Lecture Notes

Axiomatic Semantics and Program Verification

Paul Attie

College of Computer Science
Northeastern University

September 2002

Abstract

These lecture notes provide an introduction to the verification of programs correctness using Hoare logic and weakest preconditions. Chapters 1 and 2 provide some needed backgound on propositional and first-order logic, respectively.

Contents

Chapter 1

The Propositional Calculus

1.1 Propositions

A *proposition* is a statement that can be either *true* or *false*. For example:

> it rains
> I'll stay at home

On the other hand, statements such as:

> open the door
> why were you late?

are not propositions.

Propositions can be either *simple* or *compound*. A simple (or atomic) proposition is a proposition that contains no other proposition as a part. The two propositions given above are simple. A compound proposition is a proposition that is built up from two or more simple propositions. For example, the compound proposition

> **if** it rains **then** I'll stay at home

is built up from the two simple propositions given above using **if**...**then**. Likewise, the compound proposition

> it is Tuesday **and** the sky is blue

is built up from the two simple propositions "it is Tuesday", "the sky is blue" using **and** .

In order to translate such propositions into logical notation, we use *symbols* to represent propositions.

> it rains: ra
> I'll stay at home: st
> it is Tuesday: tu
> the sky is blue: bl

Then, the compound proposition "**if** it rains **then** I'll stay at home" can be represented by:

$$ra \Rightarrow st$$

where $\Rightarrow$ is the symbol for **if** ... **then**. The compound proposition "it is Tuesday **and** the sky is blue" is represented by:

$$tu \wedge bl$$

where $\wedge$ is the symbol for **and** .

Symbols such as ra, st, tu, bl that represent propositions are called *propositional identifiers.* When the context makes it clear, we shall use the abbreviated term *identifiers* instead.

1.2 Logical Connectives

We saw above that compound propositions are formed from simple propositions using extra words such as **if** ... **then** (or, in symbolic form, the symbol $\Rightarrow$). These extra words represent *logical connectives* or *operators.* We shall mainly be concerned with the following five logical connectives (it is possible to define others):

	symbol used in lecture notes	symbol used in text
conjunction	$\wedge$	**and**
disjunction	$\vee$	**or**
negation	$\neg$	**not**
implication	$\Rightarrow$	$\Rightarrow$
double-implication	$\Leftrightarrow$	$\Leftrightarrow$

All of the connectives take two propositions as input, except for negation, which takes one. conjunction represents the informal concept of "and". disjunction represents the informal concept of "inclusive or" (one or the other or both). negation represents the informal concept of "not," i.e., the logical "opposite." implication represents the informal concept of "if ... then." This concept is very important in deducing a *conclusion* logically from a set of assumptions, or *premises.* Finally, double-implication represents the informal concept of logical "sameness."

1.2.1 Truth-tables

The meaning of the logical connectives can be given using *truth-tables.* A truth-table for a logical connective gives the value of a compound proposition formed using the connective in terms of the values of the simple propositions that are the inputs. As we said above, propositions can have two values only: *true* (which will be written as T from now on), and *false* (which will be written as F from now on). T and F are called *truth-values.* The truth-table contains a number of rows, one for each possible combination of values of the inputs.

The meaning of negation is given by the following table:

p	$\neg p$
T	F
F	T

Truth-table for negation

Since negation takes one proposition p as input, this table has two rows, one for each possible value of the input p.

The meaning of conjunction is given by the following table:

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

Truth-table for conjunction

Since conjunction takes two propositions p, q as input, this table has four rows. Each of the inputs p, q has two possible values, and so the number of combinations of values is $2 \times 2 = 4$.

Likewise, the truth-tables for the remaining connectives are as follows:

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Truth-table for disjunction

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Truth-table for implication

p	q	$p \Leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

Truth-table for double-implication

1.3 Syntax of Propositions – Propositional Formulae

A proposition in general is written as a *propositional formula.* In other words, a propositional formula is a particular syntactic way of expressing a proposition. Other ways are conceivable, e.g., parse tree, truth-table, etc. For our purposes, we can regard "proposition" and "propositional formula" as synonyms.

Definition 1 (*Proposition*)
Propositions are formed as follows:

1. T *and* F *are propositions*
2. *A propositional identifier is a proposition*
3. *If* p *is a proposition, then so is* $(\neg p)$
4. *If* p *and* q *are propositions, then so are* $(p \wedge q)$, $(p \vee q)$, $(p \Rightarrow q)$, $(p \Leftrightarrow q)$

You are familiar with arithmetic expressions. We can make an analogy between propositions and arithmetic expressions as follows:

1. Any integer constant is an arithmetic expression (e.g., 5, 100)
2. An integer variable is an arithmetic expression
3. If x and y are arithmetic expressions, then so are $(x+y)$, $(x \times y)$, $(x-y)$, (x/y)

Example 1 If p, q, r are propositions, then so is $((p \wedge q) \vee r)$. Figure 1.1 depicts a parse tree for $((p \wedge q) \vee r)$, showing how it is built up from p, q, r and $(p \wedge q)$. These are called *subpropositions* of $((p \wedge q) \vee r)$.

Example 2 If p, q are propositions, then so is $((\neg p) \vee q)$.

1.4 Evaluation of Propositions

1.4.1 Evaluation of Constant Propositions

A *constant proposition* is a proposition that does not contain any identifiers. In other words, constant propositions are composed entirely of the truth values T, F and the logical connectives. You evaluate a constant proposition by executing the following steps:

1. The value of T is just T, and the value of F is just F.

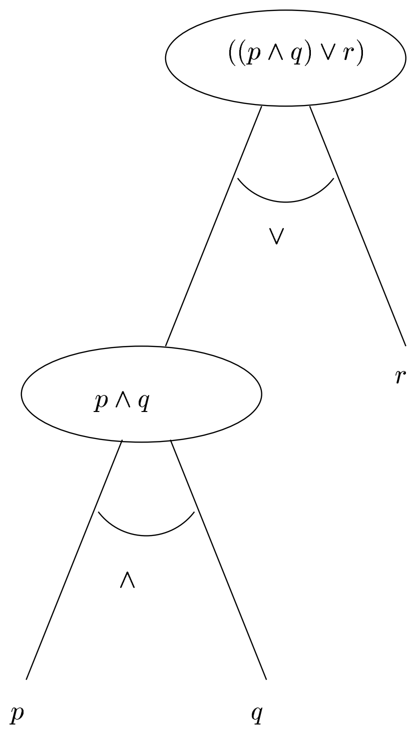


Figure 1.1: Parse tree for the proposition of example 1

2. Evaluate a constant proposition containing exactly one connective by using the truth-tables given in subsection 1.2.1.

3. Evaluate a constant proposition containing n connectives (for any $n > 1$) "inductively" as follows:

   (a) Find all the subpropositions that contain exactly one connective and evaluate them using step 2. Replace each subpropositions by the value obtained for it.

   (b) Repeat step 3a until you are left with either T or F.

Example 3 The proposition $((\mathrm{T} \wedge \mathrm{F}) \vee \mathrm{F})$ is evaluated as follows. First, the subproposition $(\mathrm{T} \wedge \mathrm{F})$ is evaluated using the truth table for conjunction (page 7). The result is F. Replacing $(\mathrm{T} \wedge \mathrm{F})$ by F, we obtain $(\mathrm{F} \vee \mathrm{F})$. This is evaluated using the truth table for disjunction (page 7), obtaining the final result of F. Figure 1.2 shows this evaluation process depicted on the parse tree for $((\mathrm{T} \wedge \mathrm{F}) \vee \mathrm{F})$.

Example 4 The proposition $((\neg \mathrm{F}) \Leftrightarrow \mathrm{T})$ is evaluated as follows. First, the subproposition $(\neg \mathrm{F})$ is evaluated using the truth table for negation (page 7). The result is T. Replacing $(\neg \mathrm{F})$ by T, we obtain $(\mathrm{T} \Leftrightarrow \mathrm{T})$. This is evaluated using the truth table for double-implication (page 7), obtaining the final result of T.

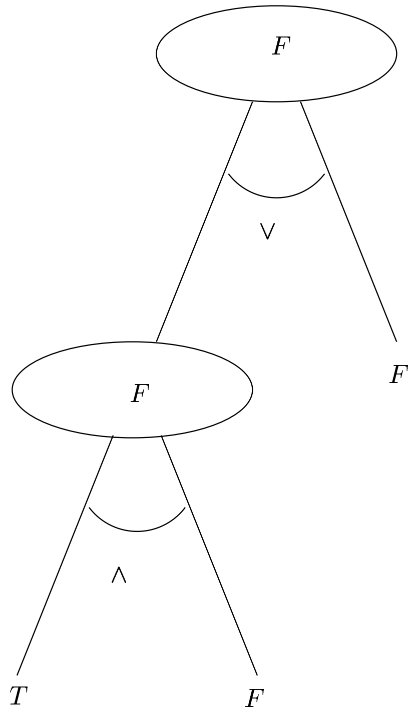


Figure 1.2: Parse tree depicting the evaluation of $((\mathrm{T} \wedge \mathrm{F}) \vee \mathrm{F})$

1.4.2 Evaluation of (General) Propositions

Now a proposition contains identifiers, in general. Hence, the proposition does not have a truth-value per se. This is because we cannot determine a truth-value for the proposition without knowing truth-values for all of the identifiers in the proposition first. For example, the proposition $p \wedge q$ is neither true nor false in itself; it is true if p and q both happen to be true (but we don't know this yet), and false otherwise.

Even though propositions do not have truth-values per se, they can be *assigned* truth-values. We assign a truth-value to a proposition by assigning truth-values to all of it's propositional identifiers. Once this is done, the truth-value of the proposition can be determined by replacing all the identifiers by their assigned values and then evaluating the resulting constant proposition as shown in subsection 1.4.1.

Propositional identifiers are assigned truth-values by means of a *state*:

Definition 2 (*State*)
A state *is a function from identifiers to truth-values.*

For example, the state $s = \{(b, \mathrm{T}), (c, \mathrm{F})\}$ assigns T to b and F to c. We use the notation $s(b)$ to denote the value that a state s assigns to an identifier b. If s assigns no value to b, then $s(b)$ is undefined. A state is sometimes also called a *truth-value assignment*. We use the term state because it is more related to the application of logic to programming, which is the focus of this class. Note that a state is somewhat like a row of a truth-table in that it assigns a value to every propositional identifier listed in the truth-table.

We say a proposition p is *well-defined* in state s iff s assigns a truth-value to every identifier in p. For example, the proposition $b \vee c$ is well-defined in the state $s = \{(b, T), (c, F)\}$, whereas the proposition $b \vee d$ is not.

If p is well-defined in s, then we use $s(p)$ to denote the truth-value assigned to p by s. $s(p)$ is evaluated as follows:

1. Replace every identifier b in p by its value $s(b)$ in state s
2. You now have a constant proposition. Evaluate it as shown above in subsection 1.4.1

Example 5 We evaluate the proposition $((p \wedge q) \vee r)$ in the state $s = \{(p, \mathrm{T}), (q, \mathrm{F}), (r, \mathrm{F})\}$. Replacing p, q, r by their values $\mathrm{T}, \mathrm{F}, \mathrm{F}$ in state s, we obtain the constant proposition $((\mathrm{T} \wedge \mathrm{F}) \vee \mathrm{F})$. From example 3, We see that this evaluates to F.

We can construct a truth-table for an arbitrary proposition by evaluating it on all 2^n possible combinations of its input values (assuming it contains n propositional identifiers).

Example 6 Truth-table for $((p \wedge q) \vee r)$. The row within lines corresponds to example 5.

p	q	r	$(p \wedge q)$	$((p \wedge q) \vee r)$
T	T	T	T	T
T	T	F	T	T
T	F	T	F	T
T	F	F	F	F
F	T	T	F	T
F	T	F	F	F
F	F	T	F	T
F	F	F	F	F

Truth-table for $((p \wedge q) \vee r)$

Example 7 We evaluate the proposition $((\neg p) \Leftrightarrow q)$ in the state $s = \{(p, \mathrm{F}), (q, \mathrm{T})\}$. Replacing p, q by their values F, T in state s, we obtain the constant proposition $((\neg \mathrm{F}) \Leftrightarrow \mathrm{T})$. From example 4, We see that this evaluates to T.

We formally define the method of evaluating propositions as follows.

Definition 3 (*Evaluation of Propositions*)
Let p, q be propositions. Then, we have

1. $s(\mathrm{T}) = \mathrm{T}$, *and* $s(\mathrm{F}) = \mathrm{F}$
2. $s(\neg p) = \neg(s(p))$
3. $s(p \wedge q) = s(p) \wedge s(q)$
4. $s(p \vee q) = s(p) \vee s(q)$
5. $s(p \Rightarrow q) = s(p) \Rightarrow s(q)$
6. $s(p \Leftrightarrow q) = s(p) \Leftrightarrow s(q)$

Note that since $s(p), s(q)$ are truth-values, it is permissible to use them as inputs to logical connectives.

Example 8 We redo example 5 using definition 3 as follows. $s((p \wedge q) \vee r) = s(p \wedge q) \vee s(r) = (s(p) \wedge s(q)) \vee s(r) = (\mathrm{T} \wedge \mathrm{F}) \vee \mathrm{F} = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$.

1.5 Precedence of Logical Connectives

In definition 1, every logical connective has a pair of associated parentheses. These parentheses are necessary so that a given proposition has a single well-defined meaning. For example, $((p \wedge q) \vee r)$ is different from $(p \wedge (q \vee r))$; in the state $s = \{(p, \mathrm{F}), (q, \mathrm{F}), (r, \mathrm{T})\}$, the first proposition evaluates to T while the second evaluates to F. Note however, that the outer parentheses are redundant in both cases, e..g, $((p \wedge q) \vee r)$ is equally well written as $(p \wedge q) \vee r$.

In general, having one pair of parentheses for each logical connective tends to result in propositions with many parentheses, which are consequently hard to read.

Precedence rules establish a convention that allows us to omit many of these parentheses. These rules are:

1. Sequences of the same connective are evaluated left to right
2. The precedence of different connectives is as follows (highest precedence first): $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Example 9 $((p \Rightarrow q) \Rightarrow r)$ can be written as $p \Rightarrow q \Rightarrow r$
$(p \Rightarrow (q \Rightarrow r))$ can be written as $p \Rightarrow (q \Rightarrow r)$
$((p \wedge q) \vee r)$ can be written as $p \wedge q \vee r$
$(p \wedge (q \vee r))$ can be written as $p \wedge (q \vee r)$
$(\ ((\neg p) \Leftrightarrow (\neg q))\ \Rightarrow r)$ can be written as $(\neg p \Leftrightarrow \neg q) \Rightarrow r$

1.6 Satisfiability and Validity, Tautologies

Definition 4 (*Satisfiable*)
A proposition p is satisfiable *iff there exists a state s such that $s(p) = \mathrm{T}$.*

We call a proposition that is not satisfiable a *contradiction* .

Definition 5 (*Valid*)
A proposition p is valid *iff for every state s such that $s(p)$ is well-defined, $s(p) = \mathrm{T}$.*

We call a proposition that is valid a *tautology*, and a proposition that is satisfiable but not valid a *contingency.*

Example 10 $\neg p \vee p$ is a tautology.
$\neg p \wedge p$ is a contradiction.
p is a contingency.

Exercise 1 Show that p is valid iff $\neg p$ is not satisfiable.
Show that p is a contingency iff both p and $\neg p$ are satisfiable.

1.7 Proving a Conjecture

Suppose we have a proposition p and we conjecture that p is valid. How do we go about actually proving this? A straightforward way would be to construct the truth-table for p as shown in example 6 and then check that the column for p contains only T's. Unfortunately, if p contains n identifiers, the truth-table for p will have 2^n rows, and so this method is not practical except for propositions containing few identifiers.

We now examine ways of showing that a proposition is valid without having to write down its entire truth-table. A central concept here is the notion of *equivalence.*

1.8 Equivalence

To show that a proposition is valid, we have to demonstrate that the proposition evaluates to T in any state that it is well-defined in. Thus, in some sense, the proposition is "equivalent" to T, as it always has the same truth-value as T (namely just T!). We can make an analogy with arithmetic expressions. The integer constant 2 is "equivalent" in some sense to the arithmetic expression $(x + x)/x$ (ignoring the case of $x = 0$).

To consider that p and q are equivalent, we want them to always have the same truth-value. In other words, we want $s(p) = s(q)$ for every state s.

Definition 6 (*Equivalent,* $\equiv$)
Propositions p and q are equivalent *if and only if, for every state s such that p and q are both well-defined in s, we have $s(p) = s(q)$. We write $p \equiv q$ for "p and q are equivalent".*

Remark 1 *$\equiv$ is commutative; $p \equiv q$ and $q \equiv p$ mean the same thing.*
$\equiv$ is not a logical-connective. We shall only use $\equiv$ in statements of the form $p \equiv q$, i.e., statements that assert the equivalence of two propositions.

Example 11 $p \wedge q$ and $q \wedge p$ are equivalent, as may easily be checked from the truth-table for conjunction. Similarly, the following pairs of propositions can be checked to be equivalent:
$\neg p \vee q$ and $p \Rightarrow q$
$p \Leftrightarrow q$ and $(p \Rightarrow q) \wedge (q \Rightarrow p)$
$p \Rightarrow (q \Rightarrow r)$ and $p \wedge q \Rightarrow r$

Example 12 The value of $p \equiv q$ for propositions p, q does not depend on the state in which evaluation is taking place. For example, let b, c be identifiers. Then

$s(b \equiv b) = \mathrm{T}$ for all states s.
$s(b \equiv c) = \mathrm{F}$ for all states s. This underlies the difference between $\equiv$ and $\Leftrightarrow$, since $s(b \Leftrightarrow c) = \mathrm{T}$ for example, if $s = \{(b, \mathrm{F}), (c, \mathrm{F})\}$.

We already have a logical connective whose meaning is "logical sameness," namely double-implication: $\Leftrightarrow$. $p \Leftrightarrow q$ is true in some state s if and only if $s(p) = s(q)$ (i.e., p and q are assigned the same truth-value by s). Hence, if $s(p) = s(q)$ in every state s, then $p \Leftrightarrow q$ is true in every state s, and vice-versa. But "$p \Leftrightarrow q$ is true in every state s" is the same as "$p \Leftrightarrow q$ is a tautology", by definition 5. Hence, we have:

Theorem 1 *$p \equiv q$ if and only if $p \Leftrightarrow q$ is a tautology.*

Proof: see the preceding discussion.

Finally, we obtain our desired characterization of validity in terms of equivalence.

Theorem 2 *p is valid if and only if $p \equiv \mathrm{T}$.*

Proof: left as an exercise.

Theorem 2 characterizes validity in terms of equivalence. This still leaves us with the problem of how to demonstrate equivalence. We do this by means of a *deductive system.*

1.9 Deductive Systems, Proofs

A deductive system, or *calculus*, is a "symbolic manipulation" system whose purpose is to "prove" statements that are "universally true" in some sense. It usually has two components:

1. A set of *axioms*: these are statements that are assumed to be universally true.
2. A set of *rules of inference*: these are rules that allow us to conclude that a particular statement is universally true from previous statements that have been shown to be universally true. Rules of inference are often written in the form

$$\frac{E_1, \ldots, E_n}{E} \qquad\qquad \frac{E_1, \ldots, E_n}{E, E'}$$

and have the following meaning:

if $E_1, \ldots, E_n$ are universally true, then so are E (and E' in the second rule)

Now given that the axioms are universally true, and that the rules of inference preserve universal truth, it follows that:

1. if we start with the axioms, and
2. conclude new statements only by applying the rules of inference to statements that have previously been shown to be universally true

then we will never incorrectly conclude that a statement is a universal truth when in fact it is not. This leads us to the following definition of proof:

Definition 7 (*Proof*)
A proof *is a finite sequence $e_1, e_2, \ldots, e_n$ of statements such that each e_i $(1 \le i \le n)$ is either an axiom, or follows from earlier statements (e_j for $1 \le j < i$) by application of a rule of inference.*

Remark 2 *Every statement that occurs in some proof is a universal truth. Every prefix of a proof is also a proof.*

We stress that the concepts of axiom, rule of inference, and proof are completely independent of the preceding material. None of the previously introduced concepts (proposition, state, evaluation, etc...) are used in the definitions of this section. A calculus is solely a “symbolic manipulation” system.

1.10 A Deductive System for Proving the Validity of Propositions

We now turn to a particular deductive system for showing validity. We stress that many (in fact, an infinite number of) such systems are possible. However, they are all “equally good” in the sense that they can all be used to prove exactly the same things, namely the set of tautologies. Since our approach to proving validity is to show that the proposition in question is equivalent to T, our particular deductive system has as its axioms the “laws of equivalence” shown in subsection 1.10.1 below. These laws give us a set of basic equivalences that we can use as starting points in our proofs. The following subsection (1.10.2) presents the rules of inference of our deductive system.

