Lecture Notes on Program Correctness

Wim H. Hesselink
Gerard R. Renardel de Lavalette

University of Groningen, the Netherlands
Last modification: February 27, 2021

Contents

Chapter 1

Introduction

1.1 The meaning of program correctness

Program Correctness deals with the correctness of computer programs (i.e., software). What does it mean: correctness? According to the *Oxford Dictionary*, the word *correct* has the meaning

> free from error, in accordance with fact or truth.

In the context of programs, however, correctness is a relative concept. Whether a program is correct or not, depends on the specification:

> a program is correct if it satisfies its specification.

For larger programs and for software systems, it is often impossible to give a complete specification. Such systems have requirements. There is a whole field within software engineering to collect requirements and to investigate whether the requirements are satisfied. In this course, however, we restrict the attention to small programs that can be specified completely.

We specify a program S by means of a precondition and a postcondition in a so-called Hoare triple:

$$\{P\}\ S\ \{Q\}$$

P is called the *precondition*, Q the *postcondition*. The meaning of the Hoare triple is:

> if we execute program S in a state where precondition P holds, execution of S terminates in a state where postcondition Q holds.

As explained in Chapter 3, the conditions P and Q are formulas from predicate logic. In Chapter 4, we present a logical formalism to prove the correctness of Hoare triples. The remainder of these lecture notes, the largest part, is devoted to the question how to do this in practice.

1.2 How to arrive at program correctness?

Now that we have an idea of what program correctness means, we turn to the question how to decide that a program is correct. More specifically, how to decide that a program S satisfies Hoare triple $\{P\}\ S\ \{Q\}$? Some conceivable answers:

Programming and testing. We investigate the specification, invent a program, and test it with test data that satisfy precondition P. There are three possibilities. (1) The program does not terminate within a reasonable time; (2) the program terminates in a state that does not satisfy Q; (3) the program terminates in a state that satisfies Q. In the cases (1) and (2), we can conclude that the program is incorrect, or at least unsatisfactory. It must be modified. What about case (3)?

The program passed *one* test. Are we allowed to conclude that it satisfies the specification? Alas, definitely not. For almost every useful specification, there are many (often infinitely many) states that satisfy the precondition, and it is impossible to test all these initial states. Conclusion: in general, this approach cannot ensure that the program satisfies its specification. Moreover, it also gives no clue how to find a program that does satisfy the specification.

Programming and operational reasoning. In this case, we invent a program and then reason operationally that the program starting in any state that satisfies precondition P, always terminates in a state that satisfies postcondition Q. Operational reasoning means that you play the role of the computer and execute the program yourself. You record the actions of the program, and the effects of these actions on the state of the computer. For many programmers, this is the usual approach, although they know that operational reasoning often fails when the program gets more complicated. Operational reasoning is often restricted to the generic case ('in general, we have ...'). It is then easy to ignore relevant special cases. When one tries to avoid this restriction to the generic case, however, one may end up with a large number of cases that have to be treated separately. A computer might be able to handle it, but we human beings are not good at it. In general, this approach does not guarantee that a program has been found that satisfies the specification.

Programming and verification. Again, we investigate the specification, invent a program, and now we try to *prove* its correctness. In order to do this, we regard the program as a *mathematical object* that can be proved to satisfy its specification by means of logical/mathematical arguments. For this purpose, we use the proof rules given in Chapter 4 below. If we succeed in constructing a correctness proof, the program satisfies its specification. The experience, however, is that this way of proving the correctness of a program usually fails because it leads to the finding of errors, big or small. After correction of the error, we come back to the proof. When we are lucky, this process ends with a verified program. Even then the program obtained in this way is often less satisfactory than the program that can be obtained by the procedure described next.

Concurrent development of the program and its proof of correctness. We investigate the precondition P and the postcondition Q as given in the specification. We then use the available program constructs, the proof rules given for them in Chapter 4, and the heuristics described in Chapter 7 to decompose the specification into smaller ones that are easier to satisfy. This approach is called *program derivation*. It is the approach taken in this course. We do not work with a concrete programming language, but with pseudocode, an abstract form of an imperative programming language (like C or Pascal). It only uses the imperative program constructs of the assignment, the sequential composition (denoted with a semicolon), the conditional construct (if/then/else), and the repetition (while/do). See Chapter 4.

The derivation of correct programs is a manual task which you can learn only by doing it yourself. These notes are therefore full of exercises. Of course, it is also useful (but definitely not sufficient) to read and analyse the examples given in these notes.

We use the proof method of annotated programs, because this enables us to integrate the program and the proof. In this course, we restrict ourselves to the correct design of small well-specified programs. This is only one of the aspects of reliable programming, but it is an aspect that can save you from drowning in a swamp of spaghetti code.

Programmers are constantly making choices, usually based on intuition or experience. We here indicate how a systematic analysis can guide you in your choices. The aim is that analysis and intuition are complementary and strengthen each other. Intuition alone often leads to commands that are not completely correct, and therefore incorrect. Analysis without intuition

can sometimes lead to a command that completely misses the mark, due to a small computation error.

1.3 Overview

The mathematical and logical background material that we need is presented in Chapter 2. Chapter 3 is about specifications. Chapter 4 treats the program constructs and their proof rules. These three chapters are only briefly treated in the course. They form material to be read when the course starts and to be consulted when specific points come up. The course proper consists of the remaining chapters.

The foundations of program correctness are laid in the Chapters 5, 6, and 7. Chapter 5 introduces annotated commands that connect programming with correctness arguments. Chapter 6 introduces a roadmap to derive programs with repetitions in a systematic way. In Chapter 7 we present heuristics to find invariants. This is the main creative challenge in the derivation of while-statements.

Chapter 8 treats search problems, in particular linear search and binary search, because such problems occur in practice very often, and they deserve to be treated carefully.

The harvest is reaped in the Chapters 9 and 10. Here, we finally treat problems for which the methods presented in this course are undoubtably beneficial and perhaps even indispensable. Chapter 9 treats problems like saddleback search and presents the method of the shrinking rectangle. Chapter 10 treats the method of strengthening of the invariant with auxiliary information, and illustrates this with segment problems.

1.4 History of these lecture notes

The Lecture Notes *Program Correctness* were written in Dutch in 1993 by Wim Hesselink. Revised versions were made by Harm Bakker and Hendrik Wietze de Haan (2005). A thorough revision was made in 2007 by Gerard Renardel de Lavalette in collaboration with Wim Hesselink. The present version is a (slightly revised) translation in English by Wim Hesselink.

The original version used the program constructs of the programming language Pascal. We later moved to Modula 3. The present version uses a form of pseudocode close to Modula 3.

We finally list the main sources for the method of programming advocated here:

- E. W. Dijkstra: *A discipline of programming.* Prentice-Hall (Englewood Cliffs), 1976.
- E. W. Dijkstra, W. H. J. Feijen: *Een methode van programmeren.* Academic Service (Den Haag), 1984.
- D. Gries: *The science of programming.* Springer-Verlag (New York, etc.), 1981.
- A. Kaldewaij: *Programming: the derivation of algorithms.* Prentice-Hall International (New York, etc.), 1990.

Chapter 2

Logical-mathematical preparation

In this chapter, we treat the concepts and notations from mathematics and logic that will be used in this course.

2.1 The specification language

We begin with a description of the language used to write specifications. This is predicate logic, with the following components.

Identifiers. These are names for the objects that occur in the specifications. In these notes, we use the following identifiers:

constants: $-\infty, \ldots, -2, -1, 0, 1, 2, \ldots, \infty$, **true**, **false**
variables $i, j, k, m, n, p, q, v, w, x, y, z$
specification constants: M, V, X, Y, Z
arrays: a, b

Strictly speaking, specification constants are logical variables. We call them specification constants because they only occur in the specifications and not in the programs. In particular, they cannot be modified by the programs.

The variables x, y, etc. can occur both free or bound. A variable can be bound by a quantifier ($\forall$, $\exists$) or in the definition of a set (e.g. in $\{x \in \mathbb{N} \mid 6 \leq x < 10\}$; more about this below). According to tradition, the argument of an array is written between square brackets, e.g. $a[4], b[i]$.

Declarations. Identifiers have a *domain*: a set that contains the values the identifier can refer to. In these notes, the following domains occur:

$\mathbb{Z}$, the set of the integers;
$\mathbb{N}$, the natural numbers 0, 1, 2, 3, ...;
$\mathbb{N}_+$, the positive integers 1, 2, 3, ...;
$\mathbb{R}$, the real numbers;
$\mathbb{B}$, the booleans **true** and **false**.

Variables and arrays are declared with an indication of the domain. We use a **var** declaration to indicate that the value of the variable can be modified by the program, and a **const** declaration otherwise. These notes do not treat array modification. All arrays are therefore constant. An example of a declaration:

const $n : \mathbb{N},\ x : \mathbb{R},\ a$: array $[0\,..\,n)$ of $\mathbb{Z}$
var $i : \mathbb{N},\ y : \mathbb{R}$

This declaration indicates that we can use the identifiers n, x, a, i, y, in the specification and the program, and it gives the domains of the identifiers. It further indicates that the program can contain assignments to the variables i and y, but not to n, x, and a.

Functions Primarily, we have the standard functions of arithmetic:

$$\cdot \quad \mathsf{div} \quad \mathsf{mod} \quad \mathsf{min} \quad \mathsf{max} \quad + \quad -$$

Here, the order indicates the priority: $\cdot$ (multiplication) binds strongest, and div and mod bind more strongly than addition and subtraction. Arithmetic rules for div and mod are given later in this chapter. We do use some other functions, such as exponentiation, gcd (greatest common divisor), and the factorial function '!'. Occasionally, we use functions f, g, h, as indicated on the spot.

Predicates We use the following standard predicates:

$$= \quad \neq \quad < \quad \leq \quad \geq \quad >$$

We prefer to use $<$ and $\leq$ rather than $>$ and $\geq$, because the values on the real axis increase from left to right.

Expressions Expressions are formed by combining identifiers, functions, predicates, logical connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), and the quantifiers ($\forall$, $\exists$). We distinguish between numerical expressions and boolean expressions. Boolean expressions are called *conditions*. In logic, they are usually called formulas. Some examples of conditions:

$$\begin{array}{ll} x + 3 > a[0]^2 & X = m \ \mathsf{mod}\ n \\ (n = 2 \wedge m \geq 0) \rightarrow m \cdot n \geq y & i \ \mathsf{mod}\ j = 3 \leftrightarrow k = 5 \\ \forall x(x + y > 7 \rightarrow \exists z(z^2 = 4 < y)) & \end{array}$$

The conditions on the first line have no logical symbols: such conditions are called *atomic*. The conditions on the first two lines have no quantifiers: these are *propositional* conditions. The following expressions are numerical. They are not conditions:

$$x \ \mathsf{mod}\ y + f(1, 3 \cdot x) \qquad a[7] \cdot X \ \mathsf{div}\ n \qquad (x + 1)^3 - 5$$

Note that the second one can only be used in a specification because of the occurrence of the specification constant X.

Occasionally, we use the conditional expression $(C \ ?\ E : F)$, with the meaning: if C then E, else F. Conditional expressions make it often possible to postpone case distinctions. This is useful because early case distinctions tend to increase your work load.

We denote expressions with the letters E, F, possibly with subscript: E_0, F_2 etc. We denote by $[F/x]E$ the result of substitution of expression F for variable x in expression E. Conditions are indicated by the capital letters A, B, C, J, P, Q, R, possibly with subscript.

Sets $\{x \mid P\}$ is the standard notation for the set that contains all objects from the domain of x for which condition P holds. In this notation, x is a bound variable. A set that contains not too many elements can also be denoted by enumerating its elements:

$$\{x \mid 0 \leq x^3 \leq 100\} = \{0, 1, 2, 3, 4\}$$

We can further write $\{E(x) \mid x : P(x)\}$ for the set of values of expression $E(x)$ for all objects x that satisfy $P(x)$: here x is a bound variable. This notation can be written in the standard notation:

$$\{E(x) \mid x : P(x)\} = \{y \mid \exists x(y = E(x) \wedge P(x))\}$$

The empty set is denoted by $\emptyset$. We write $w \in V$ (pronounced 'w in V') for the condition that w is an element of the set V.

An *interval* (alternatively called segment) is a set of consecutive integers, elements of $\mathbb{Z}$. We use the following interval notation:

$$[m\,..\,n] = \{k \mid m \leq k \leq n\}$$
$$[m\,..\,n) = \{k \mid m \leq k < n\}$$

The interval $[m\,..\,n]$ is called *closed*, the interval $[m\,..\,n)$ *half-open*. (We could also define intervals $(m\,..\,n]$ and $(m\,..\,n)$, but we do not need them here.) Working with half-open interval has a number of advantages. For instance, the number of elements of the interval $[m\,..\,n)$ equals $n-m$, provided $m \leq n$ (the interval is empty if $n \leq m$). We also have that $[m\,..\,n) \cup [n\,..\,p) = [m\,..\,p)$ provided $n \in [m\,..\,p)$.

We often use sets to restrict the domain of bound variables:

$\forall x \in V : P$ means the same as $\forall x(x \in V \rightarrow P)$;
$\exists x \in V : P$ means the same as $\exists x(x \in V \wedge P)$;
$\{x \in V \mid P\}$ means the same as $\{x \mid x \in V \wedge P\}$.

The colon in $\forall x \in V : P$, $\exists x \in V : P$ is added to improve readability.

2.2 Strength of formulas and proof format

We use the 'meta-arrow' $\Rightarrow$ for *universally valid implication*. More concretely:

> $P \Rightarrow Q$ means that $P \rightarrow Q$ holds for all assignments of values to the free variables that occur in P and Q.

When $P \Rightarrow Q$ holds, we say that P is *stronger* than Q and that Q is *weaker* than P.

We shall use relation $\Rightarrow$ frequently in expressions and derivations, as well as its reversal $\Leftarrow$. There is also a 'double meta-arrow': $P \Leftrightarrow Q$, which means ($P \Rightarrow Q$ and $Q \Rightarrow P$). We prefer, however, to use the equivalence symbol $\equiv$ with the same meaning as $\Leftrightarrow$, which we phrase as follows:

> $P \equiv Q$ means that P and Q have the same truth value for every assignment of values to the free variables in P and Q.

Our preference for the symbool $\equiv$ is based on the argument that it is more similar to the ordinary equality symbol $=$, and that this is also the way we use the symbol $\equiv$ in proofs. If we know that $P \equiv Q$, then we can replace P by Q in all other expressions without changing the meaning. The symbol $\Leftrightarrow$, however, suggests to prove $P \Leftrightarrow Q$ by separately proving $P \Rightarrow Q$ and $Q \Rightarrow P$. This of course is a valid way to prove it, but it is not always the most convenient way.

We use the following format to prove assertions like $P \Rightarrow Q$:

$$\begin{array}{ll} & P \\ \Rightarrow & \{ \text{ argument why } P \text{ implies } R \text{ } \} \\ & R \\ \Rightarrow & \{ \text{ argument why } R \text{ implies } Q \text{ } \} \\ & Q \end{array}$$

This is called a *linear annotated proof*. It consists of a sequence of formulas on lines, separated by lines that contain a transitive relation (here $\Rightarrow$) and an argument between braces { and }. Annotated linear proofs have the advantage that they are clear and easy to check. The first examples occur in Chapter 6.4.

2.3 Operators and their properties

We discuss a number of properties of the operators div, mod, #, max, min, Max, Min, Σ, Π, $\forall$, and $\exists$. We restrict ourselves to properties that are relevant for programming.

2.3.1 Division with remainder: div and mod

Integer numbers (the elements of $\mathbb{Z}$) play an important role in programming. We often need *division with remainder*, because this enables us to remain working in $\mathbb{Z}$. Example: 38 divided by 5 equals 7 with remainder 3. To express this, the operators div and mod are defined as follows: if $x, y \in \mathbb{Z}$ and $y > 0$, then x div y and x mod y are integer numbers that satisfy

$$x = y \cdot (x \text{ div } y) + x \text{ mod } y \ \wedge \ 0 \leq x \text{ mod } y < y \tag{2.1}$$

In words: the remainder x mod y is a natural number smaller than y, and the integral quotient x div y is such that x equals y times the integral quotient plus the remainder. It follows that: 38 div $5 = 7$ and 38 mod $5 = 3$. You may note that, for negative values of x, these definitions of div and mod deviate from the definitions used in the programming languages C and Java.

The integral quotient x div y is the greatest integer $\leq x/y$. This implies two properties of inequalities in which div plays a role, which we shall use later. Firstly:

$$z \leq x \text{ div } y \ \equiv \ z \cdot y \leq x \tag{2.2}$$

We derive the second property from the first one, using that $\neg(m \leq n)$ is equivalent with $n < m$, and that $\neg p \equiv \neg q$ follows from $p \equiv q$:

$$x \text{ div } y < z \ \equiv \ x < z \cdot y \tag{2.3}$$

2.3.2 The counting operator

If V is a finite set, we write $\#V$ to denote the number of elements of V. The number of elements of the empty set is zero: $\#\emptyset = 0$. This is the starting point for many computations. Two other important rules are the one-point rule, and the splitting rule:

$$\begin{array}{lcl} \#\{a\} & = & 1 \\ \#V & = & \#(V \cap W) + \#(V \setminus W) \end{array}$$

In the splitting rule, the set W is the splitting criterion. Recall that $V \cap W$ is the intersection $\{v \in V \mid v \in W\}$, and $V \setminus W$ is the set difference $\{x \in V \mid x \notin W\}$. If, e.g., we want to count all participants to some event, we may count all paying participants, and all nonpaying participants, and then add these two numbers to get the number of all participants.

Occasionally, we combine the one-point rule with the rule for the empty set to

$$\#\{x \in V \mid x = a\} = \text{ord}(a \in V)$$

Here, the function ord : $\mathbb{B} \to \mathbb{Z}$ is defined by $\text{ord}(b) = (b \,?\, 1 : 0)$.

2.3.3 Maxima and minima of sets of numbers

If V is a finite set of numbers, we write Max V for the greatest element of V and Min V for the least element of V. If V is nonempty, we have

$$\begin{array}{lcl} \text{Max } V = a & \equiv & a \in V \ \wedge \ (\forall x \in V : x \leq a) \\ \text{Min } V = a & \equiv & a \in V \ \wedge \ (\forall x \in V : a \leq x) \end{array}$$

We use the infix operators max and min for the maximum and minimum of pairs. We therefore have $x \max y = \mathsf{Max}\ \{x, y\}$ and $x \min y = \mathsf{Min}\ \{x, y\}$. The one-point rules are

$$\begin{array}{lcl} \mathsf{Max}\ \{a\} & = & a \\ \mathsf{Min}\ \{a\} & = & a \end{array}$$

We define the maximum and the minimum of the empty set by

$$\begin{array}{lcl} \mathsf{Max}\ \emptyset & = & -\infty \\ \mathsf{Min}\ \emptyset & = & \infty \end{array}$$

These (perhaps unexpected) rules are useful, because the following splitting rules then hold without exceptions:

$$\begin{array}{lcl} \mathsf{Max}\ V & = & \mathsf{Max}\ (V \cap W) \max \mathsf{Max}\ (V \setminus W) \\ \mathsf{Min}\ V & = & \mathsf{Min}\ (V \cap W) \min \mathsf{Min}\ (V \setminus W) \end{array}$$

2.3.4 Sums and products of sequences of numbers

When we take the maximum of a sequence of numbers, it does not matter whether a certain number occurs one or more times in the sequence. This does matter, however, when we take the sum or the product of a sequence of numbers. For this reason, sums and products are usually taken for sequences of numbers, and not for sets.

A *sequence* $(f(i) \mid i \in V)$ consists of the subsequent values $f(i)$ for $i \in V$. Here the set V is usually a set of consecutive integers. The sequence differs from the *set* $\{f(i) \mid i \in V\}$ in the possibility of multiple occurrences of elements. We thus have

$$\begin{array}{lcllcl} (n^2 \mid n \in [-2, 3]) & = & (4, 1, 0, 1, 4, 9) & \neq & (0, 1, 1, 4, 4, 9) \\ \{n^2 \mid n \in [-2, 3]\} & = & \{4, 1, 0, 1, 4, 9\} & = & \{0, 1, 4, 9\} \end{array}$$

Just as with sets, we often write $(f(i) \mid i : P(i))$ instead of $(f(i) \mid i \in \{x \mid P(x)\})$.

We define the sum $\Sigma(f(i) \mid i \in V)$ of the sequence inductively. The sum of the empty sequence is 0, the identity element for addition. This is the zero-point rule. The other two rules are the one-point rule and the splitting rule:

$$\begin{array}{lcl} \Sigma(f(i) \mid i \in \emptyset) & = & 0 \\ \Sigma(f(i) \mid i \in \{a\}) & = & f(a) \\ \Sigma(f(i) \mid i \in V) & = & \Sigma(f(i) \mid i \in V \cap W) + \Sigma(f(i) \mid i \in V \setminus W) \end{array}$$

We define the product $\Pi(f(i) \mid i \in V)$ of the sequence in the same way. The product of the empty sequence is 1, the identity element for multiplication. The three rules are:

$$\begin{array}{lcl} \Pi(f(i) \mid i \in \emptyset) & = & 1 \\ \Pi(f(i) \mid i \in \{a\}) & = & f(a) \\ \Pi(f(i) \mid i \in V) & = & \Pi(f(i) \mid i \in V \cap W) \cdot \Pi(f(i) \mid i \in V \setminus W) \end{array}$$

In mathematics, one usually writes $\Sigma_{i \in V} f(i)$ and $\Pi_{i \in V} f(i)$ instead of $\Sigma(f(i) \mid i \in V)$ and $\Pi(f(i) \mid i \in V)$. For programming, our notation is preferable, because we often have to modify the set V rather than argue about the terms $f(i)$.

One can also consider maxima and minima of sequences, but these satisfy the same rules as the maxima and minima of sets.

