temporary markings – contains contradictory constraints, we can erase all temporary marks and mark the test node permanently with the dual mark of its test. That is, if marking node n with v resulted in a contradiction, it will get a permanent mark $\overline{v}$, where $\overline{\mathtt{T}} = \mathtt{F}$ and $\overline{\mathtt{F}} = \mathtt{T}$; otherwise
2. if that state managed to mark *all* nodes with consistent constraints, we report these markings as a witness of satisfiability and terminate the algorithm.

If none of these cases apply, we proceed as specified: promote shared marks of the two test runs to permanent ones, if applicable.

Example 1.50 To see how one of these optimizations may make a difference, consider the DAG in Figure 1.20. If we test the indicated node with

T, contradictory constraints arise. Since any witness of satisfiability has to assign some value to that node, we infer that it cannot be T. Thus, we may permanently assign mark F to that node. For this DAG, such an optimization does not seem to help. No test of an unmarked node detects a shared mark or a shared contradiction. Our cubic SAT solver fails for this DAG.

1.7 Exercises

Exercises 1.1

1. Use $\neg$, $\rightarrow$, $\wedge$ and $\vee$ to express the following declarative sentences in propositional logic; in each case state what your respective propositional atoms p, q, etc. mean:
   * (a) If the sun shines today, then it won't shine tomorrow.
   (b) Robert was jealous of Yvonne, or he was not in a good mood.
   (c) If the barometer falls, then either it will rain or it will snow.
   * (d) If a request occurs, then either it will eventually be acknowledged, or the requesting process won't ever be able to make progress.
   (e) Cancer will not be cured unless its cause is determined and a new drug for cancer is found.
   (f) If interest rates go up, share prices go down.
   (g) If Smith has installed central heating, then he has sold his car or he has not paid his mortgage.
   * (h) Today it will rain or shine, but not both.
   * (i) If Dick met Jane yesterday, they had a cup of coffee together, or they took a walk in the park.
   (j) No shoes, no shirt, no service.
   (k) My sister wants a black and white cat.
2. The formulas of propositional logic below implicitly assume the binding priorities of the logical connectives put forward in Convention 1.3. Make sure that you fully understand those conventions by reinserting as many brackets as possible. For example, given $p \wedge q \rightarrow r$, change it to $(p \wedge q) \rightarrow r$ since $\wedge$ binds more tightly than $\rightarrow$.
   * (a) $\neg p \wedge q \rightarrow r$
   (b) $(p \rightarrow q) \wedge \neg(r \vee p \rightarrow q)$
   * (c) $(p \rightarrow q) \rightarrow (r \rightarrow s \vee t)$
   (d) $p \vee (\neg q \rightarrow p \wedge r)$
   * (e) $p \vee q \rightarrow \neg p \wedge r$
   (f) $p \vee p \rightarrow \neg q$
   * (g) Why is the expression $p \vee q \wedge r$ problematic?

Exercises 1.2

1. Prove the validity of the following sequents:
   (a) $(p \wedge q) \wedge r, s \wedge t \vdash q \wedge s$

(b) $p \wedge q \vdash q \wedge p$
* (c) $(p \wedge q) \wedge r \vdash p \wedge (q \wedge r)$
(d) $p \to (p \to q), p \vdash q$
* (e) $q \to (p \to r), \neg r, q \vdash \neg p$
* (f) $\vdash (p \wedge q) \to p$
(g) $p \vdash q \to (p \wedge q)$
* (h) $p \vdash (p \to q) \to q$
* (i) $(p \to r) \wedge (q \to r) \vdash p \wedge q \to r$
* (j) $q \to r \vdash (p \to q) \to (p \to r)$
(k) $p \to (q \to r), p \to q \vdash p \to r$
* (l) $p \to q, r \to s \vdash p \vee r \to q \vee s$
(m) $p \vee q \vdash r \to (p \vee q) \wedge r$
* (n) $(p \vee (q \to p)) \wedge q \vdash p$
* (o) $p \to q,\ r \to s \vdash p \wedge r \to q \wedge s$
(p) $p \to q \vdash ((p \wedge q) \to p) \wedge (p \to (p \wedge q))$
(q) $\vdash q \to (p \to (p \to (q \to p)))$
* (r) $p \to q \wedge r \vdash (p \to q) \wedge (p \to r)$
(s) $(p \to q) \wedge (p \to r) \vdash p \to q \wedge r$
(t) $\vdash (p \to q) \to ((r \to s) \to (p \wedge r \to q \wedge s))$; here you might be able to 'recycle' and augment a proof from a previous exercise.
(u) $p \to q \vdash \neg q \to \neg p$
* (v) $p \vee (p \wedge q) \vdash p$
(w) $r, p \to (r \to q) \vdash p \to (q \wedge r)$
* (x) $p \to (q \vee r), q \to s, r \to s \vdash p \to s$
* (y) $(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)$.

2. For the sequents below, show which ones are valid and which ones aren't:
* (a) $\neg p \to \neg q \vdash q \to p$
* (b) $\neg p \vee \neg q \vdash \neg(p \wedge q)$
* (c) $\neg p, p \vee q \vdash q$
* (d) $p \vee q, \neg q \vee r \vdash p \vee r$
* (e) $p \to (q \vee r), \neg q, \neg r \vdash \neg p$ without using the MT rule
* (f) $\neg p \wedge \neg q \vdash \neg(p \vee q)$
* (g) $p \wedge \neg p \vdash \neg(r \to q) \wedge (r \to q)$
(h) $p \to q, s \to t \vdash p \vee s \to q \wedge t$
* (i) $\neg(\neg p \vee q) \vdash p$.

