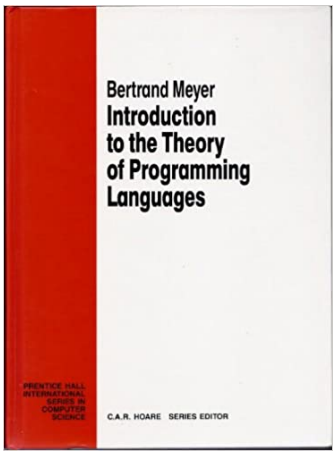
9

Axiomatic semantics



9.1 OVERVIEW

As introduced in chapter 4, the axiomatic method expresses the semantics of a programming language by associating with the language a mathematical theory for proving properties of programs written in that language.

The contrast with denotational semantics is interesting. The denotational method, as studied in previous chapters, associates a denotation with every programming language construct. In other words, it provides a **model** for the language.

This model, a collection of mathematical objects, is very abstract; but it is a model. As with all explicit specifications, there is a risk of *overspecification*: when you choose one among several possible models of a system, you risk including irrelevant details.

Some of the specifications of chapters 6 and 7 indeed appear as just one possibility among others. For example, the technique used to model block structure (chapter 7) looks very much like the corresponding implementation techniques (stack-based allocation). This was intended to make the model clear and realistic; but we may suspect that other, equally acceptable models could have been selected, and that the denotational model is more an abstract implementation than a pure specification.

The axiomatic method is immune from such criticism. It does not attempt to provide an explicit model of a programming language by attaching an explicit meaning to every construct. Instead, it defines **proof rules** which make it possible to reason about the properties of programs.

In a way, of course, the proof rules are meanings, but very abstract ones. More importantly, they are ways of reasoning *about* programs.

Particularly revealing of this difference of spirit between the axiomatic and denotational approaches is their treatment of erroneous computations:

- A denotational specification must associate a denotation with every valid language construct. As noted in 6.1, a valid construct is structurally well-formed but may still fail to produce a result (by entering into an infinite computation); or it may produce an error result. For non-terminating computations, modeling by partial functions has enabled us to avoid being over-specific; but for erroneous computations, a denotational model must spill the beans and say explicitly what special "error" values, such as *unknown* in 6.4.2, the program will yield for expressions whose value it cannot properly compute. (See also the discussion in 6.4.4.)
- In axiomatic semantics, we may often deal with erroneous cases just by making sure that no proof rule applies to them; no special treatment is required. This may be called the unobtrusive approach to erroneous cases and undefinedness.

> You may want to think of two shopkeepers with different customer policies: Billy's Denotational Emporium serves all valid requests ("No construct too big or too small" is its slogan), although the service may end up producing an error report, or fail to terminate; in contrast, a customer with an erroneous or overly difficult request will be politely but firmly informed that the management and staff of Ye Olde Axiomatic Shoppe regret their inability to prove anything useful about the request.

Because of its very abstractness, axiomatic semantics is of little direct use for some of the applications of formal language specifications mentioned in chapter 1, such as writing compilers and other language systems. The applications to which it is particularly relevant are program verification, understanding and standardizing languages, and, perhaps most importantly, providing help in the construction of correct programs.

9.2 THE NOTION OF THEORY

An axiomatic description of a language, it was said above, is a theory for that language. A theory about a particular set of objects is a set of rules to express statements about those objects and to determine whether any such statement is true or false.

> As always in this book, the word "statement" is used here in its ordinary sense of a property that may be true or false – not in its programming sense of command, for which this book always uses the word "instruction".

9.2.1 Form of theories

A theory may be viewed as a formal language, or more properly a **metalanguage**, defined by syntactic and semantic rules. (Chapter 1 discussed the distinction between language and metalanguage. Here the metalanguage of an axiomatic theory is the formalism used to reason about languages.)

The syntactic rules for the metalanguage, or **grammar**, define the meaningful statements of the theory, called **well-formed formulae**: those that are worth talking about. "Well-formed formula" will be abbreviated to "formula" when there is no doubt about well-formedness.

The semantic rules of the theory (*axioms* and *inference rules*), which only apply to well-formed formulae, determine which formulae are *theorems* and which ones are not.

9.2.2 Grammar of a theory

The grammar of a theory may be expressed using standard techniques such as BNF or abstract syntax, both of which apply to metalanguages just as well as to languages.

An example will illustrate the general form of a grammar. Consider a simple theory of natural integers. Its grammar might be defined by the following rules (based on a vocabulary comprising letters, the digit 0 and the symbols =, <, $\Longrightarrow$, $\neg$ and '):

1 • The formulae of the metalanguage are **boolean expressions**.

2 • A boolean expression is of one of the four forms

$\alpha = \beta$

$\alpha < \beta$

$\neg\ \gamma$

$\gamma \Longrightarrow \delta$

where α and β are **integer expressions** and γ and δ are boolean expressions.

3 • An integer expression is of one of the three forms

0

n

α'

where *n* is any lower-case letter from the roman alphabet and α is any integer expression.

In the absence of parentheses, the grammar is ambiguous, which is of no consequence for this discussion. (For a fully formal presentation, abstract syntax, which eliminates ambiguity, would be more appropriate.)

According to the above definition, the following are well-formed formulae:

$0 = 0$

$0 \neq 0$

$m\,'''\; <\; 0\,''$

$0 = 0 \implies 0 \neq 0$

The following, however, are not well-formed formulae (do not belong to the metalanguage of the theory):

$0 < 1$ -- Uses a symbol which is not in the vocabulary of the theory.

$0 < '\,n\,'$ -- Does not conform to the grammar.

9.2.3 Theorems and derivation

Given a grammar for a theory, which defines its well-formed formulae, we need a set of rules for **deriving** certain formulae, called **theorems** and representing true properties of the theory's objects.

The following notation expresses that a formula f is a theorem:

$\vdash f$

Only well-formed formulae may be theorems: there cannot be anything interesting to say, within the theory, about an expression which does not belong to its metalanguage. Within the miniature theory of integers, for example, it is meaningless to ask whether $0 < 1$ may be derived as a theorem since that expression simply does not belong to the metalanguage. Here as with programming languages, we never attempt to attach any meaning to a structurally invalid element. The rest of the discussion assumes all formulae to be well-formed.

> The restriction to well-formed formulae is similar, at the metalanguage level, to the conventions enforced in the specification of programming languages: as noted in 6.1, semantic descriptions apply only to statically valid constructs.

To derive theorems, a theory usually provides two kinds of rules: **axioms** and **inference rules**, together called "rules".

9.2.4 Axioms

An axiom is a rule which states that a certain formula is a theorem. The example theory might contain the axiom

A_0

$$\vdash\ 0 < 0\,'$$

This axiom reflects the intended meaning of ' as the successor operation on integers: it expresses that zero is less than the next integer (one).

9.2.5 Rule schemata

To characterize completely the meaning of ' as "successor", we need another axiom complementing A_0:

$A_{successor}$

For any integer expressions m and n:

$$\vdash\ m < n \implies m\,' < n\,'$$

This expresses that if m is less than n, the same relation applies to their successors.

$A_{successor}$ is not exactly an axiom but what is called an axiom **schema** because it refers to arbitrary integer expressions m and n. We may view it as denoting an infinity of actual axioms, each of which is obtained from the axiom schema by choosing actual integer expressions for m and n. For example, choosing $0\,''$ and 0 for m and n yields the following axiom:

$$\vdash\ 0\,'' < 0 \implies 0\,''' < 0\,'$$

In more ordinary terms, this says: "2 less than 0 implies 3 less than 1" (which happens to be a true statement, although not a very insightful one).

In practice, most interesting axioms are in fact axiom schemata.

The following discussion will simply use the term "axiom", omitting "schema" when there is no ambiguity. As a further convention, single letters such as m and n will stand for arbitrary integer expressions in a rule schema: in other words, we may omit the phrase "For any integer expressions m and n".

9.2.6 Inference rules

Inference rules are mechanisms for deriving new theorems from others. An inference rule is written in the form

$$\frac{f_1, f_2, \ldots, f_n}{f_0}$$

and means the following:

> If $f_1, f_2, \ldots, f_n$ are theorems, then f_0 is a theorem.

The formulae above the horizontal bar are called the **antecedents** of the rule; the formula below it is called its **consequent**.

As with axioms, many inference rules are in fact inference rule schemata, involving parameterization. "Inference rule" will be used to cover inference rule schemata as well.

The mini-theory of integers needs an inference rule, useful in fact for many other theories. The rule is known as **modus ponens** and makes it possible to use implication in inferences. It may be expressed as follows for arbitrary boolean expressions p and q:

MP

$$\frac{p, \quad p \implies q}{q}$$

This rule brings out clearly the distinction between logical implication ($\implies$) and inference: the $\implies$ sign belongs to the metalanguage of the theory: as an element of its vocabulary, it is similar to $<$, $'$, 0 etc. Although this symbol is usually read aloud as "implies", it does not by itself provide a proof mechanism, as does an inference rule. The role of modus ponens is precisely to make $\implies$ useful in proofs by enabling q to be derived as a theorem whenever both p and $p \implies q$ have been established as theorems.

