Mathematically Rigorous Software Design

an electronic textbook
by Robert L. Baber

2002 September 9

McMaster University
Department of Computing and Software
SFWR ENG/COMP SCI 4/6L03

Table of Contents

1. Introduction

1.1 General

Characteristic of every engineering field is a theoretical, scientific foundation upon which most of the engineer's work is based. Engineers continually, systematically, consciously and even subconsciously use mathematical models when designing and analyzing artifacts. This basis for engineering work is a prerequisite for achieving the reliability of designs to which engineers, their clients and society have become accustomed.

This electronic book presents such a theoretical, scientific foundation for designing software and for proving mathematically that it is correct, i.e. that it satisfies its specification. In particular, this book presents a mathematical model of a program and its execution which enables one to design a program to satisfy a given specification and to prove that the program does, in fact, satisfy that specification.

Today's engineering disciplines did not always have such scientific foundations and mathematical models for designing and analyzing their machines, devices and systems. The magnificent new warship Wasa sank on her maiden voyage in 1628 primarily because her designers did not know how to analyze her proposed hull shape and weight distribution in order to determine – before construction – whether or not she would be stable in the water. In the 1800s many bridges in Europe collapsed under the weight of the new locomotives, mainly because their designers did not know how to verify in advance that the bridge's proposed structure was capable of carrying the planned loads. In 1858 the first successfully laid transatlantic telegraph cable was destroyed by an excessively high input voltage. In all of these cases, the designers lacked predictive models of the artifacts they were designing, models which would enable them to determine with confidence and before construction whether or not their designs would satisfy the requirements placed on them. At those times, the several fields were in their pre-engineering phase.

In today's software development practice, similar failures occur for similar reasons, e.g. the explosion of the Ariane 5 in 1996. Software development practice is currently in its pre-engineering phase. As with the Wasa, large bridges in the 1800s, early electrical communication systems, etc., today's software developers are constructing wonderful, impressive systems, but too often those systems fail with spectacular consequences. Software developers do not yet have and regularly use predictive mathematical models of the software they design. Sooner or later, this situation will change and software development will enter its engineering phase. You, the students of today, will bring about and experience this transition during your professional careers. The material you learn from this book will help you do this.

In the now classical engineering disciplines, engineers regularly base their work on mathematical models and laws relating the various relevant variables. In the future, the same will apply in software engineering. The relevant mathematical models include variables such as the following:

Electrical engineering	Wasa/nautical engineering	Software engineering
voltage	angle of heel	precondition
current	righting torque	postcondition
resistance	heeling torque (wind on sails)	loop invariants
inductance	rotational energy	intermediate conditions
capacitance		program segment

Another prerequisite for designing large systems that work is precisely defined interfaces between the various subsystems and components. This book will identify the information required in interface specifications for software systems.

1.2 Overview of this book

The goals of this course and book are:

- to familiarize the student and reader with the basic concepts underlying correctness proofs for computer software,
- to illustrate how they can be applied to designing software,
- to show how they can be practically used to verify the correctness of a program, i.e. that the program fulfills its specification and
- to develop the student's ability to apply these concepts and techniques to practical software design problems.

The topics covered in this book are:

- A mathematical model of a program and its execution, consisting of:
  - program variable,
  - data environment,
  - value of an ordinary variable, of an array variable and of an expression in the context of a data environment,
  - the correspondence between Boolean expressions and sets of data environments,
  - program statements and constructs (assignment statement, declaration statement, release statement, sequence of statements, if construct, while loop, subprogram call without formal parameter passing, other loop structures) as functions on the set of data environments and the domains of these functions
- Preconditions (ordinary, semistrict, strict and complete), postconditions, partial, semitotal and total correctness
- Proof rules (lemmata) for the various program statements and constructs
- Handling other program constructs (e.g. input and output, files, subprogram calls with formal parameter passing) in correctness proofs
- Proving the correctness of a program:
  - decomposing a correctness statement (theorem) to be proved by applying the proof rules systematically and iteratively,
  - verifying the universal truth of the resulting (decomposed) propositions
- Designing a correct program:
  - implication of the proof rules for the design task,
  - design guidelines following from the proof rules and the requirements of a proof of correctness

- Summary

The scientific foundation and mathematical models for the artifacts we will be designing and analyzing – computer programs – can, like their counterparts in the classical engineering disciplines, be used

- informally and intuitively to provide general guidelines for designing the program,
- more formally to derive mathematically many parts of the program being designed and
- mathematically rigorously to verify formally that the design (the program) satisfies the specification.

This material both contributes to a better general understanding of the design task and provides a method for formally "calculating" the correctness of the final program.

1.3 Exercises

This book does not contain exercises for the reader to solve. See Appendix B below for sources of appropriate exercises.

2. Mathematical model of a program and its execution

2.1 Program variables

A *program variable* is a triple consisting of a name, a set, and an element *of that set*. The element is called the *value* of the program variable. A program variable is often called simply a variable. Note that the set may not be empty, because the value must be an element of it.

Example: (x, **Z**, 4)

2.2 Data environments

A *data environment* is a sequence of program variables.

Example: [(x, **Z**, 4), (y, **R**, 5.77), (z, Strings, “abc”), (x, **Z**, 6), (x, **Z**, 4)]

Note that a data environment may contain more than one program variable with the same name. Even identical program variables may appear in a data environment.

We write 𝔻 for the set of all data environments. We assume that this set exists (cf. Russell’s paradox). In practice this assumption is not problematic. The existence of this set can be ensured by restricting the sets allowed in the program variables to eliminate the possibility of recursive definitions and by limiting the length of allowed data environments, for example.

The state of execution of a program is represented by a data environment.

Two data environments are *equal* when they are identical in every respect. Two data environments are *structurally equal* when they are identical in every respect except the values of their program variables.

A data environment d *contains* the variable x when there is some program variable in d with the name x, i.e. when at least one term in d is a program variable whose name is x.

Example: d0 = [(x, **Z**, 4), (y, **R**, 5.77), (z, Strings, “abc”), (x, **Z**, 6), (x, **Z**, 4)]
d1 = [(x, **Z**, 4), (y, **R**, 5.77), (z, Strings, “abc”), (x, **Z**, 6), (x, **Z**, 4)]
d2 = [(x, **Z**, 4), (y, **R**, 5.77), (z, Strings, “xyz”), (x, **Z**, 6), (x, **Z**, 4)]

The data environments d0 and d1 are equal. The data environments d1 and d2 are not equal, but they are structurally equal. Each of the above data environments contains the variables x, y and z. None of the above data environments contains the variable a.

2.3 Values of variables and expressions in the context of a data environment

2.3.1 The value of an ordinary variable in a data environment

The value of the variable x in the data environment d is defined to be the value of the *first* program variable in d whose name is x. If the data environment d does not contain a program variable with the name x, then the value of x in d is undefined.

Example: d = [(x, **Z**, 4), (y, **R**, 5.77), (z, Strings, “abc”), (x, **Z**, 6)]

The value of x in d is 4 (not 6). The value of y in d is 5.77. The value of z in d is “abc”. The value of w in d is undefined.

A variable name can be viewed as a function on $\mathbb{D}$ which maps a data environment to a value. I.e., x.d=4, y.d=5.77, z.d=“abc” and w.d is undefined where d is as defined in the example above.

Alternatively, a data environment can be viewed as a function on the set of variable names that maps the variable name to a value.

2.3.2 The value of an expression in a data environment

To determine the value of an expression E in a data environment d, every variable name in E standing for the value of the named variable is replaced by the value of that variable in d (see definition above) and the resulting expression is evaluated in the usual mathematical way. The resulting value is the value of E in d.

An expression can be viewed as a function on $\mathbb{D}$ which maps a data environment to a value. Correspondingly, the value of an expression E in a data environment d is often written E.d.

Example: d = [(x, **Z**, 4), (y, **R**, 5.77), (z, Strings, “abc”), (x, **Z**, 6)]

The value of the expression (2*x+y) in d is

(2*x+y).d
=
(2*x.d+y.d)
=
2*4 + 5.77
=
13.77

2.3.3 The value of an array variable in a data environment

References to an array variable are typically of the form x(ie), where ie is an expression evaluating to an integer. The name of an array variable is not of the form x(ie), but rather x(1), x(2), etc. To determine the value of an array variable x(ie) in the data environment d, the index expression ie is first evaluated (in d) and then the value of the array variable with the corresponding index is determined as defined above.

Formally, we define the value of an array variable in a data environment as follows:

x(ie).d = x(ie.d).d

Example: d = [(x(1), **Z**, 5), (x(2), **Z**, 6), (j, **Z**, 3), (k, **Z**, 4)]

The value of x(k-j) in d is

x(k-j).d
=
x((k-j).d).d
=

x(k.d-j.d).d

=

x(4-3).d

=

x(1).d

=

5

2.4 Boolean expressions and sets of data environments

Often we will consider those data environments d for which some Boolean expression (function, condition) B is true, i.e. data environments d in the set ($\cup$ d : d$\in$ 𝔻 $\wedge$ B.d : {d}). (Note that this set is the preimage of the singleton set {true} under B.) Because of the canonical relationship between a condition and such a set, we will often use the same name (here B) for both the condition and the set. It will be clear from the context which is meant mathematically.

Conversely, we will refer to any condition which is true on a given set B and either false or undefined elsewhere by the same name B.

For a given condition the corresponding set is uniquely determined. For a given set, the corresponding condition is not uniquely determined: whether the condition is false or undefined for any particular element not in the set is left open.

2.5 Program statements and constructs as functions on 𝔻 to 𝔻

The execution of a program statement or segment upon a program execution state is viewed below mathematically as the application of a function corresponding to the program statement or segment in question to the data environment representing the state in question.

2.5.1 The assignment statement

2.5.1.1 Assignment to an ordinary variable

Informally we define the effect of executing the assignment statement x:=E (where x is a variable name and E is an expression) on the data environment d in the following way. The expression E is evaluated in d and the result becomes the new value of the *first* variable in d whose name is x.

Formally, this is defined as follows: (x:=E).d0 = d1, where

$d0 = [(N_{0,1}, S_{0,1}, V_{0,1}), (N_{0,2}, S_{0,2}, V_{0,2}), ...]$

$d1 = [(N_{1,1}, S_{1,1}, V_{1,1}), (N_{1,2}, S_{1,2}, V_{1,2}), ...]$

$N_{1,i} = N_{0,i}$ for all i=1, 2, ...

$S_{1,i} = S_{0,i}$ for all i=1, 2, ...

j = (min k : k$\in$ $\mathbf{N}_1$ $\wedge$ $N_{0,k}$ = “x” : k) [j is the index of the first variable named x]

$V_{1,i} = V_{0,i}$ for all $i \neq j$

$V_{1,j} = E.d0$

provided that $E.d0 \in S_{1,j}$ $(=S_{0,j})$ — otherwise $(N_{1,j}, S_{1,j}, V_{1,j})$ would not satisfy the definition of a program variable and d1 would not, therefore, satisfy the definition of a data environment. Note also that the above definition leads to a result only if d contains a program variable with the name x.

Note also that the above definition does not permit "side effects", i.e. the modification of the value of any variable other than x.

In the following, we will refer often to the set associated with a particular program variable. We define, therefore, (Set."x").d to be the set associated with the first program variable in d with the name x. If d contains no variable with the name x, we define the value of (Set."x").d to be the empty set. (Note that no set associated with a program variable may be empty, since the value of the variable must be an element of the set.)

The domain of the function x:=E is the set of all data environments d such that (1) d contains at least one variable with the name x, (2) E.d is defined and (3) E.d is an element of (Set."x").d. These requirements can be combined into the condition that $E.d \in$ (Set."x").d be true or, equivalently, that $(E \in (\text{Set."x"})).d$ be true.

2.5.1.2 Assignment to an array variable

The effect of executing the assignment statement x(ie):=e on the data environment d, where ie and e are expressions, is defined informally as follows: First the expression ie is evaluated in d to determine the index of the array variable to which a new value is to be assigned. Then the assignment statement is executed as described above.

Formally, we define (x(ie):=e).d to be (x(ie.d):=e).d

2.5.1.3 Multiple assignment statement

When the multiple assignment statement (x1, x2, ...):=(e1, e2, ...) is executed, each expression e1, e2, etc. is evaluated in the same initial data environment. The results are assigned to the variables x1, x2, ... respectively as in the case of the single assignment statement as defined above.

If the same variable name appears more than once on the left hand side of the multiple assignment statement, we require that the values of the corresponding expressions on the right hand side be equal, otherwise the effect of executing the multiple assignment statement is undefined. The same variable can appear more than once on the left hand side, for example, when array variables are referenced whose index expressions have the same value in the data environment in question.

2.5.1.4 The exchange statement

The exchange statement x:=:y is defined to have the same effect as the multiple assignment statement (x, y):=(y, x).

2.5.2 The declare statement

The execution of the statement declare (x, S, E) on the data environment d has the effect of creating a new program variable and prefixing it to d. Formally, we define

(declare (x, S, E)).d = [(x, S, E.d)] & d

where & is the concatenation operator, which combines two sequences into a single sequence.

In principle, one could also allow S to be an expression, in which case one would rewrite the right hand side of the above definition to read [(x, S.d, E.d)] & d. We will not make use of this possibility in this book.

If the variable to be declared is an array variable, the index expression is first evaluated as mentioned earlier. I.e. formally, (declare (x(ie), S, E)).d = (declare (x(ie.d), S, E)).d, which, in turn, equals [(x(ie.d), S, E.d)] & d.

If the data environment d already contains a variable named x and the statement declare (x, S, E) is executed on d, the resulting data environment will contain two variables named x. By executing still more declare statements, a data environment containing many variables with the same name can be formed. Subsequent references to x will pertain only to the most recently declared variable named x. The other variables with the same name are *hidden* (or *covered*). A hidden variable will become active again only when the more recently declared variables of the same name are released. The more recently declared variables of the same name are sometimes called *covering* variables.

The domain of the declare statement is comparable to that of the corresponding assignment statement with the exception that a variable with the name x need not be already contained in d. The domain of the above declare statement is the set of all data environments d such that E.d$\in$ S or, equivalently, (E$\in$ S).d.

If the variable being declared is an array variable, i.e. if the declare statement

declare (x(ie), S, E)

is being applied to the data environment d, then the value of ie in d must be an element of an appropriate set **Siv** (the **s**et of **i**ndex **v**alues, typically ($\cup$i : i$\in$ **Z** $\wedge$ 1$\leq$i : $\mathbf{Z}^{i}$), the set of all tuples of one or more integers). The domain of such a declare statement is, then, the set of all data environments d such that (E$\in$ S and ie$\in$ **Siv**).d.

The above expressions for the domains of declare statements for ordinary and array variables are different. This gives rise to no difficulty in manual proofs of program correctness, but mechanizing such proofs would be simpler if the need to distinguish between the two cases were eliminated. This can be done by considering an ordinary variable to be an array variable with an

empty tuple as its index and redefining **Siv** above to include the empty tuple. Then, **Siv** would typically be the set $(\cup i : i \in \mathbf{Z} \wedge 0 \leq i : \mathbf{Z}^i)$.

2.5.3 The release statement

Executing the statement release x on the data environment d removes the first program variable with the name x from the data environment.

Formally, (release x).d0 = d1, where

$d0 = [(N_{0,1}, S_{0,1}, V_{0,1}), (N_{0,2}, S_{0,2}, V_{0,2}), ...]$

$d1 = [(N_{1,1}, S_{1,1}, V_{1,1}), (N_{1,2}, S_{1,2}, V_{1,2}), ...]$

$j = (\min k : k \in \mathbf{N}_1 \wedge N_{0,k} = \text{“x”} : k)$ [j is the index of the first variable named x]

$N_{1,i} = N_{0,i}$ for all $i<j$
$N_{1,i} = N_{0,i+1}$ for all $i \geq j$

$S_{1,i} = S_{0,i}$ for all $i<j$
$S_{1,i} = S_{0,i+1}$ for all $i \geq j$

$V_{1,i} = V_{0,i}$ for all $i<j$
$V_{1,i} = V_{0,i+1}$ for all $i \geq j$

If the variable to be released is an array variable, the index expression is first evaluated as mentioned earlier. I.e. formally, (release x(ie)).d = (release x(ie.d)).d.

The declare and release statements are, in effect, push and pop operations on a stack of program variables of the same name embedded in the data environment.

The above definition yields a result for every d which contains a variable named x. I.e., the domain of release x is the set of all data environments d such that (Set.“x”).d≠∅ or, equivalently, (Set.“x”≠∅).d. Therefore, the Boolean condition characterizing the domain of the statement release x is Set.“x”≠∅.

If the variable being released is an array variable, then the value of the index expression in d must, of course, be an element of the set **Siv** (see section 2.5.2 above). The domain of the statement release x(ie) would, then, be (Set.“x(ie)”≠∅ and ie∈ **Siv**). But if the variable x(ie) is contained in the data environment in question, the value of its index ie must be an element of the permitted set of index values, so that Set.“x(ie)”≠∅ ⇒ ie∈ **Siv**. Thus, the term ie∈ **Siv** is redundant in the expression for the domain of a statement releasing an array variable.

2.5.4 The null (empty) statement

The null (empty) statement occurs mainly in the form of an empty then or else part of an if statement. It is defined to have no effect, i.e. null.d = d for all d$\in$𝔻. The domain of the null statement is 𝔻.

2.5.5 The sequence of statements

When a sequence (S1, S2) of statements is executed on the data environment d, first S1 is executed and then S2 is executed on the result of executing S1. I.e. formally, we define

(S1, S2).d = S2.(S1.d)

Mathematically, the function (S1, S2) is the composition of the functions S1 and S2. Sequencing statements is, therefore, associative. It is not, in general, commutative.

The domain of the sequence (S1, S2) of statements is the set of all data environments d such that d is in the preimage of 𝔻 under the function (S1, S2), i.e. such that d$\in S1^{-1}.(S2^{-1}.\mathbb{D})$. I.e. the domain of the sequence (S1, S2) of statements is $S1^{-1}.(S2^{-1}.\mathbb{D})$.

2.5.6 The if statement

When executing the statement

if B then S1 else S2 endif

the condition B is first evaluated. Depending upon whether its value is true or false, S1 or S2 respectively is executed. Formally, we define

(if B then S1 else S2 endif).d = S1.d, if B.d = true

= S2.d, if B.d = false

If B.d is undefined, then (if B then S1 else S2 endif).d is undefined.

Note that the above definition excludes side effects arising from evaluating the if condition B.

The domain of the if statement above consists of all data environments d such that (1) B.d is true and d is in the domain of S1 or (2) B.d is false and d is in the domain of S2. Expressed more formally, the domain is the set

$B^{-1}.\{true\}\cap S1^{-1}.\mathbb{D} \cup B^{-1}.\{false\}\cap S2^{-1}.\mathbb{D}$

or, writing Bt for the subset of 𝔻 on which B is true and Bf for the subset of 𝔻 on which B is false,

$Bt\cap S1^{-1}.\mathbb{D} \cup Bf\cap S2^{-1}.\mathbb{D}$

2.5.7 The while loop

When the while loop

while B do S endwhile

is executed, the condition B is first evaluated. If its value is true, the loop body S is executed and the entire while loop is executed again. If the value of B is false, the while loop is equivalent to the null statement; execution proceeds with the following statement, if any.

Formally, the execution of the above while loop, abbreviated W below, is defined as follows:

$$\begin{aligned} W.d &= (S, W).d, && \text{if } B.d=\text{true} \\ &= d, && \text{if } B.d=\text{false} \end{aligned}$$

Note that this definition excludes the possibility of side effects resulting from evaluating the while condition B.

While lemma 1: Applying the definition of the sequence of statements to the above definition of the while loop leads to:

$$\begin{aligned} W.d &= W.(S.d), && \text{if } B.d=\text{true} \\ &= d && \text{if } B.d=\text{false} \end{aligned}$$

While lemma 2: For all $n \in \mathbf{N}_0$ and all $d \in \mathbb{D}$

$$B.(S^n.d)=\text{false} \; \mathbf{and}_{j=0}^{n-1} \; B.(S^j.d)=\text{true} \Rightarrow W.d=S^n.d$$

Proof: induction on n.

Base case, n=0:

$$B.(S^n.d)=\text{false} \; \mathbf{and}_{j=0}^{n-1} \; B.(S^j.d)=\text{true} \Rightarrow W.d=S^n.d$$

reduces to

$$B.d=\text{false} \Rightarrow W.d=d$$

which is true by the definition of the while loop above.

Inductive step: We assume that the thesis of this lemma is valid for $n=k \geq 0$, i.e. that

$$B.(S^k.d)=\text{false} \; \mathbf{and}_{j=0}^{k-1} \; B.(S^j.d)=\text{true} \Rightarrow W.d=S^k.d$$

is true for all $d \in \mathbb{D}$, and must prove that it is true for n=k+1. Since the above expression is true for all d, it is true for the data environment S.d:

$$B.(S^k.(S.d))=\text{false} \; \mathbf{and}_{j=0}^{k-1} \; B.(S^j.(S.d))=\text{true} \Rightarrow W.(S.d)=S^k.(S.d)$$

=

$$B.(S^{k+1}.d)=\text{false} \; \mathbf{and}_{j=0}^{k-1} \; B.(S^{j+1}.d)=\text{true} \Rightarrow W.(S.d)=S^{k+1}.d$$

=

$$B.(S^{k+1}.d)=\text{false}\ \textbf{and}\ \bigwedge_{j=1}^{k} B.(S^{j}.d)=\text{true} \Rightarrow W.(S.d)=S^{k+1}.d \qquad \text{[WL2a]}$$

By while lemma 1,

$$B.d=\text{true} \Rightarrow W.d=W.(S.d) \qquad \text{[WL2b]}$$

Together [WL2a] and [WL2b] imply

$$B.(S^{k+1}.d)=\text{false}\ \textbf{and}\ B.d=\text{true}\ \textbf{and}\ \bigwedge_{j=1}^{k} B.(S^{j}.d)=\text{true} \Rightarrow W.d=W.(S.d)=S^{k+1}.d$$

⇒

$$B.(S^{k+1}.d)=\text{false}\ \textbf{and}\ \bigwedge_{j=0}^{k} B.(S^{j}.d)=\text{true} \Rightarrow W.d=S^{k+1}.d$$

I.e., the thesis of this lemma is valid for n=k+1. ■

Domain of the while statement: The domain of the while loop above can be identified iteratively. Denote the set of data environments for which the while loop terminates after exactly n executions of the loop body by Z_n. From the definition of the while loop it is evident that

$$Z_0=Bf$$

where Bf is the set of data environments upon which the while condition B is false (in other words, Bf is the preimage of the singleton set {false} under B). Z_1 is the set of data environments upon which B is true (so that the body of the loop will be executed, cf. the definition of the while loop) and upon which the execution of S will lead to a data environment in Z_0. That is, Z_1 is the intersection of Bt and the preimage of Z_0 under S:

$$Z_1 = Bt \cap S^{-1}.Z_0$$

where Bt is the set of data environments upon which the value of B is true (the preimage of {true} under B). Correspondingly,

$$Z_n = Bt \cap S^{-1}.Z_{n-1}$$

for all positive integers n. In closed form,

$$Z_n = S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt$$

or, in the form of a Boolean expression (condition),

$$Z_n.d = [B.(S^{n}.d)=\text{false}\ \textbf{and}\ \bigwedge_{j=0}^{n-1} B.(S^{j}.d)=\text{true}]$$

Cf. while lemma 2.

