we would like to express, as a Hoare triple, that the value of z upon termination is $x_0(x_0+1)/2$ where x_0 is the initial value of x. Thus, we write $(\!|x = x_0 \wedge x \geq 0|\!)\ \texttt{Sum}\ (\!|z = x_0(x_0+1)/2|\!)$.

Variables like x_0 in these examples are called *logical variables*, because they occur only in the logical formulas that constitute the precondition and postcondition; they do not occur in the code to be verified. The state of the system gives a value to each program variable, but not for the logical variables. Logical variables take a similar role to the dummy variables of the rules for ∀i and ∃e in Chapter 2.

Definition 4.10 For a Hoare triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$, its set of logical variables are those variables that are free in ϕ or ψ; and don't occur in P.

4.3 Proof calculus for partial correctness

The proof calculus which we now present goes back to R. Floyd and C. A. R. Hoare. In the next subsection, we specify proof rules for each of the grammar clauses for commands. We could go on to use these proof rules directly, but it turns out to be more convenient to present them in a different form, suitable for the construction of proofs known as *proof tableaux*. This is what we do in the subsection following the next one.

4.3.1 Proof rules

The proof rules for our calculus are given in Figure 4.1. They should be interpreted as rules that allow us to pass from simple assertions of the form $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ to more complex ones. The rule for assignment is an axiom as it has no premises. This allows us to construct some triples out of nothing, to get the proof going. Complete proofs are trees, see page 274 for an example.

Composition. Given specifications for the program fragments C_1 and C_2, say

$$(\!|\phi|\!)\ C_1\ (\!|\eta|\!) \quad \text{and} \quad (\!|\eta|\!)\ C_2\ (\!|\psi|\!),$$

where the postcondition of C_1 is also the precondition of C_2, the proof rule for sequential composition shown in Figure 4.1 allows us to derive a specification for $C_1; C_2$, namely

$$(\!|\phi|\!)\ \ C_1; C_2\ \ (\!|\psi|\!).$$

$$\frac{\{\!|\phi|\!\}\ C_1\ \{\!|\eta|\!\} \qquad \{\!|\eta|\!\}\ C_2\ \{\!|\psi|\!\}}{\{\!|\phi|\!\}\ \ C_1; C_2\ \ \{\!|\psi|\!\}}\ \text{Composition}$$

$$\frac{}{\{\!|\psi[E/x]|\!\}\ x = E\ \{\!|\psi|\!\}}\ \text{Assignment}$$

$$\frac{\{\!|\phi \wedge B|\!\}\ C_1\ \{\!|\psi|\!\} \qquad \{\!|\phi \wedge \neg B|\!\}\ C_2\ \{\!|\psi|\!\}}{\{\!|\phi|\!\}\ \texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}\ \{\!|\psi|\!\}}\ \text{If-statement}$$

$$\frac{\{\!|\psi \wedge B|\!\}\ C\ \{\!|\psi|\!\}}{\{\!|\psi|\!\}\ \texttt{while}\ B\ \{C\}\ \{\!|\psi \wedge \neg B|\!\}}\ \text{Partial-while}$$

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

Figure 4.1. Proof rules for partial correctness of Hoare triples.

Thus, if we know that C_1 takes ϕ-states to η-states and C_2 takes η-states to ψ-states, then running C_1 and C_2 in that sequence will take ϕ-states to ψ-states.

Using the proof rules of Figure 4.1 in program verification, we have to read them bottom-up: e.g. in order to prove $\{\!|\phi|\!\}\ C_1; C_2\ \{\!|\psi|\!\}$, we need to find an appropriate η and prove $\{\!|\phi|\!\}\ C_1\ \{\!|\eta|\!\}$ and $\{\!|\eta|\!\}\ C_2\ \{\!|\psi|\!\}$. If $C_1; C_2$ runs on input satisfying ϕ and we need to show that the store satisfies ψ after its execution, then we hope to show this by splitting the problem into two. After the execution of C_1, we have a store satisfying η which, considered as input for C_2, should result in an output satisfying ψ. We call η a *midcondition*.

Assignment. The rule for assignment has no premises and is therefore an axiom of our logic. It tells us that, if we wish to show that ψ holds in the state after the assignment `x = E`, we must show that $\psi[E/x]$ holds before the assignment; $\psi[E/x]$ denotes the formula obtained by taking ψ and replacing all free occurrences of x with E as defined on page 105. We read the stroke as 'in place of;' thus, $\psi[E/x]$ is ψ with E in place of x. Several explanations may be required to understand this rule.

- At first sight, it looks as if the rule has been stated in reverse; one might expect that, if ψ holds in a state in which we perform the assignment `x = E`, then surely

$\psi[E/x]$ holds in the resulting state, i.e. we just replace x by E. This is wrong. It is true that the assignment `x = E` replaces the value of x in the starting state by E, but that does not mean that we replace occurrences of x in a *condition on* the starting state by E.

For example, let ψ be $x = 6$ and E be 5. Then $\big(\!\big|\psi\big|\!\big)$ `x = 5` $\big(\!\big|\psi[x/E]\big|\!\big)$ does *not* hold: given a state in which x equals 6, the execution of `x = 5` results in a state in which x equals 5. But $\psi[x/E]$ is the formula $5 = 6$ which holds in no state.