3. Prove the validity of the sequents below:
(a) $\neg p \to p \vdash p$
(b) $\neg p \vdash p \to q$
(c) $p \vee q, \neg q \vdash p$
* (d) $\vdash \neg p \to (p \to (p \to q))$
(e) $\neg(p \to q) \vdash q \to p$
(f) $p \to q \vdash \neg p \vee q$
(g) $\vdash \neg p \vee q \to (p \to q)$

(h) $p \to (q \vee r),\ \neg q,\ \neg r \vdash \neg p$
(i) $(c \wedge n) \to t,\ h \wedge \neg s,\ h \wedge \neg(s \vee c) \to p \vdash (n \wedge \neg t) \to p$
(j) the two sequents implict in (1.2) on page 20
(k) $q \vdash (p \wedge q) \vee (\neg p \wedge q)$ using LEM
(l) $\neg(p \wedge q) \vdash \neg p \vee \neg q$
(m) $p \wedge q \to r \vdash (p \to r) \vee (q \to r)$
* (n) $p \wedge q \vdash \neg(\neg p \vee \neg q)$
(o) $\neg(\neg p \vee \neg q) \vdash p \wedge q$
(p) $p \to q \vdash \neg p \vee q$ possibly without using LEM?
* (q) $\vdash (p \to q) \vee (q \to r)$ using LEM
(r) $p \to q,\ \neg p \to r,\ \neg q \to \neg r \vdash q$
(s) $p \to q,\ r \to \neg t,\ q \to r \vdash p \to \neg t$
(t) $(p \to q) \to r,\ s \to \neg p,\ t,\ \neg s \wedge t \to q \vdash r$
(u) $(s \to p) \vee (t \to q) \vdash (s \to q) \vee (t \to p)$
(v) $(p \wedge q) \to r,\ r \to s,\ q \wedge \neg s \vdash \neg p$.

4. Explain why intuitionistic logicians also reject the proof rule PBC.
5. Prove the following theorems of propositional logic:
* (a) $((p \to q) \to q) \to ((q \to p) \to p)$
(b) Given a proof for the sequent of the previous item, do you now have a quick argument for $((q \to p) \to p) \to ((p \to q) \to q)$?
(c) $((p \to q) \wedge (q \to p)) \to ((p \vee q) \to (p \wedge q))$
* (d) $(p \to q) \to ((\neg p \to q) \to q)$.
6. Natural deduction is not the only possible formal framework for proofs in propositional logic. As an abbreviation, we write Γ to denote any finite sequence of formulas $\phi_1, \phi_2, \ldots, \phi_n$ $(n \geq 0)$. Thus, any sequent may be written as $\Gamma \vdash \psi$ for an appropriate, possibly empty, Γ. In this exercise we propose a different notion of proof, which states rules for transforming valid sequents into valid sequents. For example, if we have already a proof for the sequent $\Gamma, \phi \vdash \psi$, then we obtain a proof of the sequent $\Gamma \vdash \phi \to \psi$ by augmenting this very proof with one application of the rule $\to$i. The new approach expresses this as an inference rule between sequents:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \to\text{i}.$$

The rule 'assumption' is written as

$$\frac{}{\phi \vdash \phi}\ \text{assumption}$$

i.e. the premise is empty. Such rules are called axioms.
(a) Express all remaining proof rules of Figure 1.2 in such a form. (Hint: some of your rules may have more than one premise.)
(b) Explain why proofs of $\Gamma \vdash \psi$ in this new system have a tree-like structure with $\Gamma \vdash \psi$ as root.
(c) Prove $p \vee (p \wedge q) \vdash p$ in your new proof system.

7. Show that $\sqrt{2}$ cannot be a rational number. Proceed by proof by contradiction: assume that $\sqrt{2}$ is a fraction k/l with integers k and $l \neq 0$. On squaring both sides we get $2 = k^2/l^2$, or equivalently $2l^2 = k^2$. We may assume that any common 2 factors of k and l have been cancelled. Can you now argue that $2l^2$ has a different number of 2 factors from k^2? Why would that be a contradiction and to what?
8. There is an alternative approach to treating negation. One could simply ban the operator $\neg$ from propositional logic and think of $\phi \to \bot$ as 'being' $\neg\phi$. Naturally, such a logic cannot rely on the natural deduction rules for negation. Which of the rules $\neg$i, $\neg$e, $\neg\neg$e and $\neg\neg$i can you simulate with the remaining proof rules by letting $\neg\phi$ be $\phi \to \bot$?
9. Let us introduce a new connective $\phi \leftrightarrow \psi$ which should abbreviate $(\phi \to \psi) \wedge (\psi \to \phi)$. Design introduction and elimination rules for $\leftrightarrow$ and show that they are derived rules if $\phi \leftrightarrow \psi$ is interpreted as $(\phi \to \psi) \wedge (\psi \to \phi)$.

Exercises 1.3

In order to facilitate reading these exercises we assume below the usual conventions about binding priorities agreed upon in Convention 1.3.

1. Given the following formulas, draw their corresponding parse tree:
   (a) p
 * (b) $p \wedge q$
   (c) $p \wedge \neg q \to \neg p$
 * (d) $p \wedge (\neg q \to \neg p)$
   (e) $p \to (\neg q \vee (q \to p))$
 * (f) $\neg((\neg q \wedge (p \to r)) \wedge (r \to q))$
   (g) $\neg p \vee (p \to q)$
   (h) $(p \wedge q) \to (\neg r \vee (q \to r))$
   (i) $((s \vee (\neg p)) \to (\neg p))$
   (j) $(s \vee ((\neg p) \to (\neg p)))$
   (k) $(((s \to (r \vee l)) \vee ((\neg q) \wedge r)) \to ((\neg(p \to s)) \to r))$
   (l) $(p \to q) \wedge (\neg r \to (q \vee (\neg p \wedge r)))$.
2. For each formula below, list all its subformulas:
 * (a) $p \to (\neg p \vee (\neg\neg q \to (p \wedge q)))$
   (b) $(s \to r \vee l) \vee (\neg q \wedge r) \to (\neg(p \to s) \to r)$
   (c) $(p \to q) \wedge (\neg r \to (q \vee (\neg p \wedge r)))$.
3. Draw the parse tree of a formula ϕ of propositional logic which is
 * (a) a negation of an implication
   (b) a disjunction whose disjuncts are both conjunctions
 * (c) a conjunction of conjunctions.
4. For each formula below, draw its parse tree and list all subformulas:
 * (a) $\neg(s \to (\neg(p \to (q \vee \neg s))))$
   (b) $((p \to \neg q) \vee (p \wedge r) \to s) \vee \neg r$.

