the following trace:

after iteration	z at `l1`	y at `l1`	B at `l1`
0	0	1	true
1	1	1	true
2	2	2	true
3	3	6	true
4	4	24	true
5	5	120	true
6	6	720	false

The program execution stops when the boolean guard becomes false.

The invariant of this example is easy to see: it is '$y = z!$'. Every time we complete an execution of the body of the while-statement, this fact is true, even though the values of y and z have been changed. Moreover, this invariant has the needed properties. It is

- weak enough to be implied by the precondition of the while-statement, which we will shortly discover to be $y = 1 \wedge z = 0$ based on the initial assignments and their precondition $0! \stackrel{\text{def}}{=} 1$,
- but also strong enough that, together with the negation of the boolean guard, it implies the postcondition '$y = x!$'.

That is to say, the sequents

$$\vdash_{\text{AR}} (y = 1 \wedge z = 0) \to (y = z!) \text{ and } \vdash_{\text{AR}} (y = z! \wedge x = z) \to (y = x!) \tag{4.11}$$

are valid.

As in this example, a suitable invariant is often discovered by looking at the logical structure of the postcondition. A complete proof of the factorial example in tree form, using this invariant, was given in Figure 4.2.

How should we use the while-rule in proof tableaux? We need to think about how to push an arbitrary postcondition ψ upwards through a while-statement to meet the precondition ϕ. The steps are:

1. Guess a formula η which you hope is a suitable invariant.
2. Try to prove that $\vdash_{\text{AR}} \eta \wedge \neg B \to \psi$ and $\vdash_{\text{AR}} \phi \to \eta$ are valid, where B is the boolean guard of the while-statement. If both proofs succeed, go to 3. Otherwise (if at least one proof fails), go back to 1.
3. Push η upwards through the body C of the while-statement; this involves applying other rules dictated by the form of C. Let us name the formula that emerges η'.

4. Try to prove that $\vdash_{\mathrm{AR}} \eta \wedge B \rightarrow \eta'$ is valid; this proves that η is indeed an invariant. If you succeed, go to 5. Otherwise, go back to 1.
5. Now write η above the while-statement and write ϕ above that η, annotating that η with an instance of Implied based on the successful proof of the validity of $\vdash_{\mathrm{AR}} \phi \rightarrow \eta$ in 2. Mission accomplished!

Example 4.17 We continue the example of the factorial. The partial proof obtained by pushing $y = x!$ upwards through the while-statement – thus checking the hypothesis that $y = z!$ is an invariant – is as follows:

```
y = 1;
z = 0;
   (|y = z!|)                        ?
while (z != x) {
        (|y = z! ∧ z ≠ x|)           Invariant Hyp. ∧ guard
        (|y · (z + 1) = (z + 1)!|)   Implied
    z = z + 1;
        (|y · z = z!|)               Assignment
    y = y * z;
        (|y = z!|)                   Assignment
}
   (|y = x!|)                        ?
```

Whether $y = z!$ is a suitable invariant depends on three things:

- The ability to prove that it is indeed an invariant, i.e. that $y = z!$ implies $y \cdot (z + 1) = (z + 1)!$. This is the case, since we just multiply each side of $y = z!$ by $z + 1$ and appeal to the inductive definition of $(z + 1)!$ in Example 4.2.
- The ability to prove that η is strong enough that it and the negation of the boolean guard together imply the postcondition; this is also the case, for $y = z!$ and $x = z$ imply $y = x!$.
- The ability to prove that η is weak enough to be established by the code leading up to the while-statement. This is what we prove by continuing to push the result upwards through the code preceding the while-statement.

Continuing, then: pushing $y = z!$ through `z = 0` results in $y = 0!$ and pushing that through `y = 1` renders $1 = 0!$. The latter holds in all states as $0!$ is

defined to be 1, so it is implied by $\top$; our completed proof is:

```
    (| ⊤ |)
    (| 1 = 0! |)                          Implied
y = 1;
    (| y = 0! |)                          Assignment
z = 0;
    (| y = z! |)                          Assignment
while (z != x) {
        (| y = z! ∧ z ≠ x |)              Invariant Hyp. ∧ guard
        (| y · (z + 1) = (z + 1)! |)      Implied
    z = z + 1;
        (| y · z = z! |)                  Assignment
    y = y * z;
        (| y = z! |)                      Assignment
}
    (| y = z! ∧ ¬(z ≠ x) |)               Partial-while
    (| y = x! |)                          Implied
```

4.3.3 A case study: minimal-sum section

We practice the proof rule for while-statements once again by verifying a program which computes the minimal-sum section of an array of integers. For that, let us extend our core programming language with arrays of integers[5]. For example, we may declare an array

```
int a[n];
```

whose name is `a` and whose fields are accessed by `a[0]`, `a[1]`, ..., `a[n-1]`, where `n` is some constant. Generally, we allow any integer expression E to compute the field index, as in `a[E]`. It is the programmer's responsibility to make sure that the value computed by `E` is always within the array bounds.

Definition 4.18 Let $a[0], \dots, a[n-1]$ be the integer values of an array `a`. A section of `a` is a continuous piece $a[i], \dots, a[j]$, where $0 \le i \le j < n$. We

[5] We only read from arrays in the program `Min_Sum` which follows. Writing to arrays introduces additional problems because an array element can have several syntactically different names and this has to be taken into account by the calculus.

write $S_{i,j}$ for the sum of that section: $a[i] + a[i+1] + \cdots + a[j]$. A minimal-sum section is a section $a[i], \ldots, a[j]$ of a such that the sum $S_{i,j}$ is less than or equal to the sum $S_{i',j'}$ of any other section $a[i'], \ldots, a[j']$ of a.

Example 4.19 Let us illustrate these concepts on the example integer array $[-1, 3, 15, -6, 4, -5]$. Both $[3, 15, -6]$ and $[-6]$ are sections, but $[3, -6, 4]$ isn't since 15 is missing. A minimal-sum section for this particular array is $[-6, 4, -5]$ with sum -7; it is the only minimal-sum section in this case.

In general, minimal-sum sections need not be unique. For example, the array $[1, -1, 3, -1, 1]$ has two minimal-sum sections $[1, -1]$ and $[-1, 1]$ with minimal sum 0.

The task at hand is to

- write a program Min_Sum, written in our core programming language extended with integer arrays, which computes the sum of a minimal-sum section of a given array;
- make the informal requirement of this problem, given in the previous item, into a formal specification about the behaviour of Min_Sum;
- use our proof calculus for partial correctness to show that Min_Sum satisfies those formal specifications provided that it terminates.

There is an obvious program to do the job: we could list all the possible sections of a given array, then traverse that list to compute the sum of each section and keep the recent minimal sum in a storage location. For the example array $[-1, 3, -2]$, this results in the list

$$[-1],\ [-1, 3],\ [-1, 3, -2],\ [3],\ [3, -2],\ [-2]$$

and we see that only the last section $[-2]$ produces the minimal sum -2. This idea can easily be coded in our core programming language, but it has a serious drawback: the number of sections of a given array of size n is proportional to the square of n; if we also have to sum all those, then our task has worst-case time complexity of the order $n \cdot n^2 = n^3$. Computationally, this is an expensive price to pay, so we should inspect the problem more closely in order to see whether we can do better.

Can we compute the minimal sum over all sections in time proportional to n, by passing through the array just once? Intuitively, this seems difficult, since if we store just the minimal sum seen so far as we pass through the array, we may miss the opportunity of some large negative numbers later on because of some large positive numbers we encounter en route. For example,

suppose the array is

$$[-8, 3, -65, 20, 45, -100, -8, 17, -4, -14].$$

Should we settle for $-8+3-65$, or should we try to take advantage of the -100 – remembering that we can pass through the array only once? In this case, the whole array is a section that gives us the smallest sum, but it is difficult to see how a program which passes through the array just once could detect this.

The solution is to store two values during the pass: the minimal sum seen so far (s in the program below) and also the minimal sum seen so far of *all* sections which end at the current point in the array (t below). Here is a program that is intended to do this:

```
k = 1;
t = a[0];
s = a[0];
while (k != n) {
   t = min(t + a[k], a[k]);
   s = min(s,t);
   k = k + 1;
}
```

where `min` is a function which computes the minimum of its two arguments as specified in exercise 10 on page 301. The variable k proceeds through the range of indexes of the array and t stores the minimal sum of sections that end at $a[k]$ – whenever the control flow of the program is about to evaluate the boolean expression of its while-statement. As each new value is examined, we can either add it to the current minimal sum, or decide that a lower minimal sum can be obtained by starting a new section. The variable s stores the minimal sum seen so far; it is computed as the minimum we have seen so far in the last step, or the minimal sum of sections that end at the current point.

As you can see, it not intuitively clear that this program is correct, warranting the use of our partial-correctness calculus to prove its correctness. Testing the program with a few examples is not sufficient to find all mistakes, however, and the reader would rightly not be convinced that this program really does compute the minimal-sum section in all cases. So let us try to use the partial-correctness calculus introduced in this chapter to prove it.

We formalise our requirement of the program as two specifications[6], written as Hoare triples.

S1. $(\!|\top|\!)$ `Min_Sum` $(\!|\forall i, j\,(0 \leq i \leq j < n \rightarrow s \leq S_{i,j})|\!)$.
It says that, after the program terminates, s is less than or equal to, the sum of any section of the array. Note that i and j are logical variables in that they don't occur as program variables.

S2. $(\!|\top|\!)$ `Min_Sum` $(\!|\exists i, j\,(0 \leq i \leq j < n \wedge s = S_{i,j})|\!)$,
which says that there is a section whose sum is s.

If there is a section whose sum is s and no section has a sum less than s, then s is the sum of a minimal-sum section: the 'conjunction' of **S1** and **S2** give us the property we want.

Let us first prove **S1**. This begins with seeking a suitable invariant. As always, the following characteristics of invariants are a useful guide:

- Invariants express the fact that the computation performed so far by the while-statement is correct.
- Invariants typically have the same form as the desired postcondition of the while-statement.
- Invariants express relationships between the variables manipulated by the while-statement which are re-established each time the body of the while-statement is executed.

A suitable invariant in this case appears to be

$$\texttt{Inv1}(s, k) \stackrel{\text{def}}{=} \forall i, j\,(0 \leq i \leq j < k \rightarrow s \leq S_{i,j}) \tag{4.12}$$

since it says that s is less than, or equal to, the minimal sum observed up to the current stage of the computation, represented by k. Note that it has the same form as the desired postcondition: we replaced the n by k, since the final value of k is n. Notice that i and j are quantified in the formula, because they are logical variables; k is a program variable. This justifies the notation $\texttt{Inv1}(s, k)$ which highlights that the formula has only the program variables s and k as free variables and is similar to the use of `fun`-statements in Alloy in Chapter 2.

If we start work on producing a proof tableau with this invariant, we will soon find that it is not strong enough to do the job. Intuitively, this is because it ignores the value of t, which stores the minimal sum of all sections ending just before $a[k]$, which is crucial in the idea behind the program. A suitable invariant expressing that t is correct up to the current point of the

[6] The notation $\forall i, j$ abbreviates $\forall i \forall j$, and similarly for $\exists i, j$.

```
    (|⊤|)
    (|Inv1(a[0], 1) ∧ Inv2(a[0], 1)|)                     Implied
  k = 1;
    (|Inv1(a[0], k) ∧ Inv2(a[0], k)|)                     Assignment
  t = a[0];
    (|Inv1(a[0], k) ∧ Inv2(t, k)|)                        Assignment
  s = a[0];
    (|Inv1(s, k) ∧ Inv2(t, k)|)                           Assignment
  while (k != n) {
        (|Inv1(s, k) ∧ Inv2(t, k) ∧ k ≠ n|)               Invariant Hyp. ∧ guard
        (|Inv1(min(s, min(t + a[k], a[k])), k + 1)
          ∧Inv2(min(t + a[k], a[k]), k + 1)|)             Implied (Lemma 4.20)
      t = min(t + a[k], a[k]);
        (|Inv1(min(s, t), k + 1) ∧ Inv2(t, k + 1)|)       Assignment
      s = min(s,t);
        (|Inv1(s, k + 1) ∧ Inv2(t, k + 1)|)               Assignment
      k = k + 1;
        (|Inv1(s, k) ∧ Inv2(t, k)|)                       Assignment
  }
    (|Inv1(s, k) ∧ Inv2(t, k) ∧ ¬¬(k = n)|)               Partial-while
    (|Inv1(s, n)|)                                        Implied
```

Figure 4.3. Tableau proof for specification **S1** of `Min_Sum`.

computation is

$$\texttt{Inv2}(t, k) \stackrel{\text{def}}{=} \forall i\,(0 \leq i < k \rightarrow t \leq S_{i,k-1}) \tag{4.13}$$

saying that t is not greater than the sum of any section ending in $a[k-1]$. Our invariant is the conjunction of these formulas, namely

$$\texttt{Inv1}(s, k) \wedge \texttt{Inv2}(t, k). \tag{4.14}$$

The completed proof tableau of **S1** for `Min_Sum` is given in Figure 4.3. The tableau is constructed by

- Proving that the candidate invariant (4.14) is indeed an invariant. This involves pushing it upwards through the body of the while-statement and showing that what emerges follows from the invariant and the boolean guard. This non-trivial implication is shown in the proof of Lemma 4.20.
- Proving that the invariant, together with the negation of the boolean guard, is strong enough to prove the desired postcondition. This is the last implication of the proof tableau.

- Proving that the invariant is established by the code before the while-statement. We simply push it upwards through the three initial assignments and check that the resulting formula is implied by the precondition of the specification, here $\top$.

As so often the case, in constructing the tableau, we find that two formulas meet; and we have to prove that the first one implies the second one. Sometimes this is easy and we can just note the implication in the tableau. For example, we readily see that $\top$ implies $\texttt{Inv1}(a[0], 1) \wedge \texttt{Inv2}(a[0], 1)$: k being 1 forces i and j to be zero in order that the assumptions in $\texttt{Inv1}(a[0], k)$ and $\texttt{Inv2}(a[0], k)$ be true. But this means that their conclusions are true as well. However, the proof obligation that the invariant hypothesis imply the precondition computed within the body of the while-statement reveals the complexity and ingenuity of this program and its justification needs to be taken off-line:

Lemma 4.20 Let s and t be any integers, n the length of the array `a`, and k an index of that array in the range of $0 < k < n$. Then $\texttt{Inv1}(s, k) \wedge \texttt{Inv2}(t, k) \wedge k \neq n$ implies

1. $\texttt{Inv1}(\min(s, \min(t + a[k], a[k])), k + 1)$ as well as
2. $\texttt{Inv2}(\min(t + a[k], a[k]), k + 1)$.

Proof:

1. Take any i with $0 \leq i < k + 1$; we will prove that $\min(t + a[k], a[k]) \leq S_{i,k}$. If $i < k$, then $S_{i,k} = S_{i,k-1} + a[k]$, so what we have to prove is $\min(t + a[k], a[k]) \leq S_{i,k-1} + a[k]$; but we know $t \leq S_{i,k-1}$, so the result follows by adding $a[k]$ to each side. Otherwise, $i = k$, $S_{i,k} = a[k]$ and the result follows.
2. Take any i and j with $0 \leq i \leq j < k + 1$; we prove that $\min(s, t + a[k], a[k]) \leq S_{i,j}$. If $i \leq j < k$, then the result is immediate. Otherwise, $i \leq j = k$ and the result follows from part 1 of the lemma.

$\square$

4.4 Proof calculus for total correctness

In the preceding section, we developed a calculus for proving *partial* correctness of triples $(\!|\phi|\!)\ P\ (\!|\psi|\!)$. In that setting, proofs come with a disclaimer: *only if* the program P terminates an execution does a proof of $\vdash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ tell us anything about that execution. Partial correctness does not tell us anything if P 'loops' indefinitely. In this section, we extend our proof calculus for partial correctness so that it also proves that programs terminate. In the previous section, we already pointed out that only the syntactic construct `while` B $\{C\}$ could be responsible for non-termination.

Therefore, the proof calculus for total correctness is the same as for partial correctness for all the rules except the rule for while-statements.

A proof of total correctness for a while-statement will consist of two parts: the proof of partial correctness and a proof that the given while-statement terminates. Usually, it is a good idea to prove partial correctness first since this often provides helpful insights for a termination proof. However, some programs require termination proofs as premises for establishing *partial* correctness, as can be seen in exercise 1(d) on page 303.

The proof of termination usually has the following form. We identify an integer expression whose value can be shown to *decrease* every time we execute the body of the while-statement in question, but which is always non-negative. If we can find an expression with these properties, it follows that the while-statement must terminate; because the expression can only be decremented a finite number of times before it becomes 0. That is because there is only a finite number of integer values between 0 and the initial value of the expression.

Such integer expressions are called *variants*. As an example, for the program `Fac1` of Example 4.2, a suitable variant is $x - z$. The value of this expression is decremented every time the body of the while-statement is executed. When it is 0, the while-statement terminates.

We can codify this intuition in the following rule for total correctness which replaces the rule for the while statement:

$$\frac{(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\ C\ (\!|\eta \wedge 0 \leq E < E_0|\!)}{(\!|\eta \wedge 0 \leq E|\!)\ \texttt{while}\ B\ \{C\}\ (\!|\eta \wedge \neg B|\!)}\ \textsf{Total-while}. \tag{4.15}$$

In this rule, E is the expression whose value decreases with each execution of the body C. This is coded by saying that, if its value equals that of the logical variable E_0 before the execution of C, then it is strictly less than E_0 after it – yet still it remains non-negative. As before, η is the invariant.

We use the rule Total-while in tableaux similarly to how we use Partial-while, but note that the body of the rule C must now be shown to satisfy

$$(\!|\eta \wedge B \wedge 0 \leq E = E_0|\!)\ C\ (\!|\eta \wedge 0 \leq E < E_0|\!).$$

When we push $\eta \wedge 0 \leq E < E_0$ upwards through the body, we have to prove that what emerges from the top is implied by $\eta \wedge B \wedge 0 \leq E = E_0$; and the weakest precondition for the entire while-statement, which gets written above that while-statement, is $\eta \wedge 0 \leq E$.

Let us illustrate this rule by proving that $\vdash_{\mathsf{tot}} (\!|x \geq 0|\!)\ \texttt{Fac1}\ (\!|y = x!|\!)$ is valid, where `Fac1` is given in Example 4.2, as follows:

```
y = 1;
z = 0;
while (x != z) {
    z = z + 1;
    y = y * z;
}
```

As already mentioned, $x - z$ is a suitable variant. The invariant $(y = z!)$ of the partial correctness proof is retained. We obtain the following complete proof for total correctness:

$(\!|x \geq 0|\!)$	
$(\!|1 = 0! \wedge 0 \leq x - 0|\!)$	Implied
`y = 1;`	
$(\!|y = 0! \wedge 0 \leq x - 0|\!)$	Assignment
`z = 0;`	
$(\!|y = z! \wedge 0 \leq x - z|\!)$	Assignment
`while (x != z) {`	
$(\!|y = z! \wedge x \neq z \wedge 0 \leq x - z = E_0|\!)$	Invariant Hyp. $\wedge$ guard
$(\!|y \cdot (z+1) = (z+1)! \wedge 0 \leq x - (z+1) < E_0|\!)$	Implied
`z = z + 1;`	
$(\!|y \cdot z = z! \wedge 0 \leq x - z < E_0|\!)$	Assignment
`y = y * z;`	
$(\!|y = z! \wedge 0 \leq x - z < E_0|\!)$	Assignment
`}`	
$(\!|y = z! \wedge x = z|\!)$	Total-while
$(\!|y = x!|\!)$	Implied

and so $\vdash_{\mathsf{tot}} (\!|x \geq 0|\!)\ \texttt{Fac1}\ (\!|y = x!|\!)$ is valid. Two comments are in order:

- Notice that the precondition $x \geq 0$ is crucial in securing the fact that $0 \leq x - z$ holds right before the while-statements gets executed: it implies the precondition $1 = 0! \wedge 0 \leq x - 0$ computed by our proof. In fact, observe that `Fac1` does not terminate if x is negative initially.
- The application of Implied within the body of the while-statement is valid, but it makes vital use of the fact that the boolean guard is true. This is an example of a while-statement whose boolean guard is needed in reasoning about the correctness of *every* iteration of that while-statement.

One may wonder whether there is a program that, given a while-statement and a precondition as input, decides whether that while-statement terminates on all runs whose initial states satisfy that precondition. One can prove that there cannot be such a program. This suggests that the automatic extraction of useful termination expressions E cannot be realized either. Like most other such universal problems discussed in this text, the wish to completely mechanise such decision or extraction procedures cannot be realised. Hence, finding a working variant E is a creative activity which requires skill, intuition and practice.

Let us consider an example program, `Collatz`, that conveys the challenge one may face in finding suitable termination variants E:

```
c = x;
while (c != 1) {
  if (c % 2 == 0) { c = c / 2; }
  else { c = 3*c + 1; }
}
```

This program records the initial value of `x` in `c` and then iterates an if-statement until, and if, the value of `c` equals 1. The if-statement tests whether `c` is even – divisible by 2 – if so, `c` stores its current value divided by 2; if not, `c` stores 'three times its current value plus 1.' The expression `c / 2` denotes integer division, so `11 / 2` renders `5` as does `10 / 2`.

To get a feel for this algorithm, consider an execution trace in which the value of `x` is 5: the value of `c` evolves as `5 16 8 4 2 1`. For another example, if the value of `x` is initially 172, the evolution of `c` is

```
172 86 43 130 65 196 98 49 148 74 37 112 56 28 14 7 22
11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

This execution requires 32 iterations of the while-statement to reach a terminating state in which the value of `c` equals 1. Notice how this trace reaches 5, from where on the continuation is as if 5 were the initial value of `x`.

For the initial value 123456789 of `x` we abstract the evolution of `c` with $+$ (its value increases in the else-branch) and $-$ (its value decreases in the if-branch):

```
+ - - - - - - + - - - + - + - - + - + - + - + - + - + - - + - - -
- + - - - - + - - + - - + - - + - + - - - + - + - - - - - + - - +
- + - - + - - - - + - - - - - - + - - + - + - - + - + - + - - + -
+ - + - + - - + - - - + - + - + - - + - + - - + - + - + - + - + -
+ - - - + - + - + - + - - - - + - - + - - + - - - - + - - - + - +
- + - - - - - + - - - -
```

This requires 177 iterations of the while-statement to reach a terminating state. Although it is re-assuring that some program runs terminate, the irregular pattern of $+$ and $-$ above make it seem very hard, if not impossible, to come up with a variant that proves the termination of `Collatz` on all executions in which the initial value of `x` is positive.

Finally, let's consider a *really big* integer:

```
32498723462509735034567279652376420563047563456356347563\\
96598734085384756074086560785607840745067340563457640875\\
62984573756306537856405634056245634578692825623542135761\\
9519765129854122965424895465956457
```

where \\ denotes concatenation of digits. Although this is a very large number indeed, our program `Collatz` requires only 4940 iterations to terminate. Unfortunately, nobody knows a suitable variant for this program that could prove the validity of $\vdash_{\mathsf{tot}} (\!| 0 < x |\!)$ `Collatz` $(\!| \top |\!)$. Observe how the use of $\top$ as a postcondition emphasizes that this Hoare triple is merely concerned about program termination as such. Ironically, there is also no known initial value of `x` greater than 0 for which `Collatz` doesn't terminate. In fact, things are even subtler than they may appear: if we replace `3*c + 1` in `Collatz` with a different such linear expression in `c`, the program may not terminate despite meeting the precondition $0 < x$; see exercise 6 on page 303.

4.5 Programming by contract

For a valid sequent $\vdash_{\mathsf{tot}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$, the triple $(\!| \phi |\!)\ P\ (\!| \psi |\!)$ may be seen as a *contract* between a supplier and a consumer of a program P. The supplier insists that consumers run P only on initial state satisfies ϕ. In that case, the supplier promises the consumer that the final state of that run satisfies ψ. For a valid $\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$, the latter guarantee applies only when a run terminates.

For imperative programming, the validation of Hoare triples can be interpreted as the validation of contracts for method or procedure calls. For example, our program fragment `Fac1` may be the `...` in the method body

```
int factorial (x: int) { ... return y; }
```

The code for this method can be annotated with its contractual assumptions and guarantees. These annotations can be checked off-line by humans, during compile-time or even at run-time in languages such as Eiffel. A possible format for such contracts for the method `factorial` is given in Figure 4.4.