The domain of the while loop is the union of all Z_n:

$$\cup_{n=0}^{\infty} Z_n$$

=

$$\cup_{n=0}^{\infty} S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt$$

or, in the form of a Boolean expression (condition)

$$\textbf{or}_{n=0}^{\infty} B.(S^n.d)=\text{false } \textbf{and}_{j=0}^{n-1} B.(S^j.d)=\text{true}$$

2.5.8 The subprogram call without formal parameters

If a procedure (subprogram) P consists of the program segment S, then the effect of calling P is the same as executing S. Formally,

(call P).d = S.d

for all $d \in \mathbb{D}$.

The domain of call P is the same as the domain of S. Note that the domain of any program segment S is $S^{-1}.\mathbb{D}$, the preimage of $\mathbb{D}$ under S.

2.5.9 Basic vs. compound program statements

The above eight program statements fall naturally into two categories: basic statements (assignment, declare, release and null) and the compound statements (sequence, if, while and subprogram call). This distinction will be of some interest later.

2.5.10 Other loop structures

Other loop structures can be defined in terms of the while loop. For example,

repeat S until B endrepeat

is defined as the sequence of S followed by a corresponding while loop:

S, while not B do S endwhile

The loop with an internal exit

loop
S1
if B then exit
S2
endloop

is defined as

S1, while not B do S2, S1 endwhile

Still other loop structures, such as the for loop, can be defined in a similar manner, whereby implementational variations must be taken into account.

2.5.11 The subprogram call with formal parameters

Subprogram calls with formal parameter passing will be modelled by equivalent combinations of the program statements already defined above. Details of the mechanisms invoked by formal parameter passing vary somewhat from implementation to implementation in ways which can and do affect the correctness of a program. They must, therefore, be considered explicitly by the software developer responsible for the correctness of the program in which these mechanisms are used.

Most implemented schemes for passing formal parameters are either the classical call by value or call by name mechanisms or variants thereof.

A call by value causes a new, local variable to be declared whose initial value is the value of the actual parameter. This local variable is released at the end of the subprogram.

With the call by name, every reference to the formal parameter in the subprogram is a reference to the corresponding actual parameter in the calling environment. In effect, the name of the formal parameter is changed throughout the subprogram to the actual parameter (the variable name or the expression, not the value of the actual parameter) and the resulting subprogram executed. The original of the subprogram is effectively a template for generating the subprogram to be executed, not the subprogram itself. Naming conflicts are eliminated by changing names of variables in a suitable manner.

Consider the following call to the subprogram P. The expressions a1 and a2 are the actual parameters and f1 and f2, the formal parameters. The formal parameter f1 is called by value and f2 is called by name. The subprogram call is

call P(a1, a2)

and the subprogram P is defined to be

subprogram P(f1: **Z**; value, f2: name)
f2:=f1/5
end subprogram

The call with formal parameters (call P(a1, a2)) is defined to be equivalent to the call without parameters

call Pn

where the subprogram Pn is

declare (f1, **Z**, a1)
a2:=f1/5
release f1

Note that in reality there is no program variable f2; all original references to f2 are really references to the actual parameter a2 in the calling environment.

Many variants of these two mechanisms for passing parameters between the calling and the called environment will be found in implemented systems. They can be quite convenient in programming practice, but can affect the results of executing the program in which they are used in important ways. The software developer using them remains responsible for the correctness of the program and must, therefore, fully understand the mechanisms invoked.

2.5.12 Input/output

Many programming languages have various statements for performing input and output operations, e.g. read, write, get, put, print, seek, etc. These statements are nothing other than assignment statements and subprograms whose primary functional components are assignment statements in disguise.

E.g. the statement

```
print x; y
```

can be viewed as the sequence

```
screen(currentline):=str(x)&str(y)
currentline:=currentline+1
```

where currentline is an internal system variable and screen is an array whose last several elements are displayed on the video display screen by the hardware (possibly in combination with system software).

Fields in direct access files are probably most conveniently modelled as array variables. For example, the sequence of commands

```
seek#1, r; read#1, x
```

may be defined or modelled by the statements

```
pos(1):=r; x:=filerecord(1, pos(1))
```

where pos is an array of internal system variables and the array filerecord encompasses the values stored in all files.

3. Preconditions and postconditions

Mathematical theorems have the form: if a specified hypothesis is true, then the specified thesis is true. Theorems about the correctness of a program also have this form. Typically such correctness propositions have the more detailed form: if X is true before the program is executed, then Y will be true afterward. The hypothesis X is called the precondition of the program and the thesis Y, the postcondition. Most frequently, preconditions and postconditions refer to the values of program variables, but statements about the structure of the data environments before and after execution of the program in question are also formulated and proved. In addition, statements about which variables are and are not modified by the execution of a program are of interest.

Together, a precondition and a postcondition represent a specification of a program segment — a specification of its interface with other parts of the program, in particular, of the interface between the calling program and the called subprogram.

If the truth of a condition V before the execution of a program segment S ensures the truth of a condition P afterward, we say that V is a precondition of P with respect to S. This statement is still vague regarding whether the effect of executing S is defined or not and, more particularly, whether the truth of V before execution ensures that that effect is defined or not. The different possibilities lead to the definition of different types of preconditions.

3.1 Ordinary preconditions

A subset V of $\mathbb{D}$ is a *precondition* of a given *postcondition* P (also a subset of $\mathbb{D}$) with respect to the statement S if S.d is in P for every d in V and in the domain of S. One writes {V}S{P} for this relationship. Formally,

$$\{V\}S\{P\} = (\mathbf{A}\ d : d \in V \cap S^{-1}.\mathbb{D} : S.d \in P) \qquad \text{[formal definition of } \{V\}S\{P\}\text{]}$$

Note that if V and P are viewed as Boolean functions (conditions) instead of as sets (cf. section 2.4 above), this definition becomes

$$\{V\}S\{P\} = (\mathbf{A}\ d : d \in S^{-1}.\mathbb{D} \wedge V.d : P.(S.d))$$

A precondition includes (1) arbitrary data environments which the program segment maps to data environments in P and (2) arbitrary data environments which are outside of the domain of the program segments. A precondition does not include any data environments which the program segment maps to data environments outside of P.

Thus, the truth of (an ordinary precondition) V before execution of S ensures that the result of executing S — if any — will satisfy the postcondition P. Expressed somewhat differently, the prior truth of V guarantees that the execution of S will not yield a defined but incorrect result.

Lemma for an ordinary precondition: $\{V\}S\{P\} = (V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P)$

Proof:

$$\{V\}S\{P\}$$
$$= \qquad \text{[definition of } \{V\}S\{P\}\text{]}$$

$(\mathbf{A}\ d : d \in V \cap S^{-1}.\mathbb{D} : S.d \in P)$

= [follows from the definition of a preimage]

$(\mathbf{A}\ d : d \in V \cap S^{-1}.\mathbb{D} : d \in S^{-1}.P)$

= [definition of a subset]

$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P$ ■

3.2 Strict preconditions

When the prior truth of a condition V ensures that the execution of S yields both a defined and a correct result (i.e. a result satisfying the postcondition P), then we say that V is a *strict* precondition of P with respect to S. For this relationship one writes {V}S{P} strictly. This will be the case if V is both an ordinary precondition (see above) and a subset of the domain of S. Formally,

$\{V\}S\{P\}$ strictly = $(\{V\}S\{P\} \wedge V \subseteq S^{-1}.\mathbb{D})$ [formal definition of {V}S{P} strictly]

A strict precondition is, of course, also an ordinary precondition.

Lemma for a strict precondition: $\{V\}S\{P\}$ strictly = $(V \subseteq S^{-1}.P)$

Proof:

$\{V\}S\{P\}$ strictly

= [definition of {V}S{P} strictly]

$\{V\}S\{P\} \wedge V \subseteq S^{-1}.\mathbb{D}$

= [lemma for an ordinary precondition]

$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P \wedge V \subseteq S^{-1}.\mathbb{D}$

= $[V \subseteq S^{-1}.\mathbb{D} \Rightarrow V \cap S^{-1}.\mathbb{D} = V]$

$V \subseteq S^{-1}.P \wedge V \subseteq S^{-1}.\mathbb{D}$

= $[P \subseteq \mathbb{D},\ S^{-1}.P \subseteq S^{-1}.\mathbb{D}]$

$V \subseteq S^{-1}.P$ ■

3.3 Partial and total correctness

The literature on proving programs correct distinguishes between partial and total correctness. A program is said to be partially correct if its execution yields a correct result — when it yields a result at all, which is not guaranteed. I.e. a program is partially correct if it never yields an incorrect result.

A program is said to be totally correct when its execution is guaranteed to yield a defined result which is correct. I.e. a program is totally correct if its execution always yields a correct result.

An ordinary precondition as defined above corresponds to partial correctness; a strict precondition, to total correctness.

It is useful to distinguish between partial and total correctness — or, correspondingly, between ordinary and strict preconditions — because the approaches employed to prove the two are often

quite different and involve different arguments, especially regarding the termination of loops. Proofs are usually simplified, often considerably, by separating these two concerns.

3.4 Complete preconditions

The literature on proving programs correct often refers to weakest preconditions, but does not usually deal with the domains of the various program statements in the detail considered here. As a result, the concept of a weakest precondition in the strict mathematical sense is not particularly meaningful in the present context. The essence of the concept of a weakest precondition is that it encompasses *all* initial data environments (program states) which the program segment in question maps to data environments satisfying the postcondition. These observations motivate the following definition of a third type of precondition:

A subset V of $\mathbb{D}$ is a *complete precondition* of a given postcondition P with respect to the program segment S if V is an ordinary precondition of P with respect to S and the preimage of P under S is a subset of V. For this relationship one writes {V}S{P} completely. Formally,

$\{V\}S\{P\}$ completely = $\{V\}S\{P\} \wedge S^{-1}.P \subseteq V$ [formal definition of {V}S{P} completely]

Lemma for a complete precondition: $\{V\}S\{P\}$ completely = $(V \cap S^{-1}.\mathbb{D} = S^{-1}.P)$

Proof:

$\{V\}S\{P\}$ completely

= [definition of {V}S{P} completely]

$\{V\}S\{P\} \wedge S^{-1}.P \subseteq V$

= [lemma for an ordinary precondition]

$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P \wedge S^{-1}.P \subseteq V$

= [$P \subseteq \mathbb{D}$, $S^{-1}.P \subseteq S^{-1}.\mathbb{D}$]

$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P \wedge S^{-1}.P \subseteq V \wedge S^{-1}.P \subseteq S^{-1}.\mathbb{D}$

=

$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P \wedge S^{-1}.P \subseteq V \cap S^{-1}.\mathbb{D}$

=

$V \cap S^{-1}.\mathbb{D} = S^{-1}.P$ ■

Lemma for a strict and complete precondition: $\{V\}S\{P\}$ strictly and completely = $(V = S^{-1}.P)$

Proof:

$\{V\}S\{P\}$ strictly and $\{V\}S\{P\}$ completely

= [definition of a complete precondition]

$\{V\}S\{P\}$ strictly $\wedge$ $(S^{-1}.P \subseteq V)$

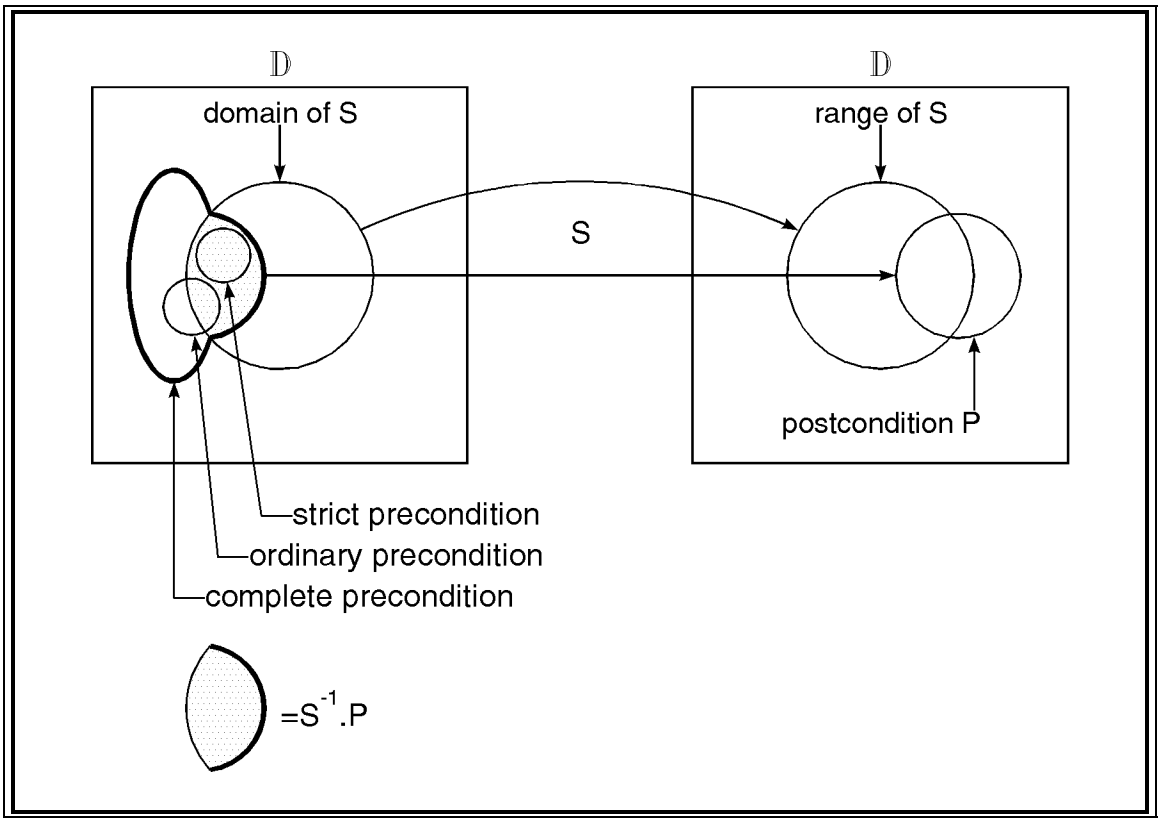
= [lemma for a strict precondition]

$(V \subseteq S^{-1}.P) \wedge (S^{-1}.P \subseteq V)$

=

$V = S^{-1}.P$ ■

The following diagram shows the relationships between the domain of a statement, an ordinary precondition, a strict precondition, a complete precondition and the preimage of the postcondition.



Relationships between the domain of a statement, an ordinary precondition, a strict precondition, a complete precondition and the preimage of the postcondition

3.5 Summary of the lemmata for preconditions

The four lemmata for an ordinary, a strict, a complete and a strict and complete precondition are summarized below. The equivalent expression for the proposition {V}S{P} depends upon whether the precondition is ordinary, strict, complete or strict and complete and is as follows:

	not strict	strict
not complete	$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P$	$V \subseteq S^{-1}.P$
complete	$V \cap S^{-1}.\mathbb{D} = S^{-1}.P$	$V = S^{-1}.P$

3.6 Referring in the postcondition to values of variables prior to execution

Sometimes it is desirable to refer in the postconditon to values of variables in the data environment before execution of the program segment in question. There are several ways of looking at such a situation mathematically, one of which is the following.

When we write

$$\{V\}\ S\ \{P(x, x', y)\}$$

where the postcondition P(x, x', y) is an expression in which x represents the value of the program variable x *after* execution of S, x' represents the value of the program variable x *before* execution of S and y represents the value of another program variable y *after* execution of S, we mean

$$(\mathbf{A}\ x' : x' \in M : \{V \text{ and } x=x'\}\ S\ \{P(x, x', y)\})$$

where M is a suitable set usually evident from the context. E.g. M is typically the set associated with the variable x in the initial data environment. In this view, x' is a parameter of the correctness proposition {V} S {P(x, x', y)} or {V and x=x'} S {P(x, x', y)}. We require that the correctness proposition be true for *all* values of this parameter. Such a parameter x' is sometimes called a *specification variable* (see Kaldewaij, Anne, *Programming: The Derivation of Algorithms*, Prentice-Hall International, 1990, section 2.0).

Note that x' is *not* a program variable.

By formulating a postcondition in this manner, relationships between values of variables before and after execution of S can be expressed in the postcondition.

This technique can often be used, for example, to prove that the execution of the body of a loop increases or decreases the value of some variable or expression by at least a certain amount, e.g.

$$\{I \text{ and } B \text{ and } i=i'\}\ S\ \{I \text{ and } i \leq i'-1\}$$

which can contribute to proving that the loop terminates. See also section 4.7.3 below.

4. Proof rules

A number of relationships between preconditions and postconditions generally apply and are of value in practical applications. Many such relationships, usually called *proof rules*, or more simply *rules*, are formulated and proved below. Different rules are formulated for the several proof related tasks and for the several types of program statements.

Generally, the rules are used as lemmata to decompose a larger proof or design task into one or more smaller tasks. This process is repeated iteratively until only rather simple, basic tasks remain to be solved, usually by manipulation of Boolean algebraic expressions.

4.1 The several proof related tasks

Each proof related task relates to a correctness proposition. The goal is either to determine one of the elements of the correctness proposition or to verify (i.e. prove) that the correctness proposition is true:

1. {V?} S {P} find a precondition for the given postcondition and program segment
2. {V} S? {P} design a program segment for the given precondition and postcondition
3. {V} S {P?} find a postcondition for a given precondition and program segment
4. {V} S {P} ? prove that {V} S {P} is true

Tasks 2 (program design) and 4 (verifying correctness) are the typical, classical tasks arising in any engineering discipline. In the course of proving the correctness of a program segment (task 4), the need to derive a precondition of a component part of the program segment in question often arises, i.e. task 1. Task 3 does not arise in this process; we will not, therefore, consider rules especially suited for task 3 (finding a postcondition for a given precondition and program segment) — although such rules do exist. Furthermore, task 3 seldom arises in practice because engineering is a goal-directed process. That goal is expressed in the postcondition, which must, therefore, be specified before meaningful design or verification work can start.

4.2 Rule P1: strengthening a precondition and weakening a postcondition

If

$V \Rightarrow V1$ and
{V1} S {P1} and
$P1 \Rightarrow P$

then

{V} S {P}

Proof:

$(V \Rightarrow V1) \wedge (\{V1\}\ S\ \{P1\}) \wedge (P1 \Rightarrow P)$
= [Lemma for an ordinary precondition]
$(V \subseteq V1) \wedge (V1 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P1) \wedge (P1 \subseteq P)$
$\Rightarrow$

$(V \cap S^{-1}.\mathbb{D} \subseteq V1 \cap S^{-1}.\mathbb{D}) \wedge (V1 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P1) \wedge (S^{-1}.P1 \subseteq S^{-1}.P)$

$\Rightarrow$

$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P$

= [Lemma for an ordinary precondition]

{V} S {P} ■

When working backward (upward) through a program, one may strengthen conditions. When working forward (downward) through a program, one may weaken conditions.

4.3 "Divide and conquer" rules

The following rules permit the software developer to "divide and conquer" lengthy preconditions and postconditions when proving a program correct. The rules below are formulated for ordinary preconditions, but strict and complete versions can also be formulated and proved.

By applying the "divide and conquer" rules one decomposes a proof task into two or more proof tasks involving shorter preconditions and/or postconditions. While this does not reduce the total amount of work involved, it can contribute significantly to a better organization of the proof, which is typically clearer and easier to understand and follow. The individual steps in the algebraic manipulations become shorter and simpler. Even very long and complex expressions yield to this strategy.

4.3.1 Rule DC1

If

{V1} S {P1} and
{V2} S {P2}

then

{V1 and V2} S {P1 and P2}

Proof:

{V1} S {P1} and {V2} S {P2}

= [Lemma for an ordinary precondition]

$(V1 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P1) \wedge (V2 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P2)$

$\Rightarrow$

$V1 \cap V2 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P1 \cap S^{-1}.P2$

=

$V1 \cap V2 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.(P1 \cap P2)$

= [Lemma for an ordinary precondition]

{V1 and V2} S {P1 and P2} ■

4.3.2 Rule DC2

If

{V1} S {P1} and
{V2} S {P2}

then

{V1 or V2} S {P1 or P2}

Proof:

{V1} S {P1} and {V2} S {P2}
= [Lemma for an ordinary precondition]
$(V1 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P1) \wedge (V2 \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P2)$
$\Rightarrow$
$(V1 \cap S^{-1}.\mathbb{D} \cup V2 \cap S^{-1}.\mathbb{D}) \subseteq (S^{-1}.P1 \cup S^{-1}.P2)$
=
$(V1 \cup V2) \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P1 \cup S^{-1}.P2$
=
$(V1 \cup V2) \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.(P1 \cup P2)$
= [Lemma for an ordinary precondition]
{V1 or V2} S {P1 or P2} ■

4.3.3 Rule DC3

If

{V} S {P1} and
{V} S {P2}

then

{V} S {P1 and P2}

Proof: This rule (DC3) is rule DC1 with V1=V2=V. ■

The reverse of this rule also applies (by rule P1). Therefore, the statement of this rule can be strengthened to "if and only if". As a result, a postcondition P can be separated in any way into P1 and P2 without introducing the possibility that the proof cannot be completed. For this (as well as other) reasons, DC3 is probably the most useful of the "divide and conquer" rules.

4.3.4 Rule DC4

If

{V} S {P1} and
{V} S {P2}

then

{V} S {P1 or P2}

Proof: This rule (DC4) is rule DC2 with V1=V2=V. ■

4.4 Rules for the assignment statement

In this section two rules for the assignment statement are formulated and proved. One enables finding a precondition of a given postcondition with respect to a given assignment statement and the other is intended for verifying a given correctness proposition about an assignment statement.

The assignment statement x:=E, where x is a variable name and E is an expression, is sometimes abbreviated A below.

Before stating and proving the rules for an assignment statement, is is useful to note the following consequence of the definition of an assignment statement as a function on $\mathbb{D}$ to $\mathbb{D}$.

Lemma for the assignment statement: Let d1=(x:=E).d0 (i.e. d1=A.d0). Then

x.d1=E.d0 and
y.d1=y.d0, for all variable names y other than x.

Proof: This lemma follows directly from the definition of an assignment statement as a function on $\mathbb{D}$, see section 2.5.1 above. ■

4.4.1 Rule A1

Let the condition P and the assignment statement A (x:=E) be given. If one forms the condition V by replacing in P every *reference to the value* of the variable x by the expression E (in parentheses), then V is a complete precondition of P with respect to the assignment statement x:=E.

One writes P_E^x for the expression resulting from replacing x in P by E as described above. We write rule A1 accordingly:

$\{P_E^x\}$ x:=E {P} completely [rule A1]

Proof: Consider any data environment d0 in the domain of A, i.e. $d0 \in A^{-1}.\mathbb{D}$. The postcondition P is an expression (function) in which the variable x as well as other variables y may appear: P(x, y). The purported precondition P_E^x is P(E, y). The value of P after execution of the assignment statement is P.d1, where d1=A.d0. But

P.d1
=
P(x, y).d1
=

$P(x.d1, y.d1)$

$=$ [lemma for the assignment statement]

$P(E.d0, y.d0)$

$=$

$P(E, y).d0$

$=$

$P_E^x.d0$

Thus, $P_E^x.d0 = P.d1 = P.(A.d0)$. From this it follows that $d0 \in P_E^x = A.d0 \in P$. But $A.d0 \in P$ is equivalent to $d0 \in A^{-1}.P$. That is, $(\mathbf{A}\ d0 : d0 \in A^{-1}.\mathbb{D} : d0 \in P_E^x = d0 \in A^{-1}.P)$. But

$(\mathbf{A}\ d0 : d0 \in A^{-1}.\mathbb{D} : d0 \in P_E^x = d0 \in A^{-1}.P)$

$=$ $[(\mathbf{A}\ x : x \in X : x \in Y = x \in Z) = (Y \cap X = Z \cap X)]$

$P_E^x \cap A^{-1}.\mathbb{D} = A^{-1}.P \cap A^{-1}.\mathbb{D}$

$=$ $[P \subseteq \mathbb{D},\ A^{-1}.P \subseteq A^{-1}.\mathbb{D}]$

$P_E^x \cap A^{-1}.\mathbb{D} = A^{-1}.P$

$=$ [lemma for a complete precondition]

$\{P_E^x\}$ x:=E $\{P\}$ completely ■

Note that only references to the *value* of the variable x are to be replaced by the expression E (in parentheses). Occurences of the name x that do *not* refer to the value of x should not modified as described above. In some cases, a meaningless expression would result. A common example is a postcondition containing terms of the form Set.“x”, i.e. references to the set associated with the variable x, not to the value of x.