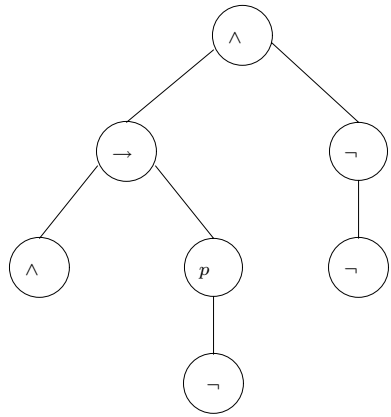


Figure 1.21. A tree that represents an ill-formed formula.

* 5. For the parse tree in Figure 1.22 find the logical formula it represents.

6. For the trees below, find their linear representations and check whether they correspond to well-formed formulas:
   (a) the tree in Figure 1.10 on page 44
   (b) the tree in Figure 1.23.

* 7. Draw a parse tree that represents an ill-formed formula such that
   (a) one can extend it by adding one or several subtrees to obtain a tree that represents a well-formed formula;
   (b) it is inherently ill-formed; i.e. any extension of it could not correspond to a well-formed formula.

8. Determine, by trying to draw parse trees, which of the following formulas are well-formed:
   (a) $p \wedge \neg(p \vee \neg q) \rightarrow (r \rightarrow s)$
   (b) $p \wedge \neg(p \vee q \wedge s) \rightarrow (r \rightarrow s)$
   (c) $p \wedge \neg(p \vee \wedge s) \rightarrow (r \rightarrow s)$.
   Among the ill-formed formulas above which ones, and in how many ways, could you 'fix' by the insertion of brackets only?

Exercises 1.4

* 1. Construct the truth table for $\neg p \vee q$ and verify that it coincides with the one for $p \rightarrow q$. (By 'coincide' we mean that the respective columns of `T` and `F` values are the same.)

2. Compute the complete truth table of the formula
   * (a) $((p \rightarrow q) \rightarrow p) \rightarrow p$
   (b) represented by the parse tree in Figure 1.3 on page 34

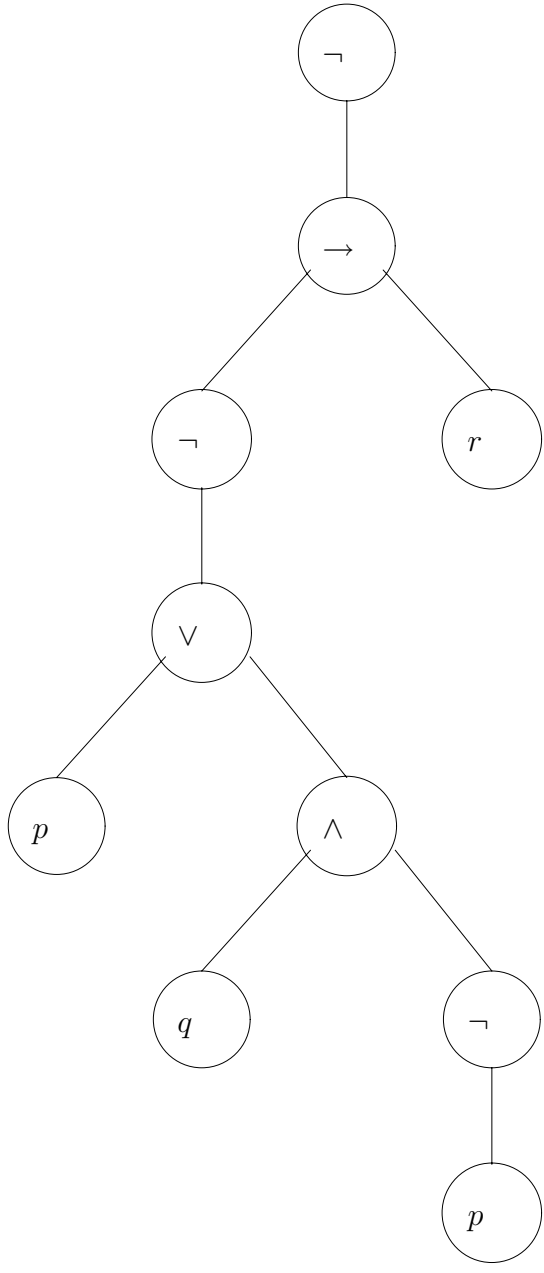


Figure 1.22. A parse tree of a negated implication.

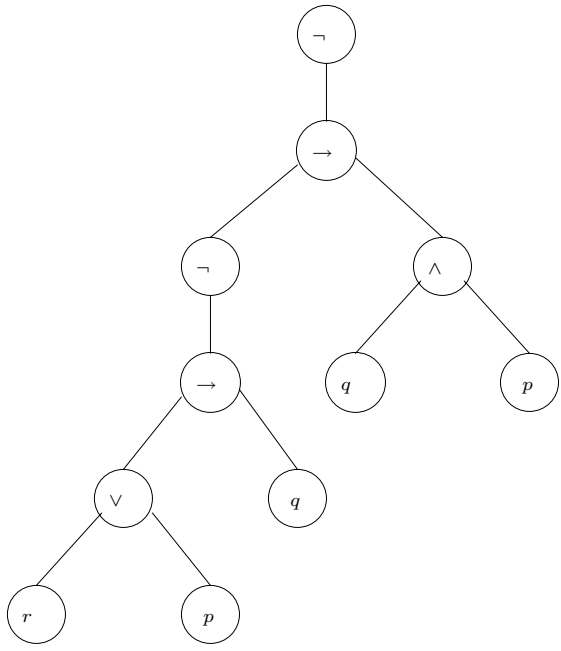


Figure 1.23. Another parse tree of a negated implication.