Another inference rule, essential for proofs of properties of integers, is the rule of induction, of which a simple form may be stated as:

IND

$$\frac{\phi\,(0), \quad \phi\,(n) \implies \phi\,(n')}{\phi\,(n)}$$

9.2.7 Proofs

The notions of axiom and inference rule lead to a precise definition of theorems:

> **Definition** (Theorem): A theorem t in a theory is a well-formed formula of the theory, such that t may be derived from the axioms by zero or more applications of the inference rules.

The mechanism for deriving a theorem, called a **proof**, follows a precise format, already outlined in 4.6.3 (see figure 4.10). If the proof rigorously adheres to this format, no human insight is required to determine whether the proof is correct or not; indeed the task of checking the proof can be handed over to a computer program. *Discovering* the proof requires insight, of course, but not checking it if it is expressed in all rigor.

The format of a proof is governed by the following rules:

1 • The proof is a sequence of lines.

2 • Each line is numbered.

3 • Each line contains a formula, which the line asserts to be a theorem. (So you may consider that the formula on each line is preceded by an implicit $\vdash$.)

4 • Each line also contains an argument showing unambiguously that the formula of the line is indeed a theorem. This is called the **justification** of the line.

The justification (rule 4) must be one of the following:

A • The name of an axiom or axiom schema of the theory, in which case the formula must be the axiom itself or an instance of the axiom schema.

B • A list of references to previous lines, followed by a semicolon and the name of an inference rule or inference rule schema of the theory.

In case B, the formulae on the lines referenced must coincide with the antecedents of the inference rule, and the formula on the current line must coincide with the consequent of the rule. (In the case of a rule schema, the coincidence must be with the antecedents and consequents of an instance of the rule.)

As an example, the following is a proof of the theorem

$$\vdash \quad i < i'$$

(that is to say, every number is less than its successor) in the the above mini-theory.

[9.1]

Number	**Formula**	**Justification**
M.1	$0 < 0'$	A_0
M.2	$i < i' \implies i' < i''$	$A_{successor}$
M.3	$i < i'$	M.1, M.2; IND

On line M.2 the axiom schema $A_{successor}$ is instantiated by taking i for m and i' for n. On line M.3 the inference rule IND is instantiated by taking $\phi(n)$ to be $i < i'$. Note that for the correctness of the proof to be mechanically checkable, as claimed above, the justification field should include, when appropriate, a description of how a rule or axiom schema is instantiated.

The strict format described here may be somewhat loosened in practice when there is no ambiguity; it is common, for example, to merge the application of more than one rule on a single line for brevity (as with the proof of figure 4.10). The present discussion, however, is devoted to a precise analysis of the axiomatic method, and it needs at least initially to be a little pedantic about the details of proof mechanisms.

9.2.8 Conditional proofs and proofs by contradiction

[This section may be skipped on first reading.]

Practical proofs in theories which support implication and negation often rely on two useful mechanisms: conditional proofs and proofs by contradiction.

A conditional proof works as follows:

[9.2]

Definition (Conditional Proof): To prove $P \implies Q$ by conditional proof, prove that Q may be derived under the assumption that P is a theorem.

A conditional proof, usually embedded in a larger proof, will be written in the form illustrated below.

Number	Formula	Justification
$i-1$	...	...
$i.1$	P	Assumption
$i.2$	...	...
...		
$i.n$	Q	...
i	$P \implies Q$	Conditional Proof
$i+1$	...	...

Figure 9.1: A conditional sub-proof

The goal of the proof is a property of the form P **implies** Q, appearing on a line numbered i. The proof appears on a sequence of lines, called the **scope** of the proof and numbered $i.1$, $i.2$, ..., $i.n$ (for some $n \geq 1$). These lines appear just before line i. The formula stated on the first line of the scope, $i.1$, must be P; the justification field of this line, instead of following the usual requirements given on page 305, simply indicates

"Assumption". The formula proved on the last line of the scope, $i.j$, must be Q. The justification field of line i simply indicates "Conditional Proof".

Conditional proofs may be nested; lines in internal scopes will be numbered $i.j.k$, $i.j.k.l$ etc.

The proof of the conclusion Q in lines $i.2$ to $i.n$ may use P, from line $i.1$, as a premise. It may also use any property established on a line preceding $i.1$, if the line is not part of the scope of another conditional proof. (For a nested conditional proof, lines established as part of enclosing scopes are applicable.)

P is stated on line $i.1$ only as assumption for the conditional proof; it and any formula deduced from it may not be used as justifications outside the scope of that proof.

Proofs by contradiction apply to theories which support negation:

> **Definition** (Proof by Contradiction): To prove P by contradiction, prove that *false* may be derived under the assumption that $\neg P$ is a theorem.

The general form of the proof is the same as above; here the goal on line i is P, with "Contradiction", instead of "Conditional Proof", in its justification field. The property proved on the last line of the scope, $i.j$, must be *false*.

9.2.9 Interpretations and models

As presented so far, a theory is a purely formal mechanism to derive certain formulae as theorems. No commitment has been made as to what the formulae actually represent, except for the example, which was interpreted as referring to integers.

In practice, theories are developed not just for pleasure but also for profit: to deduce useful properties of actual mathematical entities. To do so requires providing **interpretations** of the theory. Informally, you obtain an interpretation by associating a member of some mathematical domain with every element of the theory's vocabulary, in such a way that a boolean property of the domain is associated with every well-formed formula. A **model** is an interpretation which associates a true property with every theorem of the theory. The only theories of interest are those which have at least one model.

When a theory has a model, it often has more than one. The example theory used above has a model in which the integer zero is associated with the symbol 0, the successor operation on integers with the symbol ', the integer equality relation with = and so on. But other models are also possible; for example, the set of all persons past and present (assumed to be infinite), with 0 interpreted as modeling some specific person (say the reader), x' interpreted as the mother of x, $x < y$ interpreted as "y is a maternal ancestor of x" and so on, would provide another model.

Often, a theory is developed with one particular model in mind. This was the case for the example theory, which referred to the integer model, so much so that the vocabulary of its metalanguage was directly borrowed from the language of integers. Similarly, the theories developed in the sequel are developed for a specific application such as the semantics of programs or, in the example of the next section, lambda expressions. But when we study axiomatic semantics we must forget about the models and concentrate on the mechanisms for deriving theorems through purely logical rules.

9.2.10 Discussion

As a conclusion of this quick review of the notion of theory and proof, some qualifications are appropriate. As defined by logicians, theories are purely formal objects, and proof is a purely formal game. The aim pursued by such rigor (where, in the words of [Copi 1973], "a system has rigor when no formula is asserted to be a theorem unless it is logically entailed by the axioms") is to spell out the intellectual mechanisms that underlie mathematical reasoning.

It is well known that ordinary mathematical discourse is not entirely formal, as this would be unbearably tedious; the proof process leaves some details unspecified and skips some steps when it appears that they do not carry any conceptual difficulty.

The need for a delicate balance between rigor and informality is well accepted in ordinary mathematics, and in most cases this works to the satisfaction of everyone concerned – although "accidents" do occur, of which the most famous historically is the very first proof of Euclid's *Elements*, where the author relied at one point on geometrical intuition, instead of restricting himself to his explicitly stated axioms. Formal logic, of course, is more demanding.

Although purely formal in principle, theories are subject to some plausibility tests. Two important properties are:

- Soundness: a theory is sound if for no well-formed formula f the rules allow deriving both f and $\neg f$.
- Completeness: a theory is complete if for any well-formed formula f the rules allow the derivation of f or $\neg f$.

Both definitions assume that the metalanguage of the theory includes a symbol $\neg$ (not) corresponding to denial.

Soundness is also called "non-contradiction" or "consistency". It can be shown that a theory is sound if and only if it has a model, and that it is complete if and only if every true property of any model may be derived as a theorem.

An unsound theory is of little interest; any proposed theory should be checked for its soundness. One would also expect all "good" theories to be complete, but this is not the case: among the most importants results of mathematical logic are the incompleteness of such theories as predicate calculus or arithmetic. The study of completeness and soundness, however, falls beyond the scope of this book.

9.3 AN EXAMPLE: TYPED LAMBDA CALCULUS

[This section may be skipped on first reading. It assumes an understanding of sections 5.4 to 5.10.]

Before introducing theories of actual programming languages, it is interesting to study a small and elegant theory, due to Cardelli, which shows well the spirit of the axiomatic method, free of any imperative concern.

Chapter 5 introduced the notion of typed lambda calculus and defined (5.10.3) a mechanism which, when applied to a lambda expression, yields its type. The theory introduced below makes it possible to prove that a certain formula has a certain type – which is of course the same one as what the typing mechanism of chapter 5 would compute.

