Formal Verification of Programs

Péter Battyányi
# Table of Contents

Introduction ....................................................................................................................... iv
1. While programs .................................................................................................................. 1
   1. The operational semantics of while programs .......................................................... 1
   2. Denotational semantics of while programs ............................................................... 5
   3. Partial correctness of while-programs ...................................................................... 9
   4. The stepwise Floyd–Naur method ............................................................................ 9
   5. Hoare logic from a semantical point of view ......................................................... 13
   6. Proof outlines ............................................................................................................ 16
   7. Proof rules for the Hoare calculus ........................................................................... 19
   8. The wp-calculus ........................................................................................................ 22
   9. Relative completeness of the Hoare partial correctness calculus ......................... 25
  10. The wp-calculus ......................................................................................................... 26
  11. Total correctness ....................................................................................................... 28
  12. Soundness of the Hoare total correctness calculus ................................................. 30
  13. Relative completeness of the Hoare total correctness calculus .............................. 33
2. Recursive programs ........................................................................................................ 35
   1. Proving partial correctness for recursive procedures .............................................. 35
   2. Total correctness of recursive procedures ............................................................. 41
3. Disjoint parallel programs ............................................................................................. 44
4. Shared variable parallelism ............................................................................................ 49
   1. The syntax of shared variable parallel programs ..................................................... 49
   2. Operational semantics ............................................................................................. 49
   3. Deduction system ..................................................................................................... 50
   4. Soundness and completeness ................................................................................... 54
   5. Case study ................................................................................................................ 56
5. Synchronization ............................................................................................................. 62
   1. Syntax ....................................................................................................................... 62
   2. Semantics ................................................................................................................. 62
   3. Proof rules for partial correctness .......................................................................... 62
   4. Mutual exclusion ...................................................................................................... 64
A. Mathematical background ............................................................................................. 70
   1. Sets and relations ..................................................................................................... 70
   2. Complete partial orders and fixpoint theorems ....................................................... 71
B. Exercises ....................................................................................................................... 75
   1. Operational semantics ............................................................................................. 75
   2. Denotational semantics ............................................................................................ 77
   3. Partial correctness in Hoare logic ............................................................................ 78
   4. Total correctness of while programs ....................................................................... 91
   5. The wp-calculus ....................................................................................................... 99
   6. Recursive programs ................................................................................................. 103
Bibliography ...................................................................................................................... cxiii
Introduction

This course material is an introduction to some of the most relevant techniques and methods of proving correctness of programs. The approach is assertional, we take the logical standpoint: specifications are formulated as pairs of sets of program states and we seek the proof for the correctness of programs in a formal logical setting: either by model theoretic reasoning or as a result of a formal derivation in a deduction system.

The first chapter is concerned with the case of while programs, which are the simplified representatives of programs written in an ALGOL-like programming language. The basic components are constructs for assignment, conditional, composition and a tool for formulating a while loop. We glance through the inevitable notions underlying the theory: those of a state, a formal specification and formulation of correctness statements, together with introducing the concept of operational and denotational program semantics. Then we turn to the axiomatic treatment: a Hoare-logic is defined for the verification of partial and of total correctness of while programs. We treat in some extent the questions of soundness and completeness with respect to these calculi, introducing on the meantime two forms of the Dijkstra predicate transformation semantics.

The second chapter deals with the verification of recursive programs. The simplest case is considered: we deal with parameterless recursive procedures only. The main axiomatic tools: the recursion rule and the adaptation rule are introduced and the underlying fixpoint theory is also treated in some length.

The next three chapters are concerned with the primary notions related to parallel programs. We give an outline of the fundamental concepts: starting from disjoint parallel programs we continue with parallel programs that can communicate via shared variables. In the fifth chapter we discuss some problems of synchronization and some solutions given to them.

The last but one chapter recalls the notions of fixpoint theory, and the course material is concluded with a collection of solved exercises.
Chapter 1. While programs

1. The operational semantics of while programs

While programs constitute a set of deterministic programs. A while program is a string of symbols, they represent the core of ALGOL-like languages. An abstract syntax is given for the formulation of while programs:

Definition 1.

\[
C = \text{skip} \mid X := E \mid (C_1; C_2) \mid (\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi}) \mid (\text{while } B \text{ do } C \text{ od}),
\]

where \( \{X, Y, Z, \ldots\} \) form the set of variables and \( B \) denotes a Boolean and \( E \) stands for an arithmetical expression, respectively. We write \( \mathcal{C} \) for the set of while programs.

Unless otherwise stated by Boolean expressions we mean the set of formal arithmetic expressions built from natural numbers, variables and usual number theoretic functions and relations like summation, subtraction, multiplication, division, equality, less than or equals and so on.

Example 2. \( Z := 1 \)

while \( Y \neq 0 \) do

if odd(\( Y \)) then

\( Y := Y - 1; Z := Z \cdot X \)

else

\( Y := Y \div 2; X := X \cdot X \)

fi

od

For the sake of clarity, we give some elementary definitions of these notions. For the time being, we restrict ourselves to basic arithmetic operations. We assume a fixed domain \( \mathcal{D} \) to be given, it is either the set of natural numbers or the set of integers. Unless otherwise stated let \( \mathcal{D} \) denote the natural numbers. Let capital letters \( X, Y, Z, \ldots \) denote variables, and let \( n, m, \ldots \) stand for elements of \( \mathcal{D} \). Then

\[ a = n \mid X \mid (a_0 + a_1) \mid (a_0 \cdot a_1) \mid (a_0 \div a_1) \]

are arithmetical expressions. The set of arithmetical expressions is denoted by \( \mathcal{A}_{\exp} \). If \( \mathcal{D} \) represents the natural numbers, then subtraction is understood as follows:

\[
m \div n = \begin{cases} 
m - n & \text{if } m \geq n \\
0 & \text{otherwise} 
\end{cases}
\]

Otherwise it is the usual subtraction of integers. In the sequel, by an abuse of notation, we use \( m - n \) instead of \( m \div n \). In the definition below let \( a_0, a_1 \) be arithmetical expressions. Then

\[ B = \text{true} \mid \text{false} \mid (a_0 = a_1) \mid (a_0 \leq a_1) \mid \neg B \mid (B_0 \land B_1) \mid (B_0 \lor B_1) \]

are Boolean expressions. The set of Boolean expressions is denoted by \( \mathcal{B}_{\exp} \). In order to facilitate writing expressions we use usual conventions for the order of evaluation of operators: multiplication is evaluated before addition and subtraction etc. We also introduce convenient abbreviations, like

\[(B_0 \lor B_1) \equiv (\neg B_0 \lor B_1),\]

\[(B_0 \land B_1) \equiv (\neg B_0 \land B_1),\]

\[(B_0 \rightarrow B_1) \equiv (\neg B_0 \lor B_1),\]

where \( \rightarrow \) stands for implication, \( \land \) for conjunction, \( \lor \) for disjunction, \( \neg \) for negation.
While programs

and

\((B_0 \equiv B_1) \Leftrightarrow (B_0 \supset B_1) \land (B_1 \supset B_0)\).

If possible, we omit parentheses in Boolean expressions, stipulating that conjunction and disjunction bind stronger than implication, which is stronger than equivalence. Negation is stronger than any of the two argument logical operations.

With this in hand we can turn to the informal explanations of the elements of a program. The meaning of \texttt{skip} is self explanatory: it always terminates and leaves everything as was before its execution. We can define the abbreviation then

\[ \text{if } B \text{ then } C \text{ fi } \equiv \text{if } B \text{ then } C \text{ else skip fi}. \]

The command \( X := E \), where \( X \) is a variable and \( E \) is an arithmetical expression, is called an assignment. The meaning of the \textit{conditional statement} is again self explanatory, whereas the \textit{while statement}, or loop, acts as follows. It evaluates the Boolean expression following the \textit{while}, which is the head of the loop, and, if it evaluates to true, the execution continues with the body of the loop. After the body of the loop has terminated, the head of the loop is evaluated again. The \textit{while statement} terminates when the head of the loop evaluates to false. Of course, for proving statements about programs this informal explanation is not enough. That’s why we turn to a more rigorous description of the behaviour of programs. We need the notion of a state:

\begin{definition}
Let \( s : \text{Var} \rightarrow \mathcal{D} \) be a function. Then \( s \) is called a state. The set of states is denoted by \( \mathcal{S} \). If we are interested in the values assigned to the variables \( X_1, \ldots, X_n \) by \( s \), then we write \( s(X_1, \ldots, X_n) = (a_1, \ldots, a_n) \), provided \( s(X_1) = a_1, \ldots \), \( s(X_n) = a_n \).
\end{definition}

In effect, the while program is a function manipulating states. We start from the initial state, and to every execution of a component of the program a state is attached. When the program terminates, we obtain the result of the execution by reading the values assigned to the variables of the program by the final state, which is the state attached to the program at the time of termination. We can make precise these notions as follows. We introduce a more general form of a transition system than needed for our present purposes, since we will be able to make use of this definition even in the subsequent parts of the course material.

\begin{definition}
A pair \( (\mathcal{S}, A) \) labelled transition system (LTS), if \( a \subseteq \mathcal{S} \times \mathcal{S} \) for every \( a \in A \). If \( (p, q) \in a \in A \), then we call \( p \) the source, \( q \) the target, and we say that \( q \) is a derivative of \( p \) under \( a \). In notation: \( p \xrightarrow{a} q \). We use the notation \( q \xrightarrow{a} p \), if \( w = a_1 a_2 \ldots a_n \in A^* \). In case of \( n = 0 \), we let \( p \equiv q \). We write \( \xrightarrow{a} = \bigcup_{a \in A} \{ \xrightarrow{a} \} \).
\end{definition}

We define the operational semantics of a program as a transition system. The definition is compositional: knowing the transitions for the subprograms of the program we can determine the behaviour of the program itself. Beforehand, we have to define the meaning of the arithmetical and Boolean expressions, respectively. We assume an interpretation \( \mathcal{I} = (\mathcal{D}, \mathcal{I}_0) \) is given, where \( \mathcal{I}_0 \) is a mapping assigning concrete values to constants and functions and predicate symbols appearing in arithmetical expressions. Then the truth values of Boolean expressions are obtained in the usual way. As an abuse of notation, we denote the values assigned by \( \mathcal{I}_0 \) to the constant \( c \), function symbol \( f \), by \( \mathcal{I}[c] \) and \( \mathcal{I}[f] \) instead of \( \mathcal{I}_0[c] \) and \( \mathcal{I}_0[f] \), respectively. If \( B \) is true in the model with respect to an interpretation \( s \), we write \( \mathcal{I} \models B(s) \), or \( \mathcal{I}[B(s)] = \text{true} \). If the states are not interesting for our treatment, we may omit them. Moreover, for a Boolean expression \( B \), let \( \overline{B} = \{ s \mid \mathcal{I}[B(s)] = \text{true} \} \). Thus \( \overline{B} \) identifies a subset of \( \mathcal{S} \).

In the definition below let \( s, s_1, s_2 \) be states, \( B \) be a Boolean expression and \( C, C_1, C_2 \) be programs, respectively. The elements of the transition system belonging to a program \( C \) are elements of \( \mathcal{S} \cup \text{sub}(C) \times \mathcal{S} \), where \( \text{sub}(C) \) are the subprograms of \( C \). We denote by \( \Gamma = C \times \mathcal{S} \cup \mathcal{S} \). The transitions are defined as follows:
Definition 5.

1. 
\[ \langle \text{skip}, s \rangle \rightarrow s \]

2. 
\[ \langle X := E, s \rangle \rightarrow s[X := E(s)] \]

3. 
\[ \langle C_1, s_1 \rangle \rightarrow \langle C_1, s_2 \rangle \quad \text{implies} \quad \langle C_1; C_2, s_1 \rangle \rightarrow \langle C_1; C_2, s_2 \rangle \quad \text{for every} \]
\[ C_1 \in \text{sub}(C) \]

4. 
\[ \langle \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi }, s \rangle \rightarrow \langle C_1, s \rangle \quad \text{provided} \quad s \in B \]

5. 
\[ \langle \text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi }, s \rangle \rightarrow \langle C_2, s \rangle \quad \text{provided} \quad s \notin B \]

6. 
\[ \langle \text{while } B \text{ do } C \text{ od}, s \rangle \rightarrow \langle C; \text{while } B \text{ do } C \text{ od}, s \rangle \quad \text{if} \quad s \in B \]

7. 
\[ \langle \text{while } B \text{ do } C \text{ od}, s \rangle \rightarrow s \quad \text{if} \quad s \notin B \]

Definition 6. A transition sequence
\[ \langle C, s \rangle \rightarrow \langle C_1, s_1 \rangle \rightarrow \ldots \rightarrow \langle C_n, s_n \rangle \ldots \]
starting from \( s \) is a computation if it is infinite or cannot be extended any further. A computation is terminating if its last configuration is an element of \( S \). \( C \) can diverge from \( s \) if there is a diverging computation starting with \( \langle C, s \rangle \).

Example 7. Let \( C \) be the program of Example 2. Let \( s(X) = 3 \), \( s(Y) = 2 \) and \( s(Z) = 0 \) otherwise. For the moment, let us denote \( s \) by a triple representing the values assigned to \( X \), \( Y \) and \( Z \), respectively. Then
\[ \quad \langle C, (3, 2, 0) \rangle \rightarrow \langle C_1, (3, 2, 1) \rangle \rightarrow \]
\[ \langle C_0; C_1, (3, 2, 0) \rangle \rightarrow^* \langle C_1, (9, 1, 1) \rangle \rightarrow \]
\[ \langle C_0; C_1, (9, 1, 1) \rangle \rightarrow^* \langle C_1, (9, 0, 1) \rangle \rightarrow (9, 0, 1), \]
where \( C_1 \) is \( C \) with the first assignment \( Z := 1 \) omitted, and \( C_0 \) is the body of the while loop in \( C \).

Lemma 8. For any while program \( C \) and any state \( s \) there is a unique element \( \gamma \in \Gamma \) such that \( \langle C, s \rangle \rightarrow \gamma \).
Lemma 9. For any while program $C$ and any state $s$ there is exactly one computation starting from $s$.

Lemma 10. Let $C$ be a while program, assume $s$ and $s'$ are two states such that $s(X) = s'(X)$, if $X$ occurs in $C$. Then $\langle C, s \rangle$ terminates iff $\langle C, s' \rangle$ terminates, and, in this case, $\langle C, s \rangle \rightarrow^* s''$ iff $\langle C, s' \rangle \rightarrow^* s''$ for every state $s''$.

Definition 11. The operational semantics or the meaning of the while program in the operational way is defined as follows. Let $C$ be a while program, then the meaning of $C$ in the operational way is the set of pairs

$$M_{op}(C) = \{(s, s') \mid \langle C, s \rangle \rightarrow^* s'\}.$$

An alternative approach to the operational semantics is to construct a set of pairs as the operational semantics of a program in a direct way (cf. [3]). As before, let $\Gamma = C \times S \cup S$. Then $Op : C \rightarrow \mathcal{P}(\Gamma \times \Gamma)$, where $\mathcal{P}(A)$ is the set of subsets of $A$, is defined as follows. In the definition below, let $s[X/E]$, where $s$ is a state and $E$ is an arithmetical expression, denote the state which is identical to $s$ except for the value at $X$, for which $s[X/E](X) = E$.

Definition 12.

1. $Op(skip) = \{(s, s) \mid s \in S\}$
2. $Op(X := E) = \{(s[X := E], s[X/E(s)]) \mid s \in S\}$
3. $Op(C_1; C_2) = \{(s[C_1; C_2], s[C_1', C_2', s']) \mid (s[C_1, s_1], s[C_1', s_1']) \in Op(C_1) \} \cup \{(s[C_1; C_2, s_1], s[C_2, s_1']) \mid (s[C_1, s_1], s_1') \in Op(C_1) \} \cup Op(C_2)$
4. $Op(if \ B \ then \ C_1 \ else \ C_2 \ fi) = \{(s[if \ B \ then \ C_1 \ else \ C_2 \ fi], s[C_1, s]) \mid s \in B \} \cup Op(C_1) \cup \{(s[if \ B \ then \ C_1 \ else \ C_2 \ fi], s[C_2, s]) \mid s \notin B \} \cup Op(C_2)$
5. $Op(while \ B \ do \ C \ od) = \{(s[while \ B \ do \ C \ od], s[C, while \ B \ do \ C \ od, s]) \mid s \in B \} \cup \{(s[C', while \ B \ do \ C \ od, s'], s[C', while \ B \ do \ C \ od, s'']) \mid (s[C', s'], s'') \in Op(C) \} \cup \{(s[C', while \ B \ do \ C \ od, s'], s[C, while \ B \ do \ C \ od, s'']) \mid (s[C', s'], s'') \in Op(C) \} \cup \{(s[while \ B \ do \ C \ od], s) \mid s \notin \bar{B} \}$

As an example let us consider the following simple program.

Example 13. $Y := 0$;
while $X > 0$ do
\[X := X - 1; Y := Y + 1\]
end
Intuitively, the program terminates for every $X > 0$, and when it terminates the value of $Y$ equals the starting value of $X$. Let $C$ denote the whole program, $C_1$ be the while loop and let $C_2$ stand for the body of the loop. Then:

Example 14.
While programs

\[ Op(C) = \{(Y := 0; C_1, s), (C_1, s[Y/0])\} \cup Op(C_1) \]
\[ = \{(Y := 0; C_1, s), (C_1, s[Y/0])\} \]
\[ \cup \{(C_1, s, (C_2; C_1, s)) \mid s(X) > 0\} \]
\[ \cup \{(C_2; C_1, s'), (Y := Y + 1; C_1, s'[X/X - 1]) \mid s' \in S\} \cup Op(Y := Y + 1; C_1) \]
\[ \cup \{(C_1, s, s) \mid s(X) \leq 0\} \]
\[ = \{(Y := 0; C_1, s), (C_1, s[Y/0])\} \]
\[ \cup \{(C_1, s, (C_2; C_1, s)) \mid s(X) > 0\} \]
\[ \cup \{(C_2; C_1, s'), (Y := Y + 1; C_1, s'[X/X - 1]) \mid s' \in S\} \]
\[ \cup \{(Y := Y + 1; C_1, s''), (C_1, s''[Y/Y + 1]) \mid s'' \in S\} \]
\[ \cup \{(C_1, s, s) \mid s(X) \leq 0\} \]

From the set of pairs \( Op(C) \) we can extract a computation sequence for any given state \( s \). For example, let \( s(X) = 3 \), assume all the other values of \( s \) are zero. In the example below we denote \( s \) by its \( X \) and \( Y \) values. Then

\[
\langle (Y := 0; C_1, (3,0)), (C_1, (3,0)) \rangle \rightarrow \langle (C_1, (3,0)), (C_2; C_1, (3,0)) \rangle \\
\langle (C_2; C_1, (3,0)), (Y := Y + 1; C_1, (2,0)) \rangle \rightarrow \langle (Y := Y + 1; C_1, (2,0)), (C_1, (2,1)) \rangle \\
\langle (C_1, (2,1)), (C_2; C_1, (2,1)) \rangle \rightarrow \langle (C_2; C_1, (2,1)), (Y := Y + 1; C_1, (1,1)) \rangle \\
\langle (Y := Y + 1; C_1, (1,1)), (C_1, (1,2)) \rangle \rightarrow \langle (C_1, (1,2)), (C_2; C_1, (1,2)) \rangle \\
\langle (C_2; C_1, (1,2)), (Y := Y + 1; C_1, (0,2)) \rangle \rightarrow \langle (Y := Y + 1; C_1, (0,2)), (C_1, (0,3)) \rangle \\
\langle (C_1, (0,3)), (0,3) \rangle 
\]

is a computation for \( C \) with \( s(X, Y) = (3, 0) \).

Without proof we state the equivalence of the two approaches. In the theorem below, let \( Op^*(C) \) denote the reflexive and transitive closure of \( Op(C) \).

**Theorem 15.** Let \( C \) be a while program. Then, for any states \( s \) and \( s' \),

\[ \langle s, s' \rangle \in M_{Op}(C) \text{ iff } \langle (C, s), s' \rangle \in Op^*(C). \]

## 2. Denotational semantics of while programs

The denotational semantics intends to view the program behaviour from a more abstract aspect. Two programs are hard to compare, if we take into account all the smaller steps of program execution. Rather than evaluating a program from a given state step by step, the denotational semantics renders a denotation to the program, which is a partial function from states to states. We consider two programs equal if their denotations coincide. More formally,

\[ C_0 \sim C_1 \text{ iff } \forall s_0 \exists s_1 \left( \langle C_0, s_0 \rangle \rightarrow^* s_1 \text{ iff } \langle C_1, s_0 \rangle \rightarrow^* s_1 \right). \]

The question is whether we are able to give a more direct way to compute the denotations of program than the operational approach. Before giving the denotations of programs, we should give the denotations of arithmetic and Boolean expressions, but this is straightforward, so we ignore this task. We turn directly to defining the denotations of programs. The denotation of a program \( C \) is the partial function \( \mathcal{C} : \mathcal{S}_C \to \mathcal{S} \), where \( \mathcal{S}_C \) is the subset of \( \mathcal{S} \) on which \( C \) is defined. In what follows, we may use the more convenient notation \( \langle s, s' \rangle \in \mathcal{C} \) rather than \( \mathcal{C}(s) = s' \). Moreover, to make our notation more illustrative, we write \( B(s) = \text{true} \), if \( s \in B \), and \( B(s) = \text{false} \), if \( s \notin B \).

**Definition 16.**
Remark 17. We remark that in the above definition the composition of relations is used in a manner somewhat different from the treatment of some of the textbooks. If \( A, B \), and \( C \) are arbitrary sets and \( \mathcal{C} \) is a relation over \( A \times B \), and \( \mathcal{D} \) is a relation over \( B \times C \), then in a large number of the textbooks the composition \( \{ (s, s'') \mid \exists s' (s', s'' \in \mathcal{D} \land (s', s'' \in \mathcal{C}) \} \) is denoted by \( \mathcal{D} \circ \mathcal{C} \). To facilitate readability, however, we distinguish the notation of the composition reflecting a different order of the components: we use \( \mathcal{C} \circ \mathcal{D} \) instead of \( \mathcal{D} \circ \mathcal{C} \), as detailed in the Appendix.

In Definition 16 the least fixpoint of an operator is determined. By the Kanster-Tarski theorem it exists provided \( \Lambda : \mathcal{P}(\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{P}(\mathcal{S} \times \mathcal{S}) \) is continuous.

Lemma 18. \( \Lambda : \mathcal{P}(\mathcal{S} \times \mathcal{S}) \rightarrow \mathcal{P}(\mathcal{S} \times \mathcal{S}) \) is continuous.

Let us calculate the denotational semantics of the program of Example 13. As before, let \( C \) denote the whole program, \( C_1 \) denote the loop and \( C_2 \) denote the body of the loop.

Example 19.

\[
\begin{align*}
\mathcal{C} & = C_1 \circ \mathcal{Y} = 0 \\
& = \bigcup_{i=0}^{\infty} \Lambda_0 \circ \{(s, s[Y/0]) \mid s \in \mathcal{S}\} \\
& = \bigcup_{i=0}^{\infty} \{(s, s[Y/0]) \mid s \in \mathcal{S}\},
\end{align*}
\]

where \( \Lambda_i \) are the approximations of \( \Lambda \) according to Theorem 142. Let us calculate the values of \( \Lambda_i : \)

\[
\begin{align*}
\Lambda_0 & = \{(s, s) \mid s(X) \leq 0\} \\
\Lambda_1 & = \{(s, s') \mid s(X) > 0 \land \exists s'' ((s, s'') \in C_2 \land (s'', s'') \in \Delta_0) \} \cup \{(s, s) \mid s(X) \leq 0\} \\
& = \{(s, s[X/X - 1][Y/Y + 1]) \mid s(X) > 0 \land s[X/X - 1][Y/Y + 1](X) \leq 0\} \\
& \cup \{(s, s) \mid s(X) \leq 0\} \\
& = \{(s, s[X/X - 1][Y/Y + 1]) \mid s(X) = 1\} \cup \{(s, s) \mid s(X) \leq 0\} \\
\Lambda_2 & = \{(s, s') \mid s(X) > 0 \land \exists s'' ((s, s'') \in C_2 \land (s'', s'') \in \Delta_1) \} \cup \{(s, s) \mid s(X) \leq 0\} \\
& = \{(s, s[X/X - 2][Y/Y + 2]) \mid s(X) = 2\} \\
& \cup \{(s, s[X/X - 1][Y/Y + 1]) \mid s(X) = 1\} \\
& \cup \{(s, s) \mid s(X) \leq 0\}
\end{align*}
\]

Continuing in this way we obtain that
\[ \Delta_n = \bigcup_{0 < m \leq n} \{ (s, s[X/X - m][Y/Y + m]) \mid s(X) = m > 0 \} \cup \{ (s, s) \mid s(X) \leq 0 \}. \]

Thus
\[ \mathcal{C} = \bigcup_n \{ (s, s[X/0][Y/s(X)]) \mid s(X) > 0 \} \cup \{ (s, s[Y/0]) \mid s(X) \leq 0 \}. \]

So far we have defined two different approaches to the meanings of programs. The question arises naturally how these approaches are connected. We state and prove the following theorem about the relation of the two approaches.

**Theorem 20.** The operational and denotational meanings of programs coincide. More precisely, for every program \( C \),

\[ \mathcal{M}_{op}(C) = \mathcal{C}. \]

**Proof.**

- (\( \subseteq \)) Assume \( (s_0, s_1) \in \mathcal{M}_{op}(C) \), that is, \( (C, s_0) \rightarrow^n s_1 \). The proof goes by induction on the length of the reduction sequence.

- If \( n = 1 \), the statement trivially holds.

- Assume \( n > 1 \), then we can distinguish several subcases.
  
  - \( C = C_0; C_1 \) : assume \( s_0 \in B \). Hence, \( (C, s_0) \rightarrow (C_0; C_1; s_0) \rightarrow^{n-1} s_1 \). This involves, by induction hypothesis, \( (s_0, s_2) \in C_0 \) and \( (s_2, s_1) \in C_1 \).
    
    By Definition 16, this implies \( (s_0, s_1) \in C_0; C_1 \).

  - \( C = \text{if } B \text{ then } C_0 \text{ else } C_1 \) \( fi \) : assume \( s_0 \in B \). Hence, \( (C, s_0) \rightarrow (C_0; C_1; s_0) \rightarrow^{n-1} s_1 \). This involves, by induction hypothesis, \( (s_0, s_2) \in C_0 \) and \( (s_2, s_1) \in C_1 \). The case for \( s \notin B \) is treated similarly, and, by Definition 16, we obtain the result.

  - \( C = \text{while } B \text{ do } C_0 \text{ od } \) : assume \( s_0 \in B \). Then \( (C, s_0) \rightarrow (C_0; C_1; s_0) \rightarrow^{*} s_1 \). By induction hypothesis we obtain \( (s_0, s_1) \in C_0; C_1 \). Let \( s_0 \notin B \). Then \( (C, s_0) \rightarrow s_0 \), which is again in \( C \).
    
    Definition 16 gives the result.

- (\( \supseteq \)) The proof goes by induction on the complexity of \( C \). If \( C \) is \textit{skip}, or an assignment, then the result is immediate.

  - \( C = C_0; C_1 \) : assume \( (s_0, s_1) \in C_0; C_1 \). Then, by Definition 16, we have \( (s_0, s_2) \in C_0 \) and \( (s_2, s_1) \in C_1 \) for some \( s_2 \). By the induction hypothesis, \( (C_0, s_0) \rightarrow^{*} s_2 \) and \( (C_1, s_2) \rightarrow^{*} s_1 \). This gives \( (C_0; C_1, s_0) \rightarrow^{*} (C_1, s_2) \rightarrow^{*} s_1 \), which was to be proved.
While programs

- \( C = \text{if } B \text{ then } C_0 \text{ else } C_1 \) let \((s_0, s_1) \in C_0\) \(\text{if } B \text{ then } C_0 \text{ else } C_1 \) fi.

- \( C = \text{while } B \text{ do } C_0 \text{ od} \) let \((s_0, s_1) \in \text{while } B \text{ do } C_0 \text{ od} \) assume \(s_0 \in B\).

Then \(\text{while } B \text{ do } C_0 \text{ od} = \text{Ifp}(\Delta)\), where

\[
\Delta(\phi) = \{(s, s') \mid s \in B \land \exists s'', (s, s') \in C_0 \land (s'', s') \in \phi\} \\
\cup \{(s, s) \mid s \notin B\}.
\]

Let us consider the \(\Delta_0, \Delta_1, \ldots, \Delta_n, \ldots\).

\[
\begin{align*}
\Delta_0 &= \{(s, s) \mid s \notin B\} \\
\Delta_1 &= \{(s, s') \mid s \in B \land s' \notin B \land (s, s') \in C_0\} \\
&\quad \cup \{(s, s) \mid s \notin B\} \\
&\vdots \\
\Delta_{n+1} &= \{(s, s') \mid s \in B \land \exists s'', (s, s'') \in C_0 \land (s'', s') \in \Delta_n\} \\
&\quad \cup \{(s, s) \mid s \notin B\} \\
&\vdots
\end{align*}
\]

By Theorem 142, \(\text{Ifp}(\Delta) = \bigcup_{n=0}^{\infty} \Delta_n\). We prove by induction on \(n\) that \(\Delta_n \subseteq M_{op}(\text{while } B \text{ do } C_0 \text{ od})\).

1. \(n = 0\): by Definition 12 the assertion trivially holds.

2. \(n = k + 1\) with \(k \geq 0\): assume we have the result for \(k\). Let \((s, s'') \in C_0\) and \((s'', s') \in \Delta_n\) for some \(s''\) with \(s \in B\). Then, by the assumption for \(n\) and by induction hypothesis for \(C_0\), we have \(\langle C_0, s \rangle \rightarrow^* s''\) and \(\langle \text{while } B \text{ do } C_0 \text{ od}, s'' \rangle \rightarrow^* s'\), which, by Definitions 5 and 16, implies the result.

We should observe that if we use the notation of the previous chapter we can reformulate the appearance of \(\text{while } B \text{ do } C \text{ od}\) as

\[
\text{while } B \text{ do } C \text{ od} = \text{Ifp} \lambda \Phi. B^* (C \circ \Phi) \cup \delta[\neg B]
\]

\[
= \text{Ifp} \lambda \Phi. (B | C) \circ \Phi \cup \delta[\neg B].
\]

Additionally, we prove another handy characterization of the denotational semantics of the while loop.

**Lemma 21.** \(\text{while } B \text{ do } C \text{ od} = (B | C)^* [\neg B]\).

**Proof.** Let \(F(\phi) = (B | C) \circ \phi \cup \delta[\neg B]\) and \(G(\phi) = (B | C) \circ \phi \cup \delta\), where \(\phi \in \mathcal{P}(S \times S)\). We prove that

\(\text{Ifp}(F) = \text{Ifp}(G)[\neg B]\),

which, by Lemma 143, gives the result. To this end, we prove by induction on \(n\) that
$F^n = G^n \models \neg B_n,$

where $F^n$ and $G^n$ are the usual approximations of $\text{Ifp}(F)$ and $\text{Ifp}(G)$, respectively. For $\overline{n} = 0$ the statement is trivial. Assume $\overline{n} = k + 1$ for some $k \geq 0$, and the equation holds for $k$. Then

$$F^{k+1} = (B \mid C) \circ F^k \cup \delta \mid \neg B$$

$$= (B \mid C) \circ (G^k \models \neg B) \cup \delta \mid \neg B$$

$$= ((B \mid C) \circ G^k) \models \neg B \cup \delta \mid \neg B$$

$$= ((B \mid C) \circ G^k \cup \delta) \models \neg B$$

$$= G^{k+1} \models \neg B$$

By this, the proof is complete.

3. Partial correctness of while-programs

In this section we lay the foundations for the systematic verification of programs. To this extent, we augment the expressibility of our language a little. Firstly, we add variables representing natural numbers to our arithmetic constructions. Thus an arithmetical expression will look like as follows

$$a = n \mid i \mid X \mid (a_0 + a_1) \mid (a_0 \cdot a_1) \mid (a_0 - a_1),$$

where the new member is $i$, a variable denoting an integer value or a natural number. Next we extend Boolean expressions to be appropriate for making more complex statements about natural numbers or integers. We obtain the set of first order formulas or first order expressions in this way.

$$B = \text{true} \mid \text{false} \mid (a_0 = a_1) \mid (a_0 \leq a_1) \mid \neg B \mid (B_0 \land B_1) \mid (B_0 \lor B_1)$$

$$\quad (B_0 \implies B_1) \mid \exists i. B \mid \forall i. B$$

We define free and bound variables, substitution, renaming as usual. As to the abbreviation of formulas, we stipulate that the quantifiers should be the first in priority, which is followed by negation. Conjunction and disjunction have equal strength, they come after negation but precede implication and equivalence, which is the weakest of all operators. As mentioned before, an execution of a while program can be considered as a state transformation: we start from one state and through consecutive steps, if the program halts, we obtain the final state where no more command can be executed. This approach manifests itself most obviously in the definition of the denotational semantics of programs. Therefore, we can describe the execution of programs by a pair of sets of states.

Definition 22. Let $p, q \subseteq S$. Then the pair $(p, q)$ is called a specification. We say that the program $C$ is correct with respect to the specification $(p, q)$ if, for every $s \in p$, if there is an $s'$ such that $(s, s') \in C$, then $s' \in q$. More formally, $C$ is correct with respect to the specification $(p, q)$, if

$$p \models C \subseteq S \times q.$$  

We use the notation $\{p\}C\{q\}$ for the value of the predicate $p \models C \subseteq S \times q$.