```
method name:            factorial
input:                  x ofType int
assumes:                0 <= x
guarantees:             y = x!
output:                 ofType int
modifies only:          y
```

Figure 4.4. A contract for the method `factorial`.

The keyword `assumes` states all preconditions, the keyword `guarantees` lists all postconditions. The keyword `modifies only` specifies which program variables may change their value during an execution of this method.

Let us see why such contracts are useful. Suppose that your boss tells you to write a method that computes $\binom{n}{k}$ – read 'n choose k' – a notion of combinatorics where $1/\binom{49}{6}$ is your change of getting all six lottery numbers right out of 49 numbers total. Your boss also tells you that

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} \tag{4.16}$$

holds. The method `factorial` and its contract (Figure 4.4) is at your disposal. Using (4.16) you can quickly compute some values, such as $\binom{5}{2} = 5!/(2! \cdot 3!) = 10$, $\binom{10}{0} = 1$, and $\binom{49}{6} = 13983816$. You then write a method `choose` that makes calls to the method `factorial`, e.g. you may write

```
int choose(n : int, k : int) {
  return factorial(n) / (factorial(k) * factorial (n - k));
}
```

This method body consists of a `return`-statement only which makes three calls to method `factorial` and then computes the result according to (4.16). So far so good. But programming by contract is not just about writing programs, it is also about writing the *contracts* for such programs! The static information about `choose` – e.g. its name – are quickly filled into that contract. But what about the preconditions (`assumes`) and postconditions (`guarantees`)?

At the very least, you must state preconditions that ensure that all method calls within this method's body satisfy *their* preconditions. In this case, we only call `factorial` whose precondition is that its input value be non-negative. Therefore, we require that n, k, and $n - k$ be non-negative. The latter says that n is not smaller than k.

What about the postconditions of `choose`? Since the method body declared no local variables, we use `result` to denote the return value of this

method. The postcondition then states that `result` equals $\binom{n}{k}$ – assuming that you boss' equation (4.16) is correct for your preconditions $0 \leq k$, $0 \leq n$, and $k \leq n$. The contract for `choose` is therefore

```
method name:                choose
input:                      n ofType int, k ofType int
assumes:                    0 <= k, 0 <= n, k <= n
guarantees:                 result = 'n choose k'
output:                     ofType int
modifies only local variables
```

From this we learn that programming by contract uses contracts

1. as assume-guarantee abstract interfaces to methods;
2. to specify their method's header information, output type, when calls to its method are 'legal,' what variables that method modifies, and what its output satisfies on all 'legal' calls;
3. to enable us to prove the validity of a contract C for method `m` by ensuring that all method calls within `m`'s body meet the preconditions of these methods and using that all such calls then meet their respective postconditions.

Programming by contract therefore gives rise to *program validation by contract*. One proves the 'Hoare triple' $(\!|\texttt{assume}|\!)\ \texttt{method}\ (\!|\texttt{guarantee}|\!)$ very much in the style developed in this chapter, except that for all method invocations within that body we can assume that *their* Hoare triples are correct.

Example 4.21 We have already used program validation by contract in our verification of the program that computes the minimal sum for all sections of an array in Figure 4.3 on page 291. Let us focus on the proof fragment

$$(\!|\texttt{Inv1}(\min(s, \min(t + a[k], a[k])), k + 1) \wedge \texttt{Inv2}(\min(t + a[k], a[k]), k + 1)|\!)$$

Implied (Lemma 4.20)

```
t = min(t + a[k], a[k]);
```

$(\!|\texttt{Inv1}(\min(s, t), k + 1) \wedge \texttt{Inv2}(t, k + 1)|\!)$ Assignment

```
s = min(s,t);
```

$(\!|\texttt{Inv1}(s, k + 1) \wedge \texttt{Inv2}(t, k + 1)|\!)$ Assignment

Its last line serves as the postcondition which gets pushed through the assignment `s = min(s,t)`. But `min(s,t)` is a method call whose guarantees are specified as '`result` equals $\min(s, t)$,' where $\min(s, t)$ is a mathematical notation for the smaller of the numbers s and t. Thus, the rule Assignment does not substitute the syntax of the method invocation `min(s,t)` for all occurrences of s in $\texttt{Inv1}(s, k + 1) \wedge \texttt{Inv2}(t, k + 1)$, but changes all such s to the guarantee $\min(s, t)$ of the method call `min(s,t)` – program validation

by contract in action! A similar comment applies for the assignment `t = min(t + a[k], a[k])`.

Program validation by contract has to be used wisely to avoid circular reasoning. If each method is a node in a graph, let's draw an edge from method `n` to method `m` iff within the body of `n` there is a call to method `m`. For program validation by contract to be sound, we require that there be no cycles in this method-dependency graph.

4.6 Exercises

Exercises 4.1

* 1. If you already have written computer programs yourself, assemble for each programming language you used a list of features of its software development environment (compiler, editor, linker, run-time environment etc) that may improve the likelihood that your programs work correctly. Try to rate the effectiveness of each such feature.

2. Repeat the previous exercise by listing and rating features that may decrease the likelihood of procuding correct and reliable programs.

Exercises 4.2

* 1. In what circumstances would `if` (B) $\{C_1\}$ `else` $\{C_2\}$ fail to terminate?

* 2. A familiar command missing from our language is the for-statement. It may be used to sum the elements in an array, for example, by programming as follows:

```
s = 0;
for (i = 0; i <= max; i = i+1) {
   s = s + a[i];
}
```

After performing the initial assignment `s = 0`, this executes `i = 0` first, then executes the body `s = s + a[i]` and the incrementation `i = i + 1` continually until `i <= max` becomes false. Explain how `for` $(C_1; B; C_2)$ $\{C_3\}$ can be defined as a derived program in our core language.

3. Suppose that you need a language construct `repeat` $\{C\}$ `until` (B) which repeats C until B becomes true, i.e.
   i. executes C in the current state of the store;
   ii. evaluates B in the resulting state of the store;
   iii. if B is false, the program resumes with (i); otherwise, the program `repeat` $\{C\}$ `until` (B) terminates.

   This construct sometimes allows more elegant code than a corresponding while-statement.

(a) Define `repeat` C `until` B as a derived expression using our core language.
(b) Can one define every `repeat` expression in our core language extended with for-statements? (You might need the empty command `skip` which does nothing.)

Exercises 4.3

1. For any store l as in Example 4.4 (page 264), determine which of the relations below hold; justify your answers:
   * (a) $l \vDash (x + y < z) \to \neg(x * y = z)$
   (b) $l \vDash \forall u\,(u < y) \lor (u * z < y * z)$
   * (c) $l \vDash x + y - z < x * y * z$.
* 2. For any ϕ, ψ and P explain why $\vDash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds whenever the relation $\vDash_{\mathsf{tot}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds.
3. Let the relation $P \vdash l \leadsto l'$ hold iff P's execution in store l terminates, resulting in store l'. Use this formal judgment $P \vdash l \leadsto l'$ along with the relation $l \vDash \phi$ to define $\vDash_{\mathsf{par}}$ and $\vDash_{\mathsf{tot}}$ symbolically.
4. Another reason for proving partial correctness in isolation is that some program fragments have the form `while (true) {C}`. Give useful examples of such program fragments in application programming.
* 5. Use the proof rule for assignment and logical implication as appropriate to show the validity of
   (a) $\vdash_{\mathsf{par}} (\!|x > 0|\!)$ `y = x + 1` $(\!|y > 1|\!)$
   (b) $\vdash_{\mathsf{par}} (\!|\top|\!)$ `y = x; y = x + x + y` $(\!|y = 3 \cdot x|\!)$
   (c) $\vdash_{\mathsf{par}} (\!|x > 1|\!)$ `a = 1; y = x; y = y - a` $(\!|y > 0 \land x > y|\!)$.
* 6. Write down a program P such that
   (a) $(\!|\top|\!)\ P\ (\!|y = x + 2|\!)$
   (b) $(\!|\top|\!)\ P\ (\!|z > x + y + 4|\!)$
   holds under partial correctness; then prove that this is so.
7. For all instances of Implied in the proof on page 274, specify their corresponding $\vdash_{\mathrm{AR}}$ sequents.
8. There is a safe way of relaxing the format of the proof rule for assignment: as long as no variable occurring in E gets updated in between the assertion $\psi[E/x]$ and the assignment `x = E` we may conclude ψ right after this assignment. Explain why such a proof rule is sound.
9. (a) Show, by means of an example, that the 'reversed' version of the rule Implied

$$\frac{\vdash_{\mathrm{AR}} \phi \to \phi' \qquad (\!|\phi|\!)\ C\ (\!|\psi|\!) \qquad \vdash_{\mathrm{AR}} \psi' \to \psi}{(\!|\phi'|\!)\ C\ (\!|\psi'|\!)}\ \text{Implied_Reversed}$$

   is unsound for partial correctness.
   (b) Explain why the modified rule If-Statement in (4.7) is sound with respect to the partial and total satisfaction relation.

* (c) Show that any instance of the modified rule If-Statement in a proof can be replaced by an instance of the original If-statement and instances of the rule Implied. Is the converse true as well?

* 10. Prove the validity of the sequent $\vdash_{\mathsf{par}} (\!|\top|\!)\ P\ (\!|z = \min(x, y)|\!)$, where $\min(x, y)$ is the smallest number of x and y – e.g. $\min(7, 3) = 3$ – and the code of P is given by

```
if (x > y) {
   z = y;
} else {
   z = x;
}
```

11. For each of the specifications below, write code for P and prove the partial correctness of the specified input/output behaviour:
   * (a) $(\!|\top|\!)\ P\ (\!|z = \max(w, x, y)|\!)$, where $\max(w, x, y)$ denotes the largest of w, x and y.
   * (b) $(\!|\top|\!)\ P\ (\!|((x = 5) \to (y = 3)) \wedge ((x = 3) \to (y = -1))|\!)$.

12. Prove the validity of the sequent $\vdash_{\mathsf{par}} (\!|\top|\!)$ `Succ` $(\!|y = x + 1|\!)$ without using the modified proof rule for if-statements.

* 13. Show that $\vdash_{\mathsf{par}} (\!|x \geq 0|\!)$ `Copy1` $(\!|x = y|\!)$ is valid, where `Copy1` denotes the code

```
a = x;
y = 0;
while (a != 0) {
   y = y + 1;
   a = a - 1;
}
```

* 14. Show that $\vdash_{\mathsf{par}} (\!|y \geq 0|\!)$ `Multi1` $(\!|z = x \cdot y|\!)$ is valid, where `Multi1` is:

```
a = 0;
z = 0;
while (a != y) {
  z = z + x;
  a = a + 1;
}
```

15. Show that $\vdash_{\mathsf{par}} (\!|y = y_0 \wedge y \geq 0|\!)$ `Multi2` $(\!|z = x \cdot y_0|\!)$ is valid, where `Multi2` is:

```
z = 0;
while (y != 0) {
  z = z + x;
  y = y - 1;
}
```

16. Show that $\vdash_{\mathsf{par}} (\!|x \geq 0|\!)$ `Copy2` $(\!|x = y|\!)$ is valid, where `Copy2` is:

```
y = 0;
while (y != x) {
  y = y + 1;
}
```

17. The program Div is supposed to compute the dividend of integers x by y; this is defined to be the unique integer d such that there exists some integer r – the remainder – with $r < y$ and $x = d \cdot y + r$. For example, if $x = 15$ and $y = 6$, then $d = 2$ because $15 = 2 \cdot 6 + 3$, where $r = 3 < 6$. Let Div be given by:

```
r = x;
d = 0;
while (r >= y) {
  r = r - y;
  d = d + 1;
}
```

Show that $\vdash_{\mathsf{par}} (\!|\neg(y = 0)|\!)$ Div $(\!|(x = d \cdot y + r) \wedge (r < y)|\!)$ is valid.

* 18. Show that $\vdash_{\mathsf{par}} (\!|x \geq 0|\!)$ Downfac $(\!|y = x!|\!)$ is valid[7], where Downfac is:

```
a = x;
y = 1;
while (a > 0) {
  y = y * a;
  a = a - 1;
}
```

19. Why can, or can't, you prove the validity of $\vdash_{\mathsf{par}} (\!|\top|\!)$ Copy1 $(\!|x = y|\!)$?
20. Let all while-statements while (B) {C} in P be annotated with invariant candidates η at the and of their bodies, and $\eta \wedge B$ at the beginning of their body.
    (a) Explain how a proof of $\vdash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ can be automatically reduced to showing the validity of some $\vdash_{\mathrm{AR}} \psi_1 \wedge \cdots \wedge \psi_n$.
    (b) Identify such a sequent $\vdash_{\mathrm{AR}} \psi_1 \wedge \cdots \wedge \psi_n$ for the proof in Example 4.17 on page 287.
21. Given $n = 5$ test the correctness of Min_Sum on the arrays below:
    * (a) $[-3, 1, -2, 1, -8]$
    (b) $[1, 45, -1, 23, -1]$
    * (c) $[-1, -2, -3, -4, 1097]$.
22. If we swap the first and second assignment in the while-statement of Min_Sum, so that it first assigns to s and then to t, is the program still correct? Justify your answer.

* 23. Prove the partial correctness of **S2** for Min_Sum.

24. The program Min_Sum does not reveal where a minimal-sum section may be found in an input array. Adapt Min_Sum to achieve that. Can you do this with a single pass through the array?
25. Consider the proof rule

$$\frac{(\!|\phi|\!)\ C\ (\!|\psi_1|\!) \qquad (\!|\phi|\!)\ C\ (\!|\psi_2|\!)}{(\!|\phi|\!)\ C\ (\!|\psi_1 \wedge \psi_2|\!)}\ \mathsf{Conj}$$

[7] You may have to strengthen your invariant.

for Hoare triples.

(a) Show that this proof rule is sound for $\vDash_{\mathsf{par}}$.
(b) Derive this proof rule from the ones on page 270.
(c) Explain how this rule, or its derived version, is used to establish the overall correctness of `Min_Sum`.

26. The maximal-sum problem is to compute the maximal sum of all sections on an array.
(a) Adapt the program from page 289 so that it computes the maximal sum of these sections.
(b) Prove the partial correctess of your modified program.
(c) Which aspects of the correctness proof given in Figure 4.3 (page 291) can be 're-used?'

Exercises 4.4

1. Prove the validity of the following total-correctness sequents:
* (a) $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `Copy1` $(\!| x = y |\!)$
* (b) $\vdash_{\mathsf{tot}} (\!| y \geq 0 |\!)$ `Multi1` $(\!| z = x \cdot y |\!)$
(c) $\vdash_{\mathsf{tot}} (\!| (y = y_0) \wedge (y \geq 0) |\!)$ `Multi2` $(\!| z = x \cdot y_0 |\!)$
* (d) $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `Downfac` $(\!| y = x! |\!)$
* (e) $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `Copy2` $(\!| x = y |\!)$, does your invariant have an active part in securing correctness?
(f) $\vdash_{\mathsf{tot}} (\!| \neg(y = 0) |\!)$ `Div` $(\!| (x = d \cdot y + r) \wedge (r < y) |\!)$.
2. Prove total correctness of **S1** and **S2** for `Min_Sum`.
3. Prove that $\vdash_{\mathsf{par}}$ is sound for $\vDash_{\mathsf{par}}$. Just like in Section 1.4.3, it suffices to assume that the premises of proof rules are instances of $\vDash_{\mathsf{par}}$. Then, you need to prove that their respective conclusion must be an instance of $\vDash_{\mathsf{par}}$ as well.
4. Prove that $\vdash_{\mathsf{tot}}$ is sound for $\vDash_{\mathsf{tot}}$.
5. Implement program `Collatz` in a programming language of your choice such that the value of `x` is the program's input and the final value of `c` its output. Test your program on a range of inputs. Which is the biggest integer for which your program terminates without raising an exception or dumping the core?
6. A function over integers $f\colon \mathbb{I} \to \mathbb{I}$ is affine iff there are integers a and b such that $f(x) = a \cdot x + b$ for all $x \in \mathbb{I}$. The else-branch of the program `Collatz` assigns to `c` the value $f(c)$, where f is an affine function with $a = 3$ and $b = 1$.
(a) Write an parameterized implementation of `Collatz` in which you can initially specify the values of a and b either statically or through keyboard input such that the else-branch assigns to `c` the value of $f(c)$.
(b) Determine for which pairs $(a, b) \in \mathbb{I} \times \mathbb{I}$ the set $\mathrm{Pos} \stackrel{\mathrm{def}}{=} \{x \in \mathbb{I} \mid 0 < x\}$ is invariant under the affine function $f(x) = a \cdot x + b$: for all $x \in \mathrm{Pos}$, $f(x) \in \mathrm{Pos}$.
* (c) Find an affine function that leaves Pos invariant, but not the set $\mathrm{Odd} \stackrel{\mathrm{def}}{=} \{x \in \mathbb{I} \mid \exists y \in \mathbb{I}\colon x = 2 \cdot y + 1\}$, such that there is an input drawn from Pos whose

execution with the modified `Collatz` program eventually enters a cycle, and therefore does not terminate.

Exercises 4.5

1. Consider methods of the form `boolean certify_V(c : Certificate)` which return `true` iff the certificate `c` is judged valid by the verifier `V`, a class in which method `certify_V` resides.
   * (a) Discuss how programming by contract can be used to delegate the judgment of a certificate to another verifier.
   * (b) What potential problems do you see in this context if the resulting method-dependency graph is circular?

* 2. Consider the method

```
boolean withdraw(amount: int) {
  if (amount < 0 && isGood(amount))
     { balance = balance - amount;
        return true;
     } else { return false; }
}
```

named `withdraw` which attempts to withdraw `amount` from an integer field `balance` of the class within which method `withdraw` lives. This method makes use of another method `isGood` which returns `true` iff the value of `balance` is greater or equal to the value of `amount`.
   (a) Write a contract for method `isGood`.
   (b) Use that contract to show the validity of the contract for `withdraw`:

```
method name:               withdraw
input:                     amount of Type int
assumes:                   0 <= balance
guarantees:                0 <= balance
output:                    of Type boolean
modifies only:             balance
```

   Notice that the precondition and postcondition of this contract are the same and refer to a field of the method's object. Upon validation, this contract establishes that all calls to `withdraw` leave (the 'object invariant') `0 <= balance` invariant.

4.7 Bibliographic notes

An early exposition of the program logics for partial and total correctness of programs written in an imperative while-language can be found in [Hoa69]. The text [Dij76] contains a formal treatment of weakest preconditions.

Backhouse's book [Bac86] describes program logic and weakest preconditions and also contains numerous examples and exercises. Other books giving more complete expositions of program verification than we can in this chapter are [AO91, Fra92]; they also extend the basic core language to include features such as procedures and parallelism. The issue of writing to arrays and the problem of array cell aliasing are described in [Fra92]. The original article describing the minimal-sum section problem is [Gri82]. A gentle introduction to the mathematical foundations of functional programming is [Tur91]. Some web sites deal with software liability and possible standards for intellectual property rights applied to computer programs[8] [9]. Text books on systematic programming language design by uniform extensions of the core language we presented at the beginning of this chapter are [Ten91, Sch94]. A text on functional programming on the freely available language Standard ML of New Jersey is [Pau91].

[8] `www.opensource.org`
[9] `www.sims.berkeley.edu/~pam/papers.html`

5
Modal logics and agents

5.1 Modes of truth

In propositional or predicate logic, formulas are either true, or false, in any model. Propositional logic and predicate logic do not allow for any further possibilities. From many points of view, however, this is inadequate. In natural language, for example, we often distinguish between various 'modes' of truth, such as *necessarily true*, *known to be true*, *believed to be true* and *true in the future*. For example, we would say that, although the sentence

George W. Bush is president of the United States of America.

is currently true, it will not be true at some point in the future. Equally, the sentence

There are nine planets in the solar system.

while true, and maybe true for ever in the future, is not necessarily true, in the sense that it could have been a different number. However, the sentence

The cube root of 27 is 3.

as well as being true is also necessarily true and true in the future. It does not enjoy all modes of truth, however. It may not be known to be true by some people (children, for example); it may not be believed by others (if they are mistaken).

In computer science, it is often useful to reason about modes of truth. In Chapter 3, we studied the logic CTL in which we could distinguish not only between truth at different points in the future, but also between different futures. Temporal logic is thus a special case of modal logic. The modalities of CTL allow us to express a host of computational behaviour of systems. Modalities are also extremely useful in modelling other domains of computer science. In artificial intelligence, for example, scenarios with several

interacting agents are developed. Each agent may have different knowledge about the environment and also about the knowledge of other agents. In this chapter, we will look in depth at modal logics applied to reasoning about knowledge.

Modal logic adds unary connectives to express one, or more, of these different modes of truth. The simplest modal logics just deal with one concept – such as knowledge, necessity, or time. More sophisticated modal logics have connectives for expressing several modes of truth in the same logic; we will see some of these towards the end of this chapter.

We take a *logic engineering* approach in this chapter, in which we address the following question: given a particular mode of truth, how may we develop a logic capable of expressing and formalising that concept? To answer this question, we need to decide what properties the logic should have and what examples of reasoning it should be able to express. Our main case study will be the logic of *knowledge in a multi-agent system*. But first, we look at the syntax and semantics of basic modal logic.

5.2 Basic modal logic

5.2.1 Syntax

The language of basic modal logic is that of propositional logic with two extra connectives, $\Box$ and $\Diamond$. Like negation ($\neg$), they are *unary* connectives as they apply themselves to a single formula only. As done in Chapters 1 and 3, we write $p, q, r, p_3 \ldots$ to denote atomic formulas.

Definition 5.1 The formulas of basic modal logic ϕ are defined by the following Backus Naur form (BNF):

$$\phi ::= \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \mid (\phi \leftrightarrow \phi) \mid (\Box\phi) \mid (\Diamond\phi) \tag{5.1}$$

where p is any atomic formula.

Example formulas of basic modal logic are $(p \wedge \Diamond(p \to \Box\neg r))$ and $\Box((\Diamond q \wedge \neg r) \to \Box p)$, having the parse trees shown in Figure 5.1. The following strings are *not* formulas, because they cannot be constructed using the grammar in (5.1): $(p\Box \to q)$ and $(p \to \Diamond(q \Diamond r))$.

Convention 5.2 As done in Chapter 1, we assume that the unary connectives ($\neg$, $\Box$ and $\Diamond$) bind most closely, followed by $\wedge$ and $\vee$ and then followed by $\to$ and $\leftrightarrow$.

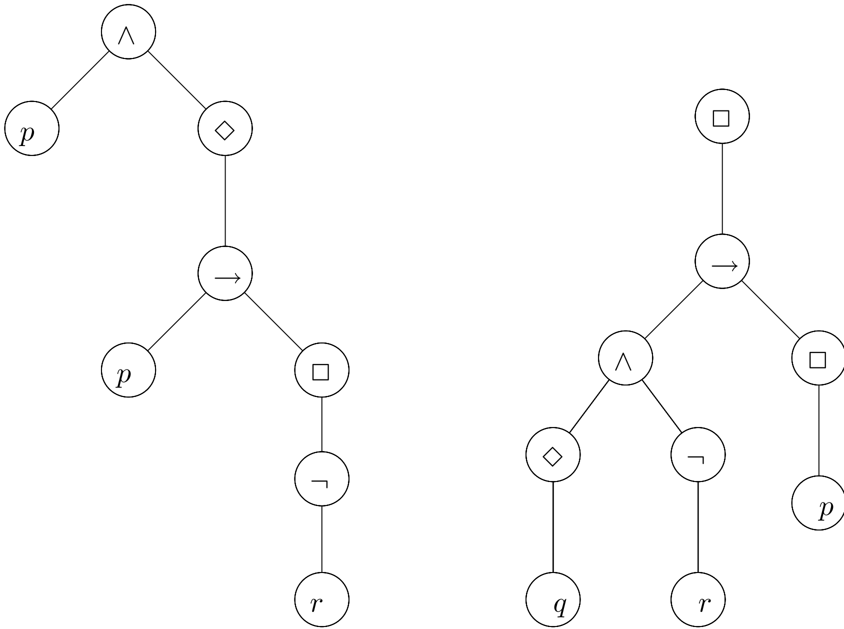

Figure 5.1. Parse trees for $(p \wedge \Diamond(p \to \Box\neg r))$ and $\Box((\Diamond q \wedge \neg r) \to \Box p)$.

This convention allows us to remove many sets of brackets, retaining them only to avoid ambiguity, or to override these binding priorities. For example, $\Box((\Diamond q \wedge \neg r) \to \Box p)$ can be written $\Box(\Diamond q \wedge \neg r \to \Box p)$. We cannot omit the remaining brackets, however, for $\Box\Diamond q \wedge \neg r \to \Box p$ has quite a different parse tree (see Figure 5.2) from the one in Figure 5.1.

In basic modal logic, $\Box$ and $\Diamond$ are read 'box' and 'diamond,' but, when we apply modal logics to express various modes of truth, we may read them appropriately. For example, in the logic that studies necessity and possibility, $\Box$ is read 'necessarily' and $\Diamond$ 'possibly;' in the logic of agent Q's knowledge, $\Box$ is read 'agent Q knows' and $\Diamond$ is read 'it is consistent with agent Q's knowledge that,' or more colloquially, 'for all Q knows.' We will see why these readings are appropriate later in the chapter.

5.2.2 Semantics

For a formula of propositional logic, a model is simply an assignment of truth values to each of the atomic formulas present in that formula – we called such models valuation in Chapter 1. However, this notion of model is inadequate for modal logic, since we want to distinguish between different modes, or degrees, of truth.

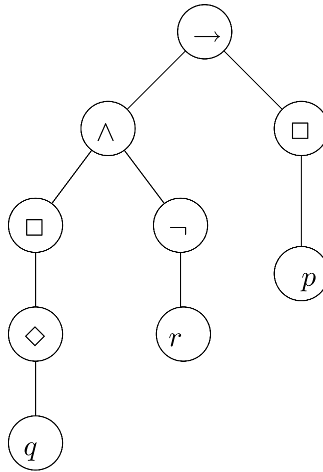

Figure 5.2. The parse tree for $\Box\Diamond q \wedge \neg r \to \Box p$.

Definition 5.3 A model $\mathcal{M}$ of basic modal logic is specified by three things:

1. A set W, whose elements are called worlds;
2. A relation R on W ($R \subseteq W \times W$), called the accessibility relation;
3. A function $L : W \to \mathcal{P}(\texttt{Atoms})$, called the labelling function.

We write $R(x, y)$ to denote that (x, y) is in R.

These models are often called *Kripke models*, in honour of S. Kripke who invented them and worked extensively in modal logic in the 1950s and 1960s. Intuitively, $w \in W$ stands for a possible world and $R(w, w')$ means that w' is a world *accessible from* world w. The actual nature of that relationship depends on what we intend to model. Although the definition of models looks quite complicated, we can use an easy graphical notation to depict finite models. We illustrate the graphical notation by an example. Suppose W equals $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ and the relation R is given as follows:

- $R(x_1, x_2)$, $R(x_1, x_3)$, $R(x_2, x_2)$, $R(x_2, x_3)$, $R(x_3, x_2)$, $R(x_4, x_5)$, $R(x_5, x_4)$, $R(x_5, x_6)$; and no other pairs are related by R.

Suppose further that the labelling function behaves as follows:

x	x_1	x_2	x_3	x_4	x_5	x_6
$L(x)$	$\{q\}$	$\{p, q\}$	$\{p\}$	$\{q\}$	$\emptyset$	$\{p\}$

Then, the Kripke model is illustrated in Figure 5.3. The set W is drawn as a set of circles, with arrows between them showing the relation R. Within each circle is the value of the labelling function in that world. If you have read Chapter 3, then you might have noticed that Kripke structures are also the models for CTL, where W is S, the set of states; R is $\rightarrow$, the relation of state transitions; and L is the labelling function.

Definition 5.4 Let $\mathcal{M} = (W, R, L)$ be a model of basic modal logic. Suppose $x \in W$ and ϕ is a formula of (5.1). We will define when formula ϕ is true in the world x. This is done via a satisfaction relation $x \Vdash \phi$ by structural induction on ϕ:

$$
\begin{array}{rl}
x \Vdash \top & \\
x \nVdash \bot & \\
x \Vdash p & \text{iff } p \in L(x) \\
x \Vdash \neg\phi & \text{iff } x \nVdash \phi \\
x \Vdash \phi \wedge \psi & \text{iff } x \Vdash \phi \text{ and } x \Vdash \psi \\
x \Vdash \phi \vee \psi & \text{iff } x \Vdash \phi \text{, or } x \Vdash \psi \\
x \Vdash \phi \rightarrow \psi & \text{iff } x \Vdash \psi \text{, whenever we have } x \Vdash \phi \\
x \Vdash \phi \leftrightarrow \psi & \text{iff } (x \Vdash \phi \text{ iff } x \Vdash \psi) \\
x \Vdash \Box\psi & \text{iff, for each } y \in W \text{ with } R(x, y) \text{, we have } y \Vdash \psi \\
x \Vdash \Diamond\psi & \text{iff there is a } y \in W \text{ such that } R(x, y) \text{ and } y \Vdash \psi.
\end{array}
$$

When $x \Vdash \phi$ holds, we say 'x satisfies ϕ,' or 'ϕ is true in world x.' We write $\mathcal{M}, x \Vdash \phi$ if we want to stress that $x \Vdash \phi$ holds in the model $\mathcal{M}$.

The first two clauses just express the fact that $\top$ is always true, while $\bot$ is always false. Next, we see that $L(x)$ is the set of all the atomic formulas that are true at x. The clauses for the boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$) should also be straightforward: they mean that we apply the usual truth-table semantics of these connectives in the current world x. The interesting cases are those for $\Box$ and $\Diamond$. For $\Box\phi$ to be true at x, we require that ϕ be true in all the worlds accessible by R from x. For $\Diamond\phi$, it is required that there is at least one accessible world in which ϕ is true. Thus, $\Box$ and $\Diamond$ are a bit like the quantifiers $\forall$ and $\exists$ of predicate logic, except that they do not take variables as arguments. This fact makes them conceptually much simpler than quantifiers. The modal operators $\Box$ and $\Diamond$ are also rather like AX and EX in CTL – see Section 3.4.1. Note that the meaning of $\phi_1 \leftrightarrow \phi_2$ coincides with that of $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$; we call it 'if and only if.'

Definition 5.5 A model $\mathcal{M} = (W, R, L)$ of basic modal logic is said to satisfy a formula if every state in the model satisfies it. Thus, we write $\mathcal{M} \vDash \phi$ iff, for each $x \in W$, $x \Vdash \phi$.

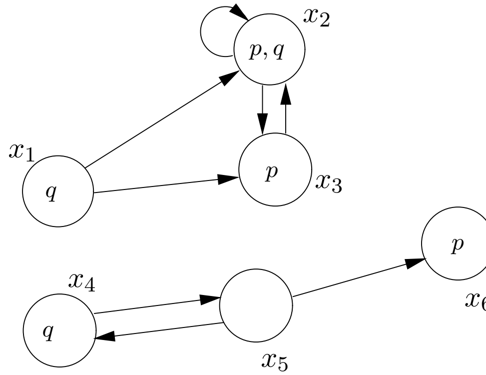

Figure 5.3. A Kripke model.

Examples 5.6 Consider the Kripke model of Figure 5.3. We have:

- $x_1 \Vdash q$, since $q \in L(x_1)$.
- $x_1 \Vdash \Diamond q$, for there is a world accessible from x_1 (namely, x_2) which satisfies q. In mathematical notation: $R(x_1, x_2)$ and $x_2 \Vdash q$.
- $x_1 \nVdash \Box q$, however. This is because $x_1 \Vdash \Box q$ says that all worlds accessible from x_1 (i.e. x_2 and x_3) satisfy q; but x_3 does not.
- $x_5 \nVdash \Box p$ and $x_5 \nVdash \Box q$. Moreover, $x_5 \nVdash \Box p \vee \Box q$. However, $x_5 \Vdash \Box(p \vee q)$.
  To see these facts, note that the worlds accessible from x_5 are x_4 and x_6. Since $x_4 \nVdash p$, we have $x_5 \nVdash \Box p$; and since $x_6 \nVdash q$, we have $x_5 \nVdash \Box q$. Therefore, we get that $x_5 \nVdash \Box p \vee \Box q$. However, $x_5 \Vdash \Box(p \vee q)$ holds because, in each of x_4 and x_6, we find p or q.
- The worlds which satisfy $\Box p \to p$ are x_2, x_3, x_4, x_5 and x_6; for x_2, x_3 and x_6 this is so since they already satisfy p; for x_4 this is true since it does not satisfy $\Box p$ – we have $R(x_4, x_5)$ and x_5 does not satisfy p; a similar reason applies to x_5. As for x_1, it cannot satisfy $\Box p \to p$ since it satisfies $\Box p$ but not p itself.

Worlds like x_6 that have no world accessible to them deserve special attention in modal logic. Observe that $x_6 \nVdash \Diamond \phi$, no matter what ϕ is, because $\Diamond \phi$ says 'there is an accessible world which satisfies ϕ.' In particular, 'there is an accessible world,' which in the case of x_6 there is not. Even when ϕ is $\top$, we have $x_6 \nVdash \Diamond \top$. So, although $\top$ is satisfied in every world, $\Diamond \top$ is not necessarily. In fact, $x \Vdash \Diamond \top$ holds iff x has at least one accessible world.

A dual situation exists for the satisfaction of $\Box \phi$ in worlds with no accessible world. No matter what ϕ is, we find that $x_6 \Vdash \Box \phi$ holds. That is because $x_6 \Vdash \Box \phi$ says that ϕ is true in all worlds accessible from x_6. There are no such worlds, so ϕ is vacuously true in all of them: there is simply nothing to check. This reading of 'for all accessible worlds' may seem surprising, but it secures the de Morgan rules for the box and diamond modalities shown

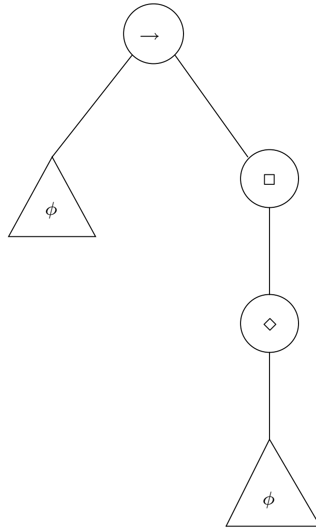

Figure 5.4. The parse tree of the formula scheme $\phi \to \Box\Diamond\phi$.

below. Even $\Box\bot$ is true in x_6. If you wanted to convince someone that $\Box\bot$ was not true in x_6, you'd have to show that there is a world accessible from x_6 in which $\bot$ is not true; but you can't do this, for there are no worlds accessible from x_6. So again, although $\bot$ is false in every world, $\Box\bot$ might not be false. In fact, $x \Vdash \Box\bot$ holds iff x has no accessible worlds.

Formulas and formula schemes The grammar in (5.1) specifies exactly the formulas of basic modal logic, given a set of atomic formulas. For example, $p \to \Box\Diamond p$ is such a formula. It is sometimes useful to talk about a whole family of formulas which have the same 'shape;' these are called *formula schemes*. For example, $\phi \to \Box\Diamond\phi$ is a formula scheme. Any formula which has the shape of a certain formula scheme is called an *instance* of the scheme. For example,

- $p \to \Box\Diamond p$
- $q \to \Box\Diamond q$
- $(p \wedge \Diamond q) \to \Box\Diamond(p \wedge \Diamond q)$

are all instances of the scheme $\phi \to \Box\Diamond\phi$. An example of a formula scheme of propositional logic is $\phi \wedge \psi \to \psi$. We may think of a formula scheme as an under-specified parse tree, where certain portions of the tree still need to be supplied – e.g. the tree of $\phi \to \Box\Diamond\phi$ is found in Figure 5.4.

Semantically, a scheme can be thought of as the conjunction of all its instances – since there are generally infinitely many such instances, this cannot be carried out syntactically! We say that a world/model satisfies a scheme if it satisfies all its instances. Note that an instance being satisfied in a Kripke model does not imply that the whole scheme is satisfied. For example, we may have a Kripke model in which all worlds satisfy $\neg p \lor q$, but at least one world does not satisfy $\neg q \lor p$; the scheme $\neg\phi \lor \psi$ is not satisfied.

Equivalences between modal formulas

Definition 5.7 1. We say that a set of formulas Γ of basic modal logic semantically entails a formula ψ of basic modal logic if, in any world x of any model $\mathcal{M} = (W, R, L)$, we have $x \Vdash \psi$ whenever $x \Vdash \phi$ for all $\phi \in \Gamma$. In that case, we say that $\Gamma \vDash \psi$ holds.

2. We say that ϕ and ψ are semantically equivalent if $\phi \vDash \psi$ and $\psi \vDash \phi$ hold. We denote this by $\phi \equiv \psi$.

Note that $\phi \equiv \psi$ holds iff any world in any model which satisfies one of them also satisfies the other. The definition of semantic equivalence is based on semantic entailment in the same way as the corresponding one for formulas of propositional logic. However, the underlying notion of semantic entailment for modal logic is quite different, as we will see shortly.

Any equivalence in propositional logic is also an equivalence in modal logic. Indeed, if we take any equivalence in propositional logic and substitute the atoms uniformly for any modal logic formula, the result is also an equivalence in modal logic. For example, take the equivalent formulas $p \to \neg q$ and $\neg(p \land q)$ and now perform the substitution

$$\begin{array}{rcl} p & \mapsto & \Box p \land (q \to p) \\ q & \mapsto & r \to \Diamond(q \lor p). \end{array}$$

The result of this substitution is the pair of formulas

$$\begin{array}{l} \Box p \land (q \to p) \to \neg(r \to \Diamond(q \lor p)) \\ \neg((\Box p \land (q \to p)) \land (r \to \Diamond(q \lor p))) \end{array} \tag{5.2}$$

which are equivalent as formulas of basic modal logic.

We have already noticed that $\Box$ is a universal quantifier on accessible worlds and $\Diamond$ is the corresponding existential quantifier. In view of these facts, it is not surprising to find that de Morgan rules apply for $\Box$ and $\Diamond$:

$$\neg\Box\phi \equiv \Diamond\neg\phi \text{ and } \neg\Diamond\phi \equiv \Box\neg\phi.$$

Moreover, $\Box$ distributes over $\wedge$ and $\Diamond$ distributes over $\vee$:

$$\Box(\phi \wedge \psi) \equiv \Box\phi \wedge \Box\psi \text{ and } \Diamond(\phi \vee \psi) \equiv \Diamond\phi \vee \Diamond\psi.$$

These equivalences correspond closely to the quantifier equivalences discussed in Section 2.3.2. It is also not surprising to find that $\Box$ does *not* distribute over $\vee$ and $\Diamond$ does *not* distribute over $\wedge$, i.e. we do not have equivalences between $\Box(\phi \vee \psi)$ and $\Box\phi \vee \Box\psi$, or between $\Diamond(\phi \wedge \psi)$ and $\Diamond\phi \wedge \Diamond\psi$. For example, in the fourth item of Example 5.6 we had $x_5 \Vdash \Box(p \vee q)$ and $x_5 \nVdash \Box p \vee \Box q$.

Note that $\Box\top$ is equivalent to $\top$, but *not* to $\Diamond\top$, as we saw earlier. Similarly, $\Diamond\bot \equiv \bot$ but they are not equivalent to $\Box\bot$.

Another equivalence is $\Diamond\top \equiv \Box p \to \Diamond p$. For suppose $x \Vdash \Diamond\top$ – i.e. x has an accessible world, say y – and suppose $x \Vdash \Box p$; then $y \Vdash p$, so $x \Vdash \Diamond p$. Conversely, suppose $x \Vdash \Box p \to \Diamond p$; we must show it satisfies $\Diamond\top$. Let us distinguish between the cases $x \Vdash \Box p$ and $x \nVdash \Box p$; in the former, we get $x \Vdash \Diamond p$ from $x \Vdash \Box p \to \Diamond p$ and so x must have an accessible world; and in the latter, x must again have an accessible world in order to avoid satisfying $\Box p$. Either way, x has an accessible world, i.e. satisfies $\Diamond\top$. Naturally, this argument works for any formula ϕ, not just an atom p.

Valid formulas

Definition 5.8 A formula ϕ of basic modal logic is said to be valid if it is true in every world of every model, i.e. iff $\vDash \phi$ holds.

Any propositional tautology is a valid formula and so is any substitution instance of it. A substitution instance of a formula is the result of uniformly substituting the atoms of the formula by other formulas as done in (5.2). For example, since $p \vee \neg p$ is a tautology, performing the substitution $p \mapsto \Box p \wedge (q \to p)$ gives us a valid formula $(\Box p \wedge (q \to p)) \vee \neg(\Box p \wedge (q \to p))$.

As we may expect from equivalences above, these formulas are valid:

$$\begin{aligned} &\neg\Box\phi \leftrightarrow \Diamond\neg\phi \\ &\Box(\phi \wedge \psi) \leftrightarrow \Box\phi \wedge \Box\psi \\ &\Diamond(\phi \vee \psi) \leftrightarrow \Diamond\phi \vee \Diamond\psi. \end{aligned} \tag{5.3}$$

To prove that the first of these is valid, we reason as follows. Suppose x is a world in a model $\mathcal{M} = (W, R, L)$. We want to show $x \Vdash \neg\Box\phi \leftrightarrow \Diamond\neg\phi$, i.e. that $x \Vdash \neg\Box\phi$ iff $x \Vdash \Diamond\neg\phi$. Well, using Definition 5.4,

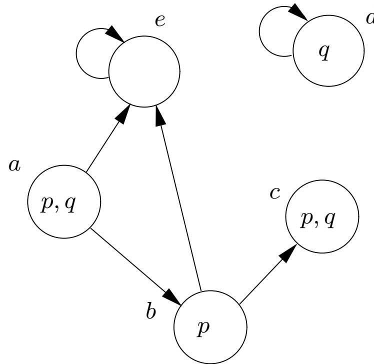

Figure 5.5. Another Kripke model.

$x \Vdash \neg\Box\phi$
  iff it isn't the case that $x \Vdash \Box\phi$
  iff it isn't the case that, for all y such that $R(x, y)$, $y \Vdash \phi$
  iff there is some y such that $R(x, y)$ and not $y \Vdash \phi$
  iff there is some y such that $R(x, y)$ and $y \Vdash \neg\phi$
  iff $x \Vdash \Diamond\neg\phi$.

Proofs that the other two are valid are similarly routine and left as exercises.

Another important formula which can be seen to be valid is the following:

$$\Box(\phi \to \psi) \land \Box\phi \to \Box\psi.$$

It is sometimes written in the equivalent, but slightly less intuitive, form $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$. This formula scheme is called K in most books about modal logic, honouring the logician S. Kripke who, as we mentioned earlier, invented the so-called 'possible worlds semantics' of Definition 5.4.

To see that K is valid, again suppose we have some world x in some model $\mathcal{M} = (W, R, L)$. We have to show that $x \Vdash \Box(\phi \to \psi) \land \Box\phi \to \Box\psi$. Again referring to Definition 5.4, we assume that $x \Vdash \Box(\phi \to \psi) \land \Box\phi$ and try to prove that $x \Vdash \Box\psi$:

$x \Vdash \Box(\phi \to \psi) \land \Box\phi$
  iff $x \Vdash \Box(\phi \to \psi)$ and $x \Vdash \Box\phi$
  iff for all y with $R(x, y)$, we have $y \Vdash \phi \to \psi$ and $y \Vdash \phi$
    implies that, for all y with $R(x, y)$, we have $y \Vdash \psi$
  iff $x \Vdash \Box\psi$.

There aren't any other interesting valid formulas in basic modal logic. Later, we will see additional valid formulas in extended modal logics of interest.

5.3 Logic engineering

Having looked at the framework for basic modal logic, we turn now to how one may formalise the different modes of truth discussed at the beginning of this chapter. The basic framework is quite general and can be refined in various ways to give us the properties appropriate for the intended applications. Logic engineering is the subject of engineering logics to fit new applications. It is potentially a very broad subject, drawing on all branches of logic, computer science and mathematics. In this chapter, however, we are restricting ourselves to the particular engineering of *modal* logics.

We will consider how to re-engineer basic modal logic to fit the following readings of $\Box\phi$:

- It is necessarily true that ϕ
- It will always be true that ϕ
- It ought to be that ϕ
- Agent Q believes that ϕ
- Agent Q knows that ϕ
- After any execution of program P, ϕ holds.

As modal logic automatically gives us the connective $\Diamond$, which is equivalent to $\neg\Box\neg$, we can find out what the corresponding readings of $\Diamond$ in our system will be. For example, 'it is *not* necessarily true that *not* ϕ' means that it is possibly true that ϕ. You could work this out in steps:

It is *not* necessarily true that ϕ
    = it is possible that *not* ϕ.

Therefore,

It is *not* necessarily true that *not* ϕ
    = it is possible that *not not* ϕ
    = it is possible that ϕ.

Let us work this out with the reading 'agent Q knows ϕ' for $\Box\phi$. Then, $\Diamond\phi$ is read as

agent Q does *not* know *not* ϕ
    = as far as Q's knowledge is concerned, ϕ could be the case
    = ϕ is consistent with what agent Q knows
    = for all agent Q knows, ϕ.

The readings for $\Diamond$ for the other modes are given in Table 5.6.

Table 5.6. The readings of $\Diamond$ corresponding to each reading of $\Box$.

$\Box\phi$	$\Diamond\phi$
It is necessarily true that ϕ	It is possibly true that ϕ
It will always be true that ϕ	Sometime in the future ϕ
It ought to be that ϕ	It is permitted to be that ϕ
Agent Q believes that ϕ	ϕ is consistent with Q's beliefs
Agent Q knows that ϕ	For all Q knows, ϕ
After any execution of program P, ϕ holds	After some execution of P, ϕ holds

5.3.1 The stock of valid formulas

We saw in the last section some valid formulas of basic modal logic, such as instances of the axiom scheme K: $\Box(\phi \to \psi) \to (\Box\phi \to \Box\psi)$ and of the schemes in (5.3). Many other formulas, such as

- $\Box p \to p$
- $\Box p \to \Box\Box p$
- $\neg\Box p \to \Box\neg\Box p$
- $\Diamond\top$

are *not* valid. For example, for each one of these, there is a world in the Kripke model of Figure 5.3 which does not satisfy the formula. The world x_1 satisfies $\Box p$, but it does not satisfy p, so it does not satisfy $\Box p \to p$. If we add $R(x_2, x_1)$ to our model, then x_1 still satisfies $\Box p$ but does not satisfy $\Box\Box p$. Thus, x_1 fails to satisfy $\Box p \to \Box\Box p$. If we change $L(x_4)$ to $\{p, q\}$, then x_4 does not satisfy $\neg\Box p \to \Box\neg\Box p$, because it satisfies $\neg\Box p$, but it does not satisfy $\Box\neg\Box p$ – the path $R(x_4, x_5)R(x_5, x_4)$ serves as a counter example. Finally, x_6 does not satisfy $\Diamond\top$, for this formula states that there is an accessible world satisfying $\top$, which is not the case.

If we are to build a logic capturing the concept of necessity, however, we must surely have that $\Box p \to p$ is valid; for anything which is *necessarily true* is also simply true. Similarly, we would expect $\Box p \to p$ to be valid in the case that $\Box p$ means 'agent Q knows p,' for anything which is known must also be true. We cannot *know* something which is false. We can, however, *believe* falsehoods, so in the case of a logic of belief, we would *not* expect $\Box p \to p$ to be valid.

Part of the job of logic engineering is to determine what formula schemes should be valid and to craft the logic in such a way that precisely those ones are valid.

Table 5.7 shows six interesting readings for $\Box$ and eight formula schemes. For each reading and each formula scheme, we decide whether we should expect the scheme to be valid. Notice that we should only put a tick if the

$\Box\phi$	$\Box\phi \to \phi$	$\Box\phi \to \Box\Box\phi$	$\Diamond\phi \to \Box\Diamond\phi$	$\Diamond\top$	$\Box\phi \to \Diamond\phi$	$\Box\phi \lor \Box\neg\phi$	$\Box(\phi \to \psi) \land \Box\phi \to \Box\psi$	$\Diamond\phi \land \Diamond\psi \to \Diamond(\phi \land \psi)$
It is necessarily true that ϕ	√	√	√	√	√	×	√	×
It will always be true that ϕ	×	√	×	×	×	×	√	×
It ought to be that ϕ	×	×	×	√	√	×	√	×
Agent Q believes that ϕ	×	√	√	√	√	×	√	×
Agent Q knows that ϕ	√	√	√	√	√	×	√	×
After any execut'n of prgrm P, ϕ holds	×	×	×	×	×	×	√	×

Table 5.7. Which formula schemes should hold for these readings of $\Box$?

formula should be valid for all cases of ϕ and ψ. If it could be valid for some cases, but not for others, we put a cross.

