4
Program verification

The methods of the previous chapter are suitable for verifying systems of communicating processes, where control is the main issue, but there are no complex *data*. We relied on the fact that those (abstracted) systems are in a *finite state*. These assumptions are not valid for sequential programs running on a single processor, the topic of this chapter. In those cases, the programs may manipulate non-trivial data and – once we admit variables of type integer, list, or tree – we are in the domain of machines with *infinite* state space.

In terms of the classification of verification methods given at the beginning of the last chapter, the methods of this chapter are

Proof-based. We do not exhaustively check every state that the system can get in to, as one does with model checking; this would be impossible, given that program variables can have infinitely many interacting values. Instead, we construct a proof that the system satisfies the property at hand, using a proof calculus. This is analogous to the situation in Chapter 2, where using a suitable proof calculus avoided the problem of having to check infinitely many models of a set of predicate logic formulas in order to establish the validity of a sequent.

Semi-automatic. Although many of the steps involved in proving that a program satisfies its specification are mechanical, there are some steps that involve some intelligence and that cannot be carried out algorithmically by a computer. As we will see, there are often good heuristics to help the programmer complete these tasks. This contrasts with the situation of the last chapter, which was fully automatic.

Property-oriented. Just like in the previous chapter, we verify properties of a program rather than a full specification of its behaviour.

Application domain. The domain of application in this chapter is sequential transformational programs. 'Sequential' means that we assume the program runs on a single processor and that there are no concurrency issues. 'Transformational' means that the program takes an input and, after some computation, is expected to terminate with an output. For example, methods of objects in Java are often programmed in this style. This contrasts with the previous chapter which focuses on reactive systems that are not intended to terminate and that react continually with their environment.

Pre/post-development. The techniques of this chapter should be used during the coding process for small fragments of program that perform an identifiable (and hence, specifiable) task and hence should be used during the development process in order to avoid functional bugs.

4.1 Why should we specify and verify code?

The task of specifying and verifying code is often perceived as an unwelcome addition to the programmer's job and a dispensable one. Arguments in favour of verification include the following:

- **Documentation:** The specification of a program is an important component in its documentation and the process of documenting a program may raise or resolve important issues. The logical structure of the formal specification, written as a formula in a suitable logic, typically serves as a guiding principle in trying to write an implementation in which it holds.
- **Time-to-market:** Debugging big systems during the testing phase is costly and time-consuming and local 'fixes' often introduce new bugs at other places. Experience has shown that verifying programs with respect to formal specifications can significantly cut down the duration of software development and maintenance by eliminating most errors in the planning phase and helping in the clarification of the roles and structural aspects of system components.
- **Refactoring:** Properly specified and verified software is easier to reuse, since we have a clear specification of what it is meant to do.
- **Certification audits:** Safety-critical computer systems – such as the control of cooling systems in nuclear power stations, or cockpits of modern aircrafts – demand that their software be specified and verified with as much rigour and formality as possible. Other programs may be commercially critical, such as accountancy software used by banks, and they should be delivered with a warranty: a guarantee for correct performance within proper use. The proof that a program meets its specifications is indeed such a warranty.

The degree to which the software industry accepts the benefits of proper verification of code depends on the perceived extra cost of producing it and the perceived benefits of having it. As verification technology improves, the costs are declining; and as the complexity of software and the extent to which society depends on it increase, the benefits are becoming more important. Thus, we can expect that the importance of verification to industry will continue to increase over the next decades. Microsoft's emergent technology A# combines program verification, testing, and model-checking techniques in an integrated in-house development environment.

Currently, many companies struggle with a legacy of ancient code without proper documentation which has to be adapted to new hardware and network environments, as well as ever-changing requirements. Often, the original programmers who might still remember what certain pieces of code are for have moved, or died. Software systems now often have a longer life-expectancy than humans, which necessitates a durable, transparent and portable design and implementation process; the year-2000 problem was just one such example. Software verification provides some of this.

4.2 A framework for software verification

Suppose you are working for a software company and your task is to write programs which are meant to solve sophisticated problems, or computations. Typically, such a project involves an outside customer – a utility company, for example – who has written up an informal description, in plain English, of the real-world task that is at hand. In this case, it could be the development and maintenance of a database of electricity accounts with all the possible applications of that – automated billing, customer service etc. Since the informality of such descriptions may cause ambiguities which eventually could result in serious and expensive design flaws, it is desirable to condense all the requirements of such a project into formal specifications. These formal specifications are usually symbolic encodings of real-world constraints into some sort of logic. Thus, a framework for producing the software could be:

- Convert the informal description R of requirements for an application domain into an 'equivalent' formula ϕ_R of some symbolic logic;
- Write a program P which is meant to realise ϕ_R in the programming environment supplied by your company, or wanted by the particular customer;
- *Prove* that the program P satisfies the formula ϕ_R.

