terminates, the entire computation is correct. Let us formalize invariants and then study how to discover them.

**Definition 4.15** An invariant of the while-statement \( \text{while } (B) \{ C \} \) is a formula \( \eta \) such that \( \text{while } (B) \{ C \} (\eta) \) holds; i.e., for all states \( l \), if \( \eta \) and \( B \) are true in \( l \) and \( C \) is executed from state \( l \) and terminates, then \( \eta \) is again true in the resulting state.

Note that \( \eta \) does not have to be true continuously during the execution of \( C \); in general, it will not be. All we require is that, if it is true before \( C \) is executed, then it is true (if and) when \( C \) terminates.

For any given while-statement there are several invariants. For example, \( \top \) is an invariant for *any* while-statement; so is \( \bot \), since the premise of the implication "if \( \bot \land B \) is true, then \( \ldots \)" is false, so that implication is true.

The formula \( \neg B \) is also an invariant of \( \text{while } (B) \{ C \} \); but most of these invariants are useless to us, because we are looking for an invariant \( \eta \) for which the sequents \( \Gamma \vdash \phi \rightarrow \eta \) and \( \Gamma \vdash \eta \land \neg B \rightarrow \psi \), are valid, where \( \phi \) and \( \psi \) are the preconditions and postconditions of the while-statement. Usually, this will single out just one of all the possible invariants — up to logical equivalence.

A useful invariant expresses a relationship between the variables manipulated by the body of the while-statement which is preserved by the execution of the body, even though the values of the variables themselves may change. The invariant can often be found by constructing a trace of the while-statement in action.

**Example 4.16** Consider the program `Fac1` from page 262, annotated with location labels for our discussion:

\[
y = 1; \\
z = 0; \\
11: \text{while } (z \neq x) \{ \\
\quad z = z + 1; \\
\quad y = y \times z; \\
12: \}
\]

Suppose program execution begins in a store in which \( x \) equals 6. When the program flow first encounters the while-statement at location 11, \( z \) equals 0 and \( y \) equals 1, so the condition \( z \neq x \) is true and the body is executed. Thereafter at location 12, \( z \) equals 1 and \( y \) equals 1 and the boolean guard is still true, so the body is executed again. Continuing in this way, we obtain

<table>
<thead>
<tr>
<th>after iteration</th>
<th>( z ) at 11</th>
<th>( y ) at 11</th>
<th>( B ) at 11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>24</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>120</td>
<td>true</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>720</td>
<td>false</td>
</tr>
</tbody>
</table>

The program execution stops when the boolean guard becomes false.

The invariant of this example is easy to see: it is \( y = x! \). Every time we complete an execution of the body of the while-statement, this fact is true, even though the values of \( y \) and \( x \) 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 \( y = 1 \land z = 0 \) based on the initial assignments and their precondition \( B \equiv 1 \);
- 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

\[
\Gamma \vdash (y = 1 \land z = 0) \rightarrow (y = x!) \quad \text{and} \quad \Gamma \vdash (y = x! \land z = x) \rightarrow (y = x!).
\]

(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, is 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 \( \psi \) upwards through a while-statement to meet the precondition \( \phi \). The steps are:

1. Guess a formula \( \eta \) which you hope is a suitable invariant.
2. Try to prove that \( \Gamma \vdash \eta \land \neg B \rightarrow \psi \) and \( \Gamma \vdash \phi \rightarrow \eta \) 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 \( \eta \) 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 \( \eta' \).
4 Program verification

4. Try to prove that \( T \land B \rightarrow \eta \) is valid; this proves that \( \eta \) is indeed an invariant. If you succeed, go to 5. Otherwise, go back to 1.

5. Now write \( \eta \) above the while-statement and write \( \phi \) above that \( \eta \), annotating that \( \eta \) with an instance of Implied based on the successful proof of the validity of \( T \land B \rightarrow \eta \) in 2. Mission accomplished!