There are many points worth noting about Table 5.7. First, observe that it is rather debatable whether to put a tick, or a cross, in some of the cells. We need to be precise about the concept of truth we are trying to formalise, in order to resolve any ambiguity.

Necessity. When we ask ourselves whether $\Box\phi \to \Box\Box\phi$ and $\Diamond\phi \to \Box\Diamond\phi$ should be valid, it seems to depend on what notion of necessity we are referring to. These formulas are valid if that which is necessary is *necessarily* necessary. If we are dealing with *physical necessity*, then this amounts to: are the laws of the universe themselves physically necessary, i.e. do they entail that they should be the laws of the universe? The answer seems to be no. However, if we meant *logical necessity*, it seems that we should give the answer yes, for the laws of logic are meant to be those assertions whose truth cannot be denied. The row is filled on the understanding that we mean logical necessity.

Always in the future. We must be precise about whether or not the future includes the present; this is precisely what the formula $\Box\phi \to \phi$ states. It is a matter of convention whether the future includes the present, or not. In Chapter 3, we saw that CTL adopts the convention that it does. For variety, therefore, let us assume that the future does not include the present in this row of the table. That means that $\Box\phi \to \phi$ fails. What about $\Diamond\top$? It says that there is a future world in which $\top$ is true. In particular, then, there is a future world, i.e. time has no end. Whether we regard this as true or not depends on exactly what notion of 'the future' we are trying to model. We assumed the validity of $\Diamond\top$

in Chapter 3 on CTL since this resulted in an easier presentation of our model-checking algorithms, but we might choose to model it otherwise, as in Table 5.7.

Ought. In this case the formulas $\Box\phi \rightarrow \Box\Box\phi$ and $\Diamond\phi \rightarrow \Box\Diamond\phi$ state that the moral codes we adopt are themselves forced upon us by morality. This seems not to be the case; for example, we may believe that *'It ought to be the case that we wear a seat-belt,'* but this does not compel us to believe that *'It ought to be the case that we ought to wear a seat-belt.'* However, anything which ought to be so should be permitted to be so; therefore, $\Box\phi \rightarrow \Diamond\phi$.

Belief. To decide whether $\Diamond\top$, let us express it as $\neg\Box\bot$, for this is semantically equivalent. It says that agent Q does not believe any contradictions. Here we must be precise about whether we are modelling human beings, with all their foibles and often plainly contradictory beliefs, or whether we are modelling idealised agents that are logically omniscient – i.e. capable of working out the logical consequences of their beliefs. We opt to model the latter concept. The same issue arises when we consider, for example, $\Diamond\phi \rightarrow \Box\Diamond\phi$, which – when we rewrite it as $\neg\Box\psi \rightarrow \Box\neg\Box\psi$ – says that, if agent Q doesn't believe something, then he believes that he doesn't believe it. Validity of the formula $\Box\phi \vee \Box\neg\phi$ would mean that Q has an opinion on every matter; we suppose this is unlikely. What about $\Diamond\phi \wedge \Diamond\psi \rightarrow \Diamond(\phi \wedge \psi)$? Let us rewrite it as $\neg\Diamond(\phi \wedge \psi) \rightarrow \neg(\Diamond\phi \wedge \Diamond\psi)$, i.e. $\Box(\neg\phi \vee \neg\psi) \rightarrow (\Box\neg\phi \vee \Box\neg\psi)$ or – if we subsume the negations into the ϕ and ψ – the formula $\Box(\phi \vee \psi) \rightarrow (\Box\phi \vee \Box\psi)$. This seems not to be valid, for agent Q may be in a situation in which she or he believes that there is a key in the red box, or in the green box, without believing that it is in the red box and also without believing that it is in the green box.

Knowledge. It seems to differ from belief only in respect of the first formula in Table 5.7; while agent Q can have false beliefs, he can only *know* that which is true. In the case of knowledge, the formulas $\Box\phi \rightarrow \Box\Box\phi$ and $\neg\Box\psi \rightarrow \Box\neg\Box\psi$ are called *positive introspection* and *negative introspection*, respectively, since they state that the agent can introspect upon her knowledge; if she knows something, she knows that she knows it; and if she does not know something, she again knows that she doesn't know it. Clearly, this represents *idealised* knowledge, since most humans – with all their hang-ups and infelicities – do not satisfy these properties. The formula scheme K is sometimes referred to as *logical omniscience* in the logic of knowledge, since it says that the agent's knowledge is closed under logical consequence. This means that the agent knows all the

consequences of anything he knows, which is unfortunately (or fortunately?) true only for idealised agents, not humans.

Execution of programs. Not many of our formulas seem to hold in this case. The scheme $\Box\phi \to \Box\Box\phi$ says that running the program twice is the same as running it once, which is plainly wrong in the case of a program which deducts money from your bank account. The formula $\Diamond\top$ says that there is an execution of the program which terminates; this is false for some programs.

The formula schemes $\Diamond\top$ and $\Box\phi \to \Diamond\phi$ were seen to be equivalent in the preceding section and, indeed, we see that they get the same pattern of ticks and crosses. We can also show that $\Box\phi \to \phi$ entails $\Diamond\top$ – i.e. $(\Box\phi \to \phi) \to \Diamond\top$ is valid – so whenever the former gets a tick, so should the latter. This is indeed the case, as you can verify in Table 5.7.

5.3.2 Important properties of the accessibility relation

So far, we have been engineering logics at the level of deciding what formulas should be valid for the various readings of $\Box$. We can also engineer logics at the level of Kripke models. For each of our six readings of $\Box$, there is a corresponding reading of the accessibility relation R which will then suggest that R enjoys certain properties such as reflexivity or transitivity.

Let us start with necessity. The clauses

$$
\begin{array}{ll}
x \Vdash \Box\psi & \text{iff for each } y \in W \text{ with } R(x,y) \text{ we have } y \Vdash \psi \\
x \Vdash \Diamond\psi & \text{iff there is a } y \in W \text{ such that } R(x,y) \text{ and } y \Vdash \psi
\end{array}
$$

from Definition 5.4 tell us that ϕ is necessarily true at x if ϕ is true in all worlds y accessible from x in a certain way; but accessible in what way? Intuitively, necessarily ϕ is true if ϕ is true in all *possible* worlds; so $R(x,y)$ should be interpreted as meaning that y is a possible world according to the information in x.

In the case of knowledge, we think of $R(x,y)$ as saying: y could be the actual world according to agent Q's knowledge at x. In other words, if the actual world is x, then agent Q – who is not omniscient – cannot rule out the possibility of it being y. If we plug this definition into the clause above for $x \Vdash \Box\phi$, we find that agent Q knows ϕ iff ϕ is true in all the worlds that, for all he knows, could be the actual world. The meaning of R for each of the six readings of $\Box$ is shown in Table 5.8.

Recall that a given binary relation R may be:

- *reflexive*: if, for every $x \in W$, we have $R(x,x)$;
- *symmetric*: if, for every $x, y \in W$, we have $R(x,y)$ implies $R(y,x)$;

Table 5.8. For each reading of $\Box$, the meaning of R is given.

$\Box\phi$	$R(x, y)$
It is necessarily true that ϕ	y is possible world according to the information at x
It will always be true that ϕ	y is a future world of x
It ought to be that ϕ	y is an acceptable world according to the information at x
Agent Q believes that ϕ	y could be the actual world according to Q's beliefs at x
Agent Q knows that ϕ	y could be the actual world according to Q's knowledge at x
After any execution of P, ϕ holds	y is a possible resulting state after execution of P at x

- *serial*: if, for every x there is a y such that $R(x, y)$;
- *transitive*: if, for every $x, y, z \in W$, we have $R(x, y)$ and $R(y, z)$ imply $R(x, z)$;
- *Euclidean*: if, for every $x, y, z \in W$ with $R(x, y)$ and $R(x, z)$, we have $R(y, z)$;
- *functional*: if, for each x there is a unique y such that $R(x, y)$;
- *linear*: if, for every $x, y, z \in W$, we have that $R(x, y)$ and $R(x, z)$ together imply that $R(y, z)$, or y equals z, or $R(z, y)$;
- *total*: if for every $x, y \in W$ we have $R(x, y)$ or $R(y, x)$; and
- an *equivalence relation*: if it is reflexive, symmetric and transitive.

Now, let us consider this question: according to the various readings of R, which of these properties do we expect R to have?

Example 5.9 If $\Box\phi$ means 'agent Q knows ϕ,' then $R(x, y)$ means y could be the actual world according to Q's knowledge at x.

- Should R be reflexive? This would say: x could be the actual world according to Q's knowledge at x. In other words, Q cannot know that things are different from how they really are – i.e., Q cannot have false knowledge. This is a desirable property for R to have. Moreover, it seems to rest on the same intuition – i.e. the impossibility of false knowledge – as the validity of the formula $\Box\phi \to \phi$. Indeed, the validity of this formula and the property of reflexivity are closely related, as we see later on.
- Should R be transitive? It would say: if y is possible according to Q's knowledge at x and z is possible according to her knowledge at y, then z is possible according to her knowledge at x.

  Well, this seems to be true. For suppose it was not true, i.e. at x she knew something preventing z from being the real world. Then, she would know she knew this thing at x; therefore, she would know something at y which prevented z from being the real world; which contradicts our premise.

In this argument, we relied on positive introspection, i.e. the formula $\Box\phi \to \Box\Box\phi$. Again, we will shortly see that there is a close correspondence between R being transitive and the validity of this formula.

5.3.3 Correspondence theory

We saw in the preceding section that there appeared to be a correspondence between the validity of $\Box\phi \to \phi$ and the property that the accessibility relation R is reflexive. The connection between them is that both relied on the intuition that anything which is known by an agent is true. Moreover, there also seemed to be a correspondence between $\Box\phi \to \Box\Box\phi$ and R being transitive; they both seem to assert the property of *positive introspection*, i.e. that which is known is known to be known.

In this section, we will see that there is a precise mathematical relationship between these formulas and properties of R. Indeed, to every formula scheme there corresponds a property of R. From the point of view of logic engineering, it is important to see this relationship, because it helps one to understand the logic being studied. For example, if you believe that a certain formula scheme should be accepted in the system of modal logic you are engineering, then it is well worth looking at the corresponding property of R and checking that this property makes sense for the application, too. Alternatively, the meaning of some formulas may seem difficult to understand, so looking at their corresponding properties of R can help.

To state the relationship between formula schemes and their corresponding properties, we need the notion of a (modal) frame.

Definition 5.10 A frame $\mathcal{F} = (W, R)$ is a set W of worlds and a binary relation R on W.

A frame is like a Kripke model (Definition 5.3), except that it has no labelling function. From any model we can extract a frame, by just forgetting about the labelling function; for example, Figure 5.9 shows the frame extracted from the Kripke model of Figure 5.3. A frame is just a set of worlds and an accessibility relationship between them. It has no information about what atomic formulas are true at the various worlds. However, it is useful to say sometimes that the frame, as a whole, satisfies a formula. This is defined as follows.

Definition 5.11 A frame $\mathcal{F} = (W, R)$ satisfies a formula of basic modal logic ϕ if, for each labelling function $L : W \to \mathcal{P}(\mathtt{Atoms})$ and each $w \in W$,

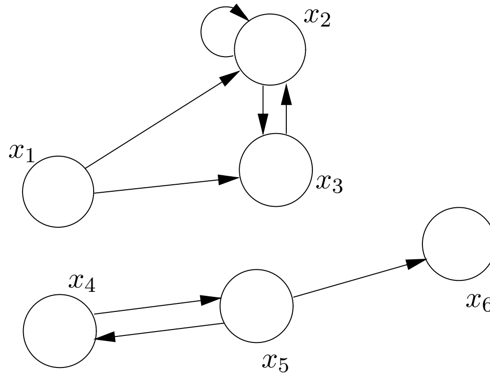

Figure 5.9. The frame of the model in Figure 5.3.

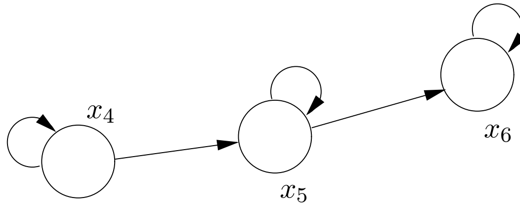

Figure 5.10. Another frame.

the relation $\mathcal{M}, w \Vdash \phi$ holds, where $\mathcal{M} = (W, R, L)$ – recall the definition of $\mathcal{M}, w \Vdash \phi$ on page 310. In that case, we say that $\mathcal{F} \vDash \phi$ holds.

One can show that, if a frame satisfies a formula, then it also satisfies every substitution instance of that formula. Conversely, if a frame satisfies an instance of a formula scheme, it satisfies the whole scheme. This contrasts markedly with models. For example, the model of Figure 5.3 satisfies $p \vee \Diamond p \vee \Diamond\Diamond p$, but doesn't satisfy every instance of $\phi \vee \Diamond\phi \vee \Diamond\Diamond\phi$; for example, x_6 does not satisfy $q \vee \Diamond q \vee \Diamond\Diamond q$. Since frames don't contain any information about the truth or falsity of propositional atoms, they can't distinguish between different atoms; so, if a frame satisfies a formula, it also satisfies the formula scheme obtained by substituting its atoms $p, q, \ldots$ by $\phi, \psi, \ldots$

Examples 5.12 Consider the frame $\mathcal{F}$ in Figure 5.10.

1. $\mathcal{F}$ satisfies the formula $\Box p \to p$. To see this, we have to consider any labelling function of the frame – there are eight such labelling functions, since p could be true or false in each of the three worlds – and show that each world satisfies the formula for each labelling. Rather than really doing this literally, let us

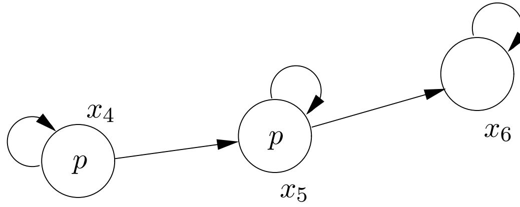

Figure 5.11. A model.

give a generic argument: let x be any world. Suppose that $x \Vdash \Box p$; we want to show $x \Vdash p$. We know that $R(x, x)$ because each x is accessible from itself in the diagram; so, it follows from the clause for $\Box$ in Definition 5.4 that $x \Vdash p$.
2. Therefore, our frame $\mathcal{F}$ satisfies any formula of this shape, i.e. it satisfies the formula scheme $\Box\phi \rightarrow \phi$.
3. The frame does not satisfy the formula $\Box p \rightarrow \Box\Box p$. For suppose we take the labelling of Figure 5.11; then $x_4 \Vdash \Box p$, but $x_4 \nVdash \Box\Box p$.

If you think about why the frame of Figure 5.10 satisfied $\Box p \rightarrow p$ and why it did not satisfy $\Box p \rightarrow \Box\Box p$, you will probably guess the following:

Theorem 5.13 Let $\mathcal{F} = (W, R)$ be a frame.

1. The following statements are equivalent:
   - R is reflexive;
   - $\mathcal{F}$ satisfies $\Box\phi \rightarrow \phi$;
   - $\mathcal{F}$ satisfies $\Box p \rightarrow p$;
2. The following statements are equivalent:
   - R is transitive;
   - $\mathcal{F}$ satisfies $\Box\phi \rightarrow \Box\Box\phi$;
   - $\mathcal{F}$ satisfies $\Box p \rightarrow \Box\Box p$.

PROOF: Each item 1 and 2 requires us to prove three things: (a) that, if R has the property, then the frame satisfies the formula scheme; and (b) that, if the frame satisfies the formula scheme, it satisfies the instance of it; and (c) that, if the frame satisfies a formula instance, then R has the property.

1. (a) Suppose R is reflexive. Let L be a labelling function, so now $\mathcal{M} = (W, R, L)$ is a model of basic modal logic. We need to show $\mathcal{M} \vDash \Box\phi \rightarrow \phi$. That means we need to show $x \Vdash \Box\phi \rightarrow \phi$ for any $x \in W$, so pick any x. Use the clause for implication in Definition 5.4. Suppose $x \Vdash \Box\phi$; since $R(x, x)$, it immediately follows from the clause for $\Box$ in Definition 5.4 that $x \Vdash p$. Therefore, we have shown $x \Vdash \Box\phi \rightarrow \phi$.
   (b) We just set ϕ to be p.

Table 5.12. Properties of R corresponding to some formulas.

name	formula scheme	property of R
T	$\Box\phi \to \phi$	reflexive
B	$\phi \to \Box\Diamond\phi$	symmetric
D	$\Box\phi \to \Diamond\phi$	serial
4	$\Box\phi \to \Box\Box\phi$	transitive
5	$\Diamond\phi \to \Box\Diamond\phi$	Euclidean
	$\Box\phi \leftrightarrow \Diamond\phi$	functional
	$\Box(\phi \wedge \Box\phi \to \psi) \vee \Box(\psi \wedge \Box\psi \to \phi)$	linear

(c) Suppose the frame satisfies $\Box p \to p$. Take any x; we're going to show $R(x,x)$. Take a labelling function L such that $p \notin L(x)$ and $p \in L(y)$ for all worlds y except x. Proof by contradiction: Assume we don't have $R(x,x)$. Then, $x \Vdash \Box p$, since all the worlds accessible from x satisfy p – this is because all the worlds except x satisfy p; but since $\mathcal{F}$ satisfies $\Box p \to p$, it follows that $x \Vdash \Box p \to p$; therefore, putting $x \Vdash \Box p$ and $x \Vdash \Box p \to p$ together, we get $x \Vdash p$. This is a contradiction to the assumption that we don't have $R(x,x)$, since we said that $p \notin L(x)$. So we must have $R(x,x)$ in our frame!

2. (a) Suppose R is transitive. Let L be a labelling function and $\mathcal{M} = (W, R, L)$. We need to show $M \Vdash \Box\phi \to \Box\Box\phi$. That means we need to show $x \Vdash \Box\phi \to \Box\Box\phi$ for any $x \in W$. Suppose $x \Vdash \Box\phi$; we need to show $x \Vdash \Box\Box\phi$. That is, using the clause for $\Box$ in Definition 5.4, that any y such that $R(x,y)$ satisfies $\Box\phi$; that is, for any y, z with $R(x,y)$ and $R(y,z)$, we have $z \Vdash \phi$.
Well, suppose we did have y and z with $R(x,y)$ and $R(y,z)$. By the fact that R is transitive, we obtain $R(x,z)$. But we're supposing that $x \Vdash \Box\phi$, so from the meaning of $\Box$ we get $z \Vdash \phi$, which is what we needed to prove.
(b) Again, just set ϕ to be p.
(c) Suppose the frame satisfies $\Box p \to \Box\Box p$. Take any x, y and z with $R(x,y)$ and $R(y,z)$; we are going to show $R(x,z)$.
Define a labelling function L such that $p \notin L(z)$ and $p \in L(w)$ for all worlds w except z. Suppose we don't have $R(x,z)$; then $x \Vdash \Box p$, since $w \Vdash p$ for all $w \neq z$. Using the axiom $\Box p \to \Box\Box p$, it follows that $x \Vdash \Box\Box p$. So $y \Vdash \Box p$ holds since $R(x,y)$. The latter and $R(y,z)$ then render $z \Vdash p$, a contradiction. Thus, we must have $R(x,z)$. □

This picture is completed in Table 5.12, which shows, for a collection of formulas, the corresponding property of R. What this table means mathematically is the following:

Theorem 5.14 A frame $\mathcal{F} = (W, R)$ satisfies a formula scheme in Table 5.12 iff R has the corresponding property in that table.

The names of the formulas in the left-hand column are historical, but have stuck and are still used widely in books.

5.3.4 Some modal logics

The logic engineering approach of this section encourages us to design logics by picking and choosing a set $\mathbb{L}$ of formula schemes, according to the application at hand. Some examples of formula schemes that we may wish to consider for a given application are those in Tables 5.7 and 5.12.

Definition 5.15 Let $\mathbb{L}$ be a set of formula schemes of modal logic and $\Gamma \cup \{\psi\}$ a set of formulas of basic modal logic.

1. The set Γ is closed under substitution instances iff whenever $\phi \in \Gamma$, then any substitution instance of ϕ is also in Γ.
2. Let $\mathbb{L}_c$ be the smallest set containing all instances of $\mathbb{L}$.
3. Γ semantically entails ψ in $\mathbb{L}$ iff $\Gamma \cup \mathbb{L}_c$ semantically entails ψ in basic modal logic. In that case, we say that $\Gamma \vDash_{\mathbb{L}} \psi$ holds.

Thus, we have $\Gamma \vDash_{\mathbb{L}} \psi$ if every Kripke model and every world x satisfying $\Gamma \cup \mathbb{L}_c$ therein also satisfies ψ. Note that for $\mathbb{L} = \emptyset$ this definition is consistent with the one of Definition 5.7, since we then have $\Gamma \cup \mathbb{L}_c = \Gamma$. For logic engineering, we require that $\mathbb{L}$ be

- closed under substitution instances; otherwise, we won't be able to characterize $\mathbb{L}_c$ in terms of properties of the accessibility relation; and
- *consistent* in that there is a frame $\mathcal{F}$ such that $\mathcal{F} \vDash \phi$ holds for all $\phi \in \mathbb{L}$; otherwise, $\Gamma \vDash_{\mathbb{L}} \psi$ holds for all Γ and ψ! In most applications of logic engineering, consistency is easy to establish.

We now study a few important modal logics that extend basic modal logic with a consistent set of formula schemes $\mathbb{L}$.

The modal logic K The weakest modal logic doesn't have any chosen formula schemes, like those of Tables 5.7 and 5.12. So $\mathbb{L} = \emptyset$ and this modal logic is called K as it satisfies all instances of the formula scheme K; modal logics with this property are called normal and all modal logics we study in this text are normal.

The modal logic KT45 A well-known modal logic is KT45 – also called S5 in the technical literature – where $\mathbb{L} = \{T, 4, 5\}$ with T, 4 and 5 from Table 5.12. This logic is used to reason about knowledge; $\Box\phi$ means that the agent Q knows ϕ. Table 5.12 tell us, respectively, that

T. Truth: the agent Q knows only true things.

4. Positive introspection: if the agent Q knows something, then she knows that she knows it.
5. Negative introspection: if the agent Q doesn't know something, then she knows that she doesn't know it.

In this application, the formula scheme K means logical omniscience: the agent's knowledge is closed under logical consequence. Note that these properties represent idealisations of knowledge. Human knowledge has none of these properties! Even computer agents may not have them all. There are several attempts in the literature to define logics of knowledge that are more realistic, but we will not consider them here.

The semantics of the logic KT45 must consider only relations R which are: reflexive (T), transitive (4) and Euclidean (5).

Fact 5.16 A relation is reflexive, transitive and Euclidean iff it is reflexive, transitive and symmetric, i.e. if it is an equivalence relation.

KT45 is simpler than K in the sense that it has few essentially different ways of composing modalities.

Theorem 5.17 Any sequence of modal operators and negations in KT45 is equivalent to one of the following: $-$, $\Box$, $\Diamond$, $\neg$, $\neg\Box$ and $\neg\Diamond$, where $-$ indicates the absence of any negation or modality.

The modal logic KT4 The modal logic KT4, that is $\mathbb{L}$ equals $\{T, 4\}$, is also called S4 in the literature. Correspondence theory tells us that its models are precisely the Kripke models $\mathcal{M} = (W, R, L)$, where R is reflexive and transitive. Such structures are often very useful in computer science. For example, if ϕ stands for the type of a piece of code – ϕ could be $\texttt{int} \times \texttt{int} \to \texttt{bool}$, indicating some code which expects a pair of integers as input and outputs a boolean value – then $\Box\phi$ could stand for *residual code* of type ϕ. Thus, in the current world x this code would not have to be executed, but could be saved (= residualised) for execution at a later computation stage. The formula scheme $\Box\phi \to \phi$, the axiom T, then means that code may be executed right away, whereas the formula scheme $\Box\phi \to \Box\Box\phi$, the axiom 4, allows that residual code remain residual, i.e. we can repeatedly postpone its execution in future computation stages. Such type systems have important applications in the specialisation and partial evaluation of code. We refer the interested reader to the bibliographic notes at the end of the chapter.

Theorem 5.18 Any sequence of modal operators and negations in KT4 is equivalent to one of the following: $-$, $\Box$, $\Diamond$, $\Box\Diamond$, $\Diamond\Box$, $\Box\Diamond\Box$, $\Diamond\Box\Diamond$, $\neg$, $\neg\Box$, $\neg\Diamond$, $\neg\Box\Diamond$, $\neg\Diamond\Box$, $\neg\Box\Diamond\Box$ and $\neg\Diamond\Box\Diamond$.

Intuitionistic propositional logic In Chapter 1, we gave a natural deduction system for propositional logic which was sound and complete with

respect to semantic entailment based on truth tables. We also pointed out that the proof rules PBC, LEM and $\neg\neg$e are questionable in certain computational situations. If we disallow their usage in natural deduction proofs, we obtain a logic, called *intuitionistic propositional logic*, together with its own proof theory. So far so good; but it is less clear what sort of semantics one could have for such a logic – again with soundness and completeness in mind. This is where certain models of KT4 will do the job quite nicely. Recall that correspondence theory implies that a model $\mathcal{M} = (W, R, L)$ of KT4 is such that R is reflexive and transitive. The only additional requirement we impose on a model for intuitionistic propositional logic is that its labelling function L be *monotone* in R: $R(x, y)$ implies that $L(x)$ is a subset of $L(y)$. This models that the truth of atomic positive formulas persist throughout the worlds that are reachable from a given world.

Definition 5.19 A model of intuitionistic propositional logic is a model $\mathcal{M} = (W, R, L)$ of KT4 such that $R(x, y)$ always implies $L(x) \subseteq L(y)$. Given a propositional logic formula as in (1.3), we define $x \Vdash \phi$ as in Definition 5.4 exception for the clauses $\rightarrow$ and $\neg$. For $\phi_1 \rightarrow \phi_2$ we define $x \Vdash \phi_1 \rightarrow \phi_2$ iff for all y with $R(x, y)$ we have $y \Vdash \phi_2$ whenever we have $y \Vdash \phi_1$. For $\neg\phi$ we define $x \Vdash \neg\phi$ iff for all y with $R(x, y)$ we have $y \nVdash \phi$.

As an example, consider the model $W = \{x, y\}$ with accessibility relation $R = \{(x, x), (x, y), (y, y)\}$, which is indeed reflexive and transitive. For a labelling function L with $L(x) = \emptyset$ and $L(y) = \{p\}$, we claim that $x \nVdash p \vee \neg p$. (Recall that $p \vee \neg p$ is an instance of LEM which we proved in Chapter 1 with the full natural deduction calculus.) We do not have $x \Vdash p$, for p is not in the set $L(x)$ which is empty. Thus, Definition 5.4 for the case $\vee$ implies that $x \Vdash p \vee \neg p$ can hold only if $x \Vdash \neg p$ holds. But $x \Vdash \neg p$ simply does not hold, since there is a world y with $R(x, y)$ such that $y \Vdash p$ holds, for $p \in L(y)$. The availability of possible worlds in the models of KT4 together with a 'modal interpretation' of $\rightarrow$ and $\neg$ breaks down the validity of the theorem LEM in classical logic.

One can now define semantic entailment in the same manner as for modal logics. Then, one can prove soundness and completeness of the reduced natural deduction system with respect to this semantic entailment, but those proofs are beyond the scope of this book.

5.4 Natural deduction

Verifying semantic entailment $\Gamma \vDash_{\mathbb{L}} \psi$ by appealing to its definition directly would be rather difficult. We would have to consider every Kripke model