These rules allows us to substitute one proposition for another equivalent one (thereby allowing us to use "subproofs"), and to reduce the proof of an equivalence to a proof of several "intermediate" equivalences that are (presumably) easier to establish. Both rules facilitate the decomposition of a proof problem into several simpler "subproblems".

1.10.1 The Axioms: Laws of Equivalence

1. **Commutative Laws**:
   $(p \wedge q) \equiv (q \wedge p)$
   $(p \vee q) \equiv (q \vee p)$
   $(p \Leftrightarrow q) \equiv (q \Leftrightarrow p)$

2. **Associative Laws**:
   $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$
   $p \vee (q \vee r) \equiv (p \vee q) \vee r$

3. **Distributive Laws**:
   $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
   $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$

4. **De Morgans Laws**:
   $\neg(p \wedge q) \equiv \neg p \vee \neg q$
   $\neg(p \vee q) \equiv \neg p \wedge \neg q$

5. **Law of Negation**: $\neg(\neg p) \equiv p$
6. **Law of The Excluded Middle**: $p \vee \neg p \equiv \mathrm{T}$
7. **Law of Contradiction**: $p \wedge \neg p \equiv \mathrm{F}$
8. **Law of Implication**: $p \Rightarrow q \equiv \neg p \vee q$
9. **Law of Double-implication**: $(p \Leftrightarrow q) \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$
10. **Laws of or-simplification**:
    $p \vee p \equiv p$
    $p \vee \mathrm{T} \equiv \mathrm{T}$
    $p \vee \mathrm{F} \equiv p$
    $p \vee (p \wedge q) \equiv p$

11. **Laws of and-simplification**:
    $p \wedge p \equiv p$
    $p \wedge \mathrm{T} \equiv p$
    $p \wedge \mathrm{F} \equiv \mathrm{F}$

$p \wedge (p \vee q) \equiv p$

12. **Law of Identity**: $p \equiv p$

1.10.2 The Rules of Inference: the rules of Substitution and Transitivity

The rule of substitution allows us to substitute one proposition for another if they have been previously shown to be equivalent.

Rule of Substitution
Let $p \equiv q$ and let $E(b)$ be a proposition, written as a function of one of its identifiers b. Then $E(p) \equiv E(q)$ and $E(q) \equiv E(p)$.

Expressed formally, this is:

$$\frac{p \equiv q}{E(p) \equiv E(q), \quad E(q) \equiv E(p)}$$

Example 13 Let $E(b) = b \vee r$. Now $p \Rightarrow q \equiv \neg p \vee q$ by the law of implication. Hence $(p \Rightarrow q) \vee r \equiv (\neg p \vee q) \vee r$ by the rule of substitution.

The rule of transitivity allows us to "string together" two equivalences that have a common proposition.

Rule of Transitivity
If $p \equiv q$ and $q \equiv r$, then $p \equiv r$.

Expressed formally, this is:

$$\frac{p \equiv q, \quad q \equiv r}{p \equiv r}$$

Example 14 $p \Rightarrow q \equiv \neg p \vee q$ by the law of negation. Also, $\neg p \vee q \equiv q \vee \neg p$ by the law of commutativity. Hence $p \Rightarrow q \equiv q \vee \neg p$ by the rule of transitivity.

Both of these rules facilitate the decomposition of a proof problem into several simpler "subproblems".

1.11 Example Proofs, and the Simplified Proof Format

Our first example is a proof of $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$.

1.	$p \Rightarrow q \equiv \neg p \vee q$	implication
2.	$\neg p \vee q \equiv q \vee \neg p$	commutativity
3.	$\neg\neg q \equiv q$	negation
4.	$q \vee \neg p \equiv \neg\neg q \vee \neg p$	(3), substitution with $E(b) = b \vee \neg p$
5.	$\neg q \Rightarrow \neg p \equiv \neg\neg q \vee \neg p$	implication
6.	$\neg\neg q \vee \neg p \equiv \neg q \Rightarrow \neg p$	(5), substitution with $E(b) = b$
7.	$p \Rightarrow q \equiv q \vee \neg p$	(1), (2), transitivity
8.	$p \Rightarrow q \equiv \neg\neg q \vee \neg p$	(4), (7), transitivity
9.	$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$	(6), (8), transitivity

Proof of $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$

Note our proof format. On the left, we number each line. In the middle, we write down the statement that is used to build up the sequence of statements that will constitute the proof (see definition 7). On the right, we include a comment that explains the reason we are able to append the associated statement to the proof. Typically, this will contain a number (or numbers) that refer(s) to previous statements, as well as the names of the axioms (i.e., laws of equivalence) and rules of inference (i.e., rules of substitution and transitivity) that are used.

We make several remarks. First, the rule of substitution is used very often. So much so, that we will, in general, use it implicitly and omit reference to it. Second, there is a lot of repetition in the above proof. For example, every statement has a part "$\equiv p \Rightarrow q$" that is never manipulated. If we use the above format, this will often be the case. Hence we use the more economical format illustrated by the following proof of the same statement:

	$p \Rightarrow q$	
$\equiv$	$\neg p \vee q$	implication
$\equiv$	$q \vee \neg p$	commutativity
$\equiv$	$\neg\neg q \vee \neg p$	negation, substitution
$\equiv$	$\neg q \Rightarrow \neg p$	implication

Proof of $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$

Here, every statement is equivalent to the immediately preceding statement (using various axioms and rules). Hence we do not need to number the statements, but merely insert an $\equiv$ sign between each succeeding pair to indicate that succeeding pairs of statements are equivalent. This format can be used because, in the previous proof, every statement follows from the immediately preceding statement (using various axioms and rules). Note also that the rule of transitivity is being used implicitly in this proof format, as the final desired statement, namely $p \Rightarrow q \equiv \neg q \Rightarrow \neg p$, follows from all the intermediate equivalences using repeated applications of the rule of transitivity.

We define this simplified proof format as follows.

Definition 8 (*Simplified Proof Format*)
A proof in simplified proof format *of the statement $e_1 \equiv e_n$ is a finite sequence $e_1, e_2, \ldots, e_n$ of statements such that, for all i such that $(1 \leq i \leq n-1)$, $e_i \equiv e_{i+1}$ can be proven using axioms and laws of equivalence.*

We give another example of use of this proof format.

$$
\begin{array}{lll}
 & p \Rightarrow (q \Rightarrow r) & \\
\equiv & \neg p \vee (q \Rightarrow r) & \text{implication} \\
\equiv & \neg p \vee (\neg q \vee r) & \text{implication} \\
\equiv & (\neg p \vee \neg q) \vee r & \text{associative} \\
\equiv & \neg(p \wedge q) \vee r & \text{DeMorgan} \\
\equiv & (p \wedge q) \Rightarrow r & \text{implication}
\end{array}
$$

Proof of $p \Rightarrow (q \Rightarrow r) \equiv (p \wedge q) \Rightarrow r$.

1.12 Soundness of the Deductive System

We now show that every statement that is "proven" by our deductive system is in fact true. This very important property of our deductive system is called *soundness*.

Theorem 3 (*Soundness*)
If there exists a proof for e (in the sense of definition 7), then e is true.

Proof: Each law of equivalence can be shown to be true by writing down the truth tables of both of its sides, and verifying that these have the same result column.

For the rule of substitution: suppose $p \equiv q$ is true. Let s be any state whatsoever (we usually say: let s be an *arbitrary* state). By definition of how a proposition is evaluated (subsection 1.4.2), $s(E(p))$ and $(E(q))$ are computed by replacing all occurrences of p, q in $E(p), E(q)$ by $s(p), s(q)$ respectively. But $s(p) = s(q)$ since $p \equiv q$. Hence $s(E(p))$ must have the same value as $s(E(q))$. Thus $E(p) \equiv E(q)$ is true. A similar argument shows that $E(q) \equiv E(p)$ is true.

For the rule of transitivity: suppose $p \equiv q$ and $q \equiv r$ are true. Let s be any state whatsoever. Then from $p \equiv q$, we have $s(p) = s(q)$, and from $q \equiv r$ we have $s(q) = s(r)$. Hence $s(p) = s(r)$. Since s was any state whatsoever, we conclude $s(p) = s(r)$ in <u>every</u> state s. Hence $p \equiv r$ is true.

We have shown that the axioms of our deductive system (namely the laws of equivalence) are true, and the rules of inference (namely the rules of substitution and transitivity) preserve truth. Hence, any statement proven by our deductive system must be a true statement.

(end of proof)

1.13 Propositions as Sets of States

Suppose we have four states s, t, u, v where:

$s = \{(p, \mathrm{T}), (q, \mathrm{T})\}$
$t = \{(p, \mathrm{T}), (q, \mathrm{F})\}$

$u = \{(p, \mathrm{F}), (q, \mathrm{T})\}$
$v = \{(p, \mathrm{F}), (q, \mathrm{F})\}$

Consider the proposition p. Since $s(p) = \mathrm{T}$ and $t(p) = \mathrm{T}$, we cannot distinguish between states s and t by evaluating p ($s(p) = t(p)$). Similarly, we cannot distinguish between states u and v by evaluating p, as $u(p) = v(p)$. We can, however, distinguish between any state in the set $\{s, t\}$ and any state in the set $\{u, v\}$ by evaluating p, since in the first case we get T, and in the second case we get F.

We associate the set $\{s, t\}$ with the proposition p, since $\{s, t\}$ is the set of states in which p evaluates to T. Similarly, we associate:

$\{u, v\}$ with $\neg p$ since $u(\neg p) = v(\neg p) = \neg\mathrm{F} = \mathrm{T}$
$\{s, u\}$ with q since $s(q) = u(q) = \mathrm{T}$
$\{t, v\}$ with $\neg q$ since $t(\neg q) = v(\neg q) = \neg\mathrm{F} = \mathrm{T}$

Figure 1.3 shows the states s, t, u, v as points, and the sets of states corresponding to the propositions $p, \neg p, q, \neg q$ as ovals surrounding the appropriate states. [1]

In general, we associate with a proposition p the set of all states in which p is true:

Definition 9 (*truestates*)
$truestates(p) = \{s \mid s(p) = \mathrm{T}\}$

We can also go the other way — start with a set A of states and derive a proposition p such that $s(p) = \mathrm{T}$ if and only if $s \in A$. In other words, we can associate with a set of states A a proposition p that is true in every state in A, and false in every state not in A. In other words, we want a function *prop* from sets of states to propositions that is the "inverse" of the *truestates* function: $p \equiv prop(A)$ iff $A = truestates(p)$.

First let us consider the case where A contains a single state, say s. Hence, we want $prop(A)$ to be a proposition that is true in state s and false in every other state. Let $b, c, d, \ldots$ be all the identifiers that are assigned T by s, and $b', c', d', \ldots$ be all the identifiers that are assigned F by s. Then, we define:

$$prop(s) = (b \land c \land d \land \ldots) \land (\neg b' \land \neg c' \land \neg d' \land \ldots)$$

If $p = prop(s)$, then we have:
$s(p) =$
$s((b \land c \land d \land \ldots) \land (\neg b' \land \neg c' \land \neg d' \land \ldots)) =$
$(s(b) \land s(c) \land s(d) \land \ldots) \land (\neg s(b') \land \neg s(c') \land \neg s(d') \land \ldots) =$
$(\mathrm{T} \land \mathrm{T} \land \mathrm{T} \land \ldots) \land (\neg\mathrm{F} \land \neg\mathrm{F} \land \neg\mathrm{F} \land \ldots) =$
$\mathrm{T} \land \mathrm{T} =$
T

On the other hand, if t differs from s in its assignment to at least one proposition, then $t(p) = \mathrm{F}$ as we now demonstrate There are two cases.

[1] This notation is basically that of *Venn diagrams*.

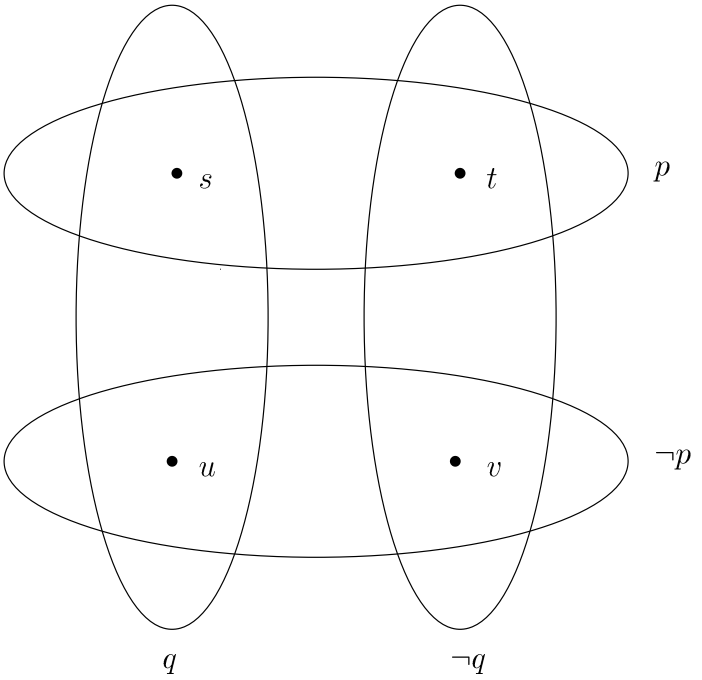


Figure 1.3: Sets of states corresponding to $p, \neg p, q, \neg q$

Case 1: t assigns F to at least one identifier of $b, c, d, \ldots$.

$t(p) =$
$t((b \wedge c \wedge d \wedge \ldots) \wedge (\neg b' \wedge \neg c' \wedge \neg d' \wedge \ldots)) =$
$(t(b) \wedge t(c) \wedge t(d) \wedge \ldots) \wedge (\neg t(b') \wedge \neg t(c') \wedge \neg t(d') \wedge \ldots) =$
$(\mathrm{T} \wedge \mathrm{T} \wedge \mathrm{T} \wedge \ldots \wedge \mathrm{F} \wedge \ldots) \wedge (\neg \mathrm{F} \wedge \neg \mathrm{F} \wedge \neg \mathrm{F} \wedge \ldots) =$
$\mathrm{F} \wedge \mathrm{T} =$
F

Case 2: t assigns T to at least one identifier of $b', c', d', \ldots$

$t(p) =$
$t((b \wedge c \wedge d \wedge \ldots) \wedge (\neg b' \wedge \neg c' \wedge \neg d' \wedge \ldots)) =$
$(t(b) \wedge t(c) \wedge t(d) \wedge \ldots) \wedge (\neg t(b') \wedge \neg t(c') \wedge \neg t(d') \wedge \ldots) =$
$(\mathrm{T} \wedge \mathrm{T} \wedge \mathrm{T} \wedge \ldots) \wedge (\neg \mathrm{F} \wedge \neg \mathrm{F} \wedge \neg \mathrm{F} \wedge \ldots \wedge \neg \mathrm{T} \wedge \ldots) =$
$\mathrm{T} \wedge \mathrm{F} =$
F

Lemma 4 *Let s, t be unequal states. Then*

1. $s(prop(s)) = \mathrm{T}$
2. $t(prop(s)) = \mathrm{F}$

Proof: see preceding discussion.

Now we consider the general case, where $A = \{s, t, u, \ldots\}$. We define:

$$prop(A) = prop(s) \vee prop(t) \vee prop(u) \vee \ldots$$

To see that this definition works, let $p = prop(A)$ and consider $v(p)$ for some state v. If $v \in A$, then

$v(p) =$
$v(prop(s) \vee prop(t) \vee prop(u) \vee \ldots \vee prop(v) \vee \ldots) =$
$v(prop(s)) \vee v(prop(t)) \vee v(prop(u)) \vee \ldots \vee v(prop(v)) \vee \ldots) =$ (by lemma 4)
$\mathrm{F} \vee \mathrm{F} \vee \mathrm{F} \vee \ldots \vee \mathrm{T} \vee \ldots =$
T.

On the other hand, if $v \notin A$, then

$v(p) =$
$v(prop(s) \vee prop(t) \vee prop(u) \vee \ldots) =$
$v(prop(s)) \vee v(prop(t)) \vee v(prop(u)) \vee \ldots) =$
$\mathrm{F} \vee \mathrm{F} \vee \mathrm{F} \vee \ldots =$
F.

We summarize the above discussion as follows.

Definition 10 ($prop$)
$prop(s) = (b \wedge c \wedge d \wedge \ldots) \wedge (\neg b' \wedge \neg c' \wedge \neg d' \wedge \ldots)$

where $b, c, d, \ldots$ *are all the identifiers that are assigned* T *by* s*, and* $b', c', d', \ldots$ *are all the identifiers that are assigned* F *by* s*.*

$prop(A) = prop(s) \vee prop(t) \vee prop(u) \vee \ldots$

where $A = \{s, t, u, \ldots\}$*.*

Theorem 5 *Let* s *be a state and* A *be a set of states. Then*

$$s(prop(A)) = \mathrm{T} \text{ if and only if } s \in A$$

Proof: Left as an exercise.

Exercise 2 If $A = \emptyset$, i.e., A is empty, what is $prop(A)$?

1.14 Normal Forms

It is occasionally very useful to be able to convert a proposition into an equivalent proposition that has a particular syntactic form. Two forms in particular shall concern us — *disjunctive normal form* and *conjunctive normal form.*

Definition 11 (*Literal*)
A literal *is either a propositional identifier or the negation of a propositional identifier.*

Definition 12 (*Disjunctive Normal Form*)
A proposition is in disjunctive normal form *iff it is a disjunction of conjunctions of literals.*

Definition 13 (*Conjunctive Normal Form*)
A proposition is in conjunctive normal form *iff it is a conjunction of disjunctions of literals.*

Theorem 6 *For every proposition* p*, there is an equivalent proposition in disjunctive normal form.*

Proof: Left as an exercise.

Theorem 7 *For every proposition* p*, there is an equivalent proposition in conjunctive normal form.*

Proof: Left as an exercise.

Example 15 The proposition $p \Leftrightarrow q$ can be expressed in disjunctive normal form as $(p \wedge q) \vee (\neg p \wedge \neg q)$. It can be expressed in conjunctive normal form as $(\neg p \vee q) \wedge (p \vee \neg q)$.

Chapter 2

The Predicate Calculus

2.1 Predicates

A *predicate* is like a proposition, except that propositional identifiers may be replaced by any expression that has value T or F, e.g.:

1. Arithmetic inequalities: $=, \neq, <, \leq, >, \geq$

2. Logical quantifiers: These allow you to express "for all" and "there exists" in formal logic.

These expressions are called *atomic predicates.* Atomic predicates play an analogous role in predicates that propositional identifiers do in propositions. They provide the expressions that are evaluated in a given state to provide truth-values. These truth-values are combined using the logical connectives to produce the final truth-value of a predicate.

The syntax of atomic predicates is defined as follows.

Definition 14 (*Atomic Predicate*)
Atomic Predicates are formed as follows:

1. *T and F are atomic predicates.*

2. *A propositional identifier is an atomic predicate.*

3. *exp op exp′ is an atomic predicate, where exp, exp′ are arithmetic expressions, and op is one of* $\{=, \neq, <, \leq, >, \geq\}$.

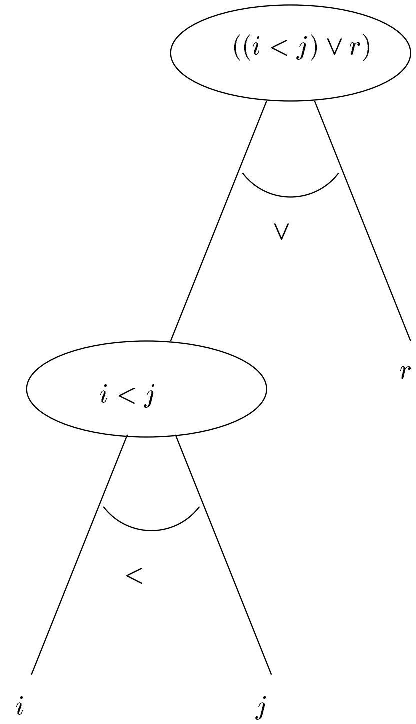
4. $\mathbf{LQ}(i : range : quantified-expression)$, *where* $\mathbf{LQ}$ *is a logical quantifier,* i *is an integer variable, and* $range$, $quantified-expression$ *are both predicates.*

The syntax of predicates is defined as follows.

Definition 15 (*Predicate*)
Predicates are formed as follows:

1. *An atomic predicate is a predicate.*
2. *If p is a predicate, then so is $(\neg p)$.*
3. *If p and q are predicates, then so are $(p \wedge q)$, $(p \vee q)$, $(p \Rightarrow q)$, $(p \Leftrightarrow q)$.*

Example 16 If i, j are integer variables and r is a proposition, then $((i < j) \vee r)$ is a predicate.



Parse tree for $((i < j) \vee r)$

2.1.1 Precedence of Operators in a Predicate

The operators, such as $<, =$, used in atomic predicates have higher precedence than logical connectives.

Example 17 $((i < j) \vee r)$ can be rewritten as $i < j \vee r$.