The right way to understand the Assignment rule is to think about what you would have to prove about the initial state in order to prove that ψ holds in the resulting state. Since ψ will – in general – be saying something about the value of x, whatever it says about that value must have been true of E, since in the resulting state the value of x is E. Thus, ψ with E in place of x – which says whatever ψ says about x but applied to E – must be true in the initial state.

- The axiom $\big(\!\big|\psi[E/x]\big|\!\big)\, x = E\, \big(\!\big|\psi\big|\!\big)$ is best applied backwards than forwards in the verification process. That is to say, if we know ψ and we wish to find ϕ such that $\big(\!\big|\phi\big|\!\big)\, x = E\, \big(\!\big|\psi\big|\!\big)$, it is easy: we simply set ϕ to be $\psi[E/x]$; but, if we know ϕ and we want to find ψ such that $\big(\!\big|\phi\big|\!\big)\, x = E\, \big(\!\big|\psi\big|\!\big)$, there is no easy way of getting a suitable ψ. This backwards characteristic of the assignment and the composition rule will be important when we look at how to construct proofs; we will work from the end of a program to its beginning.
- If we apply this axiom in this backwards fashion, then it is completely mechanical to apply. It just involves doing a substitution. That means we could get a computer to do it for us. Unfortunately, that is not true for all the rules; application of the rule for while-statements, for example, requires ingenuity. Therefore a computer can at best assist us in performing a proof by carrying out the mechanical steps, such as application of the assignment axiom, while leaving the steps that involve ingenuity to the programmer.
- Observe that, in computing $\psi[E/x]$ from ψ, we replace all the free occurrences of x in ψ. Note that there cannot be problems caused by *bound* occurrences, as seen in Example 2.9 on page 106, *provided that preconditions and postconditions quantify over logical variables only.* For obvious reasons, this is recommended practice.

Examples 4.11

1. Suppose P is the program `x = 2`. The following are instances of axiom Assignment:
   a $\big(\!\big|2 = 2\big|\!\big)\, P\, \big(\!\big|x = 2\big|\!\big)$
   b $\big(\!\big|2 = 4\big|\!\big)\, P\, \big(\!\big|x = 4\big|\!\big)$
   c $\big(\!\big|2 = y\big|\!\big)\, P\, \big(\!\big|x = y\big|\!\big)$
   d $\big(\!\big|2 > 0\big|\!\big)\, P\, \big(\!\big|x > 0\big|\!\big)$.

These are all correct statements. Reading them backwards, we see that they say:

a If you want to prove $x = 2$ after the assignment `x = 2`, then we must be able to prove that $2 = 2$ before it. Of course, 2 is equal to 2, so proving it shouldn't present a problem.
b If you wanted to prove that $x = 4$ after the assignment, the only way in which it would work is if $2 = 4$; however, unfortunately it is not. More generally, $(\!|\bot|\!)\ x = E\ (\!|\psi|\!)$ holds for any E and ψ – why?
c If you want to prove $x = y$ after the assignment, you will need to prove that $2 = y$ before it.
d To prove $x > 0$, we'd better have $2 > 0$ prior to the execution of P.

2. Suppose P is $\mathtt{x} = \mathtt{x} + \mathtt{1}$. By choosing various postconditions, we obtain the following instances of the assignment axiom:

a $(\!|x + 1 = 2|\!)\, P\, (\!|x = 2|\!)$
b $(\!|x + 1 = y|\!)\, P\, (\!|x = y|\!)$
c $(\!|x + 1 + 5 = y|\!)\, P\, (\!|x + 5 = y|\!)$
d $(\!|x + 1 > 0 \wedge y > 0|\!)\, P\, (\!|x > 0 \wedge y > 0|\!)$.

Note that the precondition obtained by performing the substitution can often be simplified. The proof rule for implications below will allow such simplifications which are needed to make preconditions appreciable by human consumers.

If-statements. The proof rule for if-statements allows us to prove a triple of the form

$$(\!|\phi|\!)\ \mathtt{if}\ B\ \{C_1\}\ \mathtt{else}\ \{C_2\}\ (\!|\psi|\!)$$

by decomposing it into two triples, subgoals corresponding to the cases of B evaluating to true and to false. Typically, the precondition ϕ will not tell us anything about the value of the boolean expression B, so we have to consider both cases. If B is true in the state we start in, then C_1 is executed and hence C_1 will have to translate ϕ states to ψ states; alternatively, if B is false, then C_2 will be executed and will have to do that job. Thus, we have to prove that $(\!|\phi \wedge B|\!)\ C_1\ (\!|\psi|\!)$ and $(\!|\phi \wedge \neg B|\!)\ C_2\ (\!|\psi|\!)$. Note that the preconditions are augmented by the knowledge that B is true and false, respectively. This additional information is often crucial for completing the respective subproofs.

While-statements. The rule for while-statements given in Figure 4.1 is arguably the most complicated one. The reason is that the while-statement is the most complicated construct in our language. It is the only command that 'loops,' i.e. executes the same piece of code several times. Also, unlike as the for-statement in languages like Java we cannot generally predict how

many times while-statements will 'loop' around, or even whether they will terminate at all.