4. The stepwise Floyd–Naur method

The stepwise Floyd–Naur method is considered as an induction principle checking the validity of the property through the subsequent parts of the program. We can identify an invariant property of the program, which is a property remaining true during the course of the execution of the program. The invariance of that property can be checked by verifying local invariance conditions: if the property holds at a point of the execution then it will
While programs

hold at the next point, too. It only remains to check that if we start from the precondition \( p \), then the set of states at the termination of the program is contained in the postcondition \( q \). We start with the necessary terminology.

We call \( i \in \mathcal{P}(\Gamma) \) a global verification condition, or global invariant, if the following holds:

\[
gvc(p, q, i) \equiv ((\forall s \in p) \langle s, C \rangle \in i) \land (i \models \text{Op}(C) \subseteq \Gamma \times i) \land (i \cap \mathcal{S} \subseteq q).
\]

We can assert the following claim

**Lemma 23.** \( \{p\}C\{q\} \iff \text{there exists an } i \text{ such that } \text{gvc}(p, q, i) \).

**Proof.** \((\Rightarrow)\) \( \{p\}C\{q\} \iff p \models C \subseteq \mathcal{S} \times q \). This means

\[
(\forall s \in p)(\forall s'((\langle s, C \rangle, s') \in \text{Op}(C) \supset s' \in q)). \tag{1.1}
\]

Let

\[
\langle p, C \rangle \models \text{Op}(C) = \{\gamma \mid \langle s, C \rangle, \gamma \in \text{Op}(C) \land s \in p\}.
\]

Trivially \( \{s, C \mid s \in p\} \subseteq \langle p, C \rangle \models \text{Op}(C) \) and \( \langle p, C \rangle \models \text{Op}(C) \cap \mathcal{S} \subseteq q \), by Equation 1.1. We have to prove \( \langle p, C \rangle \models \text{Op}(C) \subseteq \Gamma \times \langle p, C \rangle \models \text{Op}(C) \). But this is immediate from the definition of \( \langle p, C \rangle \models \text{Op}(C) \). We can conclude that \( \langle p, C \rangle \models \text{Op}(C) \) is a global verification condition for \( C \) with \( p \) and \( q \).

\((\Leftarrow)\) Assume \( \text{gvc}(p, q, i) \) for some \( i \in \Gamma \). Then \( \{s, C \mid s \in p\} \subseteq i \). Moreover, by induction on \( n \) we can see that \( i \models \text{Op}(C) \subseteq \Gamma \times i \), thus \( i \models \text{Op}(C) \subseteq \Gamma \times i \).

Hence, if \( s \in p \),

\[
\langle s, C \rangle \models \text{Op}(C) \models \mathcal{S} \subseteq i \models \text{Op}(C) \models \mathcal{S}
\]

\[
\subseteq (\Gamma \times i) \models \mathcal{S}
\]

\[
\subseteq \Gamma \times (i \cap \mathcal{S})
\]

\[
\subseteq \Gamma \times q
\]

Hence the partial correctness with respect to \( p \) and \( q \) indeed holds.

**Remark 24.** We state without proof that, if \( \{p\}C\{q\} \), then \( \langle p, C \rangle \models \text{Op}(C) \) is the strongest global verification condition for \( C \) with \( p \) and \( q \). In other words, if \( \text{gvc}(p, q, I) \), then \( \langle p, C \rangle \models \text{Op}(C) \subseteq I \).

Instead of global invariance we consider local invariants at certain program points in practice. In fact, the designations of program points mimic program executions. Local invariants attached to program points can be corresponded to global invariants in a bijective way. A label is a program: intuitively, we mark every program point with a label, which is the part of the original program yet to be executed. We denote the set of labels of \( C \) by \( \text{Labels}(C) \). Let \( i \in \mathcal{P}(\Gamma) \) be a global invariant, then \( \text{inv}(i) \) is a local invariant, where \( \text{inv} : \mathcal{P}(\Gamma) \rightarrow (\text{Labels}(C) \rightarrow \text{Assert}) \) and

\[
\text{inv}(i)(L) = \langle s \mid \langle s, L \rangle \in i\rangle,
\]

\[
\text{inv}(i)(\checkmark) = i \cap \mathcal{S}, \text{ if } \checkmark \text{ stands for the endlabel of } C.
\]
Conversely, let \( \text{inv} : \{\text{Labels}(C) \times \text{Assert}(C)\} \rightarrow \text{Prop}(T) \) be the local invariants, then \( I(\text{inv}) \) is a global invariant, where for the function

\[
I(\text{inv}) = \{\langle s, L \rangle \mid s \in \text{inv}(L) \land L \in \text{Labels}(C) \setminus \{\checkmark\} \} \cup \text{inv}(\checkmark),
\]

where \( \checkmark \) is the endlabel symbol.

**Example 25.** Consider the program \( C \) computing the largest common multiplier of \( a \) and \( b \).

\[
\begin{align*}
X & := a; \\
Y & := b; \\
\text{while } X \neq Y \text{ do} \\
\quad \text{if } X > Y \text{ then} \\
\quad \quad X & := X - Y \\
\quad \quad \text{else} \\
\quad \quad Y & := Y - X \\
\quad \text{fi} \\
\text{od}
\end{align*}
\]

Firstly, we determine the labels of \( C \). Let

\[
\begin{align*}
C_0 & = X := a, \\
C_{10} & = Y := b, \\
C_{11} & = \text{while } X \neq Y \text{ do if } X > Y \text{ then } X := X - Y \text{ else } Y := Y - X \text{ fi } \text{ od} \\
C_{110} & = \text{if } X > Y \text{ then } X := X - Y \text{ else } Y := Y - X \text{ fi} \\
C_{1100} & = X := X - Y \\
C_{1101} & = Y := Y - X
\end{align*}
\]

Then

\[
\begin{align*}
L_1 & = C \\
L_2 & = (C_{10}; C_{11}) \\
L_3 & = C_{11} \\
L_4 & = (C_{110}; C_{11}) \\
L_5 & = (C_{1100}; C_{11}) \\
L_6 & = (C_{1101}; C_{11}) \\
L_7 & = \checkmark
\end{align*}
\]

Assign set of states to the labels in the following way. As an abuse of notation, we omit the distinction between a first order formula \( P \) and the value of the formula \( \bar{P} \), which denotes the set of \( s \in \mathcal{S} \) such that \( \mathcal{I} \models \bar{P}(s) \) in our fixed interpretation \( \mathcal{I} \). To make the relation of the assertions assigned to labels more discernible, we indicate the possible parameters \( X \) and \( Y \) of every \( P_n \) when writing down \( P_n \).
While programs

\[ P_1(X, Y) = \text{true} \]
\[ P_2(X, Y) = (X, b) = (a, b) \]
\[ P_3(X, Y) = (X, Y) = (a, b) \]
\[ P_4(X, Y) = (X, Y) = (a, b) \land X \neq Y \]
\[ P_5(X, Y) = (X, Y) = (a, b) \land X > Y \]
\[ P_6(X, Y) = (X, Y) = (a, b) \land X < Y \]
\[ P_7(X, Y) = X = (a, b) \]

If we assign assertion \( P_n \) to label \( L_n \), we find that the assertions \( P_n \) satisfy the following local verification conditions. Let us define \( P \Rightarrow Q \) iff \( P \subseteq Q \), where \( P \in \text{Assert} \) denotes the set of states which make \( P \) true. Then:

\[
\begin{align*}
  a > 0 \land b > 0 & \Rightarrow P_1(X, Y) \\
P_1(X, Y) & \Rightarrow P_2(a, Y) \\
P_2(X, Y) & \Rightarrow P_3(X, b) \\
P_3(X, Y) \land X \neq Y & \Rightarrow P_4(X, Y) \\
P_4(X, Y) \land X > Y & \Rightarrow P_5(X, Y) \\
P_5(X, Y) \land X < Y & \Rightarrow P_6(X, Y) \\
P_6(X, Y) & \Rightarrow P_7(X, Y - X) \\
P_7(X, Y) & \Rightarrow X = (a, b)
\end{align*}
\]

By this, we can conclude that \( \text{inv} : L_n \to P_n \ (1 \leq n \leq 7) \) define local invariants for program \( C \). Moreover, if we define \( i \) as

\[ i = \bigcap_{n=1}^{6} \{ (s, L_n) \mid s \in \text{inv}(L_n) \} \cup \text{inv}(L_7), \]

then we have

\[ i \upharpoonright \text{Op}(C) \subseteq \Gamma \times i \]

and, if \( s \in a > 0 \land b > 0 \), then \( (s, L_1) \in i \). In addition, \( S \cap i \subseteq X = (a, b) \).

hence, we can conclude that \( i \) is a global invariant for \( C \).

In order to state the next theorem, we define informally the notion of local invariance condition. Let \( C \) be a program, assume \( P, Q \in \text{Assert} \). We say that \( \text{inv} : \text{Lab}(C) \to \text{Assert} \) is a local invariance condition for \( C \), \( P \) and \( Q \), if the following hold: let \( L \in \text{Lab}(C) \), assume \( L' \in \text{Lab}(C) \) is the label of the next program point and \( C' \) is the command to be executed next. Then:

1. \( P \subseteq \text{inv}(C) \)
2. \( \text{inv}(L) \subseteq \text{inv}(L'), \) if \( C' \) is \text{skip} 
3. \( \text{inv}(L)[X/E] \subseteq \text{inv}(L'), \) where \( C' \) is \( X := E \) and \( p[X/E] = \{ s[X/E(s)] \mid s \in p \} \) for any \( p \in \text{Assert} \)
• $\text{inv}(L) \cap B \subseteq \text{inv}(L')$, if $L$ begins with a while- or conditional instruction with condition $B$, and $L'$ is the next label when $B$ is true.

• $\text{inv}(L) \cap \neg B \subseteq \text{inv}(L')$, if $L$ begins with a while- or conditional instruction with condition $B$, and $L'$ is the next label when $B$ is false.

• $\text{inv}(\sqrt{\ }) \subseteq q$

We assert the following theorem without proof.

**Theorem 26.** Let $C$ be a program, assume $p, q \in \text{Assert}$. Let $\text{inv} : \text{Lab}(C) \rightarrow \text{Assert}$ define local invariants for $C$. Then $\{p\} C \{q\}$ holds true, if $\text{inv}$ is a local invariance condition for $C$, $p$ and $q$.

As a corollary, we can state the semantical soundness and completeness of the Floyd–Naur stepwise method.

**Theorem 27.** (semantical soundness and completeness of the stepwise Floyd–Naur method)

$\{p\} C \{q\}$ iff there exists a local verification condition for $C$, $p$, and $q$.

**Proof.** We give a sketch of the proof. First of all, we should notice that the if-case is the statement of the previous theorem. For the other direction we can observe that if $\{p\} C \{q\}$, then $i = \{\gamma \mid \exists \gamma' (\langle \gamma', \gamma \rangle \in \{\langle s, C \rangle \mid s \in p\} \cap Op^*(C))\}$ is a global verification condition for $C$, $p$, and $q$. Then it is not hard to check that $\text{inv}(i) : \text{Labels}(C) \rightarrow \text{Assert}$ defined as

$\text{inv}(i)(L) = \{s \mid \langle s, L \rangle \in i\}$,

$\text{inv}(i)(\sqrt{\ }) = i \cap S$

satisfies the local invariance condition. By Theorem 26, the result follows.

We remark that there also exists a compositional presentation of the Floyd–Naur method, which is equivalent in strength to the stepwise method illustrated above. We omit the detailed description of the compositional method, the interested reader is referred to [3].

### 5. Hoare logic from a semantical point of view

Assume $C$ is a program with precondition $p$ and postcondition $q$, respectively. We can prove the validity of $\{p\} C \{q\}$ by dissecting the proof into verifications for program components. This leads to the idea of a compositional correctness proof which consists of the following substeps. In what follows, $\{p\} C \{q\}$ stands for the truth value of the assertion $\{p\} C \{q\}$.

**Theorem 28.**

1. $\{p\} \text{skip}\{p\} = \text{true}$

2. $\{s \mid s[X/E(s)] \in q\} X := E\{q\} = \text{true}$

3. 

Created by XMLmind XSL-FO Converter.
While programs

\[ \{p\}C_1; C_2 \{q\} = true \iff (\exists i \in Assert)(\{p\}C_1\{i\} = true) \]

and

4.

\[ \{p\}if\ B\ then\ C_1\ else\ C_2\ fi\{q\} = true \iff \{p \land B\}C_1\{q\} = true \land \{p \land \lnot B\}C_2\{q\} = true \]

5.

\[ \{p \land B\}C\{p\} = true \implies \{p\}while\ B;\ do\ C\ od\{p \land \lnot B\} = true \]

6.

\[ \{p\}C\{q\} = true \iff \exists p', q' (p \subseteq p' \land q \subseteq q' \land \{p'\}C\{q'\} = true) \]

The following formulation of the correctness condition for while-loops can also be useful.

**Lemma 29.**

\[ \{p\}while\ B;\ do\ C\ od\{q\} = true \iff (\exists i \in Assert)(p \subseteq i \land \{i \land B\}C\{i\} = true \land (i \land \lnot B) \subseteq q) \]

**Proof of Theorem 28.** In what follows, we prove some of the cases of Theorem 28.

- \[ \{p\}skip\{p\} = p \iff skip \subseteq S \times p \], which trivially holds.

- \[ \{p\}X := E\{q\} = p \iff X := E \subseteq S \times q \], which is true if and only if \[ \{s[X/E(s)] \mid s \in p\} \subseteq q \]. But then \[ \{s[X/E(s)] \mid s \in \{s[X/E(s)] \mid s \in q\}\} \] is trivially contained in \[ q \].

- \[ \{p\}C_1; C_2\{q\} = p \iff C_1 ; C_2 \subseteq S \times q = p \iff C_1 ; C_2 \subseteq S \times q \], by Definition 16. Consider the assertion \[ i = \{s \mid (\exists s' \in p)((s', s) \in C_1)\} \]. Then \[ i \] is an appropriate choice for the intermediate assertion in the theorem.

- \[ \{p\}if\ B\ then\ C_1\ else\ C_2\ fi\{q\} = (p \land B) \iff C_1 \cup (p \land \lnot B) \subseteq S \times q \], by Definitions 16 and 22. The latter is equivalent to \[ \{p \land B\}C_1\{q\} = true \land \{p \land \lnot B\}C_2\{q\} = true \].

- \[ \{p\}while\ B;\ do\ C\ od\{p \land \lnot B\} = p \iff (B \cup C)^* \subseteq S \times (p \land \lnot B) \], by Lemma 21. By induction on \( n \), making use of \[ \{p \land B\}C\{p\} = true \], we can show that \[ p \subseteq (B \cup C)^n \subseteq S \times p \]. This involves \[ \{s' \mid (\exists s \in p)((s, s') \in B \cup C)^* \subseteq p \land \lnot B\}. \]

- \[ p \subseteq p', q \subseteq q' \land \{p'\}C\{q'\} \] for some \( p' \) and \( q' \): then

\[ p \mid C \subseteq p' \mid C \subseteq S \times q' \subseteq S \times q \]

Created by XMLmind XSL-FO Converter.
Proof of Lemma 29. (\(\Rightarrow\)) Assume first \(\{p\} \text{while } B; \text{do } C \text{ od}\{q\}\), which is equivalent to

\[
p \vdash (B \uparrow C)^* \vdash \neg B \subseteq S \times q.
\]  
(1.2)

Let

\[
i = \{s \mid (\exists s')(s' \in p \land (s', s) \in (B \uparrow C)^*)\}.
\]

Then \(p \subseteq i\), and \(\{i \cap B\} C\{i\}\). Moreover, by Equation 1.2, \(i \cap \neg B \subseteq q\).

(\(\Leftarrow\)) Let \(i \in \text{Assert}\) as in the statement of the lemma. Then \(\{i \cap B\} C\{i\}\) is equivalent to

\[
i \vdash (B \uparrow C) \subseteq S \times i.
\]

We can deduce from the previous relation, by induction on \(n\), that

\[
i \vdash (B \uparrow C)^n \subseteq S \times i,
\]

which implies

\[
i \vdash (B \uparrow C)^* \subseteq S \times i.
\]  
(1.3)

Applying Equation 1.3, we obtain the result as follows:

\[
p \vdash (B \uparrow C)^* \vdash \neg B \subseteq i \vdash (B \uparrow C)^* \vdash \neg B \subseteq (S \times i) \vdash \neg B = S \times (i \cap \neg B) \subseteq S \times q.
\]

Thus \(\{p\} \text{while } B; \text{do } C \text{ od}\{q\}\) is proven.

The relations of Theorem 28 give a proof method for a compositional verification of partial correctness of while programs. Thus, we can make use of the statements of Theorem 28 as the proof rules and axioms of a formal, semantical proof of partial correctness. The following theorems are devoted to this idea, the first of which is a reformulation of Theorem 28.

Theorem 30. The compositional proof method of Theorem 28 is semantically sound with respect to partial correctness of programs. In other words, \(\{p\} C\{q\}\) is proven by applying the points of Theorem 28. Then \(\{p\} C\{q\}\) is true in the sense of Definition 22.

The other direction is called semantical completeness.

Theorem 31. Let \(C\) be a program, \(p, q \in \text{Assert}\). Assume \(\{p\} C\{q\}\) holds. Then we can obtain \(\{p\} C\{q\}\) by subsequent applications of the points of Theorem 28 as axioms and proof rules.
Proof. Assume \( C, P \) and \( Q \) are such that \( \{P\}C\{Q\} \). We prove the statement by induction on the structure of \( C \). We consider only some of the cases. We refer to Point 1 of Theorem 28 as 28.1, etc.

- \( C \) is \textit{skip}: \( \{P\}\text{skip}\{Q\} \iff P \subseteq Q \). By 28.1, \( \{P\}\text{skip}\{P\} \). Moreover, \( P \subseteq P \) and \( P \subseteq Q \), together with 28.6, give the result.

- \( C \) is \( X := E \): by Definition 16, \( \{P\}C\{Q\} \iff \)

\[
\{s[X/E(s)] \mid s \in P\} \subseteq Q.
\] (1.4)

28.2 states \( \{s \mid s[X/E(s)] \in Q\}X := E\{Q\} \). But (1.4) implies \( P \subseteq \{s \mid s[X/E(s)] \in Q\} \), hence an application of 28.6 gives the result.

- \( C \) is \( C_1; C_2 \): by Definition 16, we have \( P \vdash (C_1 \circ C_2) \subseteq S \times Q \). Let \( i = \{s' \mid (\exists s \in P)(\langle s, s' \rangle \in C_1)\} \), then \( \{P\}C_1\{i\} \) and \( \{i\}C_2\{Q\} \). By induction hypothesis we obtain the provability of the latter two relations, which entails, by 28.3, the result.

- \( C \) is \textit{while} \( B \) do \( C_0 \) od: let \( i \) be as in Lemma 29. By induction hypothesis we know that \( \{i \cap B\}C_0\{i\} \) is deducible, which implies, making use of 28.5, \( \{i\} \text{while} B \text{ do} C_0 \text{ od}\{i \cap \neg B\} \). We also have, by Lemma 29, \( P \subseteq i \), and \( i \cap \neg B \subseteq Q \), which, together with 28.6, yield the result.

We used the terminology semantic soundness and completeness in the sense of [3]. That is, soundness and completeness is understood relative to the partial correctness definition of Definition 22. This means that partial correctness is defined without reference to a mathematical logical language: the set of states used here as pre- and postconditions are arbitrary subsets of \( S \). We will see in later chapters that this picture considerably changes if we allow only sets of states emerging as meanings of logical formulas.

6. Proof outlines

In order to facilitate the presentation of proofs, we can give them in the form of proof outlines: in this case local invariants are attached to certain program points. For the sake of readability, we give the rules for constructing proof outlines in the forms of derivations, like this: \( \Phi_1 \Phi_2 \ldots \Phi_n \) with the meaning that if \( \Phi_1 \ldots \Phi_n \) are proof outlines, then \( \Phi \) is also a proof outline. Let \( \{P\}C\{Q\} \) denote the the proof outline assigned to the structure \( \{P\}C\{Q\} \).

Definition 32.

1. \( \{P\}\text{skip}\{P\} \)

2. \( \{P[X/E]\}X := E\{P\} \)
While programs

\[ \{p\} C'_{\uparrow}\{r\} \quad \{r\} C'_{\downarrow}\{q\} \]
\[ \{p\} C'_{\downarrow}; \{r\} C'_{\downarrow}\{q\} \]

4.
\[ \{p \cap B\} C'_{\uparrow}\{q\} \quad \{p \cap \neg B\} C'_{\downarrow}\{q\} \]
\[ \{p\} if\ B\ then\ \{p \cap B\} C'_{\uparrow}\{q\} \ else\ \{p \cap \neg B\} C'_{\downarrow}\{q\} \fi\{q\} \]

5.
\[ \{p \cap B\} C^*\{p\} \]
\[ \{inv : p\} while\ B\ do\ \{p \cap B\} C^*\{p\} \ od\{p \cap \neg B\} \]

6.
\[ p \subseteq p' \quad \{p'\} C^*\{q'\} \quad q' \subseteq q \]
\[ \{p'\} \{p\} C^*\{q\}\{q'\} \]

7.
\[ \{p\} C^*\{q\} \]
\[ \{p\} C^{**}\{q\} \]

where \( C^{**} \) is obtained from \( C^* \) by omitting zero or more annotations except for annotations of the form \( \{inv : p\} \) for some \( p \). Let \( \{p\} C^*\{q\} \) be a proof outline. Then \( \{p\} C^*\{q\} \) is called standard, if every subprogram \( T \) of \( C \) is preceded at least one annotation and, for any two consecutive annotations \( \{p_1\} \) and \( \{p_2\} \), either \( p_1 = p \) or \( p_2 = q \). This means in effect that in a standard proof outline every proper subprogram \( T \) of \( C \) is preceded exactly one annotation, which is called \( pre(T) \). Additionally, if, for the partial correctness assertion \( \{p\} S\{q\} \), and, for some subprogram \( T \), \( pre(T) = p \) holds, we omit \( pre(T) \) and consider the remaining proof outline as standard.

The lemma below sheds light on the straightforward connection between proof outlines and partial correctness proofs à la Hoare.

**Lemma 33.**

1.

Let \( \{p\} C^*\{q\} \) hold as a proof outline. Then \( \{p\} C\{q\} \) is provable by the rules obtained from Theorem 28.

2.

Assume \( \{p\} C\{q\} \) is provable applying Theorem 28. Then there is a derivable standard proof outline \( \{p\} C^*\{q\} \).

In fact, there is also a close relation between proof outlines and compositional partial correctness proofs in Floyd–Naur style: the two methods are basically equivalent. The interested reader can find more details on the subject in [3]. We can put this relation on other words by saying that the precondition of a subprogram of \( C \) is assertion assigned to the label corresponding to the point of execution belonging to that subprogram.
Example 34. Let us take the program of Example 25. We give a proof by annotations of the partial correctness statement

\( \{ a > 0 \land b > 0 \} \cup \{ X = (a, b) \} \).

\( X := a \; ; \)
\( \{ X > 0 \land b > 0 \} \)
\( Y := b \; ; \)
\( \{ X > 0 \land Y > 0 \} \)
\( \{ \text{inv} : (X, Y) = (a, b) \} \)

while \( X \neq Y \) do

\( \{ (X, Y) = (a, b) \land X \neq Y \} \)

if \( X > Y \) then

\( \{ (X, Y) = (a, b) \land X \neq Y \land X > Y \} \)
\( \{ (X - Y, Y) = (a, b) \} \)
\( X := X - Y \)
\( \{ (X, Y) = (a, b) \} \)

else

\( \{ (X, Y) = (a, b) \land X \neq Y \land X < Y \} \)
\( \{ (X, Y - X) = (a, b) \} \)
\( Y := Y - X \)
\( \{ (X, Y) = (a, b) \} \)

fi
\( \{ (X, Y) = (a, b) \} \)

od
\( \{ (X, Y) = (a, b) \land X = Y \} \)
\( \{ X = (a, b) \} \)

Observe that in order to obtain a valid proof outline we have to ensure, by Point 6 of Definition 32, that for the consequent annotations the upper one implies the lower one. Thus we have to check the validity of the relations.
While programs

\[ X > 0 \land Y > 0 \subseteq \text{inv} : (X, Y) = (a, b), \]
\[ (X, Y) = (a, b) \land X \neq Y \land X > Y \subseteq (X - Y, Y) = (a, b), \]
\[ (X, Y) = (a, b) \land X \neq Y \land X < Y \subseteq (X, Y - X) = (a, b), \]
\[ (X, Y) = (a, b) \land X = Y \subseteq X = (a, b). \]

All of them trivially hold in our standard interpretation.

7. Proof rules for the Hoare calculus

In this section we turn to the task of giving a formal system for the verification of partial correctness assertions for while programs. Hoare’s proof system manipulates certain formulas, called Hoare formulas. The proof system is built up according to the traditions of logical calculi: it consists of a set of axioms together with a set of rules to derive conclusions from hypothesis. The axioms are themselves axiom schemata: the substitution of concrete elements in place of metavariables of the axioms render the concrete forms of the axioms. For example, if \( \{ P \} \text{skip} \{ P \} \) stands for the skip axiom, then \( \{ X > 1 \} \text{skip} \{ X > 1 \} \) is an instance of it. We defined an extended notion arithmetical expressions and logical formulas in the course of Section 1.3. Let \( Ae.exp \) and \( Form \), respectively, stand for the sets of arithmetical expressions and first order formulas defined there. We define the set of Hoare formulae as follows.

**Definition 35.** Let \( P, Q \in Form \). Let \( C \) be a program, then \( \{ P \} C \{ Q \} \) is a Hoare correctness formula. The set of Hoare correctness formulae are denoted by \( H \), while \( \overline{H} = H \cup Form \) gives the formulas of Hoare logic.

Now we present the axioms and rules of the Hoare calculus. The names next to the rules or axioms are the rule- or axiom names.

**Definition 36.**

\[ \{ P \} \text{skip} \{ P \} \] (skip)
\[ \{ P \} C_1 \{ R \} \quad \{ R \} C_2 \{ Q \} \quad \{ P \} C_1 ; C_2 \{ Q \} \] (comp)
\[ \{ P \} \text{while } B \text{ do } C \odo \{ P \land \neg B \} \] (while)
\[ \{ P \} \text{if } B \text{ then } C_1 \text{ else } C_2 \{ Q \} \] (cond)
\[ \{ P \} \text{assign } \{ P \} \text{cons } \{ P \} \]

In the consequence rule \( \text{cons} \) we denoted by \( P \Rightarrow Q \) the relation \( P \subseteq Q \), where \( P = \{ s \mid P^x(s) = \text{true} \} \) in the standard interpretation \( I \). In the next example we show in detail how to apply the Hoare rules as formal tools in proving a partial correctness assertion. We present the formal proof in a linear style rather than in a tree like form, we indicate by indices the order of deduction in the argument.

**Example 37.** Let \( C \) be \( Y := 1; \text{while } X > 0 \text{ do } Y := Y \ast X; X := X - 1 \odo \). We intend to prove \( \{ X := n \land n \geq 0 \} C \{ Y = n! \} \) as a loop invariant, we use the formula \( i = Y \ast X! = n! \land X \geq 0 \).
Definition 38. Let \( T \h \) be a set of Hoare formulas, that is \( T \h \subseteq H L \). We define inductively when a Hoare correctness formula \( \{ P \} C \{ Q \} \) is provable in the Hoare logic from \( T \h \). In notation \( \vdash T \h \{ P \} C \{ Q \} \).

1. \( \vdash T \h \ P \), if \( P \in T \h \)
2. \( \vdash T \h \ P \), if \( P \) is an axiom
3. \( \vdash T \h \ F \), if \( F \) is a rule and \( \vdash T \h \ F_i \quad (1 \leq i \leq k) \).

If we fix an interpretation, we can talk about the meaning of Hoare formulas. As before, assume that the base set of the interpretation is the set of natural numbers, function symbols like \( +, \times, - \), etc., are interpreted as the usual functions on natural numbers, similar suppositions are made concerning the predicate symbols like \( =, \leq, <, \) etc. For the sake of completeness, we give here the interpretation of terms and formulas. As before, let \( \mathcal{I} = (\mathcal{D}, \mathcal{I}_0) \) be an interpretation, where \( \mathcal{D} \) is the base set- in our case the set of natural numbers-, and \( \mathcal{I}_0 \) is an interpretation of the constants, the and function- and predicate symbols. Let \( s : V a r \rightarrow \mathcal{D} \) be a state, we denote by \( s[X \leftarrow d] \) the state, which is the same as \( s \) except for the value at \( X \), for which \( s[X \leftarrow d](X) = d \) holds. Then the interpretation of terms is as follows:

Definition 39.

1. \( \mathcal{I}(X)(s) = s^X \)
2. \( \mathcal{I}(c)(s) = c^X \)
3. \( \mathcal{I}(f(t_1, \ldots, t_k))(s) = f^X(\mathcal{I}(t_1)(s), \ldots, \mathcal{I}(t_k)(s)) \)

For a fixed interpretation \( \mathcal{I} \), we denote by \( \mathcal{I} \) the function \( \mathcal{I} : S \rightarrow \mathcal{D} \) given by the expression \( \mathcal{I}(\bar{t}) \). Now we can define the interpretation of formulas. Let \( A, B \) be subsets of some set \( S \). Then \( A \Rightarrow B \) should denote the set \( (S \setminus A) \cup B \).
Definition 40.

1.
\[ \mathcal{I}(t_1 = t_2) = \{ s \mid t_1(s) = t_2(s) \} \]

2.
\[ \mathcal{I}(R(t_1, \ldots, t_k)) = \{ s \mid R^\mathcal{I}(t_1(s), \ldots, t_k(s)) = \text{true} \} \]

3.
\[ \mathcal{I}(\neg P) = S \setminus \mathcal{I}(P) \]

4.
\[ \mathcal{I}(P_1 \land P_2) = \mathcal{I}(P_1) \cap \mathcal{I}(P_2) \]

5.
\[ \mathcal{I}(P_1 \lor P_2) = \mathcal{I}(P_1) \cup \mathcal{I}(P_2) \]

6.
\[ \mathcal{I}(P_1 \supset P_2) = \mathcal{I}(P_1) \Rightarrow \mathcal{I}(P_2) \]

7.
\[ \mathcal{I}(\exists i P) = \{ s \mid \{ s[i \leftarrow d] \mid d \in D \} \cap \mathcal{I}(P) \neq \emptyset \} \]

8.
\[ \mathcal{I}(\forall i P) = \{ s \mid \{ s[i \leftarrow d] \mid d \in D \} \subseteq \mathcal{I}(P) \} \]

We may also apply the notation \( P^\mathcal{I} \subseteq S \) instead of \( \mathcal{I}(P) \). \( P^\mathcal{I} \) is called the meaning of the predicate \( P \) under the interpretation \( \mathcal{I} \). If \( \mathcal{I} \) is fixed, we simply write \( P \). As an abuse of notation we identify sets of states and their characteristic functions. Thus, \( P^\mathcal{I} \) may also stand for \( P : S \rightarrow \{ \text{true}, \text{false} \} \), where \( P(s) = \text{true} \) iff \( s \in \mathcal{I}(P) \) for a fixed interpretation \( \mathcal{I} \). We say that \( P \) is true for the fixed interpretation \( \mathcal{I} \), if \( P^\mathcal{I} = \text{true} \) for every \( s \in S \). We may apply the notation \( s \models B, s \models p \) and \( s \models P \), if \( B(s), s \in p \) and \( P(s) \) are true, respectively.

The interpretation of a Hoare correctness formula is defined as follows.

Definition 41. Let \( \{ P \} C \{ Q \} \) be a Hoare correctness formula. Then the meaning of the formula is

\[ \{ P \} C \{ Q \}^\mathcal{I} = \mathcal{I}(\{ P \} C \{ Q \}) = \{ P^\mathcal{I} \} C \{ Q^\mathcal{I} \}, \]

which is \( P^\mathcal{I} \subseteq S \times Q^\mathcal{I} \), by Definition 22. We may write \( \models^\mathcal{I} \{ P \} C \{ Q \} \) for \( \mathcal{I}(\{ P \} C \{ Q \}) \), as well. We omit the superscripts \( \mathcal{I} \), if the interpretation is fixed.

It is a natural question to ask whether our deductive system is sound and complete. The soundness of Hoare logic is an easy consequence of Theorem 28, we state it without repeating the proof. The only change is the presence of the assumption that the formulas in \( T \) should hold in the interpretation \( \mathcal{I} \).
Theorem 42. Let \( Th \subseteq HL \), let \( I \) be an interpretation. Assume \( H \) is true in \( I \), if . Then
\[
\vdash_{Th} \{ P \} C \{ Q \} \Rightarrow \{ P \} C \{ Q \}^I.
\]

The case of the completeness assertion is somewhat more elaborate. First of all, observe that in rule \((cons)\) the consequence depends on hypothesis two of which must be provable arithmetical formulas. But, by Gödel’s results, we know that there exist true formulas in usual first order arithmetic (Peano arithmetic) which are not provable. This already gives a boundary of provability in Hoare logic also. Besides this, there emerge other problems, too. Assume by some oracle we know about the truth values of the arithmetical formulas we use in our proofs. In the course of the proof of Theorem 30, for example, we encountered the necessity of finding an assertion \( i \) for which \( \{ p \} C_1 \{ i \} \) and \( \{ i \} C_2 \{ q \} \) hold, given that \( \{ p \} C_1; C_2 \{ q \} \) is true. This could be done by defining \( i \) as \( i = \{ s' \mid \exists s \in p \})((s, s') \in C_1) \}. \) In the case of Hoare logic, for pre- and postconditions of programs only assertions defined by first order Hoare formulas are allowed. Is it the case for every \( B \in Form \) and program \( C \) that we can find a formula \( A \) such that \( A(s) \) is true iff \( (s, s') \in C \) implies \( s' \in B \). In general, the answer is no. But in the case of Peano arithmetic, we can find an affirmative answer, which the next section is about.