that satisfies all formulas of Γ and every world in it. Fortunately, we have a much more usable approach, which is an extension, respectively adaptation, of the systems of natural deduction met in Chapters 1 and 2. Recall that we presented natural deduction proofs as linear representations of proof trees which may involve proof boxes which control the scope of assumptions, or quantifiers. The proof boxes have formulas and/or other boxes inside them. There are rules which dictate how to construct proofs. Boxes open with an *assumption*; when a box is closed – in accordance with a rule – we say that its assumption is *discharged*. Formulas may be repeated and brought into boxes, but may not be brought out of boxes. Every formula must have some justification to its right: a justification can be the name of a rule, or the word 'assumption,' or an instance of the proof rule copy; see e.g. page 13.

Natural deduction works in a very similar way for modal logic. The main difference is that we introduce a new kind of proof box, to be drawn with dashed lines. This is required for the rules for the connective $\Box$. The dashed proof box has a completely different role from the solid one. As we saw in Chapter 1, going into a solid proof box means making an assumption. Going into a dashed box means *reasoning in an arbitrary accessible world.* If at any point in a proof we have $\Box\phi$, we could open a dashed box and put ϕ in it. Then, we could work on this ϕ, to obtain, for example, ψ. Now we could come out of the dashed box and, since we have shown ψ in an arbitrary accessible world, we may deduce $\Box\psi$ in the world outside the dashed box.

Thus, the rules for bringing formulas into dashed boxes and taking formulas out of them are the following:

- Wherever $\Box\phi$ occurs in a proof, ϕ may be put into a subsequent dashed box.
- Wherever ψ occurs at the end of a dashed box, $\Box\psi$ may be put after that dashed box.

We have thus added two rules, $\Box$ introduction and $\Box$ elimination:

$$\frac{\begin{array}{|c|}\hline \vdots \\ \phi \\ \hline\end{array}}{\Box\phi}\ \Box\mathrm{i} \qquad\qquad \frac{\Box\phi}{\begin{array}{|c|}\hline \vdots \\ \phi \\ \vdots \\ \hline\end{array}}\ \Box\mathrm{e}$$

In modal logic, natural deduction proofs contain both solid and dashed boxes, nested in any way. Note that there are no explicit rules for $\Diamond$, which must be written $\neg\Box\neg$ in proofs.

The extra rules for KT45 The rules $\Box$i and $\Box$e are sufficient for capturing semantic entailment of the modal logic K. Stronger modal logics, e.g. KT45, require extra rules if one wants to capture their semantic entailment via proofs. In the case of KT45, this extra strength is expressed by rule schemes for the axioms T, 4 and 5:

$$\frac{\Box\phi}{\phi}\, T \qquad\qquad \frac{\Box\phi}{\Box\Box\phi}\, 4 \qquad\qquad \frac{\neg\Box\phi}{\Box\neg\Box\phi}\, 5$$

An equivalent alternative to the rules 4 and 5 would be to stipulate relaxations of the rules about moving formulas in and out of dashed boxes. Since rule 4 allows us to double-up boxes, we could instead think of it as allowing us to move formulas beginning with $\Box$ into dashed boxes. Similarly, axiom 5 has the effect of allowing us to move formulas beginning with $\neg\Box$ into dashed boxes. Since 5 is a scheme and since ϕ and $\neg\neg\phi$ are equivalent in basic modal logic, we could write $\neg\phi$ instead of ϕ throughout without changing the expressive power and meaning of that axiom.

Definition 5.20 Let $\mathbb{L}$ be a set of formula schemes. We say that $\Gamma \vdash_{\mathbb{L}} \psi$ is valid if ψ has a proof in the natural deduction system for basic modal logic extended with the axioms from $\mathbb{L}$ and premises from Γ.

Examples 5.21 We show that the following sequents are valid:

1. $\vdash_{\mathrm{K}} \Box p \wedge \Box q \rightarrow \Box(p \wedge q)$.

1	$\Box p \wedge \Box q$	assumption
2	$\Box p$	$\wedge\mathrm{e}_1$ 1
3	$\Box q$	$\wedge\mathrm{e}_2$ 1
4	p	$\Box$e 2
5	q	$\Box$e 3
6	$p \wedge q$	$\wedge$i 4, 5
7	$\Box(p \wedge q)$	$\Box$i 4–6
8	$\Box p \wedge \Box q \rightarrow \Box(p \wedge q)$	$\rightarrow$i 1–7

2. $\vdash_{\mathrm{KT45}} p \to \Box\Diamond p$.

1	p	assumption
2	$\Box\neg p$	assumption
3	$\neg p$	T 2
4	$\bot$	$\neg$e 1, 3
5	$\neg\Box\neg p$	$\neg$i 2−4
6	$\Box\neg\Box\neg p$	axiom 5 on line 5
7	$p \to \Box\neg\Box\neg p$	$\to$i 1−6

3. $\vdash_{\mathrm{KT45}} \Box\Diamond\Box p \to \Box p$.

1	$\Box\neg\Box\neg\Box p$	assumption
2	$\neg\Box\neg\Box p$	$\Box$e 1
3	$\neg\Box p$	assumption
4	$\Box\neg\Box p$	axiom 5 on line 3
5	$\bot$	$\neg$e 4, 2
6	$\neg\neg\Box p$	$\neg$i 3−5
7	$\Box p$	$\neg\neg$e 6
8	p	T 7
9	$\Box p$	$\Box$i 2−8
10	$\Box\neg\Box\neg\Box p \to \Box p$	$\to$i 1−9

5.5 Reasoning about knowledge in a multi-agent system

In a *multi-agent system*, different agents have different knowledge of the world. An agent may need to reason about its own knowledge about the world; it may also need to reason about what other agents know about the world. For example, in a bargaining situation, the seller of a car must consider what a buyer knows about the car's value. The buyer must also consider what the seller knows about what the buyer knows about that value and so on.

Reasoning about knowledge refers to the idea that agents in a group take into account not only the facts of the world, but also the knowledge of other agents in the group. Applications of this idea include: games, economics,

cryptography and protocols. It is not very easy for humans to follow the thread of such nested sentences as

Dean doesn't know whether Nixon knows that Dean knows that Nixon knows that McCord burgled O'Brien's office at Watergate.

However, computer agents are better than humans in this respect.

5.5.1 Some examples

We start with some classic examples about reasoning in a multi-agent environment. Then, in the next section, we engineer a modal logic which allows for a formal representation of these examples via sequents and which solves them by proving them in a natural deduction system.

The wise-men puzzle There are three wise men. It's common knowledge – known by everyone and known to be known by everyone, etc. – that there are three red hats and two white hats. The king puts a hat on each of the wise men in such a way that they are not able to see their own hat, and asks each one in turn whether they know the colour of the hat on their head. Suppose the first man says he does not know; then the second says he does not know either.

It follows that the third man must be able to say that he knows the colour of his hat. Why is this? What colour has the third man's hat?

To answer these questions, let us enumerate the seven possibilities which exist: they are

R R R
R R W
R W R
R W W

W R R
W R W
W W R

where, for example, R W W refers to the situation that the first, second and third men have red, white and white hats, respectively. The eighth possibility, W W W, is ruled out as there are only two white hats.

Now let's think of it from the second and third men's point of view. When they hear the first man speak, they can rule out the possibility of the true situation being R W W, because if it were this situation, then the first man, seeing that the others were wearing white hats and knowing that there are only two white hats, would have concluded that his hat must be red. As he said that he did not know, the true situation cannot be R W W. Notice that the second and third men must be intelligent in order to perform

this reasoning; and they must know that the first man is intelligent and truthful as well. In the puzzle, we assume the truthfulness and intelligence and perceptiveness of the men are common knowledge – known by everyone and known to be known by everyone, etc.

When the third man hears the second man speak, he can rule out the possibility of the true situation being W R W, for similar reasons: if it were that, the second man would have said that he knew his hat was red, but he did not say this. Moreover, the third man can also rule out the situation R R W when he hears the second man's answer, for this reason: if the second man had seen that the first was wearing red and the third white, he would have known that it must be R W W or R R W; but he would have known from the first man's answer that it couldn't be R W W, so he would have concluded it was R R W and that he was wearing a red hat; but he did not draw this conclusion, so, reasons the third man, it cannot be R R W.

Having heard the first and second men speak, the third man has eliminated R W W, W R W and R R W; leaving only R R R, R W R, W R R and W W R. In all of these he is wearing a red hat, so he concludes that he must be wearing a red hat.

Notice that the men learn a lot from hearing the other men speak. We emphasise again the importance of the assumption that they tell the truth about their state of knowledge and are perceptive and intelligent enough to come to correct conclusions. Indeed, it is not enough that the three men are truthful, perceptive and intelligent; they must be known to be so by the others and, in later examples, this fact must also be known etc. Therefore, we assume that all this is common knowledge.

The muddy-children puzzle This is one of the many variations on the wise-men puzzle; a difference is that the questions are asked in parallel rather than sequentially. There is a large group of children playing in the garden – their perceptiveness, truthfulness and intelligence being common knowledge, it goes without saying. A certain number of children, say $k \geq 1$, get mud on their foreheads. Each child can see the mud on others, but not on his own forehead. If $k > 1$, then each child can see another with mud on its forehead, so each one knows that at least one in the group is muddy. Consider these two scenarios:

Scenario 1. The father repeatedly asks the question 'Does any of you know whether you have mud on your own forehead?' The first time they all answer 'no;' but, unlike in the wise-men example, they don't learn anything by hearing the others answer 'no,' so they go on answering 'no' to the father's repeated questions.

Scenario 2. The father first announces that at least one of them is muddy – which is something they know already; and then, as before, he repeatedly asks them 'Does any of you know whether you have mud on your own forehead?' The first time they all answer 'no.' Indeed, they go on answering 'no' to the first $k - 1$ repetitions of that same question; but at the kth those with muddy foreheads are able to answer 'yes.'

At first sight, it seems rather puzzling that the two scenarios are different, given that the only difference in the events leading up to them is that in the second one the father announces something that they already know. It would be wrong, however, to conclude that the children learn nothing from this announcement. Although everyone knows the content of the announcement, the father's saying it makes it common knowledge among them, so now they all know that everyone else knows it, etc. This is the crucial difference between the two scenarios.

To understand scenario 2, consider a few cases of k.

$k = 1$, i.e. just one child has mud. That child is immediately able to answer 'yes,' since she has heard the father and doesn't see any other child with mud.

$k = 2$, say only the children Ramon and Candy have mud. Everyone answers 'no' the first time. Now Ramon thinks: since Candy answered 'no' the first time, she must see someone with mud. Well, the only person I can see with mud is Candy, so if she can see someone else it must be me. So Ramon answers 'yes' the second time. Candy reasons symmetrically about Ramon and also answers 'yes' the second time round.

$k = 3$, say only the children Alice, Bob, and Charlie have mud. Everyone answers 'no' the first two times. But now Alice thinks: if it was just Bob and Charlie with mud, they would have answered 'yes' the second time; making the argument for $k = 2$ above. So there must be a third person with mud; since I can see only Bob and Charlie having mud, the third person must be me. So Alice answers 'yes' the third time. For symmetrical reasons, so do Bob and Charlie.

And similarly for other cases of k.

To see that it was not common knowledge before the father's announcement that one of the children was muddy, consider again $k = 2$, with Ramon and Candy. Of course, Ramon and Candy both know someone is muddy – they see each other; but, for example, Ramon doesn't know that Candy knows that someone is dirty. For all Ramon knows, Candy might be the only dirty one and therefore not be able to see a dirty child.

5.5.2 The modal logic KT45^n

We now generalise the modal logic KT45 given in Section 5.3.4. Instead of having just one $\Box$, it will have many, one for each agent i from a fixed set $\mathcal{A} = \{1, 2, \ldots, n\}$ of agents. We write those modal connectives as K_i (for each agent $i \in \mathcal{A}$); the K is to emphasise the application to *knowledge*. We assume a collection $p, q, r, \ldots$ of atomic formulas. The formula $K_i\, p$ means that agent i knows p; so, for example, $K_1\, p \wedge K_1 \neg K_2 K_1\, p$ means that agent 1 knows p, but knows that agent 2 doesn't know he knows it.

We also have the modal connectives E_G, where G is any subset of $\mathcal{A}$. The formula $E_G\, p$ means everyone in the group G knows p. If $G = \{1, 2, 3, \ldots, n\}$, then $E_G\, p$ is equivalent to $K_1\, p \wedge K_2\, p \wedge \cdots \wedge K_n\, p$. We assume similar binding priorities to those put forward on page 307.

Convention 5.22 The binding priorities of KT45^n are the ones of basic modal logic, if we think of each modality K_i, E_G and C_G as 'being' $\Box$.

One might think that ϕ could not be more widely known than everyone knowing it, but this is not the case. It could be, for example, that everyone knows ϕ, but they might not know that they all know it. If ϕ is supposed to be a secret, it might be that you and your friend both know it, but your friend does not know that you know it and you don't know that your friend knows it. Thus, $E_G E_G\, \phi$ is a state of knowledge even greater than $E_G\, \phi$ and $E_G E_G E_G\, \phi$ is greater still. We say that ϕ is *common knowledge among* G, written $C_G\, \phi$, if everyone knows ϕ and everyone knows that everyone knows it; and everyone knows that; and knows *that* etc. So we may think of $C_G\, \phi$ as an infinite conjunction

$$E_G\, \phi \wedge E_G E_G\, \phi \wedge E_G E_G E_G\, \phi \wedge \ldots .$$

However, since our logics only have finite conjunctions, we cannot reduce C_G to something which is already in the logic. We have to express the *infinite* aspect of C_G via its semantics and retain it as an additional modal connective. Finally, $D_G\, \phi$ means the knowledge of ϕ is distributed among the group G; although no-one in G may know it, they would be able to work it out if they put their heads together and combined the information distributed among them.

Definition 5.23 A formula ϕ in the multi-modal logic of KT45^n is defined by the following grammar:

$$\begin{aligned}\phi ::= {} & \bot \mid \top \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\phi \leftrightarrow \phi) \mid \\ & (K_i\, \phi) \mid (E_G\, \phi) \mid (C_G\, \phi) \mid (D_G\, \phi)\end{aligned}$$

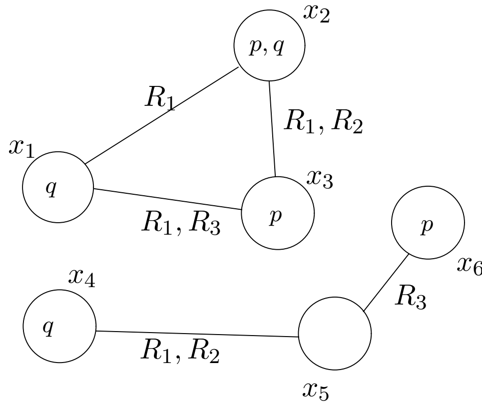

Figure 5.13. A KT45^n model for $n = 3$.

where p is any atomic formula, $i \in \mathcal{A}$ and $G \subseteq \mathcal{A}$. We simply write E, C and D without subscripts if we refer to $E_{\mathcal{A}}$, $C_{\mathcal{A}}$ and $D_{\mathcal{A}}$.

Compare this definition with Definition 5.1. Instead of $\Box$, we have several modalities K_i and we also have E_G, C_G and D_G for each $G \subseteq \mathcal{A}$. Actually, all of these connectives will shortly be seen to be 'box-like' rather than 'diamond-like', in the sense that they distribute over $\wedge$ rather than over $\vee$ – compare this to the discussion of equivalences on page 308. The 'diamond-like' correspondents of these connectives are not explicitly in the language, but may of course be obtained using negations, i.e. $\neg K_i \neg$, $\neg C_G \neg$ etc.

Definition 5.24 A model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ of the multi-modal logic KT45^n with the set $\mathcal{A}$ of n agents is specified by three things:

1. a set W of possible worlds;
2. for each $i \in \mathcal{A}$, an equivalence relation R_i on W ($R_i \subseteq W \times W$), called the accessibility relations; and
3. a labelling function $L : W \to \mathcal{P}(\texttt{Atoms})$.

Compare this with Definition 5.3. The difference is that, instead of just one accessibility relation, we now have a family, one for each agent in $\mathcal{A}$; and we assume the accessibility relations are equivalence relations.

We exploit these properties of R_i in the graphical illustrations of Kripke models for KT45^n. For example, a model of KT45^3 with set of worlds $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ is shown in Figure 5.13. The links between the worlds have to be labelled with the name of the accessibility relation, since we have several relations. For example, x_1 and x_2 are related by R_1, whereas x_4 and

x_5 are related both by R_1 and by R_2. We simplify by no longer requiring arrows on the links. This is because we know that the relations are symmetric, so the links are bi-directional. Moreover, the relations are also reflexive, so there should be loops like the one on x_4 in Figure 5.11 in all the worlds and for all of the relations. We can simply omit these from the diagram, since we don't need to distinguish between worlds which are self-related and those which are not.

Definition 5.25 Take a model $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$ of KT45^n and a world $x \in W$. We define when ϕ is true in x via a satisfaction relation $x \Vdash \phi$ by induction on ϕ:

$$
\begin{array}{rl}
x \Vdash p & \text{iff } p \in L(x) \\
x \Vdash \neg\phi & \text{iff } x \nVdash \phi \\
x \Vdash \phi \wedge \psi & \text{iff } x \Vdash \phi \text{ and } x \Vdash \psi \\
x \Vdash \phi \vee \psi & \text{iff } x \Vdash \phi \text{ or } x \Vdash \psi \\
x \Vdash \phi \to \psi & \text{iff } x \Vdash \psi \text{ whenever we have } x \Vdash \phi \\
x \Vdash K_i\,\psi & \text{iff, for each } y \in W,\ R_i(x, y) \text{ implies } y \Vdash \psi \\
x \Vdash E_G\,\psi & \text{iff, for each } i \in G,\ x \Vdash K_i\,\psi \\
x \Vdash C_G\,\psi & \text{iff, for each } k \geq 1, \text{ we have } x \Vdash E_G^k\psi, \\
 & \quad \text{where } E_G^k \text{ means } E_G E_G \dots E_G - k \text{ times} \\
x \Vdash D_G\,\psi & \text{iff, for each } y \in W, \text{ we have } y \Vdash \psi, \\
 & \quad \text{whenever } R_i(x, y) \text{ for all } i \in G.
\end{array}
$$

Again, we write $\mathcal{M}, x \Vdash \phi$ if we want to emphasise the model $\mathcal{M}$.

Compare this with Definition 5.4. The cases for the boolean connectives are the same as for basic modal logic. Each K_i behaves like a $\Box$, but refers to its own accessibility relation R_i. As already stated, there are no equivalents of $\Diamond$, but we can recover them as $\neg K_i \neg$. The connective E_G is defined in terms of the K_i and C_G is defined in terms of E_G.

Many of the results we had for basic modal logic with a single accessibility relation also hold in this more general setting of several accessibility relations. Summarising,

- a *frame* $\mathcal{F}$ for KT45^n $(W, (R_i)_{i \in \mathcal{A}})$ for the modal logic KT45^n is a set W of worlds and, for each $i \in \mathcal{A}$, an equivalence relation R_i on W.
- a frame $\mathcal{F} = (W, (R_i)_{i \in \mathcal{A}})$ for KT45^n is said to satisfy ϕ if, for each labelling function $L \colon W \to \mathcal{P}(\texttt{Atoms})$ and each $w \in W$, we have $\mathcal{M}, w \Vdash \phi$ holds, where $\mathcal{M} = (W, (R_i)_{i \in \mathcal{A}}, L)$. In that case, we say that $\mathcal{F} \vDash \phi$ holds.

The following theorem is useful for answering questions about formulas involving E and C. Let $\mathcal{M} = (W,\ (R_i)_{i \in \mathcal{A}}, L)$ be a model for KT45^n

and $x, y \in W$. We say that y is *G-reachable in k steps* from x if there are $w_1, w_2, \ldots, w_{k-1} \in W$ and $i_1, i_2, \ldots, i_k$ in G such that

$$x\, R_{i_1}\, w_1\, R_{i_2}\, w_2\, \ldots\, R_{i_{k-1}}\, w_{k-1}\, R_{i_k}\, y$$

meaning $R_{i_1}(x, w_1)$, $R_{i_2}(w_1, w_2)$, $\ldots$, $R_{i_k}(w_k, y)$. We also say that y is *G-reachable* from x if there is some k such that it is G-reachable in k steps.

Theorem 5.26

1. $x \Vdash E_G^k \phi$ iff, for all y that are G-reachable from x in k steps, we have $y \Vdash \phi$.
2. $x \Vdash C_G\, \phi$ iff, for all y that are G-reachable from x, we have $y \Vdash \phi$.

PROOF:

1. First, suppose $y \Vdash \phi$ for all y G-reachable from x in k steps. We will prove that $x \Vdash E_G^k \phi$ holds. It is sufficient to show that $x \Vdash K_{i_1} K_{i_2} \ldots K_{i_k}\, \phi$ for any $i_1, i_2, \ldots, i_k \in G$. Take any $i_1, i_2, \ldots, i_k \in G$ and any w_1, $w_2, \ldots$, w_{k-1} and y such that there is a path of the form $x\, R_{i_1}\, w_1\, R_{i_2}\, w_2\, \ldots\, R_{i_{k-1}}\, w_{k-1}\, R_{i_k}\, y$. Since y is G-reachable from x in k steps, we have $y \Vdash \phi$ by our assumption, so $x \Vdash K_{i_1} K_{i_2} \ldots K_{i_k}\, \phi$ as required.
   Conversely, suppose $x \Vdash E_G^k \phi$ holds and y is G-reachable from x in k steps. We must show that $y \Vdash \phi$ holds. Take $i_1, i_2, \ldots, i_k$ by G-reachability; since $x \Vdash E_G^k \phi$ implies $x \Vdash K_{i_1} K_{i_2} \ldots K_{i_k}\, \phi$, we have $y \Vdash \phi$.
2. This argument is similar.

Some valid formulas in KT45^n The formula K holds for the connectives K_i, E_G, C_G and D_G, i.e. we have the corresponding formula schemes

$$\begin{aligned} &K_i\, \phi \land K_i\, (\phi \to \psi) \to K_i\, \psi \\ &E_G\, \phi \land E_G\, (\phi \to \psi) \to E_G\, \psi \\ &C_G\, \phi \land C_G\, (\phi \to \psi) \to C_G\, \psi \\ &D_G\, \phi \land D_G\, (\phi \to \psi) \to D_G\, \psi. \end{aligned}$$

This means that these different 'levels' of knowledge are closed under logical consequence. For example, if certain facts are common knowledge and some other fact follows logically from them, then that fact is also common knowledge.

Observe that E, C and D are 'box-like' connectives, in the sense that they quantify universally over certain accessibility relations. That is to say, we may define the relations R_{E_G}, R_{D_G} and R_{C_G} in terms of the relations R_i, as follows:

$$\begin{array}{llll} R_{E_G}(x, y) & \text{iff} & R_i(x, y) & \text{for some } i \in G \\ R_{D_G}(x, y) & \text{iff} & R_i(x, y) & \text{for all } i \in G \\ R_{C_G}(x, y) & \text{iff} & R_{E_G}^k(x, y) & \text{for each } k \geq 1. \end{array}$$

It follows from this that E_G, D_G and C_G satisfy the K formula with respect to the accessibility relations R_{E_G}, R_{D_G} and R_{C_G}, respectively.

What about other valid formulas? Since we have stipulated that the relations R_i are equivalence relations, it follows from the multi-modal analogues of Theorem 5.13 and Table 5.12 that the following formulas are valid in $\mathrm{KT45}^n$ for each agent i:

$K_i\,\phi \to K_i K_i\,\phi$	positive introspection
$\neg K_i\,\phi \to K_i \neg K_i\,\phi$	negative introspection
$K_i\,\phi \to \phi$	truth.

These formulas also hold for D_G, since R_{D_G} is also an equivalence relation, but these don't automatically generalise for E_G and C_G. For example, $E_G\,\phi \to E_G E_G\,\phi$ is not valid; if it were valid, it would imply that common knowledge was nothing more than knowledge by everybody. The scheme $\neg E_G\,\phi \to E_G \neg E_G\,\phi$ is also not valid. The failure of these formulas to be valid can be traced to the fact that R_{E_G} is not necessarily an equivalence relation, even though each R_i is an equivalence relation. However, R_{E_G} is reflexive, so $E_G\,\phi \to \phi$ is valid, provided that $G \neq \emptyset$. If $G = \emptyset$, then $E_G\,\phi$ holds vacuously, even if ϕ is false.

Since R_{C_G} is an equivalence relation, the formulas T, 4 and 5 above do hold for C_G, although the third one still requires the condition that $G \neq \emptyset$.

5.5.3 Natural deduction for $\mathrm{KT45}^n$

The proof system for KT45 is easily extended to $\mathrm{KT45}^n$; but for simplicity, we omit reference to the connective D.

1. The dashed boxes now come in different 'flavours' for different modal connectives; we'll indicate the modality in the top left corner of the dashed box.
2. The axioms T, 4 and 5 can be used for any K_i, whereas axioms 4 and 5 can be used for C_G, but not for E_G – recall the discussion in Section 5.5.2.
3. In the rule CE, we may deduce $E_G^k\phi$ from $C_G\,\phi$ for any k; or we could go directly to $K_{i_1} \dots K_{i_k}\,\phi$ for any agents $i_1, \dots, i_{k \in G}$ by using the rule CK. Strictly speaking, these rules are a whole set of such rules, one for each choice of k and $i_1, \dots, i_k$, but we refer to all of them as CE and CK respectively.
4. Applying rule EK_i, we may deduce $K_i\,\phi$ from $E_G\,\phi$ for any $i \in G$. From $\bigwedge_{i \in G} K_i\,\phi$ we may deduce $E_G\,\phi$ by virtue of rule KE. Note that the proof rule EK_i is like a generalised and-elimination rule, whereas KE behaves like an and-introduction rule.

The proof rules for $\mathrm{KT45}^n$ are summarised in Figure 5.14. As before, we can think of the rules $K4$ and $K5$ and $C4$ and $C5$ as relaxations of the

$$\frac{\boxed{K_i \quad \vdots \quad \phi}}{K_i\phi}\ K_i\text{i} \qquad \frac{\boxed{E_G \quad \vdots \quad \phi}}{E_G\phi}\ E_G\text{i} \qquad \frac{\boxed{C_G \quad \vdots \quad \phi}}{C_G\phi}\ C_G\text{i}$$

$$\frac{K_i\phi}{\boxed{K_i \quad \vdots \quad \phi \quad \vdots}}\ K_i\text{e} \qquad \frac{E_G\phi}{\boxed{E_G \quad \vdots \quad \phi \quad \vdots}}\ E_G\text{e} \qquad \frac{C_G\phi}{\boxed{C_G \quad \vdots \quad \phi \quad \vdots}}\ C_G\text{e}$$

$$\frac{K_i\,\phi \text{ for each } i \in G}{E_G\,\phi}\ KE \qquad \frac{E_G\,\phi \quad i \in G}{K_i\,\phi}\ EK_i \qquad \frac{C_G\,\phi}{E_G \dots E_G\,\phi}\ CE$$

$$\frac{C_G\,\phi \quad i_j \in G}{K_{i_1} \dots K_{i_k}\,\phi}\ CK \qquad \frac{C_G\,\phi}{C_G C_G\,\phi}\ C4 \qquad \frac{\neg C_G\,\phi}{C_G \neg C_G\,\phi}\ C5$$

$$\frac{K_i\,\phi}{\phi}\ KT \qquad \frac{K_i\,\phi}{K_i K_i\,\phi}\ K4 \qquad \frac{\neg K_i\,\phi}{K_i \neg K_i\,\phi}\ K5$$

Figure 5.14. Natural deduction rules for $\mathrm{KT45}^n$.

rules about moving formulas in and out of dashed proof boxes. Since rule $K4$ allows us to double-up K_i, we could instead think of it as allowing us to move formulas beginning with K_i into K_i-dashed boxes. Similarly, rule $C5$ has the effect of allowing us to move formulas beginning with $\neg C_G$ into C_G-dashed boxes.

An intuitive way of thinking about the dashed boxes is that formulas in them are known to the agent in question. When you open a K_i-dashed box, you are considering what agent i knows. It's quite intuitive that an ordinary formula ϕ cannot be brought into such a dashed box, because the mere truth of ϕ does not mean that agent i knows it. In particular, you can't use the rule $\neg$i if one of the premises of the rule is outside the dashed box you're working in.

1	$C(p \vee q)$	premise
2	$K_1(K_2\, p \vee K_2\, \neg p)$	premise
3	$K_1 \neg K_2\, q$	premise
4	$K_1 K_2\, (p \vee q)$	CK 1
5	K_1: $K_2\, (p \vee q)$	K_1e 4
6	$K_2\, p \vee K_2\, \neg p$	K_1e 2