2.3.5 Distribution

Multiplication distributes over sums:

$$\Sigma(x \cdot f(i) \mid i \in V) \quad = \quad x \cdot \Sigma(f(i) \mid i \in V)$$

Addition distributes over maxima and minima:

$$\begin{array}{lcl} \mathsf{Max}\ (x + f(i) \mid i \in V) & = & x + \mathsf{Max}\ (f(i) \mid i \in V) \\ \mathsf{Min}\ (x + f(i) \mid i \in V) & = & x + \mathsf{Min}\ (f(i) \mid i \in V) \end{array}$$

Multiplication with a positive number x also distributes over (nonempty) maxima and minima, while multiplication with a negative number interchanges maxima and minima:

$$\begin{array}{lcll} x \geq 0 \wedge V \neq \emptyset & \Rightarrow & \mathsf{Max}\ (x \cdot f(i) \mid i \in V) & = & x \cdot \mathsf{Max}\ (f(i) \mid i \in V) \\ & \wedge & \mathsf{Min}\ (x \cdot f(i) \mid i \in V) & = & x \cdot \mathsf{Min}\ (f(i) \mid i \in V)\ , \\ x \leq 0 \wedge V \neq \emptyset & \Rightarrow & \mathsf{Max}\ (x \cdot f(i) \mid i \in V) & = & x \cdot \mathsf{Min}\ (f(i) \mid i \in V) \\ & \wedge & \mathsf{Min}\ (x \cdot f(i) \mid i \in V) & = & x \cdot \mathsf{Max}\ (f(i) \mid i \in V) \end{array}$$

Using this with $x = -1$, one can prove that

$$\mathsf{Max}\ (x - f(i) \mid i \in V) = x - \mathsf{Min}\ (f(i) \mid i \in V)$$

2.3.6 The logical quantifiers

The logical quantifiers $\forall$ and $\exists$ behave as operators on sets. In fact, we can regard $\forall$ and $\exists$ as repeated conjunction ($\wedge$) and repeated disjunction ($\vee$), respectively. The neutral elements of $\wedge$ and $\vee$ are **true** and **false**, respectively. In the following rules, $A(x)$ is a condition and V and W are sets. The zero-point rules, the one-point rules, and the splitting rules are:

$$\begin{array}{lcl} (\forall x \in \emptyset : A(x)) & \equiv & \textbf{true} \\ (\exists x \in \emptyset : A(x)) & \equiv & \textbf{false} \\ (\forall x \in \{a\} : A(x)) & \equiv & A(a) \\ (\exists x \in \{a\} : A(x)) & \equiv & A(a) \\ (\forall x \in V : A(x)) & \equiv & (\forall x \in V \cap W : A(x)) \wedge (\forall x \in V \setminus W : A(x)) \\ (\exists x \in V : A(x)) & \equiv & (\exists x \in V \cap W : A(x)) \vee (\exists x \in V \setminus W : A(x)) \end{array}$$

Disjunction with a condition C that does not depend on x distributes over $\forall$. Similarly, conjunction with C distributes over $\exists$:

$$\begin{array}{lcl} (\forall x \in V : C \vee A(x)) & \equiv & C \ \vee \ (\forall x \in V : A(x)) \\ (\exists x \in V : C \wedge A(x)) & \equiv & C \ \wedge \ (\exists x \in V : A(x)) \end{array}$$

Chapter 3

Specifications

This chapter is about specifications of programs. We use conditions to describe the program state, as given by the values of the program variables. The aim of the program is usually expressed in terms of modifications of the state. We specify the (allowed or intended) changes of the state by using the *precondition* to describe the state before execution, and the *postcondition* for the state after execution.

3.1 States

The starting point for program specification is the fact that execution of the program modifies the values stored in the memory locations associated with the program. We do not regard the values stored on the level of bits or bytes, but rather on the more abstract level of numbers or truth values. We define the *state* of a program as follows:

Definition 3.1 (state) *Let a program S be given, with a declaration of the variables $x_0, \ldots, x_{n-1}$. Then a* state *of the program is a set of pairs*

$$\{(x_0, m_0), \ldots, (x_{n-1}, m_{n-1})\}$$

where, for $i \in [0\,..\,n)$, the value m_i is an element of the domain of the variable x_i. The state thus indicates the value m_i of each program variable x_i. The set of all states of a program is called the state space *of the program.*

Example. If S is a program with the variables $x, y : \mathbb{Z}$, then e.g. $\{(x, 5), (y, 3)\}$ is a state, just as $\{(x, -15), (y, 100)\}$. When we fix an order of the variables (e.g. x before y), we can alternatively denote the state of S by a pair of numbers: $\langle 5, 3\rangle$ or $\langle -15, 100\rangle$. Then the states of S are elements of the state space $\mathbb{Z}^2 = \{\langle m, n\rangle \mid m, n \in \mathbb{Z}\}$.

Remark. In these notes, we ignore the fact that most programming languages cannot represent all integer or real values. In particular, we ignore problems with overflow and underflow, although this is an important issue for program correctness. Therefore, in these notes, $\mathbb{Z}$ and $\mathbb{R}$ are indeed the mathematical sets of integers and reals, respectively.

3.2 Programs and commands

Large programs are built from small programs. When we want to verify that a large program satisfies its specification, we need to know the specifications satisfied by the smaller programs that it contains, and the way these specifications interact. The smaller programs hardly deserve the name program because they perform so little. Yet they need to be specified. We therefore introduce the neutral term *command* for any program or component of a program that can be specified. We reserve the term 'program' for larger commands that can be discussed in isolation (we could also call these 'algorithms').

3.3 Hoare triples

In specifications of commands (or programs), we describe how execution of the command modifies or can modify the state. We do this not on the level of single states, because that would introduce unnecessary details and would be infeasible in cases with infinitely many relevant states. We therefore work with sets of states, i.e., subsets of the state space, or equivalently the conditions on the state space that determine such subsets.

Example: when we are going to specify a command that has to determine the square root of a number x, and that subsequently adds the number y to it, we only want to do this in states where the value of x is nonnegative. We therefore introduce the set of states

$$\{\langle m, n\rangle \mid m, n \in \mathbb{Z} \wedge m \geq 0\}$$

which alternatively is described by the condition $x \geq 0$.

To describe a set of states, we thus use declarations and conditions from the specification language. We formally define our specification concept as follows.

Definition 3.2 (specification, precondition, postcondition, Hoare triple) *Let the program variables $x_0, \ldots, x_{n-1}$ be declared. The* specification *of a command S in these variables consists of a* precondition P *and a* postcondition Q*. These conditions are formulas with no other free variables than the variables $x_0, \ldots, x_{n-1}$, and possibly program constants and specification constants. The specification means that, in every state that satisfies P, command S is executable and will terminate in a state that satisfies Q. We denote this specification in the form of a Hoare triple*[1]*:*

$$\{P\}\ S\ \{Q\}$$

We can therefore read $\{P\}\ S\ \{Q\}$ as:

> if we execute command S in a state where precondition P holds, execution of S will terminate in a state where postcondition Q holds.

As we shall see, in many specifications, we use specification constants to transfer information from the precondition to the postcondition. In general, this cannot be done with variables, because the value of a variable can be modified by the command!

Specification constants do not occur in commands. Therefore, when we mention one or more specification constants in the precondition, the command cannot affect the values of these constant(s), so that they have the same meaning in the postcondition.

3.4 Example of a specification

We now show how to formalize an informal specification of a command by means of a Hoare triple. Let us consider the informal specification:

> Given is an array $a[0\,..\,n)$ that contains the value w. The task is to assign to a variable x the smallest index i that satisfies $a[i] = w$.

We begin with a declaration of the relevant identifiers:

const $n : \mathbb{N},\ w : \mathbb{R},\ a$: array $[0\,..\,n)$ of $\mathbb{R}$
var $x : \mathbb{N}$

We can formalize the condition of the first sentence by

$$\exists\, i \in [0\,..\,n) : a[i] = w \tag{3.1}$$

[1] Named after the British computer scientist C.A.R. (Tony) Hoare.

We could take this as the precondition, and then choose the postcondition

$$a[x] = w \ \wedge \ \forall\, i \in [0\,..\,n) : (a[i] = w \to x \leq i)$$

In general, however, we strive to make the postcondition as simple as possible, preferably in the form $x = X$, for a specification constant X. In order to do so, the precondition must indicate the value X that x must obtain. We can do this by taking the precondition

$$X \in [0\,..\,n) \ \wedge \ a[X] = w \ \wedge \ \forall\, i \in [0\,..\,n) : (a[i] = w \to X \leq i)$$

or, shorter, by

$$X = \mathsf{Min}\ \{i \mid 0 \leq i < n \wedge a[i] = w\}$$

The latter definition of X has the additional advantage that we can replace condition (3.1) by $X < \infty$ (in fact, we have $(\mathsf{Min}\ V < \infty) \ \equiv \ (V \neq \emptyset)$ for $V \subseteq \mathbb{Z}$). In this way, we obtain the specification

$$\begin{array}{l} \{P :\ \mathsf{Min}\ \{i \mid 0 \leq i < n \wedge a[i] = w\} = X < \infty\} \\ S \\ \{Q :\ x = X\} \end{array}$$

3.5 Preregular and postregular specifications

We define a Hoare triple $\{P\}\ S\ \{Q\}$ to be *preregular* if the precondition P is independent of the values of the program variables. It can therefore only depend on program constants and specification constants. In this case, the precondition remains valid during execution of the command. It is therefore applicable at all points of the command.

We define a Hoare triple $\{P\}\ S\ \{Q\}$ to be *postregular* if the postcondition Q is a conjunction of equalities of the form $x = X$.

When there are no other considerations, we prefer to use specifications that are both preregular and postregular. We call them *biregular*.

Example. Given a constant $n \in \mathbb{N}$, a command S that assigns to a variable y the value $n\,!$, can be specified in a biregular way by

$$\{n\,! = Y\}\ S\ \{y = Y\}$$

A command T that squares the variable z, can be specified in a postregular way by

$$\{z^2 = Z\}\ T\ \{z = Z\}$$

In this case, a preregular specification is impossible because the initial value of the variable z is needed to determine the final value that is required.

3.6 Unsatisfiable specifications

Some specifications $\{P\}\ S\ \{Q\}$ with given P and Q are *unsatisfiable*, i.e., there is no command S that satisfies it. There can be two reasons for this: *contradiction* or *lack of information*. Consider for an integer variable t the specification

$$\{\mathbf{true}\}\ S\ \{t^2 = 5\}$$

This specification is unsatisfiable by contradiction, because there is no integer number with square 5. The number 9 is a square, but the specification

$$\{X^2 = 9\}\ S\ \{t = X\}$$

is unsatisfiable by lack of information. If we assign 3 to t, the postcondition is not established when $X = -3$. If we assign -3 to t, the postcondition is not established when $X = 3$.

3.7 Exercises

A number of exercises in which you have to specify a command by providing a declaration and a specifying Hoare triple.

Exercise 3.1. Specify a command that, given integers x and y, interchanges the values of these variables.

Exercise 3.2. Specify a command that, given integers x and y, stores in x the maximum of both values, and stores in y the minimum.

Exercise 3.3. Specify a command that, given m, d, x with $d > 0$ and $0 \leq x < m$, subdivides the interval $[0\,..\,m)$ in d intervals of almost-equal lengths, numbered $0, \ldots, d-1$, and determines the number of the interval that contains x. (This comes from image processing, to partition a rectangular image of width m in d vertical strips).

Exercise 3.4. Specify a command that determines a power of 2 that is greater than a given number x.

Exercise 3.5. Specify a command that, given $n \geq 0$ and x, determines the value x^n.

Exercise 3.6. Specify a command that, given $x \geq 0$, determines the integral square root of x, i.e., the greatest integer for which the square does not exceed x. Do not use the square root symbol $\surd$.

Exercise 3.7. Give a definition of the sequence of Fibonacci, and specify a command that, given $n \geq 0$, determines the nth number of the sequence.

Exercise 3.8. Specify a command that computes the sum of the elements of an array a : array $[0\,..\,n)$ of $\mathbb{Z}$.

Exercise 3.9. Specify a command that computes the maximum of the elements of an array a : array $[0\,..\,n)$ of $\mathbb{Z}$.

Exercise 3.10. Specify a command that computes the inner product of two arrays.

Exercise 3.11. Given is a function $f : \mathbb{N} \to \mathbb{Z}$ that can be used in the command. Specify a command that computes the least number n with $f(n) = 0$, given that such a number n exists.

Exercise 3.12. Specify a command that, given an array, computes the length of the longest subarray with positive values.

Exercise 3.13. Specify a command that, given an array, computes the length of the longest subarray for which all elements are different.

Exercise 3.14. Specify a command that sorts a given array in which all elements are different. You may use a specification-array A.

Exercise 3.15. Same as the previous exercise, but now without the assumption that all elements of the array are different.

Exercise 3.16. Specify a command S that assigns to the variable $x : \mathbb{Z}$ a smaller value. Is it possible to do this, in a satisfactory way, with a postregular specification?

Exercise 3.17. Specify a command that for a given array of numbers computes the greatest difference between subsequent elements. Discuss the ambiguities in the previous sentence.

Chapter 4

Program constructs

In the previous chapter we have discussed commands without telling what they are or what they look like. We come to this point here. We define pseudocode, an abstract and very simple programming language that only contains declarations and a few fundamental programming constructs: skip, the assignment, the sequential composition, the if/then/else construct, and the while/do construct. One can regard this pseudocode as the mother of all imperative programming languages.

4.1 Pseudocode

Definition 4.1 (program expression, program condition) *A* program expression *is an expression constructed in the usual way from*

integer numbers (in decimal notation),
program variables,
program constants,
arithmetic operations $+, -, \cdot,$ *div* , *mod* , *min* , *max* ,
predicates $=, \neq, <, \leq, \geq, >,$
propositional connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

A program condition *is a program expression of type* $\mathbb{B}$ *(boolean).*

We now define our pseudocode. We use declarations of program constants and program variables, just as in the specification language. We have the following commands and constructs:

1. **skip** is a command (the *empty* command);
2. $x := E$ is a command (an *assignment*), if x is a program variable and E is a program expression of the same type as x;
3. $S; T$ is a command, if S and T are commands; it is called the *sequential composition* of S and T;
4. **if** B **then** S **else** T **end** is a command, if S and T are commands and B is a program condition; it is called a *conditional* command;
5. **while** B **do** S **end** is a command, if S is a command and B a program condition; it is called a *while command* or *repetition.*

In a conditional command **if** B **then** S **else** T **end**, the condition B is called the *guard*, and the subcommands S and T are called the *branches.* In the repetition **while** B **do** S **end**, the condition B is called the *guard*, and S is called the *loop body* or *body*.

We keep the explanation short, because we assume that you have had some introductory programming course.

1. **skip** is the empty command, it does nothing.
2. $x := E$ evaluates the value of expression E and assigns this to the variable x. The evaluation of E does not change the state, but the state usually changes by the assignment to x.
3. $S; T$ executes S, and subsequently executes T.
4. **if** B **then** S **else** T **end** evaluates the truth value of B. If B is **true**, then S is executed. If B is **false** then T is executed. The evaluation of B does not change the state.
5. **while** B **do** S **end** begins with evaluating the truth value of B. If B is **true**, then S is executed, and B is evaluated once more. This is repeated as long as the evaluation of B yields **true**. The command terminates when the truth value of B is **false**. The evaluations of B do not change the state.

We can define some other program constructs in pseudocode:

$$\begin{array}{lcl} \textbf{if } B \textbf{ then } S \textbf{ end} & = & \textbf{if } B \textbf{ then } S \textbf{ else skip end} \\ \textbf{do } S \textbf{ while } B \textbf{ end} & = & S;\ \textbf{while } B \textbf{ do } S \textbf{ end} \\ \textbf{for } i := 0 \textbf{ to } n \textbf{ do } S \textbf{ end} & = & i := 0;\ \textbf{while } i \leq n \textbf{ do } S;\ i := i + 1 \textbf{ end} \end{array}$$

4.2 Proof rules

Now that we know the commands, we can give rules to reason about the Hoare triples satisfied by commands. We have the following axioms and proof rules:

$$\{P\}\ \textbf{skip}\ \{P\}$$

$$\{[E/x]P\}\ x := E\ \{P\}$$

$$\frac{\{P\}\ S\ \{Q\} \qquad \{Q\}\ T\ \{R\}}{\{P\}\ S; T\ \{R\}}$$

$$\frac{\{P \wedge B\}\ S\ \{Q\} \qquad \{P \wedge \neg B\}\ T\ \{Q\}}{\{P\}\ \textbf{if } B \textbf{ then } S \textbf{ else } T \textbf{ end}\ \{Q\}}$$

$$\frac{J \wedge B \ \Rightarrow\ \mathsf{vf} \geq 0 \qquad \{J \wedge B\}\ S\ \{J\} \qquad \{J \wedge B \wedge \mathsf{vf} = V\}\ S\ \{\mathsf{vf} < V\}}{\{J\}\ \textbf{while } B \textbf{ do } S \textbf{ end}\ \{J \wedge \neg B\}}$$

$$\frac{P \ \Rightarrow\ Q \qquad \{Q\}\ S\ \{R\}}{\{P\}\ S\ \{R\}}$$

$$\frac{\{P\}\ S\ \{Q\} \qquad Q \ \Rightarrow\ R}{\{P\}\ S\ \{R\}}$$

The first two proof rules have no premises, and are therefore called *axioms*: the **skip** axiom and the assignment axiom.

The other rules have to be read as follows. Each of them has a line that separates the premises of the rule from the conclusion of the rule. If the premises (the expressions above the line) hold, then the conclusion (the expression below the line) also holds.

The first three proper proof rules are named after the command in the conclusion. They are named therefore the *composition rule*, the *conditional rule*, and the *repetition rule*. The expression vf (*variant function*) in the repetition rule is of type $\mathbb{Z}$. It is an expression in terms of program

variables, constants, and specification constants. We give the premises in the repetition rule the following names:

$J \wedge B \;\Rightarrow\; \mathsf{vf} \geq 0$	boundedness of vf
$\{J \wedge B\}\ S\ \{J\}$	invariance of J
$\{J \wedge B \wedge \mathsf{vf} = V\}\ S\ \{\mathsf{vf} < V\}$	decreasingness of vf

The last three proof rules contain the 'meta-arrow' $\Rightarrow$ introduced in Chapter 2.2. For conditions P and Q, the universal implication $P \Rightarrow Q$ means that in all states where P holds, Q holds as well. If this is the case, replacing P by Q is called *weakening*, and replacing Q by P is called *strengthening*. Therefore, the last two proof rules are named: *strengthening of the precondition* and *weakening of the postcondition*.

We turn to the justification of the proof rules. Recall from the previous chapter that $\{P\}\ S\ \{Q\}$ means that, if we execute program S in a state where P holds, execution of S terminates in a state where Q holds. We discuss the proof rules one by one.

skip: If we execute **skip** in a state where P holds, nothing happens and P remains valid.

assignment: If we execute the assignment $x := E$, the value of E is computed without modifying the state, and the resulting value is assigned to the variable x. It follows that P holds in the poststate of the assignment, if and only if, in the prestate, the expression P' holds which is obtained as follows. The expression E is evaluated in the prestate, say to value v; now P' is obtained by substituting v for every occurrence of x in P. In other words, the poststate satisfies P if and only if the prestate satisfies $[E/x]P$. Therefore $[E/x]P$ is the weakest precondition for which the assignment establishes postcondition P.

sequential composition: The premises of the rule say that command S transforms every state in which P holds into a state where Q holds, and that command T transforms every state where Q holds into a state where R holds. This justifies the conclusion that the sequential composition $(S;T)$ of S followed by T transforms every state where P holds into a state where R holds.

conditional: The premises of the rule say that command S transforms every state where $P \wedge B$ holds, into a state where Q holds, and that T transform every state where $P \wedge \neg B$ holds, into a state where Q holds. If we now execute **if** B **then** S **else** T **end** in a state where P holds, there are two possibilities. Either B holds in this state and command S is executed, which transforms the initial state into a state where Q holds, or B is false and command T is executed, which transforms the initial state into a state where Q holds. Either way, the resulting state satisfies Q.

repetition: This is the most difficult rule. The first premise $J \wedge B \;\Rightarrow\; \mathsf{vf} \geq 0$ means that in all states where $J \wedge B$ holds, the value of vf is nonnegative. The second premise $\{J \wedge B\}\ S\ \{J\}$ means that command S transforms every state where $J \wedge B$ holds into a state where J holds. The third premise $\{J \wedge B \wedge \mathsf{vf} = V\}\ S\ \{\mathsf{vf} < V\}$ means that S transforms every state where $J \wedge B$ holds and where vf has a value V into a state where vf has a value smaller than V. As the value of specification constant V is unmodified, this means that the value of vf has become smaller.

The conclusion of the rule is about execution of **while** B **do** S **end** in a state that satisfies J. This repetition executes S as long as guard B holds. In other words, every time S is executed, it has the precondition B. Therefore, the second premise implies that, if J holds before the repetition, repeated executions of S preserve validity of J at the point where guard B is evaluated. In particular, J holds when evaluation of B yields **false**. Then the loop terminates with the postcondition $J \wedge \neg B$.

It remain to show that the loop terminates. As the loop body is executed each time with precondition $J \wedge B$, the third premise implies that the value of vf decreases. This value is an integer, and by the first premise it is nonnegative whenever $J \wedge B$ holds, in particular when the loop body is to be executed. As an integer cannot decrease infinitely many times before becoming negative, it follows that the repetition terminates because guard B becomes **false** after a finite number of executions of S.

strengthening of the precondition: The premises are that $P \Rightarrow Q$ holds and that S transforms every state where Q holds into a state where R holds. If we executes S in a state where P holds, this state also satisfies Q because of $P \Rightarrow Q$. Therefore S transforms it into a state where R holds. This proves $\{P\}\ S\ \{R\}$.

weakening of the postcondition: This is proved in the same way as the previous rule.

4.3 Exercises

Exercise 4.1. Above we defined **do** S **while** B **end** using while/do and if/then/else. Show that conversely it is also possible to define **while** B **do** S **end** using do/while and if/then/else.

Exercise 4.2. Define **for** $i := 0$ **to** n **do** S **end** by means of the do/while construct.

Exercise 4.3. How could we construct **skip** if it was not yet in the programming language (there are at least two possibilities), and how can we construct a command that never terminates?

Exercise 4.4. Give proof rules for the program constructs defined at the end of Chapter 4.1: **if** B **then** S **end**, **do** S **while** B **end**, and **for** $i := 0$ **to** n **do** S **end**.

Chapter 5

Annotated commands

In this chapter, we show how to *annotate* commands with conditions. We write conditions in the program text in the same way as comments are added. The meaning is that the program state satisfies the condition when execution of the program arrives at the point of the condition. The condition is therefore the precondition of the next command in the program text, and it is the postcondition of the previous command in the program text. This must be done in accordance with the proof rules for the constructs used in the program. Furthermore, it is possible to add reasoning steps to weaken a condition (or to keep it equivalent). Such steps are justified by means of comment between $(*$ and $*)$. A program annotated according to the annotation rules can be read as a proof of correctness. We give various examples in this chapter.

5.1 Two assignments

We begin with simple commands: assignments.

5.1.1 A first problem

Consider the specification:

$$\{x = X\}\ S\ \{x = 3 \cdot X + 1\}$$

For the sake of readability, we use the following format:

$$\begin{array}{l} \quad \{P : x = X\} \\ S \\ \quad \{Q : x = 3 \cdot X + 1\} \end{array}$$

Note that we have named the precondition P, and the postcondition Q. We shall use this abbreviation mechanism more often in annotations. It is useful, because it enables us in the remainder of the text to use the names P and Q for the (possibly complicated) expressions that they represent.