This scheme is quite crude – for example, constraints may be actual design decisions for interfaces and data types, or the specification may 'evolve' and may partly be 'unknown' in big projects – but it serves well as a first approximation to trying to define good programming methodology. Several variations of such a sequence of activities are conceivable. For example, you, as a programmer, might have been given only the formula ϕ_R, so you might have little if any insight into the real-world problem which you are supposed to solve. Technically, this poses no problem, but often it is handy to have both informal and formal descriptions available. Moreover, crafting the informal requirements R is often a mutual process between the client and the programmer, whereby the attempt at formalising R can uncover ambiguities or undesired consequences and hence lead to revisions of R.

This 'going back and forth' between the realms of informal and formal specifications is necessary since it is impossible to 'verify' whether an *informal* requirement R is equivalent to a *formal* description ϕ_R. The meaning of R as a piece of natural language is grounded in common sense and general knowledge about the real-world domain and often based on heuristics or quantitative reasoning. The meaning of a logic formula ϕ_R, on the other hand, is defined in a precise mathematical, qualitative and compositional way by structural induction on the parse tree of ϕ_R – the first three chapters contain examples of this.

Thus, the process of finding a suitable formalisation ϕ_R of R requires the utmost care; otherwise it is always possible that ϕ_R specifies behaviour which is different from the one described in R. To make matters worse, the requirements R are often inconsistent; customers usually have a fairly vague conception of what exactly a program should do for them. Thus, producing a clear and coherent description R of the requirements for an application domain is already a crucial step in successful programming; this phase ideally is undertaken by customers and project managers around a table, or in a video conference, talking to each other. We address this first item only implicitly in this text, but you should certainly be aware of its importance in practice.

The next phase of the software development framework involves constructing the program P and after that the last task is to verify that P satisfies ϕ_R. Here again, our framework is oversimplifying what goes on in practice, since often proving that P satisfies its specification ϕ_R goes hand-in-hand with inventing a suitable P. This correspondence between proving and programming can be stated quite precisely, but that is beyond the scope of this book.

4.2.1 A core programming language

The programming language which we set out to study here is the typical core language of most imperative programming languages. Modulo trivial syntactic variations, it is a subset of Pascal, C, C++ and Java. Our language consists of assignments to integer- and boolean-valued variables, if-statements, while-statements and sequential compositions. Everything that can be computed by large languages like C and Java can also be computed by our language, though perhaps not as conveniently, because it does not have any objects, procedures, threads or recursive data structures. While this makes it seem unrealistic compared with fully blown commercial languages, it allows us to focus our discussion on the process of formal program verification. The features missing from our language could be implemented on top of it; that is the justification for saying that they do not add to the power of the language, but only to the convenience of using it. Verifying programs using those features would require non-trivial extensions of the proof calculus we present here. In particular, dynamic scoping of variables presents hard problems for program-verification methods, but this is beyond the scope of this book.

Our core language has three syntactic domains: integer expressions, boolean expressions and commands – the latter we consider to be our programs. Integer expressions are built in the familiar way from variables $x, y, z, \dots$, numerals $0, 1, 2, \dots, -1, -2, \dots$ and basic operations like addition $(+)$ and multiplication $(*)$. For example,

$$\begin{array}{l} 5 \\ x \\ 4 + (x - 3) \\ x + (x * (y - (5 + z))) \end{array}$$

are all valid integer expressions. Our grammar for generating integer expressions is

$$E \ ::= \ n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E) \tag{4.1}$$

where n is any numeral in $\{\dots, -2, -1, 0, 1, 2, \dots\}$ and x is any variable. Note that we write multiplication in 'mathematics' as $2 \cdot 3$, whereas our core language writes $2 * 3$ instead.

Convention 4.1 In the grammar above, negation $-$ binds more tightly than multiplication $*$, which binds more tightly than subtraction $-$ and addition $+$.

Since if-statements and while-statements contain conditions in them, we also need a syntactic domain B of boolean expressions. The grammar in

Backus Naur form

$$B \ ::= \ \texttt{true} \ | \ \texttt{false} \ | \ (!B) \ | \ (B \,\&\, B) \ \ | \ (B \,||\, B) \ | \ (E < E) \qquad (4.2)$$

uses `!` for the negation, `&` for conjunction and `||` for disjunction of boolean expressions. This grammar may be freely expanded by operators which are definable in terms of the above. For example, the test for equality[1] $E_1 == E_2$ may be expressed via $!(E_1 < E_2) \,\&\, !(E_2 < E_1)$. We generally make use of shorthand notation whenever this is convenient. We also write $(E_1 \,!= E_2)$ to abbreviate $!(E_1 == E_2)$. We will also assume the usual binding priorities for logical operators stated in Convention 1.3 on page 5. Boolean expressions are built on top of integer expressions since the last clause of (4.2) mentions integer expressions.