2.1.2 Arithmetic Inequalities

We assume all the familiar properties of arithmetic inequalities. These can be used in proofs by giving "arithmetic" as the "law" used. Some typical properties that you might use are:

- $i < j \wedge j < k \Rightarrow i \leq k$
  $i \leq j \wedge j \leq k \Rightarrow i \leq k$
- $i \leq j \wedge j \leq i \Rightarrow i = j$
- $i < j \Rightarrow i + k < j + k$
  $i \leq j \Rightarrow i + k \leq j + k$
- $k > 0 \wedge i < j \Rightarrow k * i < k * j$
  $k \geq 0 \wedge i \leq j \Rightarrow k * i \leq k * j$
- $k < 0 \wedge i < j \Rightarrow k * i > k * j$
  $k \leq 0 \wedge i \leq j \Rightarrow k * i \geq k * j$

2.2 Quantifiers

2.2.1 Logical Quantifiers — The Universal Quantifier $\forall$

$\forall(i : r(i) : p(i))$ means:

> *For every value v of i such that $r(v)$ is true, $p(v)$ is also true.*

Example 18 Let $a[0..(n-1)]$ be an array of integer.
$\forall(i : 0 \leq i < n : a[i] > 0)$ means that every element of array a is positive.
$\forall(i : 0 \leq i < n-1 : a[i] \leq a[i+1])$ means that a is sorted in nondecreasing order.

2.2.2 Logical Quantifiers — The Existential Quantifier $\exists$

$\exists(i : r(i) : p(i))$ means:

> *There exists a value v of i such that $r(v)$ is true and $p(v)$ is also true.*

Example 19 Let $a[0..(n-1)]$ be an array of integer.
$\exists(i : 0 \leq i < n : a[i] > 0)$ means that some element of array a is positive.
$\exists(i : 0 \leq i < n : a[i] = x)$ means that some element of array a is equal to x.

2.2.3 Free and Bound Variables

We use **LQ** to stand for either $\forall$ or $\exists$. In $\mathbf{LQ}(i : r(i) : p(i))$:

- i is the *bound variable.* i is said to be *bound to* **LQ**.
- $r(i)$ is the *range of quantification* (or simply *range*)

- $p(i)$ is the *quantified predicate.*

A variable that is not bound to some quantifier is said to be *free*. In $\mathbf{LQ}(i : r(i) : p(i))$, the bound variable i is a "place holder" that can be replaced by another variable j provided that this does not cause *capture*:

$\exists(d :: x = y * d)$ and $\exists(m :: x = y * m)$ mean the same thing, namely x is a multiple of y, but $\exists(x :: x = y * x)$ means T (i.e., it is valid), since the quantified predicate $x = y * x$ is true for $x = 0$.

2.2.4 Arithmetic Expressions and Quantifiers

Arithmetic expressions are built up from inequalities, the arithmetic operators $(+, *, -, /,$ etc.), and the following:

1. $\mathbf{AQ}(i : range : quantified{-}expression)$, where **AQ** is an arithmetic quantifier, i is an integer variable, $range$ is a predicate, and $quantified - expression$ is an arithmetic expression.
2. $\mathbf{N}(i : range : quantified - expression)$ where i is an integer valued variable, and $range$, $quantified - expression$ are both predicates.

Arithmetic quantifiers usually generalize a binary arithmetic operation to a set of operands. Examples are:

$\mathbf{\Sigma}$ (sum) generalizes $+$: $\mathbf{\Sigma}(i : 0 \leq i < n : i) = 0 + 1 + \ldots + n - 1$
$\mathbf{\Pi}$ (product) generalizes $*$: $\mathbf{\Pi}(i : 1 \leq i \leq n : i) = 1 * 2 * \ldots * n$
MIN generalizes $min(x, y)$: $\mathbf{MIN}(i : 1 \leq i \leq n : i) = 1$
MAX generalizes $max(x, y)$: $\mathbf{MAX}(i : 1 \leq i \leq n : i) = n$

The Counting Quantifier N

$\mathbf{N}(i : r(i) : p(i))$ is the number of values for i within the range $r(i)$ for which $p(i)$ is true.

i.e., **N** counts the number of times that $p(i)$ is true within the range $r(i)$. **N** can be defined in terms of $\mathbf{\Sigma}$:

$$\mathbf{N}(i : r(i) : p(i)) = \mathbf{\Sigma}(i : r(i) \wedge p(i) : 1)$$

Example 20 $\mathbf{N}(i : 0 \leq i < n : even(a[i]))$ where $even(x) \equiv \exists(d :: 2 * d = x)$ gives the number of even elements in array a.

2.3 Properties of Quantifiers

We use **Q** for any quantifier except **N**. Every **Q** generalizes an associative and commutative binary operator **q** to a set of operands given by the range. There are many laws that can be used to manipulate quantifiers.

2.3.1 Quantifying Over an Empty Range

If the range of quantification is empty, then the result is the identity element of the associated binary operator:

$$\forall(i : \mathrm{F} : p(i)) = \mathrm{T}$$

$$\exists(i : \mathrm{F} : p(i)) = \mathrm{F}$$

$$\mathbf{N}(i : \mathrm{F} : p(i)) = 0$$

$$\mathbf{\Sigma}(i : \mathrm{F} : f(i)) = 0$$

$$\mathbf{\Pi}(i : \mathrm{F} : f(i)) = 1$$

$$\mathbf{MIN}(i : \mathrm{F} : f(i)) = \infty$$

$$\mathbf{MAX}(i : \mathrm{F} : f(i)) = -\infty$$

2.3.2 Quantifiers — Bound Variable Laws

a) Change of variable

$$\boxed{\mathbf{Q}(i : r(i) : f(i)) \;=\; \mathbf{Q}(k : r(k) : f(k))}$$

where k is not free in $r(i), f(i)$

Example 21 $\forall(i : 0 \leq i < n : a[i] \leq a[i+1]) \equiv \forall(k : 0 \leq k < n : a[k] \leq a[k+1])$

$\mathbf{\Sigma}(i : 0 \leq i < n : a[i]) = \mathbf{\Sigma}(k : 0 \leq k < n : a[k])$

b) Cartesian Product

$$\boxed{\mathbf{Q}(i,j : r(i) \wedge s(i,j) : f(i,j)) \;=\; \mathbf{Q}(i : r(i) : \mathbf{Q}(j : s(i,j) : f(i,j)))}$$

Example 22 Let $a[i,j]$ be an $n \times m$ array of integer.

$\exists(i,j : 0 \leq i < n \wedge 0 \leq j < m : a[i,j] = x) \equiv \exists(i,j : 0 \leq i < n : \exists(j : 0 \leq j < m : a[i,j] = x)$

2.3.3 Quantifiers — Range Laws

a) Range Translation

$$\boxed{\mathbf{Q}(i : r(i) : f(i)) \;=\; \mathbf{Q}(i : r(g(i)) : f(g(i)))}$$

where g is a 1-to-1 function

Example 23 $\mathbf{\Sigma}(i : 1 \leq i \leq n : i) = \mathbf{\Sigma}(i : 0 \leq i \leq n-1 : i+1)$

where $r(i) = 1 \leq i \leq n$, $g(i) = i+1$. Hence the range on the right hand side is: $1 \leq i+1 \leq n$, i.e., $0 \leq i \leq n-1$

b) Singleton Range

$$\boxed{\mathbf{Q}(i : i = k : f(i)) \;\; = \;\; f(k)}$$

c) Range Splitting

$$\boxed{\mathbf{Q}(i : r(i) : f(i)) \;\; = \;\; \mathbf{Q}(i : r(i) \wedge b(i) : f(i)) \;\mathbf{q}\; \mathbf{Q}(i : r(i) \wedge \neg b(i) : f(i))}$$

Example 24 $\mathbf{MIN}(i : 0 \leq i \leq j+1 : a[i]) =$
$\mathbf{MIN}(i : 0 \leq i \leq j+1 \wedge i \leq j : a[i]) \; min \; \mathbf{MIN}(i : 0 \leq i \leq j+1 \wedge i > j : a[i]) =$
$\mathbf{MIN}(i : 0 \leq i \leq j : a[i]) \; min \; \mathbf{MIN}(i : i = j+1 : a[i]) =$
$\mathbf{MIN}(i : 0 \leq i \leq j : a[i]) \; min \; a[j+1]$

d) Identity Element (Empty Range)

$$\boxed{\mathbf{Q}(i : \mathrm{F} : f(i)) \text{ is the identity element of } \mathbf{q}}$$

Note that range splitting works correctly when quantification over an empty range is defined this way: $\mathbf{Q}(i : r(i) : f(i)) \;\; = \;\; \mathbf{Q}(i : r(i) : f(i)) \;\mathbf{q}\; \mathbf{Q}(i : \mathrm{F} : f(i))$.

f) Range Disjunction

$$\boxed{\mathbf{Q}(i : r(i) \vee s(i) : f(i)) \;\; = \;\; \mathbf{Q}(i : r(i) : f(i)) \;\mathbf{q}\; \mathbf{Q}(i : s(i) : f(i))}$$

provided $\mathbf{q}$ is idempotent: $x \; q \; x = x$.

Example 25 $\mathbf{MIN}(i : 0 \leq i < n : a[i]) \;\; = \;\; \mathbf{MIN}(i : i \in \varphi : a[i]) \; min \; \mathbf{MIN}(i : i \in \psi : a[i])$

provided $\varphi \cup \psi = \{0, 1, \ldots, n-1\}$. This is useful, for example, when the $a[i]$ are distributed over a network. The minimum can then be computed by "probes" that may possibly overlap.

2.3.4 Quantifiers — Function Laws

a) Generalized Associativity

$$\boxed{\mathbf{Q}(i : r(i) : f(i) \;\mathbf{q}\; g(i)) \;\; = \;\; \mathbf{Q}(i : r(i) : f(i)) \;\mathbf{q}\; \mathbf{Q}(i : r(i) : g(i))}$$

Example 26 $\forall(i : 0 \leq i < n : a[i] > 0 \wedge even(a[i])) \equiv$
$\forall(i : 0 \leq i < n : a[i] > 0) \wedge \forall(i : 0 \leq i < n : even(a[i]))$

b) Generalized Commutativity

$$\boxed{\mathbf{Q}(i : r(i) : \mathbf{Q}(j : s(j) : f(i,j))) \ = \ \mathbf{Q}(j : s(j) : \mathbf{Q}(i : r(i) : f(i,j)))}$$

Example 27 $\forall(i : 0 \leq i < n : \forall(j : 0 \leq j < m : a[i,j] > 0)) \equiv$
$\forall(j : 0 \leq j < m : \forall(i : 0 \leq i < n : a[i,j] > 0))$

$\mathbf{\Sigma}(i : 0 \leq i < n : \mathbf{\Sigma}(j : 0 \leq j < m : a[i,j])) = \mathbf{\Sigma}(j : 0 \leq j < m : \mathbf{\Sigma}(i : 0 \leq i < n : a[i,j]))$

2.3.5 Quantifiers — Range and Function Interchange

a) $\forall$-rule

$$\boxed{\forall(i : r(i) \wedge s(i) : p(i)) \ \equiv \ \forall(i : r(i) : s(i) \Rightarrow p(i))}$$

Example 28 $\forall(i : \mathrm{T} \wedge 0 \leq i < n : a[i] = 0) \equiv \forall(i : \mathrm{T} : 0 \leq i < n \Rightarrow a[i] = 0)$

When the range is T, it can be omitted: $\forall(i :: 0 \leq i < n \Rightarrow a[i] = 0)$

b) $\exists$-rule

$$\boxed{\exists(i : r(i) \wedge s(i) : p(i)) \ \equiv \ \exists(i : r(i) : s(i) \wedge p(i))}$$

Example 29 $\exists(i : \mathrm{T} \wedge 0 \leq i < n : a[i] = x) \equiv \exists(i : \mathrm{T} : 0 \leq i < n \wedge a[i] = x)$

2.4 States

States must now assign appropriate values to all variables (depending on the *type* of the variables).

Example 30 If i, j are integer variables and r is a propositional identifier, then

$$s = \{(i, 2), (j, 3), (r, \mathrm{F})\}.$$

is an example of a state.

Variable types will either be declared, or will be easily inferable from context.

2.5 Evaluation of Predicates

2.5.1 Evaluation of Constant Predicates

A *constant predicate* is a predicate that does not contain any propositional identifiers or free variables. A constant predicate is evaluated as follows.

1. The value of T, F is T, F respectively. The value of an integer constant n is n.
2. If there is only one quantifier, arithmetic operator, or logical connective, then evaluate it according to the definitions of quantifiers, arithmetic operators, and logical connectives.
3. If there are n ($n > 1$) quantifiers, arithmetic operators, or logical connectives, then
   (a) Evaluate all subpredicates that contain exactly one quantifier, arithmetic operator, or logical connective, and replace them by their values.
   (b) Repeat the previous step until you are left with either T or F.

2.5.2 Evaluation of (General) Predicates

This is similar to the evaluation of general propositions (see chapter 1):

1. Replace all identifiers by their values in the state.
2. You now have a constant predicate. Evaluate as shown previously.

Definition 16 (*Evaluation of Predicates*)
Let p be a predicate and s be a state. Then the value of p in s is denoted by $s(p)$, and is defined as follows:

1. $s(\mathrm{T}) = \mathrm{T}$, *and* $s(\mathrm{F}) = \mathrm{F}$
2. $s(\neg p) = \neg(s(p))$,
   $s(p \wedge q) = s(p) \wedge s(q)$,
   $s(p \vee q) = s(p) \vee s(q)$,
   $s(p \Rightarrow q) = s(p) \Rightarrow s(q)$,
   $s(p \Leftrightarrow q) = s(p) \Leftrightarrow s(q)$
3. $s(n) = n$ *for any integer constant* n
4. $s(exp\ op\ exp') = s(exp)\ op\ s(exp')$ *for any arithmetic operator* op
5. $s(\,\mathbf{Q}(i : r(i) : p(i))\,) = \mathbf{Q}(i : s(r(i)) : s(p(i)))$ *for any quantifier* $\mathbf{Q}$

Example 31 $s = \{(a[0], 1), (a[1], 5), (a[2], 3), (a[3], 10), (j, 2)\}$.

$s(\,\forall(i : 0 \le i < j : a[i] \le a[i+1])\,) =$

$\forall(\,i : s(0) \le i < s(j) : s(a[i]) \le s(a[i+1])\,) =$

$\forall(\,i : 0 \le i < 2 : s(a[i]) \le s(a[i+1])\,) =$

$s(a[0]) \le s(a[1]) \;\wedge\; s(a[1]) \le s(a[2]) =$

$1 \le 5 \;\wedge\; 5 \le 3 =$

$\mathrm{T} \;\wedge\; \mathrm{F} =$

F

2.6 Notation for Functions, Sets, and Predicates

We use "functional notation." The type is determined from the definition (we use $=$ for functions and sets, $\equiv$ for predicates) and context.

Example 32 $children(i) = \{j : 2*i+1 \le j \le 2*i+2\}$ is a set which gives the children of node i in a binary heap.

$divides(q, x) \equiv \exists(d :: x = q*d)$ is a predicate which is true iff q divides x

Example 33 Nonredundant copy.

a : array$[0..m-1]$ of integer;
b : array$[0..n-1]$ of integer;

array b is a copy of a with duplicates removed:

1) Every element of a occurs in b: $\forall(i : 0 \le i < m : \exists(j : 0 \le j < n : b[j] = a[i]))$

2) b contains no duplicates: $\forall(i, j : 0 \le i, j < n \wedge i \ne j : b[i] \ne b[j])$

We can write the predicate $nonredundant - copy(b, a)$ as the conjunction of the above two predicates.

Example 34 Longest plateau.

$allequal(j, len) \equiv \forall(i, k : j - len \le i, k \le j - 1 : a[i] = a[k])$

$plateau(len, n) \equiv \exists(\ell : 0 < len \le \ell \le n : allequal(\ell, len))$

$longest - plateau - length(n) = \mathbf{MAX}(len : plateau(len, n) : len)$

Example 35 All equal values adjacent.

No two equal array values are separated by an unequal array value:

$adjacent - equal - values(n) \equiv$

$$\forall(i,j : 0 \leq i < j < n : a[i] = a[j] \Rightarrow \forall(k : i \leq k \leq j : a[k] = a[i]))$$

Example 36 Sorting.

a results from sorting b in nondecreasing order:

$is - sorted(a,b) \equiv perm(a,b) \wedge ordered - nondec(a)$

$ordered - nondec(a) \equiv \forall(i : 0 \leq i < n-1 : a[i] \leq a[i+1])$

$perm(a,b) \equiv$

$\forall(i : 0 \leq i < n :$
$\quad num(a, a[i]) = num(b, a[i]) \wedge$
$\quad num(a, b[i]) = num(b, b[i])$
$)$

$num(c,x) = \mathbf{N}(i : 0 \leq i < n : c[i] = x)$

Chapter 3

Verification of Program Correctness

3.1 Our Programming Language

We shall use a simplified programming language that consists of assignment statements, **if** statements, **while** statements, and sequential composition of statements (denoted by a semicolon). **begin** and **end** are used to bracket statements. The syntax of our programming language is as follows.

assignment statement:
    <variable> := <expression>

if statement:
    if <predicate> **then** <statement> **else** <statement> **endif** |
    if <predicate> **then** <statement> **endif**

while statement:
    while <predicate> **do** <statement> **endwhile**

3.2 Conditional Correctness of Programs: The Hoare Triple Notation {P} S {Q}

In the notation $\{P\}\,S\,\{Q\}$:

P is a predicate, called the **precondition**.
S is a statement.
Q is a predicate, called the **postcondition**.

$\{P\}\,S\,\{Q\}$ is shorthand notation for the following:

If execution of S is started in a state satisfying P, then:
    <u>if</u> execution of S terminates, the final state is guaranteed to satisfy Q.

Because termination is *assumed*, this is called *conditional correctness.*

3.2.1 Validity of $\{P\}\, S\, \{Q\}$

For $\{P\}\, S\, \{Q\}$ to have the meaning given above, we define the validity of $\{P\}\, S\, \{Q\}$ as follows.

Definition 17 (*Validity of* $\{P\}\, S\, \{Q\}$)

$\{P\}\, S\, \{Q\}$ *is* valid *iff*

For every state s *such that* $s(P) = \mathrm{T}$*:*
    If execution of S *is started in* s*, then:*
        if the execution terminates, it does
        so in some state t *such that* $t(Q) = \mathrm{T}$

3.3 Program Specification

We <u>specify</u> what a program should do by giving a precondition and postcondition for the program.

Example 37 Search a sorted array $C[0 : n-1]$ for an existing value X.

Precondition: $\forall(i : 0 \leq i < n-1 : C[i] \leq C[i+1]) \wedge \exists(i : 0 \leq i < n : C[i] = X)$.

Postcondition: $C[pos] = X$.

Example 38 Sort an array a.

Precondition: $A = a$.

Postcondition: $is - sorted(a, A)$, where $is - sorted$ is defined as follows:

$is - sorted(a, b) \equiv perm(a, b) \wedge ordered - nondec(a)$

$ordered - nondec(a) \equiv \forall(i : 0 \leq i < n-1 : a[i] \leq a[i+1])$

$perm(a, b) \equiv$
$\forall(i : 0 \leq i < n :$
    $num(a, a[i]) = num(b, a[i]) \wedge$
    $num(a, b[i]) = num(b, b[i])$
$)$

$num(c, x) = \mathbf{N}(i : 0 \leq i < n : c[i] = x)$

Here a, b, c are all arrays of integer with index range 0 to $n-1$ inclusive.

Note how, in the last example, the array A is used to store the initial value of array a. In general, when writing a specification for a program, we will often need to relate the initial values of program variables to their final values. We shall usually do this as follows:

1. We use the precondition to make a "copy" of the initial values of the variables, e.g., $A = a$ in example 38 copies the initial value of array a into array A.
2. We use the postcondition to relate the final values of the variables to the initial values, e.g., $is - sorted(a, A)$ in example 38 states that the final value of a must be the result of sorting the initial value of a (which is now given by A).

We shall therefore make the following convention

Convention: Upper-case variables are unchanged by the program.

This convention allows us to use upper-case variables to record initial values.

3.4 A Deductive System for Proving the Validity of Hoare Triples

Just as for propositions, we demonstrate the validity of Hoare triples by using a deductive system. Our deductive system has one proof rule for each type of program statement, together with two proof rules called the *rules of consequence*. Hence, to prove a given Hoare triple valid, there is usually only one proof rule that can be applied at any time. Our proof rules are presented as *rules of inference*: if the *hypotheses* (the part above the line) have been proven to be valid, then the conclusion (the part below the line) is also valid. The only exception is the *assignment axiom*, which has no hypothesis. In other words, any instance of the assignment axiom can be taken to be valid without first having to prove a hypothesis valid. We now discuss each proof rule in turn.

3.4.1 The Assignment Axiom

$$\{Q(e)\}\, x := e\, \{Q(x)\} \text{ is valid.}$$

x has the value after execution that e has before, so $Q(x)$ is true after iff $Q(e)$ is true before.