Which command S can satisfy this specification? We expect an assignment of the form $x := E$ for some term E. The proof rule for an assignment with postcondition Q is:

$$\{[E/x]Q\}\ x := E\ \{Q\}$$

By substituting E for x in $Q : x = 3 \cdot X + 1$, we obtain

$$\{E = 3 \cdot X + 1\}\ x := E\ \{Q\}$$

The precondition obtained in this way is not yet our precondition P. We have to choose a program expression E such that $E = 3 \cdot X + 1$ is implied by P, i.e., by $x = X$. Simple arithmetic shows that P implies the condition $3 \cdot x + 1 = 3 \cdot X + 1$. We can make this condition equal to

$[E/x]Q$, by choosing for E the term $3 \cdot x + 1$. We call this step the *preparation of the assignment* $x := 3 \cdot x + 1$. The resulting annotation is:

$$
\begin{array}{l}
\quad \{P : x = X\} \\
\quad\quad (* \text{ arithmetic in preparation of an assignment to } x *) \\
\quad \{3 \cdot x + 1 = 3 \cdot X + 1\} \\
x := 3 \cdot x + 1 \\
\quad \{Q : x = 3 \cdot X + 1\}
\end{array}
$$

5.1.2 A second problem

Consider now reversal of the previous specification:

$$
\begin{array}{l}
\quad \{P : x = 3 \cdot X + 1\} \\
S \\
\quad \{Q : x = X\}
\end{array}
$$

Again, we expect that S will be an assignment of the form $x := E$. How to prepare it? We need to massage precondition P in such a way that X becomes the righthand side of the equality. Starting with $3 \cdot X + 1$, we therefore have to subtract 1, followed by an integer division by 3. In this way, we can arrive at the annotation

$$
\begin{array}{l}
\quad \{P : x = 3 \cdot X + 1\} \\
\quad\quad (* \text{ arithmetic in preparation of an assignment to } x *) \\
\quad \{(x - 1) \text{ div } 3 = X\} \\
x := (x - 1) \text{ div } 3 \\
\quad \{Q : x = X\}
\end{array}
$$

To summarize:

> In an annotated command, every assignment $x := E$ is enclosed between an explicit precondition P and an explicit postcondition Q, in such a way that P is syntactically equal to the substitution $[E/x]Q$.

> If two conditions, say $\{P\}\{Q\}$, are written after each other without some command between them, then $P \Rightarrow Q$ holds, and the two conditions are separated by a comment that argues why the implication holds (this can be a standard argument like "arithmetic" or it can refer to an annotated linear proof elsewhere).

5.2 Sequential composition

We give an example with two program variables x and y. The specification is

$$
\begin{array}{l}
\quad \{P : x = y^2 + X \ \wedge \ y + 1 = Y\} \\
S \\
\quad \{Q : x = y^2 + X \ \wedge \ y = Y\}
\end{array}
\tag{5.1}
$$

How to implement this? At least we have to modify y. Let us therefore first give y the value required, i.e., establish $y = Y$. It looks as if this can be done with the assignment $y := y + 1$. As the proof rule for this assignment requires that all occurrences of y in the postcondition are replaced by $y + 1$, we also must remove isolated occurrences of y in the other conjunct of P.

After the first assignment more must be done, and eventually x must be modified. In this way, we may obtain the annotated command:

$\{x = y^2 + X \ \wedge \ y + 1 = Y\}$
$(*$ arithmetic in preparation of $y := y + 1 *)$
$\{x = (y + 1 - 1)^2 + X \ \wedge \ y + 1 = Y\}$
$y := y + 1$;
$\{x = (y - 1)^2 + X \ \wedge \ y = Y\}$
$(*$ arithmetic in preparation of $x := \ldots *)$
$\{x = y^2 - 2 \cdot y + 1 + X \ \wedge \ y = Y\}$
$(*$ arithmetic $*)$
$\{x + 2 \cdot y - 1 = y^2 + X \ \wedge \ y = Y\}$
$x := x + 2 \cdot y - 1$
$\{x = y^2 + X \ \wedge \ y = Y\}$

This is an implementation of specification (5.1) because the composition rule in Chapter 4.2 allows us to ignore the intermediate conditions. Look carefully to see that we indeed apply the composition rule here!

5.3 Conditional commands

5.3.1 A first problem

Consider the specification:

$\{P : x \text{ max } y = X \wedge x \text{ min } y = Y\}$
S
$\{Q : x = X \wedge y = Y\}$

We observe that nothing needs to be done if $x \geq y$. If $x < y$, we have to interchange the values of x and y. Interchanging can be done with an auxiliary variable, say z. We use a conditional command to make the correct choice (between doing nothing or interchanging). This leads to the following annotated implementation of the specification:

$\{P : x \text{ max } y = X \wedge x \text{ min } y = Y\}$
if $x < y$ **then**
$\{P \wedge x < y\}$
$(*$ definition of P and arithmetic $*)$
$\{y = X \wedge x = Y\}$
$z := x$;
$\{y = X \wedge z = Y\}$
$x := y$;
$\{x = X \wedge z = Y\}$
$y := z$
$\{Q : x = X \wedge y = Y\}$
else
$\{P \wedge y \leq x\}$
$(*$ definition P and arithmetic $*)$
$\{Q : x = X \wedge y = Y\}$
skip
$\{Q\}$
end $(*$ collect the branches $*)$
$\{\, Q \,\}$

To ascertain the correctness of this annotated command, we need to verify the following items:

- the annotations of the three assignments satisfy the assignment rule;
- at the two steps of arithmetic the implications holds, i.e.

  $$P \wedge x < y \;\Rightarrow\; y = X \wedge x = Y$$

  and

  $$P \wedge y \leq x \;\Rightarrow\; x = X \wedge y = Y$$

- the conditions after **then** and **else** are correct;
- both branches terminate with $\{Q\}$.

5.3.2 A second problem

Determine a command S that satisfies

$$\begin{array}{l} \{P:\ x = X\} \\ S \\ \{Q:\ x > 0 \ \wedge\ (x = X + 1 \vee x = 2 - X)\} \end{array} \tag{5.2}$$

Using distribution of $\wedge$ over $\vee$, we first observe that

$$Q \;\equiv\; (x > 0 \ \wedge\ x = X + 1) \ \vee\ (x > 0 \ \wedge\ x = 2 - X)$$

We can treat the two disjuncts of Q separately, if we are prepared to make additional assumptions. Indeed we have

$$\begin{array}{l} \{x + 1 > 0 \ \wedge\ x + 1 = X + 1\} \\ x := x + 1 \\ \{x > 0 \ \wedge\ x = X + 1\} \end{array}$$

and

$$\begin{array}{l} \{2 - x > 0 \ \wedge\ 2 - x = 2 - X\} \\ x := 2 - x \\ \{x > 0 \ \wedge\ x = 2 - X\} \end{array}$$

The conjuncts $x+1 = X+1$ and $2-x = 2-X$ follow from P. To form a conditional command, we require a guard B such that $B \;\Rightarrow\; x + 1 > 0$ and $\neg B \;\Rightarrow\; 2 - x > 0$. The guard $B:\ x \geq 0$ suffices for this purpose. We thus get the annotation:

$$\begin{array}{l} \{P:\ x = X\} \\ \textbf{if } x \geq 0 \textbf{ then} \\ \quad \{x = X \ \wedge\ x \geq 0\} \\ \quad\quad (*\ \text{arithmetic}\ *) \\ \quad \{x + 1 > 0 \ \wedge\ x + 1 = X + 1\} \\ \quad x := x + 1 \\ \quad \{x > 0 \ \wedge\ x = X + 1\} \\ \quad\quad (*\ \text{logic}\ *) \\ \quad \{Q\} \\ \textbf{else} \\ \quad \{x = X \ \wedge\ x < 0\} \\ \quad\quad (*\ \text{arithmetic}\ *) \\ \quad \{2 - x > 0 \ \wedge\ 2 - x = 2 - X\} \\ \quad x := 2 - x\ ; \\ \quad \{x > 0 \ \wedge\ x = 2 - X\} \\ \quad\quad (*\ \text{logic}\ *) \\ \quad \{Q\} \\ \textbf{end} \quad (*\ \text{collect}\ *) \\ \{Q\} \end{array}$$

Note that the guard $x \geq 0$ can be replaced by $x > 0$ or $x > 1$. The resulting commands are not equivalent, but also satisfy specification (5.2).

We summarize:

> If we have a conditional command **if** B **then** S **else** T **end** with precondition P and postcondition Q, then the **then** branch S has the precondition $P \wedge B$, and the **else** branch T has the precondition $P \wedge \neg B$, and both branches have the postcondition Q.

5.4 Additional rules for annotations

The annotation rules often require more writing that we would like. The following rules sometimes enable us to shorten the text.

Naming conditions on the fly. As we have seen, it is sometimes useful to introduce a name for a condition that occurs several times. In particular, this can apply to the precondition or the postcondition of a conditional command.

Omitting constant conditions. A condition that only contains constants (program constants and specification constants), cannot change during execution of a command. When it holds somewhere in the annotation, it holds everywhere. It is useless to write down such a condition repeatedly. Just mention that it is obviously constant, and you can use it everywhere. This applies in particular to the precondition of a preregular specification (cf. Chapter 3.5).

Simultaneous assignment. We sometimes have a situation that asks for a sequence of assignments that do not influence each other, e.g.,

$$
\begin{array}{l}
x := 0\,; \\
y := y + 1\,; \\
z := 1
\end{array}
$$

A sufficient condition for 'not influencing each other' is: *the variables assigned to are different, and each of them does not occur in the expressions assigned to the other variables of the sequence.* In this case, we can perform the assignments simultaneously, and we can derive the precondition of the sequence from the postcondition by simultaneous substitution. This eliminates some writing. We illustrate this with the above example:

$$
\begin{array}{l}
\quad \{0 \leq 0 < n \ \wedge \ 0 + 1 = y + 1\} \\
x := 0\,; \\
y := y + 1\,; \\
z := 1 \\
\quad \{0 \leq x < n \ \wedge \ x + z = y\}
\end{array}
$$

A conditional command with two different postconditions. With a conditional command, it is sometimes convenient to give the **then** branch and the **else** branch different postconditions, say Q_0 and Q_1. This is allowed when the conditional command itself has the postcondition $Q_0 \vee Q_1$ (verify this by weakening of the postconditions!).

5.5 Exercises

Unless specified otherwise, every exercise in this course that asks for the design of a command that satisfies a specification, includes the task to prove the correctness of your solution. In the exercises below, this should be done by means of a correct annotation.

Unless specified otherwise, we use the declaration: **var** $x, y, i, k, m, n, z : \mathbb{Z}$.

Exercise 5.1. Determine for each of the following Hoare triples a single assignment S that satisfies the specification, and prove this with a correct annotation.

(a) $\{x = 5 \cdot X - 7\}\ S\ \{x = X\}$
(b) $\{x = X\}\ S\ \{x = 5 \cdot X - 7\}$
(c) $\{x = X + 3 \cdot Y - 1\ \wedge\ y = Y\}\ S\ \{x = X\ \wedge\ y = Y\}$
(d) $\{x = X + Y\ \wedge\ y = 2 \cdot X - 7\}\ S\ \{x = X + Y\ \wedge\ y = Y\}$
(e) $\{7 \cdot X - 2 \leq x < 7 \cdot X + 5\}\ S\ \{x = X\}$

Exercise 5.2. Determine an annotated command S with

$$\{x = X\ \wedge\ y = Y\}\ S\ \{x = X \cdot Y\ \wedge\ y = X + Y\}$$

(Hint. Prepare an assignment $y := x + y$; then prepare an assignment to x.)

Exercise 5.3. Interchanging two values. Consider the specification

$$\{y = X\ \wedge\ x = Y\}\ S\ \{x = X\ \wedge\ y = Y\} \tag{5.3}$$

(a) Why is this specification not satisfied by the assignment $S :\ (x := y\ ;\ y := x)$? Here it suffices to give an operational argument.
(b) Determine an annotated command S that satisfies (5.3), and that uses an auxiliary variable $z : \mathbb{Z}$ to temporarily store the value of x.
(c) Determine an annotated command S of the form $x := x + y\ ;\ y := \ldots\ ;\ x := \ldots$ that satisfies (5.3).

Exercise 5.4. Assume that x and y are variables of the type $\mathbb{B}$ (boolean). Prove that specification (5.3) is satisfied by

$$\begin{array}{l} x := (x = y); \\ y := (x = y); \\ x := (x = y) \end{array}$$

Exercise 5.5. (a) Determine an annotated command S that satisfies

$$\{i = X\ \wedge\ k = Y\}\ S\ \{i = X + Y\ \wedge\ k = X\}$$

(b) Determine an annotated command T that satisfies

$$\{i = X\ \wedge\ k = Y\}\ T\ \{i = Y - X\ \wedge\ k = -Y\}$$

Exercise 5.6. Determine an annotated command that satisfies

$$\{x = 3 \cdot X - 2 \cdot Y + 1\ \wedge\ y = X\}\ S\ \{x = X\ \wedge\ y = Y\}$$

(a) You may use an auxiliary variable.
(b) Give a solution in which S consists of two assignments.

Exercise 5.7. Prove the correctness of

$$\begin{array}{l} \quad \{\mathbf{true}\} \\ \mathbf{if}\ x \geq 0\ \mathbf{then}\ y := 0\ \mathbf{else}\ y := -x\ \mathbf{end} \\ \quad \{x + y \geq 0\ \wedge\ y \geq 0\} \end{array}$$

Exercise 5.8. Prove the correctness of

$$
\begin{array}{l}
\quad \{y = 2 \cdot x \vee y = 1 - 2 \cdot x\} \\
\textbf{if } y \bmod 2 = 0 \textbf{ then} \\
\quad x := x - y \\
\textbf{else} \\
\quad x := x + y \\
\textbf{end}; \\
y := y + 1 \\
\quad \{y = 2 \cdot x \vee y = 1 - 2 \cdot x\}
\end{array}
$$

Exercise 5.9. A problem from image processing: partition a rectangular image of width m in d vertical strips of approximately equal width, and determine for a pixel with horizontal coordinate x the number of the strip it is contained in.

Given are natural numbers m and d. We partition the interval $[0\,..\,m)$ in d subintervals. The point $x : \mathbb{Z}$ is then in the subinterval with number k if and only if

$$Q :\ (k \cdot m) \operatorname{div} d \leq x < ((k+1) \cdot m) \operatorname{div} d$$

The number of integers in the subinterval with number k is

$$s(k) = ((k+1) \cdot m) \operatorname{div} d - (k \cdot m) \operatorname{div} d$$

(a) Prove that $|s(k) - \frac{m}{d}| < 1$. Hint: use formula (2.1).
(b) Prove that $Q \ \equiv\ k = (d \cdot (x+1) - 1) \operatorname{div} m$.
Hint: repeatedly use the formulas (2.2), (2.3), as well as $x < y \equiv x + 1 \leq y$.
(c) Determine an assignment S that satisfies

$$
\begin{array}{l}
\textbf{const } m, d, x : \mathbb{Z} \\
\quad \{P :\ d > 0 \ \wedge\ 0 \leq x < m\ \} \\
S \\
\quad \{Q\}
\end{array}
$$

Exercise 5.10. Determine an annotated command S that satisfies

$$\{x = X\}\ S\ \{x \geq 1 \ \wedge\ (x = X \vee x = 1 - X)\}$$

Exercise 5.11. Determine an annotated command S that satisfies

$$\{x + y = X \wedge x \cdot y = Y\}\ S\ \{x \leq y \wedge x + y = X \wedge x \cdot y = Y\}$$

Exercise 5.12. Determine an annotated command S that satisfies

$$
\begin{array}{l}
\quad \{i = X \ \wedge\ k = Y \ \wedge\ X \geq Y\} \\
S \\
\quad \{k \geq 0 \ \wedge\ ((i = X + Y \wedge k = X) \vee (i = Y - X \wedge k = -Y))\}
\end{array}
$$

Compare with Exercise 5.5.

Exercise 5.13. (a) Prove the validity for every $a \in \mathbb{Z}$ of the Hoare triple

$$\{x + y = Z\}\ x := x + a\ ;\ y := y - a\ \{x + y = Z\}$$

(b) Determine an annotated command S that satisfies

$$\{x + y = Z\}\ S\ \{x + y = Z \wedge x \cdot y < 0\}$$

Exercise 5.14. Determine an annotated command S that satisfies

$$\begin{array}{l} \quad \{X > Y \ \wedge \ ((x = X \wedge y = -2 \cdot X + Y) \ \vee \ (x = Y \wedge y = X - 2 \cdot Y))\} \\ S \\ \quad \{x = X \ \wedge \ y = Y\} \end{array}$$

Exercise 5.15. (a) Determine an annotated command S that satisfies

$$\begin{array}{l} \textbf{const } n : \mathbb{Z} \\ \quad \{x = X\} \\ S \\ \quad \{Q : x \geq n \ \wedge \ (x = 2 \cdot X - n \vee x = 3 \cdot n - 2 \cdot X + 1)\} \end{array}$$

(b) Prove the validity of

$$x = 2 \cdot X - n \ \Rightarrow \ (x + n) \bmod 2 = 0 \text{ and}$$
$$x = 3 \cdot n - 2 \cdot X + 1 \ \Rightarrow \ (x + n) \bmod 2 = 1$$

(c) Determine an annotated command T (the *inverse* of S), that satisfies

$$\{Q\} \ T \ \{x = X\}$$

Exercise 5.16. (a) Determine an annotated command S that satisfies

$$\{x = X\} \ S \ \{x > 0 \ \wedge \ (x = 2 \cdot X + 1 \vee x = -2 \cdot X)\}$$

(b) Determine an annotated command T that satisfies the reverse specification

$$\{x > 0 \ \wedge \ (x = 2 \cdot X + 1 \vee x = -2 \cdot X)\} \ T \ \{x = X\}$$

Exercise 5.17. Consider for some guard B the specified command

$$\begin{array}{l} \quad \{P : \ X \geq 0 \ \wedge \ (i = 2 \cdot X - 3 \ \vee \ 4 \cdot X + i = 0)\} \\ \textbf{if} \quad B \quad \textbf{then} \\ \quad i := (i + 3) \bmod 2 \\ \textbf{else} \\ \quad i := (-i) \bmod 4 \\ \textbf{end} \\ \quad \{Q : \ i = X\} \end{array}$$

(a) Extend the annotation as far as possible, and use this to derive sufficient conditions for guard B to ensure correctness of the command.
(b) Show that the guard $B : i \geq -3$ is wrong. Use the condition derived to construct a counterexample.
(c) Choose some correct guard B and prove its correctness.

Chapter 6

Repetitions

We turn to the derivation of correct repetitions. This is more difficult than commands without repetitions, because a repetition is a repeated sequential composition of the loop body with itself, and we do not yet know intermediate conditions. In order to reason successfully about a repetition, we use an *invariant*: a condition that can serve repeatedly as the intermediate condition. In other words, the invariant is a condition that remains valid by execution of the loop body. *Finding* a suitable invariant is often a difficult part of the derivation of a correct command: more about this in the next chapter.

We begin by recalling the proof rule for repetitions from Chapter 4, in adapted form.

$$\frac{J \wedge B \;\Rightarrow\; \mathsf{vf} \geq 0 \qquad \{J \wedge B \wedge \mathsf{vf} = V\}\; S\; \{J \wedge \mathsf{vf} < V\}}{\{J\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end}\ \{J \wedge \neg B\}}$$

Here we have combined the last two premises of the proof rule, because this often makes the task of annotating the loop body less cumbersome.

In this chapter, we give a roadmap for the derivation of repetitions, followed by some examples. The roadmap serves to structure the search for components that satisfy the premises of the repetition rule. It does this by going slightly beyond the rule itself. In order to have a general precondition P and a general postcondition Q, it allows an initialization command before the repetition, and it allows weakening of $J \wedge \neg B$ to Q.

Before proceeding, we note the relationship between the time complexity (the number of computation steps) of a repetition and the variant function:

> Let vf be a variant function for **while** B **do** S **end**, and let $K \in \mathbb{N}$. If $\mathsf{vf} < K$ holds before execution of the repetition, the loop body S is executed at most K times (because when S is executed K times, vf becomes negative).

6.1 Roadmap for the design of a repetition

In general, the design of a repetition goes as follows. The starting point is the specification

$$\{P\}\ T\ \{Q\}$$

The problem is to implement this, and to prove the correctness of the implementation obtained. For a repetition, we proceed as follows.

Step 0: choice for repetition. In view of the specification, we may decide to use a repetition.

Step 1: choice of invariant and guard, finalization. We massage postcondition Q in such a way that we can determine a suitable invariant J and a suitable guard B for which we can prove *finalization*:

$$J \wedge \neg B \;\Rightarrow\; Q$$

If we then can apply the repetition rule, the postcondition Q is within reach.

Step 2: initialization. In general, the precondition P does not imply the invariant J. We usually need to determine an *initialization command* T_0, for which we can prove

$$\{P\}\ T_0\ \{J\}$$

Usually, this initialization command consists of a number of assignments, which can be found by analyzing how precondition P (or a condition implied by P) can be obtained by substitution from the invariant J.

Step 3: variant function. We choose a suitable *integer-valued* function vf and prove the boundedness of vf:

$$J \wedge B \;\Rightarrow\; \mathsf{vf} \geq 0$$

Usually we have some idea how to reach the postcondition by repeatedly doing steps in the right direction. The function vf is a measure for the number of steps needed before the goal (the postcondition) is reached. A suitable vf is nonnegative as long as the goal has not yet been reached, and it should decrease by every step in the right direction.

Step 4: body of the repetition. We determine a command S for which we can prove the invariance of J and the decrease of vf. We often do these things simultaneously and derive:

$$\{J \wedge B \wedge \mathsf{vf} = V\}\ S\ \{J \wedge \mathsf{vf} < V\}$$

A rule of thumb for finding a suitable loop body S is that the body should make vf smaller and yet preserve the invariant J.

Step 5: conclusion. When the steps 1, 2, 3, and 4 have been taken successfully, we may conclude:

$\{P\}$
T_0 ;
{ J because of step 2: initialization }
($*$ definition of vf (step 3) $*$)
while B **do**
S
end
{ $J \wedge \neg B$, because of step 4 and the proof rule for the repetition }
($*$ by finalization in step 1 $*$)
$\{Q\}$

This proves that the composition T_0 ; **while** B **do** S **end** is a correct implementation of the specification $\{P\}\ T\ \{Q\}$.

To summarize, the steps 1, 2, 3, 4 each have their own proof obligations. The steps 1 and 3 have proof obligations ($J \wedge \neg\, B \Rightarrow Q$ and $J \wedge B \Rightarrow \mathrm{vf} \geq 0$, respectively) that can be met by annotated linear proofs, but sometimes an informal argument is acceptable. The steps 2 and 4 require an annotated command. Occasionally, step 2 does not need a command if the precondition P happens to imply the invariant J (e.g., see Chapter 6.4).

Step 5, the conclusion, may look superfluous because it can be derived mechanically from the preceding steps. It is important to add it, however, because it is the point where you can use your programmer's intuition to see that your formal activities have not gone astray.

We illustrate the roadmap by three examples: exponentiation, powers of 2, and Euclid's algorithm.

6.2 Example: exponentiation

Suppose we need to compute the nth power of a real number x for a natural number n. We can specify this by:

$$\begin{array}{l} \textbf{var}\ x, y : \mathbb{R},\ n : \mathbb{Z} \\ \quad \{P :\ n \geq 0\ \wedge\ x^n = Y\} \\ S \\ \quad \{Q : y = Y\} \end{array} \tag{6.1}$$

Note the declaration $n : \mathbb{Z}$, while the declaration $n : \mathbb{N}$ is more natural. We declare $n : \mathbb{Z}$ because we want to have the freedom to use the assignment $n := n - 1$ without having to worry whether this is allowed (i.e. $n > 0$). We introduce specification constant Y for the value we want to assign to y. If exponentiation is available in the programming language, the assignment $y := x^n$ would be the obvious solution. We therefore assume that exponentiation is not available.