Having integer and boolean expressions at hand, we can now define the syntactic domain of commands. Since commands are built from simpler commands using assignments and the control structures, you may think of commands as the actual programs. We choose as grammar for commands

$$C \quad ::= \quad \mathrm{x} = E \ | \ C;C \ | \ \texttt{if} \ B \ \{C\} \ \texttt{else} \ \{C\} \ \ | \texttt{while} \ B \ \{C\} \qquad (4.3)$$

where the braces { and } are to mark the extent of the blocks of code in the if-statement and the while-statement, as in languages such as C and Java. They can be omitted if the blocks consist of a single statement. The intuitive meaning of the programming constructs is the following:

1. The atomic command $\mathtt{x} = E$ is the usual assignment statement; it evaluates the integer expression E in the current state of the store and then overwrites the current value stored in x with the result of that evaluation.
2. The compound command $C_1; C_2$ is the sequential composition of the commands C_1 and C_2. It begins by executing C_1 in the current state of the store. If that execution terminates, then it executes C_2 in the storage state resulting from the execution of C_1. Otherwise – if the execution of C_1 does not terminate – the run of $C_1; C_2$ also does not terminate. Sequential composition is an example of a *control structure* since it implements a certain policy of flow of control in a computation.

[1] In common with languages like C and Java, we use a single equals sign $=$ to mean assignment and a double sign $==$ to mean equality. Earlier languages like Pascal used $:=$ for assignment and simple $=$ for equality; it is a great pity that C and its successors did not keep this convention. The reason that $=$ is a bad symbol for assignment is that assignment is not symmetric: if we interpret $x = y$ as the assignment, then x becomes y which is not the same thing as y becoming x; yet, $x = y$ and $y = x$ are the same thing if we mean equality. The two dots in $:=$ helped remind the reader that this is an asymmetric assignment operation rather than a symmetric assertion of equality. However, the notation $=$ for assignment is now commonplace, so we will use it.

3. Another control structure is `if` B $\{C_1\}$ `else` $\{C_2\}$. It first evaluates the boolean expression B in the current state of the store; if that result is true, then C_1 is executed; if B evaluated to false, then C_2 is executed.
4. The third control construct `while` B $\{C\}$ allows us to write statements which are executed repeatedly. Its meaning is that:

   a the boolean expression B is evaluated in the current state of the store;
   b if B evaluates to false, then the command terminates,
   c otherwise, the command C will be executed. If that execution terminates, then we resume at step (a) with a re-evaluation of B as the updated state of the store may have changed its value.

   The point of the while-statement is that it repeatedly executes the command C as long as B evaluates to true. If B never becomes false, or if one of the executions of C does not terminate, then the while-statement will not terminate. While-statements are the only real source of non-termination in our core programming language.

Example 4.2 The factorial $n!$ of a natural number n is defined inductively by

$$\begin{aligned} 0! &\stackrel{\text{def}}{=} 1 \\ (n+1)! &\stackrel{\text{def}}{=} (n+1) \cdot n! \end{aligned} \tag{4.4}$$

For example, unwinding this definition for n being 4, we get $4! \stackrel{\text{def}}{=} 4 \cdot 3! = \cdots = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 24$. The following program `Fac1`:

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

is intended to compute the factorial[2] of x and to store the result in y. We will prove that `Fac1` really does this later in the chapter.

4.2.2 Hoare triples

Program fragments generated by (4.3) commence running in a 'state' of the machine. After doing some computation, they might terminate. If they do, then the result is another, usually different, state. Since our programming

[2] Please note the difference between the formula $x! = y$, saying that the factorial of x is equal to y, and the piece of code `x != y` which says that `x` is not equal to `y`.

language does not have any procedures or local variables, the 'state' of the machine can be represented simply as a vector of values of all the variables used in the program.

What syntax should we use for ϕ_R, the formal specifications of requirements for such programs? Because we are interested in the output of the program, the language should allow us to talk about the variables in the state after the program has executed, using operators like $=$ to express equality and $<$ for less than. You should be aware of the overloading of $=$. In code, it represents an assignment instruction; in logical formulas, it stands for equality, which we write `==` within program code.

For example, if the informal requirement R says that we should

> Compute a number y whose square is less than the input x.

then an appropriate specification may be $y \cdot y < x$. But what if the input x is -4? There is no number whose square is less than a negative number, so it is not possible to write the program in a way that it will work with all possible inputs. If we go back to the client and say this, he or she is quite likely to respond by saying that the requirement is only that the program work for positive numbers; i.e., he or she *revises* the informal requirement so that it now says

> If the input x is a positive number, compute a number whose square is less than x.

This means we need to be able to talk not just about the state *after* the program executes, but also about the state *before* it executes. The assertions we make will therefore be triples, typically looking like

$$(\!|\phi|\!)\ P\ (\!|\psi|\!) \tag{4.5}$$

which (roughly) means:

> If the program P is run in a state that satisfies ϕ, then the state resulting from P's execution will satisfy ψ.