Note that the expression E should be placed in parentheses before inserting it into the postcondition. Sometimes the parentheses are superfluous, but they are never wrong. It is sometimes wrong not to include them.

Rule A1 is used to derive a precondition for a given postcondition and a given assignment statement.

Example 1: {V?} sum:=sum+z {x+y+z-sum=0} completely

V

$=$ [rule A1]

$(x+y+z-sum=0)^{sum}_{sum+z}$

$=$

$x+y+z-(sum+z)=0$

=

x+y+z-sum-z=0

=

x+y-sum=0 ■

Caution is called for when applying this rule to an assignment to an array variable. Every reference to a variable in the array in question must be examined to determine whether it must be replaced by the expression to the right of the assignment symbol (:=) or not. This may require a case analysis distinguishing between equality and inequality of the index expressions involved.

Example 2: {V?} x(i):=1 {x(i)=x(j)} completely

If one unthinkingly replaces x(i) but not x(j) in the postcondition by 1, one obtains 1=x(j) as the supposed precondition. But if i=j, then one should replace both x(i) and x(j) by 1, because both refer to the same array variable, which receives the new value 1. Thus, if i=j, the precondition is the logical constant true, and if i≠j, the precondition is 1=x(j). Combining, we obtain the expression [(i=j ∧ true) ∨ (i≠j ∧ 1=x(j))], which can be simplified to [i=j ∨ 1=x(j))] as the correct complete precondition.

More formally, we can find the complete precondition in a more systematic and generally applicable manner as follows:

V

= [rule A1]

$[x(i)=x(j)]_{1}^{x(i)}$

=

$[(i=j \vee i\neq j) \wedge x(i)=x(j)]_{1}^{x(i)}$

=

$[(i=j \wedge x(i)=x(j)) \vee (i\neq j \wedge x(i)=x(j))]_{1}^{x(i)}$

=

$[(i=j \wedge x(i)=x(i)) \vee (i\neq j \wedge x(i)=x(j))]_{1}^{x(i)}$

=

$[i=j \vee (i\neq j \wedge x(i)=x(j))]_{1}^{x(i)}$

=

$[i=j]_{1}^{x(i)} \vee [i\neq j]_{1}^{x(i)} \wedge [x(i)=x(j)]_{1}^{x(i)}$

=

$i=j \vee i\neq j \wedge [x(i)=x(j)]_{1}^{x(i)}$

=

$i=j \vee i\neq j \wedge 1=x(j)$

=

$i=j \vee 1=x(j)$ ■

Each reference in a postcondition to a variable of the array in question either (1) always refers to the array variable to which assignment is being made, (2) never refers to the array variable to which assignment is being made or (3) may or may not refer to the array variable to which assignment is being made, depending upon the possible value(s) of the index expression in question. References of the third type must be eliminated by suitable algebraic manipulation, e.g. by a case distinction as in the above example, *before* replacing references to the variable in question by the expression on the right hand side of the assignment symbol (:=).

Another method for finding a precondition does not use rule A1, but instead applies the lemma for the assignment statement directly. Employing this method, one distinguishes notationally between evaluation in the data environments before and after execution of the assignment statement, e.g. writing x' for x.d and x" for x.(A.d) etc., where A is the assignment statement in question. This method can be applied to the example above as follows:

Example 3: {V?} x(i):=1 {x(i)=x(j)} completely. We write the postcondition as x(i")"=x(j")". By the lemma for the assignment statement, i"=i', j"=j', x(i')"=1 and x(k)"= x(k)' for all $k \neq i'$. Using these equalities we manipulate the postcondition in order to eliminate all references to values of variables in the data environment after execution of the assignment statement, i.e. to eliminate all doubly primed terms, leaving only singly primed terms:

$x(i")"=x(j")"$ [postcondition]

=

$x(i')"=x(j')"$

=

$1=x(j')"$

=

$(i'=j' \vee i' \neq j') \wedge 1=x(j')"$

=

$(i'=j' \wedge 1=x(j')") \vee (i' \neq j' \wedge 1=x(j')")$

=

$(i'=j' \wedge 1=x(i')") \vee (i' \neq j' \wedge 1=x(j')")$

=

$(i'=j' \wedge 1=1) \vee (i' \neq j' \wedge 1=x(j')')$

=

$i'=j' \vee (i' \neq j' \wedge 1=x(j')')$

=

$i'=j' \vee 1=x(j')'$ [precondition]

Because this expression contains only singly primed terms, i.e. refers only to values of variables in the data environment before execution of the assignment statement, it is a suitable precondition. The primes can be dropped to obtain $i=j \vee 1=x(j)$ as the precondition being sought. ■

The method of example 3 makes all references to the values of variables in the two data environments explicit and thereby contributes to clarity, especially in particularly complicated

expressions. On the other hand, it usually leads to lengthier and more tedious derivations. Generally, one should use the method of example 2 and revert to the method of example 3 only when confusion arises.

While the method illustrated in example 2 above is almost always adequate in practice, it does rely on the assumption that the index expression contains no reference to the array variable being replaced, i.e. that the value of the index expression is not changed by the execution of the assignment statement in question. When this assumption does not apply, the method shown in example 2 cannot be used directly. Although the method shown in example 3 above can be used in such cases, a more easily mechanizable method is desirable, especially for use in automated verification systems. A generally applicable formula for replacing a reference to an array variable is:

$$[x(E1)]^{x(E2)}_{E3} = (\text{if } [E1]^{x(E2)}_{E3}=E2 \text{ then } E3 \text{ else } x([E1]^{x(E2)}_{E3}) \text{ endif})$$

where x is the name of an array and E1, E2 and E3 are any expressions. The proof of this equation is left as an exercise for the reader. Hints: Use the method of example 3 above and note that the essence of the replacement operation is given by the lemma (or axiom) that the value of x^{y}_{E} before execution of the assignment statement y:=E is the same as the value of x after execution of this assignment statement for all variables x and y (including array variables), for all expressions E and for all data environments in the domain of this assignment statement. I.e., prove that the value of the (if ... endif) expression above before execution of the assignment statement x(E2):=E3 is equal to the value of x(E1) after execution of this assignment statement.

When applying rule A1 to a multiple assignment statement or to an exchange statement, the replacements must take place simultaneously, not one after the other.

Example 4: {V?} x:=:y {x+2*y=z} completely

V

= [rule A1]

$[x+2*y=z]^{x,\ y}_{y,\ x}$

=

y+2*x=z ■

The following table summarizes the above descriptions of applying rule A1 to the various situations that can arise:

variable in the postcondition	variable to the left of :=	additional condition	example of expression	final result
simple	simple	different variables	$[x]^{y}_{E}$	x
simple	simple	same variable	$[x]^{x}_{E}$	(E)
simple	array	—	$[x]^{y(E1)}_{E2}$	x
array	simple	—	$[x(E1)]^{y}_{E2}$	$x([E1]^{y}_{E2})$
array	array	different arrays	$[x(E1)]^{y(E2)}_{E3}$	$x([E1]^{y(E2)}_{E3})$
array	array	same array	$[x(E1)]^{x(E2)}_{E3}$	if $[E1]^{x(E2)}_{E3}$=E2 then (E3) else $x([E1]^{x(E2)}_{E3})$

Results of applying rule A1 to the various types of variables in the postcondition and to the left of the assignment symbol (:=)

4.4.2 Rule A2

If

$$V \Rightarrow P^{x}_{E}$$

then

$$\{V\}\ x:=E\ \{P\}$$

Proof:

$$\{V\}\ x:=E\ \{P\}$$

$\Leftarrow$ [rule P1]

$$V \Rightarrow P^{x}_{E} \wedge \{P^{x}_{E}\}\ x:=E\ \{P\}$$

$=$ [rule A1]

$$V \Rightarrow P^{x}_{E}\ \blacksquare$$

Rule A2, which is simply a combination of rules P1 and A1, provides a way of verifying a correctness proposition about an assignment statement. To verify that {V} x:=E {P}, one verifies

that $V \Rightarrow P_E^x$. Thus the task of verifying the correctness proposition {V} x:=E {P} is reduced to the task of verifying the universal truth of a Boolean algebraic expression.

Example: {j≥0} k:=j+1 {k≥0} ?

$\{j \geq 0\}\ k:=j+1\ \{k \geq 0\}$

$\Leftarrow$ [rule A2]

$j \geq 0 \Rightarrow [k \geq 0]_{j+1}^{k}$

=

$j \geq 0 \Rightarrow j+1 \geq 0$

=

true ■

4.5 Rules for sequences of statements

4.5.1 Rule S1 for the sequence of statements

If

{V} S1 {P1} and
{P1} S2 {P}

then

{V} (S1, S2) {P}

Proof:

$\{V\}\ (S1, S2)\ \{P\}$

= [lemma for an ordinary precondition]

$V \cap (S1, S2)^{-1}.\mathbb{ID} \subseteq (S1, S2)^{-1}.P$

=

$V \cap S1^{-1}.(S2^{-1}.\mathbb{ID}) \subseteq S1^{-1}.(S2^{-1}.P)$ (1)

But

$\{V\}\ S1\ \{P1\} \wedge \{P1\}\ S2\ \{P\}$

= [lemma for an ordinary precondition]

$[V \cap S1^{-1}.\mathbb{ID} \subseteq S1^{-1}.P1] \wedge [P1 \cap S2^{-1}.\mathbb{ID} \subseteq S2^{-1}.P]$

$\Rightarrow$

$[V \cap S1^{-1}.\mathbb{ID} \subseteq S1^{-1}.P1] \wedge [S1^{-1}.(P1 \cap S2^{-1}.\mathbb{ID}) \subseteq S1^{-1}.(S2^{-1}.P)]$

=

$[V \cap S1^{-1}.\mathbb{ID} \subseteq S1^{-1}.P1] \wedge [(S1^{-1}.P1) \cap S1^{-1}.(S2^{-1}.\mathbb{ID}) \subseteq S1^{-1}.(S2^{-1}.P)]$

$\Rightarrow$

$V \cap (S1^{-1}.\mathbb{ID}) \cap S1^{-1}.(S2^{-1}.\mathbb{ID}) \subseteq S1^{-1}.(S2^{-1}.P)$

= [$S2^{-1}.\mathbb{D} \subseteq \mathbb{D}$, $S1^{-1}.(S2^{-1}.\mathbb{D}) \subseteq S1^{-1}.\mathbb{D}$]

$V \cap S1^{-1}.(S2^{-1}.\mathbb{D}) \subseteq S1^{-1}.(S2^{-1}.P)$ ■ [= (1) above]

Rule S1 may be extended in the obvious way for an arbitrarily long sequence of program statements.

Strict and complete versions of rule S1 are also valid. The proof of the complete version is the above proof of the ordinary version with the symbol $\subseteq$ replaced by =. The strict version can be proved as follows:

Rule S1, strict version: If {V} S1 {P1} strictly and {P1} S2 {P} strictly, then {V} (S1, S2) {P} strictly.

Proof:

{V} S1 {P1} strictly and {P1} S2 {P} strictly

= [lemma for a strict precondition]

$V \subseteq S1^{-1}.P1 \wedge P1 \subseteq S2^{-1}.P$

$\Rightarrow$

$V \subseteq S1^{-1}.P1 \wedge S1^{-1}.P1 \subseteq S1^{-1}.(S2^{-1}.P)$

$\Rightarrow$

$V \subseteq (S1, S2)^{-1}.P$

= [lemma for a strict precondition]

{V} (S1, S2) {P} strictly ■

Rule S1 has an important implication for practice: the postcondition may be transformed, working statement by statement from the end of a sequence of statements to the beginning, into a precondition with respect to the entire sequence. If in every step a complete precondition is derived, then the final precondition is a complete precondition with respect to the entire sequence of statements.

When applying rule S1 in order to verify a correctness proposition about a sequence of program statements, intermediate conditions (P1 above) are required and, if not given, must be derived. Thus, the application of rule S1 to the task of verifying a correctness proposition gives rise to the need for rules to derive preconditions for a given postcondition and a given statement.

4.5.2 Rule S2 for the sequence of assignment statements

Frequently a correctness proposition about a sequence of assignment statements must be proved. For such situations it is convenient to combine rules S1, A1 and A2 into rule S2:

If

$$V \Rightarrow [[\ldots [P^{xn}_{En}] \ldots]^{x2}_{E2}]^{x1}_{E1}$$

then

{V}
x1:=E1
x2:=E2
...
xn:=En
{P}

Note the sequence in which the several variable names are replaced by the corresponding expressions: from the bottom to the top of the sequence of assignment statements. In effect, rule A1 is applied to the *last* assignment statement *first*, then to the second last assignment statement, etc., and finally rule A2 is applied to the *first* assignment statement.

4.6 Rules for the if statement

Two rules for the if statement are formulated and proved below. One facilitates verifying a correctness proposition, the other gives a rule for deriving a precondition for a given postcondition and a given if statement.

We begin by considering the preimage of P (a subset of $\mathbb{D}$) under the if statement

if B then S1 else S2 endif

abbreviated S below. Furthermore, let Bt be the subset of $\mathbb{D}$ upon which B is true (the preimage of true under B) and Bf be the subset of $\mathbb{D}$ upon which B is false (the preimage of false under B). The preimage of P under S consists of those data environments which S maps into P. These are the data environments (1) which are in Bt and which S1 maps into P, i.e. which are in Bt and in the preimage of P under S1 or (2) which are in Bf and which S2 maps into P, i.e. which are in Bf and in the preimage of P under S2. (Cf. the domain of the if statement in section 2.5.6 above.) More formally,

$$S^{-1}.P = (Bt \cap S1^{-1}.P \cup Bf \cap S2^{-1}.P)$$

4.6.1 Rule IF1 for verifying a correctness proposition about an if statement

If

{V and B} S1 {P} and
{V and not B} S2 {P}

then

{V} if B then S1 else S2 endif {P}

Proof:

{V and B} S1 {P} and {V and not B} S2 {P}

$=$ [lemma for an ordinary precondition]

$(V \cap Bt \cap S1^{-1}.\mathbb{D} \subseteq S1^{-1}.P) \wedge (V \cap Bf \cap S2^{-1}.\mathbb{D} \subseteq S2^{-1}.P)$

$=$

$(V \cap Bt \cap S1^{-1}.\mathbb{D} \subseteq Bt \cap S1^{-1}.P) \wedge (V \cap Bf \cap S2^{-1}.\mathbb{D} \subseteq Bf \cap S2^{-1}.P)$

$\Rightarrow$

$(V \cap Bt \cap S1^{-1}.\mathbb{D} \cup V \cap Bf \cap S2^{-1}.\mathbb{D}) \subseteq (Bt \cap S1^{-1}.P \cup Bf \cap S2^{-1}.P)$

=

$[V \cap (Bt \cap S1^{-1}.\mathbb{D} \cup Bf \cap S2^{-1}.\mathbb{D})] \subseteq (Bt \cap S1^{-1}.P \cup Bf \cap S2^{-1}.P)$

=

$V \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.P$

= [lemma for an ordinary precondition]

{V} S {P} ■

Rule IF1 decomposes a correctness proposition to be verified into two correctness propositions about smaller program segments. Each must be verified by applying the rules appropriate for the structures of the then and the else parts of the original if statement.

The complete version of rule IF1 holds. The strict version does not hold, because V need not imply that B is defined (true or false).

4.6.2 Rule IF2 for deriving a precondition with respect to an if statement

If

{V1} S1 {P} and
{V2} S2 {P}

then

{V1 and B or V2 and not B} if B then S1 else S2 endif {P}

Proof:

{V1 and B or V2 and not B} if B then S1 else S2 endif {P}

$\Leftarrow$ [rule IF1]

{[V1 and B or V2 and not B] and B} S1 {P}
and {[V1 and B or V2 and not B] and not B} S2 {P}

=

{V1 and B} S1 {P} and {V2 and not B} S2 {P}

$\Leftarrow$ [rule P1]

{V1} S1 {P} and {V2} S2 {P} ■

The complete version of rule IF2 holds. Provided that the condition (function) B is extended to $\mathbb{D}$ in such a way that not B is true if and only if B is false, the strict version of rule IF2 also holds.

4.7 Rules for the while loop

In this section two rules for the while loop are formulated and proved. The first deals with a while loop alone. Associated with almost every loop is an initialization, so a second rule for a while loop together with its initialization will be formulated and proved. From a theoretical standpoint the second rule contains nothing new, but from a practical standpoint it is much more important. The second rule provides more insight into the practical application of these concepts, especially in connection with designing a loop.

In proving the rules for the while loop, the following preliminary facts and lemmata will be needed. The while statement while B do S endwhile is often abbreviated W below.

Preimage of a set under a while statement: In section 2.5.7 above the domain of a while statement was derived. In the same way one can derive the preimage of a subset P of $\mathbb{D}$ under a while statement. The result is:

$$W^{-1}.P = \cup_{n=0}^{\infty} S^{-n}.(P \cap Bf) \cap_{j=0}^{n-1} S^{-j}.Bt$$

Lemma 4.7 A: Given are the while statement while B do S endwhile and a subset I of $\mathbb{D}$. The subsets Bf and Bt of $\mathbb{D}$ are defined to be the sets upon which the condition B is false and true respectively. I.e. $Bf = B^{-1}.\{false\}$ and $Bt = B^{-1}.\{true\}$. Then

$$I \cap Bt \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.I$$

$\Rightarrow$

$$I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt \subseteq S^{-1}.I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt$$

for every non-negative integer n.

Proof:

$$I \cap Bt \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.I$$

$\Rightarrow$ [$\cap$ same terms to both sides]

$$I \cap Bt \cap S^{-1}.\mathbb{D} \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt \subseteq S^{-1}.I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt$$

$\Rightarrow$ [$n \geq 0$, Bt is the 0-th term in the $\cap$-series, $S^{-n}.Bf \subseteq \mathbb{D}$, $S^{-(n+1)}.Bf = S^{-1}.(S^{-n}.Bf) \subseteq S^{-1}.\mathbb{D}$]

$$I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt \subseteq S^{-1}.I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt \blacksquare$$

The essence of the proof of the first rule below for the while loop lies in the following lemma. The condition I (a subset of $\mathbb{D}$) is called the *loop invariant*. This important condition will appear again later.

Lemma 4.7 B: Given are the while statement while B do S endwhile and subsets I, Bf and Bt of $\mathbb{D}$ as in lemma 4.7 A above. Then

$$I \cap Bt \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.I$$

$\Rightarrow$

$$I \cap S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt \subseteq S^{-n}.(I \cap Bf) \cap_{j=0}^{n-1} S^{-j}.Bt$$

for every non-negative integer n.

Proof: This lemma will be proved by induction on n.

Proof of the base case: In the base case (n=0) the thesis of this lemma becomes

$$\ldots \Rightarrow I \cap Bf \subseteq I \cap Bf$$

which is clearly true.

Inductive step: In the inductive step, the truth of the lemma for n will be assumed, i.e.

$$I \cap Bt \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.I$$

$\Rightarrow$ [inductive assumption]

$$I \cap S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt \subseteq S^{-n}.(I \cap Bf) \cap_{j=0}^{n-1} S^{-j}.Bt$$

and the truth of the lemma for n+1 will be proved, i.e.

$$I \cap Bt \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.I$$

$\Rightarrow$ [to be proved]

$$I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt \subseteq S^{-(n+1)}.(I \cap Bf) \cap_{j=0}^{n} S^{-j}.Bt$$

Proof of the inductive step:

$$I \cap Bt \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.I$$

$\Rightarrow$ [inductive assumption]

$$I \cap S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt \subseteq S^{-n}.(I \cap Bf) \cap_{j=0}^{n-1} S^{-j}.Bt$$

$\Rightarrow$ [S^{-1} each side]

$$S^{-1}.I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n-1} S^{-(j+1)}.Bt \subseteq S^{-(n+1)}.(I \cap Bf) \cap_{j=0}^{n-1} S^{-(j+1)}.Bt$$

$\Rightarrow$ [redefine running variables]

$$S^{-1}.I \cap S^{-(n+1)}.Bf \cap_{j=1}^{n} S^{-j}.Bt \subseteq S^{-(n+1)}.(I \cap Bf) \cap_{j=1}^{n} S^{-j}.Bt$$

$\Rightarrow$ [$\cap Bt$ to each side]

$$S^{-1}.I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt \subseteq S^{-(n+1)}.(I \cap Bf) \cap_{j=0}^{n} S^{-j}.Bt$$

$\Rightarrow$ [lemma 4.7 A, $\subseteq$ transitive]

$$I \cap S^{-(n+1)}.Bf \cap_{j=0}^{n} S^{-j}.Bt \subseteq S^{-(n+1)}.(I \cap Bf) \cap_{j=0}^{n} S^{-j}.Bt \blacksquare$$

4.7.1 Rule W1 for the while loop without initialization

If

$$\{I \text{ and } B\}\ S\ \{I\}$$

then

{I} while B do S endwhile {I and not B}

Note: In the proof below, the entire while statement (while B do S endwhile) is abbreviated W.

Proof:

$\{I \text{ and } B\}\ S\ \{I\}$

$=$ [lemma for an ordinary precondition]

$I \cap Bt \cap S^{-1}.\mathbb{D} \subseteq S^{-1}.I$

$\Rightarrow$ [lemma 4.7 B]

$\text{and}_{n=0}^{\infty}\ [I \cap S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt \subseteq S^{-n}.(I \cap Bf) \cap_{j=0}^{n-1} S^{-j}.Bt]$

$\Rightarrow$

$\cup_{n=0}^{\infty}\ I \cap S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt \subseteq \cup_{n=0}^{\infty}\ S^{-n}.(I \cap Bf) \cap_{j=0}^{n-1} S^{-j}.Bt$

$=$

$I \cap (\cup_{n=0}^{\infty}\ S^{-n}.Bf \cap_{j=0}^{n-1} S^{-j}.Bt) \subseteq \cup_{n=0}^{\infty}\ S^{-n}.(I \cap Bf) \cap_{j=0}^{n-1} S^{-j}.Bt$

$=$

$I \cap W^{-1}.\mathbb{D} \subseteq W^{-1}.(I \cap Bf)$

$=$ [lemma for an ordinary precondition]

{I} while B do S endwhile {I and not B} ■

4.7.2 Rule W2 for the while loop with initialization

If

$\{V\}$ init $\{I\}$ and [start]
$\{I \text{ and } B\}\ S\ \{I\}$ and [during execution]
I and not $B \Rightarrow P$ [end]

then

{V} init; while B do S endwhile {P}

Proof:

{V} init; while B do S endwhile {P}

$\Leftarrow$ [rule S1]

[{V} init {I}] and [{I} while B do S endwhile {P}]

$\Leftarrow$ [rule P1]

[{V} init {I}] and [{I} while B do S endwhile {I and not B}] and [I and not B $\Rightarrow$ P]

$\Leftarrow$ [rule W1]

[{V} init {I}] and [{I and B} S {I}] and [I and not B $\Rightarrow$ P] ■

The condition V above is in general an ordinary precondition. It does not guarantee that the while loop will execute to yield a defined result, in particular, that the recursive definition of the while loop will terminate (equivalently, that the execution of the loop will terminate).