The key ingredient in the proof rule for Partial-while is the 'invariant' ψ. In general, the body C of the command `while (B) {C}` changes the values of the variables it manipulates; but the invariant expresses a relationship between those values which is preserved by any execution of C. In the proof rule, ψ expresses this invariant; the rule's premise, $(\!|\psi \wedge B|\!)\ C\ (\!|\psi|\!)$, states that, if ψ and B are true before we execute C, and C terminates, then ψ will be true after it. The conclusion of Partial-while states that, no matter how many times the body C is executed, if ψ is true initially and the while-statement terminates, then ψ will be true at the end. Moreover, since the while-statement has terminated, B will be false.

Implied. One final rule is required in our calculus: the rule Implied of Figure 4.1. It tells us that, if we have proved $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ and we have a formula ϕ' which implies ϕ and another one ψ' which is implied by ψ, then we should also be allowed to prove that $(\!|\phi'|\!)\ P\ (\!|\psi'|\!)$. A sequent $\vdash_{\mathrm{AR}} \phi \to \phi'$ is valid iff there is a proof of ϕ' in the natural deduction calculus for predicate logic, where ϕ and standard laws of arithmetic – e.g. $\forall x\,(x = x + 0)$ – are premises. Note that the rule Implied allows the precondition to be strengthened (thus, we *assume* more than we need to), while the postcondition is weakened (i.e. we *conclude* less than we are entitled to). If we tried to do it the other way around, weakening the precondition or strengthening the postcondition, then we would conclude things which are incorrect – see exercise 9(a) on page 300.

The rule Implied acts as a link between program logic and a suitable extension of predicate logic. It allows us to import proofs in predicate logic enlarged with the basic facts of arithmetic, which are required for reasoning about integer expressions, into the proofs in program logic.

4.3.2 Proof tableaux

The proof rules presented in Figure 4.1 are not in a form which is easy to use in examples. To illustrate this point, we present an example of a proof in Figure 4.2; it is a proof of the triple $(\!|\top|\!)\ \texttt{Fac1}\ (\!|y = x!|\!)$ where `Fac1` is the factorial program given in Example 4.2. This proof abbreviates rule names; and drops the bars and names for Assignment as well as sequents for $\vdash_{\mathrm{AR}}$ in all applications of the Implied rule. We have not yet presented enough information for the reader to complete such a proof on her own, but she can at least use the proof rules in Figure 4.1 to check whether all rule instances of that proof are permissible, i.e. match the required pattern.

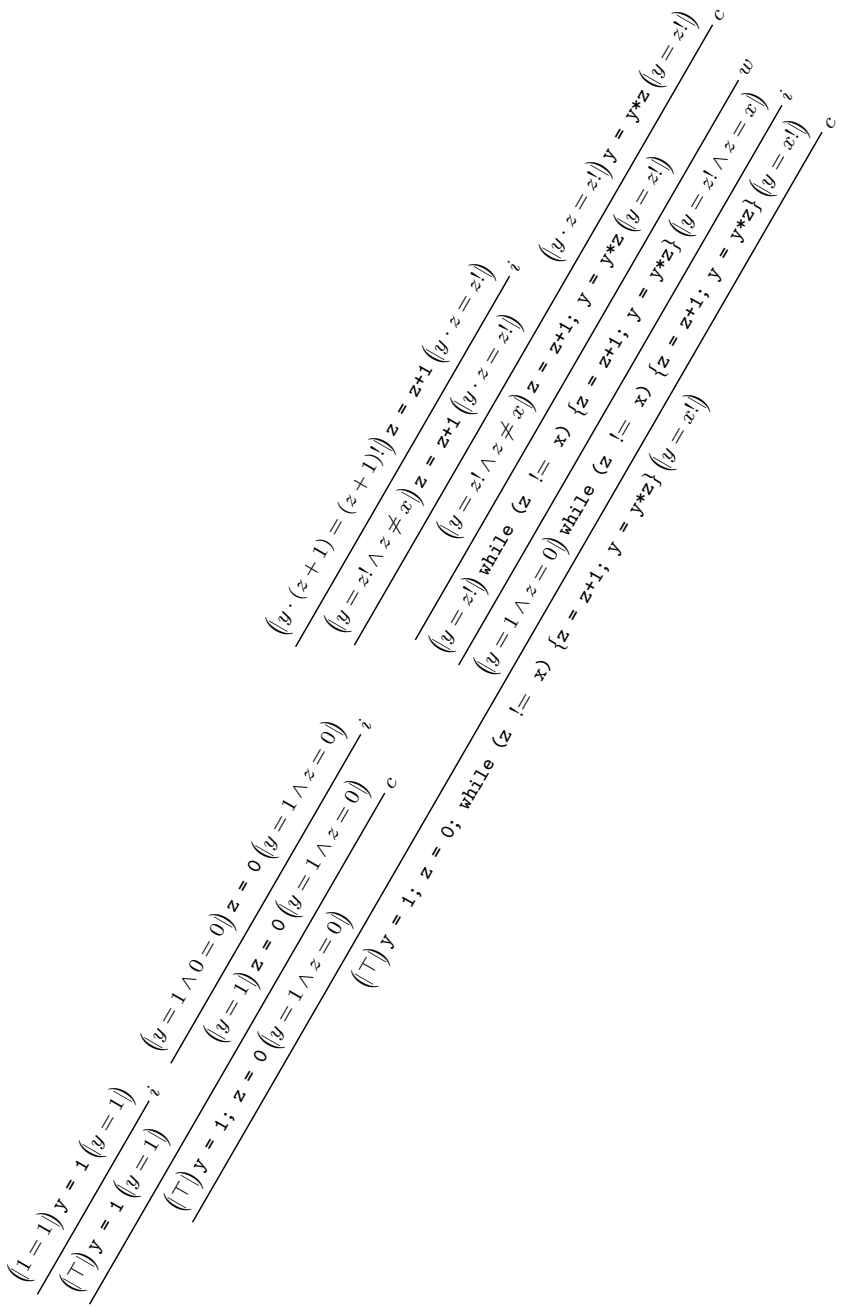

Figure 4.2. A partial-correctness proof for `Fac1` in tree form.