The specification of the program P, to calculate a number whose square is less than x, now looks like this:

$$(\!|x > 0|\!)\ P\ (\!|y \cdot y < x|\!). \tag{4.6}$$

It means that, if we run P in a state such that $x > 0$, then the resulting state will be such that $y \cdot y < x$. It does not tell us what happens if we run P in a state in which $x \leq 0$, the client required nothing for non-positive values of x. Thus, the programmer is free to do what he or she wants in that case. A program which produces 'garbage' in the case that $x \leq 0$ satisfies the specification, as long as it works correctly for $x > 0$.

Let us make these notions more precise.

Definition 4.3 1. The form $(\!|\phi|\!)\, P\, (\!|\psi|\!)$ of our specification is called a Hoare triple, after the computer scientist C. A. R. Hoare.
2. In (4.5), the formula ϕ is called the precondition of P and ψ is called the postcondition.
3. A store or state of core programs is a function l that assigns to each variable x an integer $l(x)$.
4. For a formula ϕ of predicate logic with function symbols $-$ (unary), $+$, $-$, and $*$ (binary); and a binary predicate symbols $<$ and $=$, we say that a state l satisfies ϕ or l is a ϕ-state – written $l \vDash \phi$ – iff $\mathcal{M} \vDash_l \phi$ from page 128 holds, where l is viewed as a look-up table and the model $\mathcal{M}$ has as set A all integers and interprets the function and predicate symbols in their standard manner.
5. For Hoare triples in (4.5), we demand that quantifiers in ϕ and ψ only bind variables that do not occur in the program P.

Example 4.4 For any state l for which $l(x) = -2$, $l(y) = 5$, and $l(z) = -1$, the relation

1. $l \vDash \neg(x + y < z)$ holds since $x + y$ evaluates to $-2 + 5 = 3$, z evaluates to $l(z) = -1$, and 3 is not strictly less than -1;
2. $l \vDash y - x * z < z$ does not hold, since the lefthand expression evaluates to $5 - (-2) \cdot (-1) = 3$ which is not strictly less than $l(z) = -1$;
3. $l \vDash \forall u\, (y < u \to y * z < u * z)$ does not hold; for u being 7, $l \vDash y < u$ holds, but $l \vDash y * z < u * z$ does not.

Often, we do not want to put any constraints on the initial state; we simply wish to say that, no matter what state we start the program in, the resulting state should satisfy ψ. In that case the precondition can be set to $\top$, which is – as in previous chapters – a formula which is true in any state.

Note that the triple in (4.6) does not specify a unique program P, or a unique behaviour. For example, the program which simply does `y = 0;` satisfies the specification – since $0 \cdot 0$ is less than any positive number – as does the program

```
y = 0;
while (y * y < x) {
   y = y + 1;
   }
y = y - 1;
```

This program finds the greatest y whose square is less than x; the while-statement overshoots a bit, but then we fix it after the while-statement.[3]

[3] We could avoid this inelegance by using the `repeat` construct of exercise 3 on page 299.

Note that these two programs have different behaviour. For example, if x is 22, the first one will compute $y = 0$ and the second will render $y = 4$; but both of them satisfy the specification.

Our agenda, then, is to develop a notion of proof which allows us to prove that a program P satisfies the specification given by a precondition ϕ and a postcondition ψ in (4.5). Recall that we developed proof calculi for propositional and predicate logic where such proofs could be accomplished by investigating the structure of the formula one wanted to prove. For example, for proving an implication $\phi \to \psi$ one had to assume ϕ and manage to show ψ; then the proof could be finished with the proof rule for implies-introduction. The proof calculi which we are about to develop follow similar lines. Yet, they are different from the logics we previously studied since they prove triples which are built from two different sorts of things: logical formulas ϕ and ψ versus a piece of code P. Our proof calculi have to address each of these appropriately. Nonetheless, we retain proof strategies which are *compositional*, but now in the structure of P. Note that this is an important advantage in the verification of big projects, where code is built from a multitude of modules such that the correctness of certain parts will depend on the correctness of certain others. Thus, your code might call subroutines which other members of your project are about to code, but you can already check the correctness of your code by assuming that the subroutines meet their own specifications. We will explore this topic in Section 4.5.

4.2.3 Partial and total correctness

Our explanation of when the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds was rather informal. In particular, it did not say what we should conclude if P does not terminate. In fact there are two ways of handling this situation. *Partial correctness* means that we do not require the program to terminate, whereas in *total correctness* we insist upon its termination.

Definition 4.5 (Partial correctness) We say that the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ is satisfied under partial correctness if, for all states which satisfy ϕ, the state resulting from P's execution satisfies the postcondition ψ, provided that P actually terminates. In this case, the relation $\models_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds. We call $\models_{\mathsf{par}}$ the satisfaction relation for partial correctness.