* (c) $p \vee (\neg(q \wedge (r \to q)))$
(d) $(p \wedge q) \to (p \vee q)$
(e) $((p \to \neg q) \to \neg p) \to q$
(f) $(p \to q) \vee (p \to \neg q)$
(g) $((p \to q) \to p) \to p$
(h) $((p \vee q) \to r) \to ((p \to r) \vee (q \to r))$
(i) $(p \to q) \to (\neg p \to \neg q)$.

3. Given a valuation and a parsetree of a formula, compute the truth value of the formula for that valuation (as done in a bottom-up fashion in Figure 1.7 on page 40) with the parse tree in
* (a) Figure 1.10 on page 44 and the valuation in which q and r evaluate to `T` and p to `F`;
(b) Figure 1.4 on page 36 and the valuation in which q evaluates to `T` and p and r evaluate to `F`;
(c) Figure 1.23 where we let p be `T`, q be `F` and r be `T`; and
(d) Figure 1.23 where we let p be `F`, q be `T` and r be `F`.
4. Compute the truth value on the formula's parse tree, or specify the corresponding line of a truth table where
* (a) p evaluates to `F`, q to `T` and the formula is $p \to (\neg q \vee (q \to p))$
* (b) the formula is $\neg((\neg q \wedge (p \to r)) \wedge (r \to q))$, p evaluates to `F`, q to `T` and r evaluates to `T`.

* 5. A formula is valid iff it computes `T` for all its valuations; it is satisfiable iff it computes `T` for at least one of its valuations. Is the formula of the parse tree in Figure 1.10 on page 44 valid? Is it satisfiable?

6. Let $*$ be a new logical connective such that $p * q$ does not hold iff p and q are either both false or both true.
   (a) Write down the truth table for $p * q$.
   (b) Write down the truth table for $(p * p) * (q * q)$.
   (c) Does the table in (b) coincide with a table in Figure 1.6 (page 38)? If so, which one?
   (d) Do you know $*$ already as a logic gate in circuit design? If so, what is it called?

7. These exercises let you practice proofs using mathematical induction. Make sure that you state your base case and inductive step clearly. You should also indicate where you apply the induction hypothesis.
   (a) Prove that

$$(2 \cdot 1 - 1) + (2 \cdot 2 - 1) + (2 \cdot 3 - 1) + \cdots + (2 \cdot n - 1) = n^2$$

   by mathematical induction on $n \geq 1$.
   (b) Let k and l be natural numbers. We say that k is divisible by l if there exists a natural number p such that $k = p \cdot l$. For example, 15 is divisible by 3 because $15 = 5 \cdot 3$. Use mathematical induction to show that $11^n - 4^n$ is divisible by 7 for all natural numbers $n \geq 1$.
* (c) Use mathematical induction to show that

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

   for all natural numbers $n \geq 1$.
* (d) Prove that $2^n \geq n + 12$ for all natural numbers $n \geq 4$. Here the base case is $n = 4$. Is the statement true for any $n < 4$?
   (e) Suppose a post office sells only 2¢ and 3¢ stamps. Show that any postage of 2¢, or over, can be paid for using only these stamps. Hint: use mathematical induction on n, where n¢ is the postage. In the inductive step consider two possibilities: first, n¢ can be paid for using only 2¢ stamps. Second, paying n¢ requires the use of at least one 3¢ stamp.
   (f) Prove that for every prefix of a well-formed propositional logic formula the number of left brackets is greater or equal to the number of right brackets.

* 8. The Fibonacci numbers are most useful in modelling the growth of populations. We define them by $F_1 \stackrel{\text{def}}{=} 1$, $F_2 \stackrel{\text{def}}{=} 1$ and $F_{n+1} \stackrel{\text{def}}{=} F_n + F_{n-1}$ for all $n \geq 2$. So $F_3 \stackrel{\text{def}}{=} F_1 + F_2 = 1 + 1 = 2$ etc. Show the assertion 'F_{3n} is even.' by mathematical induction on $n \geq 1$. Note that this assertion is saying that the sequence $F_3, F_6, F_9, \ldots$ consists of even numbers only.

9. Consider the function rank, defined by

$$\begin{aligned} \mathsf{rank}(p) &\stackrel{\text{def}}{=} 1 \\ \mathsf{rank}(\neg\phi) &\stackrel{\text{def}}{=} 1 + \mathsf{rank}(\phi) \\ \mathsf{rank}(\phi \circ \psi) &\stackrel{\text{def}}{=} 1 + \max(\mathsf{rank}(\phi), \mathsf{rank}(\psi)) \end{aligned}$$

where p is any atom, $\circ \in \{\rightarrow, \vee, \wedge\}$ and $\max(n, m)$ is n if $n \geq m$ and m otherwise. Recall the concept of the height of a formula (Definition 1.32 on page 44). Use mathematical induction on the height of ϕ to show that $\mathsf{rank}(\phi)$ is nothing but the height of ϕ for all formulas ϕ of propositional logic.

* 10. Here is an example of why we need to secure the base case for mathematical induction. Consider the assertion

'The number $n^2 + 5n + 1$ is even for all $n \geq 1$.'

(a) Prove the inductive step of that assertion.
(b) Show that the base case fails to hold.
(c) Conclude that the assertion is false.
(d) Use mathematical induction to show that $n^2 + 5n + 1$ is odd for all $n \geq 1$.