8. The wlp-calculus

In the previous section we mentioned that there is a difference between using arbitrary sets of states in the partial correctness assertions or sets of states expressible with first order formulas. If we intend to prove the completeness of the Hoare calculus, we need the ability to express certain assertions by first order formulas. In general, we cannot hope that to every set of states we can assign a formula true at that set. Nevertheless, if we choose Peano arithmetic as our logical formalism, it is capable of describing sets of states the expressibility of which turn out to be sufficient for the completeness of the Hoare calculus. We define below a function \( wlp : P(S \times S) \times P(S) \rightarrow P(S) \), \( wlp(r, q) \) is called the weakest liberal precondition of \( q \) with respect to \( r \).

Definition 43. \( wlp(r, q) = \{ s \mid \forall s'((s, s') \in r \supset s' \in q) \} \).

Lemma 44. \( p \subseteq wlp(C, q) \) iff \( \{ p \} C \{ q \} \).

Thus the weakest liberal precondition of \( q \) with respect to \( C \) is the set of all states \( s \) which end up in \( q \) after the execution of \( C \). Observe that, for an \( s \), being in \( wlp(C, q) \) does not require termination of \( C \) at \( s \). Hence the epithet “liberal”. The interpretation \( I \) is expressive for the underlying logical language, if the set of formulas is rich enough to capture the weakest liberal preconditions of truth sets of formulas. More precisely:

Definition 45. Let \( Form \) be the set of formulas, and \( I \) be an interpretation. We say that \( I \) is expressive for \( Form \), if, for every command \( C \) and every \( B \in Form \), there exists \( A \in Form \) such that
\[
A^I = wlp(C, B^I).
\]

In the remainder of the section we settle the interpretation \( I \) as the standard interpretation of first order arithmetic, and the logical language as the language of Peano arithmetic. Hence, we omit the superscripts standing for the interpretation. We give a sketch of the proof of the expressibility of the usual arithmetical interpretation for arithmetical formulas, since it also sheds light on the properties of the weakest liberal precondition. For the sake of readability, we ignore the distinction between formulas and their truth sets in the argument below.

Lemma 46.

1.
\[
wlp\text{skip}, Q = Q
\]
2. \[ wlp(X := E, Q) = Q[X/E] \]

3. \[ wlp(C_1;C_2, Q) = wlp(C_1, wlp(C_2, Q)) \]

4. \[ wlp(if \ B \ then \ C_1 \ else; C_2 \ fi, Q) = (B \land wlp(C_1, Q)) \lor (\neg B \land wlp(C_2, Q)) \]

**Proof.** The proof goes by induction on \( C \).

1. \( s \in wlp(skip, Q) \iff \langle s, s' \rangle \in skip \implies s' \in Q \) for every \( s' \). But \( \langle s, s' \rangle \in skip \iff s = s' \). By this, \( Q = wlp(skip, Q) \) follows.

2. \( s \in wlp(X := E, Q) \iff \langle s, s' \rangle \in X := E \implies s' \in Q \) for every \( s' \). We have \( \langle s, s' \rangle \in X := E \iff s' = s[X/E] \), which entails, by the substitution lemma, \( Q[X/E] = wlp(X := E, Q) \).

3. \( s \in wlp(C_1; C_2, Q) \iff \langle s, s' \rangle \in C_1; C_2 \implies s' \in Q \) for every \( s' \). Assume \( (C_1; C_2)(s) \) is defined. By Definition 16, there exists \( s'' \) such that \( \langle s, s'' \rangle \in C_1 \) and \( \langle s'', s' \rangle \in C_2 \). Then \( s'' \in wlp(C_2, Q) \), hence \( s \in wlp(C_1; wlp(C_2, Q)) \).

4. \( s \in wlp(if \ B \ then \ C_1 \ else \ C_2 \ fi, Q) \iff \langle s, s' \rangle \in if \ B \ then \ C_1 \ else \ C_2 \ fi \implies s' \in Q \) for every \( s' \). By Definition 16, \( if \ B \ then \ C_1 \ else \ C_2 \ fi = (\langle s, s' \rangle \mid s \in B \land \langle s, s' \rangle \in C_1) \cup (\langle s, s' \rangle \mid s \in \neg B \land \langle s, s' \rangle \in C_2) \).

Assume \( \langle s, s'' \rangle \in if \ B \ then \ C_1 \ else \ C_2 \ fi \) and \( s \in B \). Then \( \langle s, s' \rangle \in C_1 \).

Since, by assumption, \( s \in wlp(C_1, Q) \), we have \( s' \in Q \). Hence \( s \in wlp(if \ B \ then \ C_1 \ else \ C_2 \ fi, Q) \). The same argument applies for the case \( s \not\in B \).

Thus \( (B \land wlp(C_1, Q)) \lor (\neg B \land wlp(C_2, Q)) \subseteq wlp(if \ B \ then \ C_1 \ else; C_2 \ fi, Q) \).

The reverse direction can be proved in an analogous way.

**Lemma 47.** \( wlp(while \ B \ do \ C \ od, Q) \) is expressible.

**Proof.** We give a sketch of the proof. First of all, we have the following relations: \( s \in wlp(while \ B \ do \ C \ od, Q) \iff \langle s, s' \rangle \in while \ B \ do \ C \ od \) and \( s' \in Q \) for every \( s' \in S \). By Lemma 21, there is a finite sequence \( \bar{s}_0, \ldots, s_n \) such that \( s_0 = s \), \( \langle s_i, s_{i+1} \rangle \in C \) and \( s_i \in B \) for every \( 1 \leq i \leq n - 1 \) and \( s_n \in Q \cap \neg B \).

Assume \( B \) and \( C \) have only \( X \) as parameter, the treatment of the general case is similar. It is enough to find a formula \( P \) such that...
While programs

\[ P = \{ s \mid (\exists k_0, \ldots, k_n)(\forall i < n)((s[X/k_i] \in B \land \langle s[X/k_i], s[X/k_{i+1}] \rangle \in C) \land s(X) = k_n)\} . \]

By Gödel’s \( \beta \) predicate we can code finite sequences of numbers by first order formulas, thus the above description can be turned into a first order formula proving the expressibility of the while loop. The interested reader is referred to [3], or [11] for the missing details of the proof.

Expressibility in this sense will be of key importance in the next section when we treat the relative completeness of Hoare’s partial correctness calculus. The weakest liberal precondition is interesting in itself by Lemma 44. The next method will provide us an illustrative way to approximate the weakest liberal precondition for a while loop. Though the result is not presented as a first order formula, it is more applicable in a practical sense when one tries to compute the value of the function \( \text{wlp} \). We introduce a notation for the termination of a computation sequence. If \( C \) is a program, let \( \downarrow C(s) \) denote the fact that the computation sequence of the operational semantics terminates when started with \( \langle C, s \rangle \). In this case we may also say that \( C \) is defined for \( s \). Moreover, let \( \downarrow C = \{ s \mid \downarrow C(s) \} \). With an abuse of notation, we identify below first order formulas with the sets of states they represent.

**Lemma 48.** Let \( \text{while } B \text{ do } C \text{ od} \) be a program, let \( Q \) denote a set of states. Then \( \text{wlp}(\text{while } B \text{ do } C \text{ od}, Q) \) can be approximated iteratively by the union of the following sets of states.

1. \[ P_0 = \neg B \land Q \]
2. \[ P_{i+1} = B \land \text{wlp}(C, P_i) \]

Assume furthermore \( s \in \downarrow \text{while } B \text{ do } C \text{ od} \). Then \( s \in \text{wlp}(\text{while } B \text{ do } C \text{ od}, Q) \) iff \( s \in \bigcup_{i \geq 0} P_i \).

**Proof.** By Definition 51, \( s \in \text{wlp}(\text{while } B \text{ do } C \text{ od}, Q) \) iff \( \langle s, s' \rangle \in \text{while } B \text{ do } C \text{ od} \) implies \( s \in Q \) for every \( s' \in S \). By Lemma 21, \( \langle s, s' \rangle \in \text{while } B \text{ do } C \text{ od} \) iff there are \( s_0, \ldots, s_n \) such that \( s_0 = s \), \( \langle s_i, s_{i+1} \rangle \in C \) and \( s_i \in B \) for every \( 1 \leq i \leq n - 1 \) and \( s_n = s' \in \neg B \). Let this property be denoted by \( (P) \).

\( \langle \leq \rangle \) Assume \( s \in \text{wlp}(\text{while } B \text{ do } C \text{ od}, Q) \). If \( s \in \downarrow \text{while } B \text{ do } C \text{ od} \), then, by the determinism of the operational semantics, there exists \( s' \) with \( \langle s, s' \rangle \in \text{while } B \text{ do } C \text{ od} \). Let \( s_0, \ldots, s_n \) be as above. The proof proceeds by induction on \( n \). If \( n = 0 \), then we have \( s \in \neg B \land Q \). Otherwise, since \( \langle s_0, s_1 \rangle \in C \), by Lemma 21 and property \( (P) \), we can assert \( s_1 \in \text{wlp}(\text{while } B \text{ do } C \text{ od}, Q) \). By the induction hypothesis, \( s_1 \in P_k \) for some \( k \geq 0 \). It follows \( s_0 \in \text{wlp}(C, P_k) \), which, together with \( s_0 \in \neg B \), yields the result.

\( \langle \geq \rangle \) Assume \( s \in \bigcup_{i \geq 0} P_i \), let \( s \in P_j \). We prove the statement by induction on \( j \). If \( j = 0 \), then Lemma 21 immediately gives the result. Otherwise, let \( s \in P_{k+1} \land B \) for some \( k \geq 0 \). By definition, \( s \in \text{wlp}(C, P_k) \). Let \( \langle s, s' \rangle \in \text{while } B \text{ do } C \text{ od} \). Then property \( (P) \) holds for some \( s_0, \ldots, s_n \). Since \( s \in \text{wlp}(C, P_k) \), we also have
While programs

$s_1 \in P_k$. By induction hypothesis, $s_1 \in wlp(\text{while } B \text{ do } C \text{ od}, Q)$, that is, $s' \in Q$. But, together with property $(P)$, this implies $s \in wlp(\text{while } B \text{ do } C \text{ od}, Q)$, as desired.

**Remark 49.** Observe that in the proof above the assumption $s \in \downarrow \text{while } B \text{ do } C \text{ od}$ was used only in one direction. In fact

$$\bigcup_{i \geq 0} P_i \subseteq wlp(\text{while } B \text{ do } C \text{ od}, Q)$$

is valid without any restrictions. For the other direction, however, the assumption is needed, as the following example shows. Let $C = \text{while } \text{true} \text{ do } X := X + 1 \text{ od}$. Let $Q$ be $\text{true}$. Then, obviously, $wlp(C, Q) = \text{true}$. Moreover, by induction on $i$, we can convince ourselves that $P_i = \text{false}$, this means $\bigcup_{i \geq 0} P_i \subseteq wlp(C, Q)$.

### 9. Relative completeness of the Hoare partial correctness calculus

If we consider a sufficiently expressive logical language, for example, the language of first order arithmetic, then, by Gödel’s incompleteness theorems, the consequence rule incorporates incompleteness into Hoare logic. That is, there exist true partial correctness formulas which cannot be deduced by the Hoare partial correctness calculus. By making use of the expressibility result of the previous section, we can prove, however, relative completeness of the Hoare calculus. This means that, if we assume true arithmetical formulas as provable, then every true partial correctness formula is derivable in the Hoare calculus.

**Theorem 50.** Let $\mathcal{I}$ be an interpretation, assume $\mathcal{I}$ is expressive for $\text{Form}$ and $\text{Com}$, where $\text{Form}$ is the set of formulas of the underlying logical language. Let $Th = \{ P \in \text{Form} \mid \mathcal{I}(P) = \text{true} \}$. Then for every $C \in \text{Com}$ and $P$, $Q \in \text{Form}$

$$\{P\}C\{Q\} \quad \Rightarrow \quad \vdash_{Th} \{P\}C\{Q\}.$$

**Proof.** By structural induction on $\{P\}C\{Q\}$.

- If $\{P\} \text{skip}\{Q\}$, then, by Definitions 22 and 16, $P \Rightarrow Q$. The proof of $\vdash_{Th} \{P\}C\{Q\}$ is obtained by applying the consequence rule with $P \Rightarrow P$, $\{P\}C\{P\}$ and $P \Rightarrow Q$.

- $\{P\}X := E\{Q\}$; again, by Definitions 22 and 16, this implies

$$\{s[X/E(s)] \mid s \in P\} \subseteq Q$$. The result follows by applying the consequence rule to $P \Rightarrow P$, $\{P\}X := E\{P[X/E]\}$ and $P[X/E] \Rightarrow Q$.

- $\{P\}C_1; C_2\{Q\}$: then $P \subseteq wlp(C_1; C_2, Q) = wlp(C_1, wlp(C_2, Q))$. By the expressiveness, let $I = wlp(C_2, Q)$ and $J = wlp(C_1, I)$. By induction hypothesis, $\{J\}C_1\{I\}$ and $\{I\}C_2\{Q\}$ are provable. By the composition rule, $\{J\}C_1; C_2\{Q\}$ is also provable, and, taking into account $P \Rightarrow J$ and $Q \Rightarrow Q$, the consequence rule yields the result.

- $\{P\}if \; B \; then \; C_1 \; else \; C_2 \; fi\{Q\}$: then, by Definition 16,

$$\langle s, s' \rangle \in if \; B \; then \; C_1 \; else \; C_2 \; fi \iff s \in B \text{ and } \langle s, s' \rangle \in C_1 \text{ or } s \in \neg B$$
and \( \langle s, s' \rangle \in C_2 \). This means, either \( \{ P \land B \} C_1 \{ Q \} \) or \( \{ P \land \neg B \} C_2 \{ Q \} \)
holds, which, by the induction hypothesis, gives the result.

- \( \{ P \} \text{while } B \text{ do } C \text{ od } \{ Q \} \) : \( \langle s, s' \rangle \in \text{while } B \text{ do } C \text{ od } \) if and only if property \( P \) of
Lemma 48 is true. Let \( I = \text{wp}(\text{while } B \text{ do } C \text{ od } , Q) \) and \( J = \text{wp}(C, I) \).

Assume \( \tau \equiv 0 \) by property \( P \), this implies \( \neg B \subseteq Q \). Otherwise, if \( s_0 \in B \), \( s_1 \in I \)
and \( s_0 \in \text{wp}(C, I) = J \). Since \( s_1 \in I \), by Lemma 44, we have \( I \land B \Rightarrow J \). But
\( \{ J \} C \{ I \} \) holds, as well, which means \( \{ I \land B \} C \{ I \} \). By the while rule we obtain
\( \{ I \} \text{while } B \text{ do } C \text{ od } \{ I \land \neg B \} \). Lemma 44 gives \( P \Rightarrow I \), moreover, by Lemma
21, \( I \land \neg B \subseteq Q \), hence the statement follows.

The question of completeness of Hoare’s calculus with respect to various underlying theories are examined in
detail in [3]. Incompleteness either stems from the weakness of the underlying theory, like in the case of abacus
arithmetic, or, if the theory is strong enough to capture sets of states expressing pre- or postconditions in proofs
of completeness, the new obstacle is raised by Gödel’s theorem saying that there are true formulas unprovable in
Peano arithmetic. Strangely enough, there are even interpretations of arithmetic for which Hoare’s logic is
complete but the interpretations itself are not expressive. The interested reader is referred to [3].

10. The wp-calculus

We turn to the brief discussion of the termination conditions on while programs, prior to this we introduce a
notion very similar to that used in the proof of the completeness of the partial correctness calculus. The function
\( \text{wp} : \mathcal{P}(S \times S) \times \mathcal{P}(S) \rightarrow \mathcal{P}(S) \) called the weakest precondition of \( q \) with respect to \( r \) is defined as
follows.

**Definition 51.** \( \text{wp}(r, q) = \{ s \mid \exists s' (\langle s, s' \rangle \in r \land s' \in q) \} \).

The wp-calculus has very similar properties to the wp-calculus. In this section we gather some of them, and we
return to the wp-calculus in relation to the question of relative completeness in later section. The following
properties hold true, which show a great resemblance to the case of the wp-calculus.

**Lemma 52.**

1. \( \text{wp}(\text{skip}, Q) = Q \)

2. \( \text{wp}(X := E, Q) = Q[X/E] \)

3. \( \text{wp}(C_1; C_2, Q) = \text{wp}(C_1, \text{wp}(C_2, Q)) \)

4. \( \text{wp}(\text{if } B \text{ then } C_1 \text{ else } C_2, f, Q) = (B \land \text{wp}(C_1, Q)) \lor (\neg B \land \text{wp}(C_2, Q)) \)

5. \( \text{wp}(\text{while } B \text{ do } C \text{ od } Q) \land B \Rightarrow \text{wp}(C, \text{wp}(\text{while } B \text{ do } C \text{ od } Q)) \)

6. \( \text{wp}(\text{while } B \text{ do } C \text{ od } Q) \land \neg B \Rightarrow Q \)
Proof. The proofs for $skip$, assignment, conditional statement and composition is similar to that of Lemma 46. We treat only the case of the while loop.

1. Asssume $s \in wp(while \ B \ do \ C \ od, Q) \land B$. Then $s \in B$ and $\exists s'(\langle s, s' \rangle \in while \ B \ do \ C \ od \land s' \in Q)$. By Lemma 21, there exists $s''$ such that $\langle s, s'' \rangle \in C$ and $\langle s'', s' \rangle \in while \ B \ do \ C \ od \land s' \in Q$. This means exactly $s'' \in wp(while \ B \ do \ C \ od, Q)$ and, hence, $s \in wp(C, wp(while \ B \ do \ C \ od, Q))$.

2. Let $s \in wp(while \ B \ do \ C \ od, Q) \land \neg B$. This means $s \notin B$ and $\exists s'(\langle s, s' \rangle \in while \ B \ do \ C \ od \land s' \in Q)$. Again, by Lemma 21, $s \notin B$ and $\langle s, s' \rangle \in while \ B \ do \ C \ od$ entails $s = s'$, which gives the result.

By an argument similar to that of Lemma 48 one can prove a statement for the iterative computation of the weakest precondition for a while loop. Observe that in this case we do not need the assumption $s \in \downarrow (while \ B \ do \ C \ od)$ any more. Without proof, we assert the statement for the wp-calculus mutatis mutandis.

Lemma 53. Let $while \ B \ do \ C \ od$ be a program, let $Q$ denote a set of states. Then $wp(while \ B \ do \ C \ od, Q)$ can be computed iteratively as the union of the following sets of states.

1. $P_0 = \neg B \land Q$

2. $P_{i+1} = B \land wp(C, P_i)$

Then $wp(while \ B \ do \ C \ od, Q) = \bigcup_{i \geq 0} P_i$.

We illustrate by an example below how to compute the weakest precondition of the while loop with the help of Lemma 53.

Example 54. Compute $wp(while \ X \not= 1 \ do \ X := X + 2; Y := X + Y \ od, Y \not= 0)$.
While programs

\[ P_0 \quad = \quad X = 1 \land Y \neq 0 \]

\[ P_1 \quad = \quad X \neq 1 \land wp(X := X + 2; Y := X + Y, X = 1 \land Y \neq 0) \]
\[ = \quad X \neq 1 \land wp(X := X + 2; wp(Y := X + Y, X = 1 \land Y \neq 0)) \]
\[ = \quad X \neq 1 \land wp(X := X + 2, X = 1 \land X + Y \neq 0) \]
\[ = \quad X \neq 1 \land X + 2 = 1 \land X + 2 + Y \neq 0 \]
\[ = \quad X = -1 \land Y \neq -1 \]

\[ P_2 \quad = \quad X \neq 1 \land wp(X := X + 2; Y := X + Y, X = -1 \land Y \neq -1) \]
\[ = \quad X = -3 \land Y \neq 0 \]

\[ P_3 \quad = \quad X \neq 1 \land wp(X := X + 2; Y := X + Y, X = -3 \land Y \neq 0) \]
\[ = \quad X = -5 \land Y \neq 3 \]

\[ P_4 \quad = \quad X \neq 1 \land wp(X := X + 2; Y := X + Y, X = -5 \land Y \neq 3) \]
\[ = \quad X = -7 \land Y \neq 8 \]

\[ \ldots \]

In general, we can prove by induction on \( i \) that
\[ P_i = X = -2(i - 1) + 1 \land Y \neq (i - 1)^2 - 1 \quad \text{if} \quad i \quad \text{is positive}. \quad \text{Thus,} \]
\[ P = X = 1 \land Y \neq 0 \land \exists i(X = -2(i - 1) + 1 \land Y \neq (i - 1)^2 - 1) \]

can be chosen as the weakest precondition.

11. Total correctness

So far we were concerned with proving partial correctness of programs with respect to given specifications. In this section we discuss how to prove termination for while programs. Obviously, only the loop rule needs to be modified as the only source of nontermination. The new loop rule is as follows.

**Definition 55.**

In the above rule \( t \) is an arithmetical expression and \( z \) is a new variable not occurring in \( P, t, B \) or \( C \).

We may as well write the loop rule for total correctness in the form of a proof outline. We resort to Definition 32 when defining proof outlines for total correctness proofs. The only change is the case of the while loop, which is described by the following formation rule.

\[ \begin{array}{c}
[p \land B] C^{**} [p] \\
[p \land B \land t = z]C^{**} [t < z] \\
p \land B \Rightarrow t \geq 0 \\
inv : p \quad \text{while} \quad B \quad \text{do} \quad [p \land B] C^{**} [p] \quad od [p \land \neg B]
\end{array} \quad (\text{tot})\]

As before, \( C^{**} \) is the proof outline corresponding to \( C \), and \( C^{***} \) is a proof outline obtained from \( C^{**} \) by deleting any number of annotations. The annotations with labels \( inv \) and \( bd \) cannot be removed from any
proof outline. The standard proof outlines are defined similarly to the case of partial correctness. As an example, let us prove the correctness of the integer division program below.

**Example 56.** [1] Let $D$ be the program:

\[
Q := 0; \\
R := n; \\
\text{while } R \geq m \text{ do} \\
\quad R := R - m; \\
\quad Q := Q + 1 \\
\text{od}
\]

Let $P = Q \times m + R = n \land R \geq 0 \land m > 0$,

and let $l = R$. We construct a proof outline demonstrating the correctness of the specification $[n \geq 0 \land m > 0]D[Q \times m + R = n \land 0 \leq R < 0]$. 

\[
\begin{align*}
\text{[} n \geq 0 \land m > 0] \\
\text{[} 0 \times m + n = n \land n \geq 0 \land m > 0] \\
Q := 0; \\
\text{[} Q \times m + n = n \land n \geq 0 \land m > 0] \\
R := n; \\
\text{[inv : } Q \times m + R = n \land R \geq 0 \land m > 0], \text{[bd : } R] \\
\text{while } R \geq m \text{ do} \\
\quad [P \land R \geq m] \\
\quad [(Q + 1) \times m + R - m = n \land R - m \geq 0] \\
\quad R := R - m; \\
\quad [(Q + 1) \times m + R = n \land R \geq 0] \\
\quad Q := Q + 1 \\
\quad [Q \times m + R = n \land R \geq 0] \\
\text{od} \\
\quad [P \land R < m] \\
\quad [Q \times m + R = n \land 0 \leq R < 0]
\end{align*}
\]

In order to make it a valid proof outline, we have to prove $P \supset R \geq 0$ and $[P \land R \geq m \land R = z]C^*[R < z]$, where $C$ is the body of the while loop and $C^*$ is
While programs

Moreover, the relations
\[ n \geq 0 \land m > 0 \supset 0 \ast m + n = n \land n > 0 \land m > 0, \]
\[ P \land R \geq m \supset (Q + 1) \ast m + R - m = n \land R - m \geq 0 \quad \text{and} \]
\[ P \land R < m \supset Q \ast m + R = n \land 0 \leq R < m \] need to be shown. But their validity is straightforward to check. The only task remaining os the construction of the proof outline
\[ [P \land R \geq m \land R = z]C^*[R < z]. \]

Since the implication
\[ Q \ast m + R = n \land R \geq 0 \land m > 0 \land R \geq m \land R = z \supset R - m < z \] trivially holds, the proof outline above is a valid proof outline for
\[ [P \land R \geq m \land R = z]C^*[R < z]. \]

Remark 57. The verification of the total correctness formula \([P]C[Q]\) can be split into two, possibly smaller tasks: the verification of \( \{P\}C\{Q\} \) and then the demonstration of \([P]C[true]\). This method is called the decomposition method.

12. Soundness of the Hoare total correctness calculus

As by the semantics of the partial correctness calculus, we can define the meaning of terms and first order formulas for a fixed interpretation \(\mathcal{I}\). As before, if \(P\) is a formula and \(\mathcal{I}\) is the underlying interpretation, then
\[ P^\mathcal{I} = \mathcal{I}(P) \] is the set of all states in which \(P\) is true in \(\mathcal{I}\). We omit the superscript from \(P\).

Definition 58. Let \(\mathcal{P}\) and \(\mathcal{Q}\) be sets of states. Then
\[ [p](q) = p \subseteq C \triangleright q, \]
where, for \(S \subseteq S \times S \) and \(R \subseteq S\), \(S \triangleright R = \{s \mid \exists s'(s, s') \in S \land s' \in R\} \).

We demonstrate below some properties of the relation \([p]C[q]\). Prior to this, we need a lemma which justifies the choice of \(z\) in the total correctness loop rule.

Definition 59. Let \(C\) be a while program, then
\[ \text{change}(C) = \{X \mid X := E \text{ is an assignment in } C\}. \]

Moreover, let \(Var(C)\) denote the set of all variables occurring in \(C\).

Lemma 60.
Let \( C \) be a program and \( s, s' \) be such that \( C(s) = s' \). Then \( s(Y) = s'(Y) \) for any \( Y \).

2. Assume \( s(X) = s'(X) \) for all \( X \in Var(C) \). Then \( C(s)(X) = C(s')(X) \), if \( X \in Var(C) \).

**Proof.** By induction on \( C \).

**Lemma 61.**

1. \( [p]skip[p] \).

2. \( \{ s \mid s[X/E(s)] \in q \} \mid X := E[q] \).

3. \( [p]C_1; C_2[q] \iff [p]C_1[r] \) and \( [r]C_2[q] \) for some \( r \subseteq S \).

4. \( [p]if\ B\ \text{then}\ C_1\ else\ C_2\ fi[q] \iff [p \land B]C_1[q] \) and \( [p \land \neg B]C_2[q] \).

5. \( [p \land B]C[p] \) and \( [p \land B \land t = z]C[t < z] \) and \( B \supseteq t \geq 0 \) together imply \( [p]while\ B\ do\ C\ od[p \land \neg B] \).

6. \( [p']C[q'] \) and \( p \subseteq p' \) and \( q' \subseteq q \) for some \( p' \) and \( q' \iff [p]C[q] \).

**Proof.**

1. By Definition 16, \( \text{skip} = \{ \langle s, s \rangle \mid s \in S \} \).

2. \( X := E = \{ \langle s, s[X/E(s)] \rangle \mid s \in S \} \) by Definition 16. Assume \( s \in \{ s \mid s[X/E(s)] \in q \} \). Then, for \( \langle s, s[X/E(s)] \rangle \), \( s[X/E(s)] \in q \) is trivially true, which gives the result.

3. By Definition 16, \( \langle s, s' \rangle \in (C_1; C_2) \) iff \( \langle s, s' \rangle \in C_1 \odot C_2 \). Let \( r = \{ s'' \mid (\exists s \in p)(\langle s, s'' \rangle \in C_1) \} \).
Then \([p]C_1[r]\) and \([r]C_2[q]\).

The other direction follows from the fact that \(\langle s, s' \rangle \in C_1\) and \(\langle s'', s' \rangle \in C_2\) implies \(\langle s, s' \rangle \in C_1 \circ C_2\).

4. Assume 
\[
[p]i f B t h e n C_1 c l e a r C_2 f i[q] \vee \neg B]C_2
\]
and \(s \in p\). Then \(s \in B\) implies 
\[
\exists s'((s, s') \in C_1),
\]
the case for \(s \notin B\) is equally trivial. As to the other direction, if \(s \in p\) and \(s \in B\), then \(\langle s, s' \rangle \in C_1\) and \(s' \in q\) for some \(s'\). This implies 
\[
\langle s, s' \rangle \in [p]i f B t h e n C_1 c l e a r C_2 f i[q],
\]
and, furthermore, 
\[
\langle s, s' \rangle \in [p]i f B t h e n C_1 c l e a r C_2 f i[q].
\]
The case for \(s \notin B\) is similar.

5. From \([p \land B]C[p]\) and the determinism of computation sequences it follows that 
\[{p \land B}C[p]\], which implies, as in the proof of Theorem 28, 
\[{p}C[p \land \neg B]\]. It is enough to prove that there is no infinite sequence of computation starting from \(\langle \text{while } B \text{ do } C \text{ od}, s \rangle\), if \(s \in p\). Let 
\[
s \models p \land B
\]
such that \(s(t)\) is minimal among the states \(s\) with this property. By 
Definition 12, 
\[
\langle \text{while } B \text{ do } C \text{ od}, s \rangle \rightarrow \langle C; \text{ while } B \text{ do } C \text{ od}, s \rangle.
\]
By hypotheses \([p \land B]C[p]\) and \([p \land B \land t = z]C[t < z]\) and the determinism of computation sequences, we have \(s'\) such that 
\[
\langle \text{while } B \text{ do } C \text{ od}, s \rangle \rightarrow \langle C; \text{ while } B \text{ do } C \text{ od}, s \rangle
\]
and \(s' \models p \land t < z\) holds. If \(s' \notin B\), we are done. Otherwise \(s' \in B\) and, by 
Lemma 60, \(s'(t) < z\) for some \(s'(z) = s(z) = s(t)\) contradicting the assumption on \(s\).

Now we are in a position to define the meaning of a Hoare total correctness formula. Let \(\mathcal{I}\) be the standard interpretation of Peano arithmetic, as before.

**Definition 62.** Let \(P\) and \(Q\) be formulas of Peano arithmetic, let \(C\) be a program. Then the 
meaning of \([P]C[Q]\) with respect to \(\mathcal{I}\), denoted by \([P]C[Q]^{\mathcal{I}}\), is the truth value of 
the expression \([P]C[Q]^{\mathcal{I}}\). We may omit the superscript \(\mathcal{I}\) if it is clear from the context.

Let \(\vdash_{\text{tot}} [P]C[Q]\) denote that \([P]C[Q]\) is derivable in the total correctness calculus. In what follows, 
we prove the soundness of the total correctness calculus. Assume \(\mathcal{I}\) is the usual interpretation of Peano arithmetic.

**Theorem 63.** The Hoare calculus for total correctness of while programs is sound. In 
notation:
\[
\vdash_{\text{tot}} [P]C[Q] \Rightarrow [P]C[Q].
\]

**Proof.** The statement follows from Lemma 61.
As a final remark, since we know now how to interpret total correctness formulas, we would relate the meaning of total correctness formulas to weakest preconditions of programs.

Lemma 64.
\[ [p]C[q] \text{ iff } p \subseteq wp(C, q). \]

13. Relative completeness of the Hoare total correctness calculus

The natural question about the strength of the deduction system arises in this case, too. As in the case of partial correctness, it turns out that the completeness of the calculus has its limitations: there can be several reasons for the calculus to not be complete. Either, the assertions used in the consequence rules cannot be captured by a complete proof system, or not every set of states can be represented by formulas of the language used for describing the assertions and arithmetical expressions of correctness proofs, and finally, the proof system is simply not powerful enough to form a complete deduction system. The same reasoning as in the case of partial correctness shows that the first two obstacles can indeed prevent the deduction system to be complete. By Gödel’s incompleteness theorem the difficulties raised on the axiomatisability of the underlying arithmetical language exist in this case, too. Moreover, we must find a bound function when applying the loop rule for total correctness, and we have to make sure that our language is powerful enough to express these functions as terms. If all these requirements are met, we can prove that the deduction system is strong enough to ensure relative completeness with respect to usual models of arithmetic. The definability of the weakest precondition is again of key importance in the proof. First of all, we mention without proof the expressiveness of the wp-calculus and define the notions necessary for the property.

Definition 65. Let Form be the set of formulas, and \( \mathcal{I} \) be an interpretation. We say that \( \mathcal{I} \) is expressive for Form, if, for every command \( C \) and every \( B \in \text{Form} \), there exists \( A \in \text{Form} \) such that
\[ A^\mathcal{I} = wp(C, B^\mathcal{I}). \]

Without proof we state the following theorem. As before, we set the interpretation \( \mathcal{I} \) as the standard interpretation of Peano arithmetic. The next theorem says that the standard interpretation of arithmetic is expressive for the given language of Hoare logic.

Theorem 66. Let \( C \) be a while program, \( Q \in \text{Form} \). Then there exists \( P \in \text{Form} \) such that
\[ P^\mathcal{I} = wp(C, Q^\mathcal{I}). \]

The proof is very similar to that in the case of the wlp-calculus. The definability of sets of states represented by wlp-expressions were enough to prove relative completeness in the case of partial correctness. However, for total correctness, we need a little bit more: it is also necessary that the bound functions should be expressible in our arithmetical language. First of all, we define some approximation of the bound function for a while loop.