Thus, we insist on ψ being true of the resulting state only if the program P has terminated on an input satisfying ϕ. Partial correctness is rather a weak requirement, since any program which does not terminate at all satisfies its specification. In particular, the program

```
    while true { x = 0; }
```

– which endlessly ‘loops’ and never terminates – satisfies all specifications, since partial correctness only says what must happen *if* the program terminates.

Total correctness, on the other hand, requires that the program terminates in order for it to satisfy a specification.

Definition 4.6 (Total correctness) We say that the triple $(\!|\phi|\!)\ P\ (\!|\psi|\!)$ is satisfied under total correctness if, for all states in which P is executed which satisfy the precondition ϕ, P is guaranteed to terminate and the resulting state satisfies the postcondition ψ. In this case, we say that $\models_{\mathsf{tot}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds and call $\models_{\mathsf{tot}}$ the satisfaction relation of total correctness.

A program which ‘loops’ forever on all input does not satisfy any specification under total correctness. Clearly, total correctness is more useful than partial correctness, so the reader may wonder why partial correctness is introduced at all. Proving total correctness usually benefits from proving partial correctness first and then proving termination. So, although our primary interest is in proving total correctness, it often happens that we have to or may wish to split this into separate proofs of partial correctness and of termination. Most of this chapter is devoted to the proof of partial correctness, though we return to the issue of termination in Section 4.4.

Before we delve into the issue of crafting sound and complete proof calculi for partial and total correctness, let us briefly give examples of typical sorts of specifications which we would like to be able to prove.

Examples 4.7

1. Let `Succ` be the program

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

   The program `Succ` satisfies the specification $(\!|\top|\!)$ `Succ` $(\!|y = (x + 1)|\!)$ under partial and total correctness, so if we think of x as input and y as output, then `Succ` computes the successor function. Note that this code is far from optimal.

In fact, it is a rather roundabout way of implementing the successor function. Despite this non-optimality, our proof rules need to be able to prove this program behaviour.

2. The program `Fac1` from Example 4.2 terminates only if x is initially non-negative – why? Let us look at what properties of `Fac1` we expect to be able to prove.

   We should be able to prove that $\vDash_{\mathsf{tot}} (\!| x \geq 0 |\!)\ \texttt{Fac1}\ (\!| y = x! |\!)$ holds. It states that, provided $x \geq 0$, `Fac1` terminates with the result $y = x!$. However, the stronger statement that $\vDash_{\mathsf{tot}} (\!| \top |\!)\ \texttt{Fac1}\ (\!| y = x! |\!)$ holds should not be provable, because `Fac1` does not terminate for negative values of x.

   For partial correctness, both statements $\vDash_{\mathsf{par}} (\!| x \geq 0 |\!)\ \texttt{Fac1}\ (\!| y = x! |\!)$ and $\vDash_{\mathsf{par}} (\!| \top |\!)\ \texttt{Fac1}\ (\!| y = x! |\!)$ should be provable since they hold.

Definition 4.8 1. If the partial correctness of triples $(\!| \phi |\!)\ P\ (\!| \psi |\!)$ can be proved in the partial-correctness calculus we develop in this chapter, we say that the sequent $\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ is valid.

2. Similarly, if it can be proved in the total-correctness calculus to be developed in this chapter, we say that the sequent $\vdash_{\mathsf{tot}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ is valid.

Thus, $\vDash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ holds if P is partially correct, while the validity of $\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!)$ means that P can be proved to be partially-correct by our calculus. The first one means it is actually correct, while the second one means it is provably correct according to our calculus.

If our calculus is any good, then the relation $\vdash_{\mathsf{par}}$ should be contained in $\vDash_{\mathsf{par}}$! More precisely, we will say that our calculus is *sound* if, whenever it tells us something can be proved, that thing is indeed true. Thus, it is sound if it doesn't tell us that false things can be proved. Formally, we write that $\vdash_{\mathsf{par}}$ is sound if

$$\vDash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ holds whenever } \vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ is valid}$$

for all ϕ, ψ and P; and, similarly, $\vdash_{\mathsf{tot}}$ is sound if

$$\vDash_{\mathsf{tot}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ holds whenever } \vdash_{\mathsf{tot}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ is valid}$$

for all ϕ, ψ and P. We say that a calculus is *complete* if it is able to prove everything that is true. Formally, $\vdash_{\mathsf{par}}$ is complete if

$$\vdash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ is valid whenever } \vDash_{\mathsf{par}} (\!| \phi |\!)\ P\ (\!| \psi |\!) \text{ holds}$$

for all ϕ, ψ and P; and similarly for $\vdash_{\mathsf{tot}}$ being complete.

In Chapters 1 and 2, we said that soundness is relatively easy to show, since typically the soundness of individual proof rules can be established independently of the others. Completeness, on the other hand, is harder to show since it depends on the entire set of proof rules cooperating together. The same situation holds for the program logic we introduce in this chapter. Establishing its soundness is simply a matter of considering each rule in turn – done in exercise 3 on page 303 – whereas establishing its (relative) completeness is harder and beyond the scope of this book.