The condition I is true before execution of the loop begins, before and after each execution of the loop body S and after the execution of the loop has terminated. The condition I is therefore called the *loop invariant*. The loop invariant is the most important concept in designing a loop, in understanding a loop and in proving a loop to be correct. It expresses the main design decision in the construction of a loop.

In order to prove a correctness proposition about a loop, one must

1. identify a suitable loop invariant I if not already given by the software designer,
2. prove that the initialization establishes the initial truth of the loop invariant I,
3. prove that the execution of the loop body S preserves the truth of the loop invariant I,
4. prove that the truth of the loop invariant I and the falsehood of the loop condition B together imply the truth of the postcondition and
5. prove that the loop will be executed with a defined result, in particular, that the loop terminates.

Steps 1 through 4 ensure *partial* correctness of the loop. Step 5 in addition ensures *total* correctness.

For *every* loop a loop invariant!

In order to design a loop, the software developer must

1. decide upon a suitable loop invariant I,
2. design the initialization so that it establishes the initial truth of the loop invariant I,
   {V} init? {I}
3. design the while condition B so that
   I and not B? $\Rightarrow$ P
   and
4. design the body S of the loop so that it preserves the truth of the loop invariant I
   {I and B} S? {I}
   while ensuring progress toward (not B) or P, in order to ensure that the loop will terminate.

For *every* loop a loop invariant!

The first step in designing a loop must be to decide upon a loop invariant, because every other step requires it. Step 2 can be performed independently of steps 3 and 4. The result of step 3 (the while condition B) is a prerequisite for step 4.

Note that the loop invariant I is true (usually in a trivial way) before execution of the loop begins. When execution terminates, it is also true (I and not B is true). In other words, the initial and final situations are special cases of the loop invariant I. Viewed the other way around, the loop invariant is a generalization of the initial and final states. Roughly speaking, the loop invariant I can be thought of as a generalization of the precondition V and the postcondition P (even though

this statement is not mathematically correct). This observation gives us a useful guideline for designing a loop invariant. A number of rules of thumb have been formulated for deciding upon a loop invariant; all are variations of the idea of generalizing the initial and final conditions. Section 8.4 below contains additional guidelines on formulating suitable loop invariants.

The requirement [I and not B $\Rightarrow$ P] can be written in many equivalent ways, some of which suggest useful approaches for deriving B. The most useful forms are probably:

1. I and not B $\Rightarrow$ P
2. I and not P $\Rightarrow$ B
3. I $\Rightarrow$ (not B $\Rightarrow$ P)
4. I $\Rightarrow$ (not P $\Rightarrow$ B)

These forms suggest the following questions and strategies for determining a suitable while condition B:

1. What condition in addition to I ensures the truth of P? (The answer is the negation of B.)

2a. What must be true if I is true, but P is not? Use this condition as B.

2b. Form the expression for [I and not P] and simplify and weaken it to obtain B.

3. Simplify and strengthen P, assuming the truth of I, to obtain an expression for not B. Negate the result to obtain B.

4. Simplify and weaken the negation of P, assuming the truth of I, to obtain B.

In each case, B must be in a form which is syntactically permitted as a while condition. Note that (not P) is always semantically a correct candidate for B, but is seldom permitted syntactically. The goal of the algebraic manipulations outlined above is to transform the expression in question into a form which is syntactically allowed as a while condition in the target programming language(s).

4.7.3 Loop termination

A common and practical way to prove that a loop terminates uses the concept of a *loop variant*. A loop variant is an expression whose value

- is decreased (or increased) by at least a fixed amount (e.g. 1) by each execution of the body of the loop and
- has a lower (or upper) bound.

The loop variant's bound derives from either the loop invariant or the loop condition, often both.

Example: Consider the loop

```
while i<n do i:=i+1; S1 endwhile
```

where the program segment S1 does not modify the value of i. Typically, the loop invariant for such a loop would contain the term i≤n. A suitable loop variant for this loop would be (n-i). Each execution of the body of the loop decreases the value of this expression by one. The loop condi-

tion guarantees that every time execution of the body of the loop begins, 0<n-i, so 0 is a lower bound for the loop variant. Also, the loop invariant ensures that 0≤n-i, i.e. that 0 is a lower bound for the loop variant. Thus the loop cannot continue to execute indefinitely.

Note that mathematical convergence to the lower (or upper) bound is not sufficient. The bound must be potentially violated in order to prove termination. When the loop variable is not necessarily an integer, this can give rise to situations which must be very carefully analyzed, both when designing loops and when proving that they terminate. Rounding when calculating the next value of a floating point loop variable can complicate the analysis further. In practice such situations arise in which it is "intuitively clear" that a loop must terminate, but it does not, in fact, always do so. Several simple and obvious algorithms for finding the zero of a function are examples of this problem, including (1) a binary search with too small an error tolerance and (2) linear interpolation.

The informally stated requirements above are sufficient for manually verifying termination in many practical cases. When a more formal approach is desired, the following lemma is useful.

Lemma for loop termination: Let the loop

while B do S endwhile

and a suitable loop invariant I (see rules W1 and W2 above) be given. If a numerical function var and a positive constant ε exist such that

$I \Rightarrow (B \Rightarrow 0<var)$ and [alternatively: $I \Rightarrow (var \leq 0 \Rightarrow \text{not } B)$, see below]
$\{I \text{ and } B \text{ and } var = var'\}\ S\ \{var \leq var'-\varepsilon\}$

then the above while loop will terminate (i.e. S will not be executed indefinitely). ■

The condition $(B \Rightarrow 0<var)$ often provides a convenient way to determine a suitable loop variant function var. Manipulate and weaken appropriately the expression for the while condition B into the form 0<..., and the expression to the right of the < symbol is a candidate for the loop variant var. While simplifying and weakening B, one may assume that the loop invariant I is true (is satisfied).

Note that

$B \Rightarrow 0<var$

is logically equivalent to

$var \leq 0 \Rightarrow \text{not } B$

so that the first condition in the lemma above can be written in the alternative form

$I \Rightarrow (var \leq 0 \Rightarrow \text{not } B)$

When this condition is written in this form, the validity of the lemma is perhaps more obvious: Because each execution of S reduces the value of var by at least the fixed amount ε, after var'/ε executions of S the value of var will be zero or less, so B will be false and the loop will termi-

nate. I.e., var'/ε is an upper bound on the number of times S will execute, a fact that is sometimes helpful when analyzing the time complexity of such a loop.

The sketch in the preceding paragraph of the proof of the lemma highlights the fact that there is nothing special about the value 0 for the lower bound of the loop variant. All that is necessary to ensure termination is that the while condition B (together with the loop invariant if needed) implies the existence of some lower bound for the loop variant.

Often, ε is the constant 1 and the condition (B = (0<var)) – which is stronger than the first condition in the lemma above – can be easily fulfilled by choosing var suitably. Loops that sequentially process all elements of a one dimensional array are typical examples of such cases. Notice that in the previous example above, B is i<n, which is equivalent to 0<n-i, so the obvious choice for the loop variant function var is (n-i).

The equality in the condition (B = (0<var)) above is too strong for some applications. If the example referred to above were searching for an array element with a certain value, the while condition B might be i<n and Z(i)≠key (instead of just i<n). The same loop variant function (n-i) applies and enables one to easily show that the loop must terminate, regardless of whether or not an array element with the value key is present. Here, B does imply that var is positive, but the reverse is not true, so B is not equivalent to the condition that var is positive.

As an example of a situation requiring the still weaker form of the first condition in the lemma above, i.e. I ⇒ (B ⇒ 0<var), consider a loop for processing every element of a two dimensional array X(j, k) with index j between 1 to Nj inclusive and index k between 1 to Nk inclusive. If the loop invariant I contains the terms 1≤Nj and 1≤Nk and j≤Nj and k≤Nk and if the while condition B is (j<Nj or k<Nk), then one suitable loop variant function var is Nj*(Nk-k) + (Nj-j). B alone does not imply that var is positive; also I is needed in order to show that var is positive.

A rule for a strict precondition for a while loop is presented in section 7.7 below. That rule includes the above lemma for loop termination.

4.8 Rules for the subprogram

The following rules are basically notational variants of previous rules. They have been put into a form here which is particularly suited for applying to subprograms and subprogram calls. They can also be applied to any program segment.

4.8.1 Rule SP1

If the execution of the program segment S does not modify any variable appearing in the condition B, then B has the same value before and after the execution of S. B is therefore a precondition of itself with respect to S:

{B} S {B}

4.8.2 Rule SP2

If

{B} S {B} and

{V} S {P}

then

{B and V} S {B and P}

Rule SP2 follows from rules SP1 and DC1.

To apply rule SP2, separate the postcondition into two anded parts. One part should reference only variables not changed by the execution of S; this part is its own precondition and is the condition B of this rule. The second part of the postcondition contains all references to variables which may be changed by the execution of S; this part is P. The precondition is separated into corresponding parts B and V.

4.8.3 Rule SP3

If

{B} S {B} and
$V \Rightarrow V1$
{V1} S {P1}
$P1 \Rightarrow P$

then

{B and V} S {B and P}

Rule SP3 combines rules SP2 and P1.

Typically, {V1} S {P1} is the given specification of S. B is a condition established by the program before S that is not affected by the execution of S. This rule takes into account the facts that (1) the previous program may establish a condition stronger than that required by S and (2) S may establish a stronger condition (P1) than that actually required by the subsequent program (P).

4.9 Rules for the declare statement

4.9.1 Applying rules for the assignment statement to the declare statement

The declare statement has the same effect upon the *values* of program variables as the corresponding assignment statement. In particular, the lemma for the assignment statement also holds for the declare statement, provided that the postcondition does not refer to the set associated with the variable being declared (because that set may be changed by the execution of the declare statement). Because the proof of rule A1 depends only upon that lemma (and not upon other aspects of the definition of the assignment statement), rule A1 applies also to the declare statement

$\{P_E^x\}$ declare (x, S, E) {P} completely

provided that P contains no reference to Set."x". Rule A2, being nothing other than a combination of rules A1 and P1, also applies in this case to the declare statement:

$$[V \Rightarrow P^{x}_{E}] \Rightarrow [\{V\} \text{ declare } (x, S, E) \{P\}]$$

To derive a precondition or to verify a correctness proposition about a declare statement when the postcondition refers to the set associated with the variable being declared, one may use the following more general rules D1 and D2, which are corresponding extensions of the rules A1 and A2 above.

4.9.2 Rule D1

Let the condition P and the statement declare (x, S, E) be given. If one forms the condition V by replacing in P

- every *reference to the value* of the variable x by the expression E (in parentheses) and, simultaneously,
- every *reference to the set associated* with the variable x by S,

then V is a complete precondition of P with respect to the declare statement, i.e.

$$\{P^{x,\ \text{Set."x"}}_{E,\ S}\} \text{ declare } (x, S, E) \{P\} \text{ completely}$$

Proof: The proof of this rule is very similar to the proof of rule A1. ■

This rule is due to Amitha Perera [personal communication].

Note the similarity between rules A1 and D1. In rule A1 for an assignment statement, only references to the *value* of the variable x are replaced, because the assignment statement changes only the value of the variable x. In rule D1 for the declare statement, both references to the *value* of the variable x and references to the *set* associated with the variable x are replaced, because the declare statement changes both the value of the variable x and the set associated with the variable x.

If P contains no reference to the set associated with the variable x (e.g. Set."x"), rules A1 and D1 give the same precondition.

4.9.3 Rule D2

If

$$V \Rightarrow P^{x,\ \text{Set."x"}}_{E,\ S}$$

then

$$\{V\} \text{ declare } (x, S, E) \{P\}$$

Proof: See the proof of rule A2 above. ■

Rule D2, which is simply a combination of rules P1 and D1, provides a way of verifying a correctness proposition about a declare statement. To verify that {V} declare (x, S, E) {P}, one verifies that $V \Rightarrow P^{x,\ Set.\text{“}x\text{”}}_{E,\ S}$. Thus the task of verifying the correctness proposition about a declare statement is reduced to the task of verifying the universal truth of a Boolean algebraic expression.

Note the correspondence between the pair of rules A1 and A2 and the pair of rules D1 and D2.

4.10 Applying rules to other types of program statements

4.10.1 Rules applicable to the release statement

In proofs of correctness, two different aspects concerning a release statement typically arise.

First, one must usually show that a certain structure of the data environment (often the same structure as a previous data environment) is established after the release statement has been executed. Typically this can be done by applying the definition of the release statement directly.

Second, one must show that a certain postcondition holds after execution of the release statement. Because the variable being released no longer exists after execution of the release statement, the postcondition will not refer to that variable. Often it will not refer to any other variable of the same name. The postcondition is then an expression whose value is not changed by the execution of the release statement and rule SP1 above applies:

> {B} release x {B}, if the value of B is not affected by the execution of release x.

If the postcondition of a release statement does refer to a variable of the same name as the variable being released, the variable being so referenced is not the variable being released, but the next variable of the same name in the data environment before executing the release statement. In such a case it is usually convenient to consider the program segment ending with the release statement in question and beginning with the previous declare statement that declared that variable:

```
{V}
declare (x, ., .)
...
release x
{P}
```

One then notes that this program segment does not change the value of a program variable named x, if any exists in the data environment before execution of the declare statement. That part of the postcondition P referring to such a variable x should be separated from those parts referring to variables modified by the program segment and then either rule SP2 or SP3 applied.

The procedures outlined above are usually adequate when manually verifying the correctness of a program. Generally valid formal rules for declare and release statements, suitable also for mechanized verification, are given in Chapter 9 below.

4.10.2 The null statement

The null statement changes nothing in the data environment (null.d=d, for all d∈𝔻), therefore, the value of any condition remains unchanged by its execution. Therefore, {B} null {B}, for any condition B.

5. Applying the rules in proofs of correctness: examples

To verify a correctness proposition about a program segment, one applies the relevant rules iteratively in order to decompose the correctness proposition to be proved into correctness propositions about ever smaller program segments until the level of basic statements (the assignment, declare, release and null statements) is reached. The remaining correctness propositions about such statements are then proved either by applying the applicable rules and verifying the resulting Boolean algebraic expressions (implications) or by direct application of the definition of the statement in question.

At each step of this decomposition process, the particular program segment in question and the proof task (verifying a correctness proposition or deriving a precondition) together determine the rule for decomposing the correctness proposition.

The following examples illustrate this process.

5.1 Proof of correctness of the linear search

In this example, the array A(i), i=1, ... n, is searched for the value of the variable x. The first element in A equal to x is to be found and its location (the value of its index in A) recorded and made available to the succeeding or calling program.

Formally, we describe this program segment by its specified precondition V and postcondition P:

V: $n \in \mathbf{Z}$ and $0 \le n$ [empty array allowed]

P: $n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n+1$ and found$\in \mathbb{B}$ [ranges of values of variables]
and $\bigwedge_{i=1}^{k-1} A(i) \ne x$ [all A(i) before the kth $\ne x$]
and ($k \le n$ and $A(k)=x$ and found [x found]
or $k=n+1$ and not found) [x not present in A]

The correctness proposition to be verified (theorem to be proved) is:

{V}
k:=1 [initialization]
while k≤n and A(k)≠x do ⎤
k:=k+1 ⎥W
endwhile ⎦
found:=(k≤n)
{P}

The designer of this program segment specified the following loop invariant

I: $n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n+1$ and $\bigwedge_{i=1}^{k-1} A(i) \ne x$

Proof: The correctness proposition above will be true by rule S1 if both of the following are true, where P1 is a condition yet to be determined:

{V} k:=1; W {P1} [1]

{P1} found:=(k≤n) {P} [2]

• Remaining to be proved are: [1], [2]

[2] will be true by rule A1 if $P1=P^{found}_{k\le n}$, which is, after simplification, as follows:

P1: n∈ **Z** and k∈ **Z** and 1≤k≤n+1 [definition of P1]
and $\text{and}_{i=1}^{k-1}$ A(i)≠x
and (k≤n and A(k)=x or k=n+1)

• Remaining to be proved is: [1]

[1] will be true by rule W2 if the following three conditions are true:

{V} k:=1 {I} [3]
{I and k≤n and A(k)≠x} k:=k+1 {I} [4]
I and not (k≤n and A(k)≠x) ⇒ P1 [5]

• Remaining to be proved are: [3], [4], [5]

[3] will be true by rule A2 if

$V \Rightarrow I_1^k$ [6]

• Remaining to be proved are: [4], [5], [6]

[4] will be true by rule A2 if

$I \text{ and } k\le n \text{ and } A(k)\ne x \Rightarrow I_{k+1}^k$ [7]

• Remaining to be proved are: [5], [6], [7]. These propositions are all Boolean algebraic expressions whose universal truth must be verified.

Proof of [6], which is: $V \Rightarrow I_1^k$

I_1^k
=
n∈ **Z** and 1≤n+1
=
n∈ **Z** and 0≤n
=
V ■

• Remaining to be proved are: [5], [7]

Proof of [5], which is: I and not (k≤n and A(k)≠x) ⇒ P1

I and not (k≤n and A(k)≠x)
=
I and (k>n or A(k)=x)
=
I and (k>n or k≤n and A(k)=x)
=
$n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n+1$
and $\text{and}_{i=1}^{k-1} A(i) \ne x$
and (k≤n and A(k)=x or k>n)
=
$n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n+1$
and $\text{and}_{i=1}^{k-1} A(i) \ne x$
and (k≤n and A(k)=x or k=n+1)
=
P1 ■

• Remaining to be proved is: [7]

Proof of [7], which is: I and k≤n and A(k)≠x ⇒ I_{k+1}^{k}

I_{k+1}^{k}
=
$n \in \mathbf{Z}$ and $k+1 \in \mathbf{Z}$ and $1 \le k+1 \le n+1$ and $\text{and}_{i=1}^{k+1-1} A(i) \ne x$
=
$n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $0 \le k \le n$ and $\text{and}_{i=1}^{k} A(i) \ne x$ [7.1]

Furthermore,

I and k≤n and A(k)≠x
=
$n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n+1$ and $\text{and}_{i=1}^{k-1} A(i) \ne x$ and k≤n and A(k)≠x
= [note that the and series below is not empty]
$n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n$ and $\text{and}_{i=1}^{k} A(i) \ne x$
⇒

$n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $0 \le k \le n$ and $\bigwedge_{i=1}^{k} A(i) \neq x$ [7.1]

= [see above]

I_{k+1}^{k} ■

As an informal proof of termination of the loop, we note that each execution of the body of the loop increases the value of k by one. But both the while condition and the loop invariant contain upper bounds on k, which the proof above shows will not be violated. Therefore, the body of the loop can be executed only a limited number of times.

This completes the verification of the correctness proposition about the linear search program. Note how the rules have been used to decompose the original correctness proposition to be proved in a hierarchical manner corresponding to the structure of the program. The decomposition process was repeated iteratively until only Boolean algebraic expressions remained, which were then verified by algebraic manipulation. This is a typical structure for proofs of correctness of programs.

In addition, the following theorem about the behaviour of this program segment is of interest, e.g. when applying rules SP1, SP2 or SP3 to a call to this subprogram: The execution of the program segment above changes at most the values of the program variables k and found; i.e., no other variable is modified. The validity of this theorem is evident by inspection of the program segment together with the definitions of the various types of statements appearing therein.

5.2 Strengthening the postcondition of the body of a loop

In the above example, part of the loop invariant I was the term $1 \le k \le n+1$ which bounds the loop variable k above and below. Being part of the loop invariant, this term appeared in both the precondition and the postcondition of the loop body:

{... $1 \le k \le n+1$ ...}
k:=k+1
{... $1 \le k \le n+1$ ...}

This correctness proposition will be true by rule A2 if

... $1 \le k \le n+1$ ... $\Rightarrow$ [... $1 \le k \le n+1$ ...]$_{k+1}^{k}$

which is equivalent to

... $1 \le k \le n+1$ ... $\Rightarrow$ [... $0 \le k \le n$ ...]

Note that the lower and upper bounds have been shifted by the amount by which the loop variable k was incremented or decremented. One new bound (here $0 \le k$) is automatically satisfied; the required precondition is weaker than the one actually satisfied. When working backward through the program segment, one can strengthen this condition (cf. rule P1). Carrying out this strengthening already in the postcondition of the loop body often simplifies the algebraic manipulation of intermediate conditions prevailing at points within the body of the loop, e.g. by

ensuring that certain series are not empty. The other bound (here k≤n) is not automatically satisfied; some other term in the precondition must ensure that it is satisfied. This term typically comes from the loop condition, as it does in the example above.

Strengthening the postcondition in the above example would be done by requiring the following stronger correctness proposition to be satisfied:

{... 1≤k≤n+1 ...}
k:=k+1
{... 2≤k≤n+1 ...}

If this correctness proposition is satisfied, the weaker one above will be satisfied by rule P1.

If, as in the example above, the execution of the body of the loop increases the value of the loop variable, its lower bound in the postcondition may be increased correspondingly. If the execution of the body of the loop decreases the value of the loop variable, its upper bound in the postcontion may be decreased correspondingly.

Whenever the execution of the loop body increases or decreases the value of the loop variable, one should consider strengthening the postcondition of the body of the loop as illustrated above. Doing so will sometimes simplify algebraic manipulation later in the proof and is, therefore, generally to be recommended.

5.3 Proof of correctness of the merge

5.3.1 External view of the subprogram merge

In this example, the values in two arrays A and B are merged and copied to the array C. The values in the arrays A and B are assumed to be sorted before this subprogram is executed. After execution of the subprogram merge the values in C must be sorted.

Formally, we describe this program segment by its specified (ordinary) precondition V and postcondition P:

V: $na \in \mathbf{Z}$ and $0 \le na$ and $nb \in \mathbf{Z}$ and $0 \le nb$ [empty input arrays allowed]
$\text{and}_{i=1}^{na-1}\ A(i) \le A(i+1)$ $\text{and}_{i=1}^{nb-1}\ B(i) \le B(i+1)$ [A, B sorted]

P: $(\&_{i=1}^{na+nb}\ [C(i)])$ Perm $(\&_{i=1}^{na}\ [A(i)]\ \&_{i=1}^{nb}\ [B(i)])$ [final values in C come from A and B]
$\text{and}_{i=1}^{na+nb-1}\ C(i) \le C(i+1)$ [C sorted]

The function Perm (permutation) above, written as an infix operator, maps a pair of sequences to the set {false, true}. Perm is defined axiomatically as follows:

[] Perm [] [the empty sequence is a permutation of itself]

[(a & b & c) Perm (d & b & e)] = [(a & c) Perm (d & e)] [subsequences cancel]

where a, b, c, d and e are sequences.

Correctness proposition 1: {V} call merge {P}

Correctness proposition 2: For every data environment d in the domain of the subprogram merge, merge.d = d structurally.

Correctness proposition 3: The execution of the subprogram merge modifies at most the values of the variables C(i) for i∈ **Z** and 1≤i≤na+nb.

Correctness proposition 4: The execution of merge does not continue unendlessly, i.e. terminates in finite time.