Example 39 $\{x + 1 \leq 5\}\, x := x + 1\, \{x \leq 5\}$.
This reduces to: $\{x \leq 4\}\, x := x + 1\, \{x \leq 5\}$. In other words, if we want $x \leq 5$ to be true after executing $x := x + 1$, then $x \leq 4$ must be true before executing $x := x + 1$. This conforms to our intuition about the meaning of $x := x + 1$.

Example 40 $\{10 = 10\}\, x := 10\, \{x = 10\}$.
This reduces to $\{\mathrm{T}\}\, x := 10\, \{x = 10\}$. In other words, $x = 10$ is always guaranteed to be true after executing $x := 10$, since the precondition is just T, which is always true by definition.

Example 41 $\{10 = 11\}\, x := 10\, \{x = 11\}$.
This reduces to $\{\mathrm{F}\}\, x := 10\, \{x = 10\}$. In other words, $x = 11$ is never true after executing $x := 10$, since the precondition is F, which is never true by definition.

3.4.2 The two-way-if Rule

$$\frac{\{P \wedge B\}\, S_1\, \{Q\} \qquad \{P \wedge \neg B\}\, S_2\, \{Q\}}{\{P\}\, \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\, \{Q\}}$$

The hypotheses of the rule require a proof of correctness for both possible cases of execution:

- B evaluates to true and S_1 is executed, or
- B evaluates to false and S_2 is executed.

We don't know in advance which path will be taken, since this depends on the values of the program variables at run time, which cannot be predicted. Hence, we have to account for both possibilities, i.e., both paths. The rule works as follows.

Assume that the hypotheses of the rule, namely $\{P \wedge B\}\, S_1\, \{Q\}$ and $\{P \wedge \neg B\}\, S_2\, \{Q\}$, are both valid. Assume also that precondition P is true immediately before executing the **if**-statement. If the first case of execution occurs, i.e., B evaluates to true and S_1 is executed, then we know that P is true immediately before execution of S_1 (by our assumption), and that B is true immediately before execution of S_1 (otherwise S_1 would not be executed, by definition of the **if**-statement). Hence we know that $P \wedge B$ is true immediately before execution of S_1. Therefore, from $\{P \wedge B\}\, S_1\, \{Q\}$, we know that Q is true immediately after execution of S_1. On the other hand, assume that the second case of execution occurs, i.e., B evaluates to false and S_2 is executed. Then, we know that P is true immediately before execution of S_2 (by our assumption), and that B is false immediately before execution of S_2 (otherwise S_2 would not be executed, by definition of the **if**-statement). Hence we know that $P \wedge \neg B$ is true immediately before execution of S_2. Therefore, from $\{P \wedge \neg B\}\, S_2\, \{Q\}$, we know that Q is true immediately after execution of S_2.

Therefore, in both cases, we have shown that Q is true after execution of the **if**-statement. Our assumptions were: 1) the hypotheses $\{P \wedge B\}\, S_1\, \{Q\}$ and $\{P \wedge \neg B\}\, S_2\, \{Q\}$, and 2) that precondition P is true immediately before execution of the **if**-statement. In other words, given the hypotheses $\{P \wedge B\}\, S_1\, \{Q\}$ and $\{P \wedge \neg B\}\, S_2\, \{Q\}$, then if P is true before execution of the **if**-statement, Q will be true after execution of the **if**-statement.

Another way of saying this is that given the hypotheses $\{P \wedge B\}\, S_1\, \{Q\}$ and $\{P \wedge \neg B\}\, S_2\, \{Q\}$, we have proven $\{P\}\, \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2\, \{Q\}$. This is exactly the two-way-**if** rule.

Example 42 computing the max of two integers.

Prove:

$\{true\}$
if $x \geq y$ **then** $z := x$ **else** $z := y$
$\{z = max(x, y)\}$.

Using the two-way-**if** rule, this reduces to:

1) $\{x \geq y\}\ z := x\ \{z = max(x, y)\}$, and

2) $\{x < y\}$ $z := y$ $\{z = max(x, y)\}$.

3.4.3 The one-way-if Rule

$$\frac{\{P \wedge B\}\, S_1\, \{Q\} \qquad (P \wedge \neg B) \Rightarrow Q}{\{P\}\, \textbf{if}\ B\ \textbf{then}\ S_1\, \{Q\}}$$

The hypotheses of the rule require a proof of correctness for both possible cases of execution:

- B evaluates to true and S_1 is executed, or
- B evaluates to false and no statement is executed.

We don't know in advance which path will be taken, since this depends on the values of the program variables at run time, which cannot be predicted. Hence, we have to account for both possibilities, i.e., both paths. The rule works as follows.

Assume that the hypotheses of the rule, namely $\{P \wedge B\}\, S_1\, \{Q\}$ and $(P \wedge \neg B) \Rightarrow Q$, are both valid. Assume also that precondition P is true immediately before executing the **if**-statement. If the first case of execution occurs, i.e., B evaluates to true and S_1 is executed, then we know that P is true immediately before execution of S_1 (by our assumption), and that B is true immediately before execution of S_1 (otherwise S_1 would not be executed, by definition of the **if**-statement). Hence we know that $P \wedge B$ is true immediately before execution of S_1. Therefore, from $\{P \wedge B\}\, S_1\, \{Q\}$, we know that Q is true immediately after execution of S_1. On the other hand, assume that the second case of execution occurs, i.e., B evaluates to false and no statement is executed. Then, we know that P is true immediately before the **if**-statement (by our assumption), and that B is false immediately before the **if**-statement (otherwise S_1 would have been executed, by definition of the **if**-statement). Hence we know that $P \wedge \neg B$ is true immediately before the **if**-statement. Therefore, from $(P \wedge \neg B) \Rightarrow Q$, we know that Q is true immediately before the **if**-statement. Since execution of the **if**-statement involves no change of state, i.e., "no statement is executed," Q will also be true immediately after the **if**-statement.

Therefore, in both cases, we have shown that Q is true immediately after execution of the **if**-statement. Our assumptions were: 1) the hypotheses $\{P \wedge B\}\, S_1\, \{Q\}$ and $(P \wedge \neg B) \Rightarrow Q$, and 2) that precondition P is true immediately before execution of the **if**-statement. In other words, given the hypotheses $\{P \wedge B\}\, S_1\, \{Q\}$ and $(P \wedge \neg B) \Rightarrow Q$, then if P is true before execution of the **if**-statement, Q will be true after execution of the **if**-statement.

Another way of saying this is that given the hypotheses $\{P \wedge B\}\, S_1\, \{Q\}$ and $(P \wedge \neg B) \Rightarrow Q$, we have proven $\{P\}\, \textbf{if}\ B\ \textbf{then}\ S_1\, \{Q\}$. This is exactly the one-way-**if** rule.

Example 43 Computing the absolute value.

Prove:

$\{x = y\}$ **if** $x < 0$ **then** $y := -x$ $\{y = abs(x)\}$.

Using the one-way-**if** rule, this reduces to:

1) $\{x = y \wedge x < 0\}\ y := -x\ \{y = abs(x)\}$, and

2) $(x = y \wedge x \geq 0) \Rightarrow y = abs(x)$.

3.4.4 The Rules of Consequence — the left consequence-rule

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

If Q guarantees that terminating executions of S end in a state satisfying R, and P implies Q, then P also guarantees that terminating executions of S end in a state satisfying R.

This rule works in the following way. Assume that the hypotheses $P \Rightarrow Q$ and $\{Q\}\,S\,\{R\}$ are both valid. $\{Q\}\,S\,\{R\}$ says that if Q is true when execution of S begins, then R will be true when (and if) execution of S ends. $P \Rightarrow Q$ says that whenever P is true, then Q will also be true. Hence, we can conclude, that if P is true when execution of S begins, then Q will also be true at that point (by validity of $P \Rightarrow Q$), and so R will be true when (and if) execution of S ends (by validity of $\{Q\}\,S\,\{R\}$). In other words, if P is true when execution of S begins, then R will be true when (and if) execution of S ends. But this is exactly $\{P\}\,S\,\{R\}$. Hence, by assuming that $P \Rightarrow Q$ and $\{Q\}\,S\,\{R\}$ are both valid, we have shown that $\{P\}\,S\,\{R\}$ is also valid. This is exactly what the left consequence-rule states.

Example 44 Prove:

$\{x \geq y\}\ z := x\ \{z = max(x, y)\}$ (*)

By the assignment axiom:

$\{x = max(x, y)\}\ z := x\ \{z = max(x, y)\}$

$x \geq y \Rightarrow x = max(x, y)$ is valid by the properties of max.

We conclude (*) by applying the left consequence-rule:

$$\frac{\begin{array}{c} x \geq y \Rightarrow x = max(x, y) \\ \{x = max(x, y)\}\ z := x\ \{z = max(x, y)\} \end{array}}{\{x \geq y\}\ z := x\ \{z = max(x, y)\}}$$

3.4.5 The Rules of Consequence — the right consequence-rule

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

If P guarantees that terminating executions of S end in a state satisfying Q, and Q implies R, then P also guarantees that terminating executions of S end in a state satisfying R.

This rule works in the following way. Assume that the hypotheses $\{P\}\, S\, \{Q\}$ and $Q \Rightarrow R$ are both valid. $\{P\}\, S\, \{Q\}$ says that if P is true when execution of S begins, then Q will be true when (and if) execution of S ends. $Q \Rightarrow R$ says that whenever Q is true, then R will also be true. Hence, we can conclude, that if P is true when execution of S begins, then Q will be true when (and if) execution of S ends (by validity of $\{P\}\, S\, \{Q\}$), and so R will also be true at that point (by validity of $Q \Rightarrow R$). In other words, if P is true when execution of S begins, then R will be true when (and if) execution of S ends. But this is exactly $\{P\}\, S\, \{R\}$. Hence, by assuming that $\{P\}\, S\, \{Q\}$ and $Q \Rightarrow R$ are both valid, we have shown that $\{P\}\, S\, \{R\}$ is also valid. This is exactly what the right consequence-rule states.

3.4.6 The Rule of Sequential Composition

$$\frac{\{P\}\, S_1\, \{R\} \qquad \{R\}\, S_2\, \{Q\}}{\{P\}\, S_1; S_2\, \{Q\}}$$

If P guarantees that R is true after execution of S_1, and R guarantees that Q is true after execution of S_2, then P guarantees that Q is true after execution of S_1 followed by execution of S_2.

This rule works in the following way. Assume that the hypotheses $\{P\}\, S_1\, \{R\}$ and $\{R\}\, S_2\, \{Q\}$ are both valid. Assume also that precondition P is true immediately before executing $S_1; S_2$. $\{P\}\, S\, \{R\}$ says that if P is true when execution of S_1 begins, then R will be true when (and if) execution of S_1 ends. Hence we know that R will in fact be true after execution of S_1, since we assume P is true before. Since S_2 follows S_1 sequentially, we conclude that R is true immediately before execution of S_2. $\{R\}\, S_2\, \{Q\}$ says that if R is true when execution of S_2 begins, then Q will be true when (and if) execution of S_2 ends. Hence we know that Q will in fact be true after execution of S_2, since we have shown that R is true before.

Our assumptions were: 1) the hypotheses $\{P\}\, S_1\, \{R\}$ and $\{R\}\, S_2\, \{Q\}$, and 2) that precondition P is true immediately before execution of $S_1; S_2$. In other words, given the hypotheses $\{P\}\, S_1\, \{R\}$ and $\{R\}\, S_2\, \{Q\}$, then if P is true before execution of $S_1; S_2$, Q will be true after execution of $S_1; S_2$.

Another way of saying this is that given the hypotheses $\{P\}\, S_1\, \{R\}$ and $\{R\}\, S_2\, \{Q\}$, we have proven $\{P\}\, S_1; S_2\, \{Q\}$. This is exactly the rule of sequential composition.

Example 45 Prove

$$\begin{array}{ll}
I(k, sum): & \{sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])\} \\
S_3: & sum := sum + a[k]; \\
S_4: & k := k + 1 \\
I(k, sum): & \{sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])\}
\end{array}$$

Work backwards from the last assignment:

$\{I(k+1, sum)\}\ S_4\ \{I(k, sum)\}$
by assignment axiom.

Also,

$\{I(k+1, sum + a[k])\}\ S_3\ \{I(k+1, sum)\}$
by assignment axiom.

Hence

$\{I(k+1, sum + a[k])\}\ S_3; S_4\ \{I(k, sum)\}$
by sequential composition rule.

Now

$$
\begin{array}{ll}
 & I(k+1, sum + a[k]) \\
\equiv & sum + a[k] = \mathbf{\Sigma}(i : 0 \le i < k+1 : a[i]) \\
\equiv & sum + a[k] = \mathbf{\Sigma}(i : 0 \le i < k : a[i]) + a[k] \\
\equiv & sum = \mathbf{\Sigma}(i : 0 \le i < k : a[i]) \\
\equiv & I(k, sum).
\end{array}
$$

Hence we finally obtain:

$\{I(k, sum)\}\ S_3; S_4\ \{I(k, sum)\}$

as desired.

3.4.7 The while Rule

$$
\frac{\{I \wedge B\}\, S\, \{I\}}{\{I\}\, \textbf{while}\ B\ \textbf{do}\ S\, \{I \wedge \neg B\}}
$$

If the truth of I is preserved by any iteration of the loop, then, if I is true initially, it will still be true upon termination of the **while**-loop. Also, the looping condition B will be false upon termination of the loop. The predicate I is called the **invariant** of the **while**-loop.

This rule works as follows. Assume that the hypothesis of the **while**-rule, namely $\{I \wedge B\}\, S\, \{I\}$, is valid. Assume also that I is true immediately before executing the **while**-loop. $\{I \wedge B\}\, S\, \{I\}$ means that if $I \wedge B$ is true before any execution of S (i.e., any iteration of the **while**-loop), then I will be true upon termination of S. Since we assume that I is true immediately before executing the **while**-loop, we conclude, by validity of $\{I \wedge B\}\, S\, \{I\}$, that I will be true after the first iteration of the loop, if first iteration is actually executed, since $I \wedge B$ will be true before the first iteration (B must be true, otherwise the first iteration would not be executed by definition of the **while**-loop). Since the end of the first iteration is also the start of the second iteration, we can also conclude that I will be true before the second iteration of the loop. Hence, if the second iteration is executed, then by validity of $\{I \wedge B\}\, S\, \{I\}$, I will be true at the end of the second iteration. Proceeding in this way, we can show that, no matter how many iterations of the loop are actually executed, I will always be true at the beginning and the end of any iteration. Now when (and if) the loop terminates, the resulting state will be the state after some iteration. Hence I will be true upon termination of the loop. Also, we know that $\neg B$ is true upon termination of the loop, since otherwise the loop would not have terminated.

Our assumptions were: 1) the hypothesis $\{I \wedge B\}\, S\, \{I\}$, and 2) that I is true immediately before execution of the **while**-loop. In other words, given the hypothesis $\{I \wedge B\}\, S\, \{I\}$, then if I is true before execution of the **while**-loop, $I \wedge \neg B$ will be true after execution of the **while**-loop.

Another way of saying this is that given the hypothesis $\{I \wedge B\}\, S\, \{I\}$, we have proven $\{I\}\, \textbf{while}\ B\ \textbf{do}\ S\, \{I \wedge \neg B\}$. This is exactly the **while** rule.

Example 46 Summing array $a[0 : n-1]$.

$$
\begin{array}{ll}
P(k, sum): & \{k = 0 \wedge sum = 0\} \\
S_1: & \textbf{while}\ B_2 : k \neq n\ \textbf{do} \\
& \qquad S_2 : sum := sum + a[k]; k := k + 1 \\
& \textbf{endwhile} \\
Q(sum): & \{sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i])\}
\end{array}
$$

B_2 is $k \neq n$, S_2 is $sum := sum + a[k]; k := k + 1$.

Invariant $I(k, sum)$ is: $sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])$.

$\{I\}\, S_2\, \{I\}$ proven previously. Hence $\{I \wedge B_2\}\, S_2\, \{I\}$ by left consequence-rule.

Hence $\{I\}\, S_1\, \{I \wedge \neg B_2\}$ by **while** rule.

$$
\begin{array}{ll}
& P \\
\equiv & k = 0 \wedge sum = 0 \\
\equiv & k = 0 \wedge sum = 0 \wedge 0 = \mathbf{\Sigma}(i : 0 \leq i < 0 : a[i]) \\
\equiv & k = 0 \wedge sum = 0 \wedge sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \\
\Rightarrow & sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \\
\equiv & I.
\end{array}
$$

Hence $P \Rightarrow I$. Hence $\{P\}\, S_1\, \{I \wedge \neg B_2\}$ by left consequence-rule.

$$
\begin{array}{ll}
& I \wedge \neg B_2 \\
\equiv & sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge k = n \\
\equiv & sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \wedge k = n \\
\Rightarrow & sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & Q.
\end{array}
$$

Hence $I \wedge \neg B_2 \Rightarrow Q$. So, $\{P\}\, S_1\, \{Q\}$ by right consequence-rule.

3.5 Proof Tableaux

A *proof tableau* is a way of summarizing a proof of correctness in a single compact form, rather than as a large number of applications of the proof rules given above. Given a program S together with its specification, expressed as a precondition P and postcondition Q, we construct a proof tableau for $\{P\}\, S\, \{Q\}$ as follows:

1. Write down the program S together with its precondition P and postcondition Q
2. For each **while**-statement **while** B **do** S' **endwhile** that occurs in S:
   (a) Find an invariant I for the **while**-statement
   (b) Write $\{I\}$ immediately before the **while**-statement
   (c) Write $\{I \wedge \neg B\}$ immediately after the **while**-statement
   (d) Write $\{I \wedge B\}$ at the top of the body of the **while**-statement (i.e., immediately before S')
   (e) Write $\{I\}$ at the bottom of the body of the **while**-statement (i.e, immediately after S')
3. For each assignment statement $x := e$ with postcondition $R(x)$, apply the assignment axiom to obtain a precondition $R(e)$
4. For each **if**-statement **if** B **then** S_1 **else** S_2 **endif** with precondition P' and postcondition Q': [1]
   (a) Write down Q' as the postcondition of both S_1 and S_2
   (b) Write down $P' \wedge B$ as the precondition of S_1, and $P' \wedge \neg B$ as the precondition of S_2
5. Repeat steps 3 through 4 until the tableau is *complete* (see definition 18 below).
6. For each pair of predicates P', P'' such that P'' immediately follows P' in the tableau (i.e., with no statement in between them), extract the *verification condition* $P' \Rightarrow P''$.

Definition 18 (*Complete Proof Tableau*)
A proof tableau is complete *iff:*

1. *The tableau contains a precondition and postcondition for every statement.*
2. *The precondition for every assignment statement in the tableau is derived from the postcondition by applying the assignment axiom.*

Definition 19 (*Valid Proof Tableau*)
A proof tableau is valid *iff:*

1. *Every Hoare-triple in the tableau is valid.*
2. *Every verification condition extracted from the tableau is valid.*

[1] We assume, for the time being, that any proof tableau we construct will supply a precondition for every **if**-statement. Hence our only problem is to provide a postcondition for each **if**-statement.

We can interpret a valid proof tableau as follows:

> Interpretation of Valid Proof Tableau
>
> If execution is started in a state that satisfies the precondition of the program, then, when program control is "at" the location of a particular predicate in the tableau, that predicate is guaranteed to be true in that program state.

In particular, if and when execution of the program terminates, then control will be "at" the postcondition, and so the postcondition will be true at that point. This is exactly what correctness of the program requires: that the postcondition be true upon termination.

3.5.1 Extended Example: Summing an Array

We shall prove that the following program is correct with respect to the precondition $P(k, sum)$ and postcondition $Q(sum)$. Here $a[0..(n-1)]$ is an array of integer. As our first step (step 1 above), we write down the program below, together with the precondition and postcondition.

$P(k, sum)$: $\{k = 0 \wedge sum = 0 \wedge n \geq 0\}$
while $B : k \neq n$ **do**
  $sum := sum + a[k];$
  $k := k + 1$
endwhile
$Q(sum)$: $\{sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i])\}$

The next step is to write the invariant in each of the four places, as given in step 2 above. We use the invariant $I(k, sum) : sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])$. This results in the following tableau.