The theory's formulae are all of the form

$$b \mid e : t$$

where e is a typed lambda expression, t is a type and b is a binding (defined below). The informal meaning of such a formula is:

"Under b, e has type t."

Recall that a type of the lambda calculus is either:

1 • One among a set of basic predefined types (such as **N** or **B**).

2 • Of the form $\alpha \rightarrow \beta$, where α and β are types.

In case of ambiguity in multi-arrow type formulae, parentheses may be used; by default, arrows associate to the right.

A binding is a possibly empty sequence of <*identifier, type*> pairs. Such a sequence will be written under the form

$$x : \alpha + y : \beta + z : \gamma$$

and may be informally interpreted as the binding under which x has type α and so on. The notation also uses the symbol + for concatenation of bindings, as in $b + x : \alpha$ where b is a binding. The same identifier may appear twice in a binding; in this case the rightmost occurrence will take precedence, so that under the binding

$$x : \alpha + y : \beta + x : \gamma$$

x has type γ. One of the axioms below will express this property formally.

In typed lambda calculus, we declare every dummy identifier with a type (as in $\lambda\, x : \alpha \bullet e$). This means that the types of all bound identifier occurrences in a typed lambda expression are given in the expression itself. As for the free identifiers, their types

will be determined by the environment of the expression when it appears as a subexpression of a larger expression. So if an expression contains free identifier occurrences we can only define its type relative to the possible bindings of these identifiers.

To derive the type of an expression e, then, is to prove a property of the form

$$b \mid e : \alpha$$

for some type α. The binding b may only contain information on identifiers occurring free in e (any other information would be irrelevant). If no identifier occurs free in e, b will be empty.

Let us see how a system of axioms and inference rules may capture the type properties of lambda calculus. In the following rule schemata, e and f will denote arbitrary lambda expressions, x an arbitrary identifier and b an arbitrary binding.

The first axiom schema gives the basic semantics of bindings and the "rightmost strongest" convention mentioned above:

Right

$$b + x : \alpha \mid x : \alpha$$

In words: "Under binding b extended with type α for x, x has type α" – even if b gave another type for x.

Deducing types of identifiers other than the rightmost in a binding requires a simple inference rule schema:

Perm

$$\frac{b \mid x : \alpha}{b + y : \beta \mid x : \alpha}$$

(if x and y are different identifiers). In words: "If x has type α under b, x still has type α under b extended for any other identifier y with some type β".

To obtain the rules for typing the various forms of lambda expressions, we must remember that a lambda expression is one of atom, abstraction or application.

Atoms (identifiers) are already covered by Right: their types will be whatever the binding says about them. We do not need to introduce the notion of predefined identifier explicitly since the theory will yield a lambda expression's type relative to a certain binding, which expresses the types of the expression's free identifiers. If a formula is incorrect for some reason (as a lambda expression involving an identifier to which no type has been assigned), the axiomatic specification will not reject it; instead, it simply makes it impossible to prove any useful type property for this expression.

Abstractions describe functions and are covered by the following rule:

$I_{Abstraction}$

$$\frac{b + x: \alpha \mid e: \beta}{b \mid \{\lambda x: \alpha \bullet e\}: \alpha \to \beta}$$

This rule captures the type semantics of lambda abstractions: if assigning type α to x makes it possible to assign type β to e, then the abstraction $\lambda x: \alpha \bullet e$ describes a function of type $\alpha \to \beta$.

In a form of the lambda calculus that would support generic functions with implicit typing (inferred from the context rather than specified in the text of the expression), this rule could be adapted to:

$I_{Generic\_abstraction}$

$$\frac{b + x: \alpha \mid e: \beta}{b \mid \{\lambda x \bullet e\}: \alpha \to \beta}$$

making it possible, for example, to derive $\alpha \to \alpha$, for **any** α, as type of the function:

$$Id \triangleq \lambda x \bullet x$$

and similarly for other generic functions. But we shall not pursue this path any further.

In an axiomatic theory covering programming languages rather than lambda calculus, a pair of rules similar to Right and $I_{Abstraction}$ could be written to account for typing in block-structured languages, where innermost declarations have precedence.

Finally we need an inference rule for application expressions:

$I_{Application}$

$$\frac{b \mid f: \alpha \to \beta \qquad b \mid e: \alpha}{b \mid f\ (e): \beta}$$

In other words, if a function of type $\alpha \to \beta$ is applied to an argument, which must be of type α, the result is of type β. This completes the theory.

This theory is powerful enough to derive types for lambda expressions. It is interesting to compare the deduction process in this theory with the "computations" of types made possible by the techniques introduced in 5.10. That section used the following expression as example:

$$\lambda x: \mathbf{N} \to \mathbf{N} \bullet \lambda y: \mathbf{N} \to \mathbf{N} \bullet \lambda z: \mathbf{N} \bullet x\ (\{\lambda x: \mathbf{N} \bullet y\ (x)\}\ (z))$$

E.1	x: **N**→**N** + y: **N**→**N** + z: **N** ⎪ x:	**N**→**N**
		Right, Perm
E.2	x: **N**→**N** + y: **N**→**N** + z: **N** + x: **N** ⎪ z:	**N**
		Right, Perm
E.3	x: **N**→**N** + y: **N**→**N** + z: **N** + x: **N** ⎪ x:	**N**
		Right
E.4	x: **N**→**N** + y: **N**→**N** + z: **N** + x: **N** ⎪ y:	**N**→**N**
		Right, Perm
E.5	x: **N**→**N** + y: **N**→**N** + z: **N** + x: **N** ⎪ y (x):	**N**
		E.3, E.4; I_{App}
E.6	x: **N**→**N** + y: **N**→**N** + z: **N** ⎪ λ x: **N** • y (x):	**N**→**N**
		E.5; I_{Abst}
E.7	x: **N**→**N** + y: **N**→**N** + z: **N** ⎪ {λ x: **N** • y (x)} (z):	**N**
		E.2, E.6; I_{App}
E.8	x: **N**→**N** + y: **N**→**N** + z: **N** ⎪ x ({λ x: **N** • y (x)} (z)):	**N**
		E.1, E.7; I_{App}
E.9	x: **N**→**N** + y: **N**→**N** ⎪ λ z: **N** • x ({λ x: **N** • y (x)} (z)):	**N**→**N**
		E.8; I_{Abst}
E.10	x: **N**→**N** ⎪ λ y: **N**→**N** • λ z: **N** • x ({λ x: **N** • y (x)} (z)):	(**N**→**N**)→(**N**→**N**)
		E.9; I_{Abst}
E.11	⎪ λ x: **N** • λ y: **N**→**N** • λ z: **N** • x ({λ x: **N** • y (x)} (z)):	(**N**→**N**)→((**N**→**N**)→(**N**→**N**))
		E.10; I_{Abst}

Figure 9.2: A type inference in lambda calculus

The figure on the adjacent page shows how to derive the type of this expression in the theory exposed above. You are invited to compare it with the type computation of figure 5.4, which it closely parallels. (The subscripts in $I_{Application}$ and $I_{Abstraction}$ have been abbreviated as *App* and *Abst* respectively on the figure.)

The rest of this chapter investigates axiomatic theories of programming languages, which mostly address dynamic semantics: the meaning of expressions and instructions. This example just outlined, which could be transposed to programming languages, shows that the axiomatic method may be applied to static semantics as well.

9.4 AXIOMATIZING PROGRAMMING LANGUAGES

9.4.1 Assertions

The theories of most interest for this discussion apply to programming languages; the formulae should express relevant properties of programs.

For the most common class of programming languages, such properties are conveniently expressed through **assertions**. An assertion is a property of the program's objects, such as

$$x + y > 3$$

which may or may not be satisfied by a state of the program during execution. Here, for example, a state in which variables x and y have values 5 and 6 satisfies the assertion; one in which they both have value 0 does not.

For the time being, an assertion will simply be expressed as a boolean expression in concrete syntax, as in this example; this represents an assertion satisfied by all states in which the boolean expression has value true. A more precise definition of assertions in the Graal context will be given below (9.5.2).

9.4.2 Preconditions and postconditions

The formulae of an axiomatic theory for a programming language are not the assertions themselves, but expressions involving both assertions and program fragments. More precisely, the theory expresses the properties of a program fragment with respect to the assertions that are satisfied before and after execution of the fragment. Two kinds of assertion must be considered:

- **Preconditions**, assumed to be satisfied before the fragment is executed.
- **Postconditions**, guaranteed to be satisfied after the fragment has been executed.

A program or program fragment will be said to be correct with respect to a certain precondition P and a certain postcondition Q if and only if, when executed in a state in which P is satisfied, it yields a state in which Q is satisfied.

The difference of words – *assumed* vs. *ensured* – is significant: we treat preconditions and postconditions differently. Most significant program fragments are only applicable under certain input assumptions: for example, a Fortran compiler will not produce interesting results if presented with the data for the company's payroll program, and conversely. The precondition should of course be as broad as possible (for example, the behavior of the compiler for texts which differ from correct Fortran texts by a small number of common mistakes should be predictable); but the specification of any realistic program can only predict the complete behavior of the program for a subset of all possible cases.