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

\[
\begin{align*}
&y = 1; \\
&z = 0; \\
&(y = z!) \Rightarrow \text{Invar. Hyp.} \\
&\text{while } (z \neq x) \{ \\
&\quad (y = z! \land z \neq x) \Rightarrow \text{Invar. Hyp.} \land \text{guard} \\
&\quad (y \cdot (z + 1) = (z + 1)!) \Rightarrow \text{Implied} \\
&\quad z = z + 1; \\
&\quad (y \cdot z = z!) \Rightarrow \text{Assignment} \\
&\quad y = y \ast z; \\
&\quad (y = z!) \Rightarrow \text{Assignment} \\
&\}\ &\Rightarrow \text{Implied}
\end{align*}
\]

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 \( \eta \) 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 \( z = x \) imply \( y = z! \).
- The ability to prove that \( \eta \) 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 \( T \); our completed proof is:

\[
\begin{align*}
&\{ T \} \\
&\{ 1 = 0! \} \quad \text{Implied} \\
&y = 1; \\
&\{ y = 0! \} \quad \text{Assignment} \\
&z = 0; \\
&\{ y = z! \} \quad \text{Assignment} \\
&\text{while } (z \neq x) \{ \\
&\quad (y = z! \land z \neq x) \quad \text{Invar. Hyp. \land guard} \\
&\quad (y \cdot (z + 1) = (z + 1)!) \quad \text{Implied} \\
&\quad z = z + 1; \\
&\quad (y \cdot z = z!) \quad \text{Assignment} \\
&\quad y = y \ast z; \\
&\quad (y = z!) \quad \text{Assignment} \\
&\}\ &\Rightarrow \text{Partial-while} \\
&\Rightarrow \text{Implied}
\end{align*}
\]

4.3 Proof calculus for partial correctness

defined to be 1, so it is implied by \( T \); our completed proof is:

\[
\begin{align*}
&\{ T \} \\
&\{ 1 = 0! \} \quad \text{Implied} \\
&y = 1; \\
&\{ y = 0! \} \quad \text{Assignment} \\
&z = 0; \\
&\{ y = z! \} \quad \text{Assignment} \\
&\text{while } (z \neq x) \{ \\
&\quad (y = z! \land z \neq x) \quad \text{Invar. Hyp. \land guard} \\
&\quad (y \cdot (z + 1) = (z + 1)!) \quad \text{Implied} \\
&\quad z = z + 1; \\
&\quad (y \cdot z = z!) \quad \text{Assignment} \\
&\quad y = y \ast z; \\
&\quad (y = z!) \quad \text{Assignment} \\
&\}\ &\Rightarrow \text{Partial-while} \\
&\Rightarrow \text{Implied}
\end{align*}
\]

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. For example, we may declare an array

int a[n];

whose name is \( a \) and whose fields are accessed by \( a[0], a[1], \ldots, a[n-1] \), 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 \( a[0], \ldots, a[n-1] \) be the integer values of an array \( a \). A section of \( a \) is a continuous piece \( a[i], \ldots, a[j] \), where \( 0 \leq i \leq j < n \). We