Definition 67. Let \( C = \text{while } B \text{ do } C_1 \text{ od} \), and assume \( X \) is a variable not occurring in \( C \). Let
\[ C_X = X := 0; \text{while } B \text{ do } X := X + 1; C_1 \text{ od}. \]

Define a partial function \( \text{iter} : \text{dom} \text{(iter)} \subseteq S \rightarrow \mathbb{N} \) such that
\[ \text{dom} \text{(iter)} = \{ s \in S \mid \downarrow C(s) \} \text{ and } \text{iter}(s) = C(s)(X), \text{ if } \text{iter} \text{ is defined for } s. \]

The while loop \( C_X \) is called the extended loop of \( C \).

Intuitively, the value \( \text{iter}(C,s) \) supplies us with the number of iterations needed for \( C \) to come to a halt when started from state \( s \), provided \( C \) does not diverge for \( s \).
**Definition 68.** A set of arithmetical expressions is called expressive, if, for every while loop $C$, there is an expression $t$ such that

$$t(s) = \text{iter}(C, s)$$

whenever $C$ is defined for $s$.

Expressibility means the ability to represent the number of iterations needed for a loop in our language of arithmetic. Obviously, addition, multiplication and subtraction are not enough for this purpose. We assume that every partial function computable by a Turing machine is a part of our language, which guarantees expressibility. We can formulate now the result on the relative completeness of the total correctness calculus. We omit the superscripts from the denotation of the sets of states represented by formulas.

**Theorem 69.** Let $\mathcal{I}$ be an interpretation, assume $\mathcal{I}$ is expressive for $\text{Form}$ and $\text{Com}$, where $\text{Form}$ is the set of formulas of the underlying logical language. Let

$$\text{Th} = \{ P \in \text{Form} \mid \mathcal{I}(P) = \text{true} \}.$$  Assume furthermore that the set of arithmetical expressions of the language is expressive. Then, for every $C \in \text{Com}$ and $P, Q \in \text{Form}$,

$$\models [P]C[Q] \Rightarrow \vdash_{\text{Th}} [P]C'[Q].$$

**Proof.** The proof goes by induction on $C$. We discuss only the main ideas of the proof. In what follows, as an abuse of notation let $\wp(C, Q)$ denote the formula expressing $\wp(C, Q)$ itself, when $C$ is a program and $Q$ is an arithmetical formula. We concentrate only on the case of the while loop, since the other cases are almost identical to those in the proof of Theorem 50. Let $C = \text{while } B \text{ do } C_1 \text{ od}$, assume $\models [P]C[Q]$. By Lemma 64,

$$\models [\wp(C_1, \wp(C, Q))]C_1[\wp(C, Q)].$$

The induction hypothesis gives

$$\vdash_{\text{Th}} [\wp(C_1, \wp(C, Q))]C_1[\wp(C, Q)].$$

By Lemma 52, we obtain

$$\vdash_{\text{Th}} [\wp(C, Q) \land B]C_1[\wp(C, Q)].$$

The assumption on the validity of $[P]C[Q]$ implies that $\text{iter}(C, s)$ is always defined whenever $s \in P$. Then the expressiveness gives $a t$ such that $t(s) = \text{iter}(C, s)$, if $\downarrow C(s)$. Let $z$ be a new variable. Then, by Lemma 60 and the fact that $t$ strictly decreases by every iteration of $C$, we obtain

$$\models [\wp(C, Q) \land B \land t = z]C_1[t < z].$$

By the induction hypothesis, we obtain

$$\vdash_{\text{Th}} [\wp(C, Q) \land B \land t = z]C_1[t < z].$$

Moreover, $\wp(C, Q) \supset t \geq 0$, since $t$ is always nonnegative. Then the premisses of the while loop fulfill, hence we can assert

$$\vdash_{\text{Th}} [\wp(C, Q)]C[\wp(C, Q) \land \neg B],$$

which proves our claim.
Chapter 2. Recursive programs

1. Proving partial correctness for recursive procedures

In this chapter we discuss the question of verification of programs containing autonomous substructures called procedures. We augment the language of while programs by syntactical elements enabling incorporation of procedures into the program text. Let $P_1, P_2, \ldots, P_n$ denote a set of procedure names, assume $P_1, P_2, \ldots, P_n$ are variables standing for procedure names. Then a recursive program is a pair standing of a finite set of declarations and the program body itself.

**Definition 70.** $\text{Rec} : \begin{array}{l} P_1 :: C_1; P_2 :: C_2; \ldots; P_n :: C_n; C, \end{array}$

$C = \text{skip} \mid X := E \mid (C_1; C_2) \mid (\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi}) \mid (\text{while } B \text{ do } C \text{ od}) \mid P_n$

We may use the alternative formulation $\text{procedure } P_n; \text{begin } C \text{ end;}$ for $P_n :: C$ when writing concrete programs. The relational semantics of a procedure is a partial function understood as follows: $(s, s') \in P$ iff the procedure $P \in P_n$ started in the state $s$ successfully terminates in state $s'$. This means that the relational semantics of a program can only be defined as a function of the denotational (or relational) semantics of the procedures occurring in it, thus it is a function from $\mathcal{P}(S \times S)$ into $\mathcal{P}(S \times S)$.

Throughout this section we deal with one parameter procedures only. Moreover, in what follows we are concerned mostly with partial correctness questions, we deal with the termination of recursive procedures in the end of the chapter.

**Definition 71.**

1. $\text{skip}(r) = \{(s, s) \mid s \in S\}$
2. $X := E(r) = \{(s, [X/E(s)]) \mid s \in S\}$
3. $C_0; C_1(r) = C_0(r) \circ C_1(r)$, where $\circ$ is the composition of relations
4. $\text{if } B \text{ then } C_0 \text{ else } C_1 \text{ fi}(r) = (B|C_0(r)) \cup (-B|C_1(r))$
5. $\text{while } B \text{ do } C \text{ od}(r) = (B|C(r))^{\ast}[-B$
6. $P(r)$
7. $P :: C_1; C$

We have to convince ourselves that this definition makes sense. We need to establish that the least upper bound of the last point really exists. For this, observe that $\mathcal{P}(S \times S)$ is a complete lattice with respect to set inclusion as ordering. This means that the Kanster–Tarski theorem applies provided $C(r)$ is monotone for arbitrary $C$.

**Lemma 72.** Let $C$ be a program, and $C(\cdot) : \mathcal{P}(S \times S) \rightarrow \mathcal{P}(S \times S)$ be a partial function defined as in Definition 71. Then $C(\cdot)$ is monotone.

**Proof.** A straightforward induction on $C$. 

---

Created by XMLmind XSL-FO Converter.
Thus the least fixpoint required in the definition of the denotational semantics really exists. We state without proof a lemma below, which lies in the basis of reasoning with least fixpoints. This is the so-called computation induction method of Scott and de Bakker ([4]).

**Lemma 7.3.** Let \( L = \langle D, \leq, \cap, \cup, \perp \rangle \) be a complete lattice with bottom element. Let \( f : L \rightarrow L \) be monotone, and let \( P \subseteq D \). Then \( P(\perp), \forall x(P(x) \supset P(f(x))) \) and \( \forall \alpha(\forall x : \alpha \rightarrow D)(\forall \beta < \alpha P(x_\beta) \supset P(\cup_\beta x_\beta)) \) together imply \( P(\text{fix} f) \).

As a special case, we would also state the so-called Scott induction.

**Lemma 7.4.** Let \( F : I \rightarrow L \) be continuous and \( P \subseteq L \) be admissible. That is, for every sequence \( X_n \) from \( L \), if, for all \( n \), \( P(X_n) \), then \( P(\cup_n X_n) \). Then \( P(\perp) \) and \( \forall X(P(X) \supset P(F(X))) \) together imply \( P(\text{fix} f) \).

**Definition 7.5.** Let \( r \in P(S \times S) \), and let \( V \subseteq Var \). We say that \( r \) is without side effects with respect to \( V \) if, for every \( v \in V \), \( (s, s') \in r \) implies \( s(v) = s'(v) \) and, for every \( v \in V \) and \( d \in D \), where \( D \) is the base set of the underlying model, \( (s, s') \in r \) implies \( (s[v/d], s'[v/d]) \in r \). If \( C \) is a program and \( V = Var \setminus Var(C) \), then we say that \( r \) is without side effects with respect to \( C \).

We can formulate the method called computation induction for proving partial correctness of parameterless recursive procedures as in the theorem below.

**Theorem 7.6.** Let \( r \in P(S \times S) \), and \( C \) be a recursive program with one recursive variable. Assume \( r \) is without side effects with respect to \( C \). Then
\[
(\{p\}r\{q\} \Rightarrow \{p\}C\{r\}\{q\}) \Rightarrow \{p\}\text{fix}(C)\{q\}.
\]

Computation induction readily translates into an inference rule: the recursion rule.

**Definition 7.7.** Let \( Pn :: C \) be a procedure, then
\[
\{P\}Pn\{Q\} \vdash \{P\}C\{Q\} \quad (rec)
\]

The rule is understood as follows: assume we have a proof in Hoare logic for \( \{P\}Pn\{Q\} \vdash \{P\}C\{Q\} \). Then we can infer \( \{P\}Pn\{Q\} \) without hypotheses. Hoare logic with the recursion rule is powerful enough to prove statements on parameterless recursive procedures, as the following example shows.

**Example 7.8.** ([3]) procedure F;

begin
  if X=0 then
    Y:=1
  else
    begin X:=X-1; F; X:=X+1; Y:=Y*X end;
  fi
end;
F;
The partial correctness assertion $\{true\} F \{Y = X!\}$ can be proved as follows. We formulate the proof as a proof outline.

$$\{true\} F \{Y = X!\} \text{ (hypothesis)}$$

$$\{true\}$$

\begin{verbatim}
procedure F;
begin
  \{true\}
  if $X = 0$ then
    \{X = 0\}
    \{1 = X!\}
    \{$Y := 1$\}
    \{$Y = X!$\}
  else
    \{X \neq 0\}
    \{true\};
    \begin{verbatim}
    begin X := X - 1;
    \{true\};
    F:
    \{Y = X!\}
    \{$Y \cdot (X + 1) = (X + 1)!$\}
    \{$X := X + 1$\}
    \{$Y \cdot X = X!$\}
    \{Y = Y \cdot X end ;
    \{Y = X!\}
    fi
    \{Y = X!\}
    \end{verbatim}
  end
  \{Y = X!\}
\end{verbatim}
The proof outline demonstrates, from which, by the recursion rule, follows.

The previous proof system based on the recursion rule is not complete for recursive procedures. Consider, for example, the procedure of Example 78 with the precondition \( \{ X = n \} \) and postcondition \( \{ X = n \land Y = n! \} \). Obviously, the partial correctness assertion is true, though it cannot be proved solely with the recursion rule. Firstly, we present without proof a semantical method for verifying partial correctness of recursive programs: the Park’s fixpoint induction method.

**Theorem 79.** Let \( C \) be a program, let \( P, Q \subseteq S \) be sets of states. Then the following statements are valid.

1. \( (\exists r \in \mathcal{P}(S \times S))(\{ p \} r \{ q \} \land C(r) \subseteq r) \Rightarrow \{ p \} lfp(C) \{ q \} \)

2. \( \{ p \} lfp(C) \{ q \} \Rightarrow (\exists r \in \mathcal{P}(S \times S))(C(r) \subseteq r \land \{ p \} r \{ q \}) \land (\forall s, s' \in r)(\forall v \notin Var(C))(s(v) = s'(v) \land \forall d \in D(s[v/d], s'[v/d]) \in r)) \)

**Hint**

1. Observe that from \( C(r) \subseteq r \) it follows that \( r \) is a prefixpoint of \( C(\cdot) \). By the Knaster–Tarski theorem, \( lfp(C) \subseteq r \), from which, combined with \( \{ p \} r \{ q \} \), \( \{ p \} lfp(C) \{ q \} \) immediately follows.

2. If \( \{ p \} lfp(C) \{ q \} \), then, with \( r = lfp(C) \), we have \( \{ p \} r \{ q \} \land C(r) \subseteq r \). Moreover, applying the method of computational induction in Theorem 76, we can prove \( P(lfp(C)) \), where \( P(r) = (\forall s, s' \in r)(\forall v \notin Var(C))(s(v) = s'(v) \land \forall d \in D(s[v/d], s'[v/d]) \in r) \).

We illustrate the proof method by an example.

**Example 80.** (13) Let us consider the program \( Fac \) of Example 78 with the partial correctness assertion \( \{ X = n \} F \{ X = n \land Y = n! \} \). By the application of Point 1 of Theorem 79 we demonstrate that this partial correctness formula indeed holds true. Let \( r = \{ (s, s') \mid s'(X) = s(X) - 1 \land s'(Y) = s(X)! \} \). First of all, we compute \( C(r) \), where \( C \) is the body of \( F \).

\[
\begin{align*}
Y &:= 1(r) = \{ (s, s[Y/0!]) \mid s \in S \} \\
X &:= 0 | Y := 1(r) \subseteq r \\
X &:= X - 1(r) = \{ (s, s[X/s(X) - 1]) \mid s \in S \} \\
F(r) &= r \\
X &:= X - 1; F(r) = \{ (s, s') \mid s'(X) = s(X) - 1 \land s'(Y) = (s(X) - 1)! \} \\
X &:= X + 1(r) = \{ (s, s[X/s(X) + 1]) \mid s \in S \} \\
X &:= X - 1; F; X := X + 1(r) = \{ (s, s') \mid s'(X) = s(X) \land s'(Y) = (s(X) - 1)! \}
\end{align*}
\]
Recursive programs

\[
Y := Y \times X(r) = \{ (s, s') | s'(Y) = s(Y) \times s(X) \}
\]

\[\neg X = 0 \lor X := X - 1; F; X := X + 1; Y = Y \times X(r) \subseteq r\]

\[C(r) \subseteq r\]

Moreover, \(\{ X = n \} r \{ X = n \land Y = n! \}\) obviously holds, thus \(\{ X = n \} \text{if} p(C) \{ X = n \land Y = n! \}\) is valid, too.

The previous version of the fixpoint induction theorem is not expressible in Hoare logic, since it uses relation like \(C(r) \subseteq r\), which is cannot be corresponded to the usual forms of formulas of Hoare logic. A solution to this problem is to add auxiliary variables to the correctness proofs of recursive programs, which are able to store the values of the program variables before the procedure application. We state a different version of fixpoint induction theorem, with the necessary definitions prior to it.

**Definition 81.** Let \(X^\sim = \{ X_1, \ldots, X_n \}\), \(x^\sim = \{ x_1, \ldots, x_m \}\). Then the sets of vectors \(X^\sim\) and \(x^\sim\) are said to be apart (in notation: \(X^\sim \parallel x^\sim\), if \(n = m\) and \(\{ X_1, X_2, \ldots, X_n \} \cap \{ x_1, x_2, \ldots, x_n \} = \emptyset\).

The next theorem is a version of the fixpoint induction theorem, formulated with auxiliary variables. The proof is rather technical, hence omitted here.

**Theorem 82.** Let \(C\) be a program, let \(P, q \subseteq S\) be sets of states. Let \(X^\sim = V a r(C)\) and \(X^\sim \parallel x^\sim\). Then

\[
(\exists r \in P(S \times S))(C(r) \subseteq r \land \{ p \} r \{ q \})
\]

iff

\[
(\exists p', q' \in P(S))(\forall r' \in P(S \times S))(\forall (s, s') \in r')
\]

\[
(\forall v \notin V a r(C))(s(v) = s'(v) \land \forall d \in D(s[v/d], s'[v/d]) \in r) \supset
\]

\[
(\{ p' \} r'[q']) \supset (\{ p \} C(r')\{ q' \})
\]

\[
(\forall s \in p)(\forall d \in D^\sim)((\forall s' \in S)(s'[X^\sim /s(X^\sim)] \in p') \supset s'[X^\sim /d^\sim] \in q') \supset
\]

\[
s[X^\sim /d^\sim] \in q)
\]

The previous theorem directly gives a proof rule for the Hoare calculus, the so-called rule of adaptation.

**Definition 83.**

\[
\{ P' \} C \{ Q' \} / \{ \forall x^\sim (\forall y^\sim (P' \supset Q'[X^\sim /x^\sim]) \supset Q[X^\sim /x^\sim])) \} C \{ Q \}, \text{ (rule of adaptation)}
\]

where \(Fv(C) \subseteq X^\sim\), \(X^\sim \parallel x^\sim\), \(x^\sim \cap Fv(P', Q', C, Q) = \emptyset\) and \(y^\sim = Fv(P', Q') \setminus X^\sim\).

In what follows, we illustrate by some examples the proof method for partial correctness of recursive program by applying the rule of adaptation. The proof of Theorem 82, together with a historical background of the development of the adaptation rule, can be found in ([13]).

**Example 84.** ([11]) Let \(F\) be the command defined in Example 78. We prove now \(\{ X = n \} F \{ X = n \land Y = n! \}\) with the help of the adaptation rule. We present the proof in the form of a proof outline.

\[
\{ X = n \} F \{ X = n \land Y = n! \} \text{ (hypothesis)} \text{ procedure } F;
\]
Recursive programs

begin
\{X = n\} 

if X=0 then
\{X = n \land X = 0\}
\{X = n \land 1 = n!\}
Y:=1
\{X = n \land Y = n!\}

else
\{X = n \land X \neq 0\}
X:=X-1;
\{X = n - 1 \land X + 1 \neq 0\}
\{X = n - 1\}
F;
\{X + 1 = n \land Y \ast (X + 1) = n!\}
X:=X+1;
\{X = n \land Y \ast X = n!\}
Y:=Y \ast X
\{X = n \land Y = n!\}
fi
\{X = n \land Y = n!\}
end;
\{X = n\} F\{X = n \land Y = n!\}

We have to prove \(\{X = n - 1\} F\{X + 1 = n \land Y \ast (X + 1) = n!\}\) from the hypothesis \(\{X = n\} F; \{X = n \land Y = n!\}\). Applying the rule of adaptation with \(x^\sim = \{x, y\}\), \(y^\sim = \{n\}\), we obtain
\(\forall xy(\forall n(X = n \supset x = n \land y = n!) \supset x = n - 1 \land y \ast (x + 1) = n!)\) \(F\{X = n - 1 \land Y \ast (X + 1) = (n - 1)!\}\)

Furthermore,
\(X = n - 1 \supset \forall xy(\forall n(X = n \supset x = n \land y = n!) \supset x = n - 1 \land y \ast (x + 1) = n!)\)

which, together with the consequence rule, yields the result. We can now derive by the recursion rule \(\{X = n\} F; \{X = n \land Y = n!\}\).

Example 85. (??) The program below supplies the value \(2^n - 1\), if \(n_0 > 0\).
\(\{N = n \geq 0 \land S = s\} F\{N = n \geq 0 \land S = s + 2^n - 1\}\)

procedure F;
begin

\{ N = n \geq 0 \land S = s \} \\
\text{if } N > 0 \text{ then} \\
\{ N = n > 0 \land S = s \} \\
N := N - 1; \\
\{ N = n - 1 \geq 0 \land S = s \} \\
F; \\
S := S + 1; \\
\{ N = n - 1 \geq 0 \land S = s + 2^{n-1} \} \\
F; \\
\{ N = n - 1 \geq 0 \land S = s + 2^{n-1} + 2^{n-1} - 1 \} \\
N := N + 1; \\
fi; \\
\{ N = n \geq 0 \land S = s + 2^n - 1 \} \\
end;

\{ N = n_0 \geq 0 \land S = s \} F \{ N = n_0 \geq 0 \land S = s + 2^{n_0} - 1 \}

We have to fill in the gaps in the proof by applying the adaptation rule for every call of \( F \). Moreover, the derivability of the conclusion follows only by taking into account the recursion rule, as well. For example, if

\{ N = n \geq 0 \land S = s \} F \{ N = n \geq 0 \land S = s + 2^n - 1 \}

holds, then

\{ \forall m, (\forall n, (N = n \geq 0 \land S = s \lor n' = n \geq 0 \land s' = s + 2^{n-1} - 1) \lor n' = n - 1 \geq 0 \land s' = s + 2^{n-1} - 1) \} \Rightarrow \{ N = n - 1 \geq 0 \land S = s + 2^{n-1} - 1 \},

by taking into account the relation

\( N = n - 1 \geq 0 \land S = s \lor N = n \geq 0 \land S = s + 2^{n-1} - 1 \). 

and the consequence rule.

2. **Total correctness of recursive procedures**

The presence of recursive calls means another source of nontermination: it may happen that the program entangles into an infinite sequence of recursions. The easiest such example is:

**Definition 86.** ([5]) procedure Diverge; begin diverge end; diverge

Of course, in most of the cases, nontermination is not so obvious. Consider the example below:

**Definition 87.** ([5]) procedure D;

begin \\
if X = 0 then
Recursive programs

The procedure \( D \) satisfies the specification \( \{ X = x \geq 0 \} D \{ X = 2x \} \). We can observe that it even terminates when started from a value \( X \geq 0 \). If \( X = 0 \), then termination is immediate. Otherwise, assume \( X = n \geq 0 \) and we know that \( D \) terminates for all values \( m < n \). Then the assignment before the recursive call reduces the value of \( X \), and for the new value termination is established. Thus the program terminates for \( X \), too. The argument above can be formalized as a deduction rule, this is the recursion rule for total correctness.

**Definition 88.** Let \( P; n : C \) be a procedure, then

\[
\frac{P \land t < n \} P[n][Q] \vdash [P \land t = n \} C[Q], \quad P \supset t \geq 0}{P[n][Q]}, \quad \text{(totalrec)}
\]

where \( t \) is an arithmetical expression and \( n \) is a variable not occurring in \( P, Q, C \) and \( t \). Moreover, \( n \) is treated in the proof as a constant, which means \( n \) can neither be substituted for, nor can it be quantified by an existential quantifier.

**Remark 89.** Adopting the discussion of [1], we give a short justification of the necessity of the restrictions for the variable \( n \) in the previous definition. Let \( P; P \) be a recursive program. Obviously, \( P; P \) never halts. Assume we were allowed to quantify \( n \) with an existential quantifier in the premiss. Then we would obtain the following contradiction. Let the bound function \( t \) be \( X \), if \( [X \geq 0 \land t < n \} P[true] \) is our hypothesis in the adaptation rule \( (totalrec) \). Then, if existential quantification of \( n \) is allowed, we acquire \( [\exists n(X \geq 0 \land t < n \} P[true] \). But \( \exists n(X \geq 0 \land t < n \) is equivalent to \( X \geq 0 \), from which, applying the consequence rule again, we obtain \( \{ X \geq 0 \land t = n \} P[true] \), which is impossible.

On the other hand, let \( [X \geq 0 \land t < n \} P[true] \) be given. Assume \( X + 1 \) is substituted for \( n \). Then we obtain \( [X \geq 0 \land t < n \} P[true] \), and \( [X \geq 0 \land X < X + 1 \} P[true] \). The premiss in the former formula is equivalent to \( [X \geq 0 \} P[true] \), from which, by the consequent rule, \( [X \geq 0 \land t = n \} P[true] \) follows. Hence, though \( n \) being a variable, it must be treated as a constant in the course of the proof.

The Hoare calculus for the total correctness of recursive programs can be obtained if we substitute the recursion rule for the above recursion rule in the Hoare calculus for partial correctness of recursive programs with all the notations of the form \( \{ P \} C \{ Q \} \) replaced by the forms \( [ P \} C \{ Q \} \). We remark that soundness and relative completeness results similar to those in the case of while programs hold in this case, too. We end up this chapter with a proof of a total correctness specification for the recursive program of Example 87.

**Example 90.** \( [X = n \geq 0 \land X < N \} D \{ X = 2n \} \) (hypothesis)

procedure \( D \);

\begin{verbatim}
skip
else
X:=X-1
D;
X:=X+2
fi; end;
\end{verbatim}

The Hoare calculus for the total correctness of recursive programs can be obtained if we substitute the recursion rule for the above recursion rule in the Hoare calculus for partial correctness of recursive programs with all the notations of the form \( \{ P \} C \{ Q \} \) replaced by the forms \( [ P \} C \{ Q \} \). We remark that soundness and relative completeness results similar to those in the case of while programs hold in this case, too. We end up this chapter with a proof of a total correctness specification for the recursive program of Example 87.
if X=0 then

\[ X = n \geq 0 \land X = N \land X = 0 \]

[X=2n]

skip

[X=2n]

else

\[ X = n > 0 \land X = N \]

X:=X-1

\[ X = n - 1 \geq 0 \land X < N \]

D;

\[ X = 2(n - 1) \]

X:=X+2

\[ X = 2n \]

fi;

\[ X = 2n \]

end;

The step \[ X = n - 1 \geq 0 \land X < N \mid D[ X = 2(n - 1) ] \] needs to be established with the help of the adaptation rule. By the rule of adaptation, we have

\[ \forall x \forall n, N(x = n \geq 0 \land X < N \supset x = 2n) \supset x = 2(n - 1) \mid C[ X = 2(n - 1) ] \]

which, together with the consequence rule applied with

\( X = n - 1 \geq 0 \land X < N \land \forall x \forall n, N(X = n \geq 0 \land X < N \supset x = 2n) \supset x = 2(n - 1) \)

gives the result.

**Remark 91.** A decomposition strategy also works in this case. For the statement \[ P \mid C\{ Q \} \] it is enough to establish \( \{ P \} C\{ Q \} \) and then prove \( P \mid C\{ true \} \).
Chapter 3. Disjoint parallel programs

In this, and the subsequent chapters we glance through the main types of parallel programs following the monograph of Apt and de Boer and Olderog ([1]). Undoubtedly, the simplest appearance of parallelism manifests itself in the field of disjoint parallel programs. Disjoint parallel programs are programs the components of which cannot change the variables used by each other, that is, they can only share read-only variables. Let us recall: a variable is changed in a program if occurs on the left hand side of an assignment command in the program. The set of variables changed by $S$ was denoted by $\text{change}(S)$. In accordance with this, we can define the disjointness of two programs.

**Definition 92.** Let $S_1$ and $S_2$ be two programs. $S_1$ and $S_2$ are called disjoint iff

$\text{change}(S_1) \cap \text{var}(S_2) = \emptyset$

and

$\text{var}(S_1) \cap \text{change}(S_2) = \emptyset$.

**Example 93.** Let $S_1 = X := Z$ and $S_2 = Y := Z$. Then $\text{change}(S_1) = \{X\}$ and $\text{change}(S_2) = \{Y\}$, this means $S_1$ and $S_2$ are disjoint. Thus, disjoint programs can read the same variables.

Let us give the precise definition of disjoint parallel programs.

**Definition 94.** A program $D$ is a disjoint parallel program if it is formed by the Backus–Naur form as below.

$$
C = \text{skip} \mid X := E \mid (C_1; C_2) \mid (\text{if } B \text{ then } C_1 \text{ else } C_2 \text{ fi}) \mid (\text{while } B \text{ do } C \text{ od})
$$

$$
D = [C_1 \parallel C_2 \parallel \ldots \parallel C_n] \parallel D_1 \parallel D_2 \parallel (\text{if } B \text{ then } D_1 \text{ else } D_2 \text{ fi}) \parallel (\text{while } B \text{ do } D \text{ od})
$$

Thus, parallel composition can be formulated with while programs only, hence nested parallelism is not allowed. Parallelism is allowed, however, in sequential composition, conditional statements and while loops. If $S = [S_1 \parallel S_2 \parallel \ldots \parallel S_n]$, then $S$ is called a disjoint parallel composition, while $S_i$ are called the components of $S$. In a manner analogue to the definition of disjointness of programs, it can be useful to talk about a program being disjoint or apart from an expression of set of states expressed by a first order formula.

The operational interpretation of a disjoint parallel composition is rather straightforward.

**Definition 95.**

$$
\langle S_i, s \rangle \rightarrow \langle T_i, r \rangle
$$

$$
\langle [S_1 \parallel S_2 \parallel \ldots \parallel S_n], s \rangle \rightarrow \langle [T_1 \parallel T_2 \parallel \ldots \parallel T_n], r \rangle,
$$

if $1 \leq i \leq n$.

A notational convention can be agreed when producing computations of disjoint parallel programs: the program $[E \parallel E \parallel \ldots \parallel E]$ should be denoted by $E$, and it represents the end of a (terminating) computation. The absence of blocking lemma also holds in this case, mutatis mutandis.

**Lemma 96.** For any disjoint parallel program $S$ and any state $s$ there is a unique element $\gamma \in \Gamma = (\text{Com} \times S) \cup S$ such that $\langle C, s \rangle \rightarrow \gamma$.

In general, determinism is not valid in this case. We can still say something, however, on the result of a computation. We state without proofs two lemmas which assert properties of disjoint parallel programs interesting in themselves.
Definition 97. Let \((\mathcal{R}, \rightarrow)\) be a reduction system, that is, \(\mathcal{R}\) is a set and \(\rightarrow \subseteq \mathcal{R} \times \mathcal{R}\) is a relation over \(\mathcal{R}\) (we can assume now that there is only one relation over \(\mathcal{R}\)). Let \(\rightarrow\) be the reflexive, transitive closure of \(\rightarrow\). Then \(\mathcal{R}\) is said to satisfy the diamond property, if, for every \(a, b, c \in \mathcal{R}\), with \(a \rightarrow d\) and \(a \rightarrow^* c\), the relations \(b \rightarrow d\) and \(b \rightarrow^* c\) imply that there exists \(d' \in \mathcal{R}\) such that \(a \rightarrow^* b\) and \(a \rightarrow^* c\). The relation \(\rightarrow\) is said to be confluent, if, for every \(a, b, c, d \in \mathcal{R}\), \(a \rightarrow d\), \(b \rightarrow d\), and \(c \rightarrow d\) imply \(a \rightarrow c\) and \(b \rightarrow c\) for some \(d\).

There is a close connection between diamond property and confluence as Newman’s lemma below states.

Lemma 98. Let \((\mathcal{R}, \rightarrow)\) be a terminating reduction system. Then, if \(\rightarrow\) satisfies the diamond property, then it is confluent.

Proof. The proof is very well-known in the literature, we just give a sketch of it here. Figure 3.1 gives a clear insight into the main idea of the proof.

Firstly, let \(\rightarrow^n\) denote the \(n\)-fold relation obtained from \(\rightarrow\). By \(a \rightarrow^0 b\) we understand \(a = b\). By induction on \(n\) we can prove that, if \(a \rightarrow^n b\) and \(a \rightarrow c\) with \(n \geq 1\), then there exists \(d, e\) such that \(c \rightarrow \rightarrow^{n-1} d\) and, moreover, \(d \rightarrow c\) and \(b \rightarrow c\) are valid.

Let us assume \(a \rightarrow^* b\) and \(a \rightarrow^* c\). By the finiteness of \(\rightarrow\), \(a \rightarrow^n b\) and \(a \rightarrow^m c\) for some \(n, m \geq 0\). Assume \(n \geq 1\) and \(a \rightarrow a' \rightarrow^{n-1} b\). By the proof above, there exists \(c'\) with \(a \rightarrow^m c \rightarrow c'\) such that \(a \rightarrow a' \rightarrow^m c'\). Then, for \(a' \rightarrow^{n-1} b\) and \(a' \rightarrow^m c'\), the induction hypothesis applies.

![Figure 3.1. The Church–Rosser property](attachment:image.png)

We remark that Newman’s lemma is not valid in this simple form if we have several, possibly overlapping, reductions in the reduction system (cf. ([10])). The following lemma treats the case when \((\mathcal{R}, \rightarrow)\) is not finite.

Lemma 99. Assume \((\mathcal{R}, \rightarrow)\) is a reduction system which satisfies the diamond property. Let \(a \rightarrow b\) and \(a \rightarrow c\) such that \(b \neq c\). Assume \(a \rightarrow b \rightarrow b_1 \rightarrow \ldots \rightarrow b_i \rightarrow \ldots\) for some infinite reduction sequence. Then there exists an infinite reduction sequence \(a \rightarrow c \rightarrow \ldots \rightarrow \ldots\) passing through \(c\).

Without proof state the following lemma, called the yield lemma.
Disjoint parallel programs

**Lemma 100.** Let $(\mathcal{R}, \rightarrow)$ be a reduction system satisfying the diamond property. Let $a$ be a reduction sequence starting from $a$ is infinite, or there exists $b$ such that and, for every $c$, is not valid.

We can apply these results to disjoint parallel programs, as the following lemma shows.

**Lemma 101.** Let $D_1, D_2$ be disjoint parallel programs, assume $\langle D_1, s_1 \rangle \rightarrow \langle D_2, s_2 \rangle$ is the relation defined in the operational semantics. Then the reduction system $\{\langle D, s \rangle \mid D \text{ is a disjoint parallel program, and } s \in \mathcal{S} \}, \rightarrow$ satisfies the diamond property.

As a corollary, we can express Lemma 100 in terms of disjoint parallel programs.

**Corollary 102.** Let $D$ be a disjoint parallel program, and assume $s \in \mathcal{S}$. Then either every computation starting from $\langle D, s \rangle$ is infinite, or there exists a unique state $r$ such that $\langle D, s \rangle \rightarrow^* \langle E, r \rangle$.

Obviously, the sequential execution of the parallel components leads to a computation of a parallel construction. Furthermore, Corollary 102 involves that it is enough to implement a computation like this in order to get the result of a computation making use of the parallel components in any order, if it exists. Thus the following sequentialization lemma is true.