Yet, this is an easy programming exercise. The purpose for treating it here, is to illustrate the roadmap. Easy examples are needed to learn the method. More difficult examples will follow, in later chapters.

Step 0: choice for repetition. If there is no exponentiation available, the postcondition can not be reached in a fixed number of steps. We therefore choose a repetition.

Step 1: choice of invariant and guard, finalization. To find a suitable invariant, we look at the postcondition $y = Y$. Its righthand side also occurs in the precondition, we therefore leave it unchanged. As exponentiation is repeated multiplication, we expect multiplications. First, a very modest multiplication: the postcondition is equivalent with $y \cdot 1 = Y$. Looking at the precondition $x^n = Y$, we see that $y \cdot x^n = Y$ can be established from the precondition by putting $y := 1$. We therefore choose the invariant

$$J :\ n \geq 0\ \wedge\ y \cdot x^n = Y$$

Which guard B do we need? The loop can terminate when $x^n = 1$, because then J implies $y = Y$. Of course, we cannot use x^n in the guard, because this would require (repeated) computation of x^n (which is the problem we need to solve!). As $x^0 = 1$ for all x (also for $x = 0$), however, the loop can also terminate when $n = 0$. Indeed, we have $J \wedge n = 0\ \Rightarrow\ y = Y$. We therefore choose the guard

$$B :\ n \neq 0$$

The proof obligation $J \wedge \neg B \Rightarrow Q$ is fulfilled because $y \cdot x^0 = y$.

Step 2: initialization. As anticipated above, we can initialize with $y := 1$:

$$
\begin{array}{l}
\quad \{P : n \geq 0 \ \wedge \ x^n = Y\} \\
\quad\quad (* \text{ arithmetic in preparation of assignment to } y *) \\
\quad \{n \geq 0 \ \wedge \ 1 \cdot x^n = Y\} \\
y := 1 \\
\quad \{J : \ n \geq 0 \ \wedge \ y \cdot x^n = Y\}
\end{array}
$$

Step 3: variant function. Function vf should be a suitable measure for the number of steps needed to reach the postcondition. This suggests the choice $\mathsf{vf} = n$. The proof obligation $J \wedge B \Rightarrow \mathsf{vf} \geq 0$ now directly follows from the invariant $J : \ n \geq 0 \ \wedge \ y \cdot x^n = Y$.

Step 4: body of the repetition. We need a command S that satisfies

$$\{J \wedge B \wedge \mathsf{vf} = V\} \ S \ \{J \wedge \mathsf{vf} < V\}$$

In order to decrease $\mathsf{vf} = n$, it is natural to use $n := n - 1$. In order to restore the invariant J we have to multiply y with x. This is done in the following annotated command:

$$
\begin{array}{l}
\quad \{J \ \wedge \ B \ \wedge \ \mathsf{vf} = V\} \\
\quad\quad (* \text{ definitions of } J, B, \mathsf{vf} *) \\
\quad \{n \geq 0 \ \wedge \ y \cdot x^n = Y \ \wedge \ n \neq 0 \ \wedge \ n = V\} \\
\quad\quad (* \text{ arithmetic, preparing the assignment } n := n - 1 *) \\
\quad \{n - 1 \geq 0 \ \wedge \ y \cdot x \cdot x^{n-1} = Y \ \wedge \ n - 1 < V\} \\
y := y \cdot x \ ; \\
\quad \{n - 1 \geq 0 \ \wedge \ y \cdot x^{n-1} = Y \ \wedge \ n - 1 < V\} \\
n := n - 1 \ ; \\
\quad \{n \geq 0 \ \wedge \ y \cdot x^n = Y \ \wedge \ n < V\} \\
\quad\quad (* \text{ definitions of } J \text{ and } \mathsf{vf} *) \\
\quad \{J \ \wedge \ \mathsf{vf} < V\}
\end{array}
$$

Step 5: conclusion. We summarize the result:

$$
\begin{array}{l}
\quad \{P : \ n \geq 0 \ \wedge \ x^n = Y\} \\
y := 1 \ ; \\
\quad \{J : \ n \geq 0 \ \wedge \ y \cdot x^n = Y\} \\
\quad\quad (* \ \mathsf{vf} : n *) \\
\textbf{while } n \neq 0 \textbf{ do} \\
\quad y := y \cdot x \ ; \\
\quad n := n - 1 \ ; \\
\textbf{end} \\
\quad \{Q : \ y = Y\}
\end{array}
$$

In this summary, we only repeat the precondition, postcondition, invariant, and variant function. We do not repeat the detailed annotations of the steps 2 and 4, or the proofs given in the steps 1 and 3.

Remark. Before going to the next example, we have a closer look at the properties of exponentiation used in the above derivation:

$$
\begin{array}{rcl}
 & & x^0 = 1 \\
n > 0 & \Rightarrow & x^n = x \cdot x^{n-1}
\end{array}
$$

This is an example of a *recurrence relation* of a function (in this case exponentiation). It gives one or more intial values, and describes how function values can be computed from function values for smaller arguments. In these notes, we shall encounter several recurrence relations in derivations of repetitions.

6.3 Example: powers of 2

Let us design a command T that determines the least power of 2 greater than a given number $x > 0$ without using exponentiation. We specify this as follows.

const $x : \mathbb{Z}$
  $\{P : \ x > 0\}$
var $i, y : \mathbb{Z}$
T;
  $\{Q : \ x < y \leq 2 \cdot x \ \wedge \ y = 2^i\}$

The specification is preregular: P only depends on constants. We can therefore omit the condition $x > 0$ from the annotation, and use it whenever needed. We now follow the roadmap.

Step 0: choice for repetition. As we cannot use exponentiation, we expect to establish the postcondition by repeated multiplication of y by 2.

Step 1: choice of invariant and guard, finalization. We take a part of the postcondition Q as the invariant J:

$$J : \ y \leq 2 \cdot x \ \wedge \ y = 2^i$$

The choice for the guard B is easy: the negation of $x < y$, the remaining part of postcondition Q. We thus choose

$$B : \ y \leq x$$

The proof obligation $J \wedge \neg B \ \Rightarrow \ Q$ is easily fulfilled, because $J \wedge \neg B$ is clearly equivalent to Q.

Step 2: initialization. It is natural to start with 2^0. So we choose $i := 0$ and $y := 1$.

  $\{P : \ x > 0\}$
  $\{1 \leq 2 \cdot x \ \wedge \ 1 = 2^0\}$
$i := 0$;
  $\{1 \leq 2 \cdot x \ \wedge \ 1 = 2^i\}$
$y := 1$;
  $\{J : y \leq 2 \cdot x \ \wedge \ y = 2^i\}$

This annotation is constructed from bottom to top, because the invariant gives us more indication how to proceed than the precondition.

Step 3: variant function. Because of the definition of B, we are done when $x - y < 0$. We therefore decide to decrease $x - y$ in every step. We thus choose $\mathsf{vf} = x - y$. The proof obligation $J \wedge B \Rightarrow \mathsf{vf} \geq 0$ follows from

$$B \ \equiv \ x - y \geq 0$$

Step 4: body of the repetition. To decrease vf, we have to increase y because x is constant. The invariant implies $y = 2^i$. We therefore increase y by multiplying it by 2. These considera-

tions lead to the annotated body:

$$
\begin{array}{l}
\quad \{J \ \wedge \ B \ \wedge \ \mathsf{vf} = V\} \\
\quad\quad (* \text{ definitions of } J,\ B,\ \mathsf{vf} *) \\
\quad \{y \leq 2 \cdot x \ \wedge \ y = 2^i \ \wedge \ y \leq x \ \wedge \ x - y = V\} \\
\quad\quad (* \text{ arithmetic; and } y > 0 \text{ because } y = 2^i *) \\
\quad \{2 \cdot y \leq 2 \cdot x \ \wedge \ 2 \cdot y = 2^{i+1} \ \wedge \ x - 2 \cdot y < V\} \\
y := 2 \cdot y \ ; \\
\quad \{y \leq 2 \cdot x \ \wedge \ y = 2^{i+1} \ \wedge \ x - y < V\} \\
i := i + 1 \\
\quad \{y \leq 2 \cdot x \ \wedge \ y = 2^i \ \wedge \ x - y < V\} \\
\quad\quad (* \text{ definitions of } J \text{ and } \mathsf{vf} *) \\
\quad \{J \wedge \mathsf{vf} < V\}
\end{array}
$$

Step 5: conclusion.

$$
\begin{array}{l}
\quad \{P : \ x > 0\} \\
i := 0 \ ; \\
y := 1 \ ; \\
\quad \{J : y \leq 2 \cdot x \ \wedge \ y = 2^i\} \\
\quad\quad (* \ \mathsf{vf} : \ x - y *) \\
\textbf{while } x \geq y \textbf{ do} \\
\quad y := 2 \cdot y \ ; \\
\quad i := i + 1 \\
\textbf{end} \\
\quad \{Q : \ x < y \leq 2 \cdot x \ \wedge \ y = 2^i\}
\end{array}
$$

Note that in this case vf is negative after execution of the loop.

6.4 Example: Euclid's algorithm

We want to compute the greatest common divisor $\gcd(x, y)$ of the positive integers x and y, and assign it to x. We specify the command T by

$$
\begin{array}{l}
\textbf{var } x, y : \mathbb{Z} \\
\quad \{P : x > 0 \ \wedge \ y > 0 \ \wedge \ \gcd(x, y) = Z\} \\
T \\
\quad \{Q : x = Z\}
\end{array}
\tag{6.2}
$$

The Greek mathematician Euclid (around 300 BC) designed an algorithm for this that is still in use. It is based on the idea: if you need $\gcd(x, y)$, replace the arguments x and y by smaller numbers with the same gcd; repeat this until one of the arguments is 0; then the other argument is the gcd.

We first derive a recurrence relation for gcd, which we need for the algorithm. Every number $x > 0$ is the greatest common divisor of itself and 0, because it is a divisor of itself ($x \cdot 1 = x$) and of 0 ($x \cdot 0 = 0$), and the greatest one because $x > 0$. We thus have $x > 0 \ \Rightarrow \ \gcd(x, 0) = x$, the base of the recurrence relation. We shall use this in the last step of the algorithm: if one of the arguments is 0, we are done.

As long as the arguments of gcd differ from 0, we want to make them smaller. For this purpose, we use formula (2.1) from Chapter 2.3.1:

$$y > 0 \ \Rightarrow \ x = (x \ \mathsf{div} \ y) \cdot y + x \ \mathsf{mod} \ y \ \wedge \ 0 \leq x \ \mathsf{mod} \ y < y$$

For $y > 0$, we have that every common divisor of x and y is a common divisor of y and $x \bmod y$, and vice versa. This is proved in

$$
\begin{array}{ll}
 & z \text{ divides } x \text{ and } y \\
\equiv & \{\ y > 0 \text{ and (2.1)}\ \} \\
 & z \text{ divides } y \text{ and } (x \operatorname{div} y) \cdot y + x \bmod y \\
\equiv & \{\ \text{if } z \text{ divides } y, \text{ then } (z \text{ divides } i \cdot y + j) \equiv (z \text{ divides } j)\ \} \\
 & z \text{ divides } y \text{ and } x \bmod y
\end{array}
$$

This is an annotated linear proof with the relation $\equiv$. Note the braces around the arguments. The conditions are without braces, because this is not an annotated program.

It follows that $\gcd(x, y) = \gcd(y, x \bmod y)$ provided $y > 0$ (why?). We now have the following recurrence relation:

$$
\begin{array}{lcl}
x > 0 & \Rightarrow & \gcd(x, 0) = x \\
y > 0 & \Rightarrow & \gcd(x, y) = \gcd(y, x \bmod y)
\end{array}
\tag{6.3}
$$

After this preparation, we turn to the roadmap.

Step 0: choice for repetition. We choose a repetition because we expect a command that decreases the arguments of gcd in steps.

Step 1: choice of invariant and guard, finalization. Because of the first formula of (6.3), we allow $y = 0$ in the invariant. We therefore define the invariant J and the guard B by

$$
\begin{array}{l}
J:\ x > 0 \ \wedge\ y \geq 0 \ \wedge\ \gcd(x, y) = Z \\
B:\ y \neq 0
\end{array}
$$

Finalization is proved in

$$
\begin{array}{ll}
 & J \wedge \neg B:\ x > 0 \ \wedge\ y \geq 0 \ \wedge\ \gcd(x, y) = Z \ \wedge\ y = 0 \\
\Rightarrow & \{\ \text{substitute } y = 0, \text{ omit superfluous conjuncts}\ \} \\
 & x > 0 \ \wedge\ \gcd(x, 0) = Z \\
\Rightarrow & \{\ \text{first formula of (6.3)}\ \} \\
 & Q:\ x = Z
\end{array}
$$

Step 2: initialization. In this case, the precondition P happens to imply the invariant J. Initialization is therefore superfluous: the command **skip** suffices.

Step 3: variant function. We choose the variant function $\mathrm{vf} = y$. Then we have $J \ \wedge\ B \ \Rightarrow\ \mathrm{vf} \geq 0$ because J has a conjunct $y \geq 0$.

Step 4: body of the repetition. In the body of the repetition, we use an auxiliary variable $m : \mathbb{Z}$ to store the value of $x \bmod y$. The annotated loop body becomes:

$\{J \ \wedge \ B \ \wedge \ \mathsf{vf} = V\}$
$\{x > 0 \ \wedge \ y \geq 0 \ \wedge \ \gcd(x, y) = Z \ \wedge \ y \neq 0 \ \wedge \ y = V\}$
$(*$ second formula of (6.3), arithmetic $*)$
$\{y > 0 \ \wedge \ \gcd(y, x \bmod y) = Z \ \wedge \ 0 \leq x \bmod y < y = V\}$
$m := x \bmod y$;
$\{y > 0 \ \wedge \ \gcd(y, m) = Z \ \wedge \ 0 \leq m < y = V\}$
$(*$ arithmetic, omitting conjuncts $*)$
$\{y > 0 \ \wedge \ m \geq 0 \ \wedge \ \gcd(y, m) = Z \ \wedge \ m < V\}$
$x := y$;
$\{x > 0 \ \wedge \ m \geq 0 \ \wedge \ \gcd(x, m) = Z \ \wedge \ m < V\}$
$y := m$;
$\{x > 0 \ \wedge \ y \geq 0 \ \wedge \ \gcd(x, y) = Z \ \wedge \ y < V\}$
$\{J \ \wedge \ \mathsf{vf} < V\}$

Step 5: conclusion.

$\{P : \ x > 0 \ \wedge \ y > 0 \ \wedge \ \gcd(x, y) = Z\}$
$\{J : \ x > 0 \ \wedge \ y \geq 0 \ \wedge \ \gcd(x, y) = Z\}$
$(* \quad \mathsf{vf} : \ y \quad *)$
while $y \neq 0$ **do**
  var $m : \mathbb{Z}$;
  $m := x \bmod y$;
  $x := y$;
  $y := m$;
end;
$\{ Q : \ x = Z \}$

6.5 Active finalization

Initialization (step 2) of a loop usually requires a command T_0 that satisfies $\{P\} \ T_0 \ \{J\}$. Finalization (part of step 1) usually only requires a proof of $J \wedge \neg B \ \Rightarrow \ Q$. There are cases, however, in which finalization also needs a command. This occurs when the repetition establishes a condition close but unequal to the postcondition Q. More precisely, the implication $J \wedge \neg B \ \Rightarrow \ Q$ is not valid, but there is a simple command T_1 that satisfies $\{J \wedge \neg B\} \ T_1 \ \{Q\}$. In this case, you can establish postcondition Q by inserting command T_1 after the repetition. We then speak of *active finalization*.

6.6 Exercises

Exercise 6.1. Design a more efficient command T' for exponentiation according to specification (6.1) of Chapter 6.2. Choose the same J, B, and vf. Verify and use that, if n is even, the invariant is also preserved by $x := x \cdot x \ ; \ n := n \ \mathsf{div} \ 2$.

Compare the efficiency of command T' with the efficiency of the solution in Chapter 6.2. For this purpose, determine the number of assignments needed for the computation of 2^{1000} for both programs.

Exercise 6.2. Multiplication can be specified by

var $a, b, x : \mathbb{Z}$
$\{P : a \cdot b = X \ \wedge \ b \geq 0\}$
T
$\{Q : x = X\}$

Design a command for this specification that only uses addition, multiplication by 2, and division by 2 with remainder. Use the invariant $a \cdot b + x = X \ \wedge \ b \geq 0$.

Exercise 6.3. Function $f : \mathbb{N} \to \mathbb{Z}$ is defined by the recurrence relation:

$$\begin{array}{rcl} & & f(0) = 0 \\ & & f(1) = 1 \\ n \geq 1 & \Rightarrow & f(n+1) = f(n-1) + f(n) \end{array}$$

It is known as the sequence of Fibonacci. Derive a command that for given number $k \geq 0$ computes the value of $f(k)$. Use integer variables $n, x, y : \mathbb{Z}$ and the invariant

$$J : \ 0 \leq n \leq k \ \wedge \ x = f(n) \ \wedge \ y = f(n+1)$$

Exercise 6.4. Investigate what is wrong with the following annotation.

$\{J : \textbf{true}\}$
$(* \ \mathsf{vf} = \frac{1}{n} \ *)$
while $n > 0$ **do**
  $\{\textbf{true} \ \wedge \ n > 0 \ \wedge \ \frac{1}{n} = V\}$
    $(*$ arithmetic $*)$
  $\{\textbf{true} \ \wedge \ \frac{1}{n+1} < V\}$
$n := n + 1$
  $\{\textbf{true} \ \wedge \ \frac{1}{n} < V\}$
end
$\{n \leq 0\}$

Exercise 6.5. The function f is given by the recurrence relation

$$\begin{array}{rcl} y \leq 0 & \Rightarrow & f(y,z) \ = \ z \\ y > 0 & \Rightarrow & f(y,z) \ = \ 10 \cdot f(y \ \mathsf{div} \ 10, z) + y \ \mathsf{mod} \ 10 \end{array}$$

Design a command that satisfies

var $y, z : \mathbb{Z}$
  $\{ \ P : \ f(y,z) = Z \ \}$
S
  $\{ \ Q : \ z = Z \ \}$

Here you can use active finalization. Use variables m and n with the invariant

$$J : \ m \cdot f(y,z) + n = Z$$

Exercise 6.6. The function g is defined by the recurrence relation

$$\begin{array}{rcl} & & g(0) = 0 \\ n > 0 & \Rightarrow & g(n) = g(n \ \mathsf{div} \ 10) + n \ \mathsf{mod} \ 10 \end{array}$$

Derive a command S that satisfies

var $n, x : \mathbb{Z}$
  $\{P : \ n \geq 0 \ \wedge \ g(n) = Z\}$
S
  $\{Q : \ x = Z \ \}$

For this purpose, use a repetition with the invariant $J : \ n \geq 0 \ \wedge \ g(n) + x = Z$.

Exercise 6.7. The function h is defined by the recurrence relation

$$\begin{array}{rcl} & & h(0) = 0 \\ n > 0 & \Rightarrow & h(n) = 5 \cdot h(n \text{ div } 3) + n \bmod 4 \end{array}$$

Derive a command S with

$$\begin{array}{l} \textbf{var } n, x : \mathbb{Z} \\ \quad \{P :\ n \geq 0\ \wedge\ h(n) = Z\} \\ S \\ \quad \{Q :\ x = Z\} \end{array}$$

Use a repetition with an auxiliary variable $y : \mathbb{Z}$, with the invariant

$$J :\ n \geq 0\ \wedge\ y \cdot h(n) + x = Z$$

Exercise 6.8. Assume that someone follows the roadmap to find a repetition T that satisfies $\{P\}\ T\ \{Q\}$, and, in step 1, chooses J and B in such a way that $J \Rightarrow B$. Why is his effort doomed to fail? Which step of the roadmap must fail?

Chapter 7

Finding the invariant

In this chapter, we treat an essential aspect of correct programming: the search for a suitable invariant in the derivation of a repetition. We begin with some considerations concerning invariants and repetitions. Next we discuss several heuristics (rules of thumb) that can help to find suitable invariants. We finally give some applications of these heuristics.

7.1 The role of the invariant

What is the role of the invariant in the derivation of a correct command? The starting point is the observation that an invariant is indispensable for reasoning about repetitions, because the proof rule for repetitions needs an invariant. This is also clear from the roadmap in Chapter 6.1: starting with the postcondition, one invents an invariant and a guard, one then tries to initialize the invariant, one invents a variant function, and one tries to find a loop body that preserves the invariant while decreasing the variant function.

> You can often find a suitable invariant by taking a condition, weaker than the postcondition, which can easily be established from the precondition. Therefore, search for a condition J, in between P and Q, and a guard B such that Q follows from $J \wedge \neg B$.

Below we give several heuristics (rules of thumb), that can be helpful. We illustrate the heuristics by means of the examples in the previous chapter.

7.2 Heuristics for the search for an invariant

Isolating a conjunct. If the postcondition is of the form $Q_0 \wedge Q_1$, you can try to take the invariant $J = Q_0$ and the guard $B = \neg Q_1$, or the other way around. Take for J a subcondition of Q that is easy to *establish*, and for B a subcondition that is easy to *test*. Sometimes you first have to massage the postcondition into a conjunction, e.g. by using that $E < F$ is equivalent with $E \leq F \ \wedge \ E \neq F$.

Replacing an expression by a variable. If the postcondition Q contains an expression E, you can introduce a new program variable i and form a condition J by replacing some occurrences of E in Q by i. In this case, we have

$$J \ \wedge \ i = E \ \Rightarrow \ Q$$

We can therefore use the guard $B : i \neq E$. It is often useful to add to J an additional conjunct that expresses the range of variable i.

This heuristic has two special cases that are worth mentioning:

Replacing a constant by a variable. If the postcondition Q contains a constant n, e.g. 0 or 1, or a program constant n, we can use the above heuristic with n in the role of E. We thus introduce a variable i and form the invariant J by replacing some occurrences of n in Q by i. We then have $J \wedge i = n \Rightarrow Q$. Therefore, we can use the guard $B : i \neq n$. It is often useful to add to J a conjunct that expresses the range for variable i.

Splitting a variable. If k is a variable that occurs several times in postcondition Q, you can try to form an invariant J by replacing some occurrences of k in Q by a new program variable i. Again, we have $J \wedge i = k \Rightarrow Q$, and we can use the guard $B : i \neq k$. It is often useful to add to J a conjunct that expresses the range for variable i.

Generalization. If you have a precondition $P : E = X$ and a postcondition $Q : x = X$, it is often possible to get a suitable invariant J by generalizing the expression E in the precondition, say to an expression F. Then the guard B should satisfy $\neg B \Rightarrow F = x$. Two examples:

$J : x + E = X$ (with a guard B such that $\neg B \Rightarrow E = 0$);
$J : x \cdot E = X$ (with a guard B such that $\neg B \Rightarrow E = 1$).

This heuristic can be regarded as *replacing a constant by a variable in the precondition*, when we rewrite the precondition to $P : X = 0 + E$ or to $P : X = 1 \cdot E$.