It should be clear that proofs in this form are unwieldy to work with. They will tend to be very wide and a lot of information is copied from one line to the next. Proving properties of programs which are longer than `Fac1` would be very difficult in this style. In Chapters 1, 2 and 5 we abandon representation of proofs as trees for similar reasons. The rule for sequential composition suggests a more convenient way of presenting proofs in program logic, called *proof tableaux*. We can think of any program of our core programming language as a sequence

$$\begin{array}{l} C_1; \\ C_2; \\ \cdot \\ \cdot \\ \cdot \\ C_n \end{array}$$

where none of the commands C_i is a composition of smaller programs, i.e. all of the C_i above are either assignments, if-statements or while-statements. Of course, we allow the if-statements and while-statements to have embedded compositions.

Let P stand for the program $C_1; C_2; \ldots; C_{n-1}; C_n$. Suppose that we want to show the validity of $\vdash_{\mathsf{par}} (\!|\phi_0|\!)\ P\ (\!|\phi_n|\!)$ for a precondition ϕ_0 and a postcondition ϕ_n. Then, we may split this problem into smaller ones by trying to find formulas ϕ_j $(0 < j < n)$ and prove the validity of $\vdash_{\mathsf{par}} (\!|\phi_i|\!)\ C_{i+1}\ (\!|\phi_{i+1}|\!)$ for $i = 0, 1, \ldots, n-1$. This suggests that we should design a proof calculus which presents a proof of $\vdash_{\mathsf{par}} (\!|\phi_0|\!)\ P\ (\!|\psi_n|\!)$ by interleaving formulas with code as in

$$\begin{array}{ll} \quad (\!|\phi_0|\!) & \\ C_1\,; & \\ \quad (\!|\phi_1|\!) & \texttt{justification} \\ C_2\,; & \\ \cdot & \\ \cdot & \\ \cdot & \\ \quad (\!|\phi_{n-1}|\!) & \texttt{justification} \\ C_n\,; & \\ \quad (\!|\phi_n|\!) & \texttt{justification} \end{array}$$

Against each formula, we write a justification, whose nature will be clarified shortly. Proof tableaux thus consist of the program code interleaved with formulas, which we call *midconditions*, that should hold at the point they are written.

Each of the transitions

$$\begin{array}{l} \{\!|\phi_i|\!\} \\ C_{i+1} \\ \{\!|\phi_{i+1}|\!\} \end{array}$$

will appeal to one of the rules of Figure 4.1, depending on whether C_{i+1} is an assignment, an if-statement or a while-statement. Note that this notation for proofs makes the proof rule for composition in Figure 4.1 implicit.

How should the intermediate formulas ϕ_i be found? In principle, it seems as though one could start from ϕ_0 and, using C_1, obtain ϕ_1 and continue working downwards. However, because the assignment rule works backwards, it turns out that it is more convenient to start with ϕ_n and work upwards, using C_n to obtain ϕ_{n-1} etc.

Definition 4.12 The process of obtaining ϕ_i from C_{i+1} and ϕ_{i+1} is called computing the weakest precondition of C_{i+1}, given the postcondition ϕ_{i+1}. That is to say, we are looking for the logically weakest formula whose truth at the beginning of the execution of C_{i+1} is enough to guarantee ϕ_{i+1}[4].

The construction of a proof tableau for $\{\!|\phi|\!\}\, C_1; \ldots; C_n\, \{\!|\psi|\!\}$ typically consists of starting with the postcondition ψ and pushing it upwards through C_n, then $C_{n-1}, \ldots$, until a formula ϕ' emerges at the top. Ideally, the formula ϕ' represents the weakest precondition which guarantees that the ψ will hold if the composed program $C_1; C_2; \ldots; C_{n-1}; C_n$ is executed and terminates. The weakest precondition ϕ' is then checked to see whether it follows from the given precondition ϕ. Thus, we appeal to the Implied rule of Figure 4.1.

Before a discussion of how to find invariants for while-statement, we now look at the assignment and the if-statement to see how the weakest precondition is calculated for each one.

Assignment. The assignment axiom is easily adapted to work for proof tableaux. We write it thus:

[4] ϕ is weaker than ψ means that ϕ is implied by ψ in predicate logic enlarged with the basic facts about arithmetic: the sequent $\vdash_{\mathrm{AR}} \psi \to \phi$ is valid. We want the weakest formula, because we want to impose as few constraints as possible on the preceding code. In some cases, especially those involving while-statements, it might not be possible to extract the logically weakest formula. We just need one which is sufficiently weak to allow us to complete the proof at hand.

$$
\begin{array}{ll}
\quad (\!|\psi[E/x]|\!) & \\
\texttt{x = } E & \\
\quad (\!|\psi|\!) & \textsf{Assignment}
\end{array}
$$

The justification is written against the ψ, since, once the proof has been constructed, we want to read it in a forwards direction. The construction itself proceeds in a backwards direction, because that is the way the assignment axiom facilitates.