---

5 We only need from arrays in the program \texttt{MinSum} 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 $S_{ij}$ 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 $S_{ij}$ 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 MinSum, 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 MinSum;
- use our proof calculus for partial correctness to show that MinSum 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:

```plaintext
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, written as Hoare triples.

\( \{ T \} \text{Min.Sum} (\forall i, j (0 \leq i \leq j < n \to s \leq S_{i,j})) \).

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.

\( \{ T \} \text{Min.Sum} (\exists i, j (0 \leq i \leq j < n \land s = S_{i,j})) \),

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 \( S2 \) gives 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

\[ \text{Inv1}(s, k) \triangleq \forall i, j (0 \leq i \leq j < k \to s \leq S_{i,j}) \quad (4.12) \]

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 \( \text{Inv1}(s, k) \) which highlights that the formula has only the program variables \( s \) and \( k \) as free variables and is similar to the use of \texttt{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 computation is

\[ \text{Inv2}(t, k) \triangleq \forall i (0 \leq i < k \to t \leq S_{i,k-1}) \quad (4.13) \]

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

\[ \text{Inv1}(s, k) \land \text{Inv2}(t, k). \quad (4.14) \]

The completed proof tableau of \( S1 \) for \texttt{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 \( t \) 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.
4 Program verification

- 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 T.

As so often the case, in constructing the tableau, we find that two formulae 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 T implies Inv1(a[0], 1) ∧ Inv2(a[0], 1): k being 1 forces i and j to be zero in order that the assumptions in Inv1(a[j], k) and Inv2(a[j], 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 Inv1(s, k) ∧ Inv2(t, k) ∧ k ≠ n implies

1. Inv1(min(s, min(t + a[k], a[l])), k + 1) as well as
2. Inv2(min(t + a[k], a[l]), k + 1).

PROOF:

1. Take any i with 0 ≤ i < k + 1; we will prove that min(t + a[k], a[l]) ≤ 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[l]) ≤ S_{i, k-1} + a[k]; but we know t ≤ 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 ≤ i ≤ j < k + 1; we prove that min(s, t + a[k], a[l]) ≤ S_{i, j}. If i < j, then the result is immediate. Otherwise, i ≤ j = k and the result follows from part 1 of the lemma.

4.4 Proof calculus for total correctness

In the preceding section, we developed a calculus for proving partial correctness of triples (ϕ) P ψ. In that setting, proofs come with a disclaimer: only if the program P terminates an execution does a proof of ⊢_par(ϕ) P ψ 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 Fact 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:

\[
\frac{\eta \land B \land 0 \leq E = E_0 \land C (\eta \land 0 \leq E < E_0) \land (\eta \land 0 \leq E) \text{ while } B \{C\} (\eta \land \neg B)}{\text{Total-while.}}
\]

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 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

\[
(\eta \land B \land 0 \leq E = E_0 \land C (\eta \land 0 \leq E < E_0).
\]

When we push η ∧ 0 ≤ E < E_0 upwards through the body, we have to prove that what emerges from the top is implied by η ∧ B ∧ 0 ≤ E = E_0; and the weakest precondition for the entire while-statement, which gets written above that while-statement, is η ∧ 0 ≤ E.
4 Program verification

Let us illustrate this rule by proving that $\Gamma_{\text{nat}}(y \geq 0) \text{Fac1} (y = z !)$ is valid, where Fac1 is given in Example 4.2, as follows:

$$y = 1;$$
$$z = 0;$$
$$\text{while } (x != 0) \{$$
$$z = z + 1;$$
$$y = y * z;$$
$$\}$$

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

$$y = 1;$$
$$\text{while } (x != 0) \{$$
$$z = z + 1;$$
$$y = y * z;$$
$$\}$$

and so $\Gamma_{\text{nat}}(y \geq 0) \text{Fac1} (y = z !)$ is valid. Two comments are in order:

- Notice that the precondition $x \geq 0$ is crucial in ensuring the fact that $0 \leq x - z$ holds right before the while-statement gets executed; it implies the precondition $1 = 0 \land 0 \leq x - 0$ 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.

4.4 Proof calculus for total correctness

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 $E$:

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

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 $5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$. For another example, if the value of $x$ is initially 172, the evolution of $c$ is

```
172 86 43 186 98 49 148 74 37 112 56 28 14 7 22 11 34 17 62 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):
4 Program verification

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 makes 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:

\[
324997234256597353456727966237642051630475634563647563\ \\
96598734068368765607406566707684074563734066345676406075 \ \\
62984573576830653788640565305245653457892262635421356761\ \\
96197651298641229654248954656956467
\]

where \( \\backslash \) denotes concatenation of digits. Although this is a very large number indeed, our program Collatz requires only 6940 iterations to terminate. Unfortunately, nobody knows a suitable invariant for this program that could prove the validity of \( I_{\text{init}}(0 < x) \) in Collatz \( \{T\} \). Observe how the use of \( T \) 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 \( 3x + 1 \) in Collatz with a different such linear expression in \( c \), the program may not terminate despite meeting the preconditions \( 0 < x \); see exercise 6 on page 303.