5.3.2 Internal view of the subprogram merge

Correctness proposition 1:

```
{V}
declare (ia, Z, 1); declare (ib, Z, 1); declare (ic, Z, 1)          [abbreviated init below]
while ia ≤ na or ib ≤ nb do                                  ⎤      [B1]
    if ib > nb or (ia ≤ na and A(ia) ≤ B(ib))          ⎤     ⎥W     [B2]
    then C(ic) := A(ia); ia := ia + 1                  ⎥S    ⎥      [S1]
    else C(ic) := B(ib); ib := ib + 1                  ⎥     ⎥      [S2]
    endif                                              ⎦     ⎥
    ic := ic + 1                                             ⎥
endwhile                                                     ⎦
release ia; release ib; release ic                                 [R]
{P}
```

Correctness proposition 2: For all d in the domain of the subprogram merge, merge.d = d structurally. This theorem can be easily verified informally by inspection of the program segment, see the definitions of the several types of program statements involved. ■

Correctness proposition 3: The execution of the subprogram merge modifies at most the values of the variables C(i) for i∈ **Z** and 1≤i≤na+nb. With the exception of the range of i, this theorem can also be easily verified informally by inspection. The stated bounds for the values of the indices to the array C will follow from the preconditions of the assignment statements C(ic):=..., see the proof below.

The designer of this program segment specified the following loop invariant

I: I0 and I1 and I2 and I3 and I4 and I5 and I6 and I7 and I8

where

I0: na∈ **Z** and 0≤na and nb∈ **Z** and 0≤nb [range of na, nb (from precondition)]
I1: ia∈ **Z** and 1≤ia≤na+1 [range of ia]
I2: ib∈ **Z** and 1≤ib≤nb+1 [range of ib]
I3: (ic-1) = (ia-1) + (ib-1) [# in C = # from A + # from B]

I4: $(\&_{i=1}^{ic-1}\ [C(i)])$ Perm $(\&_{i=1}^{ia-1}\ [A(i)]\ \&_{i=1}^{ib-1}\ [B(i)])$ [values in C from A, B]

I5: $(1<ic \text{ and } ia\leq na) \Rightarrow C(ic-1)\leq A(ia)$ [next from A≥last into C, if any]

I6: $(1<ic \text{ and } ib\leq nb) \Rightarrow C(ic-1)\leq B(ib)$ [next from B≥last into C, if any]

I7: $\text{and}_{i=1}^{ic-2}\ C(i)\leq C(i+1)$ [C sorted]

I8: $\text{and}_{i=1}^{na-1}\ A(i)\leq A(i+1)$ and $\text{and}_{i=1}^{nb-1}\ B(i)\leq B(i+1)$ [A, B sorted]

Note that this loop invariant is symmetrical in a and b, i.e. if one replaces ia, ib, na, nb, A and B simultaneously by ib, ia, nb, na, B and A respectively, then the result is the loop invariant itself. We will make use of this symmetry in the proof below in order to reduce the amount of algebraic manipulation required to complete the proof.

Proof of correctness proposition 1: Correctness proposition 1 will be true by rule S1 if both of the following are true:

{V} init, W {P} [1]

{P} R {P} [2]

- Remaining to be proved are: [1], [2].

[2] is true by rule SP1 because P contains no references to ia, ib or ic, the variables being released by the statements in R.

- Remaining to be proved is: [1].

[1] will be true by rule W2 if all of the following are true:

{V} init {I} [3]

{I and B1} S, ic:=ic+1 {I} [4]

I and not B1 $\Rightarrow$ P [5*]

- Remaining to be proved are: [3], [4], [5*]. An asterisk (*) indicates that the proposition is a Boolean algebraic expression to be verified, which therefore does not need to be decomposed further by the application of rules.

[3] will be true by rules A1, A2 and S1 if

$V \Rightarrow [[I_1^{ic}]_1^{ib}]_1^{ia}$ [6*]

- Remaining to be proved are: [4], [5*], [6*].

[4] will be true by rules A1 and S1 if

$\{I \text{ and } B1\}\ S\ \{I_{ic+1}^{ic}\}$ [7]

- Remaining to be proved are: [5*], [6*], [7].

[7] (in which S is an if statement) will be true by rule IF1 if

$$\{\text{I and B1 and B2}\}\ \text{S1}\ \{I^{ic}_{ic+1}\} \qquad [8]$$

$$\{\text{I and B1 and not B2}\}\ \text{S2}\ \{I^{ic}_{ic+1}\} \qquad [9]$$

• Remaining to be proved are: [5*], [6*], [8], [9]. Propositions 8 and 9 are nearly symmetrical; the following steps will transform one of them so that the resulting two propositions are symmetrical, in which case only one needs to be algebraically verified.

We note that

B1 and B2
= [ia≤na and [ib>nb or A(ia)≤B(ib)]]

and that

B1 and not B2
= [ib≤nb and [ia>na or B(ib)<A(ia)]]

so that [8] and [9] can be rewritten as

{I and ia≤na and [ib>nb or A(ia)≤B(ib)]}
C(ic):=A(ia); ia:=ia+1 $\{I^{ic}_{ic+1}\}$ [8]

{I and ib≤nb and [ia>na or B(ib)<A(ia)]}
C(ic):=B(ib); ib:=ib+1 $\{I^{ic}_{ic+1}\}$ [9]

[9] will be true by rule P1 if the following correctness proposition with a weaker precondition is true:

{I and ib≤nb and [ia>na or B(ib)≤A(ia)]}
C(ic):=B(ib); ib:=ib+1 $\{I^{ic}_{ic+1}\}$ [10]

• Remaining to be proved are: [5*], [6*], [8], [10].

Propositions 8 and 10 are symmetrical in a and b, i.e. if in either one ia, ib, na, nb, A and B are simultaneously replaced by ib, ia, nb, na, B and A respectively, then the other proposition results. Therefore, a proof of either proposition can be transformed into a proof of the other by the same replacement of the symbols occurring therein. Consequently, we need not prove both [8] and [10] algebraically; a proof of either suffices.

• Remaining to be proved are: [5*], [6*], [10].

[10] will be true by rules A1, A2 and S1 if

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [11^*]$$

• Remaining to be proved are: [5*], [6*], [11*]. I.e., only Boolean algebraic expressions must still be verified. The right hand side of the implication [11*] would be long if written out completely. We therefore reduce this proposition further as follows.

[11*] will be true by a property of the implication ([(X⇒Y) and (X⇒Z)] = [X⇒(Y and Z)]) (cf. rule DC3) if

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I0]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [12^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I1]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [13^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I2]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [14^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I3]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [15^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I4]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [16^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I5]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [17^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I6]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [18^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I7]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [19^*]$$

$$I \text{ and } ib \le nb \text{ and } [ia > na \text{ or } B(ib) \le A(ia)] \Rightarrow [[I8]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)} \quad [20^*]$$

• Remaining to be proved are: [5*], [6*], [12*], [13*], [14*], [15*], [16*], [17*], [18*], [19*], [20*].

[5*] is true:

$$P$$

$$=$$

$$(\&_{i=1}^{na+nb}\,[C(i)])\ \text{Perm}\ (\&_{i=1}^{na}\,[A(i)]\ \&_{i=1}^{nb}\,[B(i)])$$

$$\text{and}_{i=1}^{na+nb-1}\ C(i) \le C(i+1)$$

$$\Leftarrow$$

$$I4 \text{ and } ic-1=na+nb \text{ and } ia-1=na \text{ and } ib-1=nb \text{ and } I7 \text{ and } ic-2=na+nb-1$$

$$\Leftarrow$$

$$I4 \text{ and } I7 \text{ and } (ic-1)=(ia-1)+(ib-1) \text{ and } ia=na+1 \text{ and } ib=nb+1$$

$$\Leftarrow$$

$$I3 \text{ and } I4 \text{ and } I7$$
$$\text{and } na \in \mathbf{Z} \text{ and } nb \in \mathbf{Z} \text{ and } ia \in \mathbf{Z} \text{ and } ia \le na+1 \text{ and } ib \in \mathbf{Z} \text{ and } ib \le nb+1$$

and ia>na and ib>nb

⇐

I0 and I1 and I2 and I3 and I4 and I7 and ia>na and ib>nb

⇐

I and ia>na and ib>nb

=

I and not B1 ■

• Remaining to be proved are: [6*], [12*], [13*], [14*], [15*], [16*], [17*], [18*], [19*], [20*].

[6*] is true:

$[[I_1^{ic}]_1^{ib}]_1^{ia}$

=

$na \in \mathbf{Z}$ and $0 \le na$ and $nb \in \mathbf{Z}$ and $0 \le nb$
and$_{i=1}^{na-1}$ $A(i) \le A(i+1)$ and$_{i=1}^{nb-1}$ $B(i) \le B(i+1)$

=

V ■

• Remaining to be proved are: [12*], [13*], [14*], [15*], [16*], [17*], [18*], [19*], [20*].

[12*] is true:

$[[I0_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)}$

= [I0 contains no references to ic, ib or C(.)]

I0

⇐

I

⇐

I and ib≤nb and [ia>na or B(ib)≤A(ia)] ■

• Remaining to be proved are: [13*], [14*], [15*], [16*], [17*], [18*], [19*], [20*].

[20*] is true:

$[[I8_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)}$

= [I8 contains no references to ic, ib or C(.)]

I8

⇐

I

⇐

I and ib≤nb and [ia>na or B(ib)≤A(ia)] ■

• Remaining to be proved are: [13*], [14*], [15*], [16*], [17*], [18*], [19*].

[13*] is true:

$$[[I1_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)}$$

= [I1 contains no references to ic, ib or C(.)]

I1

⇐

I

⇐

I and ib≤nb and [ia>na or B(ib)≤A(ia)] ■

• Remaining to be proved are: [14*], [15*], [16*], [17*], [18*], [19*].

[14*] is true:

$$[[I2_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)}$$

=

ib+1∈ **Z** and 1≤ib+1≤nb+1

=

ib∈ **Z** and 0≤ib≤nb

⇐

ib∈ **Z** and 1≤ib≤nb+1 and ib≤nb

=

I2 and ib≤nb

⇐

I and ib≤nb and [ia>na or B(ib)≤A(ia)] ■

• Remaining to be proved are: [15*], [16*], [17*], [18*], [19*].

[15*] is true:

$$[[I3_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)}$$

=

(ic+1-1) = (ia-1) + (ib+1-1)

=

(ic-1) = (ia-1) + (ib-1)

=

I3

⇐

I and ib≤nb and [ia>na or B(ib)≤A(ia)] ■

• Remaining to be proved are: [16*], [17*], [18*], [19*].

[16*] is true:

$$[[I4^{ic}_{ic+1}]^{ib}_{ib+1}]^{C(ic)}_{B(ib)}$$

=

$$[(\&^{ic}_{i=1}\,[C(i)])\ \text{Perm}\ (\&^{ia-1}_{i=1}\,[A(i)]\ \&^{ib}_{i=1}\,[B(i)])]^{C(ic)}_{B(ib)}$$

⇐

$$1{\le}ib \text{ and } 1{\le}ic \text{ and } [(\&^{ic}_{i=1}\,[C(i)])\ \text{Perm}\ (\&^{ia-1}_{i=1}\,[A(i)]\ \&^{ib}_{i=1}\,[B(i)])]^{C(ic)}_{B(ib)}$$

=

$$[1{\le}ib \text{ and } 1{\le}ic \text{ and } (\&^{ic}_{i=1}\,[C(i)])\ \text{Perm}\ (\&^{ia-1}_{i=1}\,[A(i)]\ \&^{ib}_{i=1}\,[B(i)])]^{C(ic)}_{B(ib)}$$

=

$$[1{\le}ib \text{ and } 1{\le}ic \text{ and } (\&^{ic-1}_{i=1}\,[C(i)]\ \&\ [C(ic)])\ \text{Perm}\ (\&^{ia-1}_{i=1}\,[A(i)]\ \&^{ib-1}_{i=1}\,[B(i)]\ \&\ [B(ib)])]^{C(ic)}_{B(ib)}$$

=

$$[1{\le}ib \text{ and } 1{\le}ic \text{ and } (\&^{ic-1}_{i=1}\,[C(i)]\ \&\ [B(ib)])\ \text{Perm}\ (\&^{ia-1}_{i=1}\,[A(i)]\ \&^{ib-1}_{i=1}\,[B(i)]\ \&\ [B(ib)])$$

=

$$[1{\le}ib \text{ and } 1{\le}ic \text{ and } (\&^{ic-1}_{i=1}\,[C(i)])\ \text{Perm}\ (\&^{ia-1}_{i=1}\,[A(i)]\ \&^{ib-1}_{i=1}\,[B(i)])$$

=

$$I4 \text{ and } 1{\le}ib \text{ and } 1{\le}ic$$

⇐

$$I4 \text{ and } 1{\le}ia \text{ and } 1{\le}ib \text{ and } (ic-1) = (ia-1) + (ib-1)$$

⇐

$$I1 \text{ and } I2 \text{ and } I3 \text{ and } I4$$

⇐

$$I \text{ and } ib{\le}nb \text{ and } [ia{>}na \text{ or } B(ib){\le}A(ia)]$$ ■

• Remaining to be proved are: [17*], [18*], [19*].

[19*] is true:

$$[[I7^{ic}_{ic+1}]^{ib}_{ib+1}]^{C(ic)}_{B(ib)}$$

=

$$[[[\text{and}^{ic-2}_{i=1}\ C(i){\le}C(i+1)]^{ic}_{ic+1}]^{ib}_{ib+1}]^{C(ic)}_{B(ib)}$$

=

$$[\text{and}^{ic-1}_{i=1}\ C(i){\le}C(i+1)]^{C(ic)}_{B(ib)}$$

=

$$[ic\text{-}1<1 \text{ or } ic\text{-}1\geq 1 \text{ and} \underset{i=1}{\overset{ic-1}{}} C(i)\leq C(i+1)]\underset{B(ib)}{\overset{C(ic)}{}}$$

=

$$[ic\text{-}1<1 \text{ or } ic\text{-}1\geq 1 \text{ and } C(ic-1)\leq C(ic) \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)]\underset{B(ib)}{\overset{C(ic)}{}}$$

=

$$ic\text{-}1<1 \text{ or } ic\text{-}1\geq 1 \text{ and } C(ic-1)\leq B(ib) \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)$$

=

$$[ic\text{-}1<1 \text{ or } ic\text{-}1\geq 1 \text{ and } C(ic-1)\leq B(ib)] \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)$$

=

$$[ic<2 \text{ or } C(ic-1)\leq B(ib)] \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)$$

=

$$[ic\geq 2 \Rightarrow C(ic-1)\leq B(ib)] \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)$$

⇐

$$ic\in \mathbf{Z} \text{ and } ib\leq nb \text{ and } [2\leq ic \Rightarrow C(ic-1)\leq B(ib)] \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)$$

⇐

$$ic\in \mathbf{Z} \text{ and } ib\leq nb \text{ and } [1<ic \text{ and } ib\leq nb \Rightarrow C(ic-1)\leq B(ib)] \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)$$

⇐

$$ia\in \mathbf{Z} \text{ and } ib\in \mathbf{Z} \text{ and } (ic-1) = (ia-1) + (ib-1)$$
$$\text{and } ib\leq nb \text{ and } [1<ic \text{ and } ib\leq nb \Rightarrow C(ic-1)\leq B(ib)] \text{ and} \underset{i=1}{\overset{ic-2}{}} C(i)\leq C(i+1)$$

⇐

$$I1 \text{ and } I2 \text{ and } I3 \text{ and } I6 \text{ and } I7 \text{ and } ib\leq nb$$

⇐

$$I \text{ and } ib\leq nb \text{ and } [ia>na \text{ or } B(ib)\leq A(ia)] \blacksquare$$

• Remaining to be proved are: [17*], [18*].

[18*] is true:

$$[[I6\underset{ic+1}{\overset{ic}{}}]\underset{ib+1}{\overset{ib}{}}]\underset{B(ib)}{\overset{C(ic)}{}}$$

=

$$[(1<ic+1 \text{ and } ib+1\leq nb) \Rightarrow C(ic)\leq B(ib+1)]\underset{B(ib)}{\overset{C(ic)}{}}$$

=

$$[(1<ic+1 \text{ and } ib+1\leq nb) \Rightarrow B(ib)\leq B(ib+1)]$$

=

$$[(0<ic \text{ and } ib\leq nb-1) \Rightarrow B(ib)\leq B(ib+1)]$$

⇐

$1 \le ic$ and $1 \le ib$ and $[(0 < ic$ and $ib \le nb-1) \Rightarrow B(ib) \le B(ib+1)]$

=

$1 \le ic$ and $1 \le ib$ and $[1 \le ib \le nb-1 \Rightarrow B(ib) \le B(ib+1)]$

$\Leftarrow$

$1 \le ic$ and $1 \le ib$ and $\bigwedge_{i=1}^{nb-1} B(i) \le B(i+1)$

$\Leftarrow$

$1 \le ia$ and $1 \le ib$ and $(ic-1) = (ia-1) + (ib-1)$ and $\bigwedge_{i=1}^{nb-1} B(i) \le B(i+1)$

$\Leftarrow$

I1 and I2 and I3 and I8

$\Leftarrow$

I and $ib \le nb$ and $[ia > na$ or $B(ib) \le A(ia)]$ ■

• Remaining to be proved is: [17*].

$[[I5]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)}$

=

$[[[(1 < ic$ and $ia \le na) \Rightarrow C(ic-1) \le A(ia)]_{ic+1}^{ic}]_{ib+1}^{ib}]_{B(ib)}^{C(ic)}$

=

$(1 < ic+1$ and $ia \le na) \Rightarrow B(ib) \le A(ia)$

=

$ic \le 0$ or $ia > na$ or $B(ib) \le A(ia)$

$\Leftarrow$

$ia > na$ or $B(ib) \le A(ia)$

$\Leftarrow$

I and $ib \le nb$ and $[ia > na$ or $B(ib) \le A(ia)]$ ■

This completes the proof of correctness proposition 1.

Correctness proposition 2 has already been (informally) proved, see the statement of correctness proposition 2.

Proof of correctness proposition 3: Only the stated range of values of ic ($ic \in \mathbf{Z}$ and $1 \le ic \le na+nb$) when the assignment statements C(ic):=... are executed needs to be verified (see the statement of this proposition).

The preconditions of these statements are

I and $ia \le na$ and $[ib > nb$ or $A(ia) \le B(ib)]$ and
I and $ib \le nb$ and $[ia > na$ or $B(ib) < A(ia)]$

see the alternative formulations of [8] and [9] above respectively.

It follows from the first precondition above that the value of ic is in the stated range:

I and ia≤na and [ib>nb or A(ia)≤B(ib)]

⇒

I1 and ia≤na and I2 and I3

⇒

ia∈ **Z** and 1≤ia≤na and ib∈ **Z** and 1≤ib≤nb+1 and (ic-1) = (ia-1) + (ib-1)

⇒

ia∈ **Z** and 1≤ia≤na and ib∈ **Z** and 1≤ib≤nb+1 and ic=ia+ib-1

⇒

ic∈ **Z** and 1≤ic≤na+nb ■

Similarly,

I and ib≤nb and [ia>na or B(ib)<A(ia)]

⇒

I1 and I2 and ib≤nb and I3

⇒

ia∈ **Z** and 1≤ia≤na+1 and ib∈ **Z** and 1≤ib≤nb and (ic-1) = (ia-1) + (ib-1)

⇒

ia∈ **Z** and 1≤ia≤na+1 and ib∈ **Z** and 1≤ib≤nb and ic=ia+ib-1

⇒

ic∈ **Z** and 1≤ic≤na+nb ■

Each of the assignment statements C(ic):=... will, therefore, assign a value to an array variable C(.) with index in the stated range. This completes the proof of correctness proposition 3.

Proof of correctness proposition 4: The loop in merge terminates because each execution of the loop body increases the value of the variable ic by one and the loop invariant implies an upper bound on ic:

I

⇒

I1 and I2 and I3

⇒

ia≤na+1 and ib≤nb+1 and (ic-1) = (ia-1) + (ib-1)

⇒

ic≤na+nb+1 ■

More formally, define the loop variant function to be (na+nb+1-ic) and show that the hypotheses of the lemma for loop termination in section 4.7.3 above are satisfied.

This completes the proof of correctness proposition 4.

6. Designing correct programs

6.1 General

When designing a program segment — i.e. when solving the task {V} S? {P} — one in effect designs a correctness proof and derives the various parts of the program segment so that the hypotheses of the relevant rules are satisfied. Looked at differently, but equivalently, one uses the rules to decompose the overall design task into design tasks involving smaller components of the program, repeating this process iteratively as necessary.

The designer begins by examining the given precondition V and postcondition P and considering how P can be transformed into an expression equal to or following from V (cf. rule P1). The transformation steps correspond to specific rules and their associated program statement types. From this process the program segment is derived, almost as a by-product of developing the proof. Some of the commonly useful heuristics for transforming a postcondition in such a manner are outlined in the sections below.

6.2 The interface specification

The starting point for designing a program segment is the specification of the interface between the program segment to be designed and its environment, i.e. the correctness proposition(s) the program segment must satisfy. These propositions typically include three types of information:

(1) the precondition and the postcondition,

(2) the structure of the data environment resulting from the execution of the program segment to be designed and

(3) the program variables that may not be changed by the execution of the program segment in question. This information is usually required by the prover of the correctness of the calling program segment (cf. condition B in rules SP1, SP2 and SP3). This information is most conveniently given in the form of an *exhaustive* list of all changes which the execution of the program segment to be designed may cause.

Often parts (2) and (3) above can be combined into a single statement, e.g. of the form

> (program segment).d = [(x, ., .), ...] & d for all d in some specified set

or

> (program segment).d = [(x, ., .), ...] & d except for the values of variables y, z ..., for all d in some specified set

6.3 Applying the rules to deduce the program statement types

6.3.1 Assignment and declare statements

If a given postcondition P can be reduced to a given precondition V (or to an expression which follows from V) by replacing one or more variable names by some expressions (in parentheses), then the application of rule A1 or A2 is indicated. This in turn suggests an assignment statement or a sequence of assignment statements, provided that the variable names being replaced are the

names of variables which the program segment to be designed may change. Alternatively, declare statements must be considered, because rules A1 and A2 often apply to a declare statement also. The choice between assignment and declare statements will normally follow from parts (2) or (3) of the interface specification, see section 6.2 above. If a declare statement is selected, a corresponding release at a later point in the program segment will often also be indicated by parts (2) or (3) of the interface specification.

Often it will be immediately clear which variable(s) may be changed by the program segment being designed. Such variables are candidates for the left sides of assignment statements or for corresponding declaration statements. If the expression for the new value is not immediately obvious, one can formulate the appropriate correctness proposition and solve the resulting expression for the unknown expression. For example, consider V and P to be the pre- and postconditions for an assignment statement of the form x:=E, where E is unknown. From the rule for the assignment statement it follows that we must find the expression E so that $V \Rightarrow P^{x}_{E}$. One then substitutes the actual expressions for V and P in this implication, replaces x by (the letter) E in P, and manipulates the resulting expression to find a suitable expression for E.

This approach can also be used when more than one variable can be changed by the program segment being designed. For example, if the variables x and y may be changed, one finds expressions E1 and E2 so that $V \Rightarrow [P^{y}_{E2}]^{x}_{E1}$. Then the sequence of assignment statements x:=E1; y:=E2 is a solution for the design task {V} S ? {P}.

6.3.2 Sequence of program segments

If the postcondition P consists of several terms anded together that successively contain references to additional variables to be calculated by the program segment in question, i.e. if P is of the form

P1(x1) and P2(x1, x2) and P3(x1, x2, x3) ...

then a sequence of program segments is often indicated which calculate values for the several variables one after another, e.g.

{V}
S1 [modifies x1 only]
{P1(x1)}
S2 [modifies x2 only]
{P1(x1) and P2(x1, x2)}
S3 [modifies x3 only]
{P1(x1) and P2(x1, x2) and P3(x1, x2, x3)} ...

Thus each part of the sequence of statements ensures the truth of an additional term in the given postcondition P.

More generally, each program segment in the sequence above might calculate the values of several closely related variables instead of only one as in the above example. The basic idea of

building on previous results to calculate additional results in several consecutive steps remains unchanged.