4.2.4 Program variables and logical variables

The variables which we have seen so far in the programs that we verify are called *program variables*. They can also appear in the preconditions and postconditions of specifications. Sometimes, in order to formulate specifications, we need to use other variables which do not appear in programs.

Examples 4.9

1. Another version of the factorial program might have been `Fac2`:

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

   Unlike the previous version, it 'consumes' the input x. Nevertheless, it correctly calculates the factorial of x and stores the value in y; and we would like to express that as a Hoare triple. However, it is not a good idea to write $(\!|x \geq 0|\!)$ `Fac2` $(\!|y = x!|\!)$ because, if the program terminates, then x will be 0 and y will be the factorial of the initial value of x.

   We need a way of remembering the initial value of x, to cope with the fact that it is modified by the program. Logical variables achieve just that: in the specification $(\!|x = x_0 \wedge x \geq 0|\!)$ `Fac2` $(\!|y = x_0!|\!)$ the x_0 is a logical variable and we read it as being universally quantified in the precondition. Therefore, this specification reads: for all integers x_0, if x equals x_0, $x \geq 0$ and we run the program such that it terminates, then the resulting state will satisfy y equals $x_0!$. This works since x_0 cannot be modified by `Fac2` as x_0 does not occur in `Fac2`.
2. Consider the program `Sum`:

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

   This program adds up the first x integers and stores the result in z. Thus, $(\!|x = 3|\!)$ `Sum` $(\!|z = 6|\!)$, $(\!|x = 8|\!)$ `Sum` $(\!|z = 36|\!)$ etc. We know from Theorem 1.31 on page 41 that $1 + 2 + \cdots + x = x(x + 1)/2$ for all $x \geq 0$, so 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 $\forall$i and $\exists$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 $(\!|\psi|\!)$ `x = 5` $(\!|\psi[x/E]|\!)$ 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 $(\!|\psi[E/x]|\!)\ x = E\ (\!|\psi|\!)$ 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 $(\!|\phi|\!)\ x = E\ (\!|\psi|\!)$, 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 $(\!|\phi|\!)\ x = E\ (\!|\psi|\!)$, 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 $(\!|2 = 2|\!)P(\!|x = 2|\!)$
   b $(\!|2 = 4|\!)P(\!|x = 4|\!)$
   c $(\!|2 = y|\!)P(\!|x = y|\!)$
   d $(\!|2 > 0|\!)P(\!|x > 0|\!)$.

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 $\texttt{x} = \texttt{x} + \texttt{1}$. 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

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

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 $(\!|\top|\!)$ `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.

$$
\dfrac{
\dfrac{
\dfrac{(\!|1=1|\!)\ \texttt{y = 1}\ (\!|y=1|\!)}{(\!|\top|\!)\ \texttt{y = 1}\ (\!|y=1|\!)}\,i
\qquad
\dfrac{(\!|y=1\wedge 0=0|\!)\ \texttt{z = 0}\ (\!|y=1\wedge z=0|\!)}{(\!|y=1|\!)\ \texttt{z = 0}\ (\!|y=1\wedge z=0|\!)}\,i
}{(\!|\top|\!)\ \texttt{y = 1; z = 0}\ (\!|y=1\wedge z=0|\!)}\,c
\qquad
\dfrac{
\dfrac{
\dfrac{
\dfrac{(\!|y\cdot(z+1)=(z+1)!|\!)\ \texttt{z = z+1}\ (\!|y\cdot z=z!|\!)}{(\!|y=z!\wedge z\neq x|\!)\ \texttt{z = z+1}\ (\!|y\cdot z=z!|\!)}\,i
\qquad
(\!|y\cdot z=z!|\!)\ \texttt{y = y*z}\ (\!|y=z!|\!)
}{(\!|y=z!\wedge z\neq x|\!)\ \texttt{z = z+1; y = y*z}\ (\!|y=z!|\!)}\,c
}{(\!|y=z!|\!)\ \texttt{while (z != x) \{z = z+1; y = y*z\}}\ (\!|y=z!\wedge z=x|\!)}\,w
}{(\!|y=1\wedge z=0|\!)\ \texttt{while (z != x) \{z = z+1; y = y*z\}}\ (\!|y=x!|\!)}\,i
}{(\!|\top|\!)\ \texttt{y = 1; z = 0; while (z != x) \{z = z+1; y = y*z\}}\ (\!|y=x!|\!)}\,c
$$

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 $C_1; C_2; \dots ; C_{n-1}; C_n$. 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 $i = 0, 1, \dots , n - 1$. 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

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

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

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

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 $(\!|\phi|\!)\ C_1; \dots ; C_n\ (\!|\psi|\!)$ typically consists of starting with the postcondition ψ and pushing it upwards through C_n, then $C_{n-1}, \dots$, 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 $C_1; C_2; \dots ; C_{n-1}; C_n$ 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.