Implied. In tableau form, the Implied rule allows us to write one formula ϕ_2 directly underneath another one ϕ_1 with no code in between, provided that ϕ_1 implies ϕ_2 in that the sequent $\vdash_{\mathrm{AR}} \phi_1 \to \phi_2$ is valid. Thus, the Implied rule acts as an interface between predicate logic with arithmetic and program logic. This is a surprising and crucial insight. Our proof calculus for partial correctness is a hybrid system which interfaces with another proof calculus via the Implied proof rule *only*.

When we appeal to the Implied rule, we will usually not explicitly write out the proof of the implication in predicate logic, for this chapter focuses on the program logic. Mostly, the implications we typically encounter will be easy to verify.

The Implied rule is often used to simplify formulas that are generated by applications of the other rules. It is also used when the weakest precondition ϕ' emerges by pushing the postcondition upwards through the whole program. We use the Implied rule to show that the given precondition implies the weakest precondition. Let's look at some examples of this.

Examples 4.13

1. We show that $\vdash_{\mathsf{par}} (\!|y = 5|\!)\ \texttt{x = y + 1}\ (\!|x = 6|\!)$ is valid:

$$
\begin{array}{ll}
\quad (\!|y = 5|\!) & \\
\quad (\!|y + 1 = 6|\!) & \textsf{Implied} \\
\texttt{x = y + 1} & \\
\quad (\!|x = 6|\!) & \textsf{Assignment}
\end{array}
$$

   The proof is constructed from the bottom upwards. We start with $(\!|x = 6|\!)$ and, using the assignment axiom, we push it upwards through `x = y + 1`. This means substituting $y + 1$ for all occurrences of x, resulting in $(\!|y + 1 = 6|\!)$. Now, we compare this with the given precondition $(\!|y = 5|\!)$. The given precondition and the arithmetic fact $5 + 1 = 6$ imply it, so we have finished the proof.

Although the proof is constructed bottom-up, its justifications make sense when read top-down: the second line is implied by the first and the fourth follows from the second by the intervening assignment.

2. We prove the validity of $\vdash_{\mathsf{par}} (\!|y < 3|\!)$ `y = y + 1` $(\!|y < 4|\!)$:

$$
\begin{array}{ll}
(\!|y < 3|\!) & \\
(\!|y + 1 < 4|\!) & \textsf{Implied} \\
\texttt{y = y + 1;} & \\
(\!|y < 4|\!) & \textsf{Assignment}
\end{array}
$$

Notice that Implied always refers to the immediately preceding line. As already remarked, proofs in program logic generally combine two logical levels: the first level is directly concerned with proof rules for programming constructs such as the assignment statement; the second level is ordinary entailment familiar to us from Chapters 1 and 2 plus facts from arithmetic – here that $y < 3$ implies $y + 1 < 3 + 1 = 4$.

We may use ordinary logical and arithmetic implications to change a certain condition ϕ to any condition ϕ' which is implied by ϕ for reasons which have nothing to do with the given code. In the example above, ϕ was $y < 3$ and the implied formula ϕ' was then $y + 1 < 4$. The validity of $\vdash_{\mathrm{AR}} (y < 3) \to (y + 1 < 4)$ is rooted in general facts about integers and the relation $<$ defined on them. Completely formal proofs would require separate proofs attached to all instances of the rule Implied. As already said, we won't do that here as this chapter focuses on aspects of proofs which deal directly with code.

3. For the sequential composition of assignment statements

```
z = x;
z = z + y;
u = z;
```

our goal is to show that u stores the sum of x and y after this sequence of assignments terminates. Let us write P for the code above. Thus, we mean to prove $\vdash_{\mathsf{par}} (\!|\top|\!)\ P\ (\!|u = x + y|\!)$.

We construct the proof by starting with the postcondition $u = x + y$ and pushing it up through the assignments, in reverse order, using the assignment rule.

– Pushing it up through `u = z` involves replacing all occurrences of u by z, resulting in $z = x + y$. We thus have the proof fragment

$$
\begin{array}{ll}
(\!|z = x + y|\!) & \\
\texttt{u = z;} & \\
(\!|u = x + y|\!) & \textsf{Assignment}
\end{array}
$$

– Pushing $z = x + y$ upwards through `z = z + y` involves replacing z by $z + y$, resulting in $z + y = x + y$.

– Pushing that upwards through `z = x` involves replacing z by x, resulting in $x + y = x + y$. The proof fragment now looks like this:

$$
\begin{array}{ll}
\big(x + y = x + y\big) & \\
\texttt{z = x;} & \\
\big(z + y = x + y\big) & \textsf{Assignment} \\
\texttt{z = z + y;} & \\
\big(z = x + y\big) & \textsf{Assignment} \\
\texttt{u = z;} & \\
\big(u = x + y\big) & \textsf{Assignment}
\end{array}
$$

The weakest precondition that thus emerges is $x + y = x + y$; we have to check that this follows from the given precondition $\top$. This means checking that any state that satisfies $\top$ also satisfies $x + y = x + y$. Well, $\top$ is satisfied in all states, but so is $x + y = x + y$, so the sequent $\vdash_{\mathrm{AR}} \top \to (x + y = x + y)$ is valid.
The final completed proof therefore looks like this:

$$
\begin{array}{ll}
\big(\top\big) & \\
\big(x + y = x + y\big) & \textsf{Implied} \\
\texttt{z = x;} & \\
\big(z + y = x + y\big) & \textsf{Assignment} \\
\texttt{z = z + y;} & \\
\big(z = x + y\big) & \textsf{Assignment} \\
\texttt{u = z;} & \\
\big(u = x + y\big) & \textsf{Assignment}
\end{array}
$$

and we can now read it from the top down.