7	$\neg K_2\, q$	K_1e 3
8	$K_2\, p$ assumption | $K_2\, \neg p$	assumption
9	p axiom T 8 | K_2: $\neg p$	K_2e 8
10	$p \vee q$	K_2e 5
11	q	prop 9, 10
12	$K_2\, q$	K_2i 9–11
13	$\bot$	$\neg$e 12, 7
14	p	$\bot$e 13
15	p	$\vee$e 6, 8–14, 8–14
16	$K_1\, p$	K_1i 5–15

Figure 5.15. A proof of $C(p \vee q)$, $K_1(K_2 p \vee K_2\, \neg p)$, $K_1 \neg K_2\, q \vdash K_1\, p$.

Observe the power of $C\,\phi$ in the premises: we can bring ϕ into *any* dashed box by the application of the rules CK and K_ie, no matter how deeply nested boxes are. The rule $E^k\,\phi$, on the other hand, ensures that ϕ can be brought into any dashed box with nesting $\leq k$. Compare this with Theorem 5.26.

Example 5.27 We show that the sequent[1] $C(p \vee q)$, $K_1(K_2 p \vee K_2\, \neg p)$, $K_1 \neg K_2\, q \vdash K_1\, p$ is valid in the modal logic $\mathrm{KT45}^n$. That means: if it is common knowledge that $p \vee q$; and agent 1 knows that agent 2 knows whether p is the case and also knows that agent 2 doesn't know that q is true; then agent 1 knows that p is true. See Figure 5.15 for a proof. In line 12, we derived q from $\neg p$ and $p \vee q$. Rather than show the full derivation in propositional logic, which is not the focus here, we summarise by writing 'prop' as the justification for an inference in propositional logic.

[1] In this section we simply write $\vdash$ for $\vdash_{\mathrm{KT45}^n}$, unless indicated otherwise.

5.5.4 Formalising the examples

Now that we have set up the modal logic KT45^n, we can turn our attention to the question of how to represent the wise-men and muddy-children puzzles in this logic. Unfortunately, in spite of its sophistication, our logic is too simple to capture all the nuances of those examples. Although it has connectives for representing different items of knowledge held by different agents, it does not have any temporal aspect, so it cannot directly capture the way in which the agents' knowledge changes as time proceeds. We will overcome this limitation by considering several 'snapshots' during which time is fixed.

The wise-men puzzle Recall that there are three wise men; and it's common knowledge that there are three red hats and two white hats. The king puts a hat on each of the wise men and asks them sequentially whether they know the colour of the hat on their head – they are unable to see their own hat. We suppose the first man says he does not know; then the second says he does not know. We want to prove that, whatever the distribution of hats, the third man now knows his hat is red.

Let p_i mean that man i has a red hat; so $\neg p_i$ means that man i has a white hat. Let Γ be the set of formulas

$$\begin{aligned}
\{&C(p_1 \lor p_2 \lor p_3),\\
&C(p_1 \to K_2\, p_1), C(\neg p_1 \to K_2\, \neg p_1),\\
&C(p_1 \to K_3\, p_1), C(\neg p_1 \to K_3\, \neg p_1),\\
&C(p_2 \to K_1\, p_2), C(\neg p_2 \to K_1\, \neg p_2),\\
&C(p_2 \to K_3\, p_2), C(\neg p_2 \to K_3\, \neg p_2),\\
&C(p_3 \to K_1\, p_3), C(\neg p_3 \to K_1\, \neg p_3),\\
&C(p_3 \to K_2\, p_3), C(\neg p_3 \to K_2\, \neg p_3)\}.
\end{aligned}$$

This corresponds to the initial set-up: it is common knowledge that one of the hats must be red and that each man can see the colour of the other men's hats.

The announcement that the first man doesn't know the colour of his hat amounts to the formula

$$C(\neg K_1\, p_1 \land \neg K_1\, \neg p_1)$$

and similarly for the second man.

A naive attempt at formalising the wise-men problem might go something like this: we simply prove

$$\Gamma,\ C(\neg K_1\, p_1 \land \neg K_1\, \neg p_1),\ C(\neg K_2\, p_2 \land \neg K_2\, \neg p_2) \ \vdash\ K_3\, p_3$$

i.e. if Γ is true and the announcements are made, then the third man knows his hat is red. However, this fails to capture the fact that time passes between the announcements. The fact that $C\neg K_1\,p_1$ is true after the first announcement does not mean it is true after some subsequent announcement. For example, if someone announces p_1, then Cp_1 becomes true.

The reason that this formalisation is incorrect is that, although knowledge accrues with time, *lack* of knowledge does not accrue with time. If I know ϕ, then (assuming that ϕ doesn't change) I will know it at the next time-point; but if I do *not* know ϕ, it may be that I *do* know it at the next time point, since I may acquire more knowledge.

To formalise the wise-men problem correctly, we need to break it into two entailments, one corresponding to each announcement. When the first man announces he does not know the colour of his hat, a certain *positive* formula ϕ becomes common knowledge. Our informal reasoning explained that all men could then rule out the state RWW which, given $p_1 \vee p_2 \vee p_3$, led them to the common knowledge of $p_2 \vee p_3$. Thus, ϕ is just $p_2 \vee p_3$ and we need to prove the entailment

Entailment 1. $\Gamma,\ C(\neg K_1\,p_1 \wedge \neg K_1\,\neg p_1) \vdash C(p_2 \vee p_3)$.

A proof of this sequent can be found in Figure 5.16.

Since $p_2 \vee p_3$ is a positive formula, it persists with time and can be used in conjunction with the second announcement to prove the desired conclusion:

Entailment 2. $\Gamma,\ C(p_2 \vee p_3),\ C(\neg K_2\,p_2, \wedge \neg K_2\,\neg p_2)\ \vdash\ K_3\,p_3$.

This method requires some careful thought: given an announcement of negative information such as a man declaring that he does not know what the colour of his hat is, we need to work out what positive-knowledge formula can be derived from this and such new knowledge has to be sufficient to make even more progress towards solving the puzzle in the next round.

Routine proof segments like those in lines 11–16 of Figure 5.16 may be abbreviated into one step as long as all participating proof rules are recorded. The resulting shorter representation can be seen in Figure 5.17.

In Figure 5.16, notice that the premises in lines 2 and 5 are not used. The premises in lines 2 and 3 stand for any such formula for a given value of i and j, provided $i \neq j$; this explains the inference made in line 8. In Figure 5.18, again notice that the premises in lines 1 and 5 are not used. Observe also that axiom T in conjunction with CK allows us to infer ϕ from any $C\phi$, although we had to split this up into two separate steps in lines 16 and 17. Practical implementations would probably allow for hybrid rules which condense such reasoning into one step.

1	$C(p_1 \vee p_2 \vee p_3)$	premise
2	$C(p_i \to K_j\, p_i)$	premise, $(i \neq j)$
3	$C(\neg p_i \to K_j\, \neg p_i)$	premise, $(i \neq j)$
4	$C\neg K_1\, p_1$	premise
5	$C\neg K_1\, \neg p_1$	premise
6	C	
7	$\neg p_2 \wedge \neg p_3$	assumption
8	$\neg p_2 \to K_1\, \neg p_2$	Ce 3 $(i, j) = (2, 1)$
9	$\neg p_3 \to K_1\, \neg p_3$	Ce 3 $(i, j) = (3, 1)$
10	$K_1\, \neg p_2 \wedge K_1\, \neg p_3$	prop $7, 8, 9$
11	$K_1\, \neg p_2$	$\wedge e_1$ 10
12	$K_1\, \neg p_3$	$\wedge e_2$ 10
13	K_1	
14	$\neg p_2$	K_1e 11
15	$\neg p_3$	K_1e 12
16	$\neg p_2 \wedge \neg p_3$	$\wedge$i $14, 15$
17	$p_1 \vee p_2 \vee p_3$	Ce 1
18	p_1	prop $16, 17$
19	$K_1\, p_1$	K_1i 13−18
20	$\neg K_1\, p_1$	Ce 4
21	$\bot$	$\neg$e $19, 20$
22	$\neg(\neg p_2 \wedge \neg p_3)$	$\neg$i 7−21
23	$p_2 \vee p_3$	prop 22
24	$C(p_2 \vee p_3)$	Ci 6−23

Figure 5.16. Proof of the sequent 'Entailment 1' for the wise-men puzzle.

The muddy-children puzzle Suppose there are n children. Let p_i mean that the ith child has mud on its forehead. We consider Scenario 2, in which the father announces that one of the children is muddy. Similarly to the case for the wise men, it is common knowledge that each child can see the other children, so it knows whether the others have mud, or not. Thus, for example,

1	$C(p_1 \vee p_2 \vee p_3)$	premise
2	$C(p_i \to K_j\, p_i)$	premise, $(i \neq j)$
3	$C(\neg p_i \to K_j\, \neg p_i)$	premise, $(i \neq j)$
4	$C\neg K_1\, p_1$	premise
5	$C\neg K_1\, \neg p_1$	premise
6	C	
7	$\neg p_2 \wedge \neg p_3$	assumption
8	$\neg p_2 \to K_1\, \neg p_2$	Ce 3 $(i,j) = (2,1)$
9	$\neg p_3 \to K_1\, \neg p_3$	Ce 3 $(i,j) = (3,1)$
10	$K_1\, \neg p_2 \wedge K_1\, \neg p_3$	prop $7, 8, 9$
11	K_1	
12	$\neg p_2 \wedge \neg p_3$	$\wedge$e$_1$, K_1e, $\wedge$i
13	$p_1 \vee p_2 \vee p_3$	Ce 1
14	p_1	prop 12, 13
15	$K_1\, p_1$	K_1i 11−14
16	$\neg K_1\, p_1$	Ce 4
17	$\bot$	$\neg$e $15, 16$
18	$\neg(\neg p_2 \wedge \neg p_3)$	$\neg$i 7−17
19	$p_2 \vee p_3$	prop 18
20	$C(p_2 \vee p_3)$	Ci 6−19

Figure 5.17. A more compact representation of the proof in Figure 5.16.

we have that $C(p_1 \to K_2\, p_1)$, which says that it is common knowledge that, if child 1 is muddy, then child 2 knows this and also $C(\neg p_1 \to K_2\, \neg p_1)$. Let Γ be the collection of formulas:

$$C(p_1 \vee p_2 \vee \cdots \vee p_n)$$

$$\bigwedge_{i \neq j} C(p_i \to K_j\, p_i)$$

$$\bigwedge_{i \neq j} C(\neg p_i \to K_j\, \neg p_i).$$

1	$C(p_1 \vee p_2 \vee p_3)$	premise
2	$C(p_i \to K_j\, p_i)$	premise, $(i \neq j)$
3	$C(\neg p_i \to K_j\, \neg p_i)$	premise, $(i \neq j)$
4	$C\neg K_2\, p_2$	premise
5	$C\neg K_2\, \neg p_2$	premise
6	$C(p_2 \vee p_3)$	premise
7	K_3	
8	$\neg p_3$	assumption
9	$\neg p_3 \to K_2\, \neg p_3$	CK 3 $(i,j) = (3,2)$
10	$K_2\, \neg p_3$	$\to$e 9, 8
11	K_2	
12	$\neg p_3$	K_2e 10
13	$p_2 \vee p_3$	Ce 6
14	p_2	prop 12, 13
15	$K_2\, p_2$	K_2i 11–14
16	$K_i\, \neg K_2\, p_2$	CK 4, for each i
17	$\neg K_2\, p_2$	KT 16
18	$\bot$	$\neg$e 15, 17
19	p_3	PBC 8–18
20	$K_3\, p_3$	K_3i 7–19

Figure 5.18. Proof of the sequent 'Entailment 2' for the wise-men puzzle.

Note that $\bigwedge_{i \neq j} \psi_{(i,j)}$ is a shorthand for the finite conjunction of all formulas $\psi_{(i,j)}$, where i is different from j. Let G be any set of children. We will require formulas of the form

$$\alpha_G \stackrel{\text{def}}{=} \bigwedge_{i \in G} p_i \wedge \bigwedge_{i \notin G} \neg p_i.$$

The formula α_G states that it is precisely the children in G that have muddy foreheads.

1	$\neg p_1 \wedge \neg p_2 \wedge \cdots \wedge p_i \wedge \cdots \wedge \neg p_n$	$\alpha_{\{i\}}$
2	$C(p_1 \vee \cdots \vee p_n)$	in Γ
3	$\neg p_j$	$\wedge$e 1, for each $j \neq i$
4	$\neg p_j \to K_i \neg p_j$	in Γ, for each $j \neq i$
5	$K_i \neg p_j$	$\to$e 4, 3, for each $j \neq i$
6	$K_i (p_1 \vee \cdots \vee p_n)$	CK 2
7	K_i	
8	$p_1 \vee \cdots \vee p_n$	K_i e 6
9	$\neg p_j$	K_i e 5, for each $j \neq i$
10	p_i	prop 9, 8
11	$K_i\, p_i$	K_i i

Figure 5.19. Proof of the sequent 'Entailment 1' for the muddy-children puzzle.

Suppose now that $k = 1$, i.e. that one child has mud on its forehead. We would like to show that that child knows that it is the one. We prove the following entailment.

Entailment 1. $\Gamma, \alpha_{\{i\}} \vdash K_i\, p_i$.

This says that, if the actual situation is one in which only one child called i has mud, then that child will know it. Our proof follows exactly the same lines as the intuition: i sees that no other children have mud, but knows that at least one has mud, so knows it must be itself who has a muddy forehead. The proof is given in Figure 5.19.

Note that the comment 'for each $j \neq i$' means that we supply this argument for any such j. Thus, we can form the conjunction of all these inferences which we left implicit in the inference on line 10.

What if there is more than one child with mud? In this case, the children all announce in the first parallel round that they do not know whether they are muddy or not, corresponding to the formula

$$A \stackrel{\text{def}}{=} C(\neg K_1\, p_1 \wedge \neg K_1 \neg p_1) \wedge \cdots \wedge C(\neg K_n\, p_n \wedge \neg K_n \neg p_n).$$

We saw in the wise-men example that it is dangerous to put the announcement A alongside the premises Γ, because the truth of A, which has negative claims about the children's knowledge, cannot be guaranteed to persist with

time. So we seek some positive formula which represents what the children learn upon hearing the announcement. As in the wise-men example, this formula is implicit in the informal reasoning about the muddy children given in Section 5.5.1: if it is common knowledge that there are at least k muddy children, then, after an announcement of the form A, it will be common knowledge that there are at least $k+1$ muddy children.

Therefore, after the first announcement A, the set of premises is

$$\Gamma, \bigwedge_{1 \le i \le n} C\neg\alpha_{\{i\}}.$$

This is Γ together with the common knowledge that the set of muddy children is not a singleton set.

After the second announcement A, the set of premises becomes

$$\Gamma, \bigwedge_{1 \le i \le n} C\neg\alpha_{\{i\}}, \bigwedge_{i \ne j} C\neg\alpha_{\{i,j\}}$$

which we may write as

$$\Gamma, \bigwedge_{|G| \le 2} C\neg\alpha_G.$$

Please try carefully to understand the notation:

α_G	the set of muddy children is precisely the set G
$\neg\alpha_G$	the set of muddy children is some other set than G
$\bigwedge_{\|G\| \le k} \neg\alpha_G$	the set of muddy children is of size greater than k.

The entailment corresponding to the second round is:

$$\Gamma,\ C(\bigwedge_{|G| \le 2} \neg\alpha_G),\ \alpha_H \vdash \bigwedge_{i \in H} K_i\, p_i, \qquad \text{where } |H| = 3\,.$$

The entailment corresponding to the kth round is:

Entailment 2. $\Gamma,\ C(\bigwedge_{|G| \le k} \neg\alpha_G),\ \alpha_H \vdash \bigwedge_{i \in H} K_i\, p_i,$ where $|H| = k+1$. Please try carefully to understand what this sequent is saying. 'If all the things in Γ are true and if it is common knowledge that the set of muddy children is not of size less than or equal to k and if actually it is of size $k+1$, then each of those $k+1$ children can deduce that they are muddy.' Notice how this fits with our intuitive account given earlier in this text.

1	α_H	premise
2	$C\neg\alpha_G$	premise as $\lvert G\rvert \leq k$
3	p_j	$\wedge$e 1, for each $j \in G$
4	$\neg p_k$	$\wedge$e 1, for each $k \notin H$
5	$p_j \to K_i\, p_j$	in Γ for each $j \in G$
6	$K_i\, p_j$	$\to$e 5, 4, for each $j \in G$
7	$\neg p_k \to K_i\, \neg p_k$	in Γ for each $k \notin H$
8	$K_i\, \neg p_k$	$\to$e 7, 4, for each $k \notin H$
9	$K_i\, \neg\alpha_G$	CK 2
10	K_i	
11	p_j	K_i e 6 ($j \in G$)
12	$\neg p_k$	K_i e 8 ($k \notin H$)
13	$\neg p_i$	assumption
14	α_G	$\wedge$i 11, 12, 13
15	$\neg\alpha_G$	K_i e 9
16	$\bot$	$\neg$e 14, 15
17	$\neg\neg p_i$	$\neg$i 13−16
18	p_i	$\neg\neg$e 17
19	$K_i\, p_i$	K_i i 10−18

Figure 5.20. The proof of $\Gamma,\ C(\neg\alpha_G),\ \alpha_H \vdash K_i\, p_i$, used to prove 'Entailment 2' for the muddy-children puzzle.

To prove Entailment 2, take any $i \in H$. It is sufficient to prove that

$$\Gamma,\ C(\bigwedge_{|G|\leq k} \neg\alpha_G),\ \alpha_H \vdash K_i\, p_i$$

is valid, as the repeated use of $\wedge$i over all values of i gives us a proof of Entailment 2. Let G be $H - \{i\}$; the proof that $\Gamma,\ C(\neg\alpha_G),\ \alpha_H \vdash K_i\, p_i$ is valid is given in Figure 5.20. Please study this proof in every detail and understand how it is just following the steps taken in the informal proof in Section 5.5.1.

The line 14 of the proof in Figure 5.20 applies several instances of $\wedge$i in sequence and is a legitimate step since the formulas in lines 11–13 had been shown 'for each' element in the respective set.

5.6 Exercises

Exercises 5.1

1. Think about the highly distributed computing environments of today with their dynamic communication and network topology. Come up with several kinds of modes of truth pertaining to statements made about such environments.
2. Let $\mathcal{M}$ be a model of first-order logic and let ϕ range over formulas of first-order logic. Discuss in what sense statements of the form 'Formula ϕ is true in model $\mathcal{M}$.' express a mode of truth.

Exercises 5.2

1. Consider the Kripke model $\mathcal{M}$ depicted in Figure 5.5.
   (a) For each of the following, determine whether it holds:
      i. $a \Vdash p$
      ii. $a \Vdash \Box\neg q$
      * iii. $a \Vdash q$
      * iv. $a \Vdash \Box\Box q$
      v. $a \Vdash \Diamond p$
      * vi. $a \Vdash \Box\Diamond\neg q$
      vii. $c \Vdash \Diamond\top$
      viii. $d \Vdash \Diamond\top$
      ix. $d \Vdash \Box\Box q$
      * x. $c \Vdash \Box\bot$
      xi. $b \Vdash \Box\bot$
      xii. $a \Vdash \Diamond\Diamond(p \wedge q) \wedge \Diamond\top$.
   (b) Find for each of the following a world which satisfies it:
      i. $\Box\neg p \wedge \Box\Box\neg p$
      ii. $\Diamond q \wedge \neg\Box q$
      * iii. $\Diamond p \vee \Diamond q$
      * iv. $\Diamond(p \vee \Diamond q)$
      v. $\Box p \vee \Box\neg p$
      vi. $\Box(p \vee \neg p)$.
   (c) For each formula of the previous item, find a world which does not satisfy the formula.
2. Find a Kripke model $\mathcal{M}$ and a formula scheme which is not satisfied in $\mathcal{M}$, but which has true instances in $\mathcal{M}$.

3. Consider the Kripke model $\mathcal{M} = (W, R, L)$ where $W = \{a, b, c, d, e\}$; $R = \{(a, c), (a, e), (b, a), (b, c), (d, e), (e, a)\}$; and $L(a) = \{p\}$, $L(b) = \{p, q\}$, $L(c) = \{p, q\}$, $L(d) = \{q\}$ and $L(e) = \emptyset$.
   (a) Draw a graph for $\mathcal{M}$.
   (b) Investigate which of the formulas in exercise 1(b) on page 350 have a world which satisfies it.
4. (a) Think about what you have to do to decide whether $p \to \Box\Diamond q$ is true in a model.
   * (b) Find a model in which it is true and one in which it is false.
5. For each of the following pairs of formulas, can you find a model and a world in it which distinguishes them, i.e. makes one of them true and one false? In that case, you are showing that they do not entail each other. If you cannot, it might mean that the formulas are equivalent. Justify your answer.
   (a) $\Box p$ and $\Box\Box p$
   (b) $\Box\neg p$ and $\neg\Diamond p$
   (c) $\Box(p \wedge q)$ and $\Box p \wedge \Box q$
   * (d) $\Diamond(p \wedge q)$ and $\Diamond p \wedge \Diamond q$
   (e) $\Box(p \vee q)$ and $\Box p \vee \Box q$
   * (f) $\Diamond(p \vee q)$ and $\Diamond p \vee \Diamond q$
   (g) $\Box(p \to q)$ and $\Box p \to \Box q$
   (h) $\Diamond\top$ and $\top$
   (i) $\Box\top$ and $\top$
   (j) $\Diamond\bot$ and $\bot$.
6. Show that the following formulas of basic modal logic are valid:
   * (a) $\Box(\phi \wedge \psi) \leftrightarrow (\Box\phi \wedge \Box\psi)$
   (b) $\Diamond(\phi \vee \psi) \leftrightarrow (\Diamond\phi \vee \Diamond\psi)$
   * (c) $\Box\top \leftrightarrow \top$
   (d) $\Diamond\bot \leftrightarrow \bot$
   (e) $\Diamond\top \to (\Box\phi \to \Diamond\phi)$
7. Inspect Definition 5.4. We said that we defined $x \Vdash \phi$ by structural induction on ϕ. Is this really correct? Note the implicit definition of a second relation $x \nVdash \phi$. Why is this definition still correct and in what sense does it still rely on structural induction?

Exercises 5.3

1. For which of the readings of $\Box$ in Table 5.7 are the formulas below valid?
   * (a) $(\phi \to \Box\phi) \to (\phi \to \Diamond\phi)$
   (b) $(\Box\phi \to (\phi \wedge \Box\Box\phi \wedge \Diamond\phi)) \to ((\Box\phi \to (\phi \wedge \Box\Box\phi)) \wedge (\Diamond\phi \to \Box\Diamond\phi))$.
2. Dynamic logic: Let P range over the programs of our core language in Chapter 4. Consider a modal logic whose modal operators are $\langle P\rangle$ and $[P]$ for all such programs P. Evaluate such formulas in stores l as in Definition 4.3 (page 264).

The relation $l \vDash \langle P \rangle \phi$ holds iff program P has some execution beginning in store l and terminating in a store satisfying ϕ.

* (a) Given that $\neg\langle P\rangle\neg$ equals $[P]$, spell out the meaning of $[P]$.

(b) Say that ϕ is valid iff it holds in all suitable stores l. State the total correctness of a Hoare triple as a validity problem in this modal logic.

3. For all binary relations R below, determine which of the properties reflexive through to total from page 320 apply to R where $R(x, y)$ means that

* (a) x is strictly less than y, where x and y range over all natural numbers $n \geq 1$

(b) x divides y, where x and y range over integers – e.g. 5 divides 15, whereas 7 does not

(c) x is a brother of y

* (d) there exist positive real numbers a and b such that x equals $a \cdot y + b$, where x and y range over real numbers.

* 4. Prove the Fact 5.16.

5. Prove the informal claim made in item 2 of Example 5.12 by structural induction on formulas in (5.1).

6. Prove Theorem 5.17. Use mathematical induction on the length of the sequence of negations and modal operators. Note that this requires a case analysis over the topmost operator other than a negation, or a modality.

7. Prove Theorem 5.14, but for the case in which R is reflexive, or transitive.

8. Find a Kripke model in which all worlds satisfy $\neg p \vee q$, but at least one world does not satisfy $\neg q \vee p$; i.e. show that the scheme $\neg\phi \vee \psi$ is not satisfied.

9. Below you find a list of sequents $\Gamma \vdash \phi$ in propositional logic. Find out whether you can prove them without the use of the rules PBC, LEM and $\neg\neg$e. If you cannot succeed, then try to construct a model $\mathcal{M} = (W, R, L)$ for intuitionistic propositional logic such that one of its worlds satisfies all formulas in Γ, but does not satisfy ϕ. Assuming soundness, this would guarantee that the sequent in question does not have a proof in intuitionistic propositional logic.

* (a) $\vdash (p \to q) \vee (q \to r)$

(b) The proof rule MT: $p \to q, \neg q \vdash \neg p$

(c) $\neg p \vee q \vdash p \to q$

(d) $p \to q \vdash \neg p \vee q$

(e) The proof rule $\neg\neg$e: $\neg\neg p \vdash p$

* (f) The proof rule $\neg\neg$i: $p \vdash \neg\neg p$.

10. Prove that the natural deduction rules for propositional logic without the rules $\neg\neg$e, LEM and PBC are sound for the possible world semantics of intuitionistic propositional logic. Why does this show that the excluded rules cannot be implemented using the remaining ones?

11. Interpreting $\Box\phi$ as 'agent Q believes ϕ,' explain the meaning of the following formula schemes:

(a) $\Box\phi \to \Diamond\phi$

* (b) $\Box\phi \vee \Box\neg\phi$

(c) $\Box(\phi \to \psi) \wedge \Box\phi \to \Box\psi$.

12. In the second row of Table 5.7, we adopted the convention that the future excludes the present. Which formula schemes would be satisfied in that row if instead we adopted the more common convention that the future includes the present?
13. Consider the properties in Table 5.12. Which ones should we accept if we read $\Box$ as
    * (a) knowledge
    (b) belief
    * (c) 'always in the future?'
14. Find a frame which is reflexive, transitive, but not symmetric. Show that your frame does not satisfy the formula $p \to \Box\Diamond p$, by providing a suitable labelling function and choosing a world which refutes $p \to \Box\Diamond p$. Can you find a labelling function and world which does satisfy $p \to \Box\Diamond p$ in your frame?
15. Give two examples of frames which are Euclidean – i.e. their accessibility relation is Euclidean – and two which are not. Explain intuitively why $\Diamond p \to \Box\Diamond p$ holds on the first two, but not on the latter two.
16. For each of the following formulas, find the property of R which corresponds to it.
    (a) $\phi \to \Box\phi$
    * (b) $\Box\bot$
    * (c) $\Diamond\Box\phi \to \Box\Diamond\phi$.

* 17. Find a formula whose corresponding property is density: for all $x, z \in W$ such that $R(x, z)$, there exists $y \in W$ such that $R(x, y)$ and $R(y, z)$.
18. The modal logic KD45 is used to model belief; see Table 5.12 for the axiom schemes D, 4, and 5.
    (a) Explain how it differs from KT45.
    (b) Show that $\vDash_{\mathrm{KD45}} \Box p \to \Diamond p$ is valid. What is the significance of this, in terms of knowledge and belief?
    (c) Explain why the condition of seriality is relevant to belief.
19. Recall Definition 5.7. How would you define $\equiv_L$ for a modal logic L?

Exercises 5.4

1. Find natural deduction proofs for the following sequents over the basic modal logic K.
    * (a) $\vdash_K \Box(p \to q) \vdash \Box p \to \Box q$
    (b) $\vdash_K \Box(p \to q) \vdash \Diamond p \to \Diamond q$
    * (c) $\vdash_K \vdash \Box(p \to q) \wedge \Box(q \to r) \to \Box(p \to r)$
    (d) $\vdash_K \Box(p \wedge q) \vdash \Box p \wedge \Box q$
    (e) $\vdash_K \vdash \Diamond\top \to (\Box p \to \Diamond p)$
    * (f) $\vdash_K \Diamond(p \to q) \vdash \Box p \to \Diamond q$
    (g) $\vdash_K \Diamond(p \vee q) \vdash \Diamond p \vee \Diamond q$.

2. Find natural deduction proofs for the following, in modal logic KT45.
   (a) $p \to \Box\Diamond p$
   (b) $\Box\Diamond p \leftrightarrow \Diamond p$
 * (c) $\Diamond\Box p \leftrightarrow \Box p$
   (d) $\Box(\Box p \to \Box q) \vee \Box(\Box q \to \Box p)$
   (e) $\Box(\Diamond p \to q) \leftrightarrow \Box(p \to \Box q)$.
3. Study the proofs you gave for the previous exercise to see whether any of these formula schemes could be valid in basic modal logic. Inspect where and how these proofs used the axioms T, 4 and 5 to see whether you can find a counter example, i.e. a Kripke model and a world which does not satisfy the formula.
4. Provide a sketch of an argument which shows that the natural deduction rules for basic modal logic are sound with respect to the semantics $x \Vdash \phi$ over Kripke structures.

Exercises 5.5

1. This exercise is about the wise-men puzzle. Justify your answers.
   (a) Each man is asked the question 'Do you know the colour of your hat?' Suppose that the first man says 'no,' but the second one says 'yes.' Given this information together with the common knowledge, can we infer the colour of his hat?
   (b) Can we predict whether the third man will now answer 'yes' or 'no?'
   (c) What would be the situation if the third man were blind? What about the first man?
2. This exercise is about the muddy-children puzzle. Suppose $k = 4$, say children a, b, c and d have mud on their foreheads. Explain why, before the father's announcement, it is not common knowledge that someone is dirty.
3. Write formulas for the following:
   (a) Agent 1 knows that p.
   (b) Agent 1 knows that p or q.
 * (c) Agent 1 knows p or agent 1 knows q.
   (d) Agent 1 knows whether p.
   (e) Agent 1 doesn't know whether p or q.
   (f) Agent 1 knows whether agent 2 knows p.
 * (g) Agent 1 knows whether agent 2 knows whether p.
   (h) No-one knows p.
   (i) Not everyone knows whether p.
   (j) Anyone who knows p knows q.
 * (k) Some people know p but don't know q.
   (l) Everyone knows someone who knows p.
4. Determine which of the following hold in the Kripke model of Figure 5.13 and justify your answer:

(a) $x_1 \Vdash K_1\, p$
(b) $x_3 \Vdash K_1\,(p \vee q)$
(c) $x_1 \Vdash K_2\, q$
* (d) $x_3 \Vdash E(p \vee q)$
(e) $x_1 \Vdash Cq$
(f) $x_1 \Vdash D_{\{1,3\}}p$
(g) $x_1 \Vdash D_{\{1,2\}}p$
(h) $x_6 \Vdash E\neg q$
* (i) $x_6 \Vdash C\neg q$
(j) $x_6 \Vdash C_{\{3\}}\neg q$.

5. For each of the following formulas, show that it is not valid by finding a Kripke model with a world not satisfying the formula:
   (a) $E_G\, \phi \to E_G E_G\, \phi$
   (b) $\neg E_G\, \phi \to E_G \neg E_G\, \phi$.
   Explain why these two Kripke models show that the union of equivalence relations is not necessarily an equivalence relation.
* 6. Explain why $C_G\, \phi \to C_G C_G\, \phi$ and $\neg C_G\, \phi \to C_G \neg C_G\, \phi$ are valid.
7. Prove the second part of Theorem 5.26.
8. Recall Section 3.7. Can you specify a monotone function over the power set of possible worlds which computes the set of worlds satisfying $C_G\, \phi$? Is this a least, or a greatest, fixed point?