6.3.3 If statement

If, in the course of attempting to design a program segment S so that {V} S {P}, a candidate S1 is determined which has a stronger precondition, e.g. V and B, the application of rule IF1 is often indicated, suggesting that one embed S1 in the if statement

    if B then S1 else S2 endif

leaving the second design task {V and not B} S2? {P} to be solved.

6.3.4 While loop

If the postcondition P contains a series which is not present in the precondition V, one should first consider reducing the series to an empty series or to a single term (or even to a fixed number of terms) by replacing one or more variables in the lower or upper limit of the series by some constant or expression, leading to the corresponding assignment or declare statements (see section 6.3.1 above).

If that is not permitted, e.g. because no variable appearing in the lower and upper limits may be changed by the program segment being designed, a loop is usually indicated. The designer must then (1) decide upon a loop invariant, (2) design the loop's initialization, (3) determine the loop condition and (4) design the body of the loop. The hypotheses of rule W2 provide the basis for each of these steps, especially steps 2 through 4. The loop invariant represents the most important design decision regarding the loop. The other steps 2 through 4 are then relatively straightforward and often amount to little more than algebraic derivation of the parts of the loop.

6.3.4.1 The loop invariant

The loop invariant I must represent a generalization of the situations prevailing before and after execution of the loop; these are, loosely speaking, represented by the precondition and the postcondition. A number of useful heuristics exist for determining a suitable loop invariant, all of which are special instances of generalizing the initial and final situations referred to above. Usually the postcondition is the lengthier and structurally more involved expression and it, therefore, is typically the primary point of departure for determining a loop invariant.

Sometimes a postcondition of the form P1 and P2 can be separated into P1 as the loop invariant and P2 as the negation of the loop condition, e.g. when (1) not P2 has a form which is syntactically permitted as a loop condition and (2) not P2 contains references to variables which may be changed by statements in the loop. The latter condition is necessary so that termination can be guaranteed.

Another approach is to examine the postcondition P with a view toward generalizing it in such a way that it can be easily initialized. Frequently this involves introducing a new variable (e.g. so that certain series can be made empty by assigning a suitable value to the newly introduced variable). When a new variable is introduced in order to form the loop invariant, a strong

condition on its value range should be incorporated into the loop invariant (i.e. a condition stating the set of which its values are elements and giving lower and upper bounds on its values).

In some situations the postcondition contains a series (e.g. a sum, an and-series, an or-series, etc.) that is not present in the precondition. If neither bound on the series may be changed by the program segment being designed, then one typically introduces a new loop variable to create a "hole" in the series in the loop invariant. Initially both series thus created will be empty and in the final state the hole will be empty, so that the loop invariant implies the postcondition.

A less formal, often very useful approach for determining a loop invariant consists of examining a diagrammatic form of the postcondition and a corresponding diagram representing the precondition. The two diagrams are then generalized to obtain a diagram for the loop invariant. From the diagram for the loop invariant an algebraic expression can then be formulated.

6.3.4.2 The initialization

For designing the initialization init of a loop the hypothesis {V} init {I} of rule W2 provides the main part of the specification. Typically I can be transformed into V or into an expression which follows from V by substituting constants or simple expressions for certain variables, suggesting one or more assignment or declare statements for the initialization (see section 6.3.1 above).

6.3.4.3 The while condition

The while condition B must satisfy another hypothesis of rule W2: I and not B $\Rightarrow$ P. Furthermore, B must be an expression which is syntactically permitted as a while condition. (Note that not P is semantically always a candidate for B, but is typically of a form not allowed as a while condition.)

The requirement I and not B $\Rightarrow$ P can be rewritten in any of a number of equivalent forms, e.g.

I and not P $\Rightarrow$ B	[1]
I$\Rightarrow$ (not B$\Rightarrow$P)	[2]
I$\Rightarrow$(not P$\Rightarrow$B)	[3]

etc., several of which provide a useful basis for algebraically deriving or informally deducing a suitable while condition.

The hypothesis of rule W2 itself (I and not B $\Rightarrow$ P) suggests asking the question "Given the truth of the loop invariant I, what additional information or condition is needed to guarantee that the postcondition P is also true?" That additional condition is a candidate for not B, so negating it gives B.

Expression [1] above suggests that one form the expression I and not P, simplify it (e.g. by replacing Boolean terms in P which are also in I by the constant true) and weaken it to obtain an expression which is syntactically permitted as a while condition. If termination of the loop is desired, one must be careful not to weaken the expression too much, see below.

Similarly, expressions [2] and [3] above suggest that one start with P or not P and strengthen or weaken it respectively while assuming the truth of I in order to find a suitable expression for not B or B respectively.

Clearly, weakening an expression to the extreme, i.e. to the Boolean constant true, to obtain B would always satisfy the above requirements. But then the loop would never terminate. Assuming that the loop should terminate, the designer must also consider the question, "How much can I weaken a candidate for the while condition B without making it impossible to guarantee termination?" The following observations are of use in answering this question.

(1) The while condition B must contain at least one reference to a variable which the body of the loop being designed may modify. If this condition is not met, then the value of B cannot be changed by executing the body of the loop. Therefore, if the body of the loop is ever executed, it will be executed again and again and the loop will not terminate. If the body of the loop is never executed, then the loop is unnecessary. I.e., if this condition is not met then either the loop is unnecessary or its termination cannot be guaranteed.

(2) If I⇒B, then B will always be true and the loop will not terminate. Therefore, do not weaken the candidate for B so much that it follows from I alone. Similarly, B may not follow from I and P (if I and P ⇒ B, then the loop will not terminate). These two conditions (I⇒B as well as I and P ⇒ B) are equivalent whenever I and not B ⇒ P, so the designer can restrict attention to either one alone.

(3) If a candidate for B is the conjunction of two terms (e.g. B1 and B2), and one of these terms follows from I (i.e. I⇒B1), then weaken the conjunction B1 and B2 by dropping B1 and continue with B2 as the candidate for B. Why is this valid? Why or under what conditions will B2 alone not violate guideline 2 above?

(4) A while condition B that is false whenever the postcondition P is true would seem to be an ideal choice, because it would result in termination of the loop as soon as P is satisfied. However, such a requirement restricts the choice of B too much to be practical. More precisely, it restricts the choice of B to the expression not P simplified assuming the truth of the loop invariant I but not otherwise weakened. The resulting expression often includes subexpressions not syntactically permitted in a while condition.

In summary, the following strategy is probably the generally most useful for algebraically deriving a suitable while condition: Start with the expression not P and simplify it assuming the truth of I. Weaken it by eliminating series that are syntactically not permitted in a while condition. Weaken it by applying guideline 3 above. Otherwise, weaken it as little as possible and in particular, be careful not to weaken it so much that it becomes as weak or weaker than the loop invariant I. Alternatively but equivalently, one can start with the expression (I and not P) and proceed in the same manner.

6.3.4.4 The loop body

The body S of the loop must satisfy the hypothesis of rule W2 that

{I and B} S {I}

Normally (but not always) S must also ensure progress toward termination, i.e. fulfillment of the postcondition P or, equivalently, fulfillment of the termination condition not B. (Note that the null statement for S always satisfies the requirement that {I and B} S {I}, but does not contribute toward termination of the loop.) Often S can be constructed by examining the while condition B, determining one or more statements which ensure progress toward fulfilling the condition not B, and then constructing the rest of S so that truth of the postcondition I is reestablished (of course without reversing the progress toward termination). For example, the design task

{I and B} S? {I}

might be decomposed in this manner to

{I and B} S1?; i:=i+1 {I}

and in turn into

$\{I \text{ and } B\}\ S1?\ \{I^{i}_{i+1}\}$

Often it is desirable to strengthen the postcondition of the body of the loop before designing the body of the loop, see section 5.2 above.

If the designer wishes to modify the loop variable i first and place the rest of the body of the loop later, i.e. if the designer decides upon a loop body of the form {I and B} i:=i+1; S2? {I}, then the design task for S2 becomes $\{I \text{ and } B\}^{i}_{i-1}$ S2? {I}. See section 8.5 below.

To take termination more formally into account when designing the loop body, one begins by identifying a suitable loop variant var and a suitable ε, see section 4.7.3 above. Candidates for the loop variant var can be derived from the while condition B and the loop invariant I. Then the loop body must be designed so that both of the following conditions are fulfilled:

{I and B} S? {I} [from rules W1 and W2]

{I and B and var = var'} S? {var ≤ var'-ε} [from the lemma for loop termination]

Because the term var=var' serves only to define the specification variable var', we may relax the above requirement to

{I and B and var = var'} S? {I}

{I and B and var = var'} S? {var ≤ var'-ε}

which, together, are equivalent to the single design goal for S

{I and B and var = var'} S? {I and var ≤ var'-ε}

The loop body S? can then be decomposed as described above into two parts, one decreasing the value of var and the rest reestablishing the truth of the loop invariant I.

6.3.5 The subprogram call

If the precondition and the postcondition of the program segment to be designed can be put into the form {V and B} and {P and B} respectively, where V and P constitute the specification of another program segment S — i.e. {V} S {P} — and where B refers only to variables not changed by S, then a subprogram call to S is indicated as a solution to the design task. Similarly, if V and P are (by rule P1) a precondition and a postcondition respectively of such a program segment S, a call to S in indicated. Cf. rules SP1, SP2 and SP3.

6.4 Design example: a program segment to sum the elements of an array

Consider the task of designing a program segment "Sum" to calculate the sum of the elements of an array. The interface specification is given as follows:

Correctness proposition 1: {V} Sum {P}, where V and P are as follows:

V: $n \in \mathbf{Z}$ and $0 \leq n$

P: $sum = \sum_{k=1}^{n} x(k)$

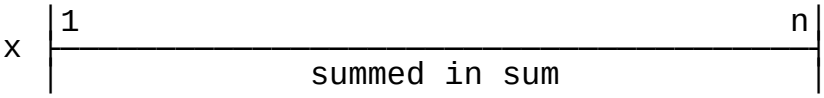
Correctness proposition 2: Sum.d = [(sum, **R**, .)] & d for every data environment d in the domain of Sum.

Note that correctness proposition 2 specifies the structure of the final data environment and requires that the program segment Sum not change any variable in the original data environment.

Because the postcondition contains a series not present in the precondition and because none of the variables in the bounds of the series may be changed by Sum, a loop is indicated as a primary structural element in the program segment to be designed. The hypotheses of rule W2 will, therefore, provide guidelines for the design of the loop. First we must decide upon a loop invariant I by generalizing the final and initial situations (roughly the postcondition and the precondition) in a suitable way. Then we must design the initialization init and the loop condition B so that {V} init {I} and so that I and not B $\Rightarrow$ P. Finally, we must design the body S of the loop so that {I and B} S {I} and so that S ensures progress toward termination. Cf. rule W2.

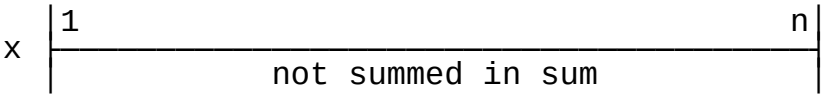
6.4.1 An informal design approach

To decide upon a suitable loop invariant I we begin by examining the postcondition. The postcondition P can be represented in a diagram as follows:



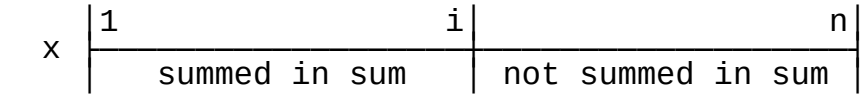
[P]

The corresponding diagram for the precondition V is:



[V]

Two different regions are present in these diagrams. Our loop invariant must contain at least these two regions, e.g.



[I]

Alternatively, the two regions could be interchanged.

From the above diagram we note that when i=0 the diagram for I becomes the diagram for V, the initial situation. When i=n the diagram for I becomes the diagram for P, the postcondition. Thus the value of i must range from 0 to n inclusive. Based on these considerations and the above diagram we write I in algebraic form:

I: $n \in \mathbf{Z}$ and $i \in \mathbf{Z}$ and $0 \le i \le n$ and sum = $\sum_{k=1}^{i} x(k)$

The above observation that the diagram for I becomes the diagram for V when i=0 leads to the decision to initialize i to 0. With i=0 the loop invariant will be true only if sum=0. Thus both i and sum must be set to 0 initially, either by assignment or declare statements. Only declare statements would be consistent with correctness proposition 2, so our initialization is:

declare (sum, **R**, 0); declare (i, **Z**, 0)

The above observation that the diagram for I becomes the diagram for P when i=n leads to the condition i=n as a candidate for not B and, therefore, i≠n as a candidate for the while condition B. Given the truth of I, i≠n is equivalent to the condition i<n. Because the latter is stronger, we select it, although either condition is perfectly suitable.

The body S of the loop must (1) ensure progress toward termination and (2) maintain the truth of the loop invariant I. Progress toward termination will be ensured by increasing the value of i by 1, which reduces the size of the region “not summed in sum” in the loop invariant I. (In the diagram for P, this region is not present, i.e. is empty.) In order to maintain the truth of the loop invariant I, the value of the element of x brought into the left region (i.e. x(i)) must be added to the program variable sum. Thus, a suitable solution for S is the sequence of statements i:=i+1; sum=sum+x(i).

In order to satisfy correctness proposition 2, the newly introduced variable i must be released after execution of the loop terminates. The entire program segment is, then:

```
declare (sum, R, 0); declare (i, Z, 0)
while i<n do
    i:=i+1; sum:=sum+x(i)
endwhile
release i
```

6.4.2 A more formal design approach

We begin by determining a suitable loop invariant I, which must be a generalization of V and P in the sense that (1) some special case of the loop invariant must imply the postcondition P and

(2) I must be easily initializable. We begin with P and ask how it must be modified so that it can be initialized easily. The summation appears in P but not in V, so the summation must be eliminated initially, e.g. by setting its upper limit n to some constant value such as 0 or 1. But correctness proposition 2 does not permit modifying the value of n. This suggests introducing a new variable (e.g. i) to replace n as the upper limit of the summation. (Alternatively, i could replace 1 as the lower limit.) This line of reasoning leads to the expression

$$\text{sum} = \sum_{k=1}^{i} x(k)$$

as the main term in the loop invariant I. A strong condition on the allowed values of the newly introduced variable i should also be incorporated in I, including a lower and an upper bound. If a special case of I is to imply P, then n must be an allowed value of i. Initially, i must be some small constant c (e.g. 0 or 1) so that the summation term can be eliminated. Thus the range of values allowed for i would be $c \le i \le n$. Because the value of n may be as small as 0 (cf. the precondition V), c may not be greater than 0, which is satisfactory as an initial value of i because it reduces the summation to 0. Thus I becomes

$$\text{I:} \quad n \in \mathbf{Z} \text{ and } i \in \mathbf{Z} \text{ and } 0 \le i \le n \text{ and sum} = \sum_{k=1}^{i} x(k)$$

The initialization init must be designed so that {V} init {I}. We ask ourselves, therefore, how I may be transformed into V by steps corresponding to rules and from the transformation steps, we deduce the associated specific program statements. Examining I we see that if i and sum are both 0, I reduces to V, i.e. $V=[I_0^i]_0^{sum}$. The form of this expression immediately suggests rule A1 and, hence, a sequence of two assignment or declare statements. Because of correctness proposition 2, only the declare statements are suitable. Our initialization is, therefore, the sequence

declare (sum, **R**, 0); declare (i, **Z**, 0)

The while condition B must be designed so that I and not B $\Rightarrow$ P or, equivalently, I and not P $\Rightarrow$ B. From the considerations leading to I (see above), we note that I and i=n $\Rightarrow$ P, suggesting $i \ne n$ as B. Any other expression which, given the truth of I, is equal to or follows from $i \ne n$ is also a suitable candidate for B, e.g. $i < n$.

Alternatively, one can derive B more formally by requiring that I and not P $\Rightarrow$ B. A suitable expression for B must be syntactically allowed as a while condition in the target programming language and should not follow from I alone (otherwise the loop would not terminate).

I and not P

=

$$n \in \mathbf{Z} \text{ and } i \in \mathbf{Z} \text{ and } 0 \le i \le n \text{ and sum} = \sum_{k=1}^{i} x(k) \text{ and sum} \ne \sum_{k=1}^{n} x(k)$$

$\Rightarrow$

$$i \le n \text{ and } \sum_{k=1}^{i} x(k) \ne \sum_{k=1}^{n} x(k)$$

$\Rightarrow$

$i \leq n$ and $i \neq n$

=

$i < n$ ■

Finally, we must design the body S of the loop so that (1) S ensures progress toward termination, i.e. toward P or toward not B and (2) {I and B} S {I}. To ensure progress toward termination we design a suitable loop variant function var, see sections 4.7.3 above and 6.3.4.4 above. Because the while condition B is $i<n$ (see above), the expression n-i is an obvious candidate for the loop variant var. Because the values of n and i are integers, the value of var will be an integer, and therefore 1 is an obvious choice for ε. (If the value of var is decreased at all, it must be decreased by at least 1.)

Our general design goal (from the lemma for loop termination in section 4.7.3 above)

{I and B and var = var'} S? {I and var ≤ var'-ε}

then becomes specifically

{I and B and (n-i) = (n-i)'} S? {I and (n-i) ≤ (n-i)'-1}

or, when simplified (n remains unchanged throughout this part of the program, cf. correctness proposition 2, so n=n' at all places),

{I and B and i = i'} S? {I and i ≥ i'+1}

We choose i:=i+1 (the smallest increment to i that can satisfy the above specification) as S or a part thereof and our design task is thereby transformed into

{I and B and i = i'} S1?; i:=i+1 {I and i ≥ i'+1}

or

{I and B and i = i'} S1? $\{I^{i}_{i+1}$ and i ≥ i'}

We must transform $\{I^{i}_{i+1}$ and i ≥ i'} into {I and B and i = i'} or into an expression which follows from {I and B and i = i'} (cf. rule P1) and from the transformation steps deduce program statements for S1. In order to do this, we examine the two expressions in question in detail:

I and B and i = i': $n \in \mathbf{Z}$ and $i \in \mathbf{Z}$ and i = i' and $0 \leq i < n$ and sum = $\sum_{k=1}^{i} x(k)$

I^{i}_{i+1} and i ≥ i': $n \in \mathbf{Z}$ and $i \in \mathbf{Z}$ and i ≥ i' and $-1 \leq i < n$ and sum = $\sum_{k=1}^{i+1} x(k)$

We see that if sum were replaced by sum+x(i+1) in the last expression above, that expression would follow from [I and B and i = i'], i.e.

[I and B and i = i'] ⇒ $[I^{i}_{i+1}$ and i ≥ i']$^{sum}_{sum+x(i+1)}$

Thus a suitable solution for S1 is the assignment statement sum:=sum+x(i+1).

This completes the design of the loop. In order to satisfy correctness proposition 2, the newly introduced variable i must be released. This can be done at the end of the program segment to be designed without affecting the value of the relevant assertion P at that point. Combining the various parts designed above, our entire program segment becomes:

```
declare (sum, R, 0); declare (i, Z, 0)
while i<n do
    sum:=sum+x(i+1); i:=i+1
endwhile
release i
```

The bodies of the loops in this program segment and in the one designed informally in section 6.4.1 above are different but equivalent. I.e. it can be shown that

(sum:=sum+x(i+1); i:=i+1).d = (i:=i+1; sum:=sum+x(i)).d

for all $d \in \mathbb{D}$, cf. the definitions of the assignment statement and of the sequence of statements.

6.5 Design example: partitioning with pointers, deriving a program from its specification

6.5.1 Specification for the subprogram "twopartition"

A subprogram "twopartition" is to be designed so that it satisfies the following specification.

Correctness proposition 1: {V} twopartition {P}, where V and P are as follows:

V: $n \in \mathbf{Z}$ and $0 \leq n$ [range of n]

P: $n \in \mathbf{Z}$ and $gr \in \mathbf{Z}$ and $0 \leq gr \leq n$ [range of n and gr]

$\text{and}_{i=1}^{gr}$ prop(X(Ptr(i))) $\text{and}_{i=gr+1}^{n}$ not prop(X(Ptr(i))) [X partitioned by property prop]

and ($\&_{i=1}^{n}$ [Ptr(i)]) Perm ($\&_{i=1}^{n}$ [i])

[Ptr is a permutation of the integers 1 through n inclusive]

and where prop(.) is a function available for use in expressions in the subprogram to be designed. This function maps the value of an element of array X to either true or false, depending upon whether or not that value satisfies a certain (but here unspecified) property.

Correctness proposition 2: For all data environments d in the domain of twopartition, twopartition.d = [(gr, **Z**, ...)] & d except for the values of gr and of the array elements Ptr(i) for i = 1, ... n.

Correctness proposition 3: The execution of the subprogram twopartition terminates.

Note that this specification is somewhat incomplete because the domain of twopartition is referred to above but is not specified. This gap will be filled in Chapter 7 below by introducing strict and semistrict preconditions and associated proof rules.

6.5.2 Overview of the design steps

Part 2 of the specification above indicates that Ptr(1), ... Ptr(n) are the only variables in the original data environment d whose values may be changed by the program to be designed. The variable gr must be newly declared. I.e., gr and Ptr(1), ... Ptr(n) are variables whose values can (and presumably should) be calculated by the program to be designed. The only program statements capable of doing this are the assignment and the declare statements, to which rules A1, A2, D1 and D2 apply. These rules call for replacing a variable name by an expression. Thus, when designing twopartition, we may replace a reference to the value of any of the variables gr, Ptr(1), ... Ptr(n) by any suitable expression if such replacement helps us to transform a postcondition into a given precondition or into a condition implied by a given precondition.

The postcondition and the precondition differ in that only the postcondition (1) refers to the variable gr which, by the second part of the specification, must be declared and calculated and (2) refers to elements of the array X and their associated values of the function prop. The truth of the two and-series in the second line of the postcondition cannot both be established with a fixed number of operations. Therefore a loop is indicated. The design problem then consists of determining a loop invariant, an initialization for the loop, the loop condition, the body of the loop and any other parts of the subprogram needed to satisfy the specification, in particular, the second part (the specification of the structure of the final data environment).

6.5.3 The loop invariant

In the loop invariant it must be possible to reduce the number of terms in each of the and-series corresponding to the second line of P to a fixed number, e.g. to make them empty. This, in turn, requires introducing a new variable (e.g. j) in order to create a gap between the bounds of the two and-series. A similar separation of the first &-series (which refers to the same elements of Ptr) seems appropriate. The number of terms in the second &-series must be adjusted accordingly. These considerations lead to the following loop invariant:

I: $n \in \mathbf{Z}$ and $gr \in \mathbf{Z}$ and $j \in \mathbf{Z}$ and $0 \le gr \le j \le n$ [range of n, gr and j]

$\text{and}_{i=1}^{gr}$ prop(X(Ptr(i))) $\text{and}_{i=j+1}^{n}$ not prop(X(Ptr(i))) [X partitioned by property prop]

and $(\&_{i=1}^{gr}$ [Ptr(i)] $\&_{i=j+1}^{n}$ [Ptr(i)]) Perm $(\&_{i=1}^{n+gr-j}$ [i])

[Ptr is a permutation of the integers 1 through n+gr-j inclusive]

6.5.4 The initialization of the loop

By comparing the expressions for V and I above, it is clear that V=I if gr=0 and j=n. I.e.

$$V = [I_0^{gr}]_n^j$$

which suggests two corresponding assignment or declare statements for the initialization. Because of the second part of the specification, we may not write assignment statements for this purpose, so the initialization becomes

initialization: declare (j, **Z**, n); declare (gr, **Z**, 0)

6.5.5 The loop condition

The considerations leading to the loop invariant (see above) lead to the observation that if gr=j, then I implies P. I.e.

