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 into, 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 avoids 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.

4.1 Why should we specify and verify code?

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.

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: Property 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 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 $\phi_R$ of some symbolic logic;
- Write a program $P$ which is meant to realise $\phi_R$ in the programming environment supplied by your company, or wanted by the particular customer;
- Prove that the program $P$ satisfies the formula $\phi_R$.

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 $\phi_R$, 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 $\phi_R$. 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 $\phi_R$, on the other hand, is defined in a precise mathematical, qualitative and compositional way by structural induction on the parse tree of $\phi_R$ — the first three chapters contain examples of this.

Thus, the process of finding a suitable formalisation $\phi_R$ of $R$ requires the utmost care; otherwise it is always possible that $\phi_R$ 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 $\phi_R$. Here again, our framework is oversimplifying what goes on in practice, since often proving that $P$ satisfies its specification $\phi_R$ 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 \(x, y, z, \ldots\), numerals \(0, 1, 2, \ldots, -1, -2, \ldots\) and basic operations like addition (+) and multiplication (*). For example,

\[
\begin{align*}
5 \\
x \\
4 + (x - 3) \\
x + (x * (y - (5 + z)))
\end{align*}
\]

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

\[
E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)
\]

where \(n\) is any numeral in \(-\ldots,-2,-1,0,1,2,\ldots\) 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

4.2 A framework for software verification

Backus Naur form

\[
B ::= \text{true} \mid \text{false} \mid (B) \mid (B \& B) \mid (B \| B) \mid (B < E)
\]

uses \(\wedge\) 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 \(E_1 == E_2\) may be expressed via \(! (E_1 < E_2) \& (E_2 < E_1)\). We generally make use of shorthand notation wherever 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

\[
C ::= x = E \mid C; C \mid \text{if } B \{C\} \text{ else } \{C\} \mid \text{while } B \{C\}
\]

where the braces \(\{\text{ and }\}\) 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 \(x = E\) 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 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.

3 In common with languages like C and Java, we use a single equals sign = to mean assignment and an equals 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.
4 Program verification

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 \( B \) evaluates 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

\[
0! \overset{\text{def}}{=} 1
\]
\[
(n + 1)! \overset{\text{def}}{=} (n + 1) \cdot n!
\]

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

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

is intended to compute the factorial\(^2\) of \( z \) 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 \( a! = y \), 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 \( y \).
Let us make these notions more precise.

**Definition 4.3.1.** The form $\langle \phi \rangle P \langle \psi \rangle$ of our specification is called a Hoare triple, after the computer scientist C. A. R. Hoare.

2. In (4.5), the formula $\phi$ is called the precondition of $P$ and $\psi$ 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 $\phi$ of predicate logic with function symbols $-$ (unary), $+$, $-$, and $*$ (binary); and a binary predicate symbols $<$ and $=$, we say that a state $l$ satisfies $\phi$ or $l \models \phi$ is a $\phi$-state - written $l \models \phi$ - iff $M \models \phi$ page 128 holds, where $l$ is viewed as a look-up table and the model $M$ 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 $\phi$ and $\psi$ 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. $l \models \neg(x + y < z)$ holds since $x + y$ evaluates to $-2 + 5 = 3$, $z$ evaluates to $l(z) = -1$, and $3$ is not strictly less than $-1$;

2. $l \models y - x > z > 2$ does not hold, since the left-hand expression evaluates to $5 - (-2) = 7$ which is not strictly less than $l(z) = -1$;

3. $l \models \forall u(y < u \rightarrow y < z < u + z)$ does not hold; for $u$ being $7$, $l \models y < z < u + z$ 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 $\psi$. 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

$$y = 0;$$

$$\text{while } (y + y < x) \{$$
$$\quad y = y + 1;$$
$$\}\}$$

$$\text{y = y - 1;}$$

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 206.
specification. In particular, the program

\[
\text{while true \{} x = 0; \}\]

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 \(\phi\), \(P\) is guaranteed to terminate and the resulting state satisfies the postcondition \(\psi\). In this case, we say that \(F_{\text{tot}}(\phi)\ P \ (\psi)\) holds and call \(F_{\text{tot}}\) 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 \(S\) be the program

\[
\begin{align*}
    a &= x + 1; \\
    \text{if} \ (a - 1 <= 0) \ {\} \\
    y &= 1; \\
    \text{else} \ {\} \\
    y &= a;
\end{align*}
\]

The program \(S\) satisfies the specification \(\{T\} \ S \ (y = (x + 1))\) under partial and total correctness, so if we think of \(x\) as input and \(y\) as output, then \(S\) 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 \(F_{\text{fact1}}\) from Example 4.2 terminates only if \(x\) is initially non-negative – why? Let us look at what properties of \(F_{\text{fact1}}\) we expect to be able to prove.