9. Use the natural deduction rules for propositional logic to justify the proof steps below which are only annotated with 'prop.'
   (a) Line 11 in Figure 5.15.
   (b) Lines 10, 18 and 23 of the proof in Figure 5.16. Of course this requires three separate proofs.
   (c) Line 14 of the proof in Figure 5.18.
   (d) Line 10 of the proof in Figure 5.19.
10. Using the natural deduction rules for $\mathrm{KT45}^n$, prove the validity of
   (a) $K_i\,(p \wedge q) \leftrightarrow K_i\, p \wedge K_i\, q$
   (b) $C(p \wedge q) \leftrightarrow Cp \wedge Cq$
   * (c) $K_i\, Cp \leftrightarrow Cp$
   (d) $C\, K_i\, p \leftrightarrow Cp$
   * (e) $\neg\phi \to K_i \neg K_i\, \phi$.
   Explain what this formula means in terms of knowledge. Do you believe it?
   (f) $\neg\phi \to K_1 K_2 \neg K_2 K_1\, \phi$
   * (g) $\neg K_1 \neg K_1 \phi \leftrightarrow K_1\, \phi$.
11. Do a natural deduction proof for a simpler version of the wise-men problem: There are two wise men; as usual, they can see each other's hats but not their own. It is common knowledge that there's only one white hat available and two red ones. So at least one of the men is wearing a red one. Man 1 informs the second that he doesn't know which hat he is wearing. Man 2 says, 'Aha, then I must be wearing a red hat.'

(a) Justify man 2's conclusion informally.

(b) Let p_1, p_2 respectively, mean man 1, 2 respectively, is wearing a red hat. So $\neg p_1, \neg p_2$ mean they (respectively) are wearing a white one. Informally justify each of the following premises in terms of the description of the problem:

i. $K_2 K_1 (p_1 \vee p_2)$

ii. $K_2(\neg p_2 \rightarrow K_1 \neg p_2)$

iii. $K_2 \neg K_1 p_1$.

(c) Using natural deduction, prove from these premises that $K_2 p_2$.

(d) Show that the third premise was essential, by exhibiting a model/world which satisfies the first two, but not the conclusion.

(e) Now is it easy to answer questions like 'If man 2 were blind would he still be able to tell?' and 'if man 1 were blind, would man 2 still be able to tell?'?

12. Recall our informal discussion on positive-knowledge formulas and negative-knowledge formulas. Give formal definitions of these notions.

5.7 Bibliographic notes

The first systematic approaches to modal logic were made by C. I. Lewis in the 1950s. The possible-worlds approach, which greatly simplified modal logic and is now almost synonymous with it, was invented by S. Kripke. Books devoted to modal logic include [Che80, Gol87, Pop94], where extensive references to the literature may be found. All these books discuss the soundness and completeness of proof calculi for modal logics. They also investigate which modal logics have the *finite-model property*: if a sequent does not have a proof, there is a finite model which demonstrates that. Not all modal logics enjoy this property, which is important for decidability. Intuitionistic propositional logic has the finite-model property; an animation which generates such finite models (called PORGI) is available from A. Stoughton's website[2].

The idea of using modal logic to reason about knowledge is due to J. Hintikka. A great deal of work on applying modal logic to multi-agent systems has been done in [FHMV95] and [MvdH95] and other work by those authors. Many examples in this chapter are taken from this literature (some of them are attributed to other people there), though our treatment of them is original.

The natural deduction proof system for modal logic presented in this chapter is based on ideas in [Fit93].

[2] `www.cis.ksu.edu/~allen/porgi.html`

An application of the modal logic KT4 (more precisely, its fragment without negation) as a type system for staged computation in a functional programming language can be found in [DP96].

We should stress that our framework was deliberately 'classical;' the thesis [Sim94] is a good source for discussions of intuitionistic modal logics; it also contains a gentle introduction to basic first-order modal logic.

6
Binary decision diagrams

6.1 Representing boolean functions

Boolean functions are an important descriptive formalism for many hardware and software systems, such as synchronous and asynchronous circuits, reactive systems and finite-state programs. Representing those systems in a computer in order to reason about them requires an efficient representation for boolean functions. We look at such a representation in this chapter and describe in detail how the systems discussed in Chapter 3 can be verified using the representation.

Definition 6.1 A boolean variable x is a variable ranging over the values 0 and 1. We write $x_1, x_2, \ldots$ and $x, y, z, \ldots$ to denote boolean variables. We define the following functions on the set $\{0, 1\}$:

- $\overline{0} \stackrel{\text{def}}{=} 1$ and $\overline{1} \stackrel{\text{def}}{=} 0$;
- $x \cdot y \stackrel{\text{def}}{=} 1$ if x and y have value 1; otherwise $x \cdot y \stackrel{\text{def}}{=} 0$;
- $x + y \stackrel{\text{def}}{=} 0$ if x and y have value 0; otherwise $x + y \stackrel{\text{def}}{=} 1$;
- $x \oplus y \stackrel{\text{def}}{=} 1$ if exactly one of x and y equals 1.

A boolean function f of n arguments is a function from $\{0, 1\}^n$ to $\{0, 1\}$. We write $f(x_1, x_2, \ldots, x_n)$, or $f(V)$, to indicate that a syntactic representation of f depends on the boolean variables in V only.

Note that $\cdot$, $+$ and $\oplus$ are boolean functions with two arguments, whereas $\bar{\ }$ is a boolean function that takes one argument. The binary functions $\cdot$, $+$ and $\oplus$ are written in infix notation instead of prefix; i.e. we write $x + y$ instead of $+(x, y)$, etc.

Example 6.2 In terms of the four functions above, we can define other boolean functions such as

(1) $f(x,y) \stackrel{\text{def}}{=} x \cdot (y + \overline{x})$
(2) $g(x,y) \stackrel{\text{def}}{=} x \cdot y + (1 \oplus \overline{x})$
(3) $h(x,y,z) \stackrel{\text{def}}{=} x + y \cdot (x \oplus \overline{y})$
(4) $k() \stackrel{\text{def}}{=} 1 \oplus (0 \cdot \overline{1})$.

6.1.1 Propositional formulas and truth tables

Truth tables and *propositional formulas* are two different representations of boolean functions. In propositional formulas, $\land$ denotes $\cdot$, $\lor$ denotes $+$, $\neg$ denotes $\bar{}$ and $\top$ and $\bot$ denote 1 and 0, respectively.

Boolean functions are represented by truth tables in the obvious way; for example, the function $f(x,y) \stackrel{\text{def}}{=} \overline{x+y}$ is represented by the truth table on the left:

x	y	$f(x,y)$
1	1	0
0	1	0
1	0	0
0	0	1

p	q	ϕ
T	T	F
F	T	F
T	F	F
F	F	T

On the right, we show the same truth table using the notation of Chapter 1; a formula having this truth table is $\neg(p \lor q)$. In this chapter, we may mix these two notational systems of boolean formulas and formulas of propositional logic whenever it is convenient. You should be able to translate expressions easily from one notation to the other and vice versa.

As representations of boolean functions, propositional formulas and truth tables have different advantages and disadvantages. Truth tables are very space-inefficient: if one wanted to model the functionality of a sequential circuit by a boolean function of 100 variables (a small chip component would easily require this many variables), then the truth table would require 2^{100} (which is more than 10^{30}) lines. Alas, there is not enough storage space (whether paper or particle) in the universe to record the information of 2^{100} different bit vectors of length 100. Although they are space inefficient, operations on truth tables are simple. Once you have computed a truth table, it is easy to see whether the boolean function represented is satisfiable: you just look to see if there is a 1 in the last column of the table.

Comparing whether two truth tables represent the same boolean function also seems easy: assuming the two tables are presented with the same order

of valuations, we simply check that they are identical. Although these operations seem simple, however, they are computationally intractable because of the fact that the number of lines in the truth table is exponential in the number of variables. Checking satisfiability of a function with n atoms requires of the order of 2^n operations if the function is represented as a truth table. We conclude that checking satisfiability and equivalence is highly inefficient with the truth-table representation.

Representation of boolean functions by propositional formulas is slightly better. Propositional formulas often provide a wonderfully compact and efficient presentation of boolean functions. A formula with 100 variables might only be about 200–300 characters long. However, deciding whether an arbitrary propositional formula is satisfiable is a famous problem in computer science: no efficient algorithms for this task are known, and it is strongly suspected that there aren't any. Similarly, deciding whether two arbitrary propositional formulas f and g denote the same boolean function is suspected to be exponentially expensive.

It is straightforward to see how to perform the boolean operations $\cdot$, $+$, $\oplus$ and $\bar{\ }$ on these two representations. In the case of truth tables, they involve applying the operation to each line; for example, given truth tables for f and g over the same set of variables (and in the same order), the truth table for $f \oplus g$ is obtained by applying $\oplus$ to the truth value of f and g in each line. If f and g do not have the same set of arguments, it is easy to pad them out by adding further arguments. In the case of representation by propositional formulas, the operations $\cdot$, $\oplus$, etc., are simply syntactic manipulations. For example, given formulas ϕ and ψ representing the functions f and g, the formulas representing $f \cdot g$ and $f \oplus g$ are, respectively, $\phi \wedge \psi$ and $(\phi \wedge \neg\psi) \vee (\neg\phi \wedge \psi)$.

We could also consider representing boolean functions by various subclasses of propositional formulas, such as conjunctive and disjunctive normal forms. In the case of disjunctive normal form (DNF, in which a formula is a disjunction of conjunctions of literals), the representation is sometimes compact, but in the worst cases it can be very lengthy. Checking satisfiability is a straightforward operation, however, because it is sufficient to find a disjunct which does not have two complementary literals. Unfortunately, there is not a similar way of checking validity. Performing $+$ on two formulas in DNF simply involves inserting $\vee$ between them. Performing $\cdot$ is more complicated; we cannot simply insert $\wedge$ between the two formulas, because the result will not in general be in DNF, so we have to perform lengthy applications of the distributivity rule $\phi \wedge (\psi_1 \vee \psi_2) \equiv (\phi \wedge \psi_1) \vee (\phi \wedge \psi_1)$. Computing the negation of a DNF formula is also expensive. The DNF formula ϕ may be

Representation of boolean functions	compact?	test for satisf'ty	test for validity	boolean operations $\cdot$	boolean operations $+$	boolean operations $\bar{}$
Prop. formulas	often	hard	hard	easy	easy	easy
Formulas in DNF	sometimes	easy	hard	hard	easy	hard
Formulas in CNF	sometimes	hard	easy	easy	hard	hard
Ordered truth tables	never	hard	hard	hard	hard	hard
Reduced OBDDs	often	easy	easy	medium	medium	easy

Figure 6.1. Comparing efficiency of five representations of boolean formulas.

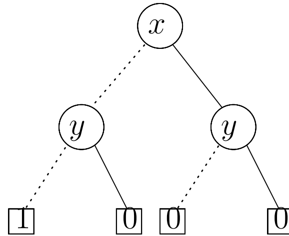

Figure 6.2. An example of a binary decision tree.

quite short, whereas the length of the disjunctive normal form of $\neg\phi$ can be exponential in the length of ϕ.

The situation for representation in conjunctive normal form is the dual. A summary of these remarks is contained in Figure 6.1 (for now, please ignore the last row).

6.1.2 Binary decision diagrams

Binary decision diagrams (BDDs) are another way of representing boolean functions. A certain class of such diagrams will provide the implementational framework for our symbolic model-checking algorithm. Binary decision diagrams were first considered in a simpler form called *binary decision trees.* These are trees whose non-terminal nodes are labelled with boolean variables $x, y, z, \ldots$ and whose terminal nodes are labelled with either 0 or 1. Each non-terminal node has two edges, one dashed line and one solid line. In Figure 6.2 you can see such a binary decision tree with two layers of variables x and y.

Definition 6.3 Let T be a finite binary decision tree. Then T determines a unique boolean function of the variables in non-terminal nodes, in the following way. Given an assignment of 0s and 1s to the boolean variables

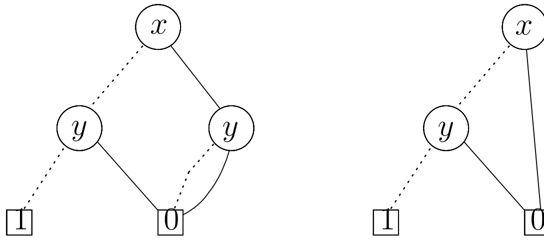

Figure 6.3. (a) Sharing the terminal nodes of the binary decision tree in Figure 6.2; (b) further optimisation by removing a redundant decision point.

occurring in T, we start at the root of T and take the dashed line whenever the value of the variable at the current node is 0; otherwise, we travel along the solid line. The function value is the value of the terminal node we reach.

For example, the binary decision tree of Figure 6.2 represents a boolean function $f(x, y)$. To find $f(0, 1)$, start at the root of the tree. Since the value of x is 0 we follow the dashed line out of the node labelled x and arrive at the leftmost node labelled y. Since y's value is 1, we follow the solid line out of that y-node and arrive at the leftmost terminal node labelled 0. Thus, $f(0, 1)$ equals 0. In computing $f(0, 0)$, we similarly travel down the tree, but now following two dashed lines to obtain 1 as a result. You can see that the two other possibilities result in reaching the remaining two terminal nodes labelled 0. Thus, this binary decision tree computes the function $f(x, y) \stackrel{\text{def}}{=} \overline{x + y}$.

Binary decision trees are quite close to the representation of boolean functions as truth tables as far as their sizes are concerned. If the root of a binary decision tree is an x-node then it has two subtrees (one for the value of x being 0 and another one for x having value 1). So if f depends on n boolean variables, the corresponding binary decision tree will have at least $2^{n+1} - 1$ nodes (see exercise 5 on page 399). Since f's truth table has 2^n lines, we see that decision trees as such are not a more compact representation of boolean functions. However, binary decision trees often contain some redundancy which we can exploit.

Since 0 and 1 are the only terminal nodes of binary decision trees, we can optimise the representation by having pointers to just one copy of 0 and one copy of 1. For example, the binary decision tree in Figure 6.2 can be optimised in this way and the resulting structure is depicted in Figure 6.3(a). Note that we saved storage space for two redundant terminal 0-nodes, but that we still have as many edges (pointers) as before.

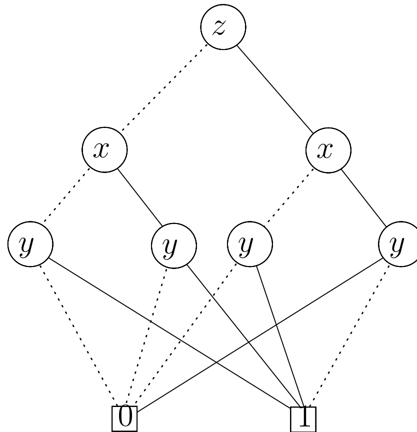

Figure 6.4. A BDD with duplicated subBDDs.

A second optimisation we can do is to remove unnecessary decision points in the tree. In Figure 6.3(a), the right-hand y is unnecessary, because we go to the same place whether it is 0 or 1. Therefore the structure could be further reduced, to the one shown on the right, (b).

All these structures are examples of *binary decision diagrams* (BDDs). They are more general than binary decision trees; the sharing of the leaves means they are not trees. As a third optimisation, we also allow subBDDs to be shared. A subBDD is the part of a BDD occurring below a given node. For example, in the BDD of Figure 6.4, the two inner y-nodes perform the same role, because the subBDDs below them have the same structure. Therefore, one of them could be removed, resulting in the BDD in Figure 6.5(a). Indeed, the left-most y-node could also be merged with the middle one; then the x-node above both of them would become redundant. Removing it would result in the BDD on the right of Figure 6.5.

To summarise, we encountered three different ways of reducing a BDD to a more compact form:

C1. Removal of duplicate terminals. If a BDD contains more than one terminal 0-node, then we redirect all edges which point to such a 0-node to just one of them. We proceed in the same way with terminal nodes labelled with 1.

C2. Removal of redundant tests. If both outgoing edges of a node n point to the same node m, then we eliminate that node n, sending all its incoming edges to m.

C3. Removal of duplicate non-terminals. If two distinct nodes n and m in the BDD are the roots of structurally identical subBDDs, then we

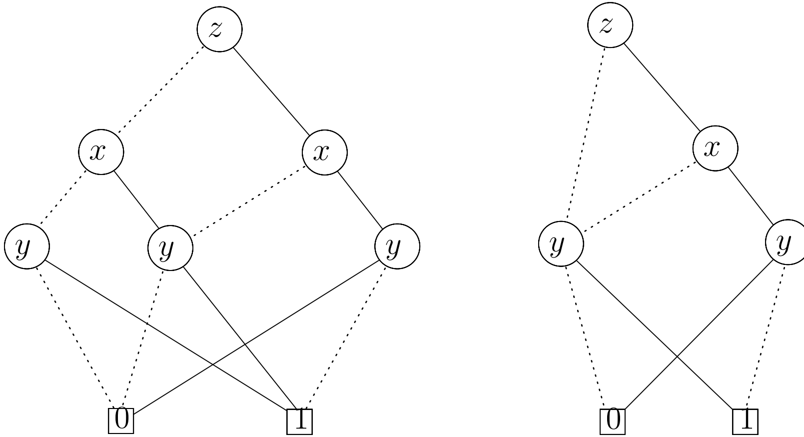

Figure 6.5. The BDD of Figure 6.4: (a) after removal of one of the duplicate y-nodes; (b) after removal of another duplicate y-node and then a redundant x-decision point.

eliminate one of them, say m, and redirect all its incoming edges to the other one.

Note that C1 is a special case of C3. In order to define BDDs precisely, we need a few auxiliary notions.

Definition 6.4 A directed graph is a set G and a binary relation $\to$ on G: $\to \subseteq G \times G$. A cycle in a directed graph is a finite path in that graph that begins and ends at the same node, i.e. a path of the form $v_1 \to v_2 \to \cdots \to v_n \to v_1$. A directed acyclic graph (dag) is a directed graph that does not have any cycles. A node of a dag is initial if there are no edges pointing to that node. A node is called terminal if there are no edges out of that node.

The directed graph in Figure 3.3 on page 179 has cycles, for example the cycle $s_0 \to s_1 \to s_0$, and is not a dag. If we interpret the links in BDDs (whether solid or dashed) as always going in a downwards direction, then the BDDs of this chapter are also directed graphs. They are also acyclic and have a unique initial node. The optimisations C1–C3 preserve the property of being a dag; and fully reduced BDDs have precisely two terminal nodes. We now formally define BDDs as certain kinds of dags:

Definition 6.5 A binary decision diagram (BDD) is a finite dag with a unique initial node, where all terminal nodes are labelled with 0 or 1 and all non-terminal nodes are labelled with a boolean variable. Each

Figure 6.6. The BDDs (a) B_0, representing the constant 0 boolean function; similarly, the BDD B_1 has only one node 1 and represents the constant 1 boolean function; and (b) B_x, representing the boolean variable x.

non-terminal node has exactly two edges from that node to others: one labelled 0 and one labelled 1 (we represent them as a dashed line and a solid line, respectively).

A BDD is said to be reduced if none of the optimisations C1–C3 can be applied (i.e. no more reductions are possible).

All the decision structures we have seen in this chapter (Figures 6.2–6.5) are BDDs, as are the constant functions B_0 and B_1, and the function B_x from Figure 6.6. If B is a BDD where $V = \{x_1, x_2, \dots, x_n\}$ is the set of labels of non-terminal nodes, then B determines a boolean function $f(V)$ in the same way as binary decision trees (see Definition 6.3): given an assignment of 0s and 1s to the variables in V, we compute the value of f by starting with the unique initial node. If its variable has value 0, we follow the dashed line; otherwise we take the solid line. We continue for each node until we reach a terminal node. Since the BDD is finite by definition, we eventually reach a terminal node which is labelled with 0 or 1. That label is the result of f for that particular assignment of truth values.

The definition of a BDD does not prohibit that a boolean variable occur more than once on a path in the dag. For example, consider the BDD in Figure 6.7.

Such a representation is wasteful, however. The solid link from the leftmost x to the 1-terminal is never taken, for example, because one can only get to that x-node when x has value 0.

Thanks to the reductions C1–C3, BDDs can often be quite compact representations of boolean functions. Let us consider how to check satisfiability and perform the boolean operations on functions represented as BDDs. A BDD represents a satisfiable function if a 1-terminal node is reachable from the root along a *consistent path* in a BDD which represents it. A consistent path is one which, for every variable, has only dashed lines or only solid lines leaving nodes labelled by that variable. (In other words, we cannot assign

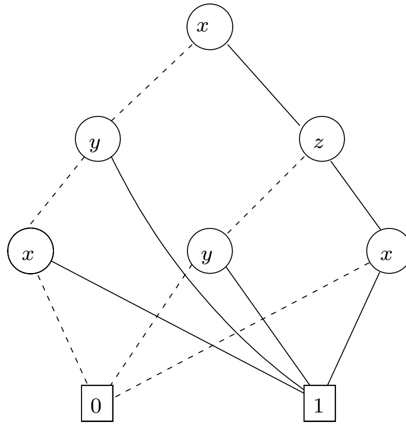

Figure 6.7. A BDD where some boolean variables occur more than once on an evaluation path.

a variable the values 0 and 1 simultaneously.) Checking validity is similar, but we check that no 0-terminal is reachable by a consistent path.

The operations $\cdot$ and $+$ can be performed by 'surgery' on the component BDDs. Given BDDs B_f and B_g representing boolean functions f and g, a BDD representing $f \cdot g$ can be obtained by taking the BDD f and replacing all its 1-terminals by B_g. To see why this is so, consider how to get to a 1-terminal in the resulting BDD. You have to satisfy the requirements for getting to a 1 imposed by both of the BDDs. Similarly, a BDD for $f + g$ can be obtained by replacing all 0 terminals of B_f by B_g. Note that these operations are likely to generate BDDs with multiple occurrences of variables along a path. Later, in Section 6.2, we will see definitions of $+$ and $\cdot$ on BDDs that don't have this undesirable effect.

The complementation operation $\bar{\ }$ is also possible: a BDD representing $\overline{f}$ can be obtained by replacing all 0-terminals in B_f by 1-terminals and vice versa. Figure 6.8 shows the complement of the BDD in Figure 6.2.

6.1.3 Ordered BDDs

We have seen that the representation of boolean functions by BDDs is often compact, thanks to the sharing of information afforded by the reductions C1–C3. However, BDDs with multiple occurrences of a boolean variable along a path seem rather inefficient. Moreover, there seems no easy way to test for equivalence of BDDs. For example, the BDDs of Figures 6.7 and 6.9 represent the same boolean function (the reader should check this). Neither of them can be optimised further by applying the rules C1–C3. However,

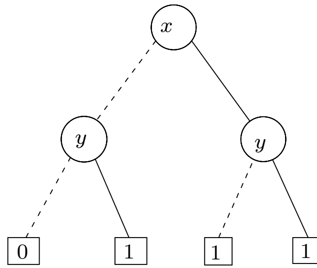

Figure 6.8. The complement of the BDD in Figure 6.2.

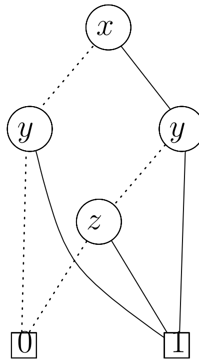

Figure 6.9. A BDD representing the same function as the BDD of Figure 6.7, but having the variable ordering $[x, y, z]$.

testing whether they denote the same boolean function seems to involve as much computational effort as computing the entire truth table for $f(x, y, z)$.

We can improve matters by imposing an ordering on the variables occurring along any path. We then adhere to that same ordering for all the BDDs we manipulate.

Definition 6.6 Let $[x_1, \ldots, x_n]$ be an ordered list of variables without duplications and let B be a BDD all of whose variables occur somewhere in the list. We say that B has the ordering $[x_1, \ldots, x_n]$ if all variable labels of B occur in that list and, for every occurrence of x_i followed by x_j along any path in B, we have $i < j$.

An ordered BDD (OBDD) is a BDD which has an ordering for some list of variables.

Note that the BDDs of Figures 6.3(a,b) and 6.9 are ordered (with ordering $[x, y]$). We don't insist that every variable in the list is used in the paths. Thus, the OBDDs of Figures 6.3 and 6.9 have the ordering $[x, y, z]$ and so

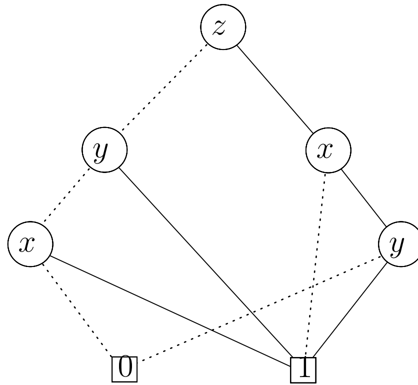

Figure 6.10. A BDD which does not have an ordering of variables.

does any list having x, y and z in it in that order, such as $[u, x, y, v, z, w]$ and $[x, u, y, z]$. Even the BDDs B_0 and B_1 in Figure 6.6 are OBDDs, a suitable ordering list being the empty list (there are no variables), or indeed *any* list. The BDD B_x of Figure 6.6(b) is also an OBDD, with any list containing x as its ordering.

The BDD of Figure 6.7 is not ordered. To see why this is so, consider the path taken if the values of x and y are 0. We begin with the root, an x-node, and reach a y-node and then an x-node again. Thus, no matter what list arrangement we choose (remembering that no double occurrences are allowed), this path violates the ordering condition. Another example of a BDD that is not ordered can be seen in Figure 6.10. In that case, we cannot find an order since the path for $(x, y, z) \Rightarrow (0, 0, 0)$ – meaning that x, y and z are assigned 0 – shows that y needs to occur before x in such a list, whereas the path for $(x, y, z) \Rightarrow (1, 1, 1)$ demands that x be before y.

It follows from the definition of OBDDs that one cannot have multiple occurrences of any variable along a path.

When operations are performed on two OBDDs, we usually require that they have *compatible variable orderings.* The orderings of B_1 and B_2 are said to be compatible if there are no variables x and y such that x comes before y in the ordering of B_1 and y comes before x in the ordering of B_2. This commitment to an ordering gives us a unique representation of boolean functions as OBDDs. For example, the BDDs in Figures 6.8 and 6.9 have compatible variable orderings.

Theorem 6.7 The reduced OBDD representing a given function f is unique. That is to say, let B and B' be two reduced OBDDs with

compatible variable orderings. If B and B represent the same boolean function, then they have identical structure.

In other words, with OBDDs we cannot get a situation like the one encountered earlier, in which we have two distinct reduced BDDs which represent the same function, provided that the orderings are compatible. It follows that checking equivalence of OBDDs is immediate. Checking whether two OBDDs (having compatible orderings) represent the same function is simply a matter of checking whether they have the same structure[1].

A useful consequence of the theorem above is that, if we apply the reductions C1–C3 to an OBDD until no further reductions are possible, then we are guaranteed that the result is always the same reduced OBDD. The order in which we applied the reductions does not matter. We therefore say that OBDDs have a *canonical form*, namely their unique reduced OBDD. Most other representations (conjunctive normal forms, etc.) do not have canonical forms.

The algorithms for $\cdot$ and $+$ for BDDs, presented in Section 6.1.2, won't work for OBDDs as they may introduce multiple occurrences of the same variable on a path. We will soon develop more sophisticated algorithms for these operations on OBDDs, which exploit the compatible ordering of variables in paths.

OBDDs allow compact representations of certain classes of boolean functions which only have exponential representations in other systems, such as truth tables and conjunctive normal forms. As an example consider the *even parity function* $f_{\text{even}}(x_1, x_2, \ldots, x_n)$ which is defined to be 1 if there is an even number of variables x_i with value 1; otherwise, it is defined to be 0. Its representation as an OBDD requires only $2n + 1$ nodes. Its OBDD for $n = 4$ and the ordering $[x_1, x_2, x_3, x_4]$ can be found in Figure 6.11.

The impact of the chosen variable ordering The size of the OBDD representing the parity functions is independent of the chosen variable ordering. This is because the parity functions are themselves independent of the order of variables: swapping the values of any two variables does not change the value of the function; such functions are called symmetric.