11. For the soundness proof of Theorem 1.35 on page 46,
(a) explain why we could not use mathematical induction but had to resort to course-of-values induction;
(b) give justifications for all inferences that were annotated with 'why?' and
(c) complete the case analysis ranging over the final proof rule applied; inspect the summary of natural deduction rules in Figure 1.2 on page 27 to see which cases are still missing. Do you need to include derived rules?

12. Show that the following sequents are not valid by finding a valuation in which the truth values of the formulas to the left of $\vdash$ are `T` and the truth value of the formula to the right of $\vdash$ is `F`.
(a) $\neg p \vee (q \rightarrow p) \vdash \neg p \wedge q$
(b) $\neg r \rightarrow (p \vee q), r \wedge \neg q \vdash r \rightarrow q$
* (c) $p \rightarrow (q \rightarrow r) \vdash p \rightarrow (r \rightarrow q)$
(d) $\neg p, p \vee q \vdash \neg q$
(e) $p \rightarrow (\neg q \vee r), \neg r \vdash \neg q \rightarrow \neg p$.

13. For each of the following invalid sequents, give examples of natural language declarative sentences for the atoms p, q and r such that the premises are true, but the conclusion false.
* (a) $p \vee q \vdash p \wedge q$
* (b) $\neg p \rightarrow \neg q \vdash \neg q \rightarrow \neg p$
(c) $p \rightarrow q \vdash p \vee q$
(d) $p \rightarrow (q \vee r) \vdash (p \rightarrow q) \wedge (p \rightarrow r)$.

14. Find a formula of propositional logic ϕ which contains only the atoms p, q and r and which is true only when p and q are false, or when $\neg q \wedge (p \vee r)$ is true.

15. Use mathematical induction on n to prove the theorem $((\phi_1 \wedge (\phi_2 \wedge (\cdots \wedge \phi_n)\ldots) \rightarrow \psi) \rightarrow (\phi_1 \rightarrow (\phi_2 \rightarrow (\ldots(\phi_n \rightarrow \psi)\ldots))))$.
16. Prove the validity of the following sequents needed to secure the completeness result for propositional logic:
    (a) $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \rightarrow \phi_2)$
    (b) $\neg\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \rightarrow \phi_2$
    (c) $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \rightarrow \phi_2$
    (d) $\phi_1 \wedge \phi_2 \vdash \phi_1 \rightarrow \phi_2$
    (e) $\neg\phi_1 \wedge \phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$
    (f) $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$
    (g) $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$
    (h) $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \vee \phi_2)$
    (i) $\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$
    (j) $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$
    (k) $\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \vee \phi_2$.
17. Does $\vDash \phi$ hold for the ϕ below? Please justify your answer.
    (a) $(p \rightarrow q) \vee (q \rightarrow r)$
    * (b) $((q \rightarrow (p \vee (q \rightarrow p))) \vee \neg(p \rightarrow q)) \rightarrow p$.

Exercises 1.5

1. Show that a formula ϕ is valid iff $\top \equiv \phi$, where $\top$ is an abbreviation for an instance $p \vee \neg p$ of LEM.
2. Which of these formulas are semantically equivalent to $p \rightarrow (q \vee r)$?
    (a) $q \vee (\neg p \vee r)$
    * (b) $q \wedge \neg r \rightarrow p$
    (c) $p \wedge \neg r \rightarrow q$
    * (d) $\neg q \wedge \neg r \rightarrow \neg p$.
3. An adequate set of connectives for propositional logic is a set such that for every formula of propositional logic there is an equivalent formula with only connectives from that set. For example, the set $\{\neg, \vee\}$ is adequate for propositional logic, because any occurrence of $\wedge$ and $\rightarrow$ can be removed by using the equivalences $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$ and $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$.
    (a) Show that $\{\neg, \wedge\}$, $\{\neg, \rightarrow\}$ and $\{\rightarrow, \bot\}$ are adequate sets of connectives for propositional logic. (In the latter case, we are treating $\bot$ as a nullary connective.)
    (b) Show that, if $C \subseteq \{\neg, \wedge, \vee, \rightarrow, \bot\}$ is adequate for propositional logic, then $\neg \in C$ or $\bot \in C$. (Hint: suppose C contains neither $\neg$ nor $\bot$ and consider the truth value of a formula ϕ, formed by using only the connectives in C, for a valuation in which every atom is assigned `T`.)
    (c) Is $\{\leftrightarrow, \neg\}$ adequate? Prove your answer.
4. Use soundness or completeness to show that a sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ has a proof iff $\phi_1 \rightarrow \phi_2 \rightarrow \ldots \phi_n \rightarrow \psi$ is a tautology.

5. Show that the relation $\equiv$ is
   (a) reflexive: $\phi \equiv \phi$ holds for all ϕ
   (b) symmetric: $\phi \equiv \psi$ implies $\psi \equiv \phi$ and
   (c) transitive: $\phi \equiv \psi$ and $\psi \equiv \eta$ imply $\phi \equiv \eta$.
6. Show that, with respect to $\equiv$,
   (a) $\wedge$ and $\vee$ are idempotent:
      i. $\phi \wedge \phi \equiv \phi$
      ii. $\phi \vee \phi \equiv \phi$
   (b) $\wedge$ and $\vee$ are commutative:
      i. $\phi \wedge \psi \equiv \psi \wedge \phi$
      ii. $\phi \vee \psi \equiv \psi \vee \phi$
   (c) $\wedge$ and $\vee$ are associative:
      i. $\phi \wedge (\psi \wedge \eta) \equiv (\phi \wedge \psi) \wedge \eta$
      ii. $\phi \vee (\psi \vee \eta) \equiv (\phi \vee \psi) \vee \eta$
   (d) $\wedge$ and $\vee$ are absorptive:
   * i. $\phi \wedge (\phi \vee \eta) \equiv \phi$
      ii. $\phi \vee (\phi \wedge \eta) \equiv \phi$
   (e) $\wedge$ and $\vee$ are distributive:
      i. $\phi \wedge (\psi \vee \eta) \equiv (\phi \wedge \psi) \vee (\phi \wedge \eta)$
   * ii. $\phi \vee (\psi \wedge \eta) \equiv (\phi \vee \psi) \wedge (\phi \vee \eta)$
   (f) $\equiv$ allows for double negation: $\phi \equiv \neg\neg\phi$ and
   (g) $\wedge$ and $\vee$ satisfies the de Morgan rules:
      i. $\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$
   * ii. $\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$.
7. Construct a formula in CNF based on each of the following truth tables:
   * (a)