The application of the axiom Assignment requires some care. We describe two pitfalls which the unwary may fall into, if the rule is not applied correctly.

- Consider the example 'proof'

$$
\begin{array}{ll}
\big(x + 1 = x + 1\big) & \\
\texttt{x = x + 1;} & \\
\big(x = x + 1\big) & \textsf{Assignment}
\end{array}
$$

which uses the rule for assignment incorrectly. Pattern matching with the assignment axiom means that ψ has to be $x = x + 1$, the expression E is $x + 1$ and $\psi[E/x]$ is $x + 1 = x + 1$. However, $\psi[E/x]$ is obtained by replacing *all* occurrences of x in ψ by E, thus, $\psi[E/x]$ would have to be equal to $x + 1 = x + 1 + 1$. Therefore, the corrected proof

$$
\begin{array}{ll}
& \{\!|x+1 = x+1+1|\!\} \\
\texttt{x = x + 1;} & \\
& \{\!|x = x+1|\!\} \qquad\qquad \textsf{Assignment}
\end{array}
$$

shows that $\vdash_{\mathsf{par}} \{\!|x+1 = x+1+1|\!\}$ `x = x + 1` $\{\!|x = x+1|\!\}$ is valid.
As an aside, this corrected proof is not very useful. The triple says that, if $x+1 = (x+1)+1$ holds in a state and the assignment `x = x + 1` is executed and terminates, then the resulting state satisfies $x = x+1$; but, since the precondition $x+1 = x+1+1$ can never be true, this triple tells us nothing informative about the assignment.

- Another way of using the proof rule for assignment incorrectly is by allowing additional assignments to happen in between $\psi[E/x]$ and `x = `E, as in the 'proof'

$$
\begin{array}{ll}
& \{\!|x+2 = y+1|\!\} \\
\texttt{y = y + 1000001;} & \\
\texttt{x = x + 2;} & \\
& \{\!|x = y+1|\!\} \qquad\qquad \textsf{Assignment}
\end{array}
$$

This is not a correct application of the assignment rule, since an additional assignment happens in line 2 right before the actual assignment to which the inference in line 4 applies. This additional assignment makes this reasoning unsound: line 2 overwrites the current value in y to which the equation in line 1 is referring. Clearly, $x+2 = y+1$ won't be true any longer. Therefore, we are allowed to use the proof rule for assignment only if there is no additional code between the precondition $\psi[E/x]$ and the assignment `x = `E.

If-statements. We now consider how to push a postcondition upwards through an if-statement. Suppose we are given a condition ψ and a program fragment `if (`B`) {`C_1`} else {`C_2`}`. We wish to calculate the weakest ϕ such that

$$\{\!|\phi|\!\}\ \texttt{if (}B\texttt{) \{}C_1\texttt{\} else \{}C_2\texttt{\}}\ \{\!|\psi|\!\}.$$

This ϕ may be calculated as follows.

1. Push ψ upwards through C_1; let's call the result ϕ_1. (Note that, since C_1 may be a sequence of other commands, this will involve appealing to other rules. If C_1 contains another if-statement, then this step will involve a 'recursive call' to the rule for if-statements.)
2. Similarly, push ψ upwards through C_2; call the result ϕ_2.
3. Set ϕ to be $(B \to \phi_1) \wedge (\neg B \to \phi_2)$.

Example 4.14 Let us see this proof rule at work on the non-optimal code for `Succ` given earlier in the chapter. Here is the code again:

```
a = x + 1;
if (a - 1 == 0)  {
  y = 1;
} else {
  y = a;
}
```

We want to show that $\vdash_{\mathsf{par}} (\!|\top|\!)\ \texttt{Succ}\ (\!|y = x+1|\!)$ is valid. Note that this program is the sequential composition of an assignment and an if-statement. Thus, we need to obtain a suitable midcondition to put between the if-statement and the assignment.

We push the postcondition $y = x + 1$ upwards through the two branches of the if-statement, obtaining

- ϕ_1 is $1 = x + 1$;
- ϕ_2 is $a = x + 1$;

and obtain the midcondition $(a - 1 = 0 \to 1 = x + 1) \wedge (\neg(a - 1 = 0) \to a = x + 1)$ by appealing to a slightly different version of the rule If-statement:

$$\frac{(\!|\phi_1|\!)\ C_1\ (\!|\psi|\!) \qquad (\!|\phi_2|\!)\ C_2\ (\!|\psi|\!)}{(\!|(B \to \phi_1) \wedge (\neg B \to \phi_2)|\!)\ \texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!)}\ \text{If-Statement} \quad (4.7)$$

