# 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 $\phi_R$ 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,

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

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 $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ 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

Backus Naur form

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

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

$$C ::= \mathtt{x} = E \mid C;C \mid \texttt{if}\; B\; \{C\}\; \texttt{else}\; \{C\} \mid \texttt{while}\; B\; \{C\} \qquad (4.3)$$

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 $\phi_R$, 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 $\phi$, then the state resulting from $P$'s execution will satisfy $\psi$.

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 $\phi$ 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 $l$ is a $\phi$-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 $\phi$ 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. $l \vDash \neg(x + y < z)$ 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 $\psi$. 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;
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 $\phi$ and a postcondition $\psi$ 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 $\phi$ and manage to show $\psi$; 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 $\phi$ and $\psi$ 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 $\phi$, the state resulting from $P$'s execution satisfies the postcondition $\psi$, provided that $P$ actually terminates. In this case, the relation $\vDash_{\mathsf{par}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds. We call $\vDash_{\mathsf{par}}$ the satisfaction relation for partial correctness.

Thus, we insist on $\psi$ being true of the resulting state only if the program $P$ has terminated on an input satisfying $\phi$. 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 $\phi$, $P$ is guaranteed to terminate and the resulting state satisfies the postcondition $\psi$. In this case, we say that $\vDash_{\mathsf{tot}} (\!|\phi|\!)\ P\ (\!|\psi|\!)$ holds and call $\vDash_{\mathsf{tot}}$ 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 $(\!|\top|\!)\ \mathtt{Succ}\ (\!|y = (x+1)|\!)$ 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 $\vDash_{\mathsf{tot}} (\!| x \geq 0 |\!)\ \mathtt{Fac1}\ (\!| y = x! |\!)$ holds. It states that, provided $x \geq 0$, `Fac1` terminates with the result $y = x!$. However, the stronger statement that $\vDash_{\mathsf{tot}} (\!| \top |\!)\ \mathtt{Fac1}\ (\!| y = x! |\!)$ holds should not be provable, because `Fac1` does not terminate for negative values of $x$.

   For partial correctness, both statements $\vDash_{\mathsf{par}} (\!| x \geq 0 |\!)\ \mathtt{Fac1}\ (\!| y = x! |\!)$ and $\vDash_{\mathsf{par}} (\!| \top |\!)\ \mathtt{Fac1}\ (\!| y = x! |\!)$ 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 $\phi$, $\psi$ 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 $\phi$, $\psi$ 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 $\phi$, $\psi$ 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 $\phi$ 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 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

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