p	q	ϕ_1
T	T	F
F	T	F
T	F	F
F	F	T

   * (b)

p	q	r	ϕ_2
T	T	T	T
T	T	F	F
T	F	T	F
F	T	T	T
T	F	F	F
F	T	F	F
F	F	T	T
F	F	F	F

(c)

r	s	q	ϕ_3
T	T	T	F
T	T	F	T
T	F	T	F
F	T	T	F
T	F	F	T
F	T	F	F
F	F	T	F
F	F	F	T

* 8. Write a recursive function `IMPL_FREE` which requires a (parse tree of a) propositional formula as input and produces an equivalent implication-free formula as output. How many clauses does your case statement need? Recall Definition 1.27 on page 32.

* 9. Compute `CNF` (`NNF` (`IMPL_FREE` $\neg(p \to (\neg(q \wedge (\neg p \to q))))))$.

10. Use structural induction on the grammar of formulas in CNF to show that the 'otherwise' case in calls to `DISTR` applies iff both η_1 and η_2 are of type D in (1.6) on page 55.

11. Use mathematical induction on the height of ϕ to show that the call `CNF` (`NNF` (`IMPL_FREE` ϕ)) returns, up to associativity, ϕ if the latter is already in CNF.

12. Why do the functions `CNF` and `DISTR` preserve NNF and why is this important?

13. For the call `CNF` (`NNF` (`IMPL_FREE` (ϕ))) on a formula ϕ of propositional logic, explain why
    (a) its output is always a formula in CNF
    (b) its output is semantically equivalent to ϕ
    (c) that call always terminates.

14. Show that all the algorithms presented in Section 1.5.2 terminate on any input meeting their precondition. Can you formalise some of your arguments? Note that algorithms might not call themselves again on formulas with smaller height. E.g. the call of `CNF` ($\phi_1 \vee \phi_2$) results in a call `DISTR` (`CNF`(ϕ_1), `CNF`(ϕ_2)), where `CNF`(ϕ_i) may have greater height than ϕ_i. Why is this not a problem?

15. Apply algorithm `HORN` from page 66 to each of these Horn formulas:
    * (a) $(p \wedge q \wedge w \to \bot) \wedge (t \to \bot) \wedge (r \to p) \wedge (\top \to r) \wedge (\top \to q) \wedge (u \to s) \wedge (\top \to u)$
    (b) $(p \wedge q \wedge w \to \bot) \wedge (t \to \bot) \wedge (r \to p) \wedge (\top \to r) \wedge (\top \to q) \wedge (r \wedge u \to w) \wedge (u \to s) \wedge (\top \to u)$
    (c) $(p \wedge q \wedge s \to p) \wedge (q \wedge r \to p) \wedge (p \wedge s \to s)$
    (d) $(p \wedge q \wedge s \to \bot) \wedge (q \wedge r \to p) \wedge (\top \to s)$
    (e) $(p_5 \to p_{11}) \wedge (p_2 \wedge p_3 \wedge p_5 \to p_{13}) \wedge (\top \to p_5) \wedge (p_5 \wedge p_{11} \to \bot)$
    (f) $(\top \to q) \wedge (\top \to s) \wedge (w \to \bot) \wedge (p \wedge q \wedge s \to \bot) \wedge (v \to s) \wedge (\top \to r) \wedge (r \to p)$

* (g) $(\top \to q) \wedge (\top \to s) \wedge (w \to \bot) \wedge (p \wedge q \wedge s \to v) \wedge (v \to s) \wedge (\top \to r) \wedge (r \to p)$.

16. Explain why the algorithm HORN fails to work correctly if we change the concept of Horn formulas by extending the clause for P on page 65 to $P ::= \bot \mid \top \mid p \mid \neg p$?
17. What can you say about the CNF of Horn formulas. More precisely, can you specify syntactic criteria for a CNF that ensure that there is an equivalent Horn formula? Can you describe informally programs which would translate from one form of representation into another?

Exercises 1.6

1. Use mathematical induction to show that, for all ϕ of (1.3) on page 33,
   (a) $T(\phi)$ can be generated by (1.10) on page 69,
   (b) $T(\phi)$ has the same set of valuations as ϕ, and
   (c) the set of valuations in which ϕ is true equals the set of valuations in which $T(\phi)$ is true.

* 2. Show that all rules of Figure 1.14 (page 71) are sound: if all current marks satisfy the invariant (1.9) from page 68, then this invariant still holds if the derived constraint of that rule becomes an additional mark.
3. In Figure 1.16 on page 73 we detected a contradiction which secured the validity of the sequent $p \wedge q \to r \vdash p \to q \to r$. Use the same method with the linear SAT solver to show that the sequent $\vdash (p \to q) \vee (r \to p)$ is valid. (This is interesting since we proved this validity in natural deduction with a judicious choice of the proof rule LEM; and the linear SAT solver does not employ any case analysis.)