We should be able to prove that \(F_{\text{tot}}(x >= 0) \ P \ (y = x!cards)\) holds. It states that, provided \(x >= 0\), \(F_{\text{fact1}}\) terminates with the result \(y = x!\). However, the stronger statement that \(F_{\text{tot}}(x!\ P \ (y = x!cards))\) holds should not be provable, because \(F_{\text{fact1}}\) does not terminate for negative values of \(x\).

For partial correctness, both statements \(F_{\text{par}}(x >= 0) \ P \ (y = x!cards)\) and \(F_{\text{par}}(x!\ P \ (y = x!cards))\) 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 \(\vdash_{\text{par}}(\phi) \ P \ (\psi)\) 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 \(\vdash_{\text{tot}}(\phi) \ P \ (\psi)\) is valid.

Thus, \(\vdash_{\text{par}}(\phi) \ P \ (\psi)\) holds if \(P\) is partially correct, while the validity of \(\vdash_{\text{par}}(\phi) \ P \ (\psi)\) 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 \(\vdash_{\text{par}}\) should be contained in \(\vdash_{\text{tot}}\). 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 \(\vdash_{\text{par}}\) is sound if

\[
\vdash_{\text{par}}(\phi) \ P \ (\psi)\] holds whenever \(\vdash_{\text{par}}(\phi) \ P \ (\psi)\) is valid

for all \(\phi\), \(\psi\) and \(P\); and, similarly, \(\vdash_{\text{tot}}\) is sound if

\[
\vdash_{\text{tot}}(\phi) \ P \ (\psi)\] holds whenever \(\vdash_{\text{tot}}(\phi) \ P \ (\psi)\) is valid

for all \(\phi\), \(\psi\) and \(P\). We say that a calculus is complete if it is able to prove everything that is true. Formally, \(\vdash_{\text{par}}\) is complete if

\[
\vdash_{\text{par}}(\phi) \ P \ (\psi)\] is valid whenever \(\vdash_{\text{par}}(\phi) \ P \ (\psi)\) holds

for all \(\phi\), \(\psi\) and \(P\); and similarly for \(\vdash_{\text{tot}}\) 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 \((x \geq 0) \text{Fac2 } (y = x)\) 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 \((x = x_0 \land x \geq 0) \text{Fac2 } (y = x_0)\) the \(x_0\) is a logical variable and we read it as being universally quantified in the preconditions. Therefore, the 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, \((x = 0) \text{Sum } (z = 0), (x = 0) \text{Sum } (z = 30)\) etc. We know from Theorem 1.31 on page 41 that \(1 + 2 + \ldots + z = \frac{z(z + 1)}{2}\) for all \(z \geq 0\), so we would like to express, as a Hoare triple, that the value of \(z\) upon termination is \(x_0(2x_0 + 1)/2\) where \(x_0\) is the initial value of \(x\). Thus, we write \((x = x_0 \land x \geq 0) \text{Sum } (z = x_0(2x_0 + 1)/2)\).