7.3 Examples of choice of the invariant

Let us first have a look at the choices of invariants in the examples of the previous chapter. We list the precondition, the invariant, the guard, and the postcondition.

Exponentiation: the precondition, invariant, guard, and postcondition are

$P : x^n = Y$
$J : y \cdot x^n = Y$
$B : n \neq 0$
$Q : y = Y$

This is an example of the heuristic *generalization* with the expressions $E = x^n$ and $F = y \cdot x^n$.

Powers of 2: the precondition, invariant, guard, and postcondition are

$P : 0 \leq A \wedge x < 2^A$
$J : y = 2^i \wedge 0 \leq i \leq A$
$B : y \leq x$
$Q : x < y \wedge y = 2^i \wedge 0 \leq i \leq A$

This is an example of the heuristic *isolating a conjunct*: we form J by removing $x < y$ from Q, and we use its negation as the guard. The operational idea could be that the part J of the postcondition is easy to establish, that B is easy to test, and that J is relatively easy to preserve in the loop body.

Euclid's algorithm: the precondition, invariant, guard, and postcondition are

$P : x > 0 \wedge y > 0 \wedge \mathsf{gcd}(x, y) = Z$
$J : x > 0 \wedge y \geq 0 \wedge \mathsf{gcd}(x, y) = Z$
$B : y \neq 0$
$Q : x = Z$

What we have done here, is a form of *generalization*. We have first rewritten the postcondition and strengthened it slightly to $x > 0 \ \wedge \ \gcd(x, 0) = Z$. We have then replaced the constant 0 by the variable y. We finally added a lower bound for y, which allows $y = 0$, but does not contradict the precondition.

To summarize, the application of the heuristics can be helpful in the search for a suitable invariant. Indiscriminate application, however, is impossible. You have to choose: which conjunct? which constant, variable, or expression, which generalization? You need experience to make suitable choices.

We proceed with yet another example.

7.4 The sum of an array of numbers

Let us compute the sum of an array of numbers. We specify this by

$$\begin{array}{l} \textbf{const } n : \mathbb{N}, a : \text{array } [0\,..\,n) \text{ of } \mathbb{R} \\ \textbf{var } x : \mathbb{R} \\ \quad \{P : \ \textbf{true}\} \\ T \\ \quad \{Q : \ x = \Sigma\ (a[i] \mid i : i \in [0\,..\,n))\} \end{array} \tag{7.1}$$

We follow the roadmap to derive command T. First, however, we derive some properties of sums that we need. For the sake of conciseness and clarity, we introduce the abbreviation

$$A(k) = \Sigma\ (a[i] \mid i : i \in [0\,..\,k))$$

We observe that $A(k)$ is defined for k with $0 \leq k \leq n$. The postcondition can now be rewritten to $Q : \ x = A(n)$. We thus have to compute $A(n)$. This is easy in the case that $n = 0$ because $A(0) = 0$ (the sum of the empty sequence). Usually, however, n is larger than 0, in which case $A(n)$ can be computed in steps. For this purpose, we derive a recurrence relation. If $0 \leq k < n$, then

$$\begin{array}{ll} & A(k+1) \\ = & \quad \{ \text{ definition } \} \\ & \Sigma\ (a[i] \mid i : i \in [0\,..\,k+1)) \\ = & \quad \{ \text{ splitting: } i = k \textbf{ or } i < k \text{ ; note that } k \in [0\,..\,k+1) \text{ ; } \\ & \qquad a[k] \text{ is defined because } 0 \leq k < n \ \} \\ & a[k] + \Sigma\ (a[i] \mid i : i \in [0\,..\,k)) \\ = & \quad \{ \text{ definition } \} \\ & a[k] + A(k) \end{array}$$

We thus have the following recurrence relation for A:

$$\begin{array}{lll} & & A(0) = 0 \\ 0 \leq k < n & \Rightarrow & A(k+1) = a[k] + A(k) \end{array} \tag{7.2}$$

We return to the roadmap.

Step 0. We choose a repetition because we expect to establish the postcondition by repeated addition.

Step 1. We use the heuristic *replacing a constant by a variable* to find an invariant. We choose to replace the constant n in the postcondition Q by the variable k. We add a range condition $0 \leq k \leq n$ to ensure that $A(k)$ is well-defined. More concretely, we take

$$J: \ 0 \leq k \leq n \ \wedge \ x = A(k)$$
$$B: \ k \neq n$$

We have $J \wedge \neg B \ \Rightarrow \ Q$ because

$$\begin{array}{ll} & J \ \wedge \ \neg B \\ \equiv & \{ \text{ definitions of } J \text{ and } B, \text{ compute } \neg B \ \} \\ & 0 \leq k \leq n \ \wedge \ x = A(k) \ \wedge \ k = n \\ \Rightarrow & \{ \text{ substitute } k = n \text{ and omit conjuncts } \} \\ & x = A(n) \\ \equiv & \{ \text{ definitions of } Q \text{ and } A \ \} \\ & Q \end{array}$$

(Note that, in this proof, we **cannot** replace the implication ($\Rightarrow$) by an equivalence ($\equiv$), because $x = A(n)$ does not say anything about k.)

Step 2. We have the following annotated initialization:

$$\begin{array}{l} \{P: \ \textbf{true}\} \\ \quad (* \ n \in \mathbb{N}, \text{ and first formula of (7.2)} \ *) \\ \quad \{0 \leq 0 \leq n \ \wedge \ 0 = A(0)\} \\ x := 0; \\ \quad \{0 \leq 0 \leq n \ \wedge \ x = A(0)\} \\ k := 0; \\ \quad \{J: \ 0 \leq k \leq n \ \wedge \ x = A(k)\} \end{array}$$

Reading from top to bottom, this annotation may look unexpected. In fact, we designed it by arguing from bottom (J) to top (P). We choose $k := 0$ because $A(0)$ is the only directly available value of A.

Step 3. As k is initially 0 and the postcondition contains the conjunct $k = n$, we expect to increase k, but not beyond n. We therefore choose the variant function

$$\mathsf{vf} = n - k$$

The invariant J implies $\mathsf{vf} \geq 0$, because we included the range condition in J.

Step 4. We obtain the following annotated loop body:

$$\begin{array}{l} \{J \ \wedge \ B \ \wedge \ \mathsf{vf} = V\} \\ \quad (* \text{ definitions of } J, B, \mathsf{vf} \ *) \\ \quad \{0 \leq k \leq n \ \wedge \ x = A(k) \ \wedge \ k \neq n \ \wedge \ n - k = V\} \\ \quad (* \text{ preparation of } k := k + 1; \text{ use the second formula of (7.2), and } 0 \leq k < n \ *) \\ \quad \{0 \leq k < n \ \wedge \ a[k] + x = A(k+1) \ \wedge \ n - k = V\} \\ x := a[k] + x \ ; \\ \quad \{0 \leq k < n \ \wedge \ x = A(k+1) \ \wedge \ n - k = V\} \\ \quad (* \text{ more preparation of } k := k + 1 \ *) \\ \quad \{0 \leq k + 1 \leq n \ \wedge \ x = A(k+1) \ \wedge \ n - (k+1) < V\} \\ k := k + 1 \ ; \\ \quad \{0 \leq k \leq n \ \wedge \ x = A(k) \ \wedge \ n - k < V\} \\ \quad (* \text{ definitions of } J \text{ and } \mathsf{vf} \ *) \\ \quad \{J \ \wedge \ \mathsf{vf} < V\} \end{array}$$

Step 5. Conclusion.

const $n : \mathbb{N}, a :$ array $[0\,..\,n)$ of $\mathbb{R}$
var $x : \mathbb{R}$
  $\{P :$ **true**$\}$
$x := 0$;
$k := 0$
  $\{J : \ 0 \leq k \leq n \ \wedge \ x = A(k)\}$
    $(*$ vf $= n - k *)$
while $k \neq n$ **do**
  $x := a[k] + x$;
  $k := k + 1$;
end
  $\{Q : \ x = \Sigma\ (a[i] \mid i : i \in [0\,..\,n))\}$

Note that the variable i is bound in the expression $\Sigma\ (a[i] \mid i : i \in [0\,..\,n))$. It is not a program variable! It would be very confusing to introduce a program variable i.

7.5 Exercises

Exercise 7.1. Derive a command to compute the second and third power of a number n, when the only allowed operations are addition, and multiplication by 2 and 3. The specification is

const $n : \mathbb{N}$
var $x, y : \mathbb{Z}$
  $\{P :$ **true**$\}$
T;
  $\{Q : \ x = n^2 \ \wedge \ y = n^3\}$

Use the heuristic *replacing a constant by a variable* to find J and B. More specifically, replace the constant n by a variable $k : \mathbb{N}$. Define vf $= n - k$. Note that, because of the invariant, the values of $(k+1)^3$ and $(k+1)^2$ can be expressed in x, y, and k:

$$J \ \Rightarrow \ (k+1)^3 = y + 3 \cdot x + 3 \cdot k + 1$$
$$J \ \Rightarrow \ (k+1)^2 = x + 2 \cdot k + 1$$

Exercise 7.2. The function factorial '!' is defined by $0! = 1$, and $n! = n \cdot (n-1)!$ for integer $n > 0$. Derive a command T that satisfies

var $x, n : \mathbb{Z}$
  $\{P : \ n \geq 0 \ \wedge \ n! = X\}$
T
  $\{Q : \ x = X \ \}$

Hint: recall the heuristic *generalization*, and compare this exercise with the example Exponentiation in Chapter 6.2.

Exercise 7.3. Computing the integral square root. Consider the specification

const $x : \mathbb{N}$
var $y : \mathbb{Z}$
  $\{P :$ **true**$\}$
T
  $\{Q : \ y \geq 0 \ \wedge \ y^2 \leq x \ \wedge \ x < (y+1)^2\}$

Derive T as follows. Use the heuristic *isolating a conjunct* to find a suitable invariant J and guard B. Use the initialization $y := 0$, the variant function $x - y$, and the body $y := y + 1$.

Exercise 7.4. A more efficient command for the computation of the integral square root according to the specification of the previous exercise.
(a) Determine an invariant and a guard with the heuristic *replacing expression by variable*, applied to the expression $y+1$ and a new variable z. Include a conjunct $y \leq z$ in the invariant. Initialize with $y := 0$ and $z := x+1$. Choose a suitable variant function, and use a body of the form

var $m : \mathbb{Z}$;
$m := (y + z)$ div 2 ;
if $m \cdot m \leq x$ **then** ? **else** ? **end**

(b) Determine the number of assignments, executed by this command in the case of $x = 1000$. Compare this with the number of assignments performed by the command from the previous exercise.

Exercise 7.5. Integral division with remainder of an integer x by a positive integer y can be specified by

const $y : \mathbb{N}_+$
var $x, q : \mathbb{Z}$
$\{P : x = X \ \wedge \ X \geq 0\}$
T
$\{Q : X = q \cdot y + x \ \wedge \ 0 \leq x < y\}$

Determine a command T that satisfies this specification and that does *not* use multiplication, div, and mod. Apply the heuristic of *isolating a conjunct* to find J and B.

Exercise 7.6. A variation on the previous exercise. Next to additions and subtractions, you may now use multiplications and divisions of the form $2 \cdot p$ and p div 2. Determine a command T_1 that satisfies

const $y : \mathbb{N}_+$
var $x, z, q, i : \mathbb{Z}$;
$\{P : x = X \ \wedge \ X \geq 0\}$
T_1
$\{Q : X = q \cdot y + x \ \wedge \ 0 \leq x < y\}$

Use the invariant

$$J : \ X = q \cdot z + x \ \wedge \ 0 \leq x < z \ \wedge \ i \geq 0 \ \wedge \ z = 2^i \cdot y$$

The initialization of this invariant requires a command T_0, similar to the repetition of Chapter 6.3. Command T_1 thus consists of two repetitions. Take the guard $z \neq y$ for the second repetition. Argue that the variable i with all its assignments can be removed from the final command [1].

Exercise 7.7. Compare the efficiency of the commands T and T_1 of the two previous exercises by determining the numbers of assignments executed in either case, expressed in the final value of q, say for $q = 37$.

Exercise 7.8. The function $f : \mathbb{N} \to \mathbb{Z}$ is defined[2] by

$$\begin{array}{lcl} f(0) & = & 0 \\ f(1) & = & 1 \\ f(2 \cdot n) & = & f(n) \\ f(2 \cdot n + 1) & = & f(n) + f(n+1) \end{array}$$

[1] Such a variable is called a ghost variable.
[2] This function was invented under the name fusc by the Dutch computer scientist Edsger W. Dijkstra.

Determine a command S that satisfies

$$
\begin{array}{l}
\textbf{var } n, x : \mathbb{Z} \\
\quad \{n \geq 0 \ \wedge \ f(n) = Z\} \\
S \\
\quad \{Q : \ x = Z\}
\end{array}
$$

Use the conjunct $y \cdot f(n) + x \cdot f(n+1) = Z$ in the invariant.

Exercise 7.9. This exercise is about the program derived in Chapter 7.4.
(a) Show that the order of the assignments in the body of the repetition cannot be reversed because, in general, the Hoare triple

$$\{J \ \wedge \ B\} \ k := k + 1; \ x := a[k] + x \ \{J\}$$

is not valid.
(b) Does the program remain correct, when guard B is replaced by $k < n$? If so, adapt the proof. If not, why?
(c) Same question, with guard B replaced by $k \leq n$.
(d) Is it possible to adapt the proof of the program to the invariant

$$J : \ 0 < k \leq n \ \wedge \ x = A(k) \ ?$$

(e) Same question, for the invariant $J : \ 0 \leq k < n \ \wedge \ x = A(k)$.

Exercise 7.10. Consider the specification

$$
\begin{array}{l}
\textbf{const } n \in \mathbb{N}, a : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{Z} \\
\textbf{var } x : \mathbb{Z} \\
\quad \{P : \ \textbf{true}\} \\
T \\
\quad \{Q : x = \mathsf{Max}\ \{a[i] \mid i : i \in [0\,..\,n)\}\ \}
\end{array}
$$

(a) Determine a command T that satisfies this specification, using the ideas of Chapter 7.4. Use the operator max and the value $-\infty$.
(b) Strengthen the precondition to $P_1 : \ n > 0$. Now determine a command T_1 that satisfies the specification and that *does not* use max and $-\infty$.

Exercise 7.11. Design a program to compute the *product* of n elements of an array. For this purpose, in the specification of Chapter 7.4, replace the operator Σ by Π. Give the formal specification. Modify the program T of Chapter 7.4 in such a way that it satisfies the new specification and adapt the proof.

Exercise 7.12. We specify a command T to determine the inner product of two arrays by

$$
\begin{array}{l}
\textbf{const } n \in \mathbb{N}, \ a, b : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{R} \\
\textbf{var } x : \mathbb{R} \\
\quad \{P : \ \textbf{true}\} \\
T \\
\quad \{Q : x = \Sigma\ (a[i] \cdot b[i] \mid i : i \in [0\,..\,n))\}
\end{array}
$$

Derive an implementation of command T.

Exercise 7.13. Design a command T that counts the number of times that the number 7 occurs in a given array a, according to the specification

$$
\begin{array}{l}
\textbf{const } n \in \mathbb{N}, a : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{Z} \\
\textbf{var } x : \mathbb{Z} \\
\quad \{P : \ \textbf{true}\} \\
T \\
\quad \{Q : \ x = \#\{i \in [0\,..\,n) \mid a[i] = 7\}\ \}
\end{array}
$$

Exercise 7.14. Determine a command T that satisfies

$$
\begin{array}{l}
\textbf{const } n \in \mathbb{N}, a : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{Z} \\
\textbf{var } x : \mathbb{Z} \\
\quad \{P :\ \textbf{true}\} \\
T \\
\quad \{Q :\ x = \Sigma\ (a[i] \cdot a[j] \mid i, j : 0 \leq i < j < n)\}
\end{array}
$$

At first sight, it may seem that implementing this specification requires $O(n^2)$ computation steps, but it can be done with linear complexity. For this purpose, first determine recurrence relations for the expressions $B(k)$ and $C(k)$, defined by

$$
\begin{array}{lcl}
B(k) & = & \Sigma\ (a[i] \cdot a[j] \mid i, j : 0 \leq i < j < k) \\
C(k) & = & \Sigma\ (a[i] \mid i : 0 \leq i < k)
\end{array}
$$

Use the recurrence relations to derive a program T that satisfies the above specification with complexity linear in n.

Exercise 7.15. A polynomial $a_0 + a_1 \cdot x + \ldots a_{n-1} \cdot x^{n-1}$ can be represented by an array $a[0\,..\,n)$, by identifying $a_i = a[i]$. We specify the computation of the value of the polynomial for the argument x by

$$
\begin{array}{l}
\textbf{const } n \in \mathbb{N},\ x \in \mathbb{R},\ a : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{R} \\
\textbf{var } y : \mathbb{R} \\
\quad \{P :\ \textbf{true}\} \\
T \\
\quad \{Q :\ y = \Sigma\ (a[i] \cdot x^i \mid i : 0 \leq i < n)\}
\end{array}
$$

Determine a command T that satisfies this. Because exponentiation is *not* in our programming language, it may seem that the implementation requires $O(n^2)$ computation steps, but there is a simple repetition that implements T with time complexity linear in n. Determine such a solution using the expression

$$S(k) = \Sigma\ (a[i] \cdot x^{i-k} \mid i : k \leq i < n)$$

First determine $S(n)$ and express $S(k-1)$ in $S(k)$ for $k > 0$.

The resulting algorithm T is known as *Horner's scheme*.

Chapter 8

Search problems

In this section, we treat some important search algorithms.

8.1 Linear search

We derive a command L that computes the smallest natural number that satisfies a given property prop, given that such a number exist. We specify the command by

$$
\begin{array}{l}
\textbf{var } k : \mathbb{Z} \\
\quad \{P :\ M = \mathsf{Min}\ \{i \in \mathbb{N} \mid \mathsf{prop}(i)\} < \infty\ \} \\
L \\
\quad \{Q :\ k = M\ \}
\end{array}
$$

Note that this specification is biregular: the precondition is constant (independent of the program variables) and the postcondition is of the form $x = X$. We can therefore keep precondition P out of the annotation, and yet use it whenever needed.

Precondition P implies $M < \infty$. It follows that M is a natural number that satisfies prop. In fact, precondition P is equivalent with

$$M \in \mathbb{N}\ \wedge\ \mathsf{prop}(M)\ \wedge\ (\forall i \in \mathbb{N} : \mathsf{prop}(i) \rightarrow M \leq i) \tag{8.1}$$

After this preparation, we start with the roadmap.

Step 0. We expect to establish the postcondition by repeatedly considering candidate values k, and therefore choose a repetition.

Step 1. We apply the heuristic *weakening the postcondition* to choose the invariant:

$$J :\ 0 \leq k \leq M$$

The idea is to search until some value k has been found that satisfies prop. We therefore choose the guard

$$B :\ \neg\, \mathsf{prop}(k)$$

In fact, we have

$$
\begin{array}{ll}
 & J\ \wedge\ \neg B \\
\equiv & \{\ \text{definitions of } J \text{ and } B\ \} \\
 & 0 \leq k \leq M\ \wedge\ \mathsf{prop}(k) \\
\Rightarrow & \{\ k \geq 0\ \wedge\ \mathsf{prop}(k)\text{: apply the third conjunct of (8.1)}\ \} \\
 & 0 \leq k \leq M\ \wedge\ M \leq k \\
\Rightarrow & \{\ \text{arithmetic}\ \} \\
 & Q :\ k = M
\end{array}
$$

Step 2. Initialization is easy:

$$
\begin{array}{l}
\quad \{\textbf{true}\} \\
\quad\quad (* \text{ use } P *) \\
\quad \{0 \le 0 \le M\} \\
k := 0 \\
\quad \{J :\ 0 \le k \le M\}
\end{array}
$$

Step 3. The invariant $J :\ 0 \le k \le M$ implies $M - k \ge 0$, and we know that $M \in \mathbb{N}$. This justifies the choice

$$\mathsf{vf} = M - k$$

Note that a specification constant like M is not allowed to occur in the program, but it may occur in vf.

Step 4. We derive the annotated loop body as follows:

$$
\begin{array}{l}
\quad \{J \ \wedge\ B \ \wedge\ \mathsf{vf} = V\} \\
\quad \{0 \le k \le M \ \wedge\ \neg\, \mathsf{prop}(k) \ \wedge\ M - k = V\} \\
\quad\quad (* \ P \text{ implies } \mathsf{prop}(M), \text{ therefore } k \ne M \ *) \\
\quad \{0 \le k + 1 \le M \ \wedge\ M - k = V\} \\
\quad\quad (* \ P \text{ also implies } M < \infty, \text{ therefore } M - (k+1) < M - k \ *) \\
\quad \{0 \le k + 1 \le M \ \wedge\ M - (k+1) < V\} \\
k :=\ k + 1\ ; \\
\quad \{J \wedge \mathsf{vf} < V :\ 0 \le k \le M \ \wedge\ M - k < V\}
\end{array}
$$

Step 5. We thus find command L (linear search):

$$
\begin{array}{l}
\quad \{\ P :\ M = \mathsf{Min}\ \{i \in \mathbb{N} \mid \mathsf{prop}(i)\} < \infty\ \} \\
k :=\ 0\ ; \\
\quad \{\ J :\ 0 \le k \le M\ \} \\
\quad\quad (* \quad \mathsf{vf} :\ M - k \quad *) \\
\textbf{while}\ \neg\, \mathsf{prop}(k)\ \textbf{do} \\
\quad k :=\ k + 1 \\
\textbf{end} \\
\quad \{\ Q :\ k = M\ \}
\end{array}
$$

8.2 Linear search of a value in an array

An important application is the search of a value w in an array a of length n. In other words, we are looking for an index i with $a[i] = w$. If w occurs in the array, the program should yield the smallest index i with $a[i] = w$. We reckon, however, with the possibility that w does not occur in the array. In such a case, we want the program to yield the number n. All this is specified in

$$
\begin{array}{l}
\textbf{const}\ n : \mathbb{N},\ w : \mathbb{Z},\ a : \mathsf{array}\ [0\,..\,n)\ \text{of}\ \mathbb{Z} \\
\textbf{var}\ k : \mathbb{Z} \\
\quad \{P :\ M = \mathsf{Min}\ \{i \in \mathbb{N} \mid n \le i \vee a[i] = w\}\ \} \\
L \\
\quad \{\ Q :\ k = M\ \}
\end{array}
\tag{8.2}
$$

Note that P implies that $M \le n$ and hence $M < \infty$. We can therefore directly apply (8.1), when we define

$$\mathsf{prop}(i) \ \equiv\ (n \le i \ \vee\ a[i] = w)$$

We then have $\neg\,\mathsf{prop}(i) \;\equiv\; (i < n \;\wedge\; a[i] \neq w)$. This results in the command:

$k := 0$;
while $k < n \;\wedge\; a[k] \neq w$ **do**
  $k := k + 1$
end

What happens in this program when w does not occur in the array? In this case, the command considers all elements of the array, and then evaluates the test $n < n \;\wedge\; a[n] \neq w$. The first conjunct yields **false**, but the expression $a[n]$ is undefined, because the index is not in the domain of the array. Therefore, the truth value of $a[n] \neq w$ cannot be determined. This truth value seems to be unnecessary because **false** $\wedge\, C$ is **false** for every value of C.