$P(k, sum)$: $\{k = 0 \wedge sum = 0 \wedge n \geq 0\}$
$\{\text{invariant } I(k, sum) : sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])\}$
while $B : k \neq n$ **do**
  $\{I(k, sum) \wedge B\}$
  $sum := sum + a[k];$
  $k := k + 1$
  $\{I(k, sum)\}$
endwhile
$\{I(k, sum) \wedge \neg B\}$
$Q(sum)$: $\{sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i])\}$

We now apply the assignment axiom to the assignment statement $k := k + 1$ and its postcondition $I(k, sum)$, resulting in the following tableau:

$$
\begin{array}{ll}
P(k, sum): & \{k = 0 \wedge sum = 0 \wedge n \geq 0\} \\
& \{\text{invariant } I(k, sum) : sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])\} \\
& \textbf{while } B : k \neq n \textbf{ do} \\
& \qquad \{I(k, sum) \wedge B\} \\
& \qquad sum := sum + a[k]; \\
& \qquad \{I(k+1, sum)\} \\
& \qquad k := k + 1 \\
& \qquad \{I(k, sum)\} \\
& \textbf{endwhile} \\
& \{I(k, sum) \wedge \neg B\} \\
Q(sum): & \{sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i])\}
\end{array}
$$

This gives us the postcondition $I(k+1, sum)$ for the assignment statement $sum := sum + a[k]$. Hence we apply the assignment axiom again, this time to $sum := sum + a[k]$ and its postcondition $I(k+1, sum)$.

$$
\begin{array}{ll}
P(k, sum): & \{k = 0 \wedge sum = 0 \wedge n \geq 0\} \\
& \{\text{invariant } I(k, sum) : sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])\} \\
& \textbf{while } B : k \neq n \textbf{ do} \\
& \qquad \{I(k, sum) \wedge B\} \\
& \qquad \{I(k+1, sum + a[k])\} \\
& \qquad sum := sum + a[k]; \\
& \qquad \{I(k+1, sum)\} \\
& \qquad k := k + 1 \\
& \qquad \{I(k, sum)\} \\
& \textbf{endwhile} \\
& \{I(k, sum) \wedge \neg B\} \\
Q(sum): & \{sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i])\}
\end{array}
$$

The tableau is now complete. We now extract the following verification conditions (step 6 above):

1) $k = 0 \wedge sum = 0 \wedge n \geq 0 \Rightarrow I(k, sum)$

2) $I(k, sum) \wedge B \Rightarrow I(k+1, sum + a[k])$

3) $I(k, sum) \wedge \neg B \Rightarrow sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i])$

We prove that the verification conditions are valid predicates using the laws of equivalence and the rules of substitution and transitivity. For condition 1, we proceed as follows.

$$
\begin{array}{ll}
 & k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; I(k, sum) \\
\equiv & /^* \text{ replace } I(k, sum) \text{ by its definition } ^*/ \\
 & k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \\
\equiv & /^* \text{ replace } k, sum \text{ on the RHS by their values given in the LHS } ^*/ \\
 & k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; 0 = \mathbf{\Sigma}(i : 0 \leq i < 0 : a[i]) \\
\equiv & /^* \text{ summation over an empty range gives 0 } ^*/ \\
 & k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; 0 = 0 \\
\equiv & \\
 & k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; \mathrm{T} \\
\equiv & /^* \text{ any predicate of the form } p \;\Rightarrow\; \mathrm{T} \text{ is valid } ^*/ \\
 & \mathrm{T}
\end{array}
$$

Note that we write our comments (indicated by /* ...*/) in between successive predicates. This is because they are too long to be written next to the predicates, like in our proof format for propositional tautologies.

For condition 2:

$$
\begin{array}{ll}
 & I(k, sum) \wedge B \;\Rightarrow\; I(k+1, sum + a[k]) \\
\equiv & /^* \text{ replace } I(k, sum) \text{ and } B \text{ by their definitions } ^*/ \\
 & sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge k \neq n \;\Rightarrow\; sum + a[k] = \mathbf{\Sigma}(i : 0 \leq i < k+1 : a[i]) \\
\equiv & /^* \; \mathbf{\Sigma}(i : 0 \leq i < k+1 : a[i]) = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) + a[k] \; ^*/ \\
 & sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge k \neq n \;\Rightarrow\; sum + a[k] = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) + a[k] \\
\equiv & /^* \text{ subtract } a[k] \text{ from both sides of the equation on the RHS} ^*/ \\
 & sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge k \neq n \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \\
\equiv & /^* \text{ any predicate of the form } p \wedge q \;\Rightarrow\; p \text{ is valid } ^*/ \\
 & \mathrm{T}
\end{array}
$$

For condition 3:

$$
\begin{array}{ll}
 & I(k, sum) \wedge \neg B \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & /^* \text{ replace } I(k, sum) \text{ and } B \text{ by their definitions } ^*/ \\
 & sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge \neg(k \neq n) \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & /^* \text{ simplify } \neg(k \neq n) \text{ to } k = n \; ^*/ \\
 & sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge k = n \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & /^* \text{ replace } k \text{ on the LHS by its value given in the LHS } ^*/ \\
 & sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \wedge k = n \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & /^* \text{ any predicate of the form } p \wedge q \;\Rightarrow\; p \text{ is valid } ^*/ \\
 & \mathrm{T}
\end{array}
$$

This proof does not deal with one particular type of program error: violation of array bounds. To deal with this, we use the invariant $I(k, sum) : sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge 0 \leq k \leq n$, i.e., we have added $0 \leq k \leq n$ as a conjunct. The tableaux used in the proof remain essentially the same (the only difference is that the definition of the invariant changes), but the proofs of the verification conditions are a little different, so we give them below. It is instructive to compare these proofs to the ones above.

For condition 1:

$$
\begin{array}{ll}
& k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; I(k, sum) \\
\equiv & \text{/* replace } I(k, sum) \text{ by its definition */} \\
& k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge 0 \leq k \leq n \\
\equiv & \text{/* replace } k, sum \text{ on the RHS by their values given in the LHS */} \\
& k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; 0 = \mathbf{\Sigma}(i : 0 \leq i < 0 : a[i]) \wedge 0 \leq 0 \leq n \\
\equiv & \text{/* summation over an empty range gives 0 */} \\
& k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; 0 = 0 \wedge 0 \leq n \\
\equiv & \text{/* and-simplification, substitution */} \\
& k = 0 \wedge sum = 0 \wedge n \geq 0 \;\Rightarrow\; n \geq 0 \\
\equiv & \text{/* any predicate of the form } p \wedge q \;\Rightarrow\; p \text{ is valid */} \\
& \mathrm{T}
\end{array}
$$

For condition 2:

$$
\begin{array}{ll}
& I(k, sum) \wedge B \;\Rightarrow\; I(k+1, sum + a[k]) \\
\equiv & \text{/* replace } I(k, sum) \text{ and } B \text{ by their definitions */} \\
& sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge 0 \leq k \leq n \wedge k \neq n \;\Rightarrow\; sum + a[k] = \mathbf{\Sigma}(i : 0 \leq i < k+1 : a[i]) \wedge \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 0 \leq k+1 \leq n \\
\equiv & \text{/* } \mathbf{\Sigma}(i : 0 \leq i < k+1 : a[i]) = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) + a[k] \text{ */} \\
& sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge 0 \leq k < n \;\Rightarrow\; sum + a[k] = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) + a[k] \wedge -1 \leq k \leq n-1 \\
\equiv & \text{/* subtract } a[k] \text{ from both sides of the equation on the RHS*/} \\
& sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge 0 \leq k < n \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge -1 \leq k < n \\
\equiv & \text{/* } sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \text{ implies itself, and } 0 \leq k < n \text{ implies } -1 \leq k < n \text{ */} \\
& \mathrm{T}
\end{array}
$$

For condition 3:

$$
\begin{array}{ll}
& I(k, sum) \wedge \neg B \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & \text{/* replace } I(k, sum) \text{ and } B \text{ by their definitions */} \\
& sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge 0 \leq k \leq n \wedge \neg(k \neq n) \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & \text{/* simplify } \neg(k \neq n) \text{ to } k = n \text{ */} \\
& sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i]) \wedge 0 \leq k \leq n \wedge k = n \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & \text{/* replace } k \text{ on the LHS by its value given in the LHS */} \\
& sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \wedge 0 \leq k \leq n \wedge k = n \;\Rightarrow\; sum = \mathbf{\Sigma}(i : 0 \leq i < n : a[i]) \\
\equiv & \text{/* any predicate of the form } p \wedge q \;\Rightarrow\; p \text{ is valid */} \\
& \mathrm{T}
\end{array}
$$

3.5.2 Another Extended Example: Finding the Minimum Element of an Array

The following program assigns to m (upon termination) the smallest value that occurs in array a. Here $a[0..(n-1)]$ is an array of integer. The precondition is $n \geq 1$, which means that array a contains at least one element ($a[0]$). The postcondition is $m = \mathbf{MIN}(i : 0 \leq i < n : a[i])$, which states that m has the minimum value that occurs in array a.

P: $\{n \geq 1\}$
$j := 1;$
$m := a[0];$

while $B_1 : j \neq n$ **do**
  if $B_2 : m > a[j]$ **then**
    $m := a[j]$
  else
    skip
  endif;
  $j := j + 1$
endwhile
$Q(m)$: $\{m = \mathbf{MIN}(i : 0 \leq i < n : a[i])\}$

Here *skip* is a statement which has no effect (it's like a "no op"). We can think of skip as being the same as the statement $x := x$ (where x is any variable of the program under consideration). For $x := x$, the assignment axiom tells us that $\{Q(x)\}\, x := x\, \{Q(x)\}$ is valid. In other words, the precondition of *skip* is the same as its postcondition. We shall use this from now on, in effect treating *skip* as an assignment statement that leaves the variable it assigns to unchanged.

The next step is to write the invariant in each of the four places, as given in step 2 above. We use the invariant $I(j, m) : m = \mathbf{MIN}(i : 0 \leq i < j : a[i])$. This results in the following tableau.

P: $\{n \geq 1\}$
$j := 1$;
$m := a[0]$;
{invariant $I(j, m) : m = \mathbf{MIN}(i : 0 \leq i < j : a[i])$}
while $B_1 : j \neq n$ **do**
  $\{I(j, m) \wedge B_1\}$
  if $B_2 : m > a[j]$ **then**
    $m := a[j]$
  else
    skip
  endif;
  $j := j + 1$
  $\{I(j, m)\}$
endwhile
$\{I(j, m) \wedge \neg B_1\}$
$Q(m)$: $\{m = \mathbf{MIN}(i : 0 \leq i < n : a[i])\}$

Next, we apply the assignment axiom to the assignment statement $j := j + 1$ and its postcondition $I(j, m)$, resulting in the following tableau:

P: $\{n \geq 1\}$
$j := 1$;
$m := a[0]$;
{invariant $I(j, m) : m = \mathbf{MIN}(i : 0 \leq i < j : a[i])$}
while $B_1 : j \neq n$ **do**

$\{I(j,m) \land B_1\}$
if $B_2 : m > a[j]$ **then**
  $m := a[j]$
else
  skip
endif;
$\{I(j+1,m)\}$
$j := j+1$
$\{I(j,m)\}$
endwhile
$\{I(j,m) \land \neg B_1\}$
$Q(m)$: $\{m = \mathbf{MIN}(i : 0 \leq i < n : a[i])\}$

Since the **if**-statement now has a precondition (namely $I(j,m) \land B_1$) and a postcondition (namely $I(j+1,m)$), we can apply step 4 of our procedure above. This results in the following tableau.

P: $\{n \geq 1\}$
$j := 1$;
$m := a[0]$;
$\{\text{invariant } I(j,m) : m = \mathbf{MIN}(i : 0 \leq i < j : a[i])\}$
while $B_1 : j \neq n$ **do**
  $\{I(j,m) \land B_1\}$
  if $B_2 : m > a[j]$ **then**
    $\{I(j,m) \land B_1 \land B_2\}$
    $m := a[j]$
    $\{I(j+1,m)\}$
  else
    $\{I(j,m) \land B_1 \land \neg B_2\}$
    skip
    $\{I(j+1,m)\}$
  endif;
  $\{I(j+1,m)\}$
  $j := j+1$
  $\{I(j,m)\}$
endwhile
$\{I(j,m) \land \neg B_1\}$
$Q(m)$: $\{m = \mathbf{MIN}(i : 0 \leq i < n : a[i])\}$

We now apply the assignment axiom to the assignment statement $m := a[j]$ and its postcondition $I(j+1,m)$. We also write down the precondition for the *skip*, which is the same as its postcondition.

P: $\{n \geq 1\}$
$j := 1$;
$m := a[0]$;

$\{\text{invariant } I(j,m) : m = \mathbf{MIN}(i : 0 \leq i < j : a[i])\}$
while $B_1 : j \neq n$ **do**
  $\{I(j,m) \wedge B_1\}$
  if $B_2 : m > a[j]$ **then**
    $\{I(j,m) \wedge B_1 \wedge B_2\}$
    $\{I(j+1, a[j])\}$
    $m := a[j]$
    $\{I(j+1, m)\}$
  else
    $\{I(j,m) \wedge B_1 \wedge \neg B_2\}$
    $\{I(j+1, m)\}$
    skip
    $\{I(j+1, m)\}$
  endif;
  $\{I(j+1, m)\}$
  $j := j+1$
  $\{I(j,m)\}$
endwhile
$\{I(j,m) \wedge \neg B_1\}$
$Q(m)$: $\{m = \mathbf{MIN}(i : 0 \leq i < n : a[i])\}$

Next, we apply the assignment axiom to the assignment statement $m := a[0]$ and its postcondition $I(j,m)$.

P: $\{n \geq 1\}$
$j := 1;$
$\{I(j, a[0])\}$
$m := a[0];$
$\{\text{invariant } I(j,m) : m = \mathbf{MIN}(i : 0 \leq i < j : a[i])\}$
while $B_1 : j \neq n$ **do**
  $\{I(j,m) \wedge B_1\}$
  if $B_2 : m > a[j]$ **then**
    $\{I(j,m) \wedge B_1 \wedge B_2\}$
    $\{I(j+1, a[j])\}$
    $m := a[j]$
    $\{I(j+1, m)\}$
  else
    $\{I(j,m) \wedge B_1 \wedge \neg B_2\}$
    $\{I(j+1, m)\}$
    skip
    $\{I(j+1, m)\}$
  endif;
  $\{I(j+1, m)\}$
  $j := j+1$
  $\{I(j,m)\}$

$$
\begin{array}{ll}
 & \textbf{endwhile} \\
 & \{I(j,m) \wedge \neg B_1\} \\
Q(m): & \{m = \textbf{MIN}(i : 0 \leq i < n : a[i])\}
\end{array}
$$

Finally, we apply the assignment axiom to the assignment statement $j := 1$ and its postcondition $I(j, a[0])$. The tableau is now complete:

$$
\begin{array}{ll}
P: & \{n \geq 1\} \\
 & \{I(1, a[0])\} \\
 & j := 1; \\
 & \{I(j, a[0])\} \\
 & m := a[0]; \\
 & \{\text{invariant } I(j,m) : m = \textbf{MIN}(i : 0 \leq i < j : a[i])\} \\
 & \textbf{while } B_1 : j \neq n \textbf{ do} \\
 & \quad \{I(j,m) \wedge B_1\} \\
 & \quad \textbf{if } B_2 : m > a[j] \textbf{ then} \\
 & \quad\quad \{I(j,m) \wedge B_1 \wedge B_2\} \\
 & \quad\quad \{I(j+1, a[j])\} \\
 & \quad\quad m := a[j] \\
 & \quad\quad \{I(j+1, m)\} \\
 & \quad \textbf{else} \\
 & \quad\quad \{I(j,m) \wedge B_1 \wedge \neg B_2\} \\
 & \quad\quad \{I(j+1, m)\} \\
 & \quad\quad \mathit{skip} \\
 & \quad\quad \{I(j+1, m)\} \\
 & \quad \textbf{endif}; \\
 & \quad \{I(j+1, m)\} \\
 & \quad j := j + 1 \\
 & \quad \{I(j,m)\} \\
 & \textbf{endwhile} \\
 & \{I(j,m) \wedge \neg B_1\} \\
Q(m): & \{m = \textbf{MIN}(i : 0 \leq i < n : a[i])\}
\end{array}
$$

From the complete tableau, we extract the following verification conditions:

1) $n \geq 1 \Rightarrow I(1, a[0])$

2) $I(j,m) \wedge B_1 \wedge B_2 \Rightarrow I(j+1, a[j])$

3) $I(j,m) \wedge B_1 \wedge \neg B_2 \Rightarrow I(j+1, m)$

4) $I(j,m) \wedge \neg B_1 \Rightarrow m = \textbf{MIN}(i : 0 \leq i < n : a[i])$

These are proven valid as follows.

Proof of verification condition 1.

$$
\begin{array}{ll}
& n \geq 1 \;\Rightarrow\; I(1, a[0]) \\
\equiv & /*\text{ law of implication, } \neg(n \geq 1) \equiv n < 1 \text{ */} \\
& n < 1 \vee I(1, a[0]) \\
\equiv & /*\text{ replace } I(j, m) \text{ by its definition */} \\
& n < 1 \vee a[0] = \mathbf{MIN}(i : 0 \leq i < 1 : a[i]) \\
\equiv & /*\text{ singleton range rule for quantification */} \\
& n < 1 \vee a[0] = a[0] \\
\equiv & \\
& n < 1 \vee \mathrm{T} \\
\equiv & /*\text{ commutativity and or-simplification */} \\
& \mathrm{T}
\end{array}
$$

Proof of verification condition 2.

$$
\begin{array}{ll}
& I(j, m) \wedge B_1 \wedge B_2 \;\Rightarrow\; I(j+1, a[j]) \\
\equiv & /*\text{ replace } I(j, m),\ B_1, \text{ and } B_2 \text{ by their definitions */} \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m > a[j] \;\Rightarrow\; a[j] = \mathbf{MIN}(i : 0 \leq i < j+1 : a[i]) \\
\equiv & /*\text{ split the range } 0 \leq i < j+1 \text{ into } 0 \leq i < j \text{ and } i = j \text{ */} \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m > a[j] \;\Rightarrow\; a[j] = (a[j]\, min\, \mathbf{MIN}(i : 0 \leq i < j : a[i])) \\
\equiv & /*\text{ from LHS, we have } a[j] \leq \mathbf{MIN}(i : 0 \leq i < j : a[i]) \text{, hence} \\
& \quad a[j]\, min\, \mathbf{MIN}(i : 0 \leq i < j : a[i]) = a[j] \text{ by definition of } min \text{ */} \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m > a[j] \;\Rightarrow\; a[j] = a[j] \\
\equiv & \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m > a[j] \;\Rightarrow\; \mathrm{T} \\
\equiv & /*\ p \;\Rightarrow\; \mathrm{T} \text{ is valid for any } p \text{ */} \\
& \mathrm{T}
\end{array}
$$

Proof of verification condition 3.

$$
\begin{array}{ll}
& I(j, m) \wedge B_1 \wedge \neg B_2 \;\Rightarrow\; I(j+1, m) \\
\equiv & /*\text{ replace } I(j, m),\ B_1, \text{ and } B_2 \text{ by their definitions */} \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m \leq a[j] \;\Rightarrow\; m = \mathbf{MIN}(i : 0 \leq i < j+1 : a[i]) \\
\equiv & /*\text{ split the range } 0 \leq i < j+1 \text{ into } 0 \leq i < j \text{ and } i = j \text{ */} \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m \leq a[j] \;\Rightarrow\; m = (a[j]\, min\, \mathbf{MIN}(i : 0 \leq i < j : a[i])) \\
\equiv & /*\text{ from LHS, we have } m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \text{ */} \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m \leq a[j] \;\Rightarrow\; m = a[j]\, min\, m \\
\equiv & /*\text{ From LHS, we have } m \leq a[j] \text{, hence } a[j]\, min\, m = m \text{ by definition of } min \text{ */} \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m \leq a[j] \;\Rightarrow\; m = m \\
\equiv & \\
& m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j \neq n \wedge m \leq a[j] \;\Rightarrow\; \mathrm{T} \\
\equiv & /*\ p \;\Rightarrow\; \mathrm{T} \text{ is valid for any } p \text{ */} \\
& \mathrm{T}
\end{array}
$$

Proof of verification condition 4.

$$
\begin{array}{ll}
 & I(j,m) \wedge \neg B_1 \;\Rightarrow\; m = \mathbf{MIN}(i : 0 \leq i < n : a[i]) \\
\equiv & /*\text{ replace } I(j,m) \text{ and } B_1 \text{ by their definitions } */ \\
 & m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge \neg(j \neq n) \;\Rightarrow\; m = \mathbf{MIN}(i : 0 \leq i < n : a[i]) \\
\equiv & /*\text{ simplify } \neg(j \neq n) \text{ to } j = n\ */ \\
 & m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge j = n \;\Rightarrow\; m = \mathbf{MIN}(i : 0 \leq i < n : a[i]) \\
\equiv & /*\text{ replace } j \text{ on the LHS by its value given in the LHS } */ \\
 & m = \mathbf{MIN}(i : 0 \leq i < n : a[i]) \wedge j = n \;\Rightarrow\; m = \mathbf{MIN}(i : 0 \leq i < n : a[i]) \\
\equiv & /*\text{ any predicate of the form } p \wedge q \;\Rightarrow\; p \text{ is valid } */ \\
 & \mathrm{T}
\end{array}
$$

Note that we did not prove that all array references are within the array bounds. This proof is exactly analogous to the proof for the previous example, i.e., use the invariant $I(j,m) : m = \mathbf{MIN}(i : 0 \leq i < j : a[i]) \wedge 1 \leq j \leq n$ and show that the verification conditions are valid for this invariant.