However, in general the chosen variable ordering makes a significant difference to the size of the OBDD representing a given function. Consider the boolean function $(x_1 + x_2) \cdot (x_3 + x_4) \cdot \cdots \cdot (x_{2n-1} + x_{2n})$; it corresponds to a propositional formula in conjunctive normal form. If we choose the

[1] In an implementation this will amount to checking whether two pointers are equal.

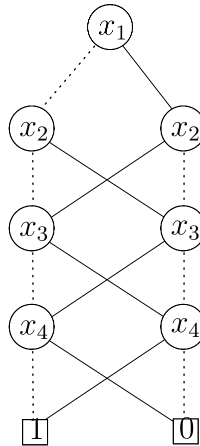

Figure 6.11. An OBDD for the even parity function for four bits.

'natural' ordering $[x_1, x_2, x_3, x_4, \dots]$, then we can represent this function as an OBDD with $2n + 2$ nodes. Figure 6.12 shows the resulting OBDD for $n = 3$. Unfortunately, if we choose instead the ordering

$$[x_1, x_3, \dots, x_{2n-1}, x_2, x_4, \dots, x_{2n}]$$

the resulting OBDD requires 2^{n+1} nodes; the OBDD for $n = 3$ can be seen in Figure 6.13.

The sensitivity of the size of an OBDD to the particular variable ordering is a price we pay for all the advantages that OBDDs have over BDDs. Although finding the optimal ordering is itself a computationally expensive problem, there are good heuristics which will usually produce a fairly good ordering. Later on we return to this issue in discussions of applications.

The importance of canonical representation The importance of having a canonical form for OBDDs in conjunction with an efficient test for deciding whether two reduced OBDDs are isomorphic cannot be overestimated. It allows us to perform the following tests:

Absence of redundant variables. If the value of the boolean function $f(x_1, x_2, \dots, x_n)$ does not depend on the value of x_i, then any reduced OBDD which represents f does not contain any x_i-node.

Test for semantic equivalence. If two functions $f(x_1, x_2, \dots, x_n)$ and $g(x_1, x_2, \dots, x_n)$ are represented by OBDDs B_f, respectively B_g, with a compatible ordering of variables, then we can efficiently decide whether f and g are semantically equivalent. We reduce B_f and B_g (if necessary); f

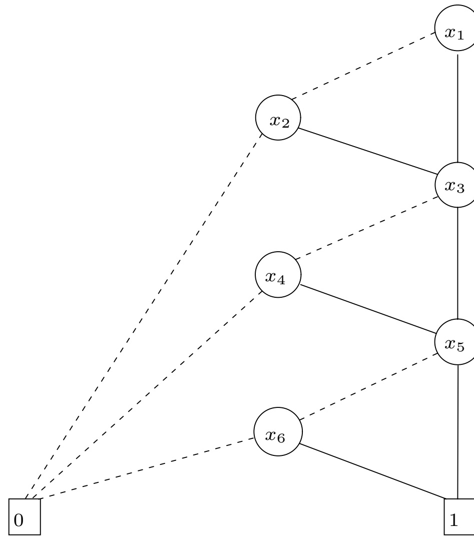

Figure 6.12. The OBDD for $(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_2, x_3, x_4, x_5, x_6]$.

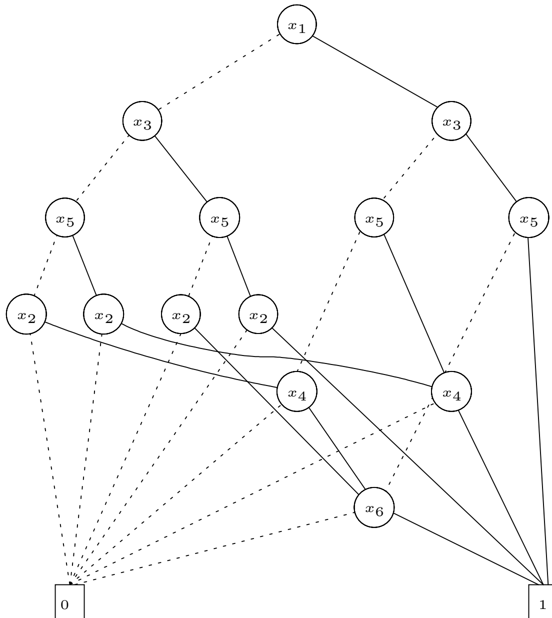

Figure 6.13. Changing the ordering may have dramatic effects on the size of an OBDD: the OBDD for $(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_3, x_5, x_2, x_4, x_6]$.

and g denote the same boolean functions if, and only if, the reduced OBDDs have identical structure.

Test for validity. We can test a function $f(x_1, x_2, \ldots, x_n)$ for validity (i.e. f always computes 1) in the following way. Compute a reduced OBDD for f. Then f is valid if, and only if, its reduced OBDD is B_1.

Test for implication. We can test whether $f(x_1, x_2, \ldots, x_n)$ implies $g(x_1, x_2, \ldots, x_n)$ (i.e. whenever f computes 1, then so does g) by computing the reduced OBDD for $f \cdot \overline{g}$. This is B_0 iff the implication holds.

Test for satisfiability. We can test a function $f(x_1, x_2, \ldots, x_n)$ for satisfiability (f computes 1 for at least one assignment of 0 and 1 values to its variables). The function f is satisfiable iff its reduced OBDD is not B_0.

6.2 Algorithms for reduced OBDDs

6.2.1 The algorithm `reduce`

The reductions C1–C3 are at the core of any serious use of OBDDs, for whenever we construct a BDD we will want to convert it to its reduced form. In this section, we describe an algorithm `reduce` which does this efficiently for ordered BDDs.

If the ordering of B is $[x_1, x_2, \ldots, x_l]$, then B has at most $l + 1$ layers. The algorithm `reduce` now traverses B layer by layer in a bottom-up fashion, beginning with the terminal nodes. In traversing B, it assigns an integer label $\mathrm{id}(n)$ to each node n of B, in such a way that the subOBDDs with root nodes n and m denote the same boolean function if, and only if, $\mathrm{id}(n)$ equals $\mathrm{id}(m)$.

Since `reduce` starts with the layer of terminal nodes, it assigns the first label (say #0) to the first 0-node it encounters. All other terminal 0-nodes denote the same function as the first 0-node and therefore get the same label (compare with reduction C1). Similarly, the 1-nodes all get the next label, say #1.

Now let us inductively assume that `reduce` has already assigned integer labels to all nodes of a layer $> i$ (i.e. all terminal nodes and x_j-nodes with $j > i$). We describe how nodes of layer i (i.e. x_i-nodes) are being handled.

Definition 6.8 Given a non-terminal node n in a BDD, we define $\mathrm{lo}(n)$ to be the node pointed to via the dashed line from n. Dually, $\mathrm{hi}(n)$ is the node pointed to via the solid line from n.

Let us describe how the labelling is done. Given an x_i-node n, there are three ways in which it may get its label:

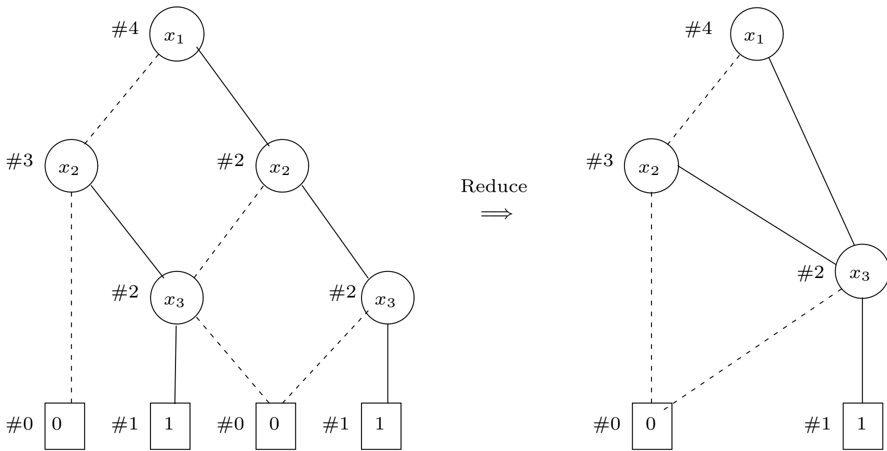

Figure 6.14. An example execution of the algorithm `reduce`.

- If the label $\mathrm{id}(\mathrm{lo}(n))$ is the same as $\mathrm{id}(\mathrm{hi}(n))$, then we set $\mathrm{id}(n)$ to be that label. That is because the boolean function represented at n is the same function as the one represented at $\mathrm{lo}(n)$ and $\mathrm{hi}(n)$. In other words, node n performs a redundant test and can be eliminated by reduction C2.
- If there is another node m such that n and m have the same variable x_i, and $\mathrm{id}(\mathrm{lo}(n)) = \mathrm{id}(\mathrm{lo}(m))$ and $\mathrm{id}(\mathrm{hi}(n)) = \mathrm{id}(\mathrm{hi}(m))$, then we set $\mathrm{id}(n)$ to be $\mathrm{id}(m)$. This is because the nodes n and m compute the same boolean function (compare with reduction C3).
- Otherwise, we set $\mathrm{id}(n)$ to the next unused integer label.

Note that only the last case creates a new label. Consider the OBDD in left side of Figure 6.14; each node has an integer label obtained in the manner just described. The algorithm `reduce` then finishes by redirecting edges bottom-up as outlined in C1–C3. The resulting reduced OBDD is in right of Figure 6.14. Since there are efficient bottom-up traversal algorithms for dags, `reduce` is an efficient operation in the number of nodes of an OBDD.

6.2.2 The algorithm `apply`

Another procedure at the heart of OBDDs is the algorithm `apply`. It is used to implement operations on boolean functions such as $+$, $\cdot$, $\oplus$ and complementation (via $f \oplus 1$). Given OBDDs B_f and B_g for boolean formulas f and g, the call `apply` (op, B_f, B_g) computes the reduced OBDD of the boolean formula $f \,\mathrm{op}\, g$, where op denotes any function from $\{0,1\} \times \{0,1\}$ to $\{0,1\}$.

The intuition behind the `apply` algorithm is fairly simple. The algorithm operates recursively on the structure of the two OBDDs:

1. let v be the variable highest in the ordering (=leftmost in the list) which occurs in B_f or B_g.
2. split the problem into two subproblems for v being 0 and v being 1 and solve recursively;
3. at the leaves, apply the boolean operation op directly.

The result will usually have to be reduced to make it into an OBDD. Some reduction can be done 'on the fly' in step 2, by avoiding the creation of a new node if both branches are equal (in which case return the common result), or if an equivalent node already exists (in which case, use it).

Let us make all this more precise and detailed.

Definition 6.9 Let f be a boolean formula and x a variable.

1. We denote by $f[0/x]$ the boolean formula obtained by replacing all occurrences of x in f by 0. The formula $f[1/x]$ is defined similarly. The expressions $f[0/x]$ and $f[1/x]$ are called restrictions of f.
2. We say that two boolean formulas f and g are semantically equivalent if they represent the same boolean function (with respect to the boolean variables that they depend upon). In that case, we write $f \equiv g$.

For example, if $f(x,y) \stackrel{\text{def}}{=} x \cdot (y + \overline{x})$, then $f[0/x](x,y)$ equals $0 \cdot (y + \overline{0})$, which is semantically equivalent to 0. Similarly, $f[1/y](x,y)$ is $x \cdot (1 + \overline{x})$, which is semantically equivalent to x.

Restrictions allow us to perform recursion on boolean formulas, by decomposing boolean formulas into simpler ones. For example, if x is a variable in f, then f is equivalent to $\overline{x} \cdot f[0/x] + x \cdot f[1/x]$. To see this, consider the case $x = 0$; the expression computes to $f[0/x]$. When $x = 1$ it yields $f[1/x]$. This observation is known as the *Shannon expansion*, although it can already be found in G. Boole's book *'The Laws of Thought'* from 1854.

Lemma 6.10 (Shannon expansion) For all boolean formulas f and all boolean variables x (even those not occurring in f) we have

$$f \equiv \overline{x} \cdot f[0/x] + x \cdot f[1/x]. \tag{6.1}$$

The function `apply` is based on the Shannon expansion for $f \operatorname{op} g$:

$$f \operatorname{op} g = \overline{x_i} \cdot (f[0/x_i] \operatorname{op} g[0/x_i]) + x_i \cdot (f[1/x_i] \operatorname{op} g[1/x_i]). \tag{6.2}$$

This is used as a control structure of `apply` which proceeds from the roots

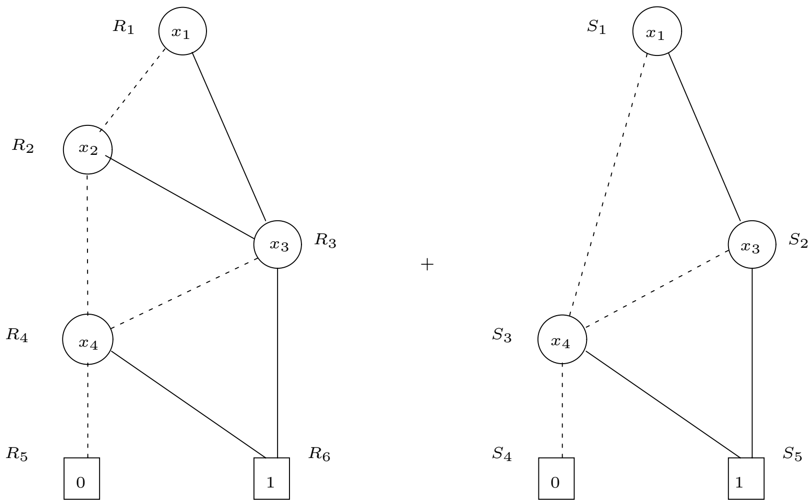

Figure 6.15. An example of two arguments for a call `apply` $(+, B_f, B_g)$.

of B_f and B_g downwards to construct nodes of the OBDD $B_{f\,\mathrm{op}\,g}$. Let r_f be the root node of B_f and r_g the root node of B_g.

1. If both r_f and r_g are terminal nodes with labels l_f and l_g, respectively (recall that terminal labels are either 0 or 1), then we compute the value $l_f\,\mathrm{op}\,l_g$ and let the resulting OBDD be B_0 if that value is 0 and B_1 otherwise.
2. In the remaining cases, at least one of the root nodes is a non-terminal. Suppose that both root nodes are x_i-nodes. Then we create an x_i-node n with a dashed line to `apply` $(\mathrm{op}, \mathrm{lo}(r_f), \mathrm{lo}(r_g))$ and a solid line to `apply` $(\mathrm{op}, \mathrm{hi}(r_f), \mathrm{hi}(r_g))$, i.e. we call `apply` recursively on the basis of (6.2).
3. If r_f is an x_i-node, but r_g is a terminal node or an x_j-node with $j > i$, then we know that there is no x_i-node in B_g because the two OBDDs have a compatible ordering of boolean variables. Thus, g is independent of x_i $(g \equiv g[0/x_i] \equiv g[1/x_i])$. Therefore, we create an x_i-node n with a dashed line to `apply` $(\mathrm{op}, \mathrm{lo}(r_f), r_g)$ and a solid line to `apply` $(\mathrm{op}, \mathrm{hi}(r_f), r_g)$.
4. The case in which r_g is a non-terminal, but r_f is a terminal or an x_j-node with $j > i$, is handled symmetrically to case 3.

The result of this procedure might not be reduced; therefore `apply` finishes by calling the function `reduce` on the OBDD it constructed. An example of `apply` (where op is $+$) can be seen in Figures 6.15–6.17. Figure 6.16 shows the recursive descent control structure of `apply` and Figure 6.17 shows the final result. In this example, the result of `apply` $(+, B_f, B_g)$ is B_f.

Figure 6.16 shows that numerous calls to `apply` occur several times with the same arguments. Efficiency could be gained if these were evaluated only

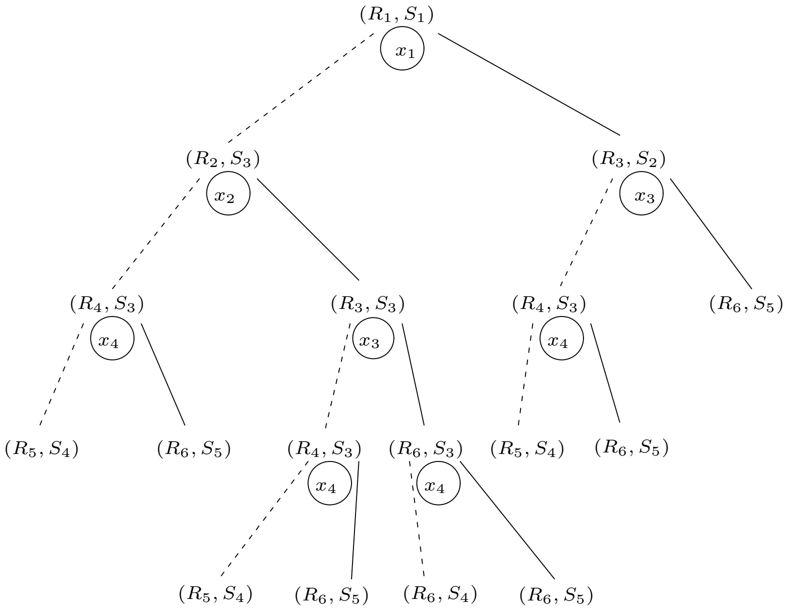

Figure 6.16. The recursive call structure of `apply` for the example in Figure 6.15 (without memoisation).

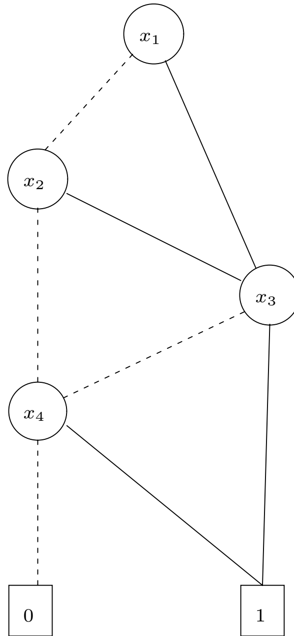

Figure 6.17. The result of `apply` $(+, B_f, B_g)$, where B_f and B_g are given in Figure 6.15.

the first time and the result remembered for future calls. This programming technique is known as memoisation. As well as being more efficient, it has the advantage that the resulting OBDD requires less reduction. (In this example, using memoisation eliminates the need for the final call to `reduce` altogether.) Without memoisation, `apply` is exponential in the size of its arguments, since each non-leaf call generates a further two calls. With memoisation, the number of calls to apply is bounded by $2 \cdot |B_f| \cdot |B_g|$, where $|B|$ is the size of the BDD. This is a worst-time complexity; the actual performance is often much better than this.

6.2.3 The algorithm `restrict`

Given an OBDD B_f representing a boolean formula f, we need an algorithm `restrict` such that the call `restrict`$(0, x, B_f)$ computes the reduced OBDD representing $f[0/x]$ using the same variable ordering as B_f. The algorithm for `restrict`$(0, x, B_f)$ works as follows. For each node n labelled with x, incoming edges are redirected to $\mathrm{lo}(n)$ and n is removed. Then we call `reduce` on the resulting OBDD. The call `restrict` $(1, x, B_f)$ proceeds similarly, only we now redirect incoming edges to $\mathrm{hi}(n)$.

6.2.4 The algorithm `exists`

A boolean function can be thought of as putting a constraint on the values of its argument variables. For example, the function $x + (\overline{y} \cdot z)$ evaluates to 1 only if x is 1; or y is 0 and z is 1 – this is a constraint on x, y, and z.

It is useful to be able to express the relaxation of the constraint on a subset of the variables concerned. To allow this, we write $\exists x. f$ for the boolean function f with the constraint on x relaxed. Formally, $\exists x. f$ is defined as $f[0/x] + f[1/x]$; that is, $\exists x. f$ is true if f could be made true by putting x to 0 or to 1. Given that $\exists x. f \stackrel{\text{def}}{=} f[0/x] + f[1/x]$ the `exists` algorithm can be implemented in terms of the algorithms `apply` and `restrict` as

$$\texttt{apply}\,(+, \texttt{restrict}\,(0, x, B_f), \texttt{restrict}\,(1, x, B_f))\ . \tag{6.3}$$

Consider, for example, the OBDD B_f for the function $f \stackrel{\text{def}}{=} x_1 \cdot y_1 + x_2 \cdot y_2 + x_3 \cdot y_3$, shown in Figure 6.19. Figure 6.20 shows `restrict`$(0, x_3, B_f)$ and `restrict`$(1, x_3, B_f)$ and the result of applying $+$ to them. (In this case the apply function happens to return its second argument.)

We can improve the efficiency of this algorithm. Consider what happens during the `apply` stage of (6.3). In that case, the `apply` algorithm works on two BDDs which are identical all the way down to the level of the x-nodes;

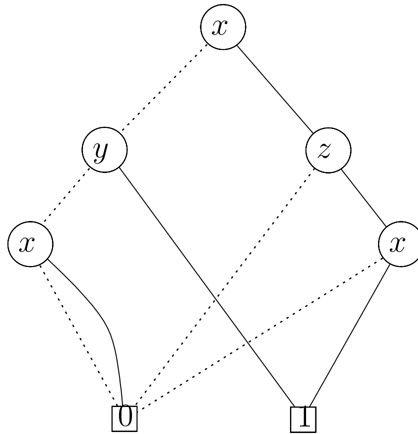

Figure 6.18. An example of a BDD which is not a read-1-BDD.

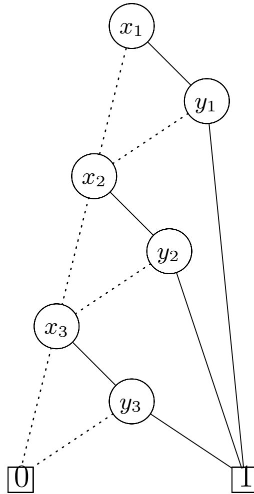

Figure 6.19. A BDD B_f to illustrate the `exists` algorithm.

therefore the returned BDD also has that structure down to the x-nodes. At the x-nodes, the two argument BDDs differ, so the `apply` algorithm will compute the apply of $+$ to these two subBDDs and return that as the subBDD of the result. This is illustrated in Figure 6.20. Therefore, we can compute the OBDD for $\exists x. f$ by taking the OBDD for f and replacing each node labelled with x by the result of calling `apply` on $+$ and its two branches.

This can easily be generalised to a sequence of `exists` operations. We write $\exists \hat{x}. f$ to mean $\exists x_1.\exists x_2.\dots\exists x_n. f$, where $\hat{x}$ denotes $(x_1, x_2, \dots, x_n)$.

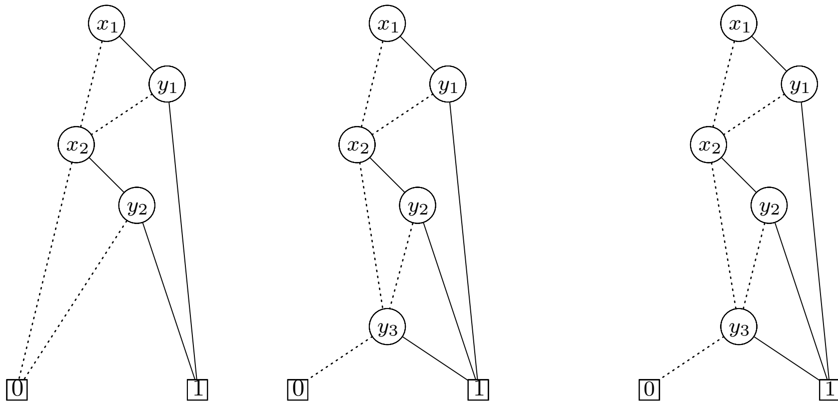

Figure 6.20. $\texttt{restrict}(0, x_3, B_f)$ and $\texttt{restrict}(1, x_3, B_f)$ and the result of applying $+$ to them.

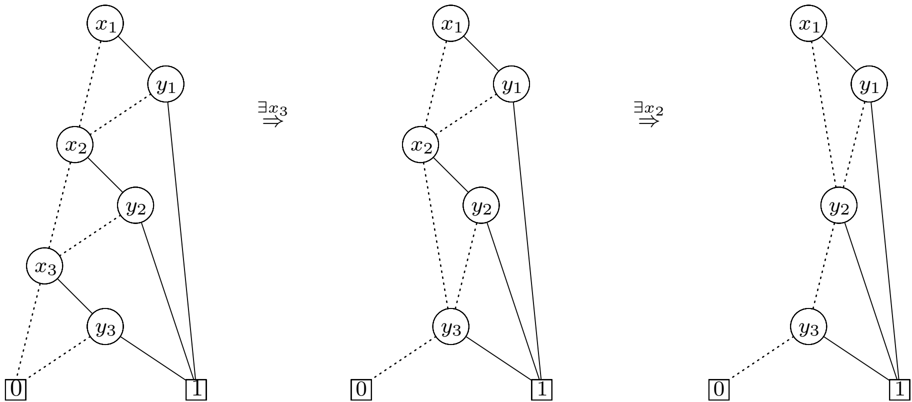

Figure 6.21. OBDDs for f, $\exists x_3.\, f$ and $\exists x_2.\exists x_3.\, f$.

The OBDD for this boolean function is obtained from the OBDD for f by replacing *every* node labelled with an x_i by the $+$ of its two branches.

Figure 6.21 shows the computation of $\exists x_3.\, f$ and $\exists x_2.\exists x_3.\, f$ (which is semantically equivalent to $x_1 \cdot y_1 + y_2 + y_3$) in this way.
The boolean quantifier $\forall$ is the dual of $\exists$:

$$\forall x.f \stackrel{\text{def}}{=} f[0/x] \cdot f[1/x]$$

asserting that f could be made false by putting x to 0 or to 1.

The translation of boolean formulas into OBDDs using the algorithms of this section is summarised in Figure 6.22.

Boolean formula f	Representing OBDD B_f
0	B_0 (Fig. 6.6)
1	B_1 (Fig. 6.6)
x	B_x (Fig. 6.6)
$\overline{f}$	swap the 0- and 1-nodes in B_f
$f + g$	`apply` $(+, B_f, B_g)$
$f \cdot g$	`apply` $(\cdot, B_f, B_g)$
$f \oplus g$	`apply` $(\oplus, B_f, B_g)$
$f[1/x]$	`restrict` $(1, x, B_f)$
$f[0/x]$	`restrict` $(0, x, B_f)$
$\exists x.f$	`apply` $(+, B_{f[0/x]}, B_{f[1/x]})$
$\forall x.f$	`apply` $(\cdot, B_{f[0/x]}, B_{f[1/x]})$

Figure 6.22. Translating boolean formulas f to OBDDs B_f, given a fixed, global ordering on boolean variables.

Algorithm	Input OBDD(s)	Output OBDD	Time-complexity
`reduce`	B	reduced B	$O(\|B\| \cdot \log \|B\|)$
`apply`	B_f, B_g (reduced)	$B_{f \text{ op } g}$ (reduced)	$O(\|B_f\| \cdot \|B_g\|)$
`restrict`	B_f (reduced)	$B_{f[0/x]}$ or $B_{f[1/x]}$ (reduced)	$O(\|B_f\| \cdot \log \|B_f\|)$
$\exists$	B_f (reduced)	$B_{\exists x_1.\exists x_2.\ldots.\exists x_n.f}$ (reduced)	NP-complete

Figure 6.23. Upper bounds in terms of the input OBDD(s) for the worst-case running times of our algorithms needed in our implementation of boolean formulas.

6.2.5 Assessment of OBDDs

Time complexities for computing OBDDs We can measure the complexity of the algorithms of the preceding section by giving upper bounds for the running time in terms of the sizes of the input OBDDs. The table in Figure 6.23 summarises these upper bounds (some of those upper bounds may require more sophisticated versions of the algorithms than the versions presented in this chapter). All the operations except nested boolean quantification are practically efficient in the size of the participating OBDDs. Thus, modelling very large systems with this approach will work if the OBDDs

which represent the systems don't grow too large too fast. If we can somehow control the size of OBDDs, e.g. by using good heuristics for the choice of variable ordering, then these operations are computationally feasible. It has already been shown that OBDDs modelling certain classes of systems and networks don't grow excessively.

The expensive computational operations are the nested boolean quantifications $\exists z_1.\ldots\exists z_n.f$ and $\forall z_1.\ldots\forall z_n.f$. By exercise 1 on page 406, the computation of the OBDD for $\exists z_1.\ldots\exists z_n.f$, given the OBDD for f, is an NP-complete problem[2]; thus, it is unlikely that there exists an algorithm with a feasible worst-time complexity. This is not to say that boolean functions modelling practical systems may not have efficient nested boolean quantifications. The performance of our algorithms can be improved by using further optimisation techniques, such as parallelisation.

Note that the operations `apply`, `restrict`, etc. are only efficient in the size of the input OBDDs. So if a function f does not have a compact representation as an OBDD, then computing with its OBDD will not be efficient. There are such nasty functions; indeed, one of them is *integer multiplication*. Let $b_{n-1}b_{n-2}\ldots b_0$ and $a_{n-1}a_{n-2}\ldots a_0$ be two n-bit integers, where b_{n-1} and a_{n-1} are the most significant bits and b_0 and a_0 are the least significant bits. The multiplication of these two integers results in a $2n$-bit integer. Thus, we may think of multiplication as $2n$ many boolean functions f_i in $2n$ variables (n bits for input b and n bits for input a), where f_i denotes the ith output bit of the multiplication. The following negative result, due to R. E. Bryant, shows that OBDDs cannot be used for implementing integer multiplication.

Theorem 6.11 Any OBDD representation of f_{n-1} has at least a number of vertices proportional to 1.09^n, i.e. its size is exponential in n.

Extensions and variations of OBDDs There are many variations and extensions to the OBDD data structure. Many of them can implement certain operations more efficiently than their OBDD counterparts, but it seems that none of them perform as well as OBDDs overall. In particular, one feature which many of the variations lack is the canonical form; therefore they lack an efficient algorithm for deciding when two objects denote the same boolean function.

One kind of variation allows non-terminal nodes to be labelled with binary operators as well as boolean variables. *Parity OBDDs* are like OBDDs in that there is an ordering on variables and every variable may occur at

[2] Another NP-complete problem is to decide the satisfiability of formulas of propositional logic.

most once on a path; but some non-terminal nodes may be labelled with $\oplus$, the exclusive-or operation. The meaning is that the function represented by that node is the exclusive-or of the boolean functions determined by its children. Parity OBDDs have similar algorithms for `apply`, `restrict`, etc. with the same performance, but they do not have a canonical form. Checking for equivalence cannot be done in constant time. There is, however, a cubic algorithm for determining equivalence; and there are also efficient probabilistic tests. Another variation of OBDDs allows complementation nodes, with the obvious meaning. Again, the main disadvantage is the lack of canonical form.

One can also allow non-terminal nodes to be unlabelled and to branch to more than two children. This can then be understood either as nondeterministic branching, or as probabilistic branching: throw a pair of dice to determine where to continue the path. Such methods may compute wrong results; one then aims at repeating the test to keep the (probabilistic) error as small as desired. This method of repeating probabilistic tests is called *probabilistic amplification*. Unfortunately, the satisfiability problem for probabilistic branching OBDDs is NP-complete. On a good note, probabilistic branching OBDDs can *verify* integer multiplication.

The development of extensions or variations of OBDDS which are customised to certain classes of boolean functions is an important area of ongoing research.

6.3 Symbolic model checking

The use of BDDs in model checking resulted in a significant breakthrough in verification in the early 1990s, because they have allowed systems with much larger state spaces to be verified. In this section, we describe in detail how the model-checking algorithm presented in Chapter 3 can be implemented using OBDDs as the basic data structure.

The pseudo-code presented in Figure 3.28 on page 227 takes as input a CTL formula ϕ and returns the set of states of the given model which satisfy ϕ. Inspection of the code shows that the algorithm consists of manipulating intermediate sets of states. We show in this section how the model and the intermediate sets of states can be stored as OBDDs; and how the operations required in that pseudo-code can be implemented in terms of the operations on OBDDs which we have seen in this chapter.

We start by showing how sets of states are represented with OBDDs, together with some of the operations required. Then, we extend that to the representation of the transition system; and finally, we show how the remainder of the required operations is implemented.

Model checking using OBDDs is called *symbolic model checking*. The term emphasises that individual states are not represented; rather, sets of states are represented symbolically, namely, those which satisfy the formula being checked.

6.3.1 Representing subsets of the set of states

Let S be a finite set (we forget for the moment that it is a set *of states*). The task is to represent the various subsets of S as OBDDs. Since OBDDs encode boolean functions, we need somehow to code the elements of S as boolean values. The way to do this in general is to assign to each element $s \in S$ a unique vector of boolean values $(v_1, v_2, \dots, v_n)$, each $v_i \in \{0, 1\}$. Then, we represent a subset T by the boolean function f_T which maps $(v_1, v_2, \dots, v_n)$ onto 1 if $s \in T$ and maps it onto 0 otherwise.

There are 2^n boolean vectors $(v_1, v_2, \dots, v_n)$ of length n. Therefore, n should be chosen such that $2^{n-1} < |S| \leq 2^n$, where $|S|$ is the number of elements in S. If $|S|$ is not an exact power of 2, there will be some vectors which do not correspond to any element of S; they are just ignored. The function $f_T : \{0, 1\}^n \to \{0, 1\}$ which tells us, for each s, represented by $(v_1, v_2, \dots, v_n)$, whether it is in the set T or not, is called the *characteristic function* of T.

In the case that S is the set of states of a transition system $\mathcal{M} = (S, \to, L)$ (see Definition 3.4), there is a natural way of choosing the representation of S as boolean vectors. The labelling function $L : S \to \mathcal{P}(\texttt{Atoms})$ (where $\mathcal{P}(\texttt{Atoms})$ is the set of subsets of `Atoms`) gives us the encoding. We assume a fixed ordering on the set `Atoms`, say $x_1, x_2, \dots, x_n$, and then represent $s \in S$ by the vector $(v_1, v_2, \dots, v_n)$, where, for each i, v_i equals 1 if $x_i \in L(s)$ and v_i is 0 otherwise. In order to guarantee that each s has a unique representation as a boolean vector, we require that, for all $s_1, s_2 \in S$, $L(s_1) = L(s_2)$ implies $s_1 = s_2$. If this is not the case, perhaps because $2^{|\texttt{Atoms}|} < |S|$, we can add extra atomic propositions in order to make enough distinctions (*Cf.* introduction of the `turn` variable for mutual exclusion in Section 3.3.4.)

From now on, we refer to a state $s \in S$ by its representing boolean vector $(v_1, v_2, \dots, v_n)$, where v_i is 1 if $x_i \in L(s)$ and 0 otherwise. As an OBDD, this state is represented by the OBDD of the boolean function $l_1 \cdot l_2 \cdot \dots \cdot l_n$, where l_i is x_i if $x_i \in L(s)$ and $\overline{x_i}$ otherwise. The set of states $\{s_1, s_2, \dots, s_m\}$ is represented by the OBDD of the boolean function

$$(l_{11} \cdot l_{12} \cdot \dots \cdot l_{1n}) + (l_{21} \cdot l_{22} \cdot \dots \cdot l_{2n}) + \dots + (l_{m1} \cdot l_{m2} \cdot \dots \cdot l_{mn})$$

where $l_{i1} \cdot l_{i2} \cdot \dots \cdot l_{in}$ represents state s_i.

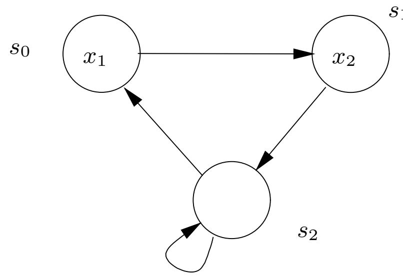

Figure 6.24. A simple CTL model (Example 6.12).

set of states	representation by boolean values	representation by boolean function
$\emptyset$		0
$\{s_0\}$	$(1,0)$	$x_1 \cdot \overline{x_2}$
$\{s_1\}$	$(0,1)$	$\overline{x_1} \cdot x_2$
$\{s_2\}$	$(0,0)$	$\overline{x_1} \cdot \overline{x_2}$
$\{s_0, s_1\}$	$(1,0)$, $(0,1)$	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$
$\{s_0, s_2\}$	$(1,0)$, $(0,0)$	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot \overline{x_2}$
$\{s_1, s_2\}$	$(0,1)$, $(0,0)$	$\overline{x_1} \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$
S	$(1,0)$, $(0,1)$, $(0,0)$	$x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2 + \overline{x_1} \cdot \overline{x_2}$

Figure 6.25. Representation of subsets of states of the model of Figure 6.24.

The key point which makes this representation interesting is that the OBDD representing a set of states may be quite small.

Example 6.12 Consider the CTL model in Figure 6.24, given by:

$$\begin{aligned} S &\stackrel{\text{def}}{=} \{s_0, s_1, s_2\} \\ \rightarrow &\stackrel{\text{def}}{=} \{(s_0, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_2)\} \\ L(s_0) &\stackrel{\text{def}}{=} \{x_1\} \\ L(s_1) &\stackrel{\text{def}}{=} \{x_2\} \\ L(s_2) &\stackrel{\text{def}}{=} \emptyset. \end{aligned}$$

Note that it has the property that, for all states s_1 and s_2, $L(s_1) = L(s_2)$ implies $s_1 = s_2$, i.e. a state is determined entirely by the atomic formulas true in it. Sets of states may be represented by boolean values and by boolean formulas with the ordering $[x_1, x_2]$, as shown in Figure 6.25.

Notice that the vector $(1,1)$ and the corresponding function $x_1 \cdot x_2$ are unused. Therefore, we are free to include it in the representation of a subset