It is the responsibility of the environment to invoke the program or program fragment only for cases that fall within the precondition; the postcondition binds the program, but only in cases when the precondition is satisfied.

So a pre-post specification is like a contract between the environment and the program: the precondition obligates the environment, and the postcondition obligates the program. If the environment does not observe its part of the deal, the program may do what it likes; but if the precondition is satisfied and the program fails to ensure the postcondition, the program is incorrect. These ideas lie at the basis of a theory of software construction which has been termed **programming by contract** (see the bibliographical notes).

Defined in this way, program correctness is only a *relative* concept: there is no such thing as an intrinsically correct or intrinsically incorrect program. We may only talk about a program being correct or incorrect with respect to a certain specification, given by a precondition and a postcondition.

9.4.3 Partial and total correctness

The above discussion is vague on whose responsibility it is to ensure that the program terminates. Two different approaches exist: *partial* and *total* correctness.

The following definitions characterize these approaches; they express the correctness, total or partial, of a program fragment a with respect to a precondition P and a postcondition Q.

Definition (Total Correctness). A program fragment a *is* totally correct for P and Q if and only if the following holds: Whenever a is executed in any state in which P is satisfied, the execution terminates and the resulting state satisfies Q.

Definition (Partial Correctness). A program fragment a is partially correct for P and Q if and only if the following holds: Whenever a is executed in any state in which P is satisfied and this execution terminates, the resulting state satisfies Q.

Partial correctness may also be called conditional correctness: to prove it, you are only required to prove that the program achieves the postcondition **if** it terminates. In contrast, proving total correctness means proving that it achieves the postcondition **and** terminates.

You might wonder why anybody should be interested in partial correctness. How good is the knowledge that a program *would* be correct if it only were so kind as to terminate? In fact, any non-terminating program is partially correct with respect to any specification. For example, the following loop

```
while 0 = 0 do
        print ("We try harder!")
end;
print ("We have proved Fermat's last theorem")
```

is partially correct with respect to the precondition *true* and a postcondition left for the reader to complete (actually, any will do).

The reason for studying partial correctness is pragmatic: methods for proving termination are often different in nature from methods for proving other program properties. This encourages proving separately that the program is partially correct and that it terminates. If you follow this approach, you must never forget that partial correctness is a useless property until you have proved termination.

9.4.4 Varieties of axiomatic semantics

The work on axiomatic semantics was initiated by Floyd in a 1967 article (see the bibliographical notes), applied to programs expressed through flowcharts rather than a high-level language.

The current frame of reference for this field is the subsequent work of Hoare, which proposes a logical system for proving properties of program fragments. The well-formed formulae in such a system will be called **pre-post formulae**. They are of the form

$$\{P\}\ a\ \{Q\}$$

where P is the precondition, a is the program fragment, and Q is the postcondition. (Hoare's original notation was $P\ \{a\}\ Q$, but this discussion will use braces according to the convention of Pascal and other languages which treat them as comment delimiters.) The notation expresses **partial correctness** of a with respect to P and Q: in this method, termination must be proved separately.

So a Hoare theory of a programming language consists of axioms and inference rules for deriving certain pre-post formulae. This approach may be called **pre-post semantics**.

Another approach was developed by Dijkstra. Its aim is to develop, rather than a logical theory, a **calculus** of programs, which makes it possible to reason on program fragments and the associated assertions in a manner similar to the way we reason on arithmetic and other expressions in calculus: through the application of well-formalized transformation rules. Another difference with pre-post semantics is that this theory handles total correctness. This approach may be called **wp-semantics**, where wp stands for "weakest precondition"; the reason for this name will become clear later (9.8).

We will look at these two approaches in turn. Of the two, only the pre-post method fits exactly in the axiomatic framework as defined above. But the spirit of wp-semantics is close.

9.5 A CLOSER LOOK AT ASSERTIONS

The theory of axiomatic semantics, in either its "pre-post" or "wp" flavor, applies to formulae whose basic constituents are assertions. To define the metalanguage of the theory properly, we must first give a precise definition of assertions and of the operators applicable to them.

Because assertions are properties involving program objects (variables, constants, arrays etc.), the assertion metalanguage may only be defined formally within the context of a particular programming language. For the discussion which follows that language will be Graal.

9.5.1 Assertions and boolean expressions

An assertion has been defined as a property of program objects, which a given state of program execution may or may not satisfy.

Graal, in common with all usual programming languages, includes a construct which seems very close to this notion: the boolean expression. A boolean expression also involves program objects, and has a value which is true or false depending on the values of these objects. For example, the boolean expression $x + y > 3$ has value true in a state if and only if the sums of the values that the program variables x and y have in this state is greater than three.

Such a boolean expression may be taken as representing an assertion as well – the assertion satisfied by those states in which the boolean expression has value true.

Does this indicate a one-to-one correspondence between assertions and boolean expressions? This is actually two questions:

1 • Given an arbitrary boolean expression of the programming language, can we always associate an assertion with it, as in the case of $x + y > 3$?

2 • Can any assertion of interest for the axiomatic theory of a programming language be expressed as a boolean expression?

For the axiomatic theory of Graal given below, the answer to these questions turns out to be yes. But this should not lead us confuse assertions with boolean expressions; there are both theoretical and practical reasons for keeping the two notions distinct.

On the theoretical side, assertions and boolean expressions belong to different worlds:

- Boolean expressions appear *in* programs: they belong to the programming language.
- Assertions express properties *about* programs: they belong to the formulae of the axiomatic theory.

On the practical side, languages with more powerful forms of expressions than Graal, including all common programming languages, may yield a negative answer to both questions 1 and 2 above. To express the assertions of interest in such languages, the formalism of boolean expressions is at the same time too powerful (not all boolean expressions can be interpreted as assertions) and not powerful enough (some assertions are not expressible as boolean expressions).

Examples of negative answers to question 1 may arise from functions with side-effects: in most languages you can write a boolean expression such as

$$f\,(x) > 0$$

where f is a function with side-effects. Such boolean expressions are clearly inadequate to represent assertions, which should be purely descriptive ("applicative") statements about program states.

As an example of why the answer to question 2 could be negative (not all assertions of interest are expressible as boolean expressions), consider an axiomatic theory for any language offering arrays. We may want to use an axiomatic theory to prove that any state immediately following the execution of a sorting routine satisfies the assertion

$$\forall\, i : 1\,..\,n-1 \bullet t\ [\,i\,] \le t\ [\,i+1]$$

where t is an array of bounds 1 and n. But this cannot be expressed as a boolean expression in ordinary languages, which do not support quantifiers such as $\forall$.

Commonly supported boolean expressions are just as unable to express a requirement such as "the values of t are a permutation of the original values" – another part of the sorting routine's specification.

To be sure, confusing assertions and boolean expressions in the specification of a language as simple as Graal would not cause any serious trouble. It is indeed often convenient to express assertions in boolean expression notation, as with $x + y > 3$ above. But to preserve the theory's general applicability to more advanced languages we should resist any temptation to identify the two notions.

9.5.2 Abstract syntax for assertions

To keep assertions conceptually separate from boolean expressions, we need a specific abstract syntactic type for assertions. For Graal it may be defined as:

Assertion ≜ *exp: Expression*

with a static validity function expressing that the acceptable expressions for *exp* must be of type boolean:

[9.3]

$$V_{Assertion}\ [a: Assertion,\ tm: Type\_map] \triangleq$$
$$V_{Expression}\ [a.exp,\ tm] \wedge expression\_type\ (a.exp,\ tm) = bt$$

This yields the following complete (if rather pedantic) form for the assertion used earlier as example under the form $x + y > 3$:

[9.4]

```
Assertion
    (exp: Expression (Binary
        (term1: Expression (binary
            (term1: Variable (id: "x");
            term2: Variable (id: "y");
            op: Operator (Arithmetic_op (Plus))));
        term2: Constant (Integer_constant (3));
        op: Operator (Relational_op (Gt)))))
```

or if we just use plain concrete syntax for expressions:

Assertion (*exp*: $x + y > 3$)

For simplicity, the rest of this chapter will use concrete syntax for simple assertions and their constituent expressions; furthermore, it will not explicitly distinguish between an assertion an the associated expression when no confusion is possible. So the above assertion will continue to be written $x + y > 3$ without the enclosing *Assertion* (*exp*: …).

In the same spirit, the discussion will freely apply boolean operators such as **and**, **or** and **not** to assertions; for example, *P* **and** *Q* will be used instead of

```
Assertion (exp: Expression (Binary
        (term1: P.exp;
        term2: Q.exp;
        op: Operator (Boolean_op (And)))))
```

It is important, however, to bear in mind that these are only notational facilities.

The next chapter shows how to give assertions a precise semantic interpretation in the context of denotational semantics.