* 4. Consider the sequent $p \vee q, p \to r \vdash r$. Determine a DAG which is not satisfiable iff this sequent is valid. Tag the DAG's root node with '1: T,' apply the forcing laws to it, and extract a witness to the DAG's satisfiability. Explain in what sense this witness serves as an explanation for the fact that $p \vee q, p \to r \vdash r$ is not valid.
5. Explain in what sense the SAT solving technique, as presented in this chapter, can be used to check whether formulas are tautologies.
6. For ϕ from (1.10), can one reverse engineer ϕ from the DAG of $T(\phi)$?
7. Consider a modification of our method which initially tags a DAG's root node with '1: F.' In that case,
   (a) are the forcing laws still sound? If so, state the invariant.
   (b) what can we say about the formula(s) a DAG represents if
      i. we detect contradictory constraints?
      ii. we compute consistent forced constraints for each node?
8. Given an arbitrary Horn formula ϕ, compare our linear SAT solver – applied to $T(\phi)$ – to the marking algorithm – applied to ϕ. Discuss similarities and differences of these approaches.

9. Consider Figure 1.20 on page 77. Verify that
   (a) its test produces contradictory constraints
   (b) its cubic analysis does not decide satisfiability, regardless of whether the two optimizations we described are present.
10. Verify that the DAG of Figure 1.17 (page 74) is indeed the one obtained for $T(\phi)$, where ϕ is the formula in (1.11) on page 73.
* 11. An implementor may be concerned with the possibility that the answers to the cubic SAT solver may depend on a particular order in which we test unmarked nodes or use the rules in Figure 1.14. Give a semi-formal argument for why the analysis results don't depend on such an order.
12. Find a formula ϕ such that our cubic SAT solver cannot decide the satisfiability of $T(\phi)$.
13. **Advanced Project:** Write a complete implementation of the cubic SAT solver described in Section 1.6.2. It should read formulas from the keyboard or a file; should assume right-associativity of $\vee$, $\wedge$, and $\rightarrow$ (respectively); compute the DAG of $T(\phi)$; perform the cubic SAT solver next. Think also about including appropriate user output, diagnostics, and optimizations.
14. Show that our cubic SAT solver specified in this section
   (a) terminates on all syntactically correct input;
   (b) satisfies the invariant (1.9) after the first permanent marking;
   (c) preserves (1.9) for all permanent markings it makes;
   (d) computes only correct satisfiability witnesses;
   (e) computes only correct 'not satisfiable' replies; and
   (f) remains to be correct under the two modifications described on page 77 for handling results of a node's two test runs.

1.8 Bibliographic notes

Logic has a long history stretching back at least 2000 years, but the truth-value semantics of propositional logic presented in this and every logic textbook today was invented only about 160 years ago, by G. Boole [Boo54]. Boole used the symbols + and · for disjunction and conjunction.

Natural deduction was invented by G. Gentzen [Gen69], and further developed by D. Prawitz [Pra65]. Other proof systems existed before then, notably axiomatic systems which present a small number of axioms together with the rule *modus ponens* (which we call $\rightarrow$e). Proof systems often present as small a number of axioms as possible; and only for an adequate set of connectives such as $\rightarrow$ and $\neg$. This makes them hard to use in practice. Gentzen improved the situation by inventing the idea of working with assumptions (used by the rules $\rightarrow$i, $\neg$i and $\vee$e) and by treating all the connectives separately.

Our linear and cubic SAT solvers are variants of Stålmarck's method [SS90], a SAT solver which is patented in Sweden and in the United States of America.

Further historical remarks, and also pointers to other contemporary books about propositional and predicate logic, can be found in the bibliographic remarks at the end of Chapter 2. For an introduction to algorithms and data structures see e.g. [Wei98].

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?)

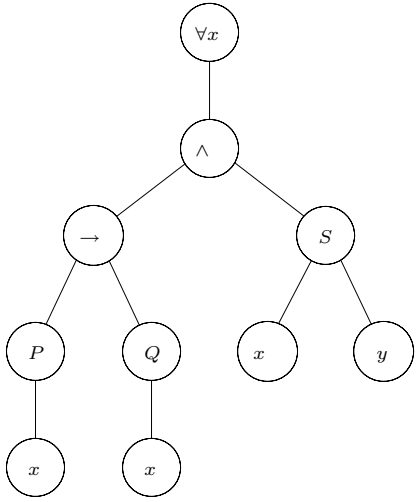


Figure 2.1. A parse tree of a predicate logic formula.

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

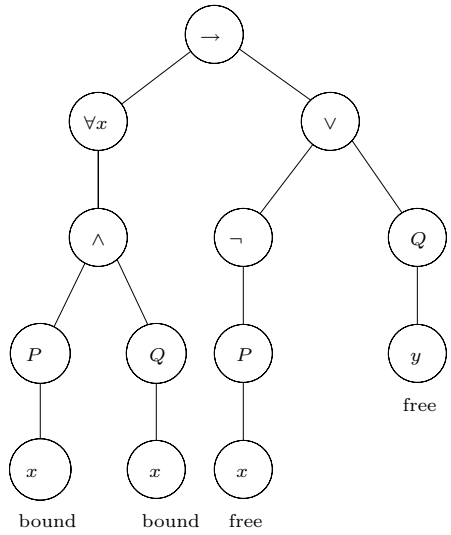


Figure 2.2. A parse tree of a predicate logic formula illustrating free and bound occurrences of variables.

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.