For most programming languages (e.g., C and Java), this is not a problem, because they evaluate conjunctions from left to right, and when the lefthand argument of a conjunction is **false**, they return **false** and do not evaluate the righthand conjunct (short circuit evaluation).

It follows that reversing the order of a conjunction can affect the execution of a command. For example, if we translate the above command in Java and replace the guard $k < n \wedge a[k] \neq w$ by the logically equivalent expression $a[k] \neq w \;\wedge\; k < n$, this would give a runtime error when value w does not occur in the array. In C, it would also be incorrect, but it would probably not result in an error message but crash instead.

8.3 Binary search in ordered sequences

Linear search is the only systematic way to search in an unordered array of objects. Searching can be done much more efficiently, when the array is ordered. How do you search for a word w in a dictionary? A rather naive but efficient method is to open the book halfway, to decide if w is in the first half or in the second half, and then to repeat this with the remaining half of the book. In each step of the program, the size of the search space is divided by 2. If the dictionary has a million words, in 20 steps the search space is a single word, because $1000000 \leq 2^{20}$, and then you see whether w occurs. This algorithm is called *binary search*.

We treat this algorithm for the search in *ordered* arrays of numbers. We start with a definition to distinguish various types of ordered arrays.

Consider an interval V of $\mathbb{Z}$ and a function $f : V \to \mathbb{R}$. We may regard f as a sequence of numbers. We define:

f is called *ascending* if $\forall i, j \in V : (i \leq j \to f(i) \leq f(j))$
f is called *increasing* if $\forall i, j \in V : (i < j \to f(i) < f(j))$
f is called *descending* if $\forall i, j \in V : (i \leq j \to f(i) \geq f(j))$
f is called *decreasing* if $\forall i, j \in V : (i < j \to f(i) > f(j))$

f is called *monotonic* if it has one of the above properties.[1]

We now consider specification (8.2) with the additional assumption that array a is ascending. If the value w occurs in the array, the smallest index i with $w = a[i]$ is also the smallest index with $w \leq a[i]$. If w does not occur, the smallest index i with $w \leq a[i]$ is also a useful result: in that case, the simple test $w = a[i]$ suffices to decide that w does not occur in the array (an example of active finalization).

We therefore change specification (8.2) into

const $n : \mathbb{N},\; w : \mathbb{Z},\; a :$ array $[0\,..\,n)$ of $\mathbb{Z}$
  { P : a is ascending }
var $k : \mathbb{Z}$;
T
  $\{Q :\; k = \mathsf{Min}\,\{i \in \mathbb{N} \mid n \leq i \;\vee\; w \leq a[i]\}\;\}$

[1] Warning: other definitions are also used (but not in these notes)!

Here, we have eliminated the specification constant M, because we want to massage the post-condition in the following way

$$
\begin{array}{ll}
& Q:\ k = \mathsf{Min}\ \{i \in \mathbb{N} \mid n \le i\ \vee\ w \le a[i]\} \\
\equiv & \{\ \text{properties of Min}\ \} \\
& 0 \le k\ \wedge\ (n \le k \vee w \le a[k])\ \wedge\ \forall i \in \mathbb{N}((n \le i \vee w \le a[i]) \to k \le i) \\
\equiv & \{\ \text{logic: contraposition}\ \} \\
& 0 \le k\ \wedge\ (n \le k \vee w \le a[k])\ \wedge\ \forall i \in \mathbb{N}(i < k \to i < n \wedge a[i] < w) \\
\equiv & \{\ \text{use } (\forall i \in \mathbb{N} : i < k \to i < n) \equiv k \le n\text{; arithmetic}\ \} \\
& 0 \le k \le n\ \wedge\ (n \le k\ \vee\ w \le a[k])\ \wedge\ \forall i \in \mathbb{N}(i < k \to a[i] < w) \\
\equiv & \{\ \text{logic; array } a \text{ is ascending}\ \} \\
& 0 \le k \le n\ \wedge\ (k = n\ \vee\ w \le a[k])\ \wedge\ (k = 0\ \vee\ a[k-1] < w) \\
\equiv & \{\ \text{we } \textit{define}\ a[n] = \infty \text{ and } a[-1] = -\infty\text{, see below}\ \} \\
& 0 \le k \le n\ \wedge\ w \le a[k]\ \wedge\ a[k-1] < w \\
\equiv & \{\ \text{rewrite}\ \} \\
& Q:\ 0 \le k \le n\ \wedge\ a[k-1] < w \le a[k]
\end{array}
$$

The array elements $a[-1]$ and $a[n]$ were not yet defined. We can therefore give them a value. In the program, however, we will not inspect the values of $a[-1]$ and $a[n]$. We introduce them only to give meaning to the boundary cases $k = 0$ and $k = n$ in Q.

It now suffices to establish the postcondition Q. The surprise is that we can do this *without using that array a is ascending*! In other words, we implement the specification

$$
\begin{array}{l}
\textbf{const}\ n : \mathbb{N},\ w : \mathbb{Z},\ a : \text{array}\ [0\,..\,n)\ \text{of}\ \mathbb{Z} \\
\quad \{\ P:\ a[-1] = -\infty\ \wedge\ a[n] = \infty\ \} \\
\textbf{var}\ k : \mathbb{Z} \\
T \\
\quad \{Q:\ 0 \le k \le n\ \wedge\ a[k-1] < w \le a[k]\ \}
\end{array}
$$

Step 1. We apply the heuristic *splitting a variable* and introduce a new variable $j : \mathbb{Z}$:

$$
\begin{array}{l}
J:\ 0 \le j \le k \le n\ \wedge\ a[j-1] < w \le a[k]\,, \\
B:\ j \ne k
\end{array}
$$

It is clear that $J \wedge \neg B\ \Rightarrow\ Q$.

Step 2. Initialization is easy, when you work from bottom to top:

$$
\begin{array}{l}
\quad \{P:\ a[-1] = -\infty\ \wedge\ a[n] = \infty\} \\
\quad\quad (*\ n \in \mathbb{N} \text{ and arithmetic}\ *) \\
\quad \{\ 0 \le 0 \le n \le n\ \wedge\ a[-1] < w \le a[n]\} \\
j := 0\ ; \\
k := n \\
\quad \{\ J:\ 0 \le j \le k \le n\ \wedge\ a[j-1] < w \le a[k]\ \}
\end{array}
$$

Step 3. We take

$$\mathsf{vf} = k - j$$

We have $J \wedge B\ \Rightarrow\ \mathsf{vf} \ge 0$ because J contains $j \le k$.

Step 4. How to find the body of the repetition? We need to decrease the difference $k - j$. We therefore choose a number $m : \mathbb{Z}$ with $j \le m < k$, and compare w with $a[m]$. If $a[m] < w$, we need to continue the search to the right of m, and we can put $j := m + 1$. If $w \le a[m]$, we

need to continue the search to the left of m, and we can put $k := m$. We postpone the details of choosing m by assuming that we have some command S_0 that satisfies

$$\begin{array}{l} \{J \ \wedge \ B \ \wedge \ \mathsf{vf} = V\} \\ S_0 \\ \{J \ \wedge \ \mathsf{vf} = V \ \wedge \ j \leq m < k\} \end{array} \tag{8.3}$$

The body S thus becomes

$$\begin{array}{l} S_0; \\ \quad \{ \ j \leq m < k \ \} \\ \textbf{if } a[m] < w \textbf{ then} \\ \quad j := m + 1 \\ \textbf{else} \\ \quad k := m \\ \textbf{end} \end{array}$$

We prove

$$\{J \ \wedge \ B \ \wedge \ \mathsf{vf} = V\} \ S \ \{J \ \wedge \ \mathsf{vf} < V\} \tag{8.4}$$

by means of the following annotation

$$\begin{array}{l} \quad \{ \ J \ \wedge \ B \ \wedge \ \mathsf{vf} = V \ \} \\ S_0; \\ \quad \{J \ \wedge \ \mathsf{vf} = V \ \wedge \ j \leq m < k\} \\ \quad\quad (* \text{ definitions of } J \text{ and } \mathsf{vf} *) \\ \quad \{0 \leq j \leq m < k \leq n \ \wedge \ a[j-1] < w \leq a[k] \ \wedge \ k - j = V\} \\ \textbf{if } a[m] < w \textbf{ then} \\ \quad\quad \{a[m] < w \ \wedge \ 0 \leq j \leq m < k \leq n \ \wedge \ a[j-1] < w \leq a[k] \ \wedge \ k - j = V\} \\ \quad\quad\quad (* \text{ logic and arithmetic } *) \\ \quad\quad \{0 \leq m + 1 \leq k \leq n \ \wedge \ a[(m+1)-1] < w \leq a[k] \ \wedge \ k - (m+1) < V\} \\ \quad j := m + 1 \ ; \\ \quad\quad \{0 \leq j \leq k \leq n \ \wedge \ a[j-1] < w \leq a[k] \ \wedge \ k - j < V\} \\ \textbf{else} \\ \quad\quad \{w \leq a[m] \ \wedge \ 0 \leq j \leq m < k \leq n \ \wedge \ a[j-1] < w \leq a[k] \ \wedge \ k - j = V\} \\ \quad\quad\quad (* \text{ logic and arithmetic } *) \\ \quad\quad \{0 \leq j \leq m \leq n \ \wedge \ a[j-1] < w \leq a[m] \ \wedge \ m - j < V\} \\ \quad k := m \\ \quad\quad \{0 \leq j \leq k \leq n \ \wedge \ a[j-1] < w \leq a[k] \ \wedge \ k - j < V\} \\ \textbf{end} \\ \quad\quad (* \text{ collect the branches; definitions } J \text{ and } \mathsf{vf} *) \\ \quad \{J \ \wedge \ \mathsf{vf} < V\} \end{array}$$

As a final building block, we need a command S_0 that satisfies (8.3). As m does not occur in J, B, and vf, it is natural to implement S_0 by an assignment to m. Every assignment to m preserves J and $\mathsf{vf} = V$. The precondition of S_0 implies $j < k$. It follows that the choices $m := j$ and $m := k - 1$ both satisfy (8.3). This would yield variations of linear search. The idea *to halve* the search area in every step corresponds to the choice:

$$S_0 : \quad m := (j + k) \ \mathsf{div} \ 2$$

To prove that this choice satisfies (8.3), it remains to prove

$$J \ \wedge \ B \ \Rightarrow \ j \leq (j + k) \ \mathsf{div} \ 2 < k$$

This is proved as follows, using two properties of div from Chapter 2.3.1.

$$
\begin{array}{ll}
 & j \le (j+k) \text{ div } 2 \ \wedge \ (j+k) \text{ div } 2 < k \\
\equiv & \{ \text{ formulas (2.2) and (2.3) } \} \\
 & j \cdot 2 \le j+k \ \wedge \ j+k < k \cdot 2 \\
\equiv & \{ \text{ arithmetic } \} \\
 & j \le k \ \wedge \ j < k \\
\equiv & \{ \text{ logic } \} \\
 & j < k \\
\Leftarrow & \{ \text{ definitions of } J \text{ and } B \} \\
 & J \ \wedge \ B
\end{array}
$$

Note that, indeed, Hoare triple (8.3) follows.

Step 5. To summarize, we have derived the command T (binary search):

```
const n : ℕ, w : ℤ, a : array [0 .. n) of ℤ
  { P :  a is ascending }
var k, j, m : ℤ;
j := 0 ;
k := n ;
  {J :  0 ≤ j ≤ k ≤ n  ∧  a[j − 1] < w ≤ a[k]}
    (∗ vf: k − j ∗)
while j ≠ k do
  m := (j + k) div 2 ;
  if a[m] < w then
    j := m + 1;
  else
    k := m;
  end
end
  {Q :  k = Min {i ∈ ℕ | n ≤ i ∨ w ≤ a[i]} }
```

This command *binary search* is much more efficient than linear search. It has logarithmic complexity. If $n \le 2^k$, the loop body is executed at most k times. Note that, as announced, the only elements inspected are $a[m]$ for $0 \le m < n$. In particular, the elements $a[-1]$ and $a[n]$ are not inspected.

8.4 Exercises

Exercise 8.1. (a) Consider functions $f, g : V \to \mathbb{Z}$ with V an interval of $\mathbb{Z}$. Is it possible that the following holds?

$$\forall i, j \in V : (i < j \to f(i) \le f(j))$$
$$\forall i, j \in V : (i \le j \to g(i) < g(j))$$

(b) Does it follow that f is ascending? Or increasing? What about g?
(c) Is it possible that a function is both ascending and descending? If so, how? If not, why not?

Exercise 8.2. Design a command T to compute the greatest index i with $a[i] = a[0]$ for an array a as in specification (8.2). First, give a biregular specification.

Exercise 8.3. Design a command T that satisfies

const $x : \mathbb{Z}$
var $n : \mathbb{N}$
  $\{P :\ M = \mathsf{Min}\ \{i \in \mathbb{N} \mid x < 2^i\}\ \}$
T
  $\{Q :\ n = M\ \}$

The command is not allowed to use exponentiation. Use a variable y with an invariant that contains $y = 2^n$. If the specification constant M does not occur in vf, there is no need to prove that $M \neq \infty$, compare Chapter 2.3.3 (it *follows* from the fact that the command terminates).

Exercise 8.4. This is a variation on the previous exercise. Determine a command T that satisfies

const $x : \mathbb{Z}$
var $n : \mathbb{N}$
  $\{P :\ M = \mathsf{Min}\ \{i \in \mathbb{N}_+ \mid x < i\,!\}\ \}$
T
  $\{Q :\ n = M\ \}$

The remarks at the previous exercise are applicable here as well.

Exercise 8.5. We consider three alternatives for command S_0 of Chapter 8.3:

(a) $m := (j + k + 1)\ \mathsf{div}\ 2$
(b) $m := (j + k - 1)\ \mathsf{div}\ 2$
(c) $m := (2 * j + k)\ \mathsf{div}\ 3$

Determine which of these alternatives are correct.

Exercise 8.6. (a) Prove, for $j, k \in \mathbb{Z}$ and $n \in \mathbb{N}$, that

$$(k - (j + k)\ \mathsf{div}\ 2 \leq 2^{n-1}\ \wedge\ (j + k)\ \mathsf{div}\ 2 - j \leq 2^{n-1})\ \equiv\ k - j \leq 2^n$$

(b) Consider command T of Chapter 8.3. Show that, in an execution of T with precondition $n < 2^r$, the loop body is executed at most r times.

Exercise 8.7. Consider a boolean function b defined on $[0\,..\,n)$ that satisfies

$$\forall i, j \in [0\,..\,n) : (i \leq j\ \wedge\ b(i)\ \rightarrow\ b(j))$$

Design a variation of binary search to compute

$$\mathsf{Min}\ \{i \in [0\,..\,n] \mid i = n \vee b(i)\}$$

Exercise 8.8. Design a command T with logarithmic complexity that satisfies

const $n : \mathbb{N}_+,\ a$: array $[0\,..\,n]$ of $\mathbb{Z}$
var $i : \mathbb{Z}$
  $\{\ P :\ a[0] = a[n]\ \}$
T
  $\{\ Q :\ 0 \leq i < n\ \wedge\ a[i] \geq a[i+1]\ \}$

Exercise 8.9. A variation on the previous exercise. Design a command T with logarithmic complexity that satisfies

const $n : \mathbb{N},\ a$: array $[0\,..\,n]$ of $\mathbb{Z}$
var $i : \mathbb{Z}$
  $\{\ P :\ a[0] \neq a[n]\}$
T;
  $\{\ Q :\ 0 \leq i < n\ \wedge\ a[0] = a[i] \neq a[i+1]\}$

Exercise 8.10. Given is an ascending function $f : \mathbb{Z} \to \mathbb{Z}$.
(a) Design a command T that satisfies

$$\begin{array}{l}\textbf{var } y : \mathbb{Z} \\ \quad \{P :\ 0 \le f(0)\ \wedge\ 0 \le X\ \wedge\ f(X) \le X\} \\ T \\ \quad \{Q :\ y = f(y)\ \wedge\ y \le X\}\end{array}$$

Use the invariant $J :\ y \le f(y)\ \wedge\ y \le X$ and the loop body $y := f(y)$.
(b) Prove that the postcondition Q of (a) can be strengthened with the conjunct

$$R :\ \forall i \in [0\,..\,y) : i < f(i)$$

For this purpose, strengthen the invariant also with this conjunct.

Exercise 8.11. The boolean function $b(x)$ is defined for all $x \in \mathbb{Z}$. It is **true** for at least one argument x. Design a command to determine such an argument.

Exercise 8.12. *The king's problem.* Given is a group of n persons with the numbers $0 \le i < n$. One of them is the king, say with number X. Every person knows him (by sight), and he knows nobody (except for, possibly, himself). The boolean function $k(i, j)$ is given and expresses that i knows j. Design a command to determine X. The formal specification is:

$$\begin{array}{l}\textbf{const } n : \mathbb{N} \\ \textbf{var } p : \mathbb{N}; \\ \quad \{\ 0 \le X < n\ \wedge\ (\forall i \in [0\,..\,n) :\ i \ne X \to k(i, X)\ \wedge\ \neg\, k(X, i))\ \} \\ T; \\ \quad \{p = X\}\end{array}$$

Introduce a variable q and use the invariant $J :\ 0 \le p \le X \le q < n$. The complexity of the command should be at most linear in n, *not quadratic*.

Exercise 8.13. Specify and design a command that, for a given array, determines whether or not the array is increasing, and that stores the result in a boolean variable x.

Exercise 8.14. Given are three ascending functions $f, g, h : \mathbb{Z} \to \mathbb{Z}$. It is also given that there exist natural numbers X, Y, Z with $f(X) = g(Y) = h(Z)$. Design a command that determines such a triple of numbers, not beyond X, Y, Z. First, give a suitable formal specification.

Exercise 8.15. Given are functions $f, g, h, k : \mathbb{N} \to \mathbb{Z}$. The functions f and g are increasing. There exist natural numbers X, Y with $f(X) < h(Y)\ \wedge\ g(Y) < k(X)$. Design an efficient command to determine natural numbers x and y that satisfy $x \le X\ \wedge\ y \le Y\ \wedge\ f(x) < h(y)\ \wedge\ g(y) < k(x)$. First give the formal specification. Hint: find invariant and guard by the heuristic of *isolating a conjunct* (two conjuncts here).

Chapter 9

Two-dimensional counting

In this chapter, we treat problems about functions of two integer variables. We are especially interested in cases where there is an efficient solution due to monotonicity properties of the function. Such cases are known as saddleback search. We concentrate on saddleback search as a counting problem.

9.1 Number of grid points

We assume that a terrain is mapped with a rectangular coordinate system, and that the map contains the points (x, y) with $0 \leq x < m$ and $0 \leq y < n$. We call the points (x, y) with integer coordinates x and y *grid points*. For each grid point, the height $h(x, y)$ is known.

Given a height value w, we want to compute the number Z of grid points (x, y) with $h(x, y) < w$. The computation of Z requires a command T with the specification

$$\begin{array}{l} \textbf{const}\ m, n, w : \mathbb{Z}; \\ \quad \{\ P :\ Z = \#\{(i, j) \in [0\,..\,m) \times [0\,..\,n) \mid h(i, j) < w\}\ \} \\ \textbf{var}\ z : \mathbb{Z}; \\ T \\ \quad \{\ Q :\ Z = z\ \} \end{array} \tag{9.1}$$

In Exercise 9.1, we ask for a direct implementation of this specification.

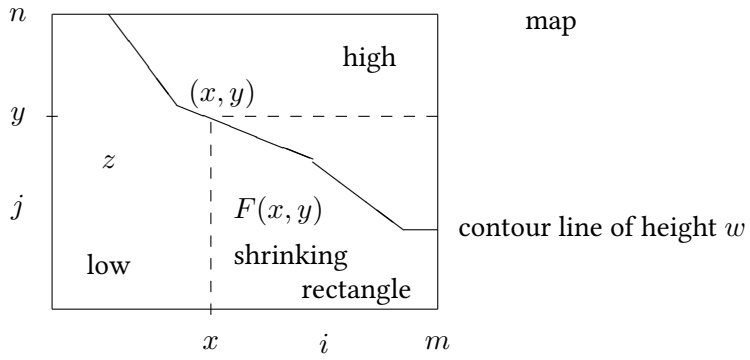
9.2 A southwestern slope

We now assume that the area of the map is a southwestern slope of a mountain. This means that, if someone goes to the north or east, he climbs, or at least he does not go down. For all x, x_0, x_1 and y, y_0, y_1, we thus have

$$\begin{array}{l} y_0 \leq y_1 \to h(x, y_0) \leq h(x, y_1) \\ x_0 \leq x_1 \to h(x_0, y) \leq h(x_1, y) \end{array} \tag{9.2}$$

In other words, the height function h is ascending in both arguments.

We treat specification (9.1) under the assumption (9.2). The additional assumption enables us to derive a more efficient solution than the one of exercise 9.1. Below we sketch an elevation map of the region. The idea of the solution is, that the area to be determined only depends on the contour line of height w. This contour line goes, roughly speaking, between NW (northwest) and SE (southeast). Note that the contour line is the separation between the grid points with height $< w$ and the points with height $\geq w$. There need not be any grid points with height w. In fact, we do not use the contour lines in the formal derivation, but only to direct the design.



The number of points to be counted is Z. We use the variable z for the number of points that have been counted, and a function $F(x, y)$ for the number of points that have yet to be counted. This gives the invariant $J : \ Z = z + F(x, y)$. The initial search area is the whole map (see the sketch). We treat the search area as a *shrinking rectangle*, determined by a search point (x, y) close to the contour line. As the contour line goes between NW and SE, we choose the shrinking rectangle to the SE of the search point (it is also possible to choose the shrinking rectangle to the NW of the search point, but the other two possibilities fail because those rectangles typically have too few points close to the contour line). The number of points to be counted in the rectangle to the SE of the search point (x, y) is

$$F(x, y) = \#\{(i, j) \mid i, j : \ x \le i < m \ \wedge \ 0 \le j < y \ \wedge \ h(i, j) < w\} \tag{9.3}$$

This definition is useful because the precondition P of (9.1) satisfies $P \ \equiv \ (Z = F(0, n))$. Note that we use half-open intervals.

The set in formula (9.3) is empty if $m \le x$, and also if $y \le 0$. As the number of elements of the empty set is 0, this proves the *base case*:

$$m \le x \ \vee \ y \le 0 \ \Rightarrow \ F(x, y) = 0 \tag{9.4}$$

We have introduced the variables x and y of formula (9.3) in order to be able to make the shrinking rectangle smaller by increasing x or decreasing y, see the sketch. We now determine how these modifications of x and y affect the value of F.

We extend the format for annotated linear proofs by allowing the introduction of conditions ("**Suppose** ...") in computation steps.