**Lemma 103.** Let $S_1, S_2, \ldots, S_n$ be pairwise disjoint while programs. Then, for every state $t$, $[S_1 \parallel S_2 \parallel \ldots \parallel S_n](t)$ is defined iff $(S_1; S_2; \ldots; S_n)(t)$ is defined, and, in this case,

$$[S_1 \parallel S_2 \parallel \ldots \parallel S_n](t) = (S_1; S_2; \ldots; S_n)(t).$$

From this result a suggestion for a rule for the partial correctness calculus of disjoint parallel programs readily emerges:

$$\frac{\{p\} S_1; S_2; \ldots; S_n \{q\}}{\{p\} [S_1 \parallel S_2 \parallel \ldots \parallel S_n] \{q\}} \quad (\text{sequent})$$

The sequentialization rule states that in order to prove a partial correctness assertion for a parallel composition, it is enough to prove the same assertion for the sequential analogue of the parallel program. The sequentialization rule seems to be a good candidate at first glance, though we would like to have something more handy, which does not make it necessary to introduce a huge number of intermediate assertions in the course of the proof. The disjoint parallelism rule was proposed by Hoare:

$$\frac{p_i}{{\Lambda}_{i=1}^n p_i} [S_1 \parallel S_2 \parallel \ldots \parallel S_n] [{\Lambda}_{i=1}^n q_i], \quad (dp)$$

where $Fv(p_i) \cup Fv(q_i) \cap \text{change}(S_j) = \emptyset$, if $i \neq j$. The stipulations in the disjoint parallel rule are absolutely necessary in order to avoid unsound conclusions.

**Example 104.** ([1]) Let $S = X := 0 \parallel Y := 0$.

Then $\{Y = 1\} X := 0 \{Y = 1\}$ and $\{true\} Y := 0 \{true\}$, which would result in the false conclusion $\{Y = 1\} [X := 0 \parallel Y := 0] \{Y = 1\}$, if we ignored the stipulations of the disjoint parallelism rule.
We quickly encounter the delimitations of using the disjoint parallel rule alone as a rule of inference for disjoint parallel programs. It can be shown, for example, (see ([1])) that the correctness formula

\[ \{X = Y\} [X := X + 1 || Y := Y + 1] \{X = Y\} \]

cannot be proven with the disjoint parallelism rule alone. Instead, the following trick helps us out.

**Example 105.** ([1]) Take a fresh variable \(Z\). Then

\[ \{X = Z\} \{X := X + 1\} \{X = Z + 1\} \]

and

\[ \{X = Z\} \{Y := Y + 1\} \{X = Z + 1\} \]

. The disjoint parallelism rule and the consequence rule gives

\[ \{X = Z \land Y = Z\} [X := X + 1 || Y := Y + 1] \{X = Y\} \]

But there is a slight problem: \(X = Z \land Y = Z\) does not follow from \(X = Y\), thus, we

are not entitled to infer

\[ \{X = Y\} [X := X + 1 || Y := Y + 1] \{X = Y\} \]

by referring to the consequence rule. We have, however,

\[ \{X = Y\} Z := X \{X = Z \land Y = Z\} \]

which means

\[ \{X = Y\} Z := X; [X := X + 1 || Y := Y + 1] \{X = Y\} \]

The moral is: introducing auxiliary variables may help the proof, but how can we get rid of the auxiliary variables afterwards? First of all, what can be considered as an auxiliary variable:

**Definition 106.** Let \(\mathcal{A}\) be a set of variables of a program \(S\). Then \(\mathcal{A}\) is a set of auxiliary variables, if every occurrence of a variable \(X \in \mathcal{A}\) lies within an assignment statement \(Z := t\), with some \(Z \in \mathcal{A}\) and expression \(t\).

**Example 107.** Let \(S = \text{if } Z := X \text{ then } X := X + 1 || Y := U \text{ else } X := Y\).

Then the following sets are the sets of auxiliary variables for \(S\): \(\emptyset\), \(\{Z\}\), \(\{X, Z\}\), \(\{X, Y, Z\}\), \(\{X, Y, Z, U\}\).

With this in hand we can formulate the missing rule for the partial correctness calculus of disjoint parallel programs: the auxiliary variable rule.

\[
\frac{\{p\} S\{q\}}{\{p\} S_0\{q\}} \quad (aux)\]

where \(F_0(q) \cap \mathcal{A} = \emptyset\) and \(S_0\) is obtained from \(S\) by writing \(skip\) in place of every assignment to a variable in \(\mathcal{A}\). If we identify programs with the programs obtained by dropping some of the \(skip\) commands from them, we can prove the partial correctness formula of Example 105 with the new set of rules. We remark that when formulating proof outlines, it must stated explicitly which variables are understood as auxiliary variables and which are not. The formation rule of a proof outline involving a parallel command is as follows:

\[
\frac{}{\{p_1\} S_1^* \{q_1\} \mid \{p_2\} S_2^* \{q_2\} \mid \ldots \mid \{p_n\} S_n^* \{q_n\} } \mid \{\bigwedge_{i=1}^{n} p_i\}\} \}
\]
where \( \{p_i\} S_1^\ast \{q_i\} \) are proof outlines corresponding to \( \{p_i\} S_i \{q_i\} \).

Finally, we remark that the calculus obtained by augmenting the total correctness calculus with the rules \((dp)\) and \((aux)\) is appropriate for proving total correctness assertions for disjoint parallel programs. It can be proved that both the partial and total correctness calculus is sound and admits a relative completeness property like in the case of while-programs.
Chapter 4. Shared variable parallelism

1. The syntax of shared variable parallel programs

Disjoint parallelism appeared to be a restricted form of parallelism, it is closer to applications if we assume that the components of a parallel programs have resources, messages, data, etc. to share. Shared variable parallelism means that there can be common variables which can be modified by several components of the program, accomplishing in this way a method of data exchange and communication. The design of shared variable parallel programs needs more care than that of disjoint parallel programs: parallel components can interfere now with each other, and some of these actions may be unwanted. The introduction of the notion of atomicity handles this phenomenon. This interference also has its effect on the device of inference rules, we have to make sure by parallel execution that none of the components invalidates the inferences made for the other components, to this end we stipulate the property interference freedom for the parallel components. To illustrate the problem mentioned in this paragraph we consider the simple example below ([1]). The syntax of shared variable parallel programs is the same as that of disjoint parallel programs, except for some minor changes which we are going to introduce later.

Example 108. Let

\[
S_0 = X := 0 \\
S_1 = X := 1; X := 1 \\
S_2 = X := 2
\]

Then the value of \( X \) upon termination of \( [S_1 || S_0] \) can be either 0 or 2. On the other hand, the value of \( X \) when \( [S_2 || S_0] \) has terminated can either be 0, or 1, or 2. The difference can be attributed to the fact that although \( S_1 \) and \( S_2 \) behave the same as parallel programs, when composed with \( S_0 \) the execution of \( S_1 \) can be interrupted while the execution of \( S_2 \) cannot.

A solution for this problem is the introduction of the notion of atomicity. We consider a component indivisible or atomic, if, during the execution of which, the other components may not change its variables. Thus a parallel computation in this setting can be considered as a subsequent execution of atomic actions lying in the various components. The execution process is asynchronous: no assumption is made about the time needed by the components for the execution of the atomic actions. Syntactically, we distinguish atomic actions by angle brackets.

Definition 109. Let \( S_0 \) be a loop-free program and without inner atomic regions. Then \( \langle S_0 \rangle \) is the atomic region consisting of \( S_0 \).

The formation rules for parallel programs are thus augmented with the rule for forming atomic regions. As before, besides explicitly defined atomic regions we consider boolean expression and \( \text{skip} \) and assignments as atomic steps. A subprogram of \( S \) is called normal, if it does not occur within an atomic region of \( S \). In what follows, by parallel program we mean shared variable parallel program unless stated otherwise.

2. Operational semantics

The semantics for the atomic region reflects our intention to treat atomic regions as uninterruptable. This is achieved by the transition rule below.

\[
\langle S, s \rangle \rightarrow^* \langle E, r \rangle \\
\langle \langle S \rangle, s \rangle \rightarrow^* \langle \langle E \rangle, r \rangle
\]
Shared variable parallelism

To interpret the execution of atomic regions as one-step program executions serves the aim of preventing other subprograms for intervening in the computation of the atomic region. The properties of absence of blocking and of determinism stated in the previous chapter can be stated only with modifications here. The absence of blocking remains valid however.

**Lemma 110.** Let $S$ be shared variable parallel program different from the empty program, let $s \in \mathcal{S}$. Then there exists $S'$ and $s'$ such that $(S, s) \rightarrow (S', s')$.

Contrary to the previous chapter we do not have the disjointness condition now. This suggests that the Church-Rosser property is not valid in this case, as the following trivial example already shows this.

**Example 111.** Consider the programs of Example 108. Let $s(X) = 0$. Then there are $s'$ and $s''$ such that

\[
\langle [S_0 | S_0], s \rangle \rightarrow^* \langle E, s' \rangle
\]

with $s'(X) = 0$ and

\[
\langle [S_0 | S_2], s \rangle \rightarrow^* \langle E, s'' \rangle
\]

with $s''(X) = 2$. Thus $\rightarrow$ is not confluent.

A statement asserting a property little weaker than nondeterminism can be formulated in this case, though. Without proofs we assert some lemmas and, finally, the lemma of bounded nondeterminism as a corollary of them.

**Lemma 112.** (Finiteness) For every parallel program $S$ and state $s$, the configuration $(S, s)$ has only finitely many successors.

The proof hinges on a straightforward case distinction with respect to the formation of $S$.

**Definition 113.** A tree is finitely branching of every node has only finitely many successors.

Thus, Lemma 112 states that for every program $S$ and state $s$ the tree depicting the computations starting from $(S, s)$ is finitely branching. The next lemma is the well-known König’s lemma.

**Lemma 114.** (König) Every finitely branching tree is either finite or possesses an infinite branch.

From this, and Lemma 112 we obtain the following result.

**Lemma 115.** (Bounded nondeterminism) Let $S$ be a parallel program and $s \in \mathcal{S}$. Then either the tree representing all the computations starting from $(S, s)$ is finite or it has an infinite branch.

3. Deduction system

We expect by the discussion above that the new syntactic element, the atomic region, can be handled in a straightforward manner in a calculus aiming to treat partial or total correctness formulas. This is indeed the case, as the rule of atomic regions defined below shows.

\[
\begin{align*}
(p) S (q) & \quad \text{(atomic)} \\
\{p\} \{S\} \{q\} & 
\end{align*}
\]

As to the proof outlines, the atomic region rule can be transformed without problem.
where \( S^* \) is a proof outline obtained from \( S \).

The situation becomes more complicated when we consider the case of parallel composition. In the shared variable case parallel composition is no more compositional, that is, we cannot infer for the partial correctness of a formula containing a parallel composition by the partial correctness of its components. A simple example is again the case of Example 108. The statements

\[
\{\text{true}\} S_i \{X = 0 \lor X = 2\}
\]

are true for every \( S_i \ (i \in \{0, 1, 2\}) \). We cannot assert, however,

\[
\{\text{true}\} [S_0 \parallel S_1] \{X = 0 \lor X = 2\}.
\]

Thus the disjoint parallel rule \((dp)\) is not appropriate in this case. As we have pointed out earlier, the problem is caused by interfering components. Thus, the solution proposed by Owicki and Gries ([9]) was to eliminate this entanglement of proofs due to interference. This leads us to the notion of interference freedom. From now on we deal with proof outlines instead of deduction trees. In the case of shared variable parallelism, a proof outline \( \{p\} C \{q\} \) is standard if, except for the main program \( C \), every normal subprogram \( T \) of \( C \) is preceded by exactly one annotation \( pre(T) \). Moreover, for any two the consecutive annotations \( \{p_1\} \) and \( \{p_2\} \), either \( p_1 = p \) or \( q = p_2 \). As before, if the standard proof outline is preceded with two assertions \( p \) and \( pre(T) \), for some \( T \), such that \( p = pre(T) \) holds, then we omit \( pre(T) \).

**Definition 116.**

1. Let \( \{p\} C \{q\} \) be a standard proof outline for a component \( C \) of a parallel program \( S \). Assume \( R \) is a subprogram of \( S \) with precondition \( pre(R) \). We say that \( R \) does not interfere with \( \{p\} C \{q\} \) if, for all assertions \( t \) in \( \{p\} C \{q\} \),

\[
\{t \land pre(R)\} R \{t\}
\]

is valid in the sense of partial correctness.

2. Let \( S = [S_1 \parallel \cdots \parallel S_n] \) be a parallel program. Then a standard proof outline \( \{p\} S^* \{q\} \) for the partial correctness assertion \( \{p\} S \{q\} \) is inference free if no normal assignment or atomic region of the proof outline belonging to \( S_j \) interferes with \( \{p_i\} S^*_i \{q_i\} \) provided \( i \neq j \), and \( \{p_i\} S^*_i \{q_i\} \) are the corresponding standard proof outlines for \( \{p_i\} S^*_i \{q_i\} \ (1 \leq i \leq n) \).

**Example 116.** ([1]) Consider the program \( [X := 0 \parallel X := 2] \) and the standard proof outlines

\[
\{\text{true}\}
\]

\[
X := 0;
\]

\[
\{X = 0 \lor X = 2\}
\]

and
\{true\}

\( X := 2; \)

\( \{ X = 0 \lor X = 2 \} \)

Then, for example, \( X := 0 \) does not interfere with the second proof outline, for \( \{true\} X := 0 \{true\} \) and \( \{(X = 0 \lor X = 2) \land true\} X := \{X = 0 \lor X = 2\} \). The case for \( X := 2 \) is similar. Thus, the above two proof outlines are interference free.

We formulate below the rule for parallelism with shared variable \((psv)\):

\[
\frac{\bigwedge_{i=1}^n p_i \mid (p_1) S_1 \{q_1\} \mid (p_2) S_2 \{q_2\} \mid \cdots \mid (p_n) S_n \{q_n\} \bigwedge_{i=1}^n q_i}{\bigwedge_{i=1}^n p_i \mid (p_1) S_1 \{q_1\} \mid (p_2) S_2 \{q_2\} \mid \cdots \mid (p_n) S_n \{q_n\} \bigwedge_{i=1}^n q_i},
\]

where \( \{p_i\} S_i \{q_i\} \) are interference free standard proof outlines corresponding to \( \{p_i\} S_i \{q_i\} \).

Observe that the rule is the same as that of for disjoint parallelism apart from the fact that interference free proof outlines are taken into consideration. In some occasions, we may abbreviate the proof outline obtained by the rule by exchanging the composition of proof outlines \( \{p_i\} S_i \{q_i\} \) in the conclusion for the composition of the components \( S_i \).

**Example 118.** Consider the program of Example 117. Since the proof outlines \( \{true\} X := 0\{X = 0 \lor X = 2\} \) and \( \{true\} X := 2\{X = 0 \lor X = 2\} \) presented there are interference free, we can conclude, by rule \((psv)\), that \( \{true\} \{true\} X := 0\{X = 0 \lor X = 2\} | | \{true\} X := 2\{X = 0 \lor X = 2\} | | X = 0 \lor X = 2 \) is a valid proof outline.

**Example 119.** ([1]) Consider the proof outline

\{true\}

\( X := 0; \)

\( \{ X = 0 \lor X = 1 \lor X = 2 \} \)

for the partial correctness assertion \( \{true\} X := 0\{X = 0 \lor X = 1 \lor X = 2\} \). Let us also consider the proof outline

\( X := 0 \)

\( X := X + 1; \)

\( \{ X = 0 \lor X = 1 \} \)

\( X := X + 1 \)

\{true\}

It can be checked that the two proof outlines are interference free. For example, \( \{X = 0 \land (X = 0 \lor X = 1 \lor X = 2)\} X := X + 1\{X = 0 \lor X = 1 \lor X = 2\} \) is true. Then \( \{X = 0\} \{X := 0; X := X + 1; X := X + 1\} \{X = 0 \lor X = 1 \lor X = 2\} \) is a valid partial correctness assertion. Observe that taking the proof outline of Example 117 for the partial correctness assertion \( \{true\} X := 0\{X = 0 \lor X = 2\} \) would not give
interference free proof outlines. This is in accordance with our knowledge that 
\( \{ X = 0 \} [ X := 0 \mid X := X + 1; X := X + 1 ] \{ X = 0 \lor X = 2 \} \) is not a valid
partial correctness formula. Finally, let us exchange in the previous formula 
\( X := X + 1; X := X + 1 \) for the atomic region \( \langle X := X + 1; X := X + 1 \rangle \).
Let us formulate the proof outlines

\[
\begin{align*}
\{ X = 0 \} \\
\langle X := X + 1; \rangle \\
\{ X = 0 \lor X = 1 \} \\
X := X + 1 \\
\{ \text{true} \}
\end{align*}
\]

and

\[
\begin{align*}
\{ \text{true} \} \\
X := 0; \\
\{ X = 0 \lor X = 2 \}
\end{align*}
\]

Then the two proof outlines are interference free. For example,
\[
\{ X = 0 \land (X = 0 \lor X = 2) \} \langle X := X + 1; X := X + 1 \rangle \{ X = 0 \lor X = 2 \}
\]

We can infer, that in this case 
\[
\{ X = 0 \} \langle X := 0 \mid \langle X := X + 1; X := X + 1 \rangle \rangle \{ X = 0 \lor X = 2 \}
\]
is valid.

In the examples above, we alluded to statements which can be considered as consequences of the soundness of
the above proof system. After a short interlude, we will take up this question again.

The two main criteria in finding the rules of a deduction system is the soundness and completeness of the
system. That is, the system be conceived in a way to not allow the deduction of false statements, and, on the
other hand, it would be advisable if it was capable to deduce as much of the true formulas as it is theoretically
possible. A not too difficult argument shows (cf. [1]) that, if we augment the rules for partial correctness of
while programs with the parallelism with shared variable rule, the proof system is not strong enough to deduce
the formula

\[
\{ \text{true} \} [ X := 0 \mid X := X + 2 ] \{ X = 0 \lor X = 2 \}
\]

On the other hand, if we introduce a new variable indicating whether the incrementation by two has taken part,
the auxiliary variable rule will help us out in this case, too. Let us consider the proof outline

\[
\{ \text{true} \}
\]

done := false

\[
\langle X := X + 2; done := \text{true} \rangle \mid X := 0
\]

\[
\{ X = 0 \lor X = 2 \}
\]
If we add the auxiliary variable rule to our deduction system, then Assertion (4.1) can also be proved, since
is a set of auxiliary variables. To prove the proof outline above, let us take the following standard proof outlines:
\[ \{ \neg \text{done} \} \]
\[ \{ X := X + 2; \text{done} := \text{true} \} \]
\[ \{ \text{true} \} \]
and
\[ \{ \text{true} \} \]
\[ X := 0 \]
\[ \{(X = 0 \lor X = 2) \land (\neg \text{done} \supset X = 0)\} \]
We have to check now that the above two proof outlines are interference free. For example,
\[ \{(X = 0 \lor X = 2) \land (\neg \text{done} \supset X = 0) \land \neg \text{done}\} \]
\[ \{ X = 0 \} \]
\[ \{ X := X + 2; \text{done} := \text{true} \} \]
\[ X = 2 \land \text{done} = \text{true} \]
\[ \{(X = 0 \lor X = 2) \land (\neg \text{done} \supset X = 0)\} \]
Now rule \((\text{psv})\) applies and we obtain
\[ \neg \text{done} \]
\[ [(X := X + 2; \text{done} := \text{true}) || X := 0] \]
\[ \{ X = 0 \lor X = 2 \} \]
Moreover, \( \{ \text{true} \} \text{done} := \text{false} \{ \neg \text{done} \} \)
trivially holds, thus, the composition rule gives the result.
Note that the atomicity of \( \{ X := X + 2; \text{done} := \text{true} \} \) was crucial in the proof, without it the partial correctness assertion would not have been valid.

A natural question arises whether it can be planned ahead how to add auxiliary variables to achieve the proof of a correctness formula. In ([7]), Lamport has shown that there is a systematic way of adding auxiliary variables to components of a parallel program, so that the execution of each component can be monitored separately leading to proofs unavailable with the rule for parallelism with shared variable alone. With this in hand, we can now turn to the soundness and completeness issues of our deduction system.

4. Soundness and completeness

The proof system for the deduction of partial correctness formulas with shared variables was formulated in terms of proof outlines in the previous section. The change in comparison with the partial correctness calculus of while programs is the addition of three more rules: the atomic region rule, the auxiliary variable rule and the rule of parallelism with shared variables. Let us denote this proof system by \( \text{PSV} \). Since in the case of shared variable parallel programs compositionality is no more valid, that is, we are not able to compute the result of a program execution from the results obtained for its parallel components, it seems more reasonable to talk about the result of a program execution in the frames of the operational semantics. Thus
\[ M_{op}(S)(s) = \{ r \mid \langle S, s \rangle \rightarrow^* \langle E, r \rangle \} \]

and

\[ M_{op}^{tot}(S)(s) = \{ r \mid \langle S, s \rangle \rightarrow^* \langle E, r \rangle \} \cup \{ \perp \mid \text{if } \langle S, s \rangle \text{ diverges} \} \].

Here the symbol \( \perp \) for divergence is not a member of \( S \). The sets of states forming a subset of \( S \) are called proper. We know from Theorem 20, that the operational semantics and the denotational semantics convey the same meaning for while programs, so, in the case of while programs, this approach does not necessitate any changes in the statements of Chapter 1 made about soundness and completeness. With this new notation a partial correctness assertion \( \{ p \} S \{ q \} \) is true if, for every \( s \in p \),

\[ M_{op}(S)(s) \subseteq q. \]

An analogue definition holds for total correctness: let \( p \), and \( q \) be proper sets of states, that is \( p, q \subseteq S \), then \( [p]S[q] \) is true if, for every \( s \in p \),

\[ M_{op}^{tot}(S)(s) \subseteq q. \]

Observe that the latter definition involves the non-divergence of \( \langle S, s \rangle \), since \( q \) is proper.

As before, let us fix an interpretation \( \mathcal{I} \). Then the meaning of a partial correctness formula is understood as

\[ \{ P \} S \{ Q \}^{\mathcal{I}} = \mathcal{I}(\{ P \} S \{ Q \}) = \{ P^{\mathcal{I}} \} S \{ Q^{\mathcal{I}} \}, \]

which is true iff, for every \( s \) such that \( P^{\mathcal{I}}(s), M_{op}(S)(s) \subseteq Q^{\mathcal{I}} \) is valid. If \( \mathcal{I} \) is known, we may simply write \( \{ P \} S \{ Q \}^{\mathcal{I}} \) for \( \{ P \} S \{ Q \}^{\mathcal{I}} \). An analogous definition can be formulated to total correctness, too.

Without proof we state the soundness of the deduction system.

**Theorem 120.** (Soundness of PSV) Let \( \{ P \} S \{ Q \} \) be a partial correctness formula for a parallel program with shared variables. Assume \( \{ P \} S \{ Q \} \) is a standard proof outline obtained from \( \{ P \} S \{ Q \}^{\mathcal{I}} \). Then \( \{ P \} S \{ Q \}^{\mathcal{I}} \) holds true for the fixed interpretation \( \mathcal{I} \).

As to the completeness, a relative completeness statement can be formulated again in this case. We state the claim for relative completeness, the proof of which can be found in ([12]).

**Theorem 121.** (Relative completeness of PSV) Let \( \mathbb{N} \) be the standard model of arithmetic. Let \( T_{\text{arith}}(\mathbb{N}) \) be the set of all true arithmetical statements. Assume \( S \) is a parallel program with shared variables. Let \( P, S \{ Q \} \) be the proof system defined as above. Then

\[ \{ P \} S \{ Q \}^{\mathbb{N}} \Rightarrow \vdash_{T_{\text{arith}}(\mathbb{N})} \{ P \} S \{ Q \}. \]

**Remark 122.** (Total correctness) If we would like to prove total correctness assertions for while programs, the loop rule for the partial correctness proof system must be modified. However, it turns out that this kind of modification of our proof system \( PSV \) does not yield a deduction system strong enough to prove the total correctness of parallel programs with shared variables. Instead, a stronger concept of termination is needed, since the various subprograms can interact with each other, hence, a loop in a computation sequence can emerge with the control switching to and fro between the otherwise terminating program components. We illustrate this phenomenon by an example taken from ([11]). Let \( S \) be

while \( X > 0 \) do;

\( Y := 0 \);

if \( Y = 0 \) then \( X := 0 \)
else \( Y := 0 \) fi

od

and let \( S_2 \) be

while \( X > 0 \) do;
  \( Y := 1 \);
  if \( Y = 1 \) then \( X := 0 \)
    else \( Y := 1 \) fi

od

Both programs possess a correct proof outline for total correctness with specifications \((true, true)\), loop invariant \(true\) and bound function \(\max(X, 0)\). In addition these proof outlines are interference free, hence, by rule \((psv)\) we would obtain

\[[true][S_1 \parallel S_2][true].\]

But the above sequence of configurations show that \( S \) can diverge. Let \( s \in S \) be such that \( s(X) = 1 \). In what follows, we distinguish reduction steps coming from \( S_1 \) and \( S_2 \) by writing upper-indexes \( \rightarrow^1 \) and \( \rightarrow^2 \), respectively. Let \( C_i \) be the conditional statement in \( S_i \) \((i \in \{1, 2\})\).

\[
\langle [S_1 \parallel S_2], s \rangle \rightarrow^1 \langle [Y := 0; C_1; S_1 \parallel S_2], s \rangle \\
\rightarrow^2 \langle [Y := 0; C_1; S_1 \parallel Y := 1; C_2; S_2], s \rangle \\
\rightarrow^1 \langle [C_1; S_1 \parallel Y := 1; C_2; S_2], s[Y/0] \rangle \\
\rightarrow^2 \langle [C_1; S_1 \parallel C_2; S_2], s[Y/1] \rangle \\
\rightarrow^1 \langle [Y := 0; S_1 \parallel C_2; S_2], s[Y/1] \rangle \\
\rightarrow^1 \langle [S_1 \parallel Y := 1; S_2], s[Y/0] \rangle \\
\rightarrow^2 \langle [S_1 \parallel S_2], s[Y/1] \rangle,
\]

and the whole process starts all over again. This example shows that total correctness needs more precaution: the loop rule for the total correctness of while programs is not enough in this case. The new loop rule must ensure that every program execution reduces the value of the bound function. We do not elaborate the question any further here, the interested reader can find a detailed exposition in ([1]).

5. Case study

We illustrate the proof method described in this chapter by an example presented in ([1]). Before discussing the example, we deal with some additional results simplifying the proof to follow.

**Theorem 123.** (Atomicity) Let \( S = S_0; [S_1 \parallel S_2 \parallel \cdots \parallel S_n] \) be a parallel program, where \( S_0 \) is a sequential one. Let \( 0 \leq i \leq n \) be fixed. Let \( T \) be the program obtained from \( S \) either

1. \( \ldots \)
by replacing in $S_i$ an atomic region of the form $\langle R_1; R_2 \rangle$ by $\langle R_1 \rangle; \langle R_2 \rangle$, where one of
and is disjoint from every
in the sense of Definition 92, or

2.

by replacing in $S_i$ an atomic region of the form $\langle \text{if } B \text{ then } R_1 \text{ else } R_2 \text{ fi} \rangle$ by
$\text{if } B \text{ then } \langle R_1 \rangle \text{ else } \langle R_2 \rangle \text{ fi}$, where $B$ is disjoint from every $S_j (j \neq i)$ in the
sense of Definition 92.

Then

$\mathcal{M}_{op}(S) = \mathcal{M}_{op}(T)$

and

$\mathcal{M}_{tot}^{op}(S) = \mathcal{M}_{tot}^{op}(T)$.

Corollary 124. Let $\mathcal{P}, \mathcal{Q}$ be assertions, assume $S$ and $T$ are as in Theorem 123. Then
$\{ \mathcal{P} \} S \{ \mathcal{Q} \}$ iff $\{ \mathcal{P} \} T \{ \mathcal{Q} \}$.

A similar result holds for total correctness, too.

Theorem 125. (Initialization) Let

$S = S_0; R_0; [S_1 || S_2 || \ldots || S_n]$

be a parallel program, where $S_0, R_0$ are sequential ones. Suppose there exists $1 \leq i \leq n$
such that $R_0$ is disjoint from all $S_j (j \neq i)$. Let

$T = S_0; [S_1 || \ldots || S_i || \ldots || S_n]$.\n
Then

$\mathcal{M}_{op}(S) = \mathcal{M}_{op}(T)$

and

$\mathcal{M}_{tot}^{op}(S) = \mathcal{M}_{tot}^{op}(T)$.

Corollary 126. Let $\mathcal{P}, \mathcal{Q}$ be assertions, assume $S$ and $T$ are as in Theorem 125. Then
$\{ \mathcal{P} \} S \{ \mathcal{Q} \}$ iff $\{ \mathcal{P} \} T \{ \mathcal{Q} \}$.

A similar result holds for total correctness, too.

Now we can set to and demonstrate the applicability of our deduction system in a partial correctness proof.

Example 127. (1) Let us find the zeros of a number theoretic function $f$. The task is
distributed between two components $S_1$ and $S_2$: $S_1$ searches among the positive and $S_2$
among the negative arguments. Thus:

$Search = found := false; [S_1 || S_2]$

where

$S_1 = x:=0;$

$S_2 = x:=0;$
while \( \neg found \) do
  \( X := X + 1; \)
  if \( f(X) = 0 \) then
    \( found := true \)
  fi
od

and

\( S_2 := Y := 1; \)
while \( \neg found \) do
  \( Y := Y - 1; \)
  if \( f(Y) = 0 \) then
    \( found := true \)
  fi
od

We wish to verify the partial correctness formula

\[ \{true\} \text{Search}\{f(X) = 0 \lor f(Y) = 0\}. \]

First we simplify the components by virtue of Theorems 125 and 123 making use of the observations that \( X \) does not occur in \( S_2 \) and \( X_1 \) does not occur in \( S_1 \). Thus the new statement is

\[ \{true\} found := false; X := 0; Y := 1; [T_1 || T_2]\{f(X) = 0 \lor f(Y) = 0\}, \]

where \( T_1 = \text{while } \neg found \text{ do } \)

\( \langle X := X + 1; \)
  if \( f(X) = 0 \) then
    \( found := true \)
  fi \rangle \)

od

and

\( T_2 = \text{while } \neg found \text{ do } \)

\( \langle Y := Y - 1; \)
  if \( f(Y) = 0 \) then
Shared variable parallelism

\[
\text{\texttt{found := true}}
\]
\[
\text{fi}
\]
\[
\text{od}
\]

Thus we need fewer statements to consider when verifying interference freedom. We choose the loop invariant \( P_1 \) for \( T_1 \) and \( P_2 \) for \( T_2 \), respectively, where \( P_1 \) and \( P_2 \) are defined as follows.

\[
P_1 = x \geq 0
\]
\[
(\text{found} \supset (X > 0 \land f(X) = 0) \lor (Y \leq 0 \land f(Y) = 0))
\]
\[
(\neg \text{found} \land X > 0 \supset f(X) \neq 0)
\]

\[
P_2 = y \leq 1
\]
\[
(\text{found} \supset (X > 0 \land f(X) = 0) \lor (Y \leq 0 \land f(Y) = 0))
\]
\[
(\neg \text{found} \land Y \leq 0 \supset f(Y) \neq 0)
\]

We try to keep track of some of the steps of the proof. For the assertion \( \{P_1\} T_1 \{P_1 \land \text{found}\} \) we have to verify

\[
\{P_1 \land \neg \text{found}\}
\]
\[
\langle X := X + 1;
\]
\[
\text{if } f(X) = 0 \text{ then}
\]
\[
\text{found := true}
\]
\[
\text{fi}
\]
\[
\{P_1\}
\]

By the assignment and the conditional rules for proof outlines we obtain

\[
\{(f(X) = 0 \land P_1[\text{found/true}]) \lor (f(X) \neq 0 \land P_1)\}
\]
\[
\text{if } f(X) = 0 \text{ then } \text{found := true } \text{ fi}
\]
\[
\{P_1\}.
\]

Moreover, the assignment and composition rules together give

\[
\{(f(X + 1) = 0 \land P_1[\text{found/true}][X/X + 1]) \lor (f(X + 1) \neq 0 \land P_1[X/X + 1])\}
\]
\[
X := X + 1; \text{ if } f(X) = 0 \text{ then } \text{found := true } \text{ fi}
\]
\[
\{P_1\}.
\]

By the rule for atomic regions we can assert

\[
\{(f(X + 1) = 0 \land P_1[\text{found/true}][X/X + 1]) \lor (f(X + 1) \neq 0 \land P_1[X/X + 1])\}
\]
\[
\langle X := X + 1; \text{ if } f(X) = 0 \text{ then } \text{found := true } \text{ fi } \rangle
\]
\{P_1\}.

What remains is to prove the validity of

\[ P_1 \land \neg\text{found} \Rightarrow (f(X + 1) = 0 \land P_1[\text{found}/\text{true}][X/X + 1]) \lor (f(X + 1) \neq 0 \land P_1[X/X + 1]). \]

Instead, we prove that the conclusion is a consequence of the premiss

\[ P_1 \land \neg\text{found} \land (f(X + 1) = 0 \lor f(X + 1) \neq 0), \]

which can be checked by using laws of first order logic. From this, by the consequence rule, the intended partial correctness assertion follows. In an analogous manner, we can demonstrate \{P_2\}T_2\{P_2 \land \text{found}\}.