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

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 $\vdash_{\mathsf{par}} (\!|y = 5|\!)$ `x = y + 1` $(\!|x = 6|\!)$ is valid:

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

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` $(\!|y < 4|\!)$:

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

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

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

– 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:

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

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:

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

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'

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

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

```
    (| x + 1 = x + 1 + 1 |)
x = x + 1;
    (| x = x + 1 |)                Assignment
```

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'

```
    (| x + 2 = y + 1 |)
y = y + 1000001;
x = x + 2;
    (| x = y + 1 |)                Assignment
```

  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

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

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 $\vdash_{\mathsf{par}} (\!|\top|\!)\ \mathtt{Succ}\ (\!|y = x+1|\!)$ 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:

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

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:

```
   $(\!|\top|\!)$
   $(\!|?|\!)$                                                     ?
a = x + 1;
   $(\!|(a-1=0 \to 1=x+1) \wedge (\neg(a-1=0) \to a=x+1)|\!)$      ?
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

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

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

```
  ⦇$\top$⦈
  ⦇$(x+1-1=0 \to 1=x+1) \land (\neg(x+1-1=0) \to x+1=x+1)$⦈    Implied
a = x + 1;
  ⦇$(a-1=0 \to 1=x+1) \land (\neg(a-1=0) \to a=x+1)$⦈          Assignment
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
```

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:

$$\frac{(\!|\eta \land B|\!)\ C\ (\!|\eta|\!)}{(\!|\eta|\!)\ \texttt{while}\ B\ \{C\}\ (\!|\eta \land \neg B|\!)}\ \text{Partial-while.} \tag{4.9}$$

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 $(\!|\eta \land B|\!)\ C\ (\!|\eta|\!)$ 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 $\eta \land \neg B$.

- 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. $\vdash_{\mathsf{par}} (\!|\eta|\!)\ \texttt{while}\ (B)\ \{C\}\ (\!|\eta \wedge \neg B|\!)$

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 $\models_{\mathsf{par}} (\!|\, \eta \wedge B \,|\!)\; C\; (\!|\, \eta \,|\!)$ 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 $y = 1 \land z = 0$ based on the initial assignments and their precondition $0! \overset{\text{def}}{=} 1$,
- 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

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

are valid.

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

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

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

4. Try to prove that $\vdash_{\mathrm{AR}} \eta \wedge B \to \eta'$ 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. ∧ 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:

```
    $(\!|\top|\!)$
    $(\!|1 = 0!|\!)$                              Implied
y = 1;
    $(\!|y = 0!|\!)$                              Assignment
z = 0;
    $(\!|y = z!|\!)$                              Assignment
while (z != x) {
        $(\!|y = z! \wedge z \neq x|\!)$              Invariant Hyp. ∧ guard
        $(\!|y \cdot (z + 1) = (z + 1)!|\!)$          Implied
    z = z + 1;
        $(\!|y \cdot z = z!|\!)$                      Assignment
    y = y * z;
        $(\!|y = z!|\!)$                              Assignment
}
    $(\!|y = z! \wedge \neg(z \neq x)|\!)$            Partial-while
    $(\!|y = x!|\!)$                              Implied
```

4.3.3 A case study: minimal-sum section

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

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

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

Definition 4.18 Let $a[0], \ldots, a[n-1]$ 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. $(\!|\top|\!)$ `Min_Sum` $(\!|\forall i, j\,(0 \le i \le j < n \to s \le S_{i,j})|\!)$.
It says that, after the program terminates, s is less than or equal to, the sum of any section of the array. Note that i and j are logical variables in that they don't occur as program variables.

S2. $(\!|\top|\!)$ `Min_Sum` $(\!|\exists i, j\,(0 \le i \le j < n \wedge s = S_{i,j})|\!)$,
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

$$\texttt{Inv1}(s,k) \overset{\text{def}}{=} \forall i, j\,(0 \le i \le j < k \to s \le S_{i,j}) \tag{4.12}$$

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 `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 $\exists i, j$.

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

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

computation is

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

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

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

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 $\texttt{while}\ B\ \{C\}$ 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 $\vdash_{\mathsf{tot}} \llparenthesis x \geq 0 \rrparenthesis$ `Fac1` $\llparenthesis y = x! \rrparenthesis$ 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:

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

and so $\vdash_{\mathsf{tot}} \llparenthesis x \geq 0 \rrparenthesis$ `Fac1` $\llparenthesis y = x! \rrparenthesis$ is valid. Two comments are in order:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 Programming by contract

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

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

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

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

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

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

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

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

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

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

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

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

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

What about the postconditions of `choose`? Since the method body declared no local variables, we use `result` to denote the return value of this method. The postcondition then states that `result` equals $\binom{n}{k}$ – assuming that you boss' equation (4.16) is correct for your preconditions $0 \le k$, $0 \le n$, and $k \le n$. The contract for `choose` is therefore

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

From this we learn that programming by contract uses contracts

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

Programming by contract therefore gives rise to *program validation by contract.* One proves the 'Hoare triple' $(\!|$`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

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

Implied (Lemma 4.20)

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

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

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

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

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

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

4.6 Exercises

Exercises 4.1

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

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

———

Exercises 4.2

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

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

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

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

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

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

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

Exercises 4.3

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

$$\frac{\vdash_{\mathrm{AR}} \phi \to \phi' \qquad (\!|\phi|\!)\ C\ (\!|\psi|\!) \qquad \vdash_{\mathrm{AR}} \psi' \to \psi}{(\!|\phi'|\!)\ C\ (\!|\psi'|\!)}\ \mathsf{Implied\_Reversed}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[7] You may have to strengthen your invariant.

for Hoare triples.

(a) Show that this proof rule is sound for $\vDash_{\mathsf{par}}$.

(b) Derive this proof rule from the ones on page 270.

(c) Explain how this rule, or its derived version, is used to establish the overall correctness of `Min_Sum`.

26. The maximal-sum problem is to compute the maximal sum of all sections on an array.

(a) Adapt the program from page 289 so that it computes the maximal sum of these sections.

(b) Prove the partial correctess of your modified program.

(c) Which aspects of the correctness proof given in Figure 4.3 (page 291) can be 're-used?'

Exercises 4.4

1. Prove the validity of the following total-correctness sequents:

* (a) $\vdash_{\mathsf{tot}} (\!|x \geq 0|\!)$ `Copy1` $(\!|x = y|\!)$

* (b) $\vdash_{\mathsf{tot}} (\!|y \geq 0|\!)$ `Multi1` $(\!|z = x \cdot y|\!)$

(c) $\vdash_{\mathsf{tot}} (\!|(y = y_0) \wedge (y \geq 0)|\!)$ `Multi2` $(\!|z = x \cdot y_0|\!)$

* (d) $\vdash_{\mathsf{tot}} (\!|x \geq 0|\!)$ `Downfac` $(\!|y = x!|\!)$

* (e) $\vdash_{\mathsf{tot}} (\!|x \geq 0|\!)$ `Copy2` $(\!|x = y|\!)$, does your invariant have an active part in securing correctness?

(f) $\vdash_{\mathsf{tot}} (\!|\neg(y = 0)|\!)$ `Div` $(\!|(x = d \cdot y + r) \wedge (r < y)|\!)$.

2. Prove total correctness of **S1** and **S2** for `Min_Sum`.
3. Prove that $\vdash_{\mathsf{par}}$ is sound for $\vDash_{\mathsf{par}}$. Just like in Section 1.4.3, it suffices to assume that the premises of proof rules are instances of $\vDash_{\mathsf{par}}$. Then, you need to prove that their respective conclusion must be an instance of $\vDash_{\mathsf{par}}$ as well.
4. Prove that $\vdash_{\mathsf{tot}}$ is sound for $\vDash_{\mathsf{tot}}$.
5. Implement program `Collatz` in a programming language of your choice such that the value of `x` is the program's input and the final value of `c` its output. Test your program on a range of inputs. Which is the biggest integer for which your program terminates without raising an exception or dumping the core?
6. A function over integers $f\colon \mathbb{I} \to \mathbb{I}$ is affine iff there are integers a and b such that $f(x) = a \cdot x + b$ for all $x \in \mathbb{I}$. The else-branch of the program `Collatz` assigns to `c` the value $f(c)$, where f is an affine function with $a = 3$ and $b = 1$.

(a) Write an parameterized implementation of `Collatz` in which you can initially specify the values of a and b either statically or through keyboard input such that the else-branch assigns to `c` the value of $f(c)$.

(b) Determine for which pairs $(a, b) \in \mathbb{I} \times \mathbb{I}$ the set $\mathrm{Pos} \stackrel{\mathrm{def}}{=} \{x \in \mathbb{I} \mid 0 < x\}$ is invariant under the affine function $f(x) = a \cdot x + b$: for all $x \in \mathrm{Pos}$, $f(x) \in \mathrm{Pos}$.

* (c) Find an affine function that leaves Pos invariant, but not the set $\mathrm{Odd} \stackrel{\mathrm{def}}{=} \{x \in \mathbb{I} \mid \exists y \in \mathbb{I}\colon x = 2 \cdot y + 1\}$, such that there is an input drawn from Pos whose

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

——

Exercises 4.5

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

* 2. Consider the method

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

named `withdraw` which attempts to withdraw `amount` from an integer field `balance` of the class within which method `withdraw` lives. This method makes use of another method `isGood` which returns `true` iff the value of `balance` is greater or equal to the value of `amount`.

(a) Write a contract for method `isGood`.

(b) Use that contract to show the validity of the contract for `withdraw`:

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

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

——

4.7 Bibliographic notes

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

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

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