An eastward step means that x is incremented. We derive

$$\begin{array}{ll}
 & F(x, y) \\
= & \quad \{ \text{ definition of } F \ \} \\
 & \#\{(i, j) \mid i, j : \ x \le i < m \ \wedge \ 0 \le j < y \ \wedge \ h(i, j) < w\} \\
= & \quad \{ \ \textbf{Suppose } x < m \ \} \\
 & \#\{(i, j) \mid i, j : \ (x + 1 \le i < m \ \vee \ i = x) \ \wedge \ 0 \le j < y \ \wedge \ h(i, j) < w\} \\
= & \quad \{ \ \text{Splitting: } x + 1 \le i \textbf{ or } i = x \ \} \\
 & F(x + 1, y) + \#\{(x, j) \mid j : \ 0 \le j < y \ \wedge \ h(x, j) < w\} \\
= & \quad \{ \ h(x, j) \text{ is ascending in } j \text{ and } j \text{ is bounded by } 0 \le j < y. \\
 & \qquad \textbf{Suppose } y > 0. \text{ Then the greatest value occurring is } h(x, y - 1). \\
 & \qquad \textbf{Suppose } h(x, y - 1) < w. \text{ Then } h(x, j) < w \text{ for all } j < y \ \} \\
 & F(x + 1, y) + \#\{(x, j) \mid j : \ 0 \le j < y\} \\
= & \quad \{ \ \text{Counting, use } y \ge 0 \ \} \\
 & F(x + 1, y) + y
\end{array}$$

This derivation proves that

$$x < m \ \wedge \ y > 0 \ \wedge \ h(x, y - 1) < w \ \Rightarrow \ F(x, y) = F(x + 1, y) + y \tag{9.5}$$

In the annotation, we supposed $h(x, y-1) < w$ (among other things). Note that it would not be useful here to suppose $h(x, y-1) = w$ or $h(x, y-1) > w$, because that gives no information about $h(x, j) < w$ for $j < y$. The supposition $h(x, y) < w$ would have been useful, but it would be too strong, and would create problems below.

A southward step means decrementing y. We derive

$$
\begin{array}{ll}
 & F(x,y) \\
= & \quad \{ \text{ definition of } F \} \\
 & \#\{(i,j) \mid i,j : \ x \le i < m \ \wedge \ 0 \le j < y \ \wedge \ h(i,j) < w\} \\
= & \quad \{ \ \textbf{Suppose } y > 0. \text{ Splitting: } j < y-1 \textbf{ or } j = y-1 \ \} \\
 & F(x,y-1) + \#\{(i,y-1) \mid i : \ x \le i < m \ \wedge \ h(i,y-1) < w\} \\
= & \quad \{ \ h(i,y-1) \text{ is ascending in } i. \text{ Therefore } h(x,y-1) \text{ is the smallest value.} \\
 & \quad\ \ \textbf{Suppose } h(x,y-1) \ge w. \text{ Then is } h(i,y-1) \ge w \text{ for all } i \text{ with } x \le i. \\
 & \quad\ \ \text{Therefore the second term is 0 because of empty domain. } \} \\
 & F(x,y-1)
\end{array}
$$

This proves

$$y > 0 \ \wedge \ h(x, y-1) \ge w \ \Rightarrow \ F(x,y) = F(x,y-1) \tag{9.6}$$

We have thus obtained the *recurrence equations* (9.4), (9.5), and (9.6) for F. Now we can turn to the roadmap.

Step 1. We introduce variables $x, y : \mathbb{Z}$ and choose the invariant:

$$J : \ Z = z + F(x,y) \tag{9.7}$$

We find a guard B that satisfies $J \wedge \neg B \Rightarrow Q$ by

$$
\begin{array}{ll}
 & Q : \ Z = z \\
\Leftarrow & \quad \{ \ \text{definition of } J \ \} \\
 & J \ \wedge \ F(x,y) = 0 \\
\Leftarrow & \quad \{ \ \text{see (9.4)} \ \} \\
 & J \ \wedge \ (x \ge m \vee y \le 0) \\
\equiv & \quad \{ \ \text{the Morgan and formula (9.8) below} \ \} \\
 & J \ \wedge \ \neg B
\end{array}
$$

where the guard B is defined by

$$B : \ x < m \ \wedge \ y > 0 \tag{9.8}$$

Step 2: Initialization.

$$
\begin{array}{l}
\quad \{ P : \ Z = \#\{(i,j) \in [0\,..\,m) \times [0\,..\,n) \mid h(i,j) < w\} \ \} \\
\quad\ \ (* \ \text{(9.1) and (9.3)} \ *) \\
\quad \{ \ Z = 0 + F(0,n) \ \} \\
z := 0 \ ; \\
x := 0 \ ; \\
y := n \\
\quad \{ \ J : \ Z = z + F(x,y) \ \}
\end{array}
$$

We thus begin counting with (x, y) in the NW corner $(0, n)$ of the map.

Step 3. A southward step decrements y. An eastward step increments x. We therefore define the variant function $\text{vf} = y + m - x$, where we added m to keep the value nonnegative. The guard $B : \ x < m \ \wedge \ y > 0$ implies the bound $\text{vf} \ge 0$.

Step 4. The annotated loop body is obtained using the other two recurrence relations:

$\{\ J\ \wedge\ B\ \wedge\ \mathsf{vf} = V\}$
  $(*$ definitions of J, B, and vf $*)$
$\{Z = z + F(x,y)\ \wedge\ x < m\ \wedge\ y > 0\ \wedge\ y + m - x = V\}$
if $h(x, y-1) < w$ **then**
  $\{\ h(x,y-1) < w\ \wedge\ Z = z + F(x,y)\ \wedge\ x < m\ \wedge\ y > 0\ \wedge\ y + m - x = V\}$
    $(*$ use (9.5); arithmetic; logic $*)$
  $\{\ Z = z + y + F(x+1,y)\ \wedge\ y + m - (x+1) < V\}$
  $z := z + y$;
  $\{\ Z = z + F(x+1,y)\ \wedge\ y + m - (x+1) < V\}$
  $x := x + 1$;
  $\{\ J\ \wedge\ \mathsf{vf} < V :\ Z = z + F(x,y)\ \wedge\ y + m - x < V\}$
else
  $\{\ h(x,y-1) \geq w\ \wedge\ Z = z + F(x,y)\ \wedge\ x < m\ \wedge\ y > 0\ \wedge\ y + m - x = V\}$
    $(*$ use (9.6); arithmetic; logic $*)$
  $\{\ Z = z + F(x,y-1)\ \wedge\ y - 1 + m - x < V\}$
  $y := y - 1$;
  $\{\ J\ \wedge\ \mathsf{vf} < V :\ Z = z + F(x,y)\ \wedge\ y + m - x < V\}$
end $(*$ collect the branches $*)$
$\{J \wedge \mathsf{vf} < V\ \}$

Step 5. Under the assumption (9.2) that h is ascending in both arguments, we thus obtain the solution:

$\{P :\ Z = F(0,n)\}$
$z := 0$;
$x := 0$;
$y := n$;
$\{J :\ Z = z + F(x,y)\}$
  $(*\ \mathsf{vf} : y + m - x\ *)$
while $x < m\ \wedge\ y > 0$ **do**
  if $h(x, y-1) < w$ **then**
    $z := y + z$;
    $x := x + 1$
  else
    $y := y - 1$
  end
end
  $\{Q :\ Z = z\}$ (9.9)

The initial value of the variant function is $\mathsf{vf} = m + n$. The loop body has the precondition $\mathsf{vf} \geq 0$. The body decreases vf. This implies that the body is executed at most $m + n + 1$ times. The time complexity is therefore linear in the sum of the lengths of the sides of the rectangle. The command of Exercise 9.1 requires $m \times n$ inspections of h, and has therefore a time complexity of order $m \times n$. This shows that command (9.9) is much more efficient than the command of Exercise 9.1.

9.3 The method of the shrinking rectangle

Let us call the method employed in Chapter 9.2 the method of the *shrinking rectangle*. The point is that we use the invariant $J :\ Z = z + F(x,y)$ and a guard B such that $\neg B$ implies that

$F(x, y) = 0$. As $F(x, y)$ is the number of points to be counted in the *shrinking rectangle*, this rectangle needs to shrink to reach the postcondition $Z = z$.

At first sight, it may look more natural to use the invariant $J : \ z = F(x, y)$. This would require initialization with an empty rectangle and steps in which the rectangle grows to the full map. It turns out that this idea of a *growing rectangle* is usually *infeasible* because it is hard to get enough information to decide how the rectangle should grow.

The command of Chapter 9.2 is important for two reasons. One reason is that it can serve as a building block for other commands. The other reason is that it is one of the simplest applications of the method of the shrinking rectangle.

> The method of the shrinking rectangle can be applicable in every search or counting problem with a function that is monotonic in two arguments.
> You first determine the direction of the contour lines, then choose a *shrinking rectangle* that can contain many points of the critical contour line, while its moving vertex remains close to this contour line. The invariant should be similar to formula (9.7).

In Chapter 9.2, we systematically worked with half-open intervals (see 2.1). This had the effect that the grid point $(x, y - 1)$ turned out to play the role we might have expected to be played by (x, y). This is not a good reason to deviate from our choice: systematically working with half-open intervals makes the probability of counting errors smaller.

When you need properties of a complicated function like $F(x, y)$ in Chapter 9.2, it is best to derive such properties *before* starting with the roadmap.

9.4 Exercises

Exercise 9.1. Design a command T that satisfies specification (9.1) with $m, n \in \mathbb{N}$. It can be done by a single repetition with the invariant

$$Z = z + \#\{(i, j) \mid 0 \le i < m \ \wedge \ y \le j < n \ \wedge \ (x \le i \vee y < j) \ \wedge \ h(i, j) < w\}$$
$$\wedge \ 0 \le x \le m \ \wedge \ 0 \le y \le n$$

Exercise 9.2. Let h be an integer-valued function of two integer arguments, which is ascending in the first argument and descending in the second argument. For given $c \in \mathbb{Z}$, we search for a grid point on the contour line of c, i.e., a point (x, y) with coordinates $x, \ y \in \mathbb{N}$ such that $h(x, y) = c$. Given is that such a point exists (possibly many).

Determine a command T that satisfies

const $c : \mathbb{Z}$
$\{ \ P : \ 0 \le X \wedge 0 \le Y \wedge h(X, Y) = c \ \}$
var $x, y : \mathbb{Z}$
T
$\{ \ Q : \ 0 \le x \le X \wedge 0 \le y \le Y \wedge h(x, y) = c \ \}$

Sketch an elevation map, with the contour line of c and a shrinking rectangle. In this case, you do not need a recurrence relation. Use the heuristic "isolating a conjunct" to find a suitable invariant.

Exercise 9.3. Given is a function $g : \mathbb{Z}^2 \to \mathbb{Z}$ that is descending in the first argument and ascending in the second argument:

$$x_0 \le x_1 \ \Rightarrow \ g(x_0, y) \ge g(x_1, y) \ \text{ and}$$
$$y_0 \le y_1 \ \Rightarrow \ g(x, y_0) \le g(x, y_1)$$

(a) Sketch an elevation map which indicates where g is high and where g is low, and how the contour lines go.

We choose a shrinking rectangle in the southwest and define

$$F(x,y) \;=\; \#\{(i,j) \mid i,j:\; 0 \le i < x \;\wedge\; 0 \le j < y \;\wedge\; g(i,j) \le 0\}$$

(b) Determine recurrence equations for the function F analogous to (9.4), (9.5), and (9.6).
(c) Construct a command that computes $F(m,n)$ for given m and n.

Exercise 9.4. Let function $h(x,y)$ be decreasing in x and ascending in y. Determine a command T that satisfies

const $m,n:\mathbb{N},\; w:\mathbb{Z}$
var $z:\mathbb{Z}$
$\quad\{Z = \#\{(i,j) \in [0\,..\,m) \times [0\,..\,n) \mid h(i,j) = w\}\}$
T
$\quad\{\,Q:\; Z = z\,\}$

Hint: you can use function ord introduced in Chapter 2.3.2 to postpone a case distinction.

Exercise 9.5. Given are constants $m,n \in \mathbb{Z}$ and a function $g:\mathbb{Z}^2 \to \mathbb{Z}$ that is increasing in both arguments. Design a command T to determine the number of pairs $(i,j) \in [0\,..\,m) \times [0\,..\,n)$ with $g(i,j) = j$. First give a formal specification.

Exercise 9.6. Given is a function $h:\mathbb{Z}^2 \to \mathbb{Z}$ which is ascending in both arguments. Determine a command T that satisfies

const $n \in \mathbb{N}, c \in \mathbb{Z}$
var $z:\mathbb{Z}$
$\quad\{P:\; Z = \#\{(i,j) \mid i,j:\; 0 \le i \le j < n \;\wedge\; h(i,j) \le c\}\}$
T
$\quad\{Q:\; Z = z\}$

In this case, the search area is triangular. Therefore, use a "shrinking triangle" rather than a shrinking rectangle.

Exercise 9.7. The function $g:\mathbb{N}^2 \to \mathbb{Z}$ is increasing in the first argument and descending in the second argument. For given constants $n \in \mathbb{N}$ and $w \in \mathbb{Z}$, specify and design a command to compute the number of pairs $(i,j) \in \mathbb{N}^2$ with $i + j < n$ and $g(i,j) = w$.

Exercise 9.8. The function $h:\mathbb{N}^2 \to \mathbb{Z}$ is ascending in both arguments. For a given constant $n \in \mathbb{N}$, specify and design a command to compute the number of pairs $(i,j) \in \mathbb{N}^2$ with $i+j < n$ and $h(i,j) > 0$.

Exercise 9.9. Let function $h:\mathbb{Z}^2 \to \mathbb{Z}$ be ascending in both arguments. Derive a command to compute

$$\#\{i \mid 0 \le i < m \;\wedge\; (\exists\, j : 0 \le j < n \;\wedge\; h(i,j) = 0)\,\}$$

For this purpose, define

$$F(x,y) = \#\{i \mid 0 \le i < x \;\wedge\; (\exists j : y \le j < n \;\wedge\; h(i,j) = 0)\,\}$$

Determine recurrence equations for this function and use the invariant (9.7).

Exercise 9.10. (a) In the situation of Chapter 9.2, determine a command to compute the number of grid points in the rectangle that satisfy $h(i,j) \le w$.
(b) Similarly for the number of grid points that satisfy $h(i,j) = w$.
Hint: combine the solutions of 9.2 and of part (a).

Exercise 9.11. Design a command to compute

$$\#\{(i,j) \in \mathbb{N}_+ \times \mathbb{N} \mid i^2 + j^2 < w\}$$

Next, use the result to determine the number of grid points within a circle with radius $\sqrt{w}$, and use this to approximate the value of π.

Exercise 9.12. The boolean function p of two integer arguments satisfies

$$p(i, j+1) \Rightarrow p(i,j)$$
$$p(i,j) \Rightarrow p(i+1, j)$$

For a given constant $m \in \mathbb{Z}$, determine an efficient command to compute

$$\#\{(i,j) \in \mathbb{N}^2 \mid i + 2 \cdot j < m \ \wedge \ p(i,j)\}$$

As before, the search area is triangular. Make a sketch of the area where p can be **true**.

Exercise 9.13. Coincidence counting. Determine a command T that satisfies

const $m, n : \mathbb{N},\ a$: array $[0\,..\,m)$ of $\mathbb{Z},\ b \in$ array $[0\,..\,n)$ of $\mathbb{Z}$
  $\{\ P :\ a$ and b are increasing $\wedge\ Z = \#\{(i,j) \in [0\,..\,m) \times [0\,..\,n) \mid a[i] = b[j]\}\}$
var $z : \mathbb{Z}$
T
  $\{\ Q :\ Z = z\ \}$

Exercise 9.14. Let function $h(x,y)$ be ascending in both x and y. Determine a command T that satisfies

const $m, n : \mathbb{N}_+$
var $z : \mathbb{Z}$
  $\{\ P :\ Z = \mathsf{Min}\ \{|h(i,j)| \mid i,j : i \in [0\,..\,m), j \in [0\,..\,n)\}\ \}$
T
  $\{\ Q :\ Z = z\}$

You may use the operator min, the value ∞, and the function $|\cdot|$ (absolute value).

Exercise 9.15. Given are positive integers a and n, and a sequence of n positive integers $f(i)$ for $0 \le i < n$. Determine a command to compute the number of contiguous subsequences of f with the sum a, i.e.,

$$\#\{(i,j) \mid 0 \le i \le j < n \ \wedge \ a = \Sigma(f(h) \mid h : i \le h < j)\}$$

First, give a formal specification.

Exercise 9.16. The function $h : \mathbb{N}^2 \to \mathbb{Z}$ is descending in both arguments. It satisfies $h(m,0) \le 0$ and $h(0,n) \le 0$.
(a) Specify and design a command to compute the maximal value of $(x+1) \cdot (y+1)$ for natural numbers x, y with $h(x,y) > 0$.
(b) Show that your solution for part (a) remains valid if you replace the expression $(x+1) \cdot (y+1)$ by $G(x,y)$, provided $G(x,y)$ is ascending in one of its arguments. Which one?

Exercise 9.17. The function $h : \mathbb{N}^2 \to \mathbb{Z}$ is increasing in the first argument and decreasing in the second argument. Given integer constants p and w, specify and design a command to compute the number of pairs $(i,j) \in \mathbb{N}^2$ that satisfy

$$i^2 + j^2 < p \ \wedge \ h(i,j) = w$$

Exercise 9.18. The function $f : \mathbb{N}^2 \to \mathbb{Z}$ is ascending in both arguments. Given integer constants n and w, specify and design a command to compute the number of pairs $(i, j) \in \mathbb{N}^2$ that satisfy

$$2 \cdot j \leq i < n \ \wedge \ f(i, j) < w$$

Exercise 9.19. The function $h : \mathbb{N}^2 \to \mathbb{Z}$ is ascending in both arguments. Design a command T, that satisfies the specification

const $m, n, w \in \mathbb{Z}$
var $z : \mathbb{Z}$
$\{P : \ Z = \mathsf{Min}\ \{i + j \mid i, j : 0 \leq i < m \ \wedge \ 0 \leq j < n \ \wedge \ h(i, j) > w\}\ \}$
T
$\{Q : \ Z = z\}$

You may use the values ∞ and $-\infty$ and the binary operator min.

Does your program and its proof remain correct when the function $h(x, y)$ is only ascending in x? Or when $h(x, y)$ is only ascending in y? In both cases: why, or why not?

Exercise 9.20. The function $f : \mathbb{N}^2 \to \mathbb{Z}$ is ascending in both arguments. Let $c \in \mathbb{Z}$ and $d \in \mathbb{N}$. Specify and design a command to compute $\#\{j \mid \exists\, i : j \leq i \leq d \wedge f(i, j) = c\}$.

Exercise 9.21. The function $h : \mathbb{N}^2 \to \mathbb{Z}$ is ascending in the first argument and increasing in the second argument. Let $c \in \mathbb{Z}$ and $n \in \mathbb{N}$ be such that $c < h(n, 0)$. Specify and design a command to compute

$$\#\ \{(i, j) \in \mathbb{N}^2 \mid h(i, j) = c\}$$

Note that there is an additional difficulty in the proof of termination. Do not solve this by complicating the guard. It can be solved by strengthening the invariant.

Exercise 9.22. The function $p : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ is *convex* in the first argument, i.e.

$$\forall x, y, z, w \in \mathbb{Z}\ (p(x, w) \ \wedge \ x \leq y \leq z \ \wedge \ p(z, w) \ \to \ p(y, w))$$

Function p is *monotonic* in the second argument, i.e.,

$$\forall x, y, z \in \mathbb{Z}\ (p(x, y) \ \wedge \ y \leq z \ \to \ p(x, z))$$

Specify and design an efficient command to compute

$$\#\{(i, j) \in [0\,..\,m) \times [0\,..\,n) \mid p(i, j)\}$$

In this case, you need a shrinking rectangle that is determined by two moving vertices. Make a sketch and use the function

$$F(x, y, z) = \#\ \{(i, j) \mid i, j : x \leq i < z \ \wedge \ 0 \leq j < y \ \wedge \ p(i, j)\}$$

Exercise 9.23. Let function $h : \mathbb{N}^2 \to \mathbb{Z}$ be ascending in the first argument (nothing is given for the second argument). Let a be an integer array of length n. Design a command to compute

$$\mathsf{Max}\ \{h(i, j) \mid i, j : 0 \leq i < n \ \wedge \ 0 \leq j < a[i]\}$$

Replace the upper limit of i and the lower bound of j by variables x and y. Add an upper bound for y to the invariant in order to form a variant function.
This shows that the method of the shrinking rectangle can be used without contour lines.

Chapter 10

Add auxiliary information to the invariant

This chapter is about repetitions where the invariant obtained from the postcondition does not contain enough information to be kept invariant, while this can be done by adding additional variables with corresponding invariants. In fact, we did this above (e.g., in Exercise 6.3 for the sequence of Fibonacci and Exercise 7.1 for n^2 and n^3).

A typical case is that we need to compute a certain value, and assign it to a variable z. We solve this by generalizing the intended value to a function, in this chapter usually of one argument. We then search for a recurrence relation for this function. In the present chapter, this recurrence relation contains a second function that is not directly available. This may repeat itself. When we have enough recurrence relations to determine the values of all occurring functions, we can turn to the roadmap. We may have to add additional conjuncts to the invariant for every auxiliary function.

We illustrate the technique by segment exercises. This is not because segment exercises themselves are very important, but because these problems are appropriate for learning the technique.

We define a *segment* of an array $a[0\,..\,n)$ to be a subarray of the form $a[p\,..\,q)$ with $0 \le p \le q \le n$. In this chapter, we use the declaration

const $n : \mathbb{N},\ a$: array $[0\,..\,n)$ of $\mathbb{Z}$
var $z : \mathbb{Z}$

To eliminate boundary conditions from the argument, we allow mentioning all values $a[i]$ with $i \in \mathbb{Z}$. In the specifications and commands, we only use $a[i]$ for indices $i \in [0\,..\,n)$.

10.1 Longest positive segments

Let us call a segment positive if all its elements are positive. We want to determine the greatest length of the positive segments of array a. We therefore introduce a predicate that expresses that the segment $a[p\,..\,q)$ is positive:

$$C(p,q) \ \equiv\ (\forall i \in [p\,..\,q) : a[i] > 0)$$

This gives the formal specification

$$\begin{array}{l} \{P:\ \textbf{true}\} \\ S \\ \{Q:\ z = \mathsf{Max}\ \{q-p \mid p,q : 0 \le p \le q \le n\ \wedge\ C(p,q)\}\} \end{array}$$

We need a thorough preparation before turning to the roadmap. We apply the heuristic *replacing a constant by a variable*, and replace the constant n by a variable k. We therefore define the function

$$L(k) \;=\; \mathsf{Max}\,\{q-p \mid p,q : 0 \le p \le q \le k \;\wedge\; C(p,q)\}$$

The postcondition thus becomes $Q:\; z = L(n)$.