To combine the two proof outlines, we have to check interference freedom. We may assume that the proof outlines in question are standard ones, that is, we may omit the subsequent assertions in the proof outlines leaving just one assertion in front of every normal subprogram of \(T_1\) and \(T_2\). Thus, it is enough to consider the proof outlines

\{P_1\}

while \(\neg\text{found}\) do

\{P_1 \land \neg\text{found}\}

\(X:=X+1;\)

if \(f(X) = 0\) then

\text{found} := \text{true}

fi

\{P_1\}

od

\{P_1 \land \text{found}\}

and

\{P_2\}

while \(\neg\text{found}\) do

\{P_2 \land \neg\text{found}\}

\(Y:=Y-1;\)

if \(f(Y) = 0\) then

\text{found} := \text{true}

fi

\{P_2\}

od

\{P_2 \land \text{found}\}
So, for example, we have to check for the independence of \( X := X + 1; \ if \ f(X) = 0 \ then \ fi \) of the proof outline for \( P_2 \). Hence, we have to check statements like

\[
\{(P_1 \land \neg \text{found}) \land P_2 \} \land (X := X + 1; \ if \ f(X) = 0 \ then \ fi).
\]

They can be verified in a manner similar to the proof above. Having done all these steps, we can deduce, by the parallelism with shared variable rule,

\[
\{P_1 \land P_2\}[T_1 \parallel T_2]\{P_1 \land P_2 \land \text{found}\}.
\]

But

\[
\{true\} \text{found} := false; X := 0; Y := 1\{P_1 \land P_2\},
\]

hence the consequence rule yields the result. We remark that proving termination is impossible under these conditions, since there are scenarios when the program \( \text{Search} \) does not terminate. Assume, for example, \( f(n) > 0 \) if \( n > 0 \), and the execution of \( \text{Search} \) consists in a sequence of executions of the component \( S_1 \). Then \( \text{Search} \) will never find a solution even if there is one, and the execution of the program will never terminate.
Chapter 5. Synchronization

1. Syntax

There are parallel programs which cannot be described only with communication by shared variables. We also need a construct which enables us to formulate programs the parts of which being able to synchronize with each other. In other words, the components are able to suspend their execution, and wait for a certain condition to be fulfilled. This is achieved by the await construct of Owicki and Gries. Parallel programs with synchronization are formed exactly like parallel programs of the previous two chapters with the additional await construct

\[ \text{await } B \text{ then } S \text{ end}. \]

Here \( S \) is a loop free sequential program not containing any await constructs. A parallel component consists of the constituents of a while program of Chapter 1 and the await construct. Again, nested parallelism is not allowed, but parallel composition can occur inside sequential composition, conditional statements and while loops. The informal explanation for the execution of the await construct is as follows: should the control reach an \( \text{await } B \text{ then } S \text{ end} \) subprogram in a command the following two cases can happen. If \( B \) evaluates to true, then \( S \) is executed uninterruptedly as an atomic region. Otherwise, the whole component stalls, and it is allowed to resume execution only if \( B \) evaluates to true due to the change of the values of the variables in \( B \) by some other components. Of this never happens, then the component remains blocked forever. We introduce two abbreviations:

\[ \langle S \rangle = \text{await } \text{true} \text{ then } S \text{ end} \]

and

\[ \text{wait } B = \text{await } B \text{ then } \text{skip} \text{ end}. \]

Thus, atomic regions are simply specially formed await statements form now on. In accordance with this, we call a subprogram normal, if it does not lie within an await statement.

2. Semantics

The transition rule for the await construct is quite straightforward. If, in a fixed interpretation, \( B \) evaluates to true, then we can infer the result of the execution of the await construct thus:

\[ \langle S, s \rangle \rightarrow^* \langle E, r \rangle \]

\[ \langle \text{await } B \text{ then } S \text{ end}, s \rangle \rightarrow \langle E, r \rangle. \]

This leads us to the following definition:

**Definition 128.** A configuration \( \langle S, s \rangle \) is a deadlock, if \( S \not\rightarrow E \) and there are no transitions starting form that configuration. a program \( S \) can deadlock from \( s \) if there is a computation sequence starting from \( \langle S, s \rangle \) and ending in a configuration which is a deadlock. \( S \) is deadlock free, if there are no states \( s \) such that \( S \) can deadlock from \( s \).

The operational semantics for total correctness is modified by reason of deadlock. Hence,

\[ M_{op}(S)(s) = \{ r \mid \langle S, s \rangle \rightarrow^* \langle E, r \rangle \}, \]

but

\[ M^{df}_{op}(S)(s) = M_{op}(S)(s) \cup \{ \bot \mid S \text{ can diverge from } s \} \cup \{ \Delta \mid S \text{ can deadlock from } s \}. \]

3. Proof rules for partial correctness
To prove partial correctness for programs with synchronization, it is enough to deal with the additional `await` construct.

\[ \{ p \land B \} S \{ q \} \quad (\text{synchronization}) \]

If \( B \) is true, then the (synchronization) rule simply transforms into the (atomic) rule for atomic regions. Proof outlines are generated with the rules applied for parallel programs with shared variables together with the rule for synchronization:

\[ \{ p \land B \} S^* \{ q \} \quad (\text{synchronization}) \]

where, as usual, \( S^* \) is some proof outline obtained from \( S \). A proof outline is standard as usual, in this case a subprogram is normal if it is not within an `await` statement. There are no assertions within `await` statements. Interference freedom \( PSY \) (partial correctness with synchronization) is understood as no normal assignment or `await` statement interferes with the proof outline of another parallel component. The proof system is formulated now, in the form of a proof outline, by the usual rules for proof outlines for while programs together with (atomicity) and (psu) is added for the parallel components. We state a soundness theorem.

**Theorem 129.** The proof system \( PSY \) is sound, that is, for a fixed interpretation \( T \), if the standard proof outline \( \{ p \} S^* \{ q \} \) is inferred, then \( \{ p \} S \{ q \}^T \) holds, as well.

As an example we prove the following partial correctness assertion.

**Example 130.** Let \( n \in \mathbb{N} \). Let us verify

\[ \{ X = n \} \]

\[ \neg \text{done}; \]

while \( \neg \text{done} \) do [await X=0 then done:=true end \( X:=X-1 \)] od

\[ \{ X = 0 \} . \]

To this end, we provide the proofs of the parallel components first, in the forms of proof outlines. As an invariant, we choose \( P = \text{done} \supset X = 0 \). First of all, we prove

\[ \{ p \land \neg \text{done} \}[\text{await } X = 0 \text{ then skip end } \| X := X - 1] \{ P \} \]

\[ \{ \neg \text{done} \} \]

wait X=0 do

\[ \{ \neg \text{done} \land X = 0 \} \]

\[ \{ P[\text{done/true}] \} \]

done:=true

\[ \{ P \} \]

end

\[ \{ P \} \]

and
We have to check interference freedom. We confine ourselves to standard proof outlines in order to have fewer assertions to take into account. Thus, there are no assertions within atomic regions. Hence, for example,

\[
\{ \neg \text{done} \} \\
\{ \delta \}\{ X := X - 1 \} \\
\{ \delta \}
\]

or

\[
\{ \delta \}\{ \text{P} \land \neg \text{done} \}\{ X := X - 1 \} \\
\{ \delta \}
\]

await \(X=0\) do

\[\text{done}:=\text{true}\]

end

\[
\{ \delta \}
\]

Combining all these together, rule \((\text{psv})\) yields

\[
\{ \neg \text{done} \} \{ \text{await } X = 0 \text{ then } \text{done} := \text{true} \text{ end } \| X := X - 1 \} \{ \text{P} \},
\]

thus

\[
\{ \text{P} \} \text{while } \neg \text{done } \{ \text{await } X = 0 \text{ then } \text{done} := \text{true} \text{ end } \| X := X - 1 \} \text{ od}\{ \text{P} \land \text{done} \}.
\]

From this the partial correctness assertion follows.

4. Mutual exclusion

As the last section of this chapter, we present one of the most widely-known problems of parallel processes with synchronization: the mutual exclusion problem. The question consists of several subquestions, the task is to find a way for synchronizing some number of processes such that

- (mutual exclusion): the operation of the processes is mutually exclusive in their critical section,

- (absence of blocking): the synchronization discipline does not prevent the processes from running indefinitely,

- (lack of starvation): if a process is trying to acquire the resource eventually it will succeed.

Following the monographs of [1] and [6] and [8] we give a short account of the main algorithms solving the mutual exclusion problem. We assume that a program \(S\) is given consisting of parallel components \(S_i\) of the form
The various parts of the $i$-th component will be denoted by $ES_i$, $CS_i$, $RS_i$, $NC\ S_i$, respectively. In fact, we assume that $S$ takes up the following form:

$$S = Init; [S_1 \ || \ldots \ || S_n],$$

where $Init$ is a loop free while program serving for the initialization of the variables of the entry and release sections. Firstly, we expound the possibly simplest solution of mutual exclusion between two processes: Peterson’s mutual exclusion algorithm.

$Init$: flag[0]:=false; flag[1]:=false; turn:=0;

and $S_1$ is

while true do
1: flag[0]:=true;
2: turn:=0;
3: while (flag[1] $\&$ turn=0) do skip od;
4: critical section
5: flag[0]:=false;
6: non-critical section
od

while $S_2$ is

while true do
7: flag[1]:=true;
8: turn:=1;
9: while (flag[0] $\&$ turn=1) do skip od;
10: critical section
11: flag[1]:=false;
12: non-critical section
od

First of all, we demonstrate informally that $S = Init; [S_1 \ || S_2]$ cannot deadlock. The only program point where $S_1$ can wait is the part with label 3. Similarly, $S_2$ can wait only at step with label 9. This can only happen if (flag[1] $\&$ turn=0) and (flag[0] $\&$ turn=1) are both true, but this implies (turn=0 $\&$ turn=1), which is impossible.
Next, we show that it is impossible that control resides in step 4 and step 10 at the same time. Assume without loss of generality that the execution of is at step 4. Then only happen if previously turn=1 was true. To enter in , must encounter or . Since is in its critical section, is true, hence turn=0 must hold. But this is impossible, as the following argument shows.

- $S_1$ enters its critical section since $flag[1]$ is false. This can only happen, when $S_2$ has not executed step 7 until then. Next, $S_2$ executes step 7 and 8, setting turn:=1. This means, $S_2$ cannot enter its critical section in this way.

- $S_1$ enters its critical section by reason of turn=1. This means, step 2 of $S_1$ must precede step 8 of $S_2$. Then $S_2$ reads flag[0] ∧ turn=1, which means again that $S_2$ cannot enter its critical section.

Finally, we prove that the lack of starvation property also holds. That is, if a process sets its flag to true, then it eventually enters its critical section. Assume $flag[0]$ is true, but $S_1$ is waiting at step 3. If $S_2$ is in $CS_2$, then eventually it sets $flag[1]$ to false, which enables $S_1$ to enter in $CS_1$. If $S_1$ did not make use of the occasion, in the subsequent steps $S_2$ sets $flag[1]$ to true and turn to 1, which results that $S_2$ must come to a halt and then $S_1$ can enter its critical section.

Peterson has also generalised his algorithm for $n$ processes, which we describe below.

Init: flag[k]=0 and turn=0 for all $1 \leq k \leq n$

Moreover, for all $S_i$ $(1 \leq i \leq n)$

while true do
  j:=1;
  while j < N-1 do
    flag[i]:=j;
    turn[j]:=i;
    while $(\exists k \neq i)(flag[k] = j$ ∧ turn[j] $= i))$ then do skip od
    j:=j+1
  od
  critical section
  flag[i]:=0
  noncritical section
  od

It can be shown that the generalized Peterson-algorithm also possesses the properties of deadlock freedom, mutual exclusion and absence of starvation. However, for $n$ processes, the above algorithm can become very difficult. In what follows, we depict an algorithm handling mutual exclusion for more than two processes in an elegant way, this is Lamport’s Bakery algorithm.

The algorithm dislocates control between $n$ processes. The actual process checks all other processes sequentially and waits for the other processes with lower index. Ties are resolved by assigning process identifiers to processes besides indices.

Init: number[1]:=0;…; number[n]:=0;
choosing[1]:=false;…; choosing[n]:=false;

and the body of the perpetual loop of component $S_i$ is

1: choosing[i]:=true;

2: number[i]:=1+max{number[1],…,number[n]};

3: choosing[i]:=false;

4: for all $j \neq i$ do

5: wait choosing[j]=false;

6: wait number[j]=0 or (number[i],i)<(number[j],j)

7: od

8: critical section

9: number[i]:=0;

10: non-critical section,

where processes are order lexicographically: $(number[i],i) < (number[j],j)$ iff $number[i] < number[j]$ or $number[i] = number[j]$ and $i < j$. Leaning on (8), we give a sketch of the proof of the fact that the Bakery algorithm fulfills the properties expected of mutual exclusion algorithms.

Firstly, a lemma ensures mutual exclusion. In what follows, the $i$-th process is simply denoted by $i$.

**Lemma 131.** Let us call steps 1-3 of $S_i$ the doorway part, steps 4-7 the bakery. Let $D$ be the set of indices of components residing in the doorway, and $B$ the set of indices of processes standing at the bakery stage. Let $C = \{j \mid j \in C \cup S_j\}$. Assume $i \in C$ and $k \in B \cup C$ for some $k \neq i$. Then

$$(number[i],i) < number([k],k).$$

**Proof.** Let $i \in C$ and $j \in B \cup C$ and $k \neq i$. Assume $i$ enters $C$ before $k$. Then, before $i$ enters $C$, it must have read $choosing[k]=false$ at step 5. Let this be event $\pi_i$. We have two possible cases:

- $k$ chooses $number[k]$ after event $\pi_i$. Then, since $number[i]$ is already settled, the only possibility is $number[i] < number[k]$.

- $k$ chooses $number[k]$ before $\pi_i$. This means that $i$, when reading $number[k]$ at step 6, reads the "correct" value of $number[k]$, and then $i$ enters $C$. But this can happen only if $(number[i],i) < number([k],k)$.

**Corollary 132.** No two processes can be in $C$ at the same time.

**Lemma 133.** (deadlock freedom) The bakery algorithm never comes to a deadlock.

**Proof.** If we assume that the non-critical sections are deadlock free, a process can wait only at step 5 and 6. Since the condition of the await statement in step 5 fulfills after finitely many steps for all of the processes, deadlock can occur only at step 6. But there are only finitely many processes in $B$, this means one of them inevitably enters in $C$, and the algorithm continues.
Lemma 134. (lack of starvation) Assume that, for every process $i$, $NCS_i$ terminates. Then any process entering $D$ reaches eventually its critical section.

Proof. Let $i$ enter $D$. Then $i$ chooses $number[i]$ and enters $B$. Every process entering $D$ after $i$ must have a greater number than $i$. Since no noncritical section can lead to an infinite computation, there must come a state when $i$ enters $C$ in order to ensure the continuity of the algorithm.

Finally, following [1], another well-known solution for the mutual exclusion problem is described, which is the algorithm of Dijkstra using semaphores. A semaphore is an integer variable, with two operations allowed on it:

$$\begin{align*}
P(sem) &= \text{await } sem > 0 \text{ then } sem := sem - 1 \text{ end} \\
V(sem) &= sem := sem + 1
\end{align*}$$

A binary semaphore is a variable taking values 0 and 1, where all operations are understood modulo 2. Let $out$ be a binary semaphore. Then the mutual exclusion algorithm of Dijkstra can be formulated as follows:

$Dijkstra = out := true; \text{ who := 0; } [S_1 \parallel \cdots \parallel S_n]$, where

$S_i = \text{while true do}$

$\quad NCS_i;$

$\quad \text{await out then out := false; who := i end;}$

$\quad CS_i;$

$\quad \{ \text{ out := true; who := 0 } \}$

$\text{ od}$

Observe that the operators $P$ and $V$ have now been extended with actions governing an auxiliary variable $who$. The semaphore $out$ indicates whether there are processes in their critical sections. Though we did not lay the theoretical foundation for a formal treatment, relying on the intuition of the reader, after ([1]), we present a proof for the mutual exclusion and absence of blocking properties. For invariant, we choose

$$I = (\bigvee_{j=0}^{n} who = j) \land (who = 0 \equiv out).$$

Then the one below is a valid proof outline for $S_i (1 \leq i \leq n)$:

$\{inv: I \land who \neq i\}$

while true do

$\{I \land who \neq i\}$

$NCS_i$

$\{I \land who \neq i\}$

$\text{await out then out := false; who := i end;}$

$\{\neg out \land who = i\}$

$CS_i$;
\{ \neg \text{out} \land \text{who} = i \}\n
\langle \text{out:=true; who:=0} \rangle \n
\text{od}
\{ \text{false} \}\n
The proof outlines in themselves are correct, we have to verify their interference freedom to be able to reason about mutual exclusion. For example, proving the independence of \( \langle \text{out:=true; who:=0} \rangle \) from the proof outline for \( S_i \) needs the verification for proof outlines like the following one

\{ \neg \text{out} \land \text{who} = i \land \neg \text{out} \land \text{who} = j \}\n
\langle \text{out:=true; who:=0} \rangle \n
\{ \neg \text{out} \land \text{who} = i \} ,

which is valid by \( i \neq j \). As to the independence of \( \text{await} \text{out} \text{then} \text{out:=false; who:=i} \text{end} \), we consider, for instance, the proof outline below

\{ I \land \text{who} \neq i \land \neg \text{out} \land \text{who} = i \}\n
\text{await} \text{out} \text{then} \text{out:=false; who:=i} \text{end};
\{ \neg \text{out} \land \text{who} = i \} ,

which can be verified by the synchronization rule. Taking all these into account, we can assert that the proof outlines for \( S_i \) are free from interference, which allows us to examine them in relation to mutual exclusion. Intuitively, two processes are mutually exclusive, if it never happens that both of them can enter their critical sections in the same time. This can be formalized as saying that the preconditions for the critical sections in the above proof outlines are mutually exclusive, that is,

\neg (\text{pre}(CS_i) \land \text{pre}(CS_j))

holds for every \( i,j (i \neq j) \). Taking into account the formulations of \( \text{pre}(CS_i) \), the validity of this property is immediate.

Leaning on the intuitive understanding of nonblocking property, we verify its validity starting from the above proof outlines. Intuitively, since \( \text{true} \) stands as the condition of the while loop for every process, blocks can take place only if all of the processes \( S_i \) are about to execute their \( \text{await} \) statements and they are blocked their. This leads us to the conclusion that in states like this the preconditions for the \( \text{await} \) statements of \( S_i \) should hold together with \( \neg \text{out} \), which prevents the processes from entering their critical sections. Thus, let \( r_i = I \land \text{who} \neq i \land \neg \text{out} \). We have to check for the absence of blocking that

\[ \bigwedge_{j=1}^{n} r_j = \bigwedge_{j=1}^{n} (I \land \text{who} \neq j \land \neg \text{out}) \]

can never be valid. This is implied by the fact \( \neg \text{out} \Rightarrow \text{who} \neq 0 \), together with the implications \( I \Rightarrow \bigvee_{j=1}^{n} \text{who} = j \) and \( \bigwedge_{j=1}^{n} r_j \Rightarrow \bigwedge_{j=1}^{n} \text{who} \neq j \).
Appendix A. Mathematical background

1. Sets and relations

In this appendix we give a brief account of the most important notions and definitions used in the course material. We take the notion of a set as granted. We define set containment with the help of ∈:

\[ A \subseteq B \iff (x \in A \Rightarrow x \in B) \]

Two sets are equal if they mutually contain each other. If \( \mathcal{U} \) is a set, a property of the elements of \( \mathcal{U} \) is a subset of \( \mathcal{U} \). If \( P: \mathcal{U} \rightarrow \{true, false\} \), then \( P \) also defines a property: this is the set of elements \( x \in \mathcal{U} \) such that \( P(x) = true \). We denote this property by \( \{x \mid P(x)\} \). We define set theoretic operations as usual. That is,

\[ X \cup Y = \{x \mid x \in X \lor x \in Y\}, \]

and

\[ X \cap Y = \{x \mid x \in X \land x \in Y\}. \]

Moreover, \( X \setminus Y = X \cap Y^c \), where \( Y^c = \{y \mid y \notin Y\} \), and

\[ X \times Y = \{(x, y) \mid x \in X \land y \in Y\}. \]

We can define intersection and union in a more general setting. Let \( \mathcal{X} \subseteq \mathcal{P}(\mathcal{U}) \), that is, assume, for every \( X \in \mathcal{X} \), \( X \subseteq \mathcal{U} \). Then

\[ \bigcup \mathcal{X} = \{a \mid (\exists X \in \mathcal{X})(a \in X)\}, \]

and

\[ \bigcap \mathcal{X} = \{a \mid (\forall X \in \mathcal{X})(a \in X)\}. \]

Let \( X, Y \) and \( Z \) be sets. Then any set \( R \subseteq X \times Y \) is called a binary relation over \( X \times Y \). If \( X = Y \), we say that \( R \) is a relation over \( X \). If \( (x, y) \in R \) and \( (x, z) \in R \) implies \( y = z \), then \( R \) is called a function from \( X \) into \( Y \), and is denoted by

\[ R: X \rightarrow Y. \]

We may write \( R(x, y) \) or \( xRy \) instead of \( (x, y) \in R \). The most widespread operations on relations are forming the inverse relation and the composition of relations. Let \( R \subseteq X \times Y \), and \( R_1 \subseteq X \times Y \) and \( R_2 \subseteq Y \times Z \). We define

\[ R^{-1} = \{(y, x) \mid (x, y) \in R\}, \]

and

\[ R_2 \circ R_1 = \{(x, z) \mid \exists y((x, y) \in R_1 \land (y, z) \in R_2)\}. \]

We understand the composition of functions as relation composition. Observe that the composition of functions is again a function. In contrast to the usual notation for relation composition, we introduce a different notation for the compound relation.
Notation 135. Let \( R_1 \subseteq X \times Y \) and \( R_2 \subseteq Y \times Z \). Then
\[
R_1 \circ R_2 \equiv R_2 \circ R_1.
\]
That is, we read the relations taking part in a composition in an order differing from the conventional one. In the rest of the section, if otherwise stated, we assume that \( R \) is a relation over a fixed set \( X \). We say that a relation
\[
\begin{array}{ll}
R \text{ is reflexive,} & \text{if } \forall x R(x, x), \\
R \text{ is transitive,} & \text{if } \forall x y z (R(x, y) \wedge R(y, z) \Rightarrow R(x, z)), \\
R \text{ is symmetric,} & \text{if } \forall x (R(x, y) \Rightarrow R(y, x)), \\
R \text{ is antisymmetric,} & \text{if } \forall x y (R(x, y) \wedge R(y, x) \Rightarrow x = y).
\end{array}
\]

A partial order is a reflexive, transitive, antisymmetric relation. An equivalence is a reflexive, symmetric, transitive relation. Let \( R \) be a relation. Let \( R^0 = id \), \( R^1 = R \) and \( R^{i+1} = R \circ R^i \) for \( i \geq 2 \). Then the reflexive, transitive closure of \( R \), which is denoted by \( R^* \), is defined as
\[
R^* = \bigcup_{i \in \mathbb{N}} R^i.
\]
It is easy to check that \( R \) is reflexive, transitive and contains \( R \). Moreover, \( R \) is the least such relation which will be demonstrated in the next section together with the relation
\[
R^* = R \circ R^* \cup \delta,
\]
where \( \delta \) is the identity function on \( X \).

Let \( R \subseteq X \times Y \), assume \( A \subseteq X \) and \( B \subseteq Y \). Then
\[
R(A) = \{ y \mid (\exists x \in A)((x, y) \in R) \},
\]
and
\[
R^{-1}(B) = \{ x \mid (\exists y \in B)((x, y) \in R) \}
\]
are the image of \( A \), and the inverse image of \( B \) with respect to \( R \), respectively. In a similar manner we can talk about images and inverse images with respect to a function \( f : X \to Y \). Moreover, a function \( f : X \to Y \) is injective, if \((x, y) \in f \) and \((z, y) \in f \) implies \( x = z \) for every \( x, y, z \in X \). \( f \) is surjective, if \( f(X) = Y \). An injective and surjective function is a bijection. Let \((x, y) \in f \). Then we apply the notation \( f(x) = y \).

2. Complete partial orders and fixpoint theorems

The pair \((D, \leq)\) is called a partial order, when \( \leq \) is a partial order on \( D \), that is, a reflexive, antisymmetric and transitive relation. Let \( X \subseteq D \). We say that \( d \) is a lower bound of \( X \) if, for all \( x \in X \), \( d \leq x \). \( d \) is the greatest lower bound (glb) of \( X \), if \( d \) is a lower bound, and, for every \( e \) such that \( e \) is a lower bound of \( X \), we have \( e \leq d \). An analogue definition holds for upper bounds and the least upper bound (lub). We say that \( \bot \) is a minimal (or bottom) element of \( D \), if \( \bot \) is a lower bound for \( D \).

Definition 136. \((D, \leq)\) is a complete partial order (cpo) if \((D, \leq)\) is a partial order, and, for every increasing, countable sequence (so-called chain) \( d_0, d_1, \cdots, d_n, \cdots \) of elements of \( D \) its least upper bound \( \bigcup_{i=0}^{\infty} d_i \) exists.

Definition 137. Let \( D, E \) be partial orders. A function \( f : D \to E \) is monotonic, if for every \( d_1, d_2 \in D \), \( d_1 \leq d_2 \) implies \( f(d_1) \leq f(d_2) \).
Definition 138. Let $D, E$ be complete partial orders. A function $f : D \to E$ is continuous, if it is monotonic, and, for every chain of elements of $D$,

$$f \left( \bigsqcup_i d_i \right) = \bigsqcup_i f(d_i).$$

The set of continuous functions from the cpo $D$ to the cpo $E$ will be denoted by $[D \to E]$.

Example 139. ([111])

1.

Every set ordered by the identity function is a discrete cpo. Any function from discrete cpo’s to discrete cpo’s are continuous.

2.

Let $B$ be the cpo consisting of elements $\bot$ and $\top$ such that $\bot \leq \top$.

3.

Let $X$ be a set. Let $\mathcal{P}(X) = \{ Y \mid Y \subseteq X \}$. Then $\mathcal{P}(X)$ is a complete lattice.

4.

Let $F = \{ f : X \to Y \mid f$ is a partial function $\}$. Then $F$ is a cpo.

5.

Let $\mathbb{N}_\infty$ be the partial order $0 \leq 1 \leq \cdots \leq n \leq \cdots \leq \infty$. Then $\mathbb{N}_\infty$ is a cpo. Let $f_n : \mathbb{N}_\infty \to B$ be such that

$$f_n(x) = \begin{cases} \top, & \text{if } n \leq x \\ \bot, & \text{otherwise} \end{cases}$$

Then $f_n$ is continuous for every $n \in \mathbb{N}$, but $f_\infty$ is not continuous.

Lemma 140. Let $f \in [D \to E]$ and $g \in [E \to H]$. Then $f \circ g \in [D \to H]$, where $(f \circ g)(x) = g(f(x))$ is the function composition of $f$ and $g$.

Definition 141. Let $(D, \preceq)$ be a partial order, let $f : D \to D$. Then

1.

$d$ is a prefixpoint of $f$ if $f(d) \preceq d$.

2.

$d$ is a postpoint of $f$ if $d \preceq f(d)$.

A fixpoint of $f$ is both a prefixpoint and a postfixpoint. $d$ is called the least fixpoint (lfp) of $f$, if $d$ is a fixpoint, and, for every $f(e) = e$, $d \preceq e$.

Without proof we state the following theorem.
Theorem 142. (Knaster and Tarski) Let \((D, \leq, \bot)\) be a cpo with bottom, assume \(\bot \leq A\). Then

\[
\text{lfp}(f) = \bigcup_{n=0}^{\infty} f^n(\bot)
\]

is the least fixpoint of \(f\), where \(f^0 = \text{id}\), \(f^1 = f\), and \(f^{i+1} = f \circ f^i\). Moreover, \(\text{lfp}(\bar{f})\) is the least prefixpoint of \(f\), that is, if \(f(d) \leq d\), then \(\text{lfp}(f) \leq d\).

Below, we consider a straightforward application of the Knaster-Tarski theorem. Let \(X\) be a set, and \(R \subseteq X \times X\) be a relation over \(X\). As before, let

\[
\begin{align*}
R^0 &= \emptyset \\
R^1 &= R \\
R^{i+1} &= R \circ R^i \\
\end{align*}
\]

and let \(R^* = \bigcup_{i=0}^{\infty} R^i\). We assert the following lemma.

Lemma 143. Let \(F(\phi) = R \circ \phi \cup \delta\), where \(\phi \subseteq S \times S\) and \(\delta\) is the identity relation over \(X\) and \(R \subseteq S \times S\). Then

\[
\text{lfp}(F) = R^*.
\]

Proof. First of all, observe that \(F : \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)\) is continuous. For, let \(\phi_0 \subseteq \phi_1 \subseteq \ldots \subseteq \phi_n \subseteq \ldots\) be a chain in \(\mathcal{P}(S \times S)\). Then

\[
\begin{align*}
F\left(\bigcup_{n=0}^{\infty} \phi_n \right) &= R \circ \left(\bigcup_{n=0}^{\infty} \phi_n \right) \cup \delta \\
&= \bigcup_{n=0}^{\infty} \left( R \circ \phi_n \right) \cup \delta \\
&= \bigcup_{n=0}^{\infty} \left( R \circ \phi_n \cup \delta \right) \\
&= \bigcup_{n=0}^{\infty} F(\phi_n) \\
\end{align*}
\]

This means, Theorem 142 is applicable. Let \(F_0 = F(\emptyset) = \delta\), and \(F_{i+1} = F(F_i)\) for \(i \geq 0\). Thus,

\[
\text{lfp}(F) = \bigcup_{i=0}^{\infty} F_i = \bigcup_{i=0}^{\infty} R^i = R^*
\]

By this, the lemma is proved.

As a consequence, we prove that \(R^\#\) indeed defines the reflexive, transitive closure of \(R\).
Corollary 144 \( R = \bigcup_{i=0}^{\infty} R^i \) Let \( R \subseteq X \times X \) and let \( Q \subseteq X \times X \) be defined as above, and let \( R \subseteq Q \). Furthermore, assume is such that \( Q \) is reflexive, transitive and . Then

\[ R^* \subseteq Q. \]

Proof. By the Knaster–Tarski theorem it is enough to verify that \( Q \) is a prefixpoint of \( F = \lambda \phi (R \circ \phi \cup \delta) \). By the reflexivity of \( Q \), we have \( \delta \subseteq Q \), moreover, by assumption, \( R \subseteq Q \), which, together with transitivity, yields \( F(Q) = R \circ Q \cup \delta \subseteq Q \).
Appendix B. Exercises

1. Operational semantics

We illustrate the methods presented in the previous sections through some solved exercises.

**Exercise 1.** Let \( C = Y := 1; Z := X; \text{while } Z > 1 \text{ do } Y := Y \ast Z; Z := Z - 1 \text{ od } \). Let \( s = \begin{pmatrix} X & Y & Z \\ 3 & 0 & 0 \end{pmatrix} \). Present at least five steps of the computation in the operational semantics starting from the configuration \( \langle C, s \rangle \). In the second member of the configurations below, the tuples \( (x, y, z) \) stand for the values \( s(X), s(Y) \) and \( s(Z) \), in this order.

**Solution.**

\[
\begin{align*}
\langle C, (3, 0, 0) \rangle & \rightarrow \langle C_1, (3, 1, 0) \rangle \\
\langle C_{11}, (3, 1, 3) \rangle & \rightarrow \langle C_{110}; C_{11}, (3, 1, 3) \rangle \\
\langle C_{110}; C_{11}, (3, 3, 3) \rangle & \rightarrow \langle C_{11}, (3, 3, 2) \rangle \\
\langle C_{110}; C_{11}, (3, 3, 2) \rangle & \rightarrow \langle C_{110}; C_{11}, (3, 6, 2) \rangle \\
\langle C_{11}, (3, 6, 1) \rangle & \rightarrow \langle 3, 6, 1 \rangle
\end{align*}
\]

where

\[
\begin{align*}
C_1 &= Z := X; \text{while } Z > 1 \text{ do } Y := Y \ast Z; Z := Z - 1 \text{ od } \\
C_{11} &= \text{while } Z > 1 \text{ do } Y := Y \ast Z; Z := Z - 1 \text{ od } \\
C_{110} &= Y := Y \ast Z; Z := Z - 1 \\
C_{1101} &= Z := Z - 1
\end{align*}
\]

**Exercise 2.** Let \( C = Y := 1; Z := X; \text{while } Z > 1 \text{ do } Y := Y \ast Z; Z := Z - 1 \text{ od } \) be the factorial program as in the previous exercise. Construct the set \( Op(C) \) in the style of Definition 12. We preserve the notation of the previous exercise concerning the subparts of program \( C \).

**Solution.**

\[
\begin{align*}
Op(C) &= \{ \langle Y := 1; C_1, s \rangle, \langle C_1, s[Y/1] \rangle \} \cup Op(C_1) \\
&= \{ \langle Y := 1; C_1, s \rangle, \langle C_1, s[Y/1] \rangle \} \\
&\cup \{ \langle Z := X; C_{11}, s \rangle, \langle C_{11}, s[Z/X] \rangle \} \cup Op(C_{11}) \\
&= \{ \langle Y := 1; C_1, s \rangle, \langle C_1, s[Y/1] \rangle \} \\
&\cup \{ \langle Z := X; C_{11}, s \rangle, \langle C_{11}, s[Z/X] \rangle \} \\
&\cup \{ \langle C_{11}, s \rangle, \langle C_{110}; C_{11}, s \rangle \mid s(Z) > 1 \} \\
&\cup \{ \langle C_{110}; C_{11}, s \rangle, \langle C_{1101}; C_{11}, s[Y/Y \ast Z] \rangle \} \\
&\cup \{ \langle C_{110}; C_{11}, s \rangle, \langle C_{11}, s[Z/Z - 1] \rangle \} \\
&\cup \{ \langle C_{11}, s \rangle, s[Z] \mid s(Z) \leq 1 \}
\end{align*}
\]
Exercise 3. Construct a computational (transitional) sequence for \( C \) in Exercise 1 starting from making use of the operational semantics defined in the previous exercise.