9.5.3 Implication

In stating the rules of axiomatic semantics for any language, we will need to express properties of the form: "Any state that satisfies P satisfies Q". This will be written using the infix **implies** operator as

P **implies** Q

When such a property holds and the reverse, Q **implies** P, does not, P will be said said to be **stronger** than Q, and Q **weaker** than P.

The **implies** operator takes two assertions as its operands; its result, however, is not an assertion but simply a boolean value that depends on P and Q. This value is true if and only if Q is satisfied whenever P is satisfied. P **implies** Q is a well-formed formula of the metalanguage of axiomatic semantics, not a programming language construct.

9.6 FUNDAMENTALS OF PRE-POST SEMANTICS

The basic concepts are now in place to introduce axiomatic theories for programming languages such as Graal, beginning with the pre-post approach.

This section examines general rules applicable to any programming language; the next section will discuss specific language constructs in the Graal context.

9.6.1 Formulae of interest in pre-post semantics

The formulae of pre-post semantics are pre-post formulae of the form $\{P\}\ a\ \{Q\}$. The purpose of pre-post semantics is to to derive certain such formulae as theorems. The intuitive meaning of a pre-post formula is the following:

[9.5]

> **Interpretation of pre-post formulae**: A pre-post formula $\{P\}\ a\ \{Q\}$ expresses that a is partially correct with respect to precondition P and postcondition Q.

From the definition of partial correctness (page 314), this means that the computation of a, started in any state satisfying P, will (if it terminates) yield a state satisfying Q. The next chapter will interpret this notion in terms of the denotational model.

9.6.2 The rule of consequence

The first inference rule (in fact, as the subsequent rules, a rule schema) is the language-independent rule of consequence, first introduced in 4.6.2. It states that "less informative" formulae may be deduced from ones that carry more information. This concept may now be expressed more rigorously using the **implies** operator on assertions:

CONS

$$\frac{\{P\}\ a\ \{Q\},\quad P' \textbf{ implies } P,\quad Q \textbf{ implies } Q'}{\{P'\}\ a\ \{Q'\}}$$

9.6.3 Facts from elementary mathematics

The axiomatic theory of a programming language is not developed in a vacuum. Programs manipulate objects which represent integers, real numbers, characters and the like. When we attempt to prove properties of these programs, we may have to rely on properties of these objects. This means that our axiomatic theories for programming languages may have to embed other, non-programming-language-specific theories.

Assume that in a program manipulating integer variables only, we are able (as will indeed be the case with the axiomatic theory of Graal) to prove

[9.6]
$$\vdash \{x + x > 2\}\ y := x + x\ \{y > 1\},$$

but what we really want to prove is

[9.7]
$$\vdash \{x > 1\}\ y := x + x\ \{y > 1\}$$

> Before going any further you should make sure that you understand the different notations involved. The formulae in braces {...} represent assertions, each defined, as we have seen, by the associated Graal boolean expression. Occurrences of arithmetic operators such as + or > in these expressions denote Graal operators – not the corresponding mathematical functions, which would be out of place here. If programming language constructs (such as the Graal operator for addition) were confused with their denotations (such as mathematical addition), there would be no use or sense for formal semantic definitions.

How can we prove [9.7] assuming we know how to prove [9.6]? The rule of consequence is the normal mechanism: from the antecedents

[9.6]
$$\{x + x > 2\}\ y := x + x\ \{y > 1\}$$

[9.8]

$\{x > 1\}$ **implies** $\{x + x > 2\}$

direct application of the rule of consequence will yield [9.7].

This assumes that we can rely on the second antecedent, [9.8]. But we cannot simply accept [9.8] as a trivial property from elementary arithmetic. Actually, its formula does not even *belong* to the language of elementary arithmetic; as just recalled, it is not a mathematical property but a well-formed formula of the Graal assertion language. Deductions involving such formulae require an appropriate theory, transposing to programming language objects the properties of the corresponding objects in mathematics – integers, boolean values, real numbers.

When applied to actual programs, written in an actual programming languages and meant to be executed on an actual computer, this theory cannot be a blind copy of standard mathematics. For integers, it needs to take size limitations and overflow into account; this is the object of exercise 9.1. For "real" numbers, it needs to describe the properties of their floating-point approximations.

For the study of Graal, which only has integers, we will accept arithmetic at face value, taking for granted all the usual properties of integers and booleans. The rest of this discussion assumes that the theory of Graal is built on top of another axiomatic theory, called EM for elementary mathematics. EM is assumed to include axioms and inference rules applicable to basic Graal operators (+, –, <, >, **and** etc.) and reflecting the properties of the corresponding mathematical operators. Whenever a proof needs a property such as [9.8] above, the justification will simply be the mention "EM".

EM also includes properties of the mathematical implication operation, transposed to the **implies** operation on assertions. An example of such a property is the transitivity of implication: for any assertions P, Q, R,

$$\vdash ((P \textbf{ implies } Q) \textbf{ and } (Q \textbf{ implies } R)) \Longrightarrow (P \textbf{ implies } R)$$

Chapter 10 will use the denotational model to define the semantics of assertions in a formal way, laying the basis for a rigorously established EM theory, although this theory will not be spelled out.

Using EM, the proof that [9.7] follows from [9.6] may be written as:

T1	[9.6] $\{x + x > 2\}$ $y := x + x$ $\{y > 1\}$	(proved separately)
T2	$x > 1$ **implies** $x + x > 2$	EM
T3	[9.7] $\{x > 1\}$ $x := x + x$ $\{y > 1\}$	T1, T2; CONS

The EM rules used in this chapter are all straightforward. In proofs of actual programs, you will find that axiomatizing the various object domains may be a major part of the task. One of the proofs below (the "tower of Hanoi" recursive routine, 9.10.9), as well as exercises 9.25 and 9.26, provide examples of building theories adapted to specific

problems. But some object domains are hard to axiomatize. For example, producing a good theory for floating-point numbers and the associated operations, as implemented by computers, is a difficult task. Such problems are among the major practical obstacles that arise in efforts to prove full-scale programs.

9.6.4 The rule of conjunction

A language-independent inference rule similar in scope to the rule of consequence is useful in some proofs. This rule states that if you can derive two postconditions you may also derive their logical conjunction:

CONJ

$$\frac{\{P\}\ a\ \{Q\},\quad \{P\}\ a\ \{R\}}{\{P\}\ a\ \{Q\ \textbf{and}\ R\}}$$

Note that conversely if you have established $\vdash\ \{P\}\ a\ \{Q\ \textbf{and}\ R\}$, then you may derive both $\vdash\ \{P\}\ a\ \{Q\}$ and $\vdash\ \{P\}\ a\ \{R\}$. This property does not need to be introduced as a rule of the theory but follows from the rule of consequence, since the following are EM theorems:

$(Q\ \textbf{and}\ R)\ \textbf{implies}\ Q$

$(Q\ \textbf{and}\ R)\ \textbf{implies}\ R$

Wp-semantics, as studied later in this chapter, will enable us to determine whether there is a corresponding "rule of disjunction" for the **or** operator (see 9.9.4).

9.7 PRE-POST SEMANTICS OF GRAAL

We now have all the necessary background for the axiomatic theory of Graal instructions.

9.7.1 Skip

The first instruction to consider is *Skip*. The pre-post axiom schema is predictably neither hard nor remarkable.

$$\mathrm{A}_{Skip}\quad \vdash\quad \{P\}\ Skip\ \{P\}$$

Skip does not do anything, so what the user of this instruction may be guaranteed on exit is no more and no less than what he is prepared to guarantee on entry.

9.7.2 Assignment

The axiom schema for assignment uses the notion of **substitution**. For any assertion Q:

$A_{Assignment}$

$$\vdash \quad \{Q\ [x \leftarrow e\,]\} \quad \textit{Assignment}\ (\textit{target}: x; \textit{source}: e) \quad \{Q\}$$

This rule introduces a new notation: $Q\ [x \leftarrow e\,]$, read as "Q with x replaced by e", is the substitution of e for all occurrences of x in Q. This notation is applicable when Q and e are expressions and x is a variable; it immediately extends to the case when Q is an assertion.

Substitution is a purely textual operation, involving no computation of the expression: to obtain $Q\ [x \leftarrow e\,]$, you take Q and replace occurrences of x by e throughout. We need to define this notion formally, of course, but let us first look at a few examples of substitution. As mentioned above, the expressions apply arithmetic and relational operations in standard concrete syntax.