3.6 Total Correctness of Programs: The Notation $\langle P\rangle\, S\, \langle Q\rangle$

So far, we have concerned ourselves with conditional correctness only: if a program terminates, then the final state will satisfy the postcondition. It is also crucial to prove that the program does in fact terminate. Towards this end we define the notation $\langle P\rangle\, S\, \langle Q\rangle$ to have the following meaning:

> If execution of S is started in a state satisfying P, then
> execution of S does in fact terminate, and the final state is guaranteed to satisfy Q.

Because termination is *guaranteed*, this is called *total correctness*. For $\langle P\rangle\, S\, \langle Q\rangle$ to have the meaning given above, we define the validity of $\langle P\rangle\, S\, \langle Q\rangle$ as follows.

Definition 20 (*Validity of* $\langle P\rangle\, S\, \langle Q\rangle$)
$\langle P\rangle\, S\, \langle Q\rangle$ *is* valid *iff*

For every state s such that $s(P) = \mathrm{T}$:
If execution of S is started in s, then:
the execution terminates in some state t such that $t(Q) = \mathrm{T}$

3.6.1 Specifying Termination Only

Total correctness requires two things: 1) the program terminates, and 2) the final state satisfies the postcondition. It is usually easier to prove each of these properties separately. We already know how to express (2), it is just conditional correctness (see definition 17).

To express (1), we use $\langle P\rangle\, S\, \langle \mathrm{T}\rangle$, which states that:

> *If execution of S is started in a state satisfying P, then execution of S terminates in a final state that is guaranteed to satisfy* T.

In other words, no constraint is made on the final state, (since any state whatsoever satisfies T). Hence, the only requirement is termination.

3.6.2 Relating Total Correctness, Conditional Correctness, and Termination

There is an important relationship between total correctness, conditional correctness, and termination. To see it, we first restate these as follows:

Conditional Correctness: If execution of S is started in a state satisfying P, then if execution of S terminates, the final state is guaranteed to satisfy Q.

Termination: If execution of S is started in a state satisfying P, then execution of S terminates.

Conditional Correctness + Termination: If execution of S is started in a state satisfying P, then execution of S terminates in a final state that is guaranteed to satisfy Q.

Comparing the statement of "conditional correctness + termination" above with that of total correctness (definition 20), we see that they are the same. We summarize this as the mnemonic equation:

$$\text{total correctness} = \text{conditional correctness} + \text{termination}$$

In terms of Hoare triples, we can write this as follows:

$$\langle P\rangle\, S\, \langle Q\rangle \quad \equiv \quad \langle P\rangle\, S\, \langle \mathrm{T}\rangle \;\wedge\; \{P\}\, S\, \{Q\}$$

3.6.3 Proving Termination: The Proof Rule for Termination of while-loops

From our informal understanding of how our programs are executed, we easily see that the only source of non-termination is the **while**-loop. That is, if a program fails to terminate, the only possible reason is that some **while**-loop in the program is "stuck" and is being executed forever. Hence, to prove termination, we only need to introduce one more proof rule, which is the following:

$$\frac{\begin{array}{c}\{I \wedge B\}\, S\, \{I\},\\ I \wedge B \Rightarrow \varphi \geq 0,\\ \langle I \wedge B \wedge \varphi = C\rangle\, S\, \langle \varphi < C\rangle\end{array}}{\langle I\rangle\ \textbf{while}\ B\ \textbf{do}\ S\, \langle \mathrm{T}\rangle}$$

φ is an integer-valued function of the program variables, called the *variant function*. $\{I \wedge B\}\, S\, \{I\}$ states that I is an invariant of the loop (see subsection 3.4.7 for a detailed discussion of this). Given that I is in fact an invariant of the loop, $I \wedge B \Rightarrow \varphi \geq 0$ states that φ is always positive at the beginning of each loop iteration, and $\langle I \wedge B \wedge \varphi = C\rangle\, S\, \langle \varphi < C\rangle$ states that each iteration of the loop terminates and decreases φ.

This rule works as follows. From $I \wedge B \Rightarrow \varphi \geq 0$, we know that φ must be positive at the beginning of every loop iteration. From $\langle I \wedge B \wedge \varphi = C\rangle\, S\, \langle \varphi < C\rangle$, we know that every iteration terminates and decreases the value of φ. Now suppose the loop does not terminate. Then φ must be decreased infinitely often. Any integer quantity that is decreased infinitely often must eventually become negative. Hence φ eventually becomes negative. But this contradicts $I \wedge B \Rightarrow \varphi \geq 0$. Hence, we conclude that the loop must terminate.

3.6.4 Proof Tableaux for Termination

To prove that a program terminates, we construct a proof tableau similar to tableaux for conditional correctness, but we use $\langle P\rangle$ instead of $\{P\}$ for the predicates that are inserted into the tableau. We can interpret a valid proof tableau for termination as follows:

> Interpretation of Valid Proof Tableau for Termination
>
> If execution is started in a state that satisfies the precondition of the program, then, when program control is "at" the location of a particular predicate in the tableau, that predicate is guaranteed to be true in that program state.
>
> Also, for every Hoare triple $\langle P\rangle\, S\, \langle Q\rangle$ in the tableau, if control reaches P, then control is guaranteed to eventually reach Q. (This guarantees termination.)

Example 47 Termination of factorial program.

$P(n)$: $\langle n \geq 0\rangle$
        $\langle I(0)\rangle$
S_1: $k := 0;$
        $\langle I(k)\rangle$
S_2: $f := 1;$
        $\langle$ invariant $I(k)$: $0 \leq k \leq n\rangle$
        /* variant $\varphi(k)$: $n - k$ */
S_3: **while** $B : k \neq n$ **do**
                $\langle I(k) \wedge k \neq n \wedge \varphi(k) = C\rangle$
                $\langle I(k+1) \wedge \varphi(k+1) < C \wedge \varphi(k) \geq 0\rangle$
                $k := k + 1;$
                $\langle I(k) \wedge \varphi(k) < C\rangle$
                $f := f * k;$
                $\langle I(k) \wedge \varphi(k) < C\rangle$
        endwhile
        $\langle \mathrm{T}\rangle$

Verification conditions:

1) $n \geq 0 \Rightarrow I(0)$

2) $I(k) \wedge B \wedge \varphi(k) = C \Rightarrow I(k+1) \wedge \varphi(k+1) < C \wedge \varphi(k) \geq 0$

Proving (2) establishes $\{I(k) \wedge B\}\, S\, \{I(k)\}$ and $\langle I(k) \wedge B \wedge \varphi(k) = C\rangle\, S\, \langle\varphi(k) < C\rangle$ and $I \wedge B \Rightarrow \varphi(k) \geq 0$ (where S = "$k := k+1; f := f * k$" is the loop body).

Once (2) is proven, we can apply the proof rule for termination of **while**-loops, and conclude $\langle I(k)\rangle\, S_3\, \langle \mathrm{T}\rangle$. Together with (1), this gives us $\langle n \geq 0\rangle\, S_1; S_2; S_3\, \langle \mathrm{T}\rangle$.

Proof of (1):

$$
\begin{array}{ll}
 & n \geq 0 \Rightarrow I(0) \\
\equiv & n \geq 0 \Rightarrow 0 \leq 0 \leq n \quad \text{/* replace } I(0) \text{ by its definition */} \\
\equiv & n \geq 0 \Rightarrow 0 \leq n \\
\equiv & \mathrm{T}
\end{array}
$$

Proof of (2):

$$
\begin{array}{ll}
 & I(k) \wedge B \wedge \varphi(k) = C \;\Rightarrow\; I(k+1) \wedge \varphi(k+1) < C \wedge \varphi(k) \geq 0 \\
\equiv & \qquad \text{/* replace } I, \varphi \text{ by their definitions */} \\
 & 0 \leq k \leq n \wedge k \neq n \wedge n-k = C \;\Rightarrow\; 0 \leq k+1 \leq n \wedge n-(k+1) < C \wedge n-k \geq 0 \\
\equiv & \\
 & 0 \leq k < n \wedge n-k = C \;\Rightarrow\; -1 \leq k \leq n-1 \wedge (n-k)-1 < C \wedge n-k \geq 0 \\
\equiv & \\
 & 0 \leq k < n \wedge n-k = C \;\Rightarrow\; -1 \leq k < n \wedge C-1 < C \wedge n-k \geq 0 \\
\equiv & \\
 & \mathrm{T}
\end{array}
$$

The procedure to construct a proof tableau for termination is the same as that for conditional correctness (section 3.5), except that step 2 is replaced by the following:

2. For each **while**-statement **while** B **do** S' **endwhile** that occurs in S:
   (a) Find an invariant I for the **while**-statement
   (b) Write $\langle I\rangle$ immediately before the **while**-statement
   (c) Write $\langle I \wedge \neg B\rangle$ immediately after the **while**-statement
   (d) Write $\langle I \wedge B \wedge \varphi = C\rangle$ at the top of the body of the **while**-statement (i.e., immediately before S')
   (e) Include $\varphi \geq 0$ as a conjunct of the predicate occurring immediately below $\langle I \wedge B \wedge \varphi = C\rangle$
   (f) Write $\langle I \wedge \varphi < C\rangle$ at the bottom of the body of the **while**-statement (i.e, immediately after S')

3.7 Procedures

Our programming language (see section 3.1) so far lacks the facility of defining procedures. We now remedy this deficiency by extending our programming language with procedures.

The syntax of procedure declaration and invocation is as follows:

Procedure declaration:
$\quad$ **procedure** $pname$(**value** $\overline{fv}$; **value−result**$\overline{fr}$) : $pbody$

Procedure invocation:
$\quad$ **call** $pname(\overline{ave}, \overline{ar})$

$\overline{fv}, \overline{fr}, \overline{ar}$ are variable lists
$\overline{ave}$ is an expression list

Procedures take two types of parameters: *value parameters* (denoted by the keyword **value**), and *value-result parameters* (denoted by the keyword **value−result**). Value parameters are treated as constants within the procedure body, i.e., they cannot be changed. They are used only to pass values into the procedure. Value-result paramenters can be changed in the procedure body. They are used both to pass values into the procedure and to return values computed by the procedure to the invoking program.

The parameters that are used to write the procedure declaration are called *formal parameters.* Formal parameters can be further subdivided into formal value parameters (given by the list $\overline{fv}$ of variables) and formal value-result parameters (given by the list $\overline{fr}$ of variables). The parameters that are passed to the procedure in a procedure invocation are called *actual parameters.* Actual parameters can be further subdivided into actual value parameters (given by the list $\overline{ave}$ of expressions) and actual value-result parameters (given by the list $\overline{ar}$ of variables). Note that since the formal value parameters do not return a value to the invoking program, the actual value parameters can be expressions instead of variables, since we do not need a variable to return the changed value.

3.7.1 Proving Conditional Correctness of Procedures

The basic principle in proving conditional correctness of procedures is as follows:

- Prove conditional correctness of the procedure body (in terms of the formal parameters).
- Replace the formal parameters by the actual parameters to conclude the conditional correctness of procedure invocations.

Effectively, this just "simulates" what happens when a procedure is invoked — the formals get replaced by the actuals. Although this strategy works in general, problems can arise in certain peculiar situations:

Example 48 Actual parameters are not distinct (aliasing).

procedure $inc1$(**value** x;**value−result** y):
$\quad$ $\{\mathrm{T}\}$
$\quad$ $y := x + 1$
$\quad$ $\{y = x + 1\}$

Replacing the formals by the actuals, we conclude $\{\mathrm{T}\}\,\mathbf{call}\ inc1(n,n)\,\{n = n+1\}$. Obviously this is invalid, since $(n = n+1) \equiv \mathrm{F}$.

Another problematic situation is:

Example 49 Formal parameters are not distinct.

$$
\begin{array}{l}
\textbf{procedure}\ copy\,(\textbf{value}\ x, x; \textbf{value}-\textbf{result}\ y):\\
\qquad \{\mathrm{T}\}\\
\qquad y := x\\
\qquad \{y = x\}
\end{array}
$$

Replacing the formals by the actuals, we conclude $\{\mathrm{T}\}\,\mathbf{call}\ copy(a,b,c)\,\{c = ?\}$. Should ? be a or b. We don't know — the final value of y is not well-defined.

To avoid problems such as those illustrated above, we make the following assumptions:

- The formal and actual parameters match with respect to number and type.
- The formal parameters are pairwise distinct (and hence the formal parameters have well-defined initial values).
- The actual parameters are pairwise distinct (no aliasing).
- The value parameters are not changed in *pbody*.
- All variables other than formal parameters are local to the procedure (i.e., no global variables).
- No mutual recursion (although simple recursion, where a procedure invokes itself, will be dealt with).

Conditional Correctness of Nonrecursive Procedures

The rule for proving conditional correctness of nonrecursive procedures is as follows.

$$
\frac{\{P(\overline{fv},\overline{fr})\}\,pbody\,\{Q(\overline{fv},\overline{fr})\}}{\{P(\overline{ave},\overline{ar})\}\,\mathbf{call}\ pname(\overline{ave},\overline{ar})\,\{Q(\overline{ave},\overline{ar})\}}
$$

This states that if *pbody* is conditionally correct with respect to precondition $P(\overline{fv},\overline{fr})$ and postcondition $Q(\overline{fv},\overline{fr})$, then the procedure invocation **call** $pname(\overline{ave},\overline{ar})$ is conditionally correct with respect to precondition $P(\overline{ave},\overline{ar})$ and postcondition $Q(\overline{ave},\overline{ar})$ (i.e., P and Q with the formal parameters replaced by actual parameters).

Example 50 Add 1 to a given value.

procedure $inc1$(**value** x; **value−result** y):
P: $\{\mathrm{T}\}$
$y := x + 1$
$Q(x, y)$: $\{y = x + 1\}$

Applying the proof rule, we conclude $\{P\}$ **call** $inc1(m, n)\ \{Q(m, n)\}$.

Replacing P, Q by their definitions, we get $\{\mathrm{T}\}$ **call** $inc1(m, n)\ \{n = m + 1\}$.

Example 51 Find the minimum of two values.

procedure min(**value** x, y; **value−result** z):
P: $\{\mathrm{T}\}$
if $x \le y$ **then** $z := x$ **else** $z := y$ **endif**
$Q(x, y, z)$: $\{z = min(x, y)\}$

Applying the proof rule, we conclude $\{P\}$ **call** $min(a, b, c)\ \{Q(a, b, c)\}$

Replacing P, Q by their definitions, we get $\{\mathrm{T}\}$ **call** $min(a, b, c)\ \{c = min(a, b)\}$

Example 52 Compute the factorial.

procedure $fact$(**value** n; **value−result** f):
$P(n)$: $\{n \ge 0\}$
$\{I(0, 1)\}$
$k := 0;$
$\{I(k, 1)\}$
$f := 1;$
$\{\text{invariant } I(k, f)\text{: } f = k!\}$
while $k \ne n$ **do**
  $\{I(k, f) \wedge k \ne n\}$
  $\{I(k + 1, f * (k + 1))\}$
  $k := k + 1;$
  $\{I(k, f * k)\}$
  $f := f * k;$
  $\{I(k, f)\}$
endwhile
$\{f = k! \wedge k = n\}$
$Q(n, f)$: $\{f = n!\}$

Applying the proof rule, we conclude $\{P(a)\}$ **call** $fact(a, b)\ \{Q(a, b)\}$.

Replacing P, Q by their definitions, we get $\{a \ge 0\}$ **call** $fact(a, b)\ \{b = a!\}$.

Dealing with Initial Values of Parameters

Value parameters cannot be changed — their value is always the initial value. Value-result parameters can be changed, and so their value at some point in the procedure body is not necessarily the initial value. In many situations (e.g., incrementing a variable, sorting an array) we need to be able to relate the final value of a value-result parameter to its initial value in order to specify correctness as a precondition and postcondition. We shall do this by recording the initial value of the value-result parameter in an "upper case" variable (that is not changed). This variable is then passed to the procedure as a value parameter.

Example 53 Increment a variable.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{procedure } inc2(\textbf{value } Y; \textbf{value}-\textbf{result } y):} \\
P(Y,y): & \{y = Y\} \\
 & y := y + 1 \\
Q(Y,y): & \{y = Y + 1\}
\end{array}
$$

Applying the proof rule, we conclude $\{P(Y,y)\}\,\textbf{call}\ inc2(Y,y)\,\{Q(Y,y)\}$.

Replacing P, Q by their definitions, we get $\{y = Y\}\,\textbf{call}\ inc2(Y,y)\,\{y = Y + 1\}$.

These extra value parameters are never referenced in code, (see above example) i.e., they don't affect execution. In the actual program, they can be omitted. Since these variables are used only to carry out the proof, and not to affect program execution, they are called *ghost*, or *auxiliary* variables.

Conditional Correctness of Recursive Procedures

The rule for proving conditional correctness of recursive procedures is as follows.

$$
\frac{\begin{array}{l}\{P(\overline{ave'},\overline{ar'})\}\,\textbf{call}\ pname(\overline{ave'},\overline{ar'})\,\{Q(\overline{ave'},\overline{ar'})\} \\ \vdash \\ \{P(\overline{fv},\overline{fr})\}\,pbody\,\{Q(\overline{fv},\overline{fr})\}\end{array}}{\{P(\overline{ave},\overline{ar})\}\,\textbf{call}\ pname(\overline{ave},\overline{ar})\,\{Q(\overline{ave},\overline{ar})\}}
$$

$H1 \vdash H2$ means $H2$ can be proven assuming $H1$ ($H1, H2$ are Hoare triples)

This states that if we can prove that *pbody* is conditionally correct with respect to precondition $P(\overline{fv},\overline{fr})$ and postcondition $Q(\overline{fv},\overline{fr})$ by assuming that all recursive invocations in *pbody* are conditionally correct, then we can conclude that the invocation **call** $pname(\overline{ave},\overline{ar})$ is conditionally correct with respect to precondition $P(\overline{ave},\overline{ar})$ and postcondition $Q(\overline{ave},\overline{ar})$ (i.e., P and Q with the formal parameters replaced by actual parameters).

In effect, we are doing induction on the "tree" of procedure invocations.

Example 54 Compute the factorial recursively.

procedure $rfact$(**value** n; **value−result** f):
$P(n)$: $\{n \geq 0\}$
S: **if** $n = 0$ **then**
  $\{n \geq 0 \wedge n = 0\}$
  $\{1 = n!\}$
  $f := 1;$
  $\{f = n!\}$
else
  $\{n \geq 0 \wedge n \neq 0\}$
    $\{n - 1 \geq 0\}$
    call $rfact(n-1, f)$;
    $\{f = (n-1)!\}$
  $\{f * n = n!\}$
  $f := f * n$
  $\{f = n!\}$
endif
$Q(n,f)$: $\{f = n!\}$

From the above tableau, we conclude:

$$\{a \geq 0\}\,\textbf{call}\ rfact(a,b)\,\{b = a!\}$$
$$\vdash$$
$$\{n \geq 0\}\,S\,\{f = n!\}$$

Here, the recursive invocation (shown indented) is with actual parameters $a = n - 1$, $b = f$.

Hence, applying the proof rule, we conclude $\{c \geq 0\}\,\textbf{call}\ rfact(c,d)\,\{c = d!\}$.

Example 55 Conditional correctness of procedure $msort$ (mergesort).

procedure $msort$(**value** A, n; **value−result** a):
/* sort array $a[0..(n-1)]$ */
  n: integer;
  $A, a : array[0..n-1]$ of integer
$P(A,n,a)$: $\{a = A \wedge n \geq 0\}$
S: **if** $n = 0 \vee n = 1$ **then** $skip$
  $\{sorted(a, A, n)\}$
else
  $\{n \geq 2\}$
  $mid := \lceil n/2 \rceil;$
  $\{1 \leq mid \leq n-1\}$
  $b := a[0..(mid-1)];$

$B := b;$
$\{b = B \wedge mid \geq 0\}$
call $msort(B, mid, b);$
$\{sorted(b, B, mid)\}$
$c := a[mid..(n-1)];$
$C := c;$
$\{c = C \wedge n - mid \geq 0\}$
call $msort(C, n - mid, c);$
$\{sorted(c, C, n - mid)\}$
$\{sorted(b, B, mid) \wedge sorted(c, C, n - mid) \wedge merged(B, C, A, mid, n - mid, n)\}$
call $merge(B, C, A, mid, n - mid, n, b, c, a);$
$\{sorted(a, A, n)\}$
endif
$\{sorted(a, A, n)\}$

where

$sorted(a, b, n) \;\equiv\; perm(a, b, n) \wedge ordered - nondec(a, n)$

$ordered - nondec(a, n) \equiv \forall(i : 0 \leq i < n - 1 : a[i] \leq a[i+1])$

$perm(a, b, n) \;\equiv\; \forall(i : 0 \leq i < n : num(a, a[i], n) = num(b, a[i], n))$

$num(c, x, n) = \mathbf{N}(i : 0 \leq i < n : c[i] = x)$

$merged(b, c, a, \ell, m, n) \equiv$
/* ℓ, m, n are the sizes of b, c, a respectively */
$\forall(i : 0 \leq i < n : num(a, a[i], n) = num(b, a[i], \ell) + num(c, a[i], m)$

$merge(B, C, A, mid, n - mid, n, b, c, a)$ is a procedure that takes two arrays b, c that are sorted in non-decreasing order and merges them into an array a that is also sorted in non-decreasing order.