Solution.

\[
\begin{align*}
\langle Y := 1; C_1, (4, 2, 1) \rangle, \langle C_1, (4, 1, 1) \rangle & \rightarrow \langle C_1, (4, 1, 1) \rangle, \langle C_{11}, (4, 1, 4) \rangle \\
\langle C_{11}, (4, 1, 4), C_{110}; C_{11}, (4, 1, 4) \rangle & \rightarrow \langle C_{110}; C_{11}, (4, 1, 4), C_{110}; C_{11}, (4, 4, 4) \rangle \\
\langle C_{110}; C_{11}, (4, 4, 4), (C_{11}, (4, 4, 3)) \rangle & \rightarrow \langle C_{11}, (4, 4, 3), C_{110}; C_{11}, (4, 4, 3) \rangle \\
\langle C_{110}; C_{11}, (4, 4, 3), C_{110}; C_{11}, (4, 12, 3) \rangle & \rightarrow \langle C_{110}; C_{11}, (4, 12, 3), C_{11}, (4, 12, 2) \rangle \\
\langle C_{11}, (4, 12, 2), (C_{110}; C_{11}, (4, 12, 2)) \rangle & \rightarrow \langle C_{110}; C_{11}, (4, 12, 2), C_{110}; C_{11}, (4, 24, 2) \rangle \\
\langle C_{110}; C_{11}, (4, 24, 2), C_{110}; C_{11}, (4, 24, 1) \rangle & \rightarrow \langle C_{11}, (4, 24, 1), C_{110}; C_{11}, (4, 24, 1) \rangle \\
\langle C_{110}; C_{11}, (4, 24, 1), (C_{110}; C_{11}, (4, 24, 1)) \rangle & \rightarrow \langle C_{11}, (4, 24, 0), (C_{11}, (4, 24, 0)) \rangle \\
\end{align*}
\]

Exercise 4. Let \( C = X := 1; Y := n; \text{while } Y > 0 \text{ do } X := m * X; Y := Y - 1 \text{ od} \). Let \( s = \begin{pmatrix} X & Y \\ 3 & 5 \end{pmatrix} \), assume \( n_1 = 3, m_1 = 4 \). Present a computation in the operational semantics starting from the configuration \( \langle C, s \rangle \). In the second member of the configurations below, the tuples \((x, y)\) stand for the values \( s(X), s(Y) \), in this order.

Solution.

\[
\begin{align*}
\langle C, (3, 5) \rangle & \rightarrow \langle C_1, (1, 5) \rangle \\
\langle C_{11}, (1, 3) \rangle & \rightarrow \langle C_{110}; C_{11}, (1, 3) \rangle \\
\langle C_{110}; C_{11}, (4, 3) \rangle & \rightarrow \langle C_{11}, (4, 2) \rangle \\
\langle C_{11}, (4, 2) \rangle & \rightarrow \langle C_{110}; C_{11}, (16, 2) \rangle \\
\langle C_{11}, (16, 1) \rangle & \rightarrow \langle C_{110}; C_{11}, (16, 1) \rangle \\
\langle C_{110}; C_{11}, (64, 1) \rangle & \rightarrow \langle C_{11}, (64, 0) \rangle \\
\langle (64, 0) \rangle & \rightarrow \langle (64, 0) \rangle \\
\end{align*}
\]

where

\[
\begin{align*}
C_1 & = Y := n; \text{while } Y > 0 \text{ do } X := m * X; Y := Y - 1 \text{ od} \\
C_{11} & = \text{while } Y > 0 \text{ do } X := m * X; Y := Y - 1 \text{ od} \\
C_{110} & = X := m * X; Y := Y - 1 \\
C_{1101} & = Y := Y - 1
\end{align*}
\]

Exercise 5. Let \( C \) be as in Exercise 4. Let us keep the notation for the labels of \( C \) of the above exercise. Formulate the operational semantics of \( C \) on the pattern of 12. Let \( n_1 = 3 \) and \( m_1 = 4 \) as before.

Solution.
Exercise 6. Construct a computational (transitional) sequence for $C$, if $C$ is as in Exercise 4, starting from
\[
\begin{pmatrix}
X \\
Y
\end{pmatrix}
\]
making use of the operational semantics defined in the previous exercise. Assume $n = 3$ and $m = 2$.

Solution.

\[
\begin{align*}
&\langle X := 1; C_1, (3, 2) \rangle, \langle C_1, (1, 2) \rangle \rightarrow \langle C_1, (1, 3) \rangle, \langle C_{11}, (1, 3) \rangle \\
&\langle C_{11}, (1, 3) \rangle, \langle C_{110}; C_{111}, (1, 3) \rangle \rightarrow \langle C_{110}; C_{111}, (1, 3) \rangle, \langle C_{110}; C_{111}, (2, 3) \rangle \\
&\langle C_{110}; C_{111}, (2, 3) \rangle, \langle C_{110}; C_{111}, (2, 2) \rangle \rightarrow \langle C_{110}; C_{111}, (2, 2) \rangle, \langle C_{110}; C_{111}, (2, 2) \rangle \\
&\langle C_{110}; C_{111}, (2, 2) \rangle, \langle C_{110}; C_{111}, (4, 2) \rangle \rightarrow \langle C_{110}; C_{111}, (4, 2) \rangle, \langle C_{110}; C_{111}, (4, 1) \rangle \\
&\langle C_{110}; C_{111}, (4, 1) \rangle, \langle C_{110}; C_{111}, (4, 1) \rangle \rightarrow \langle C_{110}; C_{111}, (4, 1) \rangle, \langle C_{110}; C_{111}, (8, 1) \rangle \\
&\langle C_{110}; C_{111}, (8, 1) \rangle, \langle C_{110}; C_{111}, (8, 0) \rangle \rightarrow \langle C_{110}; C_{111}, (8, 0) \rangle, \langle C_{110}; C_{111}, (8, 0) \rangle
\end{align*}
\]

2. Denotational semantics

Exercise 7. Let $C = X := a; Y := b; \text{while } X \neq Y \text{ do if } X > Y \text{ then } X := X - Y \text{ else } Y := Y - X \text{ fi od}$. Construct the denotational semantics of $C$, as in Definition 16.

Solution. Let
\[
\begin{align*}
C_1 &= Y := b; \text{while } X \neq Y \text{ do if } X > Y \text{ then } X := X - Y \text{ else } Y := Y - X \text{ fi od} \\
C_{11} &= \text{if } X > Y \text{ then } X := X - Y \text{ else } Y := Y - X \text{ fi od} \\
C_{110} &= \text{if } X > Y \text{ then } X := X - Y \text{ else } Y := Y - X \text{ fi od} \\
C_{1100} &= X := X - Y \\
C_{1101} &= Y := Y - X
\end{align*}
\]

\[
\begin{align*}
C &= C_1 \circ X := a \\
&= (C_1 \circ Y := b) \circ X := a \\
&= (lf_p(\Delta) \circ \{(s, s[Y/b]) \mid s \in S\}) \circ \{(s, s[X/a]) \mid s \in S\} \\
&= \left(\bigcup_{i=0}^{\infty} \Delta_i\right) \circ \{(s, s[Y/b]) \mid s \in S\} \circ \{(s, s[X/a]) \mid s \in S\} \\
&= \bigcup_{i=0}^{\infty} \left(\left(\bigcup_{i=0}^{\infty} \Delta_i\right) \circ \{(s, s[Y/b]) \mid s \in S\}\right) \circ \{(s, s[X/a]) \mid s \in S\},
\end{align*}
\]

where $\Delta_i$ are computed as follows.
3. Partial correctness in Hoare logic

Exercise 8. Let $C$ be the program computing the greatest common divisor of $a$ and $b$ of Exercise 7. Prove the following partial correctness formula:

$$\{\text{true}\}C\{X = (a, b)\}.$$  

We adopt the notation of Exercise 7 concerning the labelling of $C$.

Solution. We present the proof in the Hoare partial correctness calculus, in linear style. As loop invariant we choose the formula $P = \{\gcd(X, Y) = \gcd(a, b)\}$.

Exercise 9. Let $C$ be the program of Exercise 4. Construct a proof for the partial correctness formula

$$\{n \geq 0\}C\{X = m^n\}.$$  

We make use of the labels defined for $C$ in Exercise 4. Again, we give the proof in a linear form, for invariant we choose the formula $P = \{(X \cdot m^Y = m^n \land Y \geq 0)\}$.

Solution.
Exercise 10. Let 

\[ C = Z := 1; \]

while \( Y \neq 0 \) do

if odd(Y) then

\[ Y := Y - 1; Z := Z \times X \]

else

\[ Y := Y \div 2; X := X \times X \]

fi

od

We introduce the following labels.

\[ C_1 = \text{while } Y \neq 0 \text{ do } i \text{ j odd}(Y) \text{ then } Y := Y - 1; Z := Z \times X \text{ else } Y := Y \div 2; X := X \times X \text{ fi } od \]

\[ C_{10} = \text{if odd}(Y) \text{ then } Y := Y - 1; Z := Z \times X \text{ else } Y := Y \div 2; X := X \times X \text{ fi } \]

\[ C_{100} = Y := Y - 1; Z := Z \times X \]

\[ C_{101} = X := X := Y \div 2; X := X \times X \]

\[ C_{1001} = Z := Z \times X \]

\[ C_{1011} = X := X \times X \]

We prove the partial correctness assertion \( \{ X = m \land Y = n \} C \{ Z = m^n \} \) by giving a correct proof outline for it. We introduce \( P = \{ Z \times X^Y = m^n \} \), as invariant.

Solution.

\[ \{ X = m \land Y = n \} \]

\[ \{ X^Y = m^n \} \]

\[ Z := 1; \]

\[ \{ Z \times X^Y = m^n \} \]

while \( Y \neq 0 \) do
\{Z \ast X^Y = m^n \land Y \neq 0\}\]

if odd(Y) then

\{Z \ast X^Y = m^n \land Y \neq 0 \land \text{odd}(Y)\}\]

\{(Z \ast X) \ast X^{Y-1} = m^n\}\]

Y:=Y-1;

\{(Z \ast X) \ast X^Y = m^n\}\]

Z:=Z \ast X

\{Z \ast X^Y = m^n\}\]

else

\{Z \ast X^Y = m^n \land Y \neq 0 \land \lnot\text{odd}(Y)\}\]

\{Z \ast (X \ast X)^{Y \div 2} = m^n\}\]

Y:=Y \div 2;

\{Z \ast (X \ast X)^Y = m^n\}\]

X:=X \ast X

\{Z \ast X^Y = m^n\}\]

fi

\{Z \ast X^Y = m^n\}\]

od

\{Z \ast X^Y = m^n \land Y = 0\}\]

\{Z = m^n\}\]

Additionally, in order to ensure that we obtained a correct proof outline, we need to prove the relations

\[X = m \land Y = n \Rightarrow X^Y = m^n,\]

\[Z \ast X^Y = m^n \land Y \neq 0 \land \text{odd}(Y) \Rightarrow (Z \ast X) \ast X^{Y-1} = m^n,\]

\[Z \ast X^Y = m^n \land Y \neq 0 \land \lnot\text{odd}(Y) \Rightarrow Z \ast (X \ast X)^{Y \div 2} = m^n,\]

\[Z \ast X^Y = m^n \land Y = 0 \Rightarrow Z \ast X^Y = m^n,\]

where, as usual, \(P \Rightarrow Q\) denotes the fact that the sets of states represented by \(P\) is a subset of the states where \(Q\) hold. All the above relations are trivial arithmetical facts.

**Exercise 11.** Let the base set of the underlying interpretation be words over an alphabet \(\Sigma\).

Let

\[C = Y := \lambda ;\]
\[ Z := X; \]
\[ \text{while } Z \neq \lambda \text{ do} \]
\[ Y := f(Z)Y; \]
\[ Z := t(Z) \]
\[ \text{od}, \]

where \( f(X) \) is the head, \( t(X) \) is the tail of a non-empty word \( X \), otherwise both of them are the empty word. Let \( w^R \) denote the reverse of \( w \). We construct a correct proof outline for the partial correctness assertion \( \{ X = w \} C\{ Y = w^R \} \). For this purpose, we use the invariant \( P = \{ Z^R Y = w^R \} \).

**Solution.**

\[
\{ X = w \} \\
\{ X^R \lambda = w^R \} \\
Y := \lambda; \\ \\
\{ X^R Y = w^R \} \\
Z := X; \\ \\
\{ Z^R Y = w^R \} \\
\text{while } Z \neq \lambda \text{ do} \]
\[
\{ Z^R Y = w^R \land Z \neq \lambda \} \\
\{ t(Z)^R f(Z)Y = w^R \} \\
Y := f(Z)Y; \\ \\
\{ t(Z)^R Y = w^R \} \\
Z := t(Z) \\ \\
\{ Z^R Y = w^R \} \\
\text{od} \]
\[
\{ Z^R Y = w^R \land Z = \lambda \} \\
\{ Y = w^R \} \\
\]

In order to complete the proof, we have to prove the implications

\[
X = w \Rightarrow X^R \lambda = w^R, \\
Z^R Y = w^R \land Z \neq \lambda \Rightarrow t(Z)^R f(Z)Y = w^R, \\
Z^R Y = w^R \land Z = \lambda \Rightarrow Y = w^R, \\
\]

but they are trivially valid.
Exercise 12. Let

\[ C = Y := \lambda ; \]
\[ Z := X ; \]

while \( Z \neq \lambda \) do

if \( f(Z) = f(t(Z)) \)
then \( Y := Y f(Z) \)
else \( Y := Y f(Z) f(Z) \)
fi
\[ Z := t(Z) \]

od

Prove the correctness of the formula \( \{ X = w \} C \{ Y = inc(w) \} \), where \( inc(w) \) is obtained from \( w \) by incrementing by one the lengths of the character sequences consisting of the same character. For example, let \( w = abbaab \), then \( inc(w) = aabbaaab \). We apply as loop invariant the formula \( Y inc(Z) = inc(w) \).

Solution.

\[ \{ X = w \} \]
\[ \{ inc(X) = inc(w) \} \]
\[ Y := \lambda ; \]
\[ \{ Y inc(X) = inc(w) \} \]
\[ Z := X ; \]
\[ \{ Y inc(Z) = inc(w) \} \]

while \( Z \neq \lambda \) do

\[ \{ Y inc(Z) = inc(w) \land Z \} \]

if \( f(Z) = f(t(Z)) \)
\[ \{ Y inc(Z) = inc(w) \land Z \neq \lambda \land f(Z) = f(t(Z)) \} \]
\[ \{ Y f(Z) inc(t(Z)) = inc(w) \} \]
then \( Y := Y f(Z) \)
\[ \{ Y inc(t(Z)) = inc(w) \} \]
else
\[ \{ Y inc(Z) = inc(w) \land Z \neq \lambda \land f(Z) \neq f(t(Z)) \} \]
\{Y f(Z) f(Z) inc(t(Z)) = inc(w)\}

Y = Yf(Z) f(Z)
\{Y inc(t(Z)) = inc(w)\}

\fi
\{Y inc(t(Z)) = inc(w)\}
Z := t(Z)
\{Y inc(Z) = inc(w)\}
od
\{Y inc(Z) = inc(w) \land Z = \lambda\}
\{Y = inc(w)\}

To complete the proof we must prove the following implications:

\[ X = w \Rightarrow inc(X) = inc(w) , \]
\[ Y inc(Z) = inc(w) \land Z \neq \lambda \land f(Z) = f(t(Z)) \Rightarrow Y f(Z) inc(t(Z)) = inc(w) , \]
\[ Y inc(Z) = inc(w) \land Z \neq \lambda \land f(Z) \neq f(t(Z)) \Rightarrow Y f(Z) f(Z) inc(t(Z)) = inc(w) , \]
\[ Y inc(Z) = inc(w) \land Z = \lambda \Rightarrow Y = inc(w) \]

Among these implications the first and the last one are trivial, though the precise verification of the second and third one would need a proof by induction on the lengths of the words involved. In such cases, we omit the rigorous proofs of the statements formulated in the interpretation, we just rely on our intuition to estimate whether a certain assertion in the

Exercise 13. Let
\[ C = Y := \lambda ; \]
\[ Z := X ; \]
while \( Z \neq \lambda \) do
\[ \text{if } f(Y) = f(Z) \]
\[ \text{then } Z := t(Z) \]
\[ \text{else} \]
\[ Y := Y f(Z) \]
\[ \text{fi} \]
\[ \text{od} \]

Prove the correctness of the formula \( \{ X = w \} C \{ Y = red(w) R \} \), where \( red(w) \) is obtained from \( w \) by substituting every sequence of identical elements by one specimen of that element. For example, let \( w = abbaab \), then \( red(w) = abab \). The reverse operator is defined as before. We apply as loop invariant the formula \( Y^R red(Z) = red(w) \).

Solution.
\[
\begin{align*}
\{ X = \text{red}(w) \} \\
\{ \text{red}(X) = \text{red}(w) \} \\
Y &:= \lambda; \\
\{ Y^R \text{red}(X) = \text{red}(w) \} \\
Z &:= X; \\
\{ Y^R \text{red}(Z) = \text{red}(w) \} \\
\text{while } Z \neq \lambda \text{ do} \\
\{ Y^R \text{red}(Z) = \text{red}(w) \land Z \neq \lambda \} \\
\text{if } f(Y) = f(Z) \text{ then} \\
\{ Y^R \text{red}(Z) = \text{red}(w) \land Z \neq \lambda \land f(Y) = f(Z) \} \\
\{ Y^R \text{red}(t(Z)) = \text{red}(w) \} \\
Z &:= t(Z) \\
\{ Y^R \text{red}(Z) = \text{red}(w) \} \\
\text{else} \\
\{ Y^R \text{red}(Z) = \text{red}(w) \land Z \neq \lambda \land f(Y) \neq f(Z) \} \\
\{ (Y^f(Z))^R \text{red}(Z) = \text{red}(w) \} \\
Y &:= Y^f(Z) \\
\{ Y^R \text{red}(Z) = \text{red}(w) \} \\
\text{fi} \\
\{ Y^R \text{red}(Z) = \text{red}(w) \} \\
\text{od} \\
\{ Y^R \text{red}(Z) = \text{red}(w) \land Z = \lambda \} \\
\{ Y = \text{red}(w)^R \}
\end{align*}
\]

The proof becomes complete if we check the validity of the implications below:

\[
\begin{align*}
X = \text{red}(w) & \Rightarrow \text{red}(X) = \text{red}(w), \\
Y^R \text{red}(Z) = \text{red}(w) \land Z \neq \lambda \land f(Y) = f(Z) & \Rightarrow Y^R \text{red}(t(Z)) = \text{red}(w), \\
Y^R \text{red}(Z) = \text{red}(w) \land Z \neq \lambda \land f(Y) \neq f(Z) & \Rightarrow (Y^f(Z))^R \text{red}(Z) = \text{red}(w), \\
Y^R \text{red}(Z) = \text{red}(w) \land Z = \lambda & \Rightarrow Y = \text{red}(w)^R
\end{align*}
\]

**Exercise 14.** Let

\[ C := 1; \]
Y:=n;
while Y > 1 do
    X:=X*Y;
    Y:=Y-2
od

Prove the correctness of the formula $\{n \geq 0\} C \{X = n!!\}$, where

$$n!! = \begin{cases} n \times (n - 2)!! & \text{if } n \geq 2 \\ 1 & \text{if } n = 0 \text{ or } n = 1. \end{cases}$$

Solution. We give a proof in derivation tree form now. We choose $P = (X \times Y!! = n!! \land Y \geq 0)$ as invariant. We construct the tree step by step, proceeding from the simpler to the more compound one. First of all, we prove $\{ P \land Y > 1 \} C_0 \{ P \}$, where $C_0$ is the body of the while loop.

\[
\begin{align*}
(P[ Y / 2 ][ X / X \times Y ], X := X \times P[Y/Y-2] \land Y := Y-2 \{ P \}) & \quad (\text{comp}) \\
\{ P \land Y > 1 \} X := X \times Y; Y := Y-2 \{ P \} & \\
\{ P \} \text{while } Y > 1 \text{ do } X := X \times Y; Y := Y-2 \text{ od} \{ P \land Y \leq 1 \}
\end{align*}
\]

Let $\mathcal{D}$ denote the above proof tree. Then

\[
\begin{align*}
P \land Y > 1 & \Rightarrow P[Y/Y-2][X/X \times Y] & \mathcal{D} \\
\{ P \land Y > 1 \} X := X \times Y; Y := Y-2 \{ P \} & \\
\{ P \} \text{while } Y > 1 \text{ do } X := X \times Y; Y := Y-2 \text{ od} \{ P \land Y \leq 1 \}
\end{align*}
\]

If $\mathcal{E}$ denotes the proof tree obtained as above and $\mathcal{F}$ stands for the proof tree below

\[
\begin{align*}
(P[Y/n][X/1], X := 1 \{ P[Y/n] \}) & \quad (\text{comp}) \\
\{ P[Y/n][X/1], X := 1; Y := n \{ P \} \}
\end{align*}
\]

then

\[
\begin{align*}
n \geq 0 & \Rightarrow P[Y/n][X/1] & \quad (\text{comp}) \\
\{ P[Y/n][X/1], C \{ P \land Y \leq 1 \} \} & \quad \mathcal{F} \quad \mathcal{E} \\
\{ n \geq 0 \} C \{ X = n!! \}
\end{align*}
\]

is the proof tree searched for. The reader may have the impression that presenting a proof in a deduction tree form might be a more exhaustive task on the person constructing the proof than a proof in linear style. It seems indeed that this is the case, that’s why we prefer proofs written in linear style or in the form of a proof outline. In what follows, proofs will be presented in linear form proofs or as proof outlines in most of the cases.

Exercise 15. Let

C = Y:=0;

while $Y^2 \leq n$ do
    Y:=Y+1
Exercises

Prove $\{n \geq 0\} C \{Y = \lceil \sqrt{n} \rceil\}$, where $\lceil r \rceil$ denotes the greatest integer not greater than $r$ for a non-negative real number $r$.

**Solution.** We present the proof of the partial correctness assertion $\{n \geq 0\} C \{Y = \lceil \sqrt{n} \rceil\}$ in the form of a proof outline, providing at the same time a detailed verification of the validity of the proof outline. Let us choose $P = (n > 0 \cap (Y - 1)^2 \leq n) \land (n = 0 \cap Y = 0)$ as loop invariant. Our first aim is to support with a valid proof outline the assertion $\{P\} \text{while } Y^2 \leq n \text{ do } Y := Y + 1 \text{ od} \{P \land Y^2 > n\}$.

\[
\begin{align*}
P \land Y^2 \leq n &\Rightarrow P[Y/Y + 1] \quad \{P[Y/Y + 1]\}Y := Y + 1\{P\} \quad P \Rightarrow P \\
\{P \land Y^2 \leq n\}\{P[Y/Y + 1]\}Y := Y + 1\{P\}\{P\} \\
\{P \land Y^2 \leq n\}Y := Y + 1\{P\} \\
\{inv : P\} \text{while } Y^2 \leq n \text{ do } \{P \land Y^2 \leq n\}Y := Y + 1\{P\} \text{ od } \{P \land Y^2 > n\}
\end{align*}
\]

Let

\[
C_1 = \begin{align*}
\text{ while } Y^2 \leq n \text{ do } Y := Y - 1 \text{ od}; Y := Y + 1
\end{align*}
\]

\[
C_{10} = \begin{align*}
\text{ while } Y^2 \leq n \text{ do } Y := Y - 1 \text{ od}
\end{align*}
\]

be labels for $C$, let us denote the proof outlines corresponding to the formula $\{R\}W[Q]$ by $\{R\}W^*[Q]$ for some formulas $P$, $Q$ and command $W$. Thus, let $\{inv : P\}C_{10}^*[P \land Y^2 > n]$ stand for the proof outline obtained as the last row of the above derivation. Then we have

\[
\begin{align*}
\{inv : P\}C_{10}^*[P \land Y^2 > n] \quad \{P \land Y^2 > n\}Y := Y - 1\{P\} \\
\{P[Y/0]\}Y := 0\{P\} \quad \{inv : P\}C_{10}^*[P \land Y^2 > n]Y := Y + 1\{P[Y/Y + 1]\} \\
\{P[Y/0]\}Y := 0\{inv : P\}C_{10}^*[P \land Y^2 > n] \land (Y + 1)^2 > n
\end{align*}
\]

Denoting the last proof outline as $\{P[Y/0]\}C^*\{P[Y/Y + 1] \land (Y + 1)^2 > n\}$, we acquire

\[
\begin{align*}
n \geq 0 \Rightarrow P[Y/0] \quad \{P[Y/0]\}C^*[P[Y/Y + 1] \land (Y + 1)^2 > n] \quad P[Y/Y + 1] \land (Y + 1)^2 > n \\
\{n \geq 0\}{\{P[Y/0]\}C^*[P[Y/Y + 1] \land (Y + 1)^2 > n]} \land (Y = \sqrt{n}^+) \land (n \geq 0)C^*[Y = \sqrt{n}^+]
\end{align*}
\]

which is what was desired.

**Exercise 16.** Let

\[
C = Y := 0; \\
X := 1 \\
\text{while } X \leq n \text{ do } \\
\quad X := 2 \ast X \\
\quad Y := Y + 1
\]

\[
\text{end while}
\]

\[
\text{end program}
\]
Prove the validity of \( \{ n > 0 \} C \{ Y = \lceil \log_2(n) \rceil \} \).

**Solution.** We prove the partial correctness formula \( \{ n > 0 \} C \{ Y = \lceil \log_2(n) \rceil \} \) by presenting a valid proof outline for it. Let us choose
\[
P = (Y - 1 \leq \log_2(n) \land X = 2^Y)
\]
as loop invariant. Let
\[
\begin{align*}
C_1 &= X := 1; \text{while } X \leq n \text{ do } X := 2 \times X; Y := Y + 1 \text{ od}; Y := Y - 1 \\
C_{11} &= \text{while } X \leq n \text{ do } X := 2 \times X; Y := Y + 1 \text{ od}; Y := Y - 1 \\
C_{110} &= \text{while } X \leq n \text{ do } X := 2 \times X; Y := Y + 1 \text{ od}
\end{align*}
\]
be labels for \( C \).

Let \( \{ \text{inv} : P \} C_{110}^* \{ P \land X > n \} \) denote the proof outline obtained in the final line of the derivation tree. Then we have
\[
\{ \text{inv} : P \} C_{110}^* \{ P \land X > n \} \quad \{ P \land X > n \} Y := Y - 1 \{ P[Y/Y + 1] \land X > n \} \quad \{ \text{inv} : P \} C_{11}^* \{ P[Y/Y + 1] \land X > n \}
\]
and
\[
\{ P[X/1] \} X := 1 \{ P \} \quad \{ \text{inv} : P \} C_{11}^* \{ P[Y/Y + 1] \land X > n \} \quad \{ P[X/1] \} X := 1 \{ \text{inv} : P \} C_{11}^* \{ P[Y/Y + 1] \land X > n \} \quad \{ P[X/1] \} Y := 0 \{ P[X/1] \} \quad \{ P[X/1] \} Y := 0 \{ P[X/1] \} C_{11}^* \{ P[Y/Y + 1] \land X > n \} \quad \{ P[X/1] \} Y := 0 \{ P[X/1] \} C_{11}^* \{ P[Y/Y + 1] \land X > n \}
\]
where \( C_{11}^* = X := 1; C_{11}^* \) and \( C_{11}^* = Y := 0; C_{11}^* \). Finally, since
\[
P[Y/Y + 1] \land X > n \quad \text{yields} \quad \log_2(n) - 1 < Y \leq \log_2(n),
\]
which is equivalent to
\[
Y = \lceil \log_2(n) \rceil,
\]
we have
\[
\{ n > 0 \} \{ P[X/1] \} Y := 0 \{ P[X/1] \} C_{11}^* \{ P[Y/Y + 1] \land X > n \} \quad \{ P[Y/Y + 1] \land X > n \} \quad \{ n > 0 \} \{ P[X/1] \} Y := 0 \{ P[X/1] \} C_{11}^* \{ P[Y/Y + 1] \land X > n \} \quad \{ Y = \lceil \log_2(n) \rceil \}
\]
which completes the proof.

**Exercise 17.** Let
\[
C = \{ X := 2; \\
Y := 1 \\
\text{while } X \leq n \text{ do} \\
\text{if } X \mid n \text{ then}
\]

\begin{align*}
Y & := Y + 1 \\
\text{else} \\
& \text{skip} \\
\text{fi} \\
X & := X + 1 \\
\text{od;}
\end{align*}

Prove the validity of \( \{ n > 0 \} C \{ Y = \mu(n) \} \), where \( \mu(n) \) is the number of the divisors of \( n \).

**Solution.** We present a valid proof outline for the demonstration of the partial correctness formula. Let \( \mu_{<m}(n) \) denote the number of divisors of \( n \) less than \( m \), that is,

\[
\mu_{<m}(n) = \{ d \mid (d \mid n) \wedge d < m \}.
\]

We choose \( P = Y = \mu_{<X}(n) \land X \leq n + 1 \) as an invariant for the while loop.

\[
\{ n > 0 \} \\
\{ 1 = \mu_{<2}(n) \land 2 \leq n + 1 \} \\
X := 2; \\
\{ 1 = \mu_{<X}(n) \land X \leq n + 1 \} \\
Y := 1 \\
\{ \text{inv} : Y = \mu_{<X}(n) \land X \leq n + 1 \} \\
\text{while } X \leq n \text{ do} \\
\{ \text{inv} : Y = \mu_{<X}(n) \land X \leq n + 1 \land X \leq n \} \\
\text{if } X \mid n \text{ then} \\
\{ Y = \mu_{<X}(n) \land X \leq n + 1 \land X \leq n \land X \mid n \} \\
\{ Y + 1 = \mu_{<X+1}(n) \land X + 1 \leq n + 1 \} \\
Y := Y + 1 \\
\{ Y = \mu_{<X+1}(n) \land X + 1 \leq n + 1 \} \\
\text{else} \\
\{ Y = \mu_{<X}(n) \land X \leq n + 1 \land X \leq n \mid n \} \\
\{ Y = \mu_{<X+1}(n) \land X + 1 \leq n + 1 \} \\
\text{skip} \\
\{ Y = \mu_{<X+1}(n) \land X + 1 \leq n + 1 \} \\
\text{fi}
\]
Exercises

\{ Y = \mu_{X+1}(n) \land X + 1 \leq n + 1 \}\n
X:=X+1

\{ Y = \mu_{X}(n) \land X \leq n + 1 \}\n
od;

\{ Y = \mu_{X}(n) \land X \leq n + 1 \land X > n \}\n
\{ Y = \mu_{n+1}(n) = \mu(n) \}\n
To complete the proof, it remains to justify the implications as follows.

\begin{align*}
n > 0 \implies 1 &= \mu_{2}(n) \land 2 \leq n + 1, \\
Y = \mu_{X}(n) \land X \leq n + 1 \land X \leq n \land X \mid n &\implies Y + 1 = \mu_{X+1}(n) \land X + 1 \leq n + 1, \\
Y = \mu_{X}(n) \land X \leq n + 1 \land X \leq n \land X \mid n &\implies Y = \mu_{X+1}(n) \land X + 1 \leq n + 1, \\
Y = \mu_{X}(n) \land X \leq n + 1 \land X > n &\implies Y = \mu(n)
\end{align*}

All of the above relations represent straightforward arithmetical facts.

**Exercise 18.** Finding loop invariants is the non-trivial part of proving partial correctness. The next example illustrates a situation like this, where the loop invariant might need a little ingenuity to find. Let

C = X:=n;
P:=2;
Y:=0;
while X > 1 do
  if P \mid X then
    X:=X \div P;
    Y:=Y+1
  else
    P:=P+1
  fi
od

Prove the validity of \( \{ n > 0 \} C \{ Y = \alpha(n) \} \), where \( \alpha(b) \) denotes the integer division of \( a \) by \( b \), and \( \alpha(n) \) is the number of the prime divisors of \( n \) with multiplicity.

**Solution.** We have to find a suitable invariant reflecting precisely the operation of the while loop. Let \( I = \{ Y = \alpha(n \div X \land X \geq 1 \land 2 \leq P \leq \text{mindiv}(X)) \} \), where \( \text{mindiv}(m) \) should denote the least proper divisor of \( m \). Observe that \( \text{mindiv}(m) \) is always prime. Then we can construct the following proof outline

\{ n > 0 \} \{ I \mid Y/0 \mid [P/2][X/n] \}

X:=n;
\{I[Y/0][P/2]\}
P:=2;
\{I[Y/0]\}
Y:=0;
\{I\}
while \(X > 1\) do
    \{I \land X > 1\}
    if \(P \mid X\) then
        \{I \land X > 1 \land P \mid X\}
        \{I[Y/Y + 1][X/X \div P]\}
        X:=X \div P;
        \{I[Y/Y + 1]\}
        Y:=Y+1
        \{I\}
    else
        \{I \land X > 1 \land P \nmid X\}
        \{I[P/P + 1]\}
        P:=P+1
        \{I\}
    fi
\{I\}
\od
\{I \land X \leq 1\}
\{Y = \alpha(n)\}

Again, to complete the proof we have to check the validity of the implications below.

\[n > 0 \implies I[Y/0][P/2][X/n],\]
\[I \land X > 1 \land P \mid X \implies I[Y/Y + 1][X/X \div P],\]
\[I \land X > 1 \land P \nmid X \implies I[P/P + 1],\]
\[I \land X \leq 1 \implies Y = \alpha(n)\]

They are all straightforward arithmetical relations with the possible exception of the second and third ones, the justifications of which relies on the fact of \(P\) being the minimal proper
divisor of \( n \), which is just the minimal prime divisor of \( n \). Hence, 
\[ \alpha(n \div n) = \alpha(n \div (X \div P)) + n \div P + 1 \leq \min_{\div}(X) \], if 
\( P \div X \). All the other relations follow easily.