However, this rule can be derived using the proof rules discussed so far; see exercise 9(c) on page 301. The partial proof now looks like this:

```
  (|⊤|)
  (|?|)                                                    ?
a = x + 1;
  (|(a − 1 = 0 → 1 = x + 1) ∧ (¬(a − 1 = 0) → a = x + 1)|)   ?
if (a - 1 == 0) {
      (|1 = x + 1|)                                        If-Statement
    y = 1;
      (|y = x + 1|)                                        Assignment
} else {
      (|a = x + 1|)                                        If-Statement
    y = a;
      (|y = x + 1|)                                        Assignment
}
  (|y = x + 1|)                                            If-Statement
```

Continuing this example, we push the long formula above the if-statement through the assignment, to obtain

$$(x + 1 - 1 = 0 \to 1 = x + 1) \wedge (\neg(x + 1 - 1 = 0) \to x + 1 = x + 1) \quad (4.8)$$

We need to show that this is implied by the given precondition $\top$, i.e. that it is true in any state. Indeed, simplifying (4.8) gives

$$(x = 0 \to 1 = x + 1) \wedge (\neg(x = 0) \to x + 1 = x + 1)$$

and both these conjuncts, and therefore their conjunction, are clearly valid implications. The above proof now is completed as:

$(\!\vert \top \vert\!)$	
$(\!\vert (x + 1 - 1 = 0 \to 1 = x + 1) \wedge (\neg(x + 1 - 1 = 0) \to x + 1 = x + 1) \vert\!)$	Implied
`a = x + 1;`	
$(\!\vert (a - 1 = 0 \to 1 = x + 1) \wedge (\neg(a - 1 = 0) \to a = x + 1) \vert\!)$	Assignment
`if (a - 1 == 0) {`	
$(\!\vert 1 = x + 1 \vert\!)$	If-Statement
`y = 1;`	
$(\!\vert y = x + 1 \vert\!)$	Assignment
`} else {`	
$(\!\vert a = x + 1 \vert\!)$	If-Statement
`y = a;`	
$(\!\vert y = x + 1 \vert\!)$	Assignment
`}`	
$(\!\vert y = x + 1 \vert\!)$	If-Statement

While-statements. Recall that the proof rule for partial correctness of while-statements was presented in the following form in Figure 4.1 – here we have written η instead of ψ:

$$\frac{(\!\vert \eta \wedge B \vert\!)\ C\ (\!\vert \eta \vert\!)}{(\!\vert \eta \vert\!)\ \texttt{while}\ B\ \{C\}\ (\!\vert \eta \wedge \neg B \vert\!)}\ \textsf{Partial-while.} \tag{4.9}$$

Before we look at how Partial-while will be represented in proof tableaux, let us look in more detail at the ideas behind this proof rule. The formula η is chosen to be an invariant of the body C of the while-statement: provided the boolean guard B is true, if η is true before we start C, and C terminates, then it is also true at the end. This is what the premise $(\!\vert \eta \wedge B \vert\!)\ C\ (\!\vert \eta \vert\!)$ expresses.

Now suppose the while-statement executes a terminating run from a state that satisfies η; and that the premise of (4.9) holds.

- If B is false as soon as we embark on the while-statement, then we do not execute C at all. Nothing has happened to change the truth value of η, so we end the while-statement with $\eta \wedge \neg B$.

- If B is true when we embark on the while-statement, we execute C. By the premise of the rule in (4.9), we know η is true at the end of C.
  - if B is now false, we stop with $\eta \wedge \neg B$.
  - if B is true, we execute C again; η is again re-established. No matter how many times we execute C in this way, η is re-established at the end of each execution of C. The while-statement terminates if, and only if, B is false after some finite (zero including) number of executions of C, in which case we have $\eta \wedge \neg B$.

This argument shows that Partial-while is sound with respect to the satisfaction relation for partial correctness, in the sense that anything we prove using it is indeed true. However, as it stands it allows us to prove only things of the form $(\!|\eta|\!)$ `while (`B`) {`C`}` $(\!|\eta \wedge \neg B|\!)$, i.e. triples in which the postcondition is the same as the precondition conjoined with $\neg B$. Suppose that we are required to prove

$$(\!|\phi|\!) \texttt{ while (}B\texttt{) \{C\}} (\!|\psi|\!) \tag{4.10}$$

for some ϕ and ψ which are not related in that way. How can we use Partial-while in a situation like this?

The answer is that we must *discover* a suitable η, such that

1. $\vdash_{\mathrm{AR}} \phi \to \eta$,
2. $\vdash_{\mathrm{AR}} \eta \wedge \neg B \to \psi$ and
3. $\vdash_{\mathsf{par}} (\!|\eta|\!)$ `while (`B`) {`C`}` $(\!|\eta \wedge \neg B|\!)$

are all valid, where the latter is shown by means of Partial-while. Then, Implied infers that (4.10) is a valid partial-correctness triple.

The crucial thing, then, is the discovery of a suitable invariant η. It is a necessary step in order to use the proof rule Partial-while and in general it requires intelligence and ingenuity. This contrasts markedly with the case of the proof rules for if-statements and assignments, which are purely mechanical in nature: their usage is just a matter of symbol-pushing and does not require any deeper insight.

Discovery of a suitable invariant requires careful thought about what the while-statement is really doing. Indeed the eminent computer scientist, the late E. Dijkstra, said that to understand a while-statement is tantamount to knowing what its invariant is with respect to given preconditions and postconditions for that while-statement.