   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 \(\forall\) and \(\exists\) 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 \(\phi\) or \(\psi\), 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 tableau. 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

\[(\phi) C_1; C_2 (\psi).\]
4. Program verification

\[ \begin{align*}
(\phi) & \quad C_1 (\eta) \\
(\eta) & \quad C_2 (\psi) \\
\hline
(\phi) & \quad C_1; C_2 (\psi) \\
\hline
\end{align*} \]

Composition

\[ \begin{align*}
(\psi) & \equiv E (\psi) \\
\hline
\end{align*} \]

Assignment

\[ \begin{align*}
(\phi \land B) & \quad C_1 (\psi) \\
(\phi \land \neg B) & \quad C_2 (\psi) \\
\hline
(\phi) & \quad \text{if-statement} \\
\end{align*} \]

\[ \begin{align*}
(\psi \land B) & \quad C (\psi) \\
(\psi) & \quad \text{while-statement} \\
\hline
\end{align*} \]

Partial-while

\[ \begin{align*}
f_{\text{AR}} (\phi') \rightarrow \phi \\
(\phi) & \quad C (\psi) \\
\hline
\end{align*} \]

Implied

\[ \begin{align*}
(\phi') & \quad C (\psi') \\
\hline
\end{align*} \]

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

Thus, if we know that \( C_1 \) takes \( \phi \)-states to \( \eta \)-states and \( C_2 \) takes \( \eta \)-states to \( \psi \)-states, then running \( C_1 \) and \( C_2 \) in that sequence will take \( \phi \)-states to \( \psi \)-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 \( \eta \) and prove \((\phi) C_1 (\eta)\) and \((\eta) C_2 (\psi)\). If \( C_1; C_2 \) runs on input satisfying \( \phi \) and we need to show that the store satisfies \( \psi \) 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 \( \eta \) which, considered as input for \( C_2 \), should result in an output satisfying \( \psi \). We call \( \eta \) 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 \( \psi \) 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 \( \psi \) 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 \( \psi \) 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 \( \psi \) holds in a state in which we perform the assignment \( x = E \), then surely

4.3 Proof calculus for partial correctness

\[ \psi[E/x] \] 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 \( \psi \) be \( x = 6 \) and \( E \) be 5. Then \((\psi) x = 5 \quad (\psi[x/E])\) 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 \( 6 = 5 \) 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 \( \psi \) holds in the resulting state. Since \( \psi \) 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, \( \psi \) with \( E \) in place of \( x \) – which says whatever \( \psi \) 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 \( \psi \) and we wish to find \( \phi \) such that \((\phi) x = E(\psi)\), it is easy: we simply set \( \phi \) to be \((\psi[E/x])\); but, if we know \( \phi \) and we want to find \( \psi \) such that \((\phi) x = E(\psi)\), there is no easy way of getting a suitable \( \psi \). 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 \( \psi \), we replace all the free occurrences of \( x \) in \( \psi \). 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 \( x \leftarrow 2 \). The following are instances of axiom Assignment:

\[
\begin{align*}
\text{a} & \quad (2 = 2) P (x = 2) \\
\text{b} & \quad (2 = 4) P (x = 4) \\
\text{c} & \quad (2 = y) P (x = y) \\
\text{d} & \quad (2 > 0) P (x > 0).
\end{align*}
\]
4 Program verification

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 \( x = 2 \) 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,

\[
(L) x = E \psi \text{ holds for any } E \text{ and } \psi - \text{why?}
\]

c. If you want to prove \( x = y \) after the assignment, you will need to prove that \( x = 2 \) before it.

d. To prove \( x > 0 \), we'd better have \( x > 2 \) prior to the execution of \( P \).

2. Suppose \( P \) is \( x = x + 1 \). 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. \( \{ x + 1 = 5 \} P \{ x + 5 = y \} \)

d. \( \{ x + 1 > 0 \land y > 0 \} P \{ x > 0 \land y > 0 \} \).

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

\[
(\phi) \text{ if } B \{ C_1 \} \text{ else } \{ C_2 \} (\psi)
\]

by decomposing it into two triples, subgoals corresponding to the cases of \( B \) evaluating to true and to false. Typically, the precondition \( \phi \) 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 \( C_1 \) will be executed and hence \( C_1 \) will have to translate \( \phi \) states to \( \psi \) 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 \( (\phi \land B) \{ C_1 \} (\psi) \) and \( (\phi \land \neg B) \{ C_2 \} (\psi) \). 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 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' \( \mu \). In general, the body \( C \) of the command \( \text{while } (B) \{ C \} \) 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, \( \psi \) expresses this invariant; the rule's premise, \( (\psi \land B) \{ C \} (\psi) \), states that, if \( \psi \) and \( B \) are true before we execute \( C \), and \( C \) terminates, then \( \psi \) will be true after it. The conclusion of Partial-while states that, no matter how many times the body \( C \) is executed, if \( \psi \) is true initially and the while-statement terminates, then \( \psi \) 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 \( \phi' \) which implies \( \phi \) and another one \( \psi' \) which is implied by \( \psi \), then we should also be allowed to prove that \( (\phi') \{ P \} (\psi') \). A sequent \( \varphi \rightarrow \psi \) is valid iff there is a proof of \( \phi' \) in the natural deduction calculus for predicate logic, where \( \varphi \) 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 \( \{ T \} \text{Fac1} \{ y = z! \} \) where \( \text{Fac1} \) 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 \( \varphi \rightarrow \psi \) 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.
4.3 Proof calculus for partial correctness