4. **Total correctness of while programs**

**Exercise 19.** Let
\[
C = \begin{array}{l}
X := 1; \\
Y := 0; \\
\text{while } Y < n \text{ do} \\
\quad X := m \times X; \\
\quad Y := Y + 1 \\
\text{od}
\end{array}
\]

Prove the validity of the total correctness assertion \([n \geq 0]C[X = m^n]\).

**Solution.** The proof of total correctness needs a different formulation of the while rules in the Hoare calculus compared to partial correctness proofs. As a first example we give the proof as a derivation tree for the total correctness formula \([n \geq 0]C[X = m^n]\). As a loop invariant we choose \( P = (X = m^Y \land Y \leq n) \). Let us introduce the following labels.

\[
\begin{align*}
C_1 & = Y := 0; \text{while } Y < n \text{ do } X := m \times X; Y := Y + 1 \text{ od} \\
C_{11} & = \text{while } Y < n \text{ do } X := m \times X; Y := Y + 1 \text{ od} \\
C_{110} & = X := 2 \times X; Y := Y + 1
\end{align*}
\]

First of all, we demonstrate the validity of \([P \land Y < n]C_{110}[P]\).

To apply the while rule for total correctness, we have to show that a quantity expressed by an arithmetical expression decreases at every execution of the while loop. In other words, we have to find a bound function \( t \). For \( t \) we can choose \( n - Y \). Then we have to prove \([P \land Y < n \land t = Z]C_{110}[t < Z]\) for a new variable \( Z \). The last premiss of the while rule, \( P \land Y < n \Rightarrow t \geq 0 \) follows trivially.

If \( E \) denotes the above proof tree, then, taking into account \((t < Z)[Y/Y + 1][X/m \times X] = n - (Y + 1)\) we have

\[
\begin{align*}
P \land Y < n \land t = Z \Rightarrow n - (Y + 1) < Z \\
& \text{ E} \\
\end{align*}
\]

\[
\begin{align*}
P \land Y < n \land t = Z = X := m \times X; Y := Y + 1[t < Z]
\end{align*}
\]
Taking all these into consideration, we can deduce by the while rule for total correctness. Furthermore,

\[
(n \geq 0) \Rightarrow P[Y / 0][X / 1] = (1 - n) \land 0 \leq n
\]

\[
(n \geq 0) \Rightarrow X := 1; Y := 0[P]
\]

and

\[
(n \geq 0) \Rightarrow X := 1; Y := 0[P]
\]

\[
[P] \text{ while } Y < n \text{ do } X := m \ast X; Y := Y + 1 \text{ od } [P \land Y \geq n]
\]

\[
(n \geq 0) \Rightarrow [P \land Y \geq n]
\]

which was to be proved.

**Exercise 20.** Let

\[C = \begin{array}{l}
\text{while } n < Y^2 \text{ do } \\
\quad Y := Y - 1 \\
\text{od}
\end{array}
\]

Let

\[C = \begin{array}{l}
\text{while } Y := ((n + 1)/2) \text{ do } \\
\quad Y := Y - 1 \\
\text{od}
\end{array}
\]

Prove the validity of the total correctness assertion \([n \geq 0] C[Y = \sqrt{n}^2]\), where \(\sqrt[n]{\cdot} : \mathbb{R}_{\geq 0} \rightarrow \mathbb{N}_{\geq 0}\) is defined as in Exercise 15.

**Solution.** Total correctness proofs enjoy a certain decomposition property: for a proof of \([P] C[Q]\), it is enough to prove \([P] C[Q]\) and then verifying termination by showing \([P] C[true]\). We are going to follow this path in the present case. The proof is again presented in the form of a derivation tree. Below, let \(C_1\) stand for the while loop of \(C\), and let \(P = ((Y + 1)^2 > n)\) be an invariant for the while loop.

\[
\begin{array}{c}
\frac{P \land n < Y^2}{P[Y/Y - 1]} \\
\frac{P[Y/Y - 1]}{Y := Y - 1[P]}
\end{array}
\]

\[
\begin{array}{c}
\{P \land n < Y^2\} Y := Y - 1[P] \\
\{P\} \text{ while } n < Y^2 \text{ do } Y := Y - 1 \text{ od } [P \land n \geq Y^2]
\end{array}
\]

Moreover,

\[
\begin{array}{c}
\frac{P[Y/((n + 1)/2)]}{Y := ((n + 1)/2)[P]} \\
\frac{P[Y/((n + 1)/2)]}{C_1[P \land n \geq Y^2]}
\end{array}
\]

\[
\begin{array}{c}
\{P[Y/((n + 1)/2)]\} C_1[P \land n \geq Y^2] \\
\{P[Y/((n + 1)/2)]\} C[P \land n \geq Y^2]
\end{array}
\]

In addition,

\[
\begin{array}{c}
\frac{n \geq 0}{P[Y/((n + 1)/2)]} \\
\frac{P[Y/((n + 1)/2)]}{C[P \land n \geq Y^2]}
\end{array}
\]

which is the partial correctness assertion desired. To verify total correctness, we have to prove termination yet. To this end, the provability of \([n \geq 0] C[true]\) needs to be checked. As for the invariant of the while rule we can settle \(P = (Y \geq 0 \land n \geq 0)\). Then the proof of \([P \land n < Y^2] Y := Y - 1[P]\) is immediate. Let \(t = Y^2 - n\). Then
where the validity of \( P \land n < Y^2 \land t = U \Rightarrow t[Y - 1] < U \) follows from the fact that \( Y \geq 0 \land n \geq 0 \) and \( Y^2 > n \) together imply \( Y \geq 1 \), by which \( (Y - 1)^2 < Y^2 \) follows. \( P \land n < Y^2 \land t = U \land Y = Y - 1[t < U] \)

**Exercise 21.** Let \( C \) be the program of Example 34 computing the greatest common divisor of \( a \) and \( b \).

\[
C = \begin{array}{l}
X := a; \\
Y := b; \\
\text{while } X \neq Y \text{ do} \\
\quad \text{if } X > Y \text{ then} \\
\quad \quad X := X - Y \\
\quad \text{else} \\
\quad \quad Y := Y - X \\
\text{fi} \\
\text{od}
\end{array}
\]

Verify the validity of \( [a > 0 \land b > 0] \) \( C \equiv (a, b) \), where \((a, b)\) denotes the greatest common divisor of \( a \) and \( b \).

**Solution.** If we apply the decomposition rule, a proof outline showing \( \{a > 0 \land b > 0\} C \equiv (a, b) \) is already given in Example 34. Thus, it remains to prove termination only by presenting a proof outline for \( [a > 0 \land b > 0] C[true] \). As for \( P \), let us choose \( P = (X > 0 \land Y > 0) \), and let \( t = X + Y \). Then

\[
\begin{align*}
[a > 0 \land b > 0] \\
[P[Y/b][X/a]] \\
X := a; \\
[P[Y/b]] \\
Y := b; \\
[P] \\
\text{while } X \neq Y \text{ do} \\
[P \land X \neq Y] \\
\text{if } X > Y \text{ then} \\
[P \land X \neq Y \land X > Y] \\
[P[X/X - Y]]
\end{align*}
\]
The only interesting implications are
\[ P \land X \neq Y \land X \leq Y \]
\[ P[Y/Y - X] \]
\[ Y := Y - X \]
\[ P \]
fi
\[ P \]
od
\[ P \land X = Y \]
\[ \text{true} \]

Exercise 22. Let
\[ C = X := n; \]
\[ Y := 1; \]
\[ Z := 1; \]
while \( X > 1 \) do
\[ X := X - 1; \]
\[ Y := Y + 2; \]
\[ Z := Y + Z \]
od
Verify the total correctness assertion \([0 < n] C[Z = n^2]\).

Solution. We apply the decomposition rule again. To start out with, we check the validity of
\[ \{0 < n\} C[Z = n^2] \]
Let us choose
\[ P = (X \geq 1 \land Y = 2(n - X) + 1 \land Z = (n + 1 - X)^2) \]
as loop invariant.
\[ \{0 < n\} \]


Exercises

\[
\{ P[Z/1][Y/1][X/n] \}
\]

\begin{verbatim}
X:=n;
\{ P[Z/1][Y/1] \}
Y:=1;
\{ P[Z/1] \}
Z:=1;
\{ P \}
while X > 1 do
\{ P \wedge X > 1 \}
\{ P[Z/Y + Z][Y/Y + 2][X/X - 1] \}
X:=X-1;
\{ P[Z/Y + Z][Y/Y + 2] \}
Y:=Y+2;
\{ P[Z/Y + Z] \}
Z:=Y+Z
\{ P \}
\end{verbatim}

\[
\{ P \wedge X \leq 1 \}
\]

\[
\{ Z = n^2 \}
\]

As usual, we have to check the validity of the implications formed by subsequent assertions of the proof. The only nontrivial one is

\[
X \geq 1 \wedge X > 1 \wedge Y = 2(n-X)+1 \wedge Z = (n+1-X)^2 \Rightarrow X - 1 \geq 1 \wedge Y = 2(n-X)+3 \wedge Y + Z = (n+2-X)^2,
\]

which follows by well-known arithmetical equalities. What is left is the proof of \([n > 0] C[true]\). As loop invariant we can choose \(I = (X \geq 1)\) and \(t = X\) suffices as bound function. We omit the straightforward proof.

**Exercise 23.** Let \(\Sigma\) be an alphabet. Assume \(a\) is a word over \(\Sigma\). Let

\[
C = Z:=w;
\]

\[
Y:=a;
\]

while \(Z \neq \lambda\) do

if \(f(Z)=f(Y)\) then

skip

else


The functions \( f \) and \( t \) on words are defined as before. Let \( \text{trim}(u,v) \) be the function on words over \( \sum \) such that \( \text{trim}(u,v) \) is \( v \) with all the occurrences of the first character of \( u \) deleted from \( v \), otherwise, if \( u \) is empty, \( \text{trim}(u,v) = v \). For example, \( \text{trim}(ab,abba) = bbb \). Prove \( |a| = 1 \implies Y = \text{trim}(a,w) \).

**Solution.** We prove something more general: \( [a \neq \lambda] \implies Y = \text{trim}(f(a), w) \). From this, the original claim follows as a special case. In accordance with the decomposition rule we split the claim into two statements: \( [a \neq \lambda] \implies Y = \text{trim}(f(a), w) \) and \( [a \neq \lambda] \implies \text{true} \). We present valid proof outlines for both assertions. We deal with the partial correctness assertion first. As invariant, set \( P = (t(Y) \cdot \text{trim}(f(Y), Z) = t(a) \cdot \text{trim}(f(a), w)) \). For the sake of readability, we have explicitly indicated word concatenations in the previous equality.

\[
\begin{align*}
\{w \neq \lambda\} \\
\{P[Y/a][Z/w]\} \\
Z := w; \\
\{P[Y/a]\} \\
Y := a; \\
\{P\} \\
while Z \neq \lambda do \\
\{P \land Z \neq \lambda\} \\
if f(Z) = f(Y) then \\
\{P \land Z \neq \lambda \land f(Z) = f(Y)\} \\
\{P[Z/t(Z)]\} \\
skip \\
\{P[Z/t(Z)]\} \\
else \\
\{P \land Z \neq \lambda \land f(Z) \neq f(Y)\} \\
\{P[Z/t(Z)][Y/Yf(Z)]\} \\
Y := Yf(Z)
\end{align*}
\]
The nontrivial implications are

\[
P \land Z \neq \lambda \land f(Z) = f(Y) \quad \Rightarrow \quad P[Z/t(Z)][Y/Yf(Z)]
\]
\[
P \land Z \neq \lambda \land f(Z) \neq f(Y) \quad \Rightarrow \quad P[Z/t(Z)][Y/Yf(Z)]
\]
\[
P \land Z = \lambda \quad \Rightarrow \quad t(Y) \cdot \text{trim}(f(t(Y)), Z) = t(a) \cdot \text{trim}(f(a), w) \land Z = \lambda
\]

The latter one hinges in the fact that \( Z = \lambda \) implies \( \text{trim}(f(Y), Z) = \text{trim}(f(t(Y)), Z) = \lambda \).

It remains to check \( [w \neq \lambda]C[\text{true}] \). It is enough to choose \( P = \text{true} \) as loop invariant and \( t = |Z| \) as bound function. Then \( [Z \neq \lambda \land |Z| = U]C_1[|Z| < U] \), where \( C_1 \) is the body of the loop, and \( Z \neq \lambda \lor t \geq 0 \) follow easily.

**Exercise 24.** Let

\[
C = Z := w;
\]
\[
Y := \lambda;
\]

while \( Z \neq \lambda \) do

if \( f(Z) = f(t(Z)) \) then

\[
Y := Yf(Z)
\]

else

skip

fi;

Z := t(Z)

od
Verify the total correctness assertion \( \overline{\text{true}}C[Y = \text{dec}(w)] \), where \( \text{dec}(w) \) removes a character from each character sequence of \( w \) consisting of the same symbols. For example,

**Solution.** As before, we prove first the partial correctness assertion \( \{\text{true}\}C \{Y = \text{dec}(w)\} \), then give a valid proof outline for the claim \( \overline{\text{true}}C[\text{true}] \).

For loop invariant, we settle \( P = (Y \text{dec}(Z) = \text{dec}(w)) \).

\[
\begin{align*}
\{\text{true}\} \\
\{P[Y/\lambda][Z/w]\} \\
C = Z = w; \\
\{P[Y/\lambda]\} \\
Y := \lambda; \\
\{P\} \\
\text{while } Z \neq \lambda \text{ do} \\
\quad \{P \land Z \neq \lambda\} \\
\quad \text{if } f(Z) = f(t(Z)) \text{ then} \\
\quad \quad \{P \land Z \neq \lambda \land f(Z) = f(t(Z))\} \\
\quad \quad \{P[Z/t(Z)][Y/Yf(Z)]\} \\
\quad \quad Y := Yf(Z) \\
\quad \quad \{P[Z/t(Z)]\} \\
\quad \text{else} \\
\quad \quad \{P \land Z \neq \lambda \land f(Z) \neq f(t(Z))\} \\
\quad \quad \{P[Z/t(Z)]\} \\
\quad \text{skip} \\
\quad \{P[Z/t(Z)]\} \\
\text{fi;} \\
\{P[Z/t(Z)]\} \\
Z := t(Z) \\
\{P\} \\
\text{od} \\
\{P \land Z = \lambda\} \\
\{Y = \text{dec}(w)\}
\end{align*}
\]
As usual, we have to verify the validity of the following relations:

\[
\begin{align*}
true & \implies P[Y/\lambda,Z/w] \\
P \land Z \neq \lambda \land f(Z) = f(t(Z)) & \implies P[Z/t(Z)][Y/Yf(Z)] \\
P \land Z \neq \lambda \land f(Z) \neq f(t(Z)) & \implies P[Z/t(Z)] \\
P \land Z = \lambda & \implies Y = dec(w)
\end{align*}
\]

For the justification of the implications it is enough to check the facts below:

\[
\begin{align*}
f(u) = f(t(u)) & \implies dec(u) = f(u)dec(t(u)) \\
f(u) \neq f(t(u)) & \implies dec(u) = dec(t(u))
\end{align*}
\]

They can be verified by induction on the length of \( u \). What is left is to demonstrate \([true]C[true]\). For bound function, we choose \( I = |Z| \), and the loop invariant we define as \( I = true \). Then the necessary conditions for a valid proof outline are trivially satisfied.

5. The wp-calculus

**Exercise 25.** Let

\[
C = \text{while } X < 0 \text{ do } \\
X := X + 1; \\
Y := Y - 2 \\
\text{od}
\]

Determine the value of \( wp(C, Y \geq 0) \).

**Solution.** First of all, we apply Lemmas 52 and 53 for finding the formula requested. In addition, in what follows we also take into account the following straightforward properties of the wp-calculus.

\[
\begin{align*}
wp(C, false) = false \\
wp(C, Q_1 \land Q_2) &= wp(C, Q_1) \land wp(C, Q_2) \\
wp(C, Q_1 \lor Q_2) &= wp(C, Q_1) \lor wp(C, Q_2) \\
\text{If } Q_1 \supset Q_2, \text{ then } wp(C, Q_1) &\implies wp(C, Q_2)
\end{align*}
\]

With this in hand, we can expand \( wp(C, Y \neq 0) \) by Lemmas 52 and 53:
By induction on \( i \) we can prove the following equality

\[
P_i = X < 0 \land X \geq -i \land Y \geq 2i
\]

provided

\[
P = (X \geq 0 \land Y \geq 0) \lor (X < 0 \land (\exists i > 0)(X \geq -i \land Y \geq 2i))
\]

Thus, \( P \) can be chosen as the weakest precondition of \( Y \geq 0 \) with respect to \( C \).

**Exercise 26.** Let

\[
C = \text{while } Y \leq 2n \text{ do}
\]

\[
X := X + Y;
\]

\[
Y := Y + 2
\]

od

Determine \( \wp(C, 2 \mid Y + 1 \land X = n^2) \) provided the values of \( X \) and \( Y \) are integers.

**Solution.**
By induction on $i$ we can prove that, if $i > 0$, then

$$P_i = (Y = 2(n-i) + 1 \land X = (n-i)^2).$$

Observe, that, if $n > i$, then $Y$ becomes negative, but the relations with $Y$ and $X$ still hold. Hence,

$$wp(C, 2 \mid Y + 1 \land X = n^2) = (Y > 2n \land 2 \mid Y + 1 \land X = n^2)$$

$$\vee \ (\exists i > 0) (Y = 2(n-i) + 1 \land X = (n-i)^2)$$

Exercise 27. Let

$$C = \text{while } Y \neq \lambda \text{ do}$$

$$Z:=f(Y)Z;$$

$$Y:=t(Y)$$

od

Determine $wp(C, Z = w)$.

Solution.
In general, we obtain

\[ P^i_0 = Y = \lambda \land Z = w \]

\[ P^i_1 = Y \neq \lambda \land \wp(Z := f(Y)Z; Y := t(Y), P_0) \]

\[ = Y \neq \lambda \land \wp(Z := f(Y)Z, \wp(Y := t(Y), P_0)) \]

\[ = Y \neq \lambda \land \wp(Z := f(Y)Z, t(Y) = \lambda \land Z = w) \]

\[ = Y \neq \lambda \land t(Y) = \lambda \land f(Y)Z = w \]

\[ = |Y| = 1 \land Y^RZ = X \]

\[ P^i_2 = Y \neq \lambda \land \wp(Z := f(Y)Z; Y := t(Y), P_1) \]

\[ = Y \neq \lambda \land \wp(Z := f(Y)Z, \wp(Y := t(Y), P_1)) \]

\[ = Y \neq \lambda \land \wp(Z := f(Y)Z, |t(Y)| = 1 \land t(Y)^RZ = w) \]

\[ = Y \neq \lambda \land |t(Y)| = 1 \land t(Y)^Rf(Y)Z = w \]

\[ = |Y| = 2 \land Y^RZ = w \]

In general, we obtain

\[ P^i = |Y| = i \land Y^RZ = w \]

provided \( i > 0 \). Thus, \( \wp(C, Z = w) = (Y^RZ = w) \).

**Exercise 28.** Let

\[ C = \text{while } Y \neq \lambda \text{ do} \]

\[ Z := Zf(Y)Y ; \]

\[ Y := t(Y) \]

od

Determine \( \wp(C, Z = X) \).

**Solution.** Let \( w^D \) denote the word \( w_1 w_2 \ldots w_k w^k \), if \( w = w_1 \ldots w_k \). Then
Exercises

By induction on \( i \) we obtain

\[ P_i = |Y| = i \land Z Y^D = X. \]

Thus, \( wp(C, Z = X) = Z Y^D = X \).

6. Recursive programs

In the first part of this section we deal with partial correctness of recursive programs, then we will see some examples for proving total correctness in the presence of recursion.

Exercise 29. procedure F;

begin

if \( X \neq 1 \) then

\( X := X \div 2; \)

F;

\( X := X + 1 \)

else

\( X := 0 \)

fi;

\( X := n \)

F;

F

Prove the validity of \( \{ n > 2 \} C\{ x = log_2^* (log_2^*(n)) \} \), with the help of \( \{ 0 < n \land X = n \} F\{ X = log_2^*(n) \} \), where \( C \) is the recursive program and \( log_2^* \) denotes the function \( \lceil log_2 n \rceil \).
Solution. First of all, we prove \( \{0 < n \land X = n\} \implies \{X = \log_2(n)\} \) by applying the recursion rule. This amounts to verifying from the assumption

\[ \{0 < n \land X = n\} \implies \{X = \log_2(n)\} \]

where \( D \) is the body of the procedure. We present the proof in the form of a proof outline.

procedure F:

\[ \begin{align*}
\{0 < n \land X = n\} \\
\text{if } X \neq 1 \text{ then} \\
\{0 < n \land X = n \land X \neq 1\} \\
\{0 < n \land X = n\} \\
X := X \div 2; \\
\{0 < n \land X = n \div 2\} \\
F; \\
\{X = \log_2(n \div 2)\} \\
X := X + 1 \\
\{X = \log_2(n)\} \\
\text{else} \\
\{0 < n \land X = n \land X = 1\} \\
\{0 = \log_2(n)\} \\
X := 0 \\
\{X = \log_2(n)\} \\
\text{fi} \\
\{X = \log_2(n)\}
\end{align*} \]

We have to verify \( \{0 < n \land X = n \div 2\} \implies \{X = \log_2(n \div 2)\} \) from the assumption \( \{0 < n \land X = n\} \implies \{X = \log_2(n)\} \) by applying the adaptation rule. Let \( x^- = \{x\} \) and \( y^- = \{n\} \). Then, on one hand, we have

\[ \forall x (\forall n (0 < n \land X = n \supset x = \log_2(n) \supset x = \log_2(n \div 2))) \implies \{X = \log_2(n \div 2)\}, \]

on the other hand

\[ 0 < n \land X = n \div 2 \land \forall n (0 < n \land X = n \supset X = \log_2(n) \supset X = \log_2(n \div 2)), \]
which involves
\[ \{0 < n \land X = n \div 2\} F\{X = \log_2^*(n \div 2)\} \]

by the consequence rule. Hence, by the recursion rule, we can assert
\[ \{0 < n \land X = n\} F\{X = \log_2^*(n)\} . \]

Prior to proving \( \{n > 2\} C\{x = \log_2^*(\log_2^*(n))\} \), we have to make an observation. By Theorem 82, it follows that \( C \) is without side effect with respect to the variables outside of \( \text{Var}(C) \). This implies \( \{p\} C\{p\} \) for any assertion \( p \) not containing variables from \( C \). With this in hand, we can formulate the following proof outline.

\[ \{n > 2\} \]
\[ \{2 < n \land n = n\} \]
\( X:=n \)
\[ \{2 < n \land X = n\} \]
\( F; \)
\[ \{2 < n \land X = \log_2^*(n)\} \]
\[ \{0 < \log_2^*(n) \land X = \log_2^*(n)\} \]
\( F \)
\[ \{X = \log_2^*(\log_2^*(n))\} \]

where, for the subsequent assertions, the upper one implies the lower one easily, taking into consideration \( \{0 < n \land X = n\} F\{X = \log_2^*(n)\} \) together with the above remark.

**Exercise 30.** Let

procedure F;

begin
if \( X > 2 \) then
    \( X:=X-3; \)
    \( F; \)
    \( X:=X+1 \)
else
    \( X:=0 \)
fi
end;
\( X:=40; \)
\( F; \)
\( F; \)
Prove \( \{ \text{true} \} C \{ X = 1 \} \), where \( C \) is the main recursive program, provided \( \{ n > 0 \land X = n \} F \{ X = n \div 3 \} \).

**Solution.** We wish to apply the recursion rule first, to this end, by assuming \( \{ n \geq 0 \land X = n \} F \{ X = n \div 3 \} \) we seek to prove \( \{ n \geq 0 \land X = n \} D \{ X = n \div 3 \} \), where \( D \) is the body of \( F \). We give the proof in the form of a proof outline. \( \{ n \geq 0 \land X = n \} \)

procedure \( F \):

begin

\( \{ n \geq 0 \land X = n \} \)

if \( X > 2 \) then

\( \{ n \geq 0 \land X = n \land X > 2 \} \)

\( X := X - 3; \)

\( \{ n \geq 0 \land X = n \land X > -1 \} \)

\( \{ n - 3 \geq 0 \land X = n - 3 \} \)

\( F: \)

\( \{ X = (n - 3) \div 3 \} \)

\( \{ X + 1 = n \div 3 \} \)

\( X := X + 1 \)

\( \{ X = n \div 3 \} \)

else

\( \{ n \geq 0 \land X = n \land X \leq 2 \} \)

\( \{ 0 = n \div 3 \} \)

\( X := 0 \)

\( \{ X = n \div 3 \} \)

fi

\( \{ X = n \div 3 \} \)

end;

\( \{ X = n \div 3 \} \)

We have to justify by the adaptation rule \( \{ n - 3 \geq 0 \land X = n - 3 \} F \{ X = (n - 3) \div 3 \} \). Let \( x^- = \{ x \} \) and \( y^- = \{ n \} \). Then we have
\{\forall x (\forall n (n \geq 0 \land X = n \supset x = n \div 3) \supset x = (n - 3) \div 3)\} F\{X = (n - 3) \div 3\},

moreover,

\(n - 3 \geq 0 \land X = n - 3 \land \forall n (n \geq 0 \land X = n \supset X = n \div 3) \supset X = (n - 3) \div 3\),

which, by the consequence rule, yields

\(\{n - 3 \geq 0 \land X = n - 3\} F\{X = (n - 3) \div 3\}\).

Now we are in a position to prove \(\{true\} C\{X = 1\}\).

\{true\}

\{40 \geq 0 \land X = 40\}

\(X := 40;\)

\{40 \geq 0 \land X = 40\}

\(F;\)

\{X = 40 \div 3\}

\{0 \leq 13 \land X = 13\}

\(F;\)

\{X = 13 \div 3\}

\{0 \leq 4 \land X = 4\}

\(F\)

\{X = 4 \div 3\}

\{X = 1\}

Exercise 31. Let

procedure \(F\);

begin

if \(X \geq Y\) then

\(X := X - Y\)

\(F;\)

\(Y := Y + Z\)

else

\(Y := 1\)

fi

end

be procedure \(F\), and let
C := 5
Y := 3;
Z := 5;

F be the recursive program. Prove \{true\} C\{Y = 6\} with the help of
\{0 ≤ k ∧ 0 < m ∧ 0 ≤ n ∧ X = k ∧ Y = m ∧ Z = n\} F\{Y = (k/m)Z + 1 ∧ Z = n\}, where \((k/m)\) denotes the integer part of the result of dividing k by m.

**Solution.** For the sake of tractability, we apply the notation
\(A(k, m, n, X, Y, Z) = (0 ≤ k ∧ 0 < m ∧ 0 ≤ n ∧ X = k ∧ Y = m ∧ Z = n)\)
and \(B(k, m, n, Y, Z) = (Y = (k/m)Z + 1 ∧ Z = n)\). In order to verify \(\{A(k, m, n, X, Y, Z)\} F\{B(k, n, Y, Z)\}\), we assume the previous relation and demonstrate that \(\{A(k, m, n, X, Y, Z)\} D\{B(k, n, m, Y, Z)\}\) holds, where D is the body of procedure F.

\(\{A(k, m, n, X, Y, Z)\}\)

procedure F;

begin

\(\{A(k, m, n, X, Y, Z)\}\)

if \(X ≥ Y\) then

\(\{A(k, m, n, X, Y, Z) ∧ X ≥ Y\}\)

\(\{A(k − m, m, n, Y, Y, Z) ∧ k ≥ m\}\)

\(X := X − Y\)

\(\{A(k − m, m, n, X, Y, Z) ∧ k ≥ m\}\)

\(F:\)

\(\{B(k − m, n, m, Y, Z) ∧ k ≥ m\}\)

\(\{B(k, n, m, Y + Z, Z)\}\)

\(Y := Y + Z\)

\(\{B(k, n, m, Y, Z)\}\)

else

\(\{A(k, m, n, X, Y, Z) ∧ X < Y\}\)

\(\{B(k, n, m, 1, Z)\}\)

\(Y := 1\)

\(\{B(k, n, m, Y, Z)\}\)
We have to check, by applying the adaptation rule, the deducibility of 
\{ A(k - m, m, n, X, Y, Z) \land k \geq m \} F \{ B(k - m, n, m, Y, Z) \land k \geq m \} .
Let 
x^- = \{ x, y, z \}
and 
y^- = \{ k, m, n \}.

Then 
\{ \forall xy(z \forall k m n (A(k, m, n, X, Y, Z) \supset B(k, a, m, y, z)) \supset B(k - m, n, m, y, z)) \} F \{ B(k - m, n, m, Y, Z) \}
and 
A(k - m, m, n, X, Y, Z) \land \forall k m n (A(k, m, n, X, Y, Z) \supset B(k, n, m, Y, Z))
implies 
B(k - m, n, m, Y, Z), by which, making use of the consequent rule, we conclude 
\{ A(k - m, m, n, X, Y, Z) \} F \{ B(k - m, n, m, Y, Z) \}.
Taking into account the fact that 
change(D) \cap \{ k, m, n \} = 0,
we can deduce 
\{ k \geq m \} F \{ k \geq m \}.
Putting all these together, we obtain 
\{ A(k, m, n, X, Y, Z) \} F \{ B(k, n, m, Y, Z) \},
as desired.
Finally,
\{ true \}
\{ A(5,3,5,5,3,5) \}
X:=5
\{ A(5,3,5,5,3,5) \}
Y:=3;
\{ A(5,3,5,X,3,5) \}
Z:=5;
\{ A(5,3,5,X,Y,5) \}
F
\{ B(5,3,5,Y,Z) \}
\{ Y = 6 \},
where the justification of 
\{ A(5,3,5,X,Y,Z) \} F \{ B(5,3,5,Y,Z) \}
requires one more application of the adaptation rule.

Exercise 32. Let
procedure F;
begin
if 2 < X then
    X:=X-1;
F;
Y:=Y+2;
\[
Z := Y + Z + 1
\]
else
\[
Y := 2;
\]
\[
Z := 4
\]
fi
end;
X := n;
F;
X := Z;
F

Demonstrate making use of \( C \) is the main recursive program.

**Solution.** By the rule of recursion for total correctness, in order to prove
\[
[2 \leq m \land X = m] F[Y + 2 = 2m \land Z = m^2]
\]
we have to infer
\[
[2 \leq m \land X = m \land t < U] D[Y + 2 = 2m \land Z = m^2]
\]
with an appropriately chosen \( t \), where \( D \) is the body of \( F \). Let \( t = X \). Then
\[
[2 \leq m \land X = m \land X = U]
\]
procedure \( F \):
begin
\[
[2 \leq m \land X = m \land X = U]
\]
if \( 2 < X \) then
\[
[2 \leq m \land X = m \land X = U \land X > 2]
\]
\[
X := X - 1;
\]
\[
[2 \leq m \land X + 1 = m \land X + 1 = U \land X + 1 > 2]
\]
\[
[2 \leq m - 1 \land X = m - 1 \land X < U]
\]
F:
\[
[Y + 2 = 2(m - 1) \land Z = (m - 1)^2]
\]
\[
[Y + 4 = 2m \land Y + Z + 3 = m^2]
\]
Y := Y + 2;
\[
[Y + 2 = 2m \land Y + Z + 1 = m^2]
\]
Z := Y + Z + 1
[\[ Y + 2 = 2m \land Z = m^2 \]\]

else
[\[ 2 \leq m \land X = m \land X = U \land X \leq 2 \]\]
[\[ 2 + 2 = 2m \land 4 = m^2 \]\]
Y := 2;
[\[ Y + 2 = 2m \land 4 = m^2 \]\]
Z := 4
[\[ Y + 2 = 2m \land Z = m^2 \]\]
fi
[\[ Y + 2 = 2m \land Z = m^2 \]\]
end
[\[ Y + 2 = 2m \land Z = m^2 \]\]

Let \( x^- = \{x, y, z\} \) and \( y^- = \{m\} \). The adaptation rule yields

\[ \forall z : (2 \leq m \land X = m \land X < U \lor y + 2 = 2m \land z = m^2) \lor y + 2 = 2(m - 1) \land z = (m - 1)^2) \land Y + 2 = 2(m - 1) \land Z = (m - 1)^2 \]

moreover,
\[ 2 \leq m - 1 \land X = m - 1 \land X < U \land Y + 2 = 2(m - 1) \land Z = (m - 1)^2 \]

together yield \( Y + 2 = 2(m - 1) \land Z = (m - 1)^2 \), which was to be proved. The recursion rule for total correctness can be applied only if the additional condition
\[ 2 \leq m \land X = m \lor X \geq 0 \]
fulfills, but this is straightforward. Hence, by the recursion rule, \([2 \leq m \land X = m] \land Y + 2 = 2m \land Z = m^2\). With this in hand

\[ [n > 1] \]
\[ [2 \leq n \land n = n] \]
X := n;
[\[ 2 \leq n \land X = n \]\]
F:
[\[ Y + 2 = 2n \land Z = n^2 \land n \geq 2 \]\]
X := Z;
[\[ Y + 2 = 2n \land X = n^2 \land n \geq 2 \]\]
[\[ 2 \leq n^2 \land X = n^2 \]\]
F
[\[ Y + 2 = 2n^2 \land Z = (n^2)^2 \]\]
\([Z = n^4]\)

We also have made use of the fact that \(n \notin Var(F)\), thus, \([n \geq 2]F[n \geq 2]\). By this, the proof is completed.
Bibliography