This is because a suitable invariant can be interpreted as saying that the intended computation performed by the while-statement is correct up to the current step of the execution. It then follows that, when the execution

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 `while (`B`)` $\{C\}$ is a formula η such that $\vDash_{\mathsf{par}} (\!|\eta \wedge B|\!)\, C\, (\!|\eta|\!)$ holds; i.e. for all states l, if η and B are true in l and C is executed from state l and terminates, then η is again true in the resulting state.

Note that η 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 $\bot \wedge B$ is true, then ...' is false, so that implication is true. The formula $\neg B$ is also an invariant of `while (`B`) do` $\{C\}$; but most of these invariants are useless to us, because we are looking for an invariant η for which the sequents $\vdash_{\mathrm{AR}} \phi \to \eta$ and $\vdash_{\mathrm{AR}} \eta \wedge \neg B \to \psi$, are valid, where ϕ and ψ 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;
l1: while (z != x) {
      z = z + 1;
      y = y * z;
l2: }
```

Suppose program execution begins in a store in which x equals 6. When the program flow first encounters the while-statement at location `l1`, z equals 0 and y equals 1, so the condition $z \neq x$ is true and the body is executed. Thereafter at location `l2`, 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

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

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

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 $\vdash_{\text{AR}} \eta \wedge \neg B \rightarrow \psi$ and $\vdash_{\text{AR}} \phi \rightarrow \eta$ 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 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 $\vdash_{\mathrm{AR}} \eta \wedge B \to \eta'$ 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 $\vdash_{\mathrm{AR}} \phi \to \eta$ 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) {
```
$\{\!|y = z! \wedge z \neq x|\!\}$ Invariant Hyp. $\wedge$ guard

$\{\!|y \cdot (z+1) = (z+1)!|\!\}$ Implied
```
    z = z + 1;
```
$\{\!|y \cdot z = z!|\!\}$ 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:

$(\!\vert \top \vert\!)$	
$(\!\vert 1 = 0! \vert\!)$	Implied
`y = 1;`	
$(\!\vert y = 0! \vert\!)$	Assignment
`z = 0;`	
$(\!\vert y = z! \vert\!)$	Assignment
`while (z != x) {`	
$\quad(\!\vert y = z! \wedge z \neq x \vert\!)$	Invariant Hyp. $\wedge$ guard
$\quad(\!\vert y \cdot (z+1) = (z+1)! \vert\!)$	Implied
`  z = z + 1;`	
$\quad(\!\vert y \cdot z = z! \vert\!)$	Assignment
`  y = y * z;`	
$\quad(\!\vert y = z! \vert\!)$	Assignment
`}`	
$(\!\vert y = z! \wedge \neg(z \neq x) \vert\!)$	Partial-while
$(\!\vert y = x! \vert\!)$	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 $a[0], \ldots, a[n-1]$ 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 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. $(\!|\top|\!)\ \texttt{Min\_Sum}\ (\!|\forall i, j\,(0 \leq i \leq j < n \rightarrow s \leq S_{i,j})|\!)$.
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. $(\!|\top|\!)\ \texttt{Min\_Sum}\ (\!|\exists i, j\,(0 \leq i \leq j < n \wedge s = S_{i,j})|\!)$,
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 $\exists i, j$.

$(\!\vert \top \vert\!)$	
$(\!\vert \texttt{Inv1}(a[0],1) \wedge \texttt{Inv2}(a[0],1) \vert\!)$	Implied
`k = 1;`	
$(\!\vert \texttt{Inv1}(a[0],k) \wedge \texttt{Inv2}(a[0],k) \vert\!)$	Assignment
`t = a[0];`	
$(\!\vert \texttt{Inv1}(a[0],k) \wedge \texttt{Inv2}(t,k) \vert\!)$	Assignment
`s = a[0];`	
$(\!\vert \texttt{Inv1}(s,k) \wedge \texttt{Inv2}(t,k) \vert\!)$	Assignment
`while (k != n) {`	
$(\!\vert \texttt{Inv1}(s,k) \wedge \texttt{Inv2}(t,k) \wedge k \neq n \vert\!)$	Invariant Hyp. $\wedge$ guard
$(\!\vert \texttt{Inv1}(\min(s, \min(t + a[k], a[k])), k+1)$ $\wedge \texttt{Inv2}(\min(t + a[k], a[k]), k+1) \vert\!)$	Implied `(Lemma 4.20)`
`t = min(t + a[k], a[k]);`	
$(\!\vert \texttt{Inv1}(\min(s,t), k+1) \wedge \texttt{Inv2}(t, k+1) \vert\!)$	Assignment
`s = min(s,t);`	
$(\!\vert \texttt{Inv1}(s, k+1) \wedge \texttt{Inv2}(t, k+1) \vert\!)$	Assignment
`k = k + 1;`	
$(\!\vert \texttt{Inv1}(s,k) \wedge \texttt{Inv2}(t,k) \vert\!)$	Assignment
}	
$(\!\vert \texttt{Inv1}(s,k) \wedge \texttt{Inv2}(t,k) \wedge \neg\neg(k = n) \vert\!)$	Partial-while
$(\!\vert \texttt{Inv1}(s,n) \vert\!)$	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. □

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 $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `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 $\vdash_{\mathsf{tot}} (\!| x \geq 0 |\!)$ `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\\
95197651298541229654248954659564 57
```

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.