1 $\vdash \quad 3\ [x \leftarrow y + 1] \ = \ 3$

2 $\vdash \quad (z * 7)\ [x \leftarrow y + 1] \ = \ z * 7$

3 $\vdash \quad x\ [x \leftarrow y + 1] \ = \ y + 1$

4 $\vdash \quad (x^2 - x^3)\ [x \leftarrow y + 1] \ = \ (y + 1)^2 - (y + 1)^3$

5 $\vdash \quad (x + y)\ [x \leftarrow y + 1] \ = \ (y + 1) + y$

6 $\vdash \quad (x + y)\ [x \leftarrow x + y + 1] \ = \ (x + y + 1) + y$

In the first two examples, x does not occur in Q, so that $Q\ [x \leftarrow e\,]$ is identical to Q (a constant in the first case, a binary expression not involving x in the second). In example 3, Q is just the target x of the substitution, so that the result is e, here $y + 1$. In example 4, x appears more than once in Q and all occurrences are substituted. Example 5 shows a case when a variable, here y, appears in both Q and e; note that rules of ordinary arithmetic would allow replacement of the right-hand side by $2 * y + 1$, but this is outside the substitution mechanism. Finally, example 6 shows the important case in which x, the variable being substituted for, appears in e, the replacement.

We need a way to define substitution formally. Let

$$Q\ [x \leftarrow e\,] \ \triangleq \ \textit{subst}\ (Q, e, x\,\textbf{.}\textit{id})$$

where function *subst*, a simplified version of the substitution function introduced in 5.7 for lambda calculus (see figure 5.2), is defined by structural induction on expressions:

[9.9]

```
subst (Q: Expression, e: Expression, x: S) ≜
    case Q of
        Constant : Q |
        Variable : if Q.id = x then e else Q end |
        Binary :
            Expression (Binary (
                term1: subst (Q.term1, e, x);
                term2: subst (Q.term2, e, x);
                op: Q.op))
    end
```

We may need to compose substitutions, using the following rule:

[9.10]

$$(Q\ [a \leftarrow f])\ [b \leftarrow g] = Q\ [a \leftarrow (f\ [b \leftarrow g])]$$

This property does not hold in all cases (a counter-example is easy to produce) but is correct in the two cases for which we will need it: when a and b are the same identifier; and when b does not occur in Q. The proof by structural induction, using the definition of function *subst*, is the subject of exercise 9.20.

In the pre-post theory, *subst* will only be applied to boolean expressions (associated with assertions); but as these may be relational expressions involving sub-expressions of any type, we need *subst* to be defined for general expressions.

The pre-post axiom schema for assignment ($A_{Assignment}$) uses substitution to describe the result of an assignment. The idea is quite simple: whatever is true of x after the assignment $x := e$ must have been true of e before.

The following are simple examples of the use of the axiom schema. Carry out the substitutions by yourself to see the mechanism at work.

1 $\vdash$ $\{y > z - 2\}\ \ x := x + 1\ \ \{y > z - 2\}$

2 $\vdash$ $\{2 + 2 = 5\}\ \ x := x + 1\ \ \{2 + 2 = 5\}$

3 $\vdash$ $\{y > 0\}\ \ x := y\ \ \{x > 0\}$

4 $\vdash$ $\{x + 1 > 0\}\ \ x := x + 1\ \ \{x > 0\}$

Example 1 shows that an assertion involving only variables other than the target of an assignment is preserved by the assignment. The assertion of the second example only involves constants and is similarly maintained. Note that the rule says nothing about the precondition and postcondition being “true” or “false”: all that example 2 says is that if two plus two equaled five before the assignment this will still be the case afterwards – a theorem, although a useless one since its assumption does not hold.

Examples 3 and 4 result from straightforward application of substitution. For the latter, the assignment rule does not by itself yield a proof of

$$\{x > -1\} \quad x := x + 1 \quad \{x > 0\}$$

For this, EM and the rule of consequence are needed. The proof may be written as follows:

A1	$\{x + 1 > 0\} \quad x := x + 1 \quad \{x > 0\}$	$A_{Assignment}$
A2	$x > -1$ **implies** $x + 1 > 0$	EM
A3	$\{x > -1\} \quad x := x + 1 \quad \{x > 0\}$	A1, A2; CONS

Three important comments apply to the assignment rule.

First, the rule as given works **"backwards"**: it makes it possible to deduce a precondition $Q\ [v \leftarrow e\,]$ from the postcondition Q rather than the reverse. A forward rule is possible (see exercise 9.9), but it turns out to be less easy to apply. The observation that proofs involving assignments naturally work by sifting the postcondition back through the program to obtain the precondition has important consequences on the structure and organization of these proofs.

In a simple case, however, the backward rule yields an immediate forward property, If the source expression e for an assignment is a plain variable, rather than a constant or a composite expression, then for any assertion P:

[9.11]

$$\vdash \quad \{P\} \quad \textit{Assignment}\ (\textit{target}: x\ ;\ \textit{source}: e)\ \{P\ [e \leftarrow x\,]\}$$

provided x does not occur in P. To derive this, use $A_{Assignment}$, taking $P\ [e \leftarrow x\,]$ for Q; then $Q\ [x \leftarrow e\,]$ is P by the rule for composition of substitutions ([9.10], page 324), which is applicable here thanks to the assumption that x does not occur in P.

The second comment reflects on the nature of assignment. This instruction is one of the most imperative among the features that distinguish programming from the "applicative" tradition of mathematics (1.3). An assignment is a command, not a mathematical formula; it specifies an operation to be performed at a certain time during the execution of a program, not a relation that holds between mathematical entities. As a consequence, it may be difficult to predict the exact result of an assignment instruction in a program, especially since repeated assignments to the same variable will cancel each other's effect.

Axiom $A_{Assignment}$ establishes the mathematical respectability of assignment by enabling us to interpret this most unabashedly imperative of programming language constructs in terms of a "pure" – that is to say, applicative – mathematical concept: substitution.

The third comment limits the applicability of the rule. As given above, this rule only applies to languages (such as Graal) which draw a clear distinction between the notions of expression and instruction. In such languages, expressions produce values, with no effect on the run-time state of the program; in contrast, instructions may change the state, but do not return a value. This separation is violated if an expression may produce **side-effects**, usually through function calls. Consider for example a function

```
asking_for_trouble (x: in out INTEGER): INTEGER is
    do
        x := x + 1;
        global := global + 1;
        Result := 0
            -- The function's returns as result the final value of
            -- the predefined variable Result (Eiffel convention)
    end
```

where *global* is a variable external to *asking_for_trouble* in some fashion but declared outside of the scope of *asking_for_trouble*; for example *global* may be external in C, part of a *COMMON* in Fortran, declared in an enclosing block in Pascal, in the enclosing package in Ada or in the enclosing class in Eiffel. The following pre-post formulae are false in this case even though they would directly result from applying $A_{Assignment}$ (with a proper rule for functions):

$\{global = 0\}\ \ u := asking\_for\_trouble\ (a)\ \ \{global = 0\}$

$\{a = 0\}\ \ u := asking\_for\_trouble\ (a)\ \ \{a = 0\}$

It is possible to adapt $A_{assignment}$ to account for possible side-effect in expressions, but this makes the theory significantly more complex. Since, however, most programming languages allow functions to produce side-effects, we need a way to describe the semantics of the corresponding calls. A solution, already suggested in the discussion of denotational semantics (7.7.2), is to limit the application of $A_{Assignment}$ to assignments whose source expression does **not** include any function call. Then to deal with an assignment whose right-hand side is a function call, such as

[9.12]

```
y := asking_for_trouble (x)
```

we consider that, in abstract syntax, this is not an assignment but a routine call; the abstract syntax for such an instruction includes an input argument, here *x*, and an output result, here *y*. The instruction then falls under the scope of the inference rule for such routine calls, given later in this chapter (9.10.2).

Only for the purposes of a proof do you actually need to translate an assignment of the [9.12] form into a routine call; the translation, done in abstract syntax, leaves the original concrete program unchanged. (As noted in chapter 7, this is an example of the "two-tiered specifications" discussed in 4.3.4.)

Of course, functions which produce arbitrary side-effects are bad programming practice since they damage referential transparency. We should certainly not condone a function such as *asking_for_trouble*. But in practice many functions will need to change the state in some perfectly legitimate ways. For example any function that creates and returns a new object does perform a side-effect (by allocating memory), although from the caller's viewpoint it simply computes a result (the object) and is referentially transparent.

Because it is difficult to define useful universal rules for distinguishing between "good" and "bad" side-effects, most programming languages, even the few whose designers worried about the provability of programs, allow side-effects in functions, with few or no restrictions. To prove properties of assignments involving functions, then, you should treat them as routine calls using the transformation outlined above.

The existence of such a formal mechanism is not an excuse for undisciplined use of side-effects in expressions, especially those which do not even involve a function call, as with the infamous value-modifying C expressions of the form $x{+}{+}$ or $-{-}x$.

9.7.3 Dealing with arrays and records

The assignment axiom, as given above, is directly applicable to simple variables. How can we deal with assignments involving array elements or record fields?

Plain substitution will not work. Take for example the Pascal array assignment

$$t\ [\,i\,] := t\ [\,j\,] + 1$$

Then by naive application of axiom $\mathrm{A}_{\mathit{Assignment}}$ we could prove a property such as:

[9.13]
$$\{t\ [\,j\,] = 0\} \quad t\ [\,i\,] \ := \ t\ [\,j\,] + 1 \quad \{t\ [\,j\,] = 0\}$$

Here the substitution appears trivial since the assignment's target, $t\ [\,i\,]$, does not occur in the postcondition.