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 Fact1 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

\[ C_1; \]
\[ C_2; \]
\[ \vdots \]
\[ C_n \]

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

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

\[ (\phi_0) \]
\[ C_1; \]
\[ (\phi_1) \text{ justification} \]
\[ C_2; \]
\[ \vdots \]
\[ (\phi_{n-1}) \text{ justification} \]
\[ C_n; \]
\[ (\phi_n) \text{ justification} \]

Figure 4.2. A partial-correctness proof for Fact1 in tree form.
4 Program verification

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

Each of the transitions

\[
\begin{align*}
& (\phi_i) \\
& C_{i+1} \\
& (\phi_{i+1})
\end{align*}
\]

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 \( \phi_i \) be found? In principle, it seems as though one could start from \( \phi_0 \) and, using \( C_i \), obtain \( \phi_2 \) and continue working downwards. However, because the assignment rule works backwards, it turns out that it is more convenient to start with \( \phi_n \) and work upwards, using \( C_n \) to obtain \( \phi_{n-1} \) etc.

**Definition 4.12** The process of obtaining \( \phi_i \) from \( C_{i+1} \) and \( \phi_{i+1} \) is called computing the weakest precondition \( \psi_{C_{i+1}} \) of \( C_{i+1} \), given the postcondition \( \phi_{i+1} \).

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 \( \phi_{i+1} \).

The construction of a proof tableau for \( (\psi) C_1; \ldots; C_n (\psi) \) typically consists of starting with the postcondition \( \psi \) and pushing it upwards through \( C_n \), then \( C_{n-1} \), \ldots, until a formula \( \psi' \) emerges at the top. Ideally, the formula \( \psi' \) represents the weakest precondition which guarantees that the \( \psi \) will hold if the composed program \( C_1 C_2; \ldots; C_{n-1} C_n \) is executed and terminates. The weakest precondition \( \psi' \) is then checked to see whether it follows from the given precondition \( \psi \). 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:

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

4.3 Proof calculus for partial correctness

\[
\begin{align*}
& (\psi[E/z]) \\
& x = E \\
& (\psi) \quad \text{Assignment}
\end{align*}
\]

The justification is written against the \( \psi \), 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 \( \phi_2 \) directly underneath another one \( \phi_1 \) with no code in between, provided that \( \phi_1 \) implies \( \phi_2 \) in that the sequent \( \vdash_{PC} \phi_1 \Rightarrow \phi_2 \) 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 \( \phi' \) 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 \( \vdash_{PC} (y = 5) x = y + 1 (z = 6) \) is valid:

\[
\begin{align*}
& (y = 5) \\
& (y + 1 = 6) \quad \text{Implied} \\
& x = y + 1 \\
& (x = 6) \quad \text{Assignment}
\end{align*}
\]

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 \( y + 1 = 6 \) 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 $\vdash_{PR}(y < 3) \ y = y + 1 \ (y < 4)$:

\[
\begin{align*}
(y < 3) & \\
(y + 1 < 4) & \text{Implied} \\
y = y + 1; & \\
(y < 4) & \text{Assignment}
\end{align*}
\]

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 $\phi$ to any condition $\phi'$ which is implied by $\phi$ for reasons which have nothing to do with the given code. In the example above, $\phi$ was $y < 3$ and the implied formula $\phi'$ was then $y + 1 < 4$. The validity of $\vdash_{PR}(y < 3) \rightarrow (y + 1 < 4)$ 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

\[
\begin{align*}
z &= z; \\
z &= z + y; \\
u &= z;
\end{align*}
\]

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 $\vdash_{PR}(T) \ P (u = x + y)$.