I and gr=j $\Rightarrow$ P

This can be verified easily by examining I and P.

This observation leads to gr≠j as one possible loop condition. This is equivalent to gr<j whenever I is true, so gr<j is another possible loop condition. Since the latter is a stronger condition, we select it as our while condition, called B below:

B: gr<j

6.5.6 The body of the loop

The body of the loop must (1) make progress toward termination, i.e. must make progress toward falsifying the while condition B above, and (2) preserve the truth of the loop invariant I.

Either increasing gr or decreasing j will make progress toward termination. An increment or decrement of one will suffice. Furthermore, in view of the fact that every element of the array X must be examined to determine whether its corresponding value of prop is true or false, an increment or decrement of one seems necessary. This reasoning leads to the following two alternative loop bodies, which are shown below with their preconditions and postconditions:

{I and B} S1?; gr:=gr+1 {I} [S1 still to be designed]

{I and B} S2?; j:=j-1 {I} [S2 still to be designed]

Recalling that one can strengthen the postcondition of the body of a loop when a loop variable is increased or decreased (see section 5.2 above), we strengthen the postconditions in the above design tasks and obtain

(1) {I and B} S1?; gr:=gr+1 {I and 1≤gr} [S1 still to be designed]

(2) {I and B} S2?; j:=j-1 {I and j≤n-1} [S2 still to be designed]

Applying rules S1 and A1 to the above design tasks, we transform them into the design tasks

(3) $\{I \text{ and } B\}\ S1?\ \{[I \text{ and } 1 \le gr]^{gr}_{gr+1}\}$ [S1 still to be designed]

(4) $\{I \text{ and } B\}\ S2?\ \{[I \text{ and } j \le n-1]^{j}_{j-1}\}$ [S2 still to be designed]

Expressed more precisely, rules S1 and A1 ensure that if the correctness proposition (3) above is true, then correctness proposition (1) above is true. I.e., any S1 satisfying correctness proposition (3) above will also satisfy correctness proposition (1) above. Similarly, any S2 satisfying correctness proposition (4) above will satisfy correctness proposition (2) above.

We must, then, find either an S1 satisfying design task (3) above or an S2 satisfying design task (4) above.

In order to proceed further, we examine the three pre- and postconditions

I and B

$[\text{I and } 1 \le gr]_{gr+1}^{gr}$

$[\text{I and } j \le n-1]_{j-1}^{j}$

in design tasks (3) and (4) in detail.

The precondition in both design tasks (3) and (4) is:

I and B

=

$n \in \mathbf{Z}$ and $gr \in \mathbf{Z}$ and $j \in \mathbf{Z}$ and $0 \le gr < j \le n$ [note < instead of ≤ in I]

$\text{and}_{i=1}^{gr}$ prop(X(Ptr(i))) $\text{and}_{i=j+1}^{n}$ not prop(X(Ptr(i)))

and $(\&_{i=1}^{gr}$ [Ptr(i)] $\&_{i=j+1}^{n}$ [Ptr(i)]) Perm $(\&_{i=1}^{n+gr-j}$ [i])

If we expand and manipulate the postcondition in design task (3) with the goal of expressing it in a form as similar to the form of the precondition (I and B) as possible, we obtain:

$[\text{I and } 1 \le gr]_{gr+1}^{gr}$

=

$n \in \mathbf{Z}$ and $gr \in \mathbf{Z}$ and $j \in \mathbf{Z}$ and $0 \le gr < j \le n$

$\text{and}_{i=1}^{gr}$ prop(X(Ptr(i))) $\text{and}_{i=j+1}^{n}$ not prop(X(Ptr(i)))

and prop(X(Ptr(gr+1)))

and $(\&_{i=1}^{gr}$ [Ptr(i)] & [Ptr(gr+1)] $\&_{i=j+1}^{n}$ [Ptr(i)]) Perm $(\&_{i=1}^{n+gr-j}$ [i] & [n+gr+1-j])

Notice how strengthening the postcondition of the body of the loop has simplified the algebraic manipulation. That strengthening guaranteed that both series from which a term has been removed above were not empty. Also, that strengthening resulted in the string of inequalities in the first line of the expression above being in exactly the same form as in the expression for I and B.

The dotted lines above mark those terms not present in the precondition I and B. If the two terms enclosed in dotted lines in the last line above were equal, they could be dropped by the axiom for the function Perm. The resulting expression would be the same as I and B except for the additional condition "prop(X(Ptr(gr+1)))". This suggests replacing Ptr(gr+1) — one of the variables whose values twopartition may change — by n+gr+1-j in the above expression. Then we obtain

$$\text{I and B and prop}(X(n+gr+1-j)) = [[\text{I and } 1 \le gr]_{gr+1}^{gr}]_{n+gr+1-j}^{Ptr(gr+1)}$$

The corresponding candidate for the body of the loop, the sequence (Ptr(gr+1):=n+gr+1-j; gr:=gr+1), has a stronger precondition than specified (I and B). This suggests embedding this sequence of assignment statements in an if statement with the condition "prop(X(n+gr+1-j))":

if prop(X(n+gr+1-j)) then Ptr(gr+1):=n+gr+1-j; gr:=gr+1 else S3? endif

We now need a program segment S3 satisfying the requirement that

{I and B and not prop(X(n+gr+1-j))} S3? {I}

and which makes progress toward termination, e.g. by reducing j instead of increasing gr.

This suggests pursuing the alternative design task (4) above. We expand and manipulate its postcondition with the goal of expressing it in a form as similar to the form of the precondition (I and B) as possible, obtaining:

$$[\text{I and } j \le n-1]_{j-1}^{j}$$

=

$$n \in \mathbf{Z} \text{ and } gr \in \mathbf{Z} \text{ and } j \in \mathbf{Z} \text{ and } 0 \le gr < j \le n$$

$$\text{and}_{i=1}^{gr} \text{ prop}(X(Ptr(i))) \text{ and}_{i=j+1}^{n} \text{ not prop}(X(Ptr(i)))$$

$$\boxed{\text{and not prop}(X(Ptr(j)))}$$

$$\text{and } (\&_{i=1}^{gr} [Ptr(i)] \boxed{\&\ [Ptr(j)]} \&_{i=j+1}^{n} [Ptr(i)]) \text{ Perm } (\&_{i=1}^{n+gr-j} [i] \boxed{\&\ [n+gr+1-j]})$$

The comments above regarding design task (3) and its postcondition apply correspondingly here. If we replace Ptr(j) by n+gr+1-j in the above expression, we obtain the result that

I and B and not prop(X(n+gr+1-j)) = $[[\text{I and } j \le n-1]^{j}_{j-1}]^{Ptr(j)}_{n+gr+1-j}$

Thus, we have

{I and B and not prop(X(n+gr+1-j))} Ptr(j):=n+gr+1-j; j:=j-1 {I and j≤n-1}

and, by rule P1,

{I and B and not prop(X(n+gr+1-j))} Ptr(j):=n+gr+1-j; j:=j-1 {I}

Therefore, the sequence (Ptr(j):=n+gr+1-j; j:=j-1) satisfies the requirements for the else part of the if statement identified above.

Notice that with the exception of the expressions gr+1 and j-1 in the statements incrementing gr and decrementing j all expressions in the body of the loop were derived here by algebraic manipulation of the expressions for the precondition and postcondition. It was not necessary to interpret subjectively any of these expressions or the function to be performed by the subprogram being designed.

6.5.7 Other required parts of twopartition

In the initialization of the loop the variables gr and j were declared. The second part of the specification requires that gr be declared, but does not permit a new variable j to be present in the final data environment. Therefore, the statement

release j

must be appended to the subprogram designed above.

6.5.8 The complete subprogram twopartition

Putting the above parts together, the complete subprogram twopartition then becomes:

```
declare (j, Z, n); declare (gr, Z, 0)
while gr<j do
      if prop(X(n+gr+1-j))
      then Ptr(gr+1):=n+gr+1-j; gr:=gr+1
      else Ptr(j):=n+gr+1-j; j:=j-1
      endif
endwhile
release j
```

6.5.9 Termination

The statements to increment gr and decrement j were included to ensure termination. See the beginning of section 6.5.6 above (“The body of the loop”).

A formal proof of termination would involve

- deriving the loop variant var (see the lemma for loop termination in section 4.7.3) from the while condition B (developed in section 6.5.5 above), giving var = (j-gr),

- defining ε to be 1 (see section 6.5.6 above) and then
- applying the lemma for loop termination.

7. Rules for strict preconditions

In practice one usually wants to prove that a program segment is totally correct, i.e. that it executes in finite time with a defined result which satisfies the postcondition. Proving partial correctness is not sufficient in such situations. Expressed differently, one must show that the stated precondition is not only a precondition, but also a strict precondition. For these proof tasks we need corresponding rules.

By the definition of a strict precondition, {V} S {P} strictly if {V} S {P} and $V \subseteq S^{-1}.\mathbb{D}$. Therefore, the intersection of any precondition with the domain of the statement in question is a strict precondition with respect to that statement. This provides the basis for formulating the following rules for strict preconditions. (See the definitions of the various types of program statements and their domains in section 2.5 above and the rules for ordinary preconditions in chapter 4 above.) The rules below can be proved formally either by direct application of the definition of a strict precondition or by using the lemma for a strict precondition in section 3.2 above.

7.1 Rules for the assignment statement

7.1.1 Rule AS1

$\{P^{x}_{E}$ and $E \in$ Set."x"$\}$ x:=E {P} completely and strictly ■

7.1.2 Rule AS2

If

$V \Rightarrow P^{x}_{E}$ and $E \in$ Set."x"

then

{V} x:=E {P} strictly ■

7.2 Rules for the declare statement

7.2.1 Rule DS1

$\{P^{x,\ \text{Set."x"}}_{E,\ S}$ and $E \in S\}$ declare (x, S, E) {P} completely and strictly ■

If an array variable is being declared, then rule DS1 is

$\{P^{x(ie),\ \text{Set."x(ie)"}}_{E,\qquad S}$ and $E \in S$ and $ie \in$ **Siv**$\}$

declare (x(ie), S, E) {P} completely and strictly ■

The set **Siv** is the set of permitted index values, see section 2.5.2 above.

7.2.2 Rule DS2

If

$$V \Rightarrow P^{x,\ \text{Set.“x”}}_{E,\ S} \text{ and } E \in S$$

then

{V} declare (x, S, E) {P} strictly ■

If an array variable is being declared, then rule DS2 is as follows.

If

$$V \Rightarrow P^{x(ie),\ \text{Set.“x(ie)”}}_{E,\quad\ S} \text{ and } E \in S \text{ and } ie \in \mathbf{Siv}$$

then

{V} declare (x(ie), S, E) {P} strictly ■

The set **Siv** is the set of permitted index values, see section 2.5.2 above.

7.3 Rules for the release statement

7.3.1 Rule RS1

If

{V} release x {P} and
$V \Rightarrow \text{Set.“x”} \neq \varnothing$

then

{V} release x {P} strictly ■

7.3.2 Rule RS2

If

{V} release x {P}

then

{V and Set.“x”≠∅} release x {P} strictly ■

The special case of rule RS2 in which

- P contains no reference to x and
- V=P

occurs frequently and is therefore of considerable practical importance. Formally, it can be viewed as a

Corollary to rule RS2: If P contains no reference to x, then

{P and Set.“x”≠∅} release x {P} strictly ■

7.4 Rule NS for the null statement

For any postcondition P

{P} null {P} strictly ■

7.5 Rule SS for the sequence of statements

If

{V} S1 {P1} strictly and
{P1} S2 {P} strictly

then

{V} (S1, S2) {P} strictly ■

7.6 Rules for the if statement

7.6.1 Rule IFS1

If

V ⇒ B∈ {false, true} and
{V and B} S1 {P} strictly and
{V and not B} S2 {P} strictly

then

{V} if B then S1 else S2 endif {P} strictly ■

7.6.2 Rule IFS2

The strict version of rule IF2 for deriving a precondition with respect to an if statement may or may not apply depending upon how the Boolean function not is defined on the extended domain {false, true, undef}. The following modified version of rule IF2 avoids this potential problem and is, therefore, generally valid.

If

{V1} S1 {P} strictly and
{V2} S2 {P} strictly

then

{V1 and (B=true) or V2 and (B=false)} if B then S1 else S2 endif {P} strictly ■

7.7 Rules for the while loop

7.7.1 Rule WS1

If

$I \Rightarrow (B \in \{false, true\})$ and
$I \Rightarrow (B \Rightarrow 0<var)$ and
$\{I \text{ and } B \text{ and } var = var'\}\ S\ \{I \text{ and } var \leq var'-\varepsilon\}$ strictly

then

$\{I\}$ while B do S endwhile $\{I \text{ and not } B\}$ strictly ■

See section 4.7.3 above for the loop variant function var and the positive constant ε.

The above formulation of this rule is adequate when one requires that the loop terminate. Sometimes, however, termination is not required. To handle such situations, we define a *semistrict precondition* which does not guarantee that loops terminate but otherwise ensures that each statement in the program segment executes with a defined result. For the formal definition of a semistrict precondition see *The Spine of Software*, especially chapter 2.2, and *Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit*, especially sections 3.1.4, 3.1.5 and 3.2.5. These books are available online in pdf files, see Appendix A below.

If a program segment S contains no loop, then there is no difference between the properties semistrictness and strictness, i.e. a semistrict precondition for S is a strict precondition for S and vice versa. If S contains a loop, then strictness of the precondition is a stronger property than semistrictness, i.e. a strict precondition for S is also a semistrict precondition for S, but a semistrict precondition for S is not necessarily a strict precondition for S.

A semistrict version of rule WS1 is:

Rule WS1 (semistrict version): If

$I \Rightarrow (B \in \{false, true\})$ and
$\{I \text{ and } B\}\ S\ \{I\}$ semistrictly

then

$\{I\}$ while B do S endwhile $\{I \text{ and not } B\}$ semistrictly ■

Rules WS1 give an additional guideline for formulating a suitable loop invariant: The loop invariant I should contain the term $[B \in \{false, true\}]$ or a term or terms which imply it. Such term(s) should be anded with the rest of the loop invariant.

7.7.2 Rule WS2

If

$\{V\}$ init $\{I\}$ strictly and

$I \Rightarrow (B \in \{false, true\})$ and
$I \Rightarrow (B \Rightarrow 0<var)$ and
$\{I \text{ and } B \text{ and } var = var'\}$ S $\{I \text{ and } var \leq var'-\varepsilon\}$ strictly and
$I \text{ and not } B \Rightarrow P$

then

$\{V\}$ init; while B do S endwhile $\{P\}$ strictly ■

When only a semistrict precondition is required, the following version of rule WS2 can be used:

Rule WS2 (semistrict version): If

$\{V\}$ init $\{I\}$ semistrictly and
$I \Rightarrow (B \in \{false, true\})$ and
$\{I \text{ and } B\}$ S $\{I\}$ semistrictly and
$I \text{ and not } B \Rightarrow P$

then

$\{V\}$ init; while B do S endwhile $\{P\}$ semistrictly ■

7.8 Applying the rules for strict and semistrict preconditions

The above rules for strict preconditions imply that one can subdivide a proof of total correctness of a program segment into the following four steps:

(1) Prove the partial correctness of the program segment in question, i.e. prove that $\{V\}$ S $\{P\}$.

(2) Prove that the precondition of every basic statement in the program segment S is a strict precondition. Cf. rules AS, DS, RS and NS above.

(3) Prove that the value of every if and while condition is defined, i.e. that the precondition of each if and while statement implies that the value of the condition in question is either false or true. Cf. rules SS, IFS and WS above.

(4) Prove that every loop terminates, i.e. that the value of every while condition becomes false after a bounded number of executions of the loop body.

Alternatively, one can prove that $\{V\}$ S $\{P\}$ strictly by decomposing the correctness theorem using the strict rules and then verifying the remaining Boolean algebraic implications.

Note that strictness is a stronger property than (and hence implies) semistrictness, i.e. if $\{V\}$ S $\{P\}$ strictly, then $\{V\}$ S $\{P\}$ semistrictly. Every rule for a strict precondition can be used as a rule for a semistrict precondition by substituting “semistrict” for “strict” everywhere in the rule. The reverse does not apply, because $\{V\}$ S $\{P\}$ semistrictly does not imply that $\{V\}$ S $\{P\}$ strictly. In particular, semistrictness does not imply strictness if S contains a loop (or a call to a subprogram containing a loop or a call to a subprogram containing a loop, etc.).

8. Guidelines for specifications, conditions and proofs

8.1 Contents of an interface specification

The specification of a program segment S should normally contain the following information:

(1) A strict precondition and a postcondition, i.e. a correctness proposition of the form

{V} S {P} strictly

or alternatively, if S need not terminate, {V} S {P} semistrictly.

(2) A statement relating the structure of the final data environment and the structure of the initial data environment, e.g. a statement of the form

S.d=[(x, M, E), ...]&d structurally

for all data environments d in the precondition V.

(3) A list of all variables whose values may be changed by the execution of S.

Often items (2) and (3) above can be combined into a statement of the form

S.d=[(x, M, E), ...]&d except for the values of the variables x, ...

for all data environments d in the precondition V.

See also section 6.2 above.

8.2 Specifying a non-terminating program

To specify that a program S should continue to execute forever, i.e. should *never* terminate, the correctness proposition should have the form

{V} S {false} semistrictly

In such cases the desired behaviour of the program will normally be specified in the loop invariant.

8.3 References to sets in preconditions and postconditions

When referring to sets in preconditions and postconditions one must distinguish between

- a set of which the value of a variable is an element and
- a set associated with a program variable.

These sets need not be the same. Depending upon the usage of the program variable, one or the other type of set will be most appropriate in references to sets in conditions.

Consider for example a program variable x which is referenced in a program segment S but not modified by it. (The variable x may be thought of as an *input* variable.) The value of the variable x will, in general, be of significance, but the set associated with the program variable x in the data environment will not itself normally be of interest or concern. Thus a reference to a set in connection with the variable x in a precondition would typically be of the form $x \in M$. Usually

additional bounds on the value of x would also be desirable if M is an ordered set, e.g. $x \in M$ and $a \leq x \leq b$.

Consider a program variable y whose value is modified by the program segment S but which is not declared by S. (The variable y may be thought of as an externally declared *output* variable.) In such a case a strict precondition must ensure that any value which S may assign to y is an element of the set associated with the variable y in the initial data environment. Thus a reference to a set in connection with the variable y in a strict precondition will typically be of the form $M \subseteq$Set.“y” where M is the set of values which S may assign to y.

If a program variable z is an output variable to be declared by the program segment S (i.e. an internally declared output variable), no reference to that variable in the precondition is normally indicated.

If a program variable is both an input and an output variable then both types of comments above apply. In such a situation the precondition may, for example, contain terms of the form $y \in M1$ and $a \leq y \leq b$ and $M2 \subseteq$Set.“y”. If one of these terms implies another, then the weaker term(s) may be dropped, but this will not generally be the case.

If the variable x is an input variable only (i.e. not an output variable) it need not be mentioned for its own sake in the postcondition. Only if it is part of an expression whose main subject is an output variable will such an input variable appear in the postcondition, e.g. as or in a bound on the value of an output variable.

An output variable should be mentioned in the postcondition. The set of which the calculated value is an element and, when possible, bounds on that value should be stated in the postcondition, e.g. $y \in M$ and $a \leq y \leq b$. In the case of an internally declared output variable, the set associated with the variable should also be mentioned in the postcondition unless this information is contained in a separate statement about the structure of the final data environment, see section 8.1 above, point (2). If present in the postcondition, such a reference to the set associated with the internally declared output variable z will typically be of the form $M1 \subseteq$Set.“z”, Set.“z”$\subseteq M2$, $M1 \subseteq$Set.“z”$\subseteq M2$ or M3= Set.“z”.

A variable which is neither an input nor an output variable, i.e. a variable whose name appears nowhere in the program segment S, should not be mentioned in either the precondition or the postcondition comprising the specification of S.

These guidelines for referencing sets in preconditions and postconditions can be summarized in the following table:

	Precondition	Postcondition
input variable x	$x \in M$ and $a \le x \le b$	—
externally declared output variable y	$M \subseteq$ Set.“y”	$y \in M$ and $a \le y \le b$
internally declared output variable z	—	$z \in M$ and $a \le z \le b$ (optionally also $M1 \subseteq$ Set.“z” $\subseteq M2$ etc.)

When a variable falls into more than one of the above categories, combine the terms shown. I.e., if the variable w is both an input variable and an externally declared output variable, then the precondition should contain terms of the form $w \in M1$ and $a \le w \le b$ and $M2 \subseteq$ Set.“w” and the postcondition should contain terms of the form $w \in M3$ and $c \le w \le d$.

8.4 Guidelines for formulating a loop invariant

In form and content a loop invariant will often be very similar to the corresponding postcondition. In order to prove that {I and B} S {I} strictly (cf. rules WS1 and WS2 in section 7.7 above) it will often be necessary to include in the loop invariant one or more terms from the precondition, in particular terms referring to variables referenced or modified by the body S of the loop.

If a new variable v is introduced into the loop invariant, terms corresponding to an input and an externally declared output variable should normally be introduced into the loop invariant, i.e. terms of the form $v \in M1$ and $vmin \le v \le vmax$ and M2=Set.“v”. Here the equality can be used (instead of $\subseteq$) for the relationship between M2 and Set.“v” because v will normally be declared in the initialization of the loop and, therefore, both Set.“v” and the loop invariant are determined by the same person and as part of the same design task (designing the loop with its initialization).

8.5 Formulating a postcondition for a given precondition

Normally one does not derive a postcondition for a given precondition and a given program segment. While this is never necessary for normal design or verification tasks, it is occasionally useful to do so, e.g. when designing the latter part of a sequence of statements after the first part is known.

Such a situation arises, for example, when one is designing the body of a loop and decides to increment or decrement the value of a loop variable to ensure progress toward termination. If, for whatever reason, it is preferable to place the corresponding assignment statement at the beginning, rather than at the end, of the loop body, a design task of the following form remains:

```
{I and B}
k:=k+1
S ?
{I}
```

If a suitable condition C were known such that {I and B} k:=k+1 {C}, then this design task would become {C} S ? {I}.

Such a suitable postcondition C for a given precondition (I and B above) and the assignment statement k:=k+1 follows directly from the observation that $[C^{k}_{k-1}]^{k}_{k+1} = C$ for any condition C. After replacing every occurrence of k by (k-1) in C, the resulting expression C^{k}_{k-1} has the characteristic that every occurrence of k in it is in the subexpression (k-1). If in this expression every occurrence of k is replaced by k+1, each subexpression (k-1) is transformed into ((k+1)-1) which is equal to k. Thus the two substitutions transform C back into itself and we have

$$[C^{k}_{k-1}]^{k}_{k+1} = C$$

From this it follows by rule A1 that {C} k:=k+1 $\{C^{k}_{k-1}\}$ completely.

Referring to the design example above, this observation suggests that one design S by first noting that

$$\{I \text{ and } B\}\ k:=k+1\ \{[I \text{ and } B]^{k}_{k-1}\} \text{ completely}$$

and then designing S so that

$$\{[I \text{ and } B]^{k}_{k-1}\}\ S\ \{I\}$$

More generally, applying this technique to a precondition C and the assignment statement x:=f(x), where f is some function of x (and possibly other program variables), involves finding a function g(x) such that g(f(x)) = x for all x, from which it follows that

$$[C^{x}_{g(x)}]^{x}_{f(x)} = C$$

and, therefore,

$$\{C\}\ x:=f(x)\ \{C^{x}_{g(x)}\} \text{ completely}$$

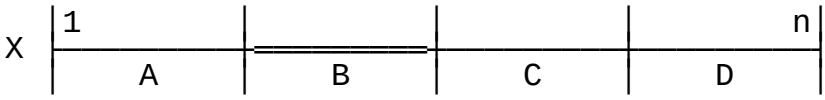
The function g is the inverse of the function f, so the above approach can be applied only when the function f has an inverse. This will typically be the case with assignment statements in which a loop variable is incremented or decremented in a loop body.