Unfortunately, the above is not a theorem since the assignment will fail to ensure the postcondition if $i = j$. The problem here is a fundamental property of arrays, dynamic indexing: when you see a reference to an array element, $t\ [\,i\,]$, the program text does not tell you which array element it denotes. So it is only at run time that you will find out whether $t\ [\,i\,]$ and $t\ [\,j\,]$ denote the same array elements or different ones. Such a situation, where two different program entities may at run time happen to denote the same object, is known as **dynamic aliasing**.

One solution is to consider an assignment to an array element as an assignment to the whole array. More precisely, we may treat this operation as a separate instruction, with abstract syntax

$$\mathit{Array\_assign} \ \triangleq \ \mathit{target}\colon \mathit{Variable}\ ;\ \ \mathit{index}\colon \mathit{Expression}\ ;\ \ \mathit{source}\colon \mathit{Expression}$$

The associated rule is a variant of $A_{Assignment}$:

A_{Array_assign}

$$\vdash \quad \{Q\ [t \leftarrow t\ (i : e)]\}\ Array\_assign\ (target : t\ ;\ index : i\ ;\ source : e) \quad \{Q\}$$

The new notation introduced, $t\ (i : e)$, denotes an array which is identical to t except that its value at index i is e. This property may be described by two axioms:

A_{Array}

$$\vdash \quad i \neq j \ \textbf{implies}\ t\ (i : e)\ [j] = t\ [j]$$

$$\vdash \quad i = j \ \textbf{implies}\ t\ (i : e)\ [j] = e$$

These rules yield the following two theorems (replacing [9.13]):

[9.14]

$$\vdash \quad \{i \neq j\ \textbf{and}\ t\ [j] = 0\} \quad t\ [i] := t\ [j] + 1 \quad \{t\ [j] = 0\}$$

$$\vdash \quad \{i = j\ \textbf{and}\ t\ [j] = 0\} \quad t\ [i] := t\ [j] + 1 \quad \{t\ [j] = 1\}$$

The proof is left as an exercise (9.10)

We may use a similar method to deal with objects of record types. (See also the denotational model in 7.2.) If x is such an object, and a is one of the component tags, we should treat the assignment $x.a := e$ as an assignment to x as a whole. In line with the technique used for arrays, $x\ (a : e)$ is defined as denoting an object identical to x except that its a component is equal to v. The axioms schemata for this operation are simpler with records than with arrays, as here there is no dynamic aliasing: an array index may only be known at run-time, but the tag of a reference to a record field is known statically[1].

A_{Record}

$$\vdash \quad (x\ (a : e)).b = x.b$$

$$\vdash \quad (x\ (a : e)).a = e$$

where: x is an object of a record type; a and b are different component tags of this type; dot notation $x.t$ denotes access to the component of x with tag t.

To obtain a variant of the assignment axiom applicable to record components, just imitate A_{Array_assign} after introducing the suitable abstract syntax.

[1] In object-oriented languages such as Eiffel or Smalltalk, the technique known as **dynamic binding** means that in some cases the actual tag must be computed at run-time.

9.7.4 Conditional

The remaining instructions are not primitive commands, but control structures used to construct complex instructions from simpler ones; as a consequence, their semantics is specified through inference rules (actually rule schemata) rather than axioms.

Here is the inference rule for conditionals:

$I_{Conditional}$

$$\frac{\{P \textbf{ and } c\}\ a\ \{Q\}, \qquad \{P \textbf{ and not } c\}\ b\ \{Q\}}{\{P\}\ \ Conditional\ (test\colon c;\ thenbranch\colon a;\ elsebranch\colon b)\ \{Q\}}$$

Let us see what this means. Assume you are requested to prove the correctness, with respect to P and Q, of the instruction given in abstract syntax at the bottom of the rule, which in more casual notation would appear as

if *c* **then** *a* **else** *b* **end**

Since the result of executing this instruction is to execute either a or b, you may proceed by proving separately that both a and b are correct with respect to P and Q; however in the case of a you may "and" the precondition with c, since this branch will only be executed when c is initially satisfied; and similarly with **not** c for the other branch.

As an example of using this rule, consider the proof of the following program fragment, which you may recognize as an extract from Euclid's algorithm, in its variant using subtraction rather than division. (The proof of the extract will be used later as part of the proof of the complete algorithm.)

[9.15]

$\{m, n, x, y > 0$ **and** $x \neq y$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$

if $x > y$ **then**

$\quad x := x - y$

else

$\quad y := y - x$

end

$\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$

where all variables are of type *INTEGER*, $gcd\ (u, v)$ denotes the greatest common divisor of two positive integers u and v and the notation $u, v, w, \ldots > 0$ is used as a shorthand for

$u > 0$ **and** $v > 0$ **and** $w > 0$ **and** ...

C1	$\{m, n, x - y, y > 0$ **and** $gcd\ (x - y, y) = gcd\ (m, n)\}$	
	$x := x - y$	
	$\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	$A_{Assignment}$
C2	$m, n, x, y > 0$ **and** $x \neq y$ **and** $gcd\ (x, y) = gcd\ (m, n)$ **and** $x > y$	
	implies	
	$m, n, x - y, y > 0$ **and** $gcd\ (x - y, y) = gcd\ (m, n)$	EM
C3	$\{m, n, x, y > 0$ **and** $x \neq y$ **and** $gcd\ (x, y) = gcd\ (m, n)$ **and** $x > y\}$	
	$x := x - y$	
	$\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	C1, C2; CONS
C4	$\{m, n, x, y - x > 0$ **and** $gcd\ (x, y - x) = gcd\ (m, n)\}$	
	$y := y - x$	
	$\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	$A_{Assignment}$
C5	$m, n, x, y > 0$ **and** $x \neq y$ **and** $gcd\ (x, y) = gcd\ (m, n)$ **and not** $x > y$	
	implies	
	$m, n, y - x, y > 0$ **and** $gcd\ (x, y - x) = gcd\ (m, n)$	EM
C6	$\{m, n, x, y > 0$ **and** $x \neq y$ **and** $gcd\ (x, y) = gcd\ (m, n)$ **and not** $x > y\}$	
	$y := y - x$	
	$\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	C4, C5; CONS
C7	$\{m, n, x, y > 0$ **and** $x \neq y$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	
	CONDIT	
	$\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	C3, C6; $I_{Conditional}$

Figure 9.3: Proof involving a conditional instruction

The proof of [9.15] is given in full detail on the adjacent page. *CONDIT* denotes the conditional instruction under scrutiny. P and Q being the precondition and postcondition, the proof proceeds by establishing two properties separately:

$\vdash \{P \textbf{ and } x > y\}\ x := x - y\ \{Q\}$ (Line C3)

$\vdash \{\text{P } \textbf{and } y > x\}\ y := y - x\ \{Q\}$ (Line C6)

Both cases are direct applications of the EM property that

[9.16]

$\vdash u > v > 0$ **implies** $gcd\ (u, v) = gcd\ (u - v, v)$

[9.16] as well as the precondition and postcondition of [9.15] illustrate the "unobtrusive approach" to undefinedness mentioned at the beginning of this chapter. The greatest common divisor of two integers is only defined if both are positive. To deal with this problem, the assertions of [9.15] include clauses, **and**ed with the rest of these assertions, stating that the elements whose gcd is needed are positive; the formula in [9.16] uses a similar condition as the left-hand side of an **implies**.

Rather than introducing explicit rules stating when an expression's value is defined and when it is not, it is usually simpler, as here, to permit the writing of potentially undefined expressions, but to ensure through the axioms and inference rules of the theory that one can never prove anything of interest about their values.

9.7.5 Compound

Two rules are needed to deal with compound instructions. The first, an axiom schema, expresses that a zero-element compound is equivalent to a *Skip*:

$A0_{Compound}$

$$\vdash \{P\}\ Compound\ (<>)\ \{P\}$$

The second rule enables us to to combine the properties of more than one compound. It assumes c is a compound and a is an instruction.