4.5 Programming by contract

For a valid sequent \( I_{\text{init}}(\phi) P (\psi) \), 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 \( \phi \). In that case, the supplier promises the consumer that the final state of that run satisfies \( \psi \). For a valid \( I_{\text{init}}(\phi) P (\psi) \), 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 \textbf{Fact} may be the \( \ldots \) in the method body

\[
\text{int factorial (x : int) \{ \ldots return y; \}}
\]

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 at run-time in languages such as Eiffel. A possible format for such contracts for the method factorial is given in Figure 4.4.

Figure 4.4. A contract for the method factorial.

The keyword \textbf{assumes} states all preconditions, the keyword \textbf{guarantees} lists all postconditions. The keyword \textbf{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 \( 1/\left(\binom{n}{k}\right) \) is your chance of getting all six lottery numbers right out of 49 numbers total. Your boss also tells you that

\[
\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}
\]

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

\[
\text{int choose(n : int, k : int) \{ return factorial(n) / (factorial(k) * factorial(n - k)); \}
\]

This method body consists of a return-statement only which makes three calls to \textbf{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 (\textbf{assumes}) and postconditions (\textbf{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 \textbf{factorial} whose precondition is that its input value is 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 \textbf{choose}? Since the method body declared no local variables, we use \textbf{result} to denote the return value of this
method. The postcondition then states that result equals \( k \) — assuming that you boss’ equation (4.16) is correct for your preconditions \( 0 \leq k \leq n \), and \( k \leq n \). The contract for choose is therefore

```java
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’ \{assume\} method \{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

\[
\{ \text{Inv1}(\min(s, \min(t + a[i], a[i])), k + 1) \wedge \text{Inv2}(\min(t + a[k], a[k]), k + 1) \} \Rightarrow \text{Implied (Lemma 4.20)}
\]

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

\[
\{ \text{Inv1}(\min(s, t), k + 1) \wedge \text{Inv2}(t, k + 1) \} \Rightarrow \text{Assignment}
\]

\[
s = \min(s, t);
\]

\[
\{ \text{Inv1}(s, k + 1) \wedge \text{Inv2}(t, k + 1) \} \Rightarrow \text{Assignment}
\]

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 \( \text{Inv1}(s, k + 1) \wedge \text{Inv2}(t, k + 1) \), 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 producing correct and reliable programs.

**Exercises 4.2**

1. In what circumstances would \{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 \( i = \text{max} \) 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 repeats (C) until (B) terminates.

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

(a) Define a repeatcount function using your code language.
(b) Can we define every repeat expression in our code language extended with for-statements? (You might need the empty command skip which does nothing.)

Exercises 4.8

1. For any store T in Example 4.1, determine which of the relations below hold; justify your answers:
   (a) $T (x < 0, x > 0, y = 0)$
   (b) $T (x < 0, x > 0, y = 0)$
   (c) $T (x < 0, y = 0, z = 0)$
2. For any $x, y$ and $P$, explain why $P (x, y) \iff P (x, y)$ holds whenever the relation $P (x, y)$ holds.
3. Let the relation $P \rightarrow T$ hold if $P$'s execution in store $T$ terminates, resulting in store $T$. Use this formal judgment $P \rightarrow T$ along with the relation $\vdash$ to define $\vdash T$ and $\vdash T$ 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) $x > 0 \vdash x + 1 > 0$ (b) $x > 1 \vdash x + 1 > 0$
   (c) $x > 1 \vdash x + y > 0$
6. Write down a program $P$ such that:
   (a) $\vdash P (x = 2)$
   (b) $\vdash P (x = y + 4)$
7. For all instances of $\vdash T$ on page 271, specify their corresponding AR sequences.
8. There is a safe way of relaxing the forms of the proof rule for assignments, as long as no variable occurring in $F$ gets updated in between the assertion $F \rightarrow P$ and the assignment $x = E$ we may conclude $P$ 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:
   \[
   \begin{align*}
   \vdash & \phi = \psi \\
   \vdash & \phi \Rightarrow \psi \\
   \end{align*}
   \]
   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.