3.7.2 Proving Termination of Procedures

In proving termination of procedures, we follow a similar strategy to proving conditional correctness of procedures, i.e.:

- Prove termination of the procedure body (in terms of the formal parameters).
- Replace the formal parameters by the actual parameters to conclude termination of the procedure invocation.

There are two main differences from the method for proving conditional correctness. First, the postcondition is simply T, since we do not care about the actual final state. Second, we construct a tableau for termination rather than a tableau for partial correctness (the tableau is constructed for the procedure body and its pre/post-conditions).

Proving Termination of Nonrecursive Procedures

The rule for proving termination of nonrecursive procedures is as follows.

$$\frac{\langle P(\overline{fv},\overline{fr})\rangle\, pbody\, \langle \mathrm{T}\rangle}{\langle P(\overline{ave},\overline{ar})\rangle\, \mathbf{call}\ pname(\overline{ave},\overline{ar})\, \langle \mathrm{T}\rangle}$$

This states that if *pbody* terminates with respect to precondition $P(\overline{fv},\overline{fr})$, then the procedure invocation **call** $pname(\overline{ave},\overline{ar})$ terminates with respect to precondition $P(\overline{ave},\overline{ar})$ (i.e., P with the formal parameters replaced by actual parameters).

Example 56 Procedure to compute the factorial.

procedure $fact$(**value** n; **value−result** f):
$P(n)$: $\langle n \geq 0\rangle$
$\langle I(0)\rangle$
$k := 0;$
$\langle I(k)\rangle$
$f := 1;$
$\langle$ invariant $I(k)$: $0 \leq k \leq n\rangle$
/* variant $\varphi(k)$: $n - k$ */
while $B : k \neq n$ **do**
  $\langle I(k) \wedge k \neq n \wedge \varphi(k) = C\rangle$
  $\langle I(k+1) \wedge \varphi(k+1) < C \wedge \varphi(k) \geq 0\rangle$
  $k := k + 1;$
  $\langle I(k) \wedge \varphi(k) < C\rangle$
  $f := f * k;$
  $\langle I(k) \wedge \varphi(k) < C\rangle$
endwhile
$\langle \mathrm{T}\rangle$

Verification conditions:

1) $n \geq 0 \Rightarrow I(0)$

2) $I(k) \wedge B \wedge \varphi(k) = C \Rightarrow I(k+1) \wedge \varphi(k+1) < C \wedge \varphi(k) \geq 0$

Proving (2) establishes $\{I(k) \wedge B\}\, S\, \{I(k)\}$ and $\langle I(k) \wedge B \wedge \varphi(k) = C\rangle\, S\, \langle \varphi(k) < C\rangle$ and $I(k) \wedge B \Rightarrow \varphi(k) \geq 0$ (where $S =$ "$k := k+1; f := f * k$" is the loop body). Once these are proven, we can apply the proof rule for termination of **while**-loops, and conclude $\langle I(k)\rangle$ **while** B **do** $S\,\langle \mathrm{T}\rangle$. Together with (1), this gives us $\langle n \geq 0\rangle\, fact{-}body\, \langle \mathrm{T}\rangle$ (where $fact{-}body$ is the body of procedure $fact$). Given that $\langle n \geq 0\rangle\, fact{-}body\, \langle \mathrm{T}\rangle$ is valid, we then apply the proof rule for termination of nonrecursive procedures, and conclude $\langle a \geq 0\rangle$ **call** $fact(a, b)\, \langle \mathrm{T}\rangle$, i.e., all invocations with actual parameter a non-negative terminate.

Proving Termination of Recursive Procedures

The rule for proving termination of recursive procedures is as follows.

$$\frac{\begin{array}{l}\langle 0 \le \varphi(\overline{ave'}, \overline{ar'}) < C \rangle \,\textbf{call}\; pname(\overline{ave'}, \overline{ar'})\, \langle \mathrm{T} \rangle \\ \vdash \\ \langle 0 \le \varphi(\overline{fv}, \overline{fr}) = C \rangle \, pbody \, \langle \mathrm{T} \rangle\end{array}}{\langle 0 \le \varphi(\overline{ave}, \overline{ar}) \rangle \,\textbf{call}\; pname(\overline{ave}, \overline{ar})\, \langle \mathrm{T} \rangle}$$

$H1 \vdash H2$ means $H2$ can be proven assuming $H1$ ($H1, H2$ are Hoare triples). φ is a "variant function" over the parameters of the procedure.

This states that if we can prove that *pbody* terminates by assuming that all recursive invocations in *pbody* with a smaller non-negative variant φ terminate, then we can conclude that all invocations with non-negative variant φ terminate.

In effect, we are doing induction on the "tree" of procedure invocations.

Example 57 Termination of procedure $rfact$.

$$\begin{array}{lllll}
\textbf{procedure}\; rfact(\textbf{value}\; n; \textbf{value}-\textbf{result}\; f): \\
 & /*\text{ variant } \varphi(n)\text{: } n\ */ \\
P(n): & \langle 0 \le n = \varphi(n) = C \rangle \\
S: & \textbf{if}\; n = 0\; \textbf{then} \\
 & \quad f := 1; \\
 & \quad \langle \mathrm{T} \rangle \\
 & \textbf{else} \\
 & \quad \langle n \ge 0 \wedge n \ne 0 \rangle \\
 & \quad\quad \langle 0 \le n - 1 = \varphi(n-1) < C \rangle \\
 & \quad\quad \textbf{call}\; rfact(n-1, f); \\
 & \quad\quad \langle \mathrm{T} \rangle \\
 & \quad f := f * n \\
 & \quad \langle \mathrm{T} \rangle \\
 & \textbf{endif} \\
 & \langle \mathrm{T} \rangle
\end{array}$$

From the above tableau, we conclude:

$$\begin{array}{l}\langle 0 \le a = \varphi(a) < C \rangle \,\textbf{call}\; rfact(a, b)\, \langle \mathrm{T} \rangle \\ \vdash \\ \langle 0 \le n = \varphi(n) = C \rangle \, S \, \langle \mathrm{T} \rangle\end{array}$$

Hence, applying the proof rule, we conclude $\langle 0 \le c = \varphi(c) \rangle \,\textbf{call}\; rfact(c, d)\, \langle \mathrm{T} \rangle$.

Example 58 Termination of procedure $msort$ (mergesort).

procedure $msort$(**value** A, n; **value−result** a):
/* sort array $a[0..(n-1)]$ */
    n: integer;
    $A, a : array[0..n-1]$ of integer
    /* variant $\varphi(n) : n$ */
$P(n)$: $\langle n \geq 0 \rangle$
    $\langle 0 \leq \varphi(n) = n = D \rangle$
S: **if** $n = 0 \vee n = 1$ **then** $skip$ $\langle \mathrm{T} \rangle$
    else
        $\langle n \geq 2 \rangle$
        $mid := \lceil n/2 \rceil$;
        $\langle 1 \leq mid \leq n-1 \rangle$
        $b := a[0..(mid-1)]$;
        $B := b$;
            $\langle 0 \leq \varphi(mid) = mid < D \rangle$
            call $msort(B, mid, b)$;
            $\langle \mathrm{T} \rangle$
        $c := a[mid..(n-1)]$;
        $C := c$;
            $\langle 0 \leq \varphi(n - mid) = n - mid < D \rangle$
            call $msort(C, n - mid, c)$;
            $\langle \mathrm{T} \rangle$
        $\langle n \geq 0 \wedge mid \geq 0 \wedge n - mid \geq 0 \rangle$
        call $merge(B, C, A, mid, n - mid, n, b, c, a)$;
        $\langle \mathrm{T} \rangle$
    endif
    $\langle \mathrm{T} \rangle$

3.8 Arrays

Let $a[0..(n-1)]$ be an array with index range 0 to $n-1$, and elements of type TP. We regard a as a function from integers in $0..(n-1)$ to values in TP. We then define $(a; i : e)$ as follows.

$$(a; i : e)[j] = \begin{cases} e & \text{if } i = j \\ a[j] & \text{if } i \neq j \end{cases}$$

$(a; i : e)$ is an array that is identical to a except for index i, where it returns e instead of $a[i]$.

Example 59 Evaluation of $(a; i : e)$

$$
\begin{array}{ll}
 & (a; i : 5)[j] = 5 \\
\equiv & \\
 & (i = j \wedge 5 = 5) \vee (i \neq j \wedge a[j] = 5) \quad \text{/* two-case analysis on value of } i \text{ */} \\
\equiv & \\
 & (i = j) \vee (i \neq j \wedge a[j] = 5) \\
\equiv & \\
 & (i = j \vee i \neq j) \wedge (i = j \vee a[j] = 5) \quad \text{/* distribution */} \\
\equiv & \\
 & (i = j \vee a[j] = 5) \quad \text{/* excluded middle, and-simplification */}
\end{array}
$$

Example 60 Evaluation of $(a; i : e)$

$$
\begin{array}{ll}
 & (a; i : 5)[j] = 6 \\
\equiv & \\
 & (i = j \wedge 5 = 6) \vee (i \neq j \wedge a[j] = 6) \quad \text{/* two-case analysis on value of } i \text{ */} \\
\equiv & \\
 & \mathrm{F} \vee (i \neq j \wedge a[j] = 5) \\
\equiv & \\
 & i \neq j \wedge a[j] = 6 \quad \text{/* or-simplification */}
\end{array}
$$

3.8.1 Assignment Axiom for Arrays

By regarding arrays as functions from index values to element values, we can think of an array as a simple variable whose type happens to be "function." We then see that the assignment $a[i] := e$ is in fact the same as the assignment $a := (a; i : e)$: changing $a[i]$ to e is the same as changing a to a new array that is the same as a in all indices except i, where it returns e. Since arrays are just simple variables, we can use the assignment axiom (subsection 3.4.1). Applying the assignment axiom to array assignment gives us the following assignment axiom for arrays:

$$\{Q((a; i : e))\}\, a[i] := e\, \{Q(a)\} \text{ is valid.}$$

a has the value after execution that $(a; i : e)$ has before, so $Q(a)$ is true after iff $Q((a; i : e))$ is true before.

The next four examples present and simplify valid Hoare triples that are deduced from the assignment axiom for arrays.

Example 61 $\{(a; i : 5)[i] = 5\}\, a[i] := 5\, \{a[i] = 5\}$.

Example 62 $\{(a; i : (a[i] + 1))[i] \leq 5\}\, a[i] := a[i] + 1\, \{a[i] \leq 5\}$.

$$
\begin{array}{ll}
 & (a; i : (a[i] + 1))[i] \leq 5 \\
\equiv & \\
 & a[i] + 1 \leq 5 \\
\equiv & \\
 & a[i] \leq 4
\end{array}
$$

hence,

$\{a[i] \leq 4\}\, a[i] := a[i] + 1\, \{a[i] \leq 5\}$.

Example 63 $\{(a; i : 5)[i] = (a; i : 5)[j]\}\, a[i] := 5\, \{a[i] = a[j]\}$.

$$
\begin{array}{ll}
 & (a; i : 5)[i] = (a; i : 5)[j] \\
\equiv & \\
 & 5 = (a; i : 5)[j] \\
\equiv & \\
 & (i = j \wedge 5 = 5) \vee (i \neq j \wedge 5 = a[j]) \quad \text{/* case analysis on } (a; i : 5)[j] \\
\equiv & \\
 & (i = j) \vee (i \neq j \wedge 5 = a[j]) \\
\equiv & \\
 & (i = j \vee i \neq j) \wedge (i = j \vee 5 = a[j]) \quad \text{/* distribution */} \\
\equiv & \\
 & i = j \vee a[j] = 5 \quad \text{/* excluded middle, and-simplification */}
\end{array}
$$

hence

$\{i = j \vee a[j] = 5\}\, a[i] := 5\, \{a[i] = a[j]\}$.

Example 64 $\{(a; a[i] : i)[i] = i\}\, a[a[i]] := i\, \{a[i] = i\}$.

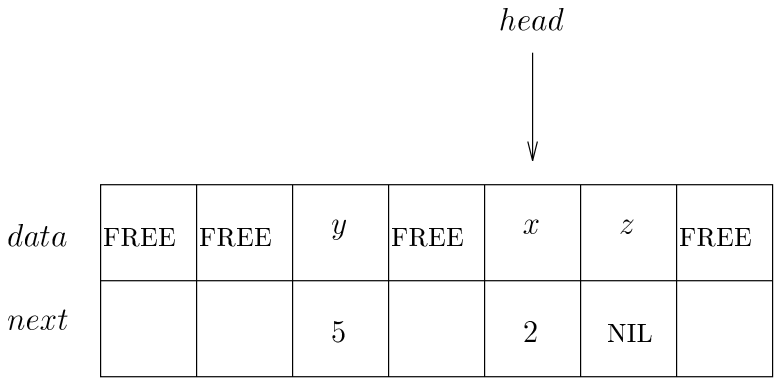
$$
\begin{array}{ll}
 & (a; a[i] : i)[i] = i \\
\equiv & \\
 & (a[i] = i \wedge i = i) \vee (a[i] \neq i \wedge a[i] = i) \quad \text{/* case analysis on } (a; a[i] : i)[i] \text{ */} \\
\equiv & \\
 & (a[i] = i \wedge i = i) \quad \text{/* contradiction, or-simplification */} \\
\equiv & \\
 & a[i] = i \quad \text{/* and-simplification */}
\end{array}
$$

hence

$\{a[i] = i\}\, a[a[i]] := i\, \{a[i] = i\}$.

3.8.2 Implementing Linked Lists Using Arrays

We discuss the implementation of a singly linked list using arrays. We use two arrays: $data[0..(n-1)]$ and $next[0..(n-1)]$. $data$ contains the data items in the linked list, and $next$ contains the pointers to the next item. Also, the index $head$ gives the index of the first element of the list.



Linked list is x, y, z,NIL

We show how the insertion and deletion operations on a linked list can be formally specified, and verify the correctness of an implementation of the insert operation.

Specifying and Verifying Insertion into Linked Lists

$insert($**value** $item, pos;$ **value−result** $data, next, head)$
/* insert $item$ into linked list stored in arrays $data$, $next$ with first element $data[head]$. Element pos is used to store x. */

precondition: $list(head, data, next) = L \ \wedge \ free(pos, data)$

postcondition: $list(head, data, next) = item \bullet L \ \wedge \ data[head] = item$

where

$list(head, data, next) \ = \ data[head] \bullet list(next[head], data, next)$
$list(NIL, data, next) = \lambda$ (empty sequence)
$free(pos, data) \equiv (data[pos] = \text{“}FREE''})$
$\bullet$ denotes sequence concatenation

The following procedure performs the insert operation into a linked list. We show it is correct with respect to the specification above.

procedure $insert($**value** $item, pos;$ **value−result** $data, next, head)$

P: $\{list(head, data, next) = L \ \wedge \ free(pos)\}$
$\{list(head, (data; pos : item), (next; pos : head)) = L\}$
/* and-simplification */
$\{list(head, (data; pos : item), (next; pos : head)) = L \ \wedge \ item = item\}$
$/ * \text{(data; pos : item)[pos]} = \text{item} * /$
$\{list(head, (data; pos : item), (next; pos : head)) = L \ \wedge \ (data; pos : item)[pos] = item\}$

$$
\begin{array}{ll}
& \qquad\qquad\qquad\qquad \text{/* array assignment axiom */} \\
& \texttt{data[pos] := item;} \\
& \{list(head, data, (next; pos : head)) = L \ \wedge \ data[pos] = item\} \\
& \qquad\qquad\qquad\qquad /* \ (\mathsf{next}; \mathsf{pos} : \mathsf{head})[\mathsf{pos}] = \mathsf{head} \ */ \\
& \{list((next; pos : head)[pos], data, (next; pos : head)) = L \ \wedge \ data[pos] = item\} \\
& \qquad\qquad\qquad\qquad \text{/* array assignment axiom */} \\
& \texttt{next[pos] := head;} \\
& \{list(next[pos], data, next) = L \ \wedge \ data[pos] = item\} \\
& \qquad\qquad\qquad\qquad \text{/* idempotence of } \wedge \text{ */} \\
& \{data[pos] = item \ \wedge \ list(next[pos], data, next) = L \ \wedge \ data[pos] = item\} \\
& \qquad\qquad\qquad\qquad /* \ \mathsf{list}(\mathsf{pos}, \mathsf{data}, \mathsf{next}) = \mathsf{data}[\mathsf{pos}] \bullet \mathsf{list}(\mathsf{next}[\mathsf{pos}], \mathsf{data}, \mathsf{next}) \ */ \\
& \{list(pos, data, next) = item \bullet L \ \wedge \ data[pos] = item\} \\
& \texttt{head := pos;} \\
Q: & \{list(head, data, next) = item \bullet L \ \wedge \ data[head] = item\}
\end{array}
$$

The verification condition

$$list(head, data, next) = L \wedge free(pos) \ \Rightarrow \ list(head, (data; pos : item), (next; pos : head)) = L$$

is seen to be valid by observing that, when $free(pos)$ is true, then pos indexes in to free positions in both $data$ and $next$. Hence, changing the value in these positions cannot change the linked list. The best way to understand this proof is to start with the postcondition and work your way backwards, using the comments to guide you in each step.

Specifying Deletion from Linked Lists

$delete$(**value−result** $data, next, head$)
/* delete first item from linked list stored in arrays $data$, $next$ */

precondition: $list(head, data, next) = X \bullet L$

postcondition: $list(head, data, next) = L$

3.9 Deriving Invariants from Postconditions

When the postcondition contains quantifications, the invariant can sometimes be obtained from the postcondition by making the range of quantification depend on the program variables:

1. The invariant is initially established by making the range empty.
2. The range extended one element at a time in a loop
3. When the range has been extended to that in the postcondition, the program can terminate.

Examples of invariants that we derived in this way are the invariants in the following programs: linear search, array sum, array minimum, bubble sort.

3.10 Directed Graphs

There are two popular representations for directed graphs:

1. Adjacency matrix: $c[i, j]$ is T iff there is an edge from i to j. If $c[i, j]$ is an integer, then it gives the "cost" of the edge from i to j. Assumes nodes are numbered.
2. Adjacency list: for each node i, there is a linked list containing the nodes j for which there is an edge from i to j

We now present a derivation of a shortest path algorithm from a formal specification.

3.10.1 All-Pairs Shortest Path Algorithm

Input:

1. Directed graph $g = (v, e)$. v is set of vertices (nodes), e is binary relation over nodes.
2. Cost matrix c for g. $c[i, j]$ = cost of the edge from i to j $(= +\infty \text{ if } \neg e(i, j))$.

Output:

Matrix a, where $a[i, j]$ = cost of a shortest path from i to j ($= +\infty$ if no path from i to j).

Precondition P: $\forall(i, j : e(i, j) : c[i, j] \geq 0)$
/* edges have non-negative cost */

Postcondition Q: $\forall(i, j : i \in v \wedge j \in v : a[i, j] = \mathbf{MIN}(\pi : path(\pi, i, j) : cost(\pi)))$

where

$$path(\pi, i, j) \equiv \quad \exists(k, \pi' : \pi = \pi' \bullet k : path(\pi', i, k) \wedge e(k, j)) \vee (c[i, j] \neq +\infty \wedge \pi = i \bullet j)$$

$$cost(\pi) = \begin{cases} 0 & \text{if } |\pi| = 1 \\ c[\pi[1], \pi[2]] + cost(tl(\pi)) & \text{if } |\pi| > 1 \end{cases}$$

π, π' are sequences of nodes (paths).
$tl(\pi)$ is π with the first node removed.

As this is a considerably more intricate algorithm than those we have covered so far, it is unclear how to formulate the invariant. We apply the method suggested in section 3.9.

For the shortest path problem, the range in the postcondition is $path(\pi, i, j)$. We restrict this by restricting the nodes that can be in π:

$$\forall(i, j : i \in v \wedge j \in v : a[i, j] = \mathbf{MIN}(\pi : path(\pi, i, j, \alpha) : cost(\pi)))$$

where

$$path(\pi, i, j, \alpha) \equiv \exists(k, \pi' : \pi = \pi' \bullet k \wedge k \in \alpha : path(\pi', i, k, \alpha) \wedge e(k, j)) \vee (c[i, j] \neq +\infty \wedge \pi = i \bullet j).$$

/* every node of π, except possibly i, j, is in $\alpha \subseteq v$ */

The invariant of our shortest path program can then be expressed informally as: "$a[i,j]$ = the length of a shortest path π such that $path(\pi,i,j,\alpha)$, where $\alpha \subseteq v$. We express this invariant $I(a,\alpha)$ formally as follows:

$$\forall(i,j : i \in v \wedge j \in v : a[i,j] = \mathbf{MIN}(\pi : path(\pi,i,j,\alpha) : cost(\pi))) \ \wedge \ \alpha \subseteq v.$$

At this point, we should:

1. Determine how to initialize the invariant, and
2. Check whether the invariant and termination condition (i.e., $\neg B$, where B is the looping condition) implies the postcondition.