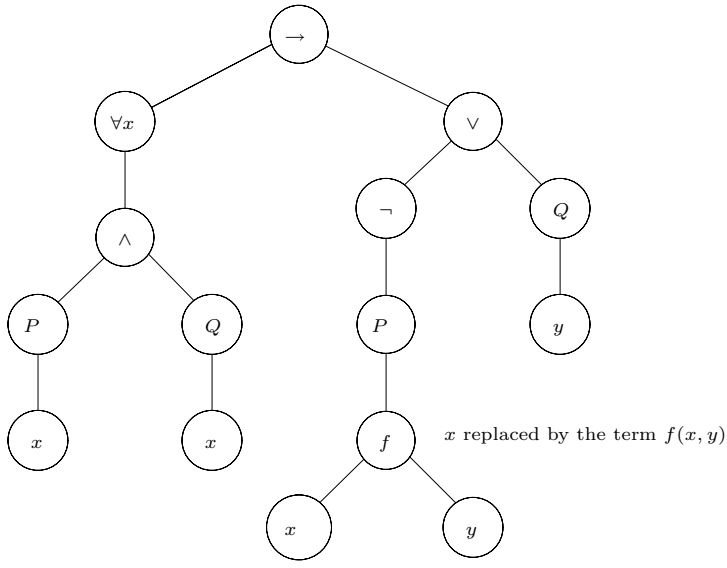


Figure 2.3. A parse tree of a formula resulting from substitution.

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.

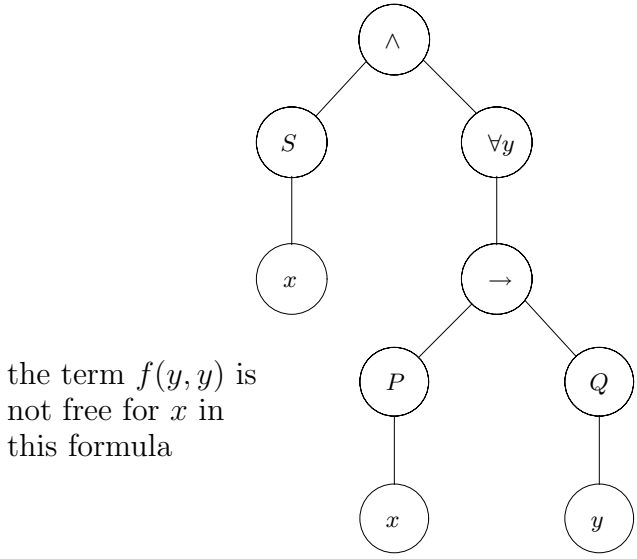


Figure 2.4. A parse tree for which a substitution has dire consequences.

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.

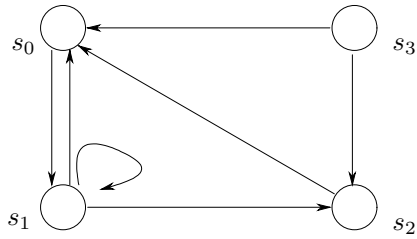


Figure 2.5. A directed graph, which is a model $\mathcal{M}$ for a predicate symbol R with two arguments. A pair of nodes (n, n') is in the interpretation $R^{\mathcal{M}}$ of R iff there is a transition (an edge) from node n to node n' in that graph.

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.

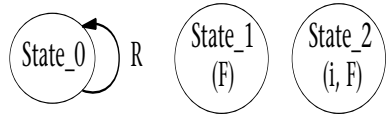


Figure 2.7. Alloy's analyzer finds a state machine model (with one transition only) within the specified scope such that the assertion `FinalNotInitial` is false: the initial state `State_2` is also final.

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

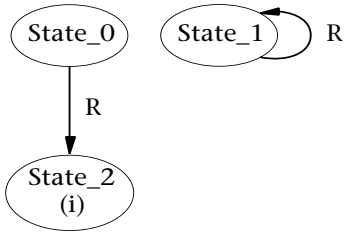


Figure 2.8. Alloy's analyzer finds a state machine model within the specified scope such that the consistency check `AGuidedSimulation` is true: there is a non-final deadlocked state, here `State_2`.

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.

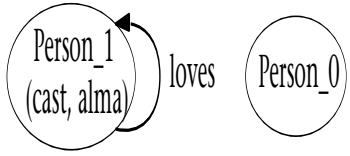


Figure 2.10. Alloy's analyzer finds a counterexample to the formula in (2.8): Alma is the only cast member and loves herself.

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

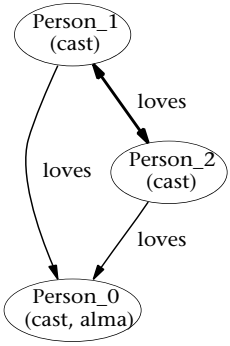


Figure 2.11. Alloy's analyzer finds a counterexample to the formula in (2.8) that meets the constraint of `NoSelfLove` with three cast members. The bidirectional arrow indicates that `Person_1` loves `Person_2` and vice versa.

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. $f(x, g(y, z), d)$

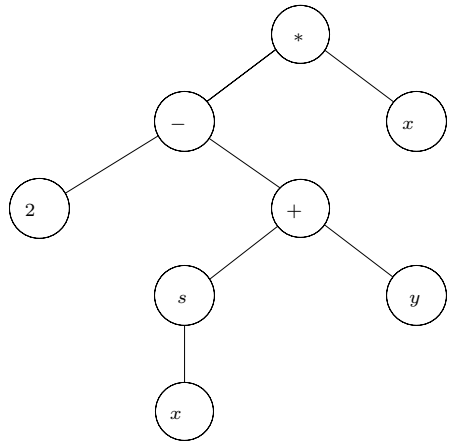


Figure 2.14. A parse tree representing an arithmetic term.

* 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;

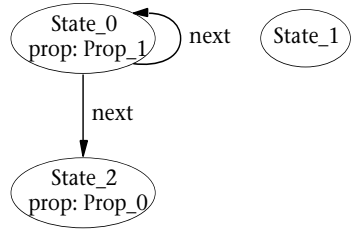


Figure 2.15. A snapshot of a non-deterministic state machine in which no non-final state deadlocks and where states that satisfy the same properties are identical.

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/`