The expression for $L(k)$ has the bound variables p and q, and can therefore be regarded as two nested Max expressions (one for p, another for q). There are two ways of nesting: with p outermost or q outermost. Because we want to express $L(k+1)$ in $L(k)$, and only q has a direct relationship with k in $L(k)$, we choose the nesting with q outermost:

$$L(k) \;=\; \mathsf{Max}\,\big\{\mathsf{Max}\,\{q-p \mid p : 0 \le p \le q \;\wedge\; C(p,q)\} \mid q : q \le k\big\}$$

We now use the rule (see Chapter 2.3.5):

$$\mathsf{Max}\,\{q-p \mid p : B(p)\} \;=\; q - \mathsf{Min}\,\{p \mid B(p)\}$$

Application of this with $B(p) \;\equiv\; (0 \le p \le q \;\wedge\; C(p,q))$ gives

$$L(k) \;=\; \mathsf{Max}\,\{q-E(q) \mid q : q \le k\} \tag{10.1}$$

where function E is defined by

$$E(q) \;=\; \mathsf{Min}\,\{p \mid 0 \le p \le q \;\wedge\; C(p,q)\}$$

The task is thus to determine the maximum of the sequence of numbers $q - E(q)$.

We first treat some simple cases. We have that predicate $C(p,p)$ holds for all numbers p because of empty domain. It follows that $E(0) = 0$ and $L(0) = 0$. Note that $E(q) \le q$ for every $q \ge 0$ because of $C(q,q)$, and that $E(q) = \infty$ if $q < 0$.

We derive a recurrence relation for L in

$$\begin{array}{ll} & L(k+1) \\ = & \quad \{\ (10.1)\ \} \\ & \mathsf{Max}\,\{q-E(q) \mid q : q \le k+1\} \\ = & \quad \{\ \text{splitting: } q = k+1 \textbf{ or } q \le k\ \} \\ & (k+1-E(k+1))\ \mathsf{max}\ \mathsf{Max}\,\{q-E(q) \mid q : q \le k\} \\ = & \quad \{\ (10.1)\ \} \\ & (k+1-E(k+1))\ \mathsf{max}\ L(k) \end{array} \tag{10.2}$$

Before deriving a relation for E, we note that, for $p \le q$, we have

$$\begin{array}{ll} & C(p,q+1) \\ \equiv & \quad \{\ \text{definition of } C\ \} \\ & \forall i \in [p\,..\,q+1) : a[i] > 0 \\ \equiv & \quad \{\ \text{splitting: } i < q \textbf{ or } i = q\text{; use } p \le q\ \} \\ & (\forall i \in [p\,..\,q) : a[i] > 0) \;\wedge\; a[q] > 0 \\ \equiv & \quad \{\ \text{definition of } C\ \} \\ & C(p,q) \;\wedge\; a[q] > 0 \end{array} \tag{10.3}$$

We use this result to derive

$$
\begin{array}{ll}
 & E(k+1) \\
= & \quad \{ \text{ definition of } E \} \\
 & \mathsf{Min}\ \{p \mid 0 \le p \le k+1 \ \wedge\ C(p, k+1)\} \\
= & \quad \{ \textbf{Suppose } k \ge 0 \text{; splitting with } p = k+1 \text{; use } C(k+1, k+1) \} \\
 & (k+1)\ \mathsf{min}\ \mathsf{Min}\ \{p \mid 0 \le p \le k \ \wedge\ C(p, k+1)\} \\
= & \quad \{ \text{ use (10.3) } \} \\
\bullet & (k+1)\ \mathsf{min}\ \mathsf{Min}\ \{p \mid 0 \le p \le k \ \wedge\ C(p, k) \ \wedge\ a[k] > 0\} \\
= & \quad \{ \textbf{Suppose } a[k] > 0 \} \\
 & (k+1)\ \mathsf{min}\ \mathsf{Min}\ \{p \mid 0 \le p \le k \ \wedge\ C(p, k)\} \\
= & \quad \{ \text{ definition of } C \text{; use } k \ge 0 \text{ and hence } E(k) \le k \} \\
 & E(k)
\end{array}
$$

If $a[k] \le 0$, we find

$$
\begin{array}{ll}
 & E(k+1) \\
= & \quad \{ \text{ see above at } \bullet \} \\
 & (k+1)\ \mathsf{min}\ \mathsf{Min}\ \{p \mid 0 \le p \le k \ \wedge\ C(p, k) \ \wedge\ a[k] > 0\} \\
= & \quad \{ a[k] \le 0 \text{, hence empty domain; arithmetic } \} \\
 & k+1
\end{array}
$$

We combine the last two results in a conditional expression to

$$k \ge 0 \ \Rightarrow\ E(k+1) = (a[k] > 0\ ?\ E(k) : k+1) \tag{10.4}$$

Formula (10.4) together with $E(0) = 0$ are enough for recursive computation of function E. We can now turn to the roadmap.

Step 1. To compute $L(n)$, we compute $L(k)$ for increasing k. We therefore introduce variables $z, k : \mathbb{Z}$ with the invariant $z = L(k)$ and the guard $k \ne n$. In each step we need $E(k)$. We therefore introduce an auxiliary variable $y : \mathbb{Z}$, and strengthen the invariant by adding $y = E(k)$. In view of formula (10.4), we add $k \ge 0$ to the invariant. We add $k \le n$ to prepare the proof of termination. We thus obtain:

$$
\begin{array}{ll}
J: & 0 \le k \le n \ \wedge\ z = L(k) \ \wedge\ y = E(k) \\
B: & k \ne n
\end{array}
$$

Finalization holds because of

$$
\begin{array}{ll}
 & J \ \wedge\ \neg B \\
\Rightarrow & \quad \{ \text{ delete superfluous conjuncts; double negation } \} \\
 & z = L(k) \ \wedge\ k = n \\
\Rightarrow & \quad \{ \text{ substitute } k := n \text{, and definitions of } L \text{ and } Q \} \\
 & Q
\end{array}
$$

Step 2. Initialization is easy:

$$
\begin{array}{l}
\{P:\ \textbf{true}\} \\
\quad (* \text{ see above; } n \in \mathbb{N} *) \\
\{0 \le 0 \le n \ \wedge\ 0 = L(0) \ \wedge\ 0 = E(0)\} \\
k := 0\ ;\ z := 0\ ;\ y := 0\ ; \\
\{J:\ 0 \le k \le n \ \wedge\ z = L(k) \ \wedge\ y = E(k)\}
\end{array}
$$

Step 3. In view of the invariant, the guard, and the initialization, we choose vf $= n - k$. Boundedness is proved in

$$
\begin{array}{ll}
 & J \ \wedge \ B \\
\Rightarrow & \{ \text{ definition of } J \text{, delete conjuncts } \} \\
 & k \leq n \\
\equiv & \{ \text{ definition vf} = n - k \ \} \\
 & \mathsf{vf} \geq 0
\end{array}
$$

Step 4. In the body of the repetition, k must be incremented, while preserving invariant J. This is done in

$$
\begin{array}{l}
\{ J \ \wedge \ B \ \wedge \ \mathsf{vf} = V \} \\
\quad (* \ \text{definitions of } J, B, \mathsf{vf}, \text{ use } B \ *) \\
\{ \ 0 \leq k < n \ \wedge \ z = L(k) \ \wedge \ y = E(k) \ \wedge \ n - k = V \ \} \\
\quad (* \ \text{formula (10.4)} \ *) \\
\{ \ 0 \leq k < n \ \wedge \ z = L(k) \ \wedge \ (a[k] > 0 \ ? \ y : k+1) = E(k+1) \ \wedge \ n - k = V \ \} \\
y := (a[k] > 0 \ ? \ y : k+1) \\
\{ \ 0 \leq k < n \ \wedge \ z = L(k) \ \wedge \ y = E(k+1) \ \wedge \ n - k = V \ \} \\
\quad (* \ \text{formula (10.2) and arithmetic} \ *) \\
\{ \ 0 \leq k+1 \leq n \wedge z \ \mathsf{max} \ (k+1-y) = L(k+1) \wedge y = E(k+1) \wedge n - (k+1) < V \ \} \\
z := z \ \mathsf{max} \ (k+1-y) \ ; \\
\{ \ 0 \leq k+1 \leq n \ \wedge \ z = L(k+1) \ \wedge \ y = E(k+1) \ \wedge \ n - (k+1) < V \ \} \\
k := k+1 \ ; \\
\{ \ 0 \leq k \leq n \ \wedge \ z = L(k) \ \wedge \ y = E(k) \ \wedge \ n - k < V \ \} \\
\{ \ J \ \wedge \ \mathsf{vf} < V \ \}
\end{array}
$$

Step 5. Conclusion.

$$
\begin{array}{l}
\textbf{const } n : \mathbb{N}, \ a : \mathsf{array} \ [0 \,..\, n) \ \mathsf{of} \ \mathbb{Z} \\
\textbf{var } z, k, y : \mathbb{Z} \\
\quad \{ \ P : \ \textbf{true} \ \} \\
k := 0 \ ; \ z := 0 \ ; \ y := 0 \ ; \\
\quad \{ \ J : \ 0 \leq k \leq n \ \wedge \ z = L(k) \ \wedge \ y = E(k) \ \} \\
\quad\quad (* \ \mathsf{vf} : n - k \ *) \\
\textbf{while } k \neq n \textbf{ do} \\
\quad y := \ (a[k] > 0 \ ? \ y : k+1) \ ; \\
\quad z := \ z \ \mathsf{max} \ (k+1-y) \ ; \\
\quad k := \ k+1 \\
\textbf{end} \\
\quad \{ \ Q : \ z = L(n) \ \}
\end{array}
$$

10.2 Longest left-minimal segments

Let us call a segment $a[p \,..\, q)$ *left-minimal* if $a[p]$ is its smallest value. The aim is to derive a command to compute the greatest length of the left-minimal segments of array a. This is expressed in the postcondition

$$Q : \ z \ = \ \mathsf{Max} \ \{ q - p \mid p, q : 0 \leq p \leq q \leq n \ \wedge \ C(p,q) \}$$

where

$$C(p,q) \ \equiv \ (\forall i \in [p \,..\, q) : a[p] \leq a[i])$$

$C(p,q)$ expresses that segment $a[p\,..\,q)$ is empty or left-minimal.

We replace the constant n by a variable k, and define

$$L(k) \;=\; \mathsf{Max}\,\{q-p \mid p,q : 0 \le p \le q \le k \;\wedge\; C(p,q)\}$$

Just as in Chapter 10.1, we have

$$\begin{array}{l} L(k) \;=\; \mathsf{Max}\,\{q - E(q) \mid q : 0 \le q \le k\}, \text{ where } E(q) \text{ is given by} \\ E(q) \;=\; \mathsf{Min}\,\{p \mid 0 \le p \le q \;\wedge\; C(p,q)\} \end{array}$$

Just as in derivation (10.2), we obtain the recurrence equation

$$L(k+1) \;=\; L(k)\ \mathsf{max}\ (k+1-E(k+1)) \tag{10.5}$$

Before turning to $E(k+1)$, we investigate function C. Its definition immediately implies

$$\forall p \in \mathbb{Z} : C(p,p+1) \tag{10.6}$$

With this we can determine the base cases for E and L:

$$E(0) = L(0) = 0 \;\wedge\; E(1) = 0 \;\wedge\; L(1) = 1 \tag{10.7}$$

On the other hand, we have

$$\begin{array}{ll} & C(p,q+1) \\ \equiv & \{\ \text{definition of } C\ \} \\ & (\forall i \in [p\,..\,q+1) :\; a[p] \le a[i]) \\ \equiv & \{\ \textbf{Suppose } p \le q. \text{ Splitting: } i < q \textbf{ or } i = q\ \} \\ & (\forall i \in [p\,..\,q) :\; a[p] \le a[i]) \;\wedge\; a[p] \le a[q] \\ \equiv & \{\ \text{definition of } C\ \} \\ & C(p,q) \;\wedge\; a[p] \le a[q] \end{array}$$

This proves

$$p \le q \;\Rightarrow\; (\,C(p,q+1) = (C(p,q) \;\wedge\; a[p] \le a[q])\,) \tag{10.8}$$

We are not yet ready to derive a recurrence relation for $E(q)$, and therefore characterize the value of $E(q)$ in a different way. For $x \in \mathbb{Z}$, we have

$$\begin{array}{ll} & 1 \le q \;\wedge\; E(q) = x \\ \equiv & \{\ \text{definition } E \text{ and (10.6)}; x < q \text{ because of } C(q-1,q)\} \\ & 0 \le x < q \;\wedge\; C(x,q) \;\wedge\; (\forall p \in [0\,..\,x) : \neg C(p,q)) \\ \equiv & \{\ (\Leftarrow) \text{ because } 0 \le p < x < q \;\wedge\; a[p] > a[x] \Rightarrow \neg\, C(p,q) \\ & \quad (\Rightarrow) \text{ Assume that } a[p] \le a[x] \text{ for some } p \in [0\,..\,x). \\ & \quad \text{Then there is } p \in [0\,..\,x) \text{ with } a[p] = \mathsf{Min}\,\{a[i] \mid i : 0 \le i \le x\}. \\ & \quad \text{By } C(x,q), \text{ this implies } C(p,q).\ \} \\ & 0 \le x < q \;\wedge\; C(x,q) \;\wedge\; (\forall p \in [0\,..\,x) : a[p] > a[x]) \end{array}$$

We now introduce the abbreviation

$$D(x) \;\equiv\; (\forall p \in [0\,..\,x) : a[p] > a[x]).$$

The above calculation thus yields the following characterization of $E(q)$:

$$1 \le q \;\wedge\; E(q) = x \quad\equiv\quad 0 \le x < q \;\wedge\; D(x) \;\wedge\; C(x,q) \tag{10.9}$$

Here, we write $D(x)$ to the left of $C(x,q)$, because $D(x)$ expresses information about the values of array a to the left of x, while $C(x,q)$ is about the values between x and q.

We now turn to recurrence relations for $E(q)$. Firstly, it is easy to see that (10.8) and (10.9) together imply

$$1 \le q \ \wedge \ E(q) = x \ \wedge \ a[x] \le a[q] \quad \Rightarrow \quad E(q+1) = x$$

On the other hand we have

$$\begin{array}{ll} & 1 \le q \ \wedge \ E(q) = x \ \wedge \ a[x] > a[q] \\ \equiv & \{\,(10.9)\,\} \\ & 0 \le x < q \ \wedge \ D(x) \ \wedge \ C(x,q) \ \wedge \ a[x] > a[q] \\ \Rightarrow & \{\,\text{definitions of } D \text{ and } C, \text{ and } (10.6)\,\} \\ & 0 \le q < q+1 \ \wedge \ D(q) \ \wedge \ C(q,q+1) \\ \Rightarrow & \{\,(10.9) \text{ with } x := q \text{ and } q := q+1\,\} \\ & E(q+1) = q \end{array}$$

We combine the last two results to the recurrence equation

$$1 \le q \ \wedge \ E(q) = x \quad \Rightarrow \quad E(q+1) = (a[x] \le a[q] \ ?\ x : q) \tag{10.10}$$

After these preparations, programming can begin. First, the specification. As the problem is trivial for $n = 0$, and formula (10.10) is not applicable if $n = 0$, we restrict the attention to the case $n > 0$. (One can easily treat the case $n = 0$ separately). We therefore choose the specification:

$$\begin{array}{l} \textbf{const } n : \mathbb{N}_+,\ a : \text{array } [0\,..\,n) \text{ of } \mathbb{Z} \\ \textbf{var } z : \mathbb{Z} \\ \quad \{\ P : \textbf{true}\} \\ T \\ \quad \{\ Q :\ z = L(n)\ \} \end{array}$$

Step 1. For the postcondition, it would be enough to introduce a variable $k : \mathbb{Z}$, and to take the invariant $z = L(k)$ and the guard $k \neq n$, because

$$z = L(k) \ \wedge \ \neg(k \neq n) \ \Rightarrow \ Q$$

In view of formulas (10.5) and (10.10), however, we introduce an auxiliary variable x with the additional invariant $E(k) = x$. This leads to the invariant and guard:

$$\begin{array}{l} J :\ 1 \le k \le n \ \wedge \ z = L(k) \ \wedge \ E(k) = x \\ B :\ k \neq n \end{array}$$

We have $J \wedge \neg B \ \Rightarrow \ Q$, as is easily verified.

Step 2. The annotated initialization becomes:

$$\begin{array}{l} \quad \{\ P :\ \textbf{true}\ \} \\ \quad\quad (*\ \text{declaration of } n \text{ and formula } (10.7)\ *) \\ \quad \{\ 1 \le 1 \le n \ \wedge \ 1 = L(1) \ \wedge \ E(1) = 0\ \} \\ x := 0\ ;\ k := 1\ ;\ z := 1\ ; \\ \quad \{\ J :\ 1 \le k \le n \ \wedge \ z = L(k) \ \wedge \ E(k) = x\ \} \end{array}$$

Step 3. We want to increment k, and therefore choose $\text{vf} = n - k$. We have $J \wedge B \Rightarrow \text{vf} \ge 0$ because J contains the conjunct $k \le n$.

Step 4. The annotated loop body:

$\{\ J\ \wedge\ B\ \wedge\ \mathsf{vf} = V\ \}$
$(*$ definitions of J, B, vf; arithmetic $*)$
$\{\ 1 \le k < n\ \wedge\ z = L(k)\ \wedge\ E(k) = x\ \wedge\ n - k = V\}$
$(*$ formula (10.10) $*)$
$\{\ 1 \le k < n\ \wedge\ z = L(k)\ \wedge\ E(k+1) = (a[x] \le a[k]\ ?\ x : k)\ \wedge\ n - k = V\}$
$x := (a[x] \le a[k]\ ?\ x : k)$;
$\{\ 1 \le k < n\ \wedge\ z = L(k)\ \wedge\ E(k+1) = x\ \wedge\ n - k = V\}$
$(*$ formula (10.5), prepare $k := k + 1$ $*)$
$\{\ 1 \le k + 1 \le n \wedge z\ \mathsf{max}\ (k + 1 - x) = L(k+1) \wedge E(k+1) = x \wedge n - (k+1) < V\}$
$k :=\ k + 1$;
$\{\ 1 \le k \le n\ \wedge\ z\ \mathsf{max}\ (k - x) = L(k)\ \wedge\ E(k) = x\ \wedge\ n - k < V\}$
$z := z\ \mathsf{max}\ (k - x)$;
$\{\ 1 \le k \le n\ \wedge\ z = L(k)\ \wedge\ E(k) = x\ \wedge\ n - k < V\}$
$\{\ J\ \wedge\ \mathsf{vf} < V\ \}$

Step 5. Conclusion.

const $n : \mathbb{N}_+,\ a : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{Z}$
var $x, k, z : \mathbb{Z}$; $\quad \{P :\ \mathbf{true}\}$
$x := 0\ ;\ k := 1\ ;\ z := 1$;
$\{\ J :\ 1 \le k \le n\ \wedge\ z = L(k)\ \wedge\ E(k) = x\ \}$
$(*\ \mathsf{vf} = n - k\ *)$
while $k \neq n$ **do**
$x := (a[x] \le a[k]\ ?\ x : k)$;
$k :=\ k + 1$
$z :=\ z\ \mathsf{max}\ (k - x)$
end $\{\ Q :\ z = L(n)\ \}$

10.3 Exercises

In each case, first give a formal specification. We use the array $a[0\,..\,n)$ declared above.

Exercise 10.1. Determine a command to compute

$$\Sigma\ (a[i] \cdot a[j] \cdot a[k] \mid i, j, k : 0 \le i < j \le k < n)$$

Take care with the inequality signs. Compare this exercise with Exercise 7.14.

Exercise 10.2. Determine a command to compute

$$\Sigma\ (a[i] \cdot a[j] \cdot a[k] \mid i, j, k : 0 \le i \le j < n\ \wedge\ i \le k < n)$$

For this purpose, express this value as $L(0)$ where the argument h of $L(h)$ replaces the *lower bound* 0 for the bound variable i. Determine a recurrence equation of $L(h-1)$ expressed in $L(h)$ and $a[h-1]$, and for $\Sigma\ (a[i] \mid i : i \in [h\,..\,n))$.

Exercise 10.3. Assume $n > 0$. Design a command to compute the greatest length of the increasing segments of array $a[0\,..\,n)$.

Exercise 10.4. Assume $n > 0$. As a variation on Chapter 10.2, design a command with the postcondition

$Q :\ z\ =\ \mathsf{Max}\ \{q - p \mid p, q : q \le n\ \wedge\ D(p,q)\}$, where
$D(p,q)\ \equiv\ p \ge 0\ \wedge\ (\forall i \in [p+1\,..\,q) : a[p] < a[i])$

Exercise 10.5. Assume that $n > 0$ and $a[0] = 0$. Determine a command to compute

$$\mathsf{Min}\ \{q - p \mid p, q : 0 \leq p < q < n \ \wedge\ a[p] = a[q] = 0\}$$

Exercise 10.6. Determine the maximal sum of the segments of array a, i.e., establish the postcondition

$$Q :\ x\ =\ \mathsf{Max}\ \{\Sigma\ (a[i] \mid i : i \in [p\,..\,q)) \mid p, q : 0 \leq p \leq q \leq n\}$$

Exercise 10.7. Determine the maximal sum of the positive segments of array a.

Exercise 10.8. The greatest common divisor of natural numbers X and Y can be expressed as an integral linear combination $s \cdot X - t \cdot Y$ of X and Y. Design a command that proves this by determining such integers s and t.

For this purpose, introduce variables x and y with the invariant that x and y are linear combinations of X and Y (initially $x = X$ and $y = Y$). Establish the postcondition $x = y = \gcd(X, Y)$.

Exercise 10.9. Design a command to compute the greatest length of segments of array $a[0\,..\,n)$ for which all elements are different. For this purpose, use the type set of $\mathbb{Z}$ with suitable corresponding operations.

Exercise 10.10. Extend the program of Chapter 10.1 in such a way that it also yields bounds p and q of a positive segment of maximal length. You need not provide a complete derivation. It suffices to give the additional conjuncts of the invariant with the proofs of the steps 1 and 2, and the result in step 5. Active finalization is an option.

Exercise 10.11. Assume $n > 0$. Design a command to compute the greatest length of segments of array a with at most *two* different values.

Exercise 10.12. The same as the previous exercise, but now for segments with at most *three* different values. Would you be able to generalize this to m different values?

Exercise 10.13. Given are two arrays declared in

$$\textbf{const}\ n : \mathbb{N}, \quad a, b : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{R}$$

Determine a command S to compute

$$\Sigma(a[i] \cdot b[j] \mid i, j :\ 0 \leq j \leq i < n)$$

The time complexity of command S should be $\mathcal{O}(n)$. First derive recurrence relations for relevant functions and give a formal specification.

Exercise 10.14. Given are arrays a and b of real numbers. All elements of b are nonnegative.

$$\textbf{const}\ n : \mathbb{N}\ , \quad a, b : \mathsf{array}\ [0\,..\,n)\ \mathsf{of}\ \mathbb{R} \qquad \{\ \forall i \in [0\,..\,n) : b[i] \geq 0\ \}$$

(a) Determine a command S to compute the maximum of the products $a[i] \cdot b[j]$ for $i < j$, as specified in

$$\mathsf{Max}\ (a[i] \cdot b[j] \mid i, j :\ 0 \leq i < j < n).$$

In the program (and its derivation), you may apply the binary operator max and the constants $+\infty$ and $-\infty$. The time complexity of command S should be $\mathcal{O}(n)$.
(b) What goes wrong when some elements of array b are negative? Can this be repaired? For instance, when all elements of array a are nonnegative?

Index