These checks should be done as soon as possible, so that errors are discovered at the earliest possible time, before more effort is wasted.

To initialize the invariant, we use $a := c; \alpha := \emptyset$. The precondition is then $I(c,\emptyset)$. Since a path from i to j contains no nodes except i and j, this path is simply the edge from i to j (if it exists). Hence, the cost of a shortest path from i to j is the cost of an edge from i to j, i.e., $c[i,j]$. This is what $I(c,\emptyset)$ asserts, hence $I(c,\emptyset)$ is valid.

Upon termination, we have $I(a,\alpha) \wedge \alpha = v$, i.e.,

$$\forall(i,j : i \in v \wedge j \in v : a[i,j] = \mathbf{MIN}(\pi : path(\pi,i,j,v) : cost(\pi))) \ \wedge \ v \subseteq v$$

By definition, $path(\pi,i,j,v) \equiv path(\pi,i,j)$. Hence:

$$\forall(i,j : i \in v \wedge j \in v : a[i,j] = \mathbf{MIN}(\pi : path(\pi,i,j) : cost(\pi)))$$

which is Q. These considerations lead to the following outline and partial tableau.

```
    /* Shortest path: find the length of a shortest path from i to j. */
P:  {∀(i, j : e(i, j) : c[i, j] ≥ 0)}
    {I(c, ∅)}
    a := c;                                   /* shorthand for matrix copy */
    {I(a, ∅)}
    α := ∅;
    {invariant I(a, α): ∀(i, j :: a[i, j] = MIN(π : path(π, i, j, α) : cost(π))) ∧ α ⊆ v}
    while α ≠ v do
        {I(a, α) ∧ α ≠ v}
        choose(v − α, n);                     /* Select some node n ∈ v − α */
        α := α ∪ {n};
        extend;                               /* re-establish I(a, α) */
        {I(a, α)}
    endwhile
    {I(a, α) ∧ α = v}
Q:  {∀(i, j :: a[i, j] = MIN(π : path(π, i, j) : cost(π)))}
```

extend re-establishes $I(a,\alpha)$, "extending" it to take account of new node n. How is $I(a,\alpha)$ re-established? The idea is as follows: adding n to α introduces new possible shortest paths. We need to check these and modify a appropriately. After adding n to α, shortest path from i to j is the shorter of:

- The previous shortest path from i to j.
- A new shortest path from i to j via n.

This new path must be a concatenation of a shortest path from i to n and a shortest path from n to j. Hence, *extend* is:

$$a[i,j] := min(a[i,j],\ a[i,n] + a[n,j])$$

executed for every $i, j \in v$. We can now complete the outline and tableau as follows.

```
    /* Shortest path: find the length of a shortest path from i to j. */
P:  {∀(i,j : e(i,j) : c[i,j] ≥ 0)}
    {I(c,∅)}
    a := c;                                  /* shorthand for matrix copy */
    {I(a,∅)}
    α := ∅;
    {invariant I(a,α) : ∀(i,j :: a[i,j] = MIN(π : path(π,i,j,α) : cost(π))) ∧ α ⊆ v}
    while α ≠ v do
        {I(a,α) ∧ α ≠ v}
        choose(v − α, n);                    /* Select some node n ∈ v − α */
        α := α ∪ {n};
        i := 0;
        while i ≠ |v| − 1 do
            j := 0;
            while j ≠ |v| − 1 do
                a[i,j] := min(a[i,j], a[i,n] + a[n,j])
            endwhile
        endwhile
        {I(a,α)}
    endwhile
    {I(a,α) ∧ α = v}
Q:  {∀(i,j :: a[i,j] = MIN(π : path(π,i,j) : cost(π)))}
```

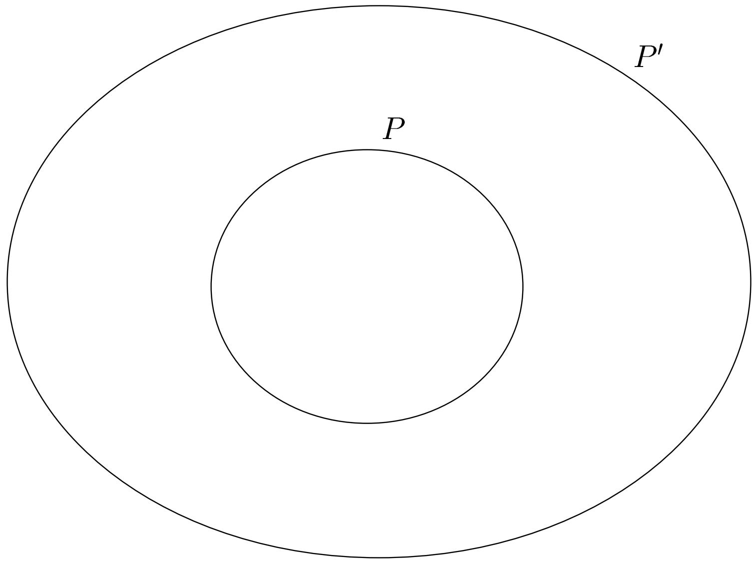
3.11 The Weakest Precondition

The weakest precondition provides an alternate way of defining program correctness. We first define the concepts "weaker" and "stronger."

If $P \Rightarrow P'$ is valid, then :

1. P is *stronger* than P'.

2. P' is *weaker* than P.

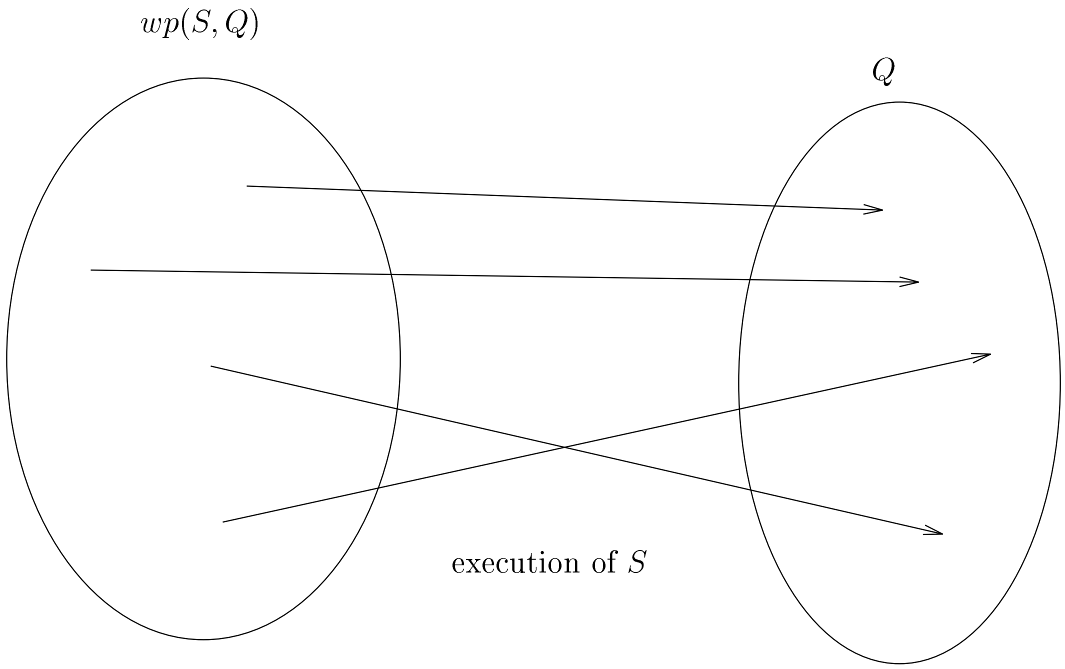


P restricts the set of possible states more than P' does. F is the strongest predicate. T is the weakest predicate.

Definition 21 ($wp(S, Q)$)
$wp(S, Q)$ is the weakest *predicate P such that:*

> *If execution of S is started with P true, then execution terminates, and the final state is guaranteed to satisfy Q.*

$wp(S, Q)$ is called the *weakest precondition of S with respect to Q*. The following figure illustrates $wp(S, Q)$. Recall the view of a predicate as the set of states in which that predicate is true (see chapter 1). The execution of program S can then be regarded as a *mapping* that takes an initial state to a final state. $wp(S, Q)$ is then the *largest* set of initial states that are guaranteed to always be mapped into some final state in Q by the execution of S.



3.11.1 Relation Between wp and Hoare Triples

The relation between weakest preconditions and Hoare triples is expressed by the following facts:

1. $\langle wp(S, Q)\rangle\, S\, \langle Q\rangle$
2. if $\langle P\rangle\, S\, \langle Q\rangle$, then $P \Rightarrow wp(S, Q)$
3. if $P \Rightarrow wp(S, Q)$, then $\langle P\rangle\, S\, \langle Q\rangle$

We summarize the above by the single fact:

$$\langle P\rangle\, S\, \langle Q\rangle \;\equiv\; P \Rightarrow wp(S, Q)$$

This expresses $\langle P\rangle\, S\, \langle Q\rangle$ as a predicate.

3.11.2 Specifying Termination Using the Weakest Precondition

When dealing with termination only, we replace the general postcondition Q by the postcondition T (true).

$wp(S, \mathrm{T})$ is the *weakest* predicate P such that:

> If execution of S is started with P true, then execution terminates (and the final state is guaranteed to satisfy T).

i.e., $wp(S, \mathrm{T})$ is the weakest precondition for termination of S.

Using this, we can express the Hoare triple for conditional correctness as a predicate:

$$\{P\}\, S\, \{Q\} \;\equiv\; (P \wedge wp(S, \mathrm{T}) \Rightarrow wp(S, Q))$$

3.11.3 Weakest Precondition of the Assignment Statement

$$wp(x := e, Q(x)) \;\equiv\; Q(e)$$

If $Q(e)$ is true before, and x is assigned e, then $Q(x)$ will be true after.

Example 65 $wp(x := x + 1, x \leq 5) \equiv (x + 1 \leq 5) \equiv x \leq 4$.

$wp(x := 10, x = 10) \equiv (10 = 10) \equiv \mathrm{T}$

$wp(x := 10, x = 11) \equiv (11 = 10) \equiv \mathrm{F}$

3.11.4 Weakest Precondition of the Assignment Statement for Arrays

$$wp(a[i] := e, Q(a)) \;\equiv\; Q((a; i : e))$$

If $Q((a; i : e))$ is true before, and a is assigned $(a; i : e)$ (see subsection 3.8.1), then $Q(a)$ will be true after.

Example 66 $wp(a[i] := 5, a[i] = 5) \;\equiv\; (a; i : 5)[i] = 5 \;\equiv\; \mathrm{T}$.

Example 67 $wp(a[i] := a[i] + 1, a[i] \leq 5) \;\equiv\; (a; i : (a[i] + 1))[i] \leq 5 \;\equiv\; a[i] \leq 4$

Example 68 $wp(a[i] := 5, a[i] = a[j]) \;\equiv\; (a; i : 5)[i] = (a; i : 5)[j] \;\equiv\; i = j \vee a[j] = 5$

Example 69 $wp(a[a[i]] := i, a[i] = i) \;\equiv\; (a; a[i] : i)[i] = i \;\equiv\; a[i] = i$.

3.11.5 Weakest Precondition of the two-way-if

$$wp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2, Q) \;\equiv\; (B \Rightarrow wp(S_1, Q)) \;\wedge\; (\neg B \Rightarrow wp(S_2, Q))$$

If B is true initially, then execution of S_1 terminates with Q true. If B is false initially, then execution of S_2 terminates with Q true. Hence, in either case, execution of **if** B **then** S_1 **else** S_2 terminates with Q true.

Example 70 Computing the maximum of two integers.

$$wp(\textbf{if } x \geq y \textbf{ then } z := x \textbf{ else } z := y, z = max(x,y))$$
$$\equiv$$
$$(x \geq y \Rightarrow wp(z := x, z = max(x,y))) \quad \wedge \quad (x < y \Rightarrow wp(z := y, z = max(x,y)))$$
$$\equiv$$
$$(x \geq y \Rightarrow x = max(x,y)) \quad \wedge \quad (x < y \Rightarrow y = max(x,y))$$
$$\equiv$$
$$\mathrm{T}$$

3.11.6 Weakest Precondition of the one-way-if

$$wp(\textbf{if } B \textbf{ then } S_1, Q) \equiv (B \Rightarrow wp(S_1, Q)) \ \wedge \ (\neg B \Rightarrow Q)$$

If B is true initially, then execution of S_1 terminates with Q true. If B is false initially, then nothing is executed, i.e., the program state is unchanged. Hence, Q must be true initially. Hence, in either case, execution of **if** B **then** S_1 terminates with Q true.

Example 71 Computing the absolute value.

$$wp(\textbf{if } x < 0 \textbf{ then } y := -x, y = abs(x))$$
$$\equiv$$
$$(x < 0 \Rightarrow wp(y := -x, y = abs(x))) \wedge (x \geq 0 \Rightarrow y = abs(x))$$
$$\equiv$$
$$(x < 0 \Rightarrow -x = abs(x)) \wedge (x \geq 0 \Rightarrow y = abs(x))$$
$$\equiv$$
$$x \geq 0 \Rightarrow y = x$$

3.11.7 Weakest Precondition of a Sequential Composition

$$wp(S_1; S_2, Q) \equiv wp(S_1, wp(S_2, Q))$$

Execution of S_1 terminates with $wp(S_2, Q)$ true. Hence, execution of S_2 following S_1 terminates with Q true.

Example 72 Compute $wp(S_3; S_4, I(k, sum))$, where

$I(k, sum)$: $sum = \mathbf{\Sigma}(i : 0 \leq i < k : a[i])$

S_3: $sum := sum + a[k]$;
S_4: $k := k + 1$

Work backwards from the last assignment:

$wp(S_4, I(k, sum)) \equiv I(k+1, sum)$

$wp(S_3, I(k+1, sum)) \equiv I(k+1, sum + a[k])$

Hence

$wp(S_3; S_4, I(k, sum)) \equiv I(k+1, sum + a[k])$

3.11.8 Weakest Precondition of the while-statement

$$wp(\textbf{while } B \textbf{ do } S, Q) \equiv \exists(k : k \geq 0 : P_k)$$

where

$P_0 \equiv \neg B \wedge Q$
$P_1 \equiv B \wedge wp(S, P_0)$
$P_2 \equiv B \wedge wp(S, P_1)$
$\vdots$
$P_k \equiv B \wedge wp(S, P_{k-1})$

P_k : The **while**-loop terminates in exactly k iterations, and the final state satisfies Q

There is no "closed form," i.e., no "formula" for the weakest precondition of a **while**-loop that can be used to mechanically calculate it, like we did above for the assignment and **if**-statements. Therefore, we still need invariants.

3.11.9 Constructing a Proof Tableau

Using the concept of weakest precondition, we can present a more comprehensive procedure for constructing a proof tableau.

1. Write down the program S together with its precondition P and postcondition Q
2. For each **while**-statement **while** B **do** S' **endwhile** that occurs in S:
   (a) Find an invariant I for the **while**-statement
   (b) Write $\{I\}$ immediately before the **while**-statement
   (c) Write $\{I \wedge \neg B\}$ immediately after the **while**-statement
   (d) Write $\{I \wedge B\}$ at the top of the body of the **while**-statement (i.e., immediately before S')
   (e) Write $\{I\}$ at the bottom of the body of the **while**-statement (i.e, immediately after S')

3. For each assignment statement $x := e$ with postcondition $R(x)$, apply the assignment axiom to obtain a precondition $R(e)$
4. For each simple **if**-statement **if** B **then** S_1 **else** S_2 **endif** with postcondition Q', calculate wp(**if** B **then** S_1 **else** S_2 **endif**, Q') and use it as the precondition for the **if**-statement.
5. For each complex **if**-statement **if** B **then** S_1 **else** S_2 **endif** with postcondition Q':
   (a) Calculate a precondition $pre(S_1, Q')$ for S_1 with respect to Q', and a precondition $pre(S_2, Q')$ for S_2 with respect to Q'.
   (b) Use $(B \Rightarrow pre(S_1, Q')) \wedge (\neg B \Rightarrow pre(S_2, Q'))$ as the precondition for the **if**-statement.
6. Repeat steps 3 through 5 until the tableau is *complete*.
7. For each pair of predicates P', P'' such that P'' immediately follows P' in the tableau (i.e., with no statement in between them), extract the *verification condition* $P' \Rightarrow P''$.

Example 73 Finding the minimum value in an array.

```
P:        {T}
          {I(0, a[0])}
          j := 0;
          {I(j, a[0])}
          m := a[0];
          {invariant I(j, m) : m = MIN(i : 0 ≤ i ≤ j : a[i])}
          while B1 : j ≠ n − 1 do
                    {I(j, m) ∧ B1}
                    {P2(j + 1, m)}
                    j := j + 1;
          P2(j, m): {wp(S, I(j, m))}
          S:        if B2 : m > a[j] then
                          m := a[j]
                    else
                          skip
                    endif
                    {I(j, m)}
          endwhile
          {I(j, m) ∧ ¬B1}
Q(m):     {m = MIN(i : 0 ≤ i < n : a[i])}
```

$P_2(j, m)$

$\equiv$

$wp(S, I(j, m))$

$\equiv$

$(m > a[j] \Rightarrow wp(m := a[j], I(j,m))) \;\; \wedge \;\; (m \leq a[j] \Rightarrow wp(skip, I(j,m)))$

$\equiv$

$(m > a[j] \Rightarrow I(j, a[j])) \;\; \wedge \;\; (m \leq a[j] \Rightarrow I(j,m))$

$\equiv$

$(m > a[j] \Rightarrow a[j] = \mathbf{MIN}(i : 0 \leq i \leq j : a[i])) \;\; \wedge \;\; (m \leq a[j] \Rightarrow m = \mathbf{MIN}(i : 0 \leq i \leq j : a[i]))$

Verification condition:

$I(j,m) \wedge B_1 \Rightarrow P_2(j+1, m)$

$\equiv$

$m = \mathbf{MIN}(i : 0 \leq i \leq j : a[i]) \wedge j \neq n-1 \Rightarrow$
$(m > a[j+1] \Rightarrow a[j+1] = \mathbf{MIN}(i : 0 \leq i \leq j+1 : a[i]) \wedge$
$m \leq a[j+1] \Rightarrow m = \mathbf{MIN}(i : 0 \leq i \leq j+1 : a[i]))$

$\equiv$

$(\ m = \mathbf{MIN}(i : 0 \leq i \leq j : a[i]) \wedge j \neq n-1 \Rightarrow (m > a[j+1] \Rightarrow a[j+1] = \mathbf{MIN}(i : 0 \leq i \leq j+1 : a[i]))$
$)$
$\wedge$
$(\ m = \mathbf{MIN}(i : 0 \leq i \leq j : a[i]) \wedge j \neq n-1 \Rightarrow (m \leq a[j+1] \Rightarrow m = \mathbf{MIN}(i : 0 \leq i \leq j+1 : a[i]))\ )$

$\equiv$

$(m = \mathbf{MIN}(i : 0 \leq i \leq j : a[i]) \wedge j \neq n-1 \wedge m > a[j+1] \Rightarrow a[j+1] = \mathbf{MIN}(i : 0 \leq i \leq j+1 : a[i]))$
$\wedge$
$(m = \mathbf{MIN}(i : 0 \leq i \leq j : a[i]) \wedge j \neq n-1 \wedge m \leq a[j+1] \Rightarrow m = \mathbf{MIN}(i : 0 \leq i \leq j+1 : a[i]))$

These two implications are easily verified using properties of **MIN**.

Example 74 Termination of a program to compute the factorial.

This program uses an **if**-statement to check that its input n is non-negative, and so avoids the possibility of nontermination that occurs in this case.

P: $\langle \mathrm{T} \rangle$
  input(n);
  $\langle (n < 0 \Rightarrow \mathrm{T}) \wedge (n \geq 0 \Rightarrow n \geq 0) \rangle$
  if $n < 0$ **then**
    $\langle \mathrm{T} \rangle$
    output("n must be non-negative")
    $\langle \mathrm{T} \rangle$
  else

$\langle n \geq 0 \rangle$
$\langle I(0) \rangle$
$k := 0;$
$\langle I(k) \rangle$
$f := 1;$
$\langle$ invariant $I(k)$: $0 \leq k \leq n \rangle$
/* variant $\varphi(k)$: $n - k$ */
while $B : k \neq n$ **do**
    $\langle I(k) \wedge k \neq n \wedge \varphi(k) = C \rangle$
    $\langle I(k+1) \wedge \varphi(k+1) < C \wedge \varphi(k) \geq 0 \rangle$
    $k := k + 1;$
    $\langle I(k) \wedge \varphi(k) < C \rangle$
    $f := f * k;$
    $\langle I(k) \wedge \varphi(k) < C \rangle$
endwhile
$\langle \mathrm{T} \rangle$
endif
$\langle \mathrm{T} \rangle$

Index