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 $z$. This results in $z = x + y$. We thus have the proof fragment

\[
\begin{align*}
(z &= x + y) \\
& (u = x + y).
\end{align*}
\]

- Pushing $z = x + y$ upwards through $z = z + y$ involves replacing $z$ by $x + y$ resulting in $x + y = x + y$.

3.4 Proof calculus for partial correctness

- Pushing this upwards through $z = x$ by $x$, resulting in $x + y = x + y$. The proof fragment now looks like this:

\[
\begin{align*}
\{(x + y = x + y) \\
z &= x; \\
\{(z + y = x + y) \\
z &= z + y; \\
\{(u = x + y).
\end{align*}
\]

The weakest precondition that thus emerges is $x + y = x + y$; we have to check that this follows from the given precondition $T$. This means checking that any state that satisfies $T$ also satisfies $x + y = x + y$. Well, $T$ is satisfied in all states, but so is $x + y = x + y$, so the sequent $\vdash_{PR} T \rightarrow (x + y = x + y)$ is valid.

The final completed proof therefore looks like this:

\[
\begin{align*}
\{(x + y = x + y) \\
z &= x; \\
\{(z + y = x + y) \\
z &= z + y; \\
\{(u = x + y).
\end{align*}
\]

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' 

\[
\begin{align*}
\{(x + 1 = x + 1) \\
x &= x + 1; \\
\{(x = x + 1).
\end{align*}
\]

which uses the rule for assignment incorrectly. Pattern matching with the assignment axiom means that $\psi$ 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 $\psi$ by $E$, thus, $\psi[E/x]$ would have to be equal to $x + 1 = x + 1$. Therefore, the corrected proof
shows that $I_{opp}(x + 1 = y + 1 + 1) \Rightarrow x = x' + 1 \Rightarrow \psi$ 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 precondi-
tion $x + 1 = x' + 1 + 1$ 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 addi-
tional assignments to happen in between $\psi[E/x]$ and $x = E$, say in the "proof"

$\{x = y + 1\}$
$y = y + 1: 000001;
\{x = y + 1\}$

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 un-
sound: line 2 overwrites the current value in $y$ to which the equation in line 1
is referring. Clearly, $x + 1 = x + 1$ 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 $\psi$ and a program fragment
if $(B) \{C_1\}$ else $(C_2)$. We wish to calculate the weakest $\phi$ such that

$\phi \text{ if } (B) \{C_1\} \text{ else } \{C_2\} \{\psi\}$.

This $\phi$ may be calculated as follows.

1. Push $\psi$ upwards through $C_1$; let's call the result $\phi_1$. (Note that, since $C_1$
is 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 $\phi$ upwards through $C_2$; call the result $\phi_2$.
3. Set $\phi$ to be $(B \rightarrow \phi_1) \land (\neg B \rightarrow \phi_2)$.

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

$\{x + 1 - 1 = 0 \rightarrow 1 = x + 1\} \land \neg (x + 1 - 1 = 0) \rightarrow x + 1 = x + 1$ (4.8)
We need to show that this is implied by the given precondition \( \Pi \), i.e., that it is true in any state. Indeed, simplifying (4.8) gives

\[
(x = 0 \rightarrow 1 = x + 1) \land (- (x + 1) \rightarrow x + 1 = x + 1)
\]

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

\[
\begin{align*}
&\{x = 0\} \rightarrow \neg \neg \Pi, \\
&\{x = 1\} \land \Pi \rightarrow \Pi,
\end{align*}
\]

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 \((\Pi) \text{ while } B \{C\} \{\Pi \land \neg \neg B\}\), i.e., triples in which the postcondition is the same as the precondition conjoined with \(\neg B\). Suppose that we are required to prove

\[
(\Pi) \text{ while } B \{C\} \{\Pi \land \neg \neg B\}
\]

We some \(\psi\) and \(\eta\), 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 \(\eta\), such that

\[
\text{ARS } \Pi \leftarrow \neg \neg B
\]

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 \(\eta\). 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; our discussion should not be seen as deviating from his ideas.