Note that the justification for the observation that $[C^{k}_{k-1}]^{k}_{k+1} = C$ depends upon the functions + and - being associative. Thus this technique should be applied with caution (if at all) when the symbols + and - refer to floating point arithmetic operations. Appropriate caution should be exercised when applying it to integer arithmetic operations on a bounded set of integers.

8.6 Variables marking boundaries of regions in arrays

When formulating preconditions, postconditions, intermediate conditions and loop invariants, one frequently defines various regions in arrays. The boundaries of these regions must be marked by program variables. The question then arises whether to indicate the left side, the right side, the insides, the outsides, the same sides or different sides of these boundaries by the corresponding variables. While logically these design decisions are of no consequence (provided, of course, that once made these decisions are followed consequently throughout the analysis and proof of correctness), they can lead to simpler or more complicated expressions to be manipulated, read, interpreted and understood. Following certain conventions typically leads to clearer, more readable and more easily manipulated expressions.

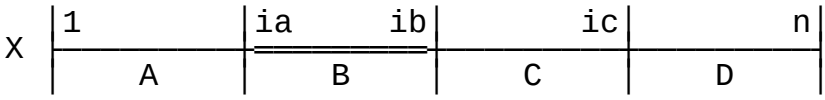
Consider, for example, the following regions of an array X about which statements will be made in the condition I (which might be a loop invariant):



[I]

In this diagram, the double line in region B indicates that that region may not be empty (i.e. must contain at least one element). The single line through the other regions indicates that any of those regions may be empty.

If the boundary variables on each side of a possibly empty region mark the same side (i.e. both indicate the right side or both indicate the left side) and the boundary variables on each side of a non-empty region mark the insides of the region, the inequalities can be written in a particularly simple form. For example, for the diagram



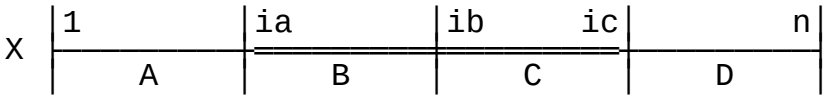
[I]

one would write the inequalities

$$1 \le ia \le ib \le ic \le n$$

Such a single sequence of inequalities is usually more readable and understandable than a conjunction of several separate terms. In particular, the relationship between non-adjacent boundary variables (e.g. ia and n) is immediately apparent (e.g. $ia \le n$).

If the above conventions cannot be followed throughout the diagram, the guideline for the possibly empty regions should usually take precedence. One can then mark the two boundaries of a non-empty region on the same sides. For example, for the diagram



[I]

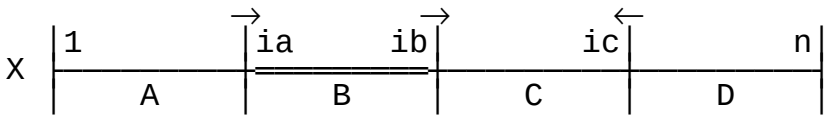
one would write the inequality

$$1 \le ia < ib \le ic \le n$$

In summary, we deduce the following guidelines:

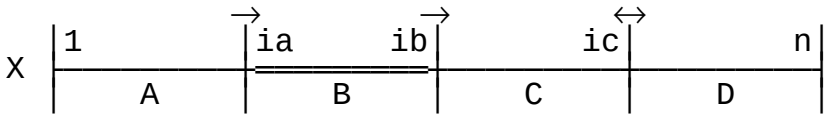
- In the case of a possibly empty region the boundary variables should mark either the same sides or the outsides of the region. The boundary variables will then be related by ≤ or < respectively.
- In the case of a non-empty region the boundary variables should mark either the insides or the same sides of the region. The boundary variables will then be related by ≤ or < respectively.
- Expressed differently but equivalently, avoid marking the insides of a possibly empty region and the outsides of a non-empty region.
- Write all inequalities between boundary variables in a single sequence in which only the names of the boundary variables (not expressions) and the relationships ≤ and < appear (as in the examples above).

Often certain boundary variables will only be increased (or only decreased). In such cases it is sometimes helpful to draw a corresponding arrow near the boundary in question, e.g.



[I]

If a certain boundary may be moved in either direction, one can indicate this possibility by a double arrow:



[I]

8.7 Presenting a proof of correctness

The primary purpose of the documentation of a correctness proof is to enable an independent reviewer to verify its validity. The documented proof should be organized so that the reviewer can quickly see and understand the structure of the proof and so that s/he can select and immediately examine any single part of the proof, skipping over those sections whose validity is clear. It is not necessary (and in general is not desirable) to explain to the reader how the proof was discovered (in particular, how the various intermediate conditions were determined).

As with any document, it should be clear, concise and complete, both in terms of organization and content.

8.7.1 Stating the theorems to be proved

As in the case of any mathematical proof, the presentation should begin with a concise and precise statement of the theorem or theorems to be proved. As described in section 8.1 above, these will normally take the forms

1. {V} S {P} strictly and
2. S.d = [(x, M, .), ...] & d, except for the values of the variables ..., for all d in V

where V, P, etc. are, of course, specifically defined.

8.7.2 Presenting the proof

Theorems of the second form in section 8.7.1 above can usually be verified by inspection, noting which variable names appear on the left side of assignment statements (and as the subject of declare statements) and applying the definitions of the declare and release statements directly. Assignment statements to array variables usually necessitate the additional step of verifying that the value of each index expression lies within the specified range; this normally involves some algebraic analysis.

Proofs of theorems of the first form in section 8.7.1 above are typically more extensive and involve firstly decomposing the theorem into a number of Boolean algebraic expressions (implications) and secondly verifying the universal truth of those implications. In the interests of clarity these two parts of the proof should be separated rigorously and totally, without exception. Mixing them almost always leads to a disorganized, confused proof which is extremely difficult to follow and verify and which appears to be much more complicated than it really is.

8.7.2.1 Proof schemes

In presenting a proof of a theorem of the form {V} S {P} strictly one should normally begin with a proof scheme consisting of the entire program segment S written out in full detail and annotated with the precondition, postcondition, all loop invariants and selected intermediate conditions. These conditions should be stated without justification (e.g. conditions derived by applying rules A1 and IF2 should be simply stated without showing how they were derived). Those conditions which are long should be abbreviated and the abbreviations defined separately; when in doubt, abbreviate. Abbreviations together with superscripts and subscripts should be used where possible and appropriate, e.g. I^{k}_{k+1}, in order to avoid introducing new abbreviations unnecessarily.

The proof scheme gives an overview of the proof. It reflects the structure of the proof and facilitates writing the Boolean algebraic expressions (implications) into which the application of the rules decompose the overall theorem to be proved. In a completely separate part of the documentation of the proof the definitions of the abbreviations are applied and the universal truth of the Boolean algebraic expressions (implications) is verified.

When first teaching and learning how to decompose a correctness proposition into the various Boolean algebraic expressions (implications), it is didactically very advantageous to draw a hierarchical diagram corresponding to the decomposition process. Later in practice, however, such diagrams are not generally recommended because (as a result of their two dimensional nature) they quickly become so large that they do not fit onto one page and therefore no longer convey the clear overview intended and required. The proof scheme outlined above and illustrated below presents the intended overview in a linear manner better suited to the restrictions of a textually based and oriented document and to the needs of a reviewer.

The proof scheme for the subprogram proved correct in section 5.1 above would, for example, be as follows. Note how the strengthening or weakening of a condition (here the postcondition of the body of the loop) is indicated.

{V}
k:=1
{I}
while k≤n and A(k)≠x do
{I and k≤n and A(k)≠x}
    k:=k+1
{I and 2≤k}
{I}
endwhile
{I and not(k≤n and A(k)≠x)}
found:=(k≤n)
{P}

where V, P and I are defined as follows:

V: $n \in \mathbf{Z}$ and $0 \le n$ [empty array allowed]

P: $n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n+1$ and found $\in \mathbb{B}$ [ranges of values of variables]
and $\bigwedge_{i=1}^{k-1} A(i) \ne x$ [all A(i) before the kth ≠x]
and (k≤n and A(k)=x and found [x found]
or k=n+1 and not found) [x not present in A]

I: $n \in \mathbf{Z}$ and $k \in \mathbf{Z}$ and $1 \le k \le n+1$ and $\bigwedge_{i=1}^{k-1} A(i) \ne x$

In the example above intermediate conditions are shown at each point in the program segment. This is not generally necessary or even desirable, e.g. within a sequence of assignment statements to ordinary variables. As a general rule, conditions should appear in a proof scheme in the following places:

- at the beginning (the precondition)
- at the end (the postcondition)
- immediately before a loop (the loop invariant)
- immediately after a loop (the loop invariant and the negation of the loop condition)
- at the beginning of the body of a loop (the loop invariant and the loop condition)
- at the end of the body of a loop
- before and after an if statement
- at the beginning and at the end of the then part of an if statement *only if* these conditions differ from the obvious ones (i.e. the precondition of the if statement anded with the if condition and the postcondition of the if statement respectively)
- at the beginning and at the end of the else part of an if statement *only if* these conditions differ from the obvious ones (i.e. the precondition of the if statement anded with the negation of the if condition and the postcondition of the if statement respectively)
- before a call to a subprogram (organize this precondition to separate the invariant and the variant terms, cf. B and V in rules SP1, SP2 and SP3)

- after a call to a subprogram (organize this postcondition to separate the invariant and the variant terms, cf. B and P in rules SP1, SP2 and SP3)
- between statements in a sequence when strengthening or weakening a condition there, cf. rule P1 (both the stronger and the weaker conditions should be shown)
- before an assignment to an array variable or a declaration of an array variable
- after a release statement, especially when the condition does not refer to the variable being released

8.7.2.2 The detailed proof

Following the proof scheme the proof should be presented as shown in section 5.3.2 above or in a similar way. Note that first all decomposition steps are completed. Only then is the universal truth of the Boolean algebraic expressions (implications) verified. As shown in section 5.3.2 above a list of the remaining proof steps to be performed should appear after each proof step. It is useful to distinguish between decompositional and algebraic steps still to be performed, e.g. by denoting algebraic verification steps with an asterisk (*) as done in section 5.3.2 above.

9. Rules for preconditions and postconditions referring to hidden variables

Some methods for handling preconditions and postconditions for declare and release statements were presented in section 4.10.1 above. While these are adequate for most manual verification tasks, it is sometimes desirable to have general rules relating preconditions and postconditions for individual declare and release statements that permit the preconditions and postconditions to refer to the variable being declared or released and to other variables of the same name. In particular, the structure of an automated verification system would be simpler using such rules. In this section, such rules are developed.

9.1 Notation for referring to hidden variables

Consider the following proof task of deriving a precondition

{V?} release x {P}

where the postcondition P refers to the value of the program variable x. The reference in P to the variable x cannot be a reference to the variable x in the data environment before the release statement is executed, because that variable is released by the statement and therefore no longer exists in the data environment to which the postcondition P applies. Instead, the reference in P to the variable x is a reference to the second variable x (the first hidden variable x) in the data environment before execution of the release statement. I.e., the desired precondition V must contain a reference to a hidden variable, so we need a notation for referring to hidden variables.

We therefore introduce the notational form x:n, where n is a non-negative integer, to refer to the nth hidden variable named x in the data environment in question. More formally, a reference to x:n in a data environment d is defined to be a reference to the variable x in the data environment $(\text{release x})^{n}$.d. Note that x:0 and x are synonyms.

Example: Consider the effect of executing the statement release x on the data environment d0, where

d0 = [(x, **Z1**, 10), (x, **Z2**, 20), (y, **Z**, 3), (x, **Z3**, 30)]

The result d1 = (release x).d0, is, of course,

d1 = [(x, **Z2**, 20), (y, **Z**, 3), (x, **Z3**, 30)]

Notice that the first hidden variable x in d0 (with the value 20) becomes the active variable x in d1. Any reference to the variable x in d1 is, in effect, a reference to the first hidden variable x in d0.

If the postcondition P is, for example, (Set.“x” = **Z2** and x + y = 23), and we wish to derive a precondition V of P with respect to the statement release x, we must replace the reference to x in P by a reference to x:1. Note that all references to x – both references to the value of x and references to the set associated with x – must be replaced. The execution of release x has no effect on variables with other names, e.g. y in this example, so such references remain unchanged. The desired precondition V becomes, then, { Set.“x:1” = **Z2** and x:1 + y = 23}. ■

9.2 Rules for the declare statement

The execution of a declare statement prefixes a new variable to the data environment, hiding existing variables of the same name as the variable being declared one additional level. I.e., each variable with the name in question is hidden one level more in the postcondition than in the precondition. Viewed the other way, each variable with the name in question is hidden one level less in the precondition than in the postcondition. Therefore, rules for the declare statement can be formulated as follows:

9.2.1 Rule DHS1

$$\{P^{x,\ \text{Set.“x”},\ x{:}1,\ \text{Set.“x:1”},\ x{:}2,\ \text{Set.“x:2”},\ \ldots}_{E,\ S,\ \quad x,\ \ \text{Set.“x”},\ \ x{:}1,\ \text{Set.“x:1”},\ \ldots} \text{ and } E \in S\}$$

declare (x, S, E) {P} completely and strictly ■

If only an ordinary precondition is needed, simply drop the “and E∈ S” term above. As a memory aid, note that one **decreases** the hiding levels when forming the precondition for a **declare** statement (“d-d”).

As with previously presented rules involving multiple replacement, all replacements indicated above are to be made simultaneously, not one after the other.

Note that the hiding levels above (e.g. E in x:E) must be integer constants, i.e. either integer numbers or integer expressions whose values are not changed by the execution of the declare statement. In situations in which executing the declare statement can change the value of a hiding level expression (i.e. if a hiding level expression contains a reference to the variable being declared), caution must be exercised to apply the above rule correctly. Such situations are very unusual and will rarely, if ever, be of practical significance.

If an array variable is being declared, then rule DHS1 is:

$$\{P^{x(ie),\ \text{Set.“x(ie)”},\ x(ie){:}1,\ \text{Set.“x(ie):1”},\ x(ie){:}2,\ \text{Set.“x(ie):2”},\ \ldots}_{E,\ \ S,\ \quad x(ie),\ \ \text{Set.“x(ie)”},\ \ x(ie){:}1,\ \text{Set.“x(ie):1”},\ \ldots} \text{ and } E \in S \text{ and } ie \in \mathbf{Siv}\}$$

declare (x(ie), S, E) {P} completely and strictly ■

The set **Siv** is the set of permitted index values, see section 2.5.2 above.

9.2.2 Rule DHS2

If

$$V \Rightarrow P^{x,\ \text{Set.“x”},\ x{:}1,\ \text{Set.“x:1”},\ x{:}2,\ \text{Set.“x:2”},\ \ldots}_{E,\ S,\ \quad x,\ \ \text{Set.“x”},\ \ x{:}1,\ \text{Set.“x:1”},\ \ldots} \text{ and } E \in S$$

then

{V} declare (x, S, E) {P} strictly ■

See also the comments after rule DHS1 in section 9.2.1 above.

If an array variable is being declared, then rule DHS2 is as follows.

If

$$V \Rightarrow P^{\substack{x(ie),\ Set.\text{“}x(ie)\text{”},\ x(ie):1,\ Set.\text{“}x(ie):1\text{”},\ x(ie):2,\ Set.\text{“}x(ie):2\text{”},\ ...\\ E,\quad S,\quad x(ie),\quad Set.\text{“}x(ie)\text{”},\quad x(ie):1,\ Set.\text{“}x(ie):1\text{”},\ ...}}$$

and $E \in S$ and $ie \in$ **Siv**

then

{V} declare (x(ie), S, E) {P} strictly ■

The set **Siv** is the set of permitted index values, see section 2.5.2 above.

9.3 Rules for the release statement

The execution of a release x statement removes the first variable x in the data environment, reducing the hiding level of all other variables named x by one. I.e., each variable with the name x is hidden one level less in the postcondition than in the precondition. Viewed the other way, each variable with the name x is hidden one level more in the precondition than in the postcondition. Therefore, rules for the release statement can be formulated as follows:

9.3.1 Rule RHS1

$$\{P^{\substack{x,\quad Set.\text{“}x\text{”},\quad x:1,\ Set.\text{“}x:1\text{”},\ x:2,\ Set.\text{“}x:2\text{”},\ ...\\ x:1,\ Set.\text{“}x:1\text{”},\ x:2,\ Set.\text{“}x:2\text{”},\ x:3,\ Set.\text{“}x:3\text{”},\ ...}} \text{ and } Set.\text{“}x\text{”} \neq \varnothing\}$$

release x {P} completely and strictly ■

If only an ordinary precondition is needed, simply drop the “and Set.“x”≠∅” term above. As a memory aid, note that one **raises** the hiding levels when forming the precondition for a **release** statement (“r-r”).

As with previously presented rules involving multiple replacement, all replacements indicated above are to be made simultaneously, not one after the other.

Note that the hiding levels above (e.g. E in x:E) must be integer constants, i.e. either integer numbers or integer expressions whose values are not changed by the execution of the release statement. In situations in which executing the release statement can change the value of a hiding level expression (i.e. if a hiding level expression contains a reference to the variable being released), caution must be exercised to apply the above rule correctly. Such situations are very unusual and will rarely, if ever, be of practical significance.

9.3.2 Rule RHS2

If

$$V \Rightarrow P^{\substack{x,\quad Set.\text{“}x\text{”},\quad x:1,\ Set.\text{“}x:1\text{”},\ x:2,\ Set.\text{“}x:2\text{”},\ ...\\ x:1,\ Set.\text{“}x:1\text{”},\ x:2,\ Set.\text{“}x:2\text{”},\ x:3,\ Set.\text{“}x:3\text{”},\ ...}} \text{ and } Set.\text{“}x\text{”} \neq \varnothing\}$$

then

{V} release x {P} strictly ■

See also the comments after rule RHS1 in section 9.3.1 above.

9.4 Proofs of the above rules

The above rules for declare and release statements whose preconditions or postconditions may contain references to hidden variables may be proved formally in ways similar to the proofs of rules A1 and A2 (see section 4.4 above).

9.5 Rules for hidden variables and statements other than declare and release statements

Only declare and release statements can change the structure of data environments. Other statements cannot change the structure of data environments and they cannot change the values of or the sets associated with any hidden variables. I.e., other statements have no effect at all on hidden variables. Therefore, all rules for other types of statements presented in earlier chapters can be applied without change to preconditions and postconditions containing references to hidden variables. Such references to hidden variables need not be changed when applying any rule for any of those other statements.

10. Conclusion

In engineering fields, it is typically understood and known that specific aspects should limit certain performance characteristics such as reliability. In the case of computer systems, it is clear that the hardware will, in principle, be subject to deterioration with age, physical stresses caused by wear and tear, temperature changes, chemical changes (even if only very slow ones) etc. Software does not change and is itself not subject to any form of natural deterioration. It follows, therefore, that the operational reliability of a computer based system should, in principle, be limited by the hardware, not the software. This was the case in the early days of the first vaccuum tube computers (with mean times to malfunction in the order of a day or so), but with the major improvement in reliability with semiconductor circuitry, software has, for many years now, been the limiting factor. This is, in my view, an indication of the relative maturity of the two fields: hardware design and construction is a mature engineering field, but software design is not (yet). The contents of this eTextbook, when properly mastered and applied, can help the software developer to correct this imbalance, so that software will no longer be the factor limiting the reliability of the system of which it is a part.

My own experience developing software over the years and my observations of others' experiences (both reported in the literature and not) have convinced me that testing is a very, very inefficient (i.e. unnecessarily expensive) way of achieving a low error rate in delivered software. These experiences and observations are not isolated cases, but arose in many different situations over many years, both before and after I learned about mathematical verification of computer programs. These experiences and observations have not just permitted me to draw this conclusion, they have forced me to that conclusion — they left no alternative open.

Designing programs based on the needs arising in mathematical verification (i.e. based on the proof rules) is, in my experience, much more efficient. It should, in my view, be the mainstay of achieving a low error rate, with testing as a secondary supportive measure, not the other way around.

Also my experience inspecting and reviewing programs confirms these conclusions. By applying the proof rules, even informally, one can more easily, quickly and systematically identify the presence or absence of errors in a program by comparing the code with, for example, diagrams of a loop invariant. Running even a few test cases will typically take much more time than inspecting the program manually, and with less conclusive results. This leads me to observe that testing is both less efficient and less effective.

Given the current state of software development practice, very extensive testing is clearly necessary. That does not logically lead to the conclusion that very extensive testing must always be the backbone of achieving a low error rate in delivered software. Society is paying a very high price for not seriously questioning the predominant role of testing in attempting (not especially successfully) to achieve software quality in practice today.

The material in this electronic book — in particular, the various proof rules and the material on how to apply them when designing and when verifying a program — can help readers make the transition from developing software subjectively and intuitively to designing software objectively and rationally as engineers in other fields design their artifacts. By doing so, software developers

can achieve error rates in their designs as low as the design error rates achieved by engineers in the traditional engineering disciplines. In turn, such software developers can become software engineers in the true sense of the word “engineer”, and software development of today can undergo a metamorphosis to software engineering in the future.

Appendix A. Bibliography

Abrial, J.-R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, Cambridge, 1996.

* Baber, Robert L., *Software Reflected: The Socially Responsible Programming of Our Computers*, North-Holland, Amsterdam, 1982.

* Baber, Robert L., *The Spine of Software: Designing Provably Correct Software — Theory and Practice*, John Wiley & Sons, Chichester, 1987.

* Baber, Robert L., *Error-free Software: Know-How and Know-Why of Program Correctness*, John Wiley & Sons, Chichester, 1991.

* Baber, Robert L., *Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit*, (in German), Walter de Gruyter, Berlin, 1995.

Backhouse, Roland C., *Program Construction and Verification*, Prentice-Hall International, Englewoods Cliffs, N. J., 1986.

Dijkstra, Edsger W., *A Discipline of Programming*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1976.

Gries, David, *The Science of Programming*, Springer-Verlag, New York, Heidelberg, Berlin, 1981.

Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", *Communications of the ACM*, Vol. 12, No. 10, p. 576-580+583, 1969 Oct.

Hoare, C.A.R. et al., "Laws of Programming", *Communications of the ACM*, Vol. 30, No. 8, p. 672-686, 1987 Aug. and Corrigenda, *Communications of the ACM*, Vol. 30, No. 9, p. 770, 1987 Sept.

Kaldewaij, Anne; *Programming: The Derivation of Algorithms*, Prentice-Hall International, 1990.

Loeckx, Jacques; Sieber, Kurt, *The Foundations of Program Verification*, B. G. Teubner, Stuttgart, und John Wiley & Sons, Chichester, 1984.

Mills, Harlan D., "The New Math of Computer Programming", *Communications of the ACM*, Vol. 18, No. 1, p. 43-48, 1975 Jan.

* The books marked * above are available online in pdf files without charge. See web page http://www.cas.mcmaster.ca/~baber/Books/Books.html for further information on each of these books and to download the files containing them.

An extensive bibliography of other relevant literature can be found in the book *Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit* listed above, p. 201-235.

Appendix B. Exercises

Exercises to accompany this electronic textbook can be found in

- the accompanying file MRSDExer.pdf,
- other files with names beginning with "MRSDExer" in the same directory and
- several books listed in Appendix A above, especially *The Spine of Software* and *Error-free Software*.