$I_{Compound}$

$$\frac{\{P\}\ c\ \{Q\}, \qquad \{Q\}\ a\ \{R\}}{\{P\} \quad c \ ++ <a> \quad \{R\}}$$

The derivation shown below illustrates the technique for proving properties of compounds, based on these two rules. The property to prove is

$$\{m, n > 0\}\ x := m;\ y := n\ \{m, n, x, y > 0 \textbf{ and } gcd\ (x, y) = gcd\ (m, n)\}$$

S1	$\{m, n > 0\}$ **implies** $\{m, n, m, n > 0$ **and** $gcd\ (m, n) = gcd\ (m, n)\}$	EM
S2	$\{m, n, m, n > 0$ **and** $gcd\ (m, n) = gcd\ (m, n)\}$ $x := m$ $\{m, n, x, n > 0$ **and** $gcd\ (x, n) = gcd\ (m, n)\}$	$A_{Assignment}$
S3	$\{m, n > 0\}$ $x := m$ $\{m, n, x, n > 0$ **and** $gcd\ (x, n) = gcd\ (m, n)\}$	S1, S2; CONS
S4	$\{m, n, x, n > 0$ **and** $gcd\ (x, n) = gcd\ (m, n)\}$ $y := n$ $\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	$A_{Assignment}$
S5	$\{m, n > 0\}$ $x := m; y := n$ $\{m, n, x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	S3, S4; $I_{Compound}$

Figure 9.4: Proof involving a compound instruction

9.7.6 Loop

The last construct to study is the loop, for which the rule is predictably more delicate. It is an inference rule, as follows:

I_{Loop}

$$\frac{\{I \textbf{ and } c\}\ b\ \{I\}}{\{I\}\ Loop\ (test: c; body: b)\ \{I\ \textbf{ and not }\ c\}}$$

This rule embodies two properties of loops. In concrete syntax, the loop considered is

while *c* **do** *b* **end**

First, the postcondition includes **not** *c* because the continuation condition *c* will not hold upon loop exit (otherwise the loop would have continued). Note that I_{Loop} is a partial correctness rule, which is of little interest if the loop does not terminate. You must prove termination separately, using techniques explained below.

The second property relates to an assertion I, called a loop **invariant**, which is assumed to be such that:

$\{I$ **and** $c\}\ b\ \{I\}$

In other words, if I is satisfied before an execution of b, I will still be satisfied after that execution – hence the name "invariant". The actual precondition in this hypothesis is actually not just I but I **and** c since executions of b are of interest only when they occur as part of loop iterations, that is to say when c is satisfied. The rule expresses that if the truth of I is maintained by one execution of b (under c), then it will also be maintained by any number of executions of b, and hence by a loop having b as body.

The expressions I **and** c and I **and not** c appearing in I_{Loop} are a slight abuse of language since **and** and **not** as defined (page 318) take assertions as operands, whereas c is just a Graal boolean expression. The correct notations would use *Assertion* (*exp*: c) rather than c.

What is a loop invariant? The consequent of the rule gives a hint. Its postcondition represents what the loop is supposed to achieve, its "goal". This goal is

I **and not** c

which makes the invariant I appear as a weakened form of the goal. But I is also the precondition of the consequent. This means that I is weak enough to be satisfied in the state preceding execution of the loop, but strong enough to yield the desired goal on exit when combined with the exit condition.

As an example, take Euclid's algorithm for computing the greatest common divisor of two positive integers m and n:

```
x := m; y := n;
while x ≠ y loop
        if x > y then
                x := x – y
        else
                y := y – x
        end
end;
g := x
```

The proofs of the previous examples show that that this loop admits the following property as invariant:

[INV]

$x > 0$ **and** $y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)$

The invariant is satisfied before the loop begins, since by straightforward application of $A_{Assignment}$, $I_{Compound}$ and EM:

$$\begin{array}{ll} \vdash & \{m > 0 \textbf{ and } n > 0\} \\ & x := m \ ; \ y := n \\ & \{x > 0 \textbf{ and } y > 0 \textbf{ and } gcd\ (x, y) = gcd\ (m, n)\} \end{array}$$

So on loop exit we may infer both the invariant, hence $gcd\ (x, y) = gcd\ (m, n)$, and the exit condition $x = y$; the conjunction of these assertions implies $x = y = gcd\ (m, n)$.

Note how INV matches the informal notion of a "weakened form of the goal":

- INV yields $x = gcd\ (m, n)$, that is to say essentially the goal, when $x = y$.
- But INV is also weaker (more general) than this goal. In fact it is weak enough to be satisfied trivially by taking $x = m, y = n$.

You may consider an execution of the loop, then, as a process designed to maintain INV, making it a little stronger (closer to the goal) on each iteration. The last part of this chapter (9.11) shows how this view leads to a systematic approach for building correct software.

Below is the formal proof. It uses a few abbreviations: *LOOP* for the loop, *CONDIT* for the loop body (which is the conditional instruction studied previously), and *EUCLID* for the whole program fragment.

L1	$\{m, n > 0\}$ $x := m; y := n$ $\{x, y > 0$ **and** $gcd\ (x, y) = gcd\ (m, n)\}$	S5 (see page 332), CONS
L2	$\{m, n > 0\}\ x := m; y := n$ {INV}	L1, definition of INV
L3	{INV **and** $x \neq y$} *CONDIT* {INV}	C7 (see page 329), CONS
L4	{INV} *LOOP* {INV **and** $x = y$}	I_{Loop}
L5	$\{m, n > 0\}\ x := m; y := n;$ *LOOP* {INV **and** $x = y$}	L2, L4; $I_{Compound}$
L6	$\{x, y > 0$ **and** $x = y$ **implies** $gcd\ (x, y) = x$	EM
L7	INV **and** $x = y$ **implies** $gcd\ (m, n) = x$	L6, definition of INV, EM
L8	{INV **and** $x = y$} $g := x$ $\{g = gcd\ (m, n)\}$	L7; CONS
L9	$\{m, n > 0\}$ *EUCLID* $\{g = gcd\ (m, n)\}$	L5, L8; CONS

Figure 9.5: Proof involving a loop

9.7.7 Termination

The previous rules are partial correctness rules; this leaves the termination problem open. As programmers know all too well, loops may fail to terminate; they are the only construct studied so far that introduces this possibility, although of a course a compound or conditional may also not terminate if one of its constituents does not.

The inference rule for loops, I_{Loop}, is clearly applicable to terminating constructs only. Otherwise the problem of automatic programming would be easy: to solve any computing problem characterized by an output condition Q, use a program of the form

while not *Q* **loop** *Skip* **end**

Using any assertion as invariant, rule I_{Loop} makes it possible to infer Q upon exit. The problem, of course, is that usually there will be no exit at all. The above loop rule is of no help here.

To prove termination, you may attempt to find a suitable **loop variant**, according to the following definition.

> **Definition** (Variant): A variant for a loop is an expression V of type integer, involving some of the program's variables, and whose possible run-time values may be proved to satisfy the following two properties:
>
> - The value of V is non-negative before the execution of the loop.
> - If the value of V is non-negative and the loop continuation condition is satisfied, an execution of the loop body will decrease the value of V by at least one while keeping it non-negative.

If these two conditions are satisfied, execution of the loop will clearly terminate, since you cannot go on indefinitely decreasing the value of an integer expression which never becomes negative.

As an example, the expression

$$V \triangleq \max(x, y)$$

is an appropriate variant for the loop in the *EUCLID* program fragment.

More general variants may be used: rather than integer, the type of the variant expression could be any well-founded set, that is to say any set in which every decreasing sequence is finite. An example of well-founded set other than **N** is the set of nodes of a possibly infinite tree, where $m \leq n$ is defined as "m is an ancestor of n or n itself". However the use of integer variants entails no loss of generality: if v is a variant in any well-founded set, then there is also an integer variant $|v|$, defined for any value n of v as the longest length of a decreasing sequence starting at n.

A variant may be viewed as an expression of the program's variables which, prior to each iteration of the loop, provides an upper bound on the number of remaining iterations. To prove that the loop terminates, you must exhibit such a bound. If your sole purpose is to prove termination, you are not required to guarantee that the bound is close to the actual number of remaining iterations, but a close enough bound will help you estimate the program's efficiency.

This informal description of the method for proving termination may now be made more precise. Consider a pre-post formula of the form

$$\{P\}\ \mathit{Loop}\ (\mathit{test}: c\ ;\ \mathit{body}: b\ )\ \{Q\}$$

To prove rigorously that the loop terminates when started with precondition P satisfied, you must find an appropriate variant expression V and prove the following properties:

[9.17]
$$P \quad \textbf{implies} \quad (V \geq 0)$$

[9.18]
$$\{(V \geq 0)\ \textbf{and}\ c\ \} \quad z := V;\ b \quad \{0 \leq V < z\ \}$$

Here z is assumed to be a fresh variable of type integer, not appearing in the loop or the rest of the program. This variable is used to record the value of the variant before execution of the loop body b, to express that b decreases V strictly. The concrete form $z := V;\ b$ has been used as an abbreviation for

$$\mathit{Compound}\ (\langle \mathit{Instruction}\ (\mathit{Assignment}\ (\mathit{source}: V\ ;\ \mathit{target}: z\ )),\ b \rangle)$$

Using this method, you are invited to carry out formally the proof that *EUCLID* terminates.

> **Note**: it seems useless to have the condition $V \geq 0$, rather than just $V > 0$, in the precondition of [9.18]. Since the postcondition shows that V is decreased by at least one, V could not possibly have had value 0 before the execution of b. Can you see why it is in reality essential to use ≥ rather than >? (For an answer, see exercise 9.17).

The termination rule may be merged with $\mathrm{I}_{\mathit{Loop}}$ to yield a total correctness rule:

$\mathrm{IT}_{\mathit{Loop}}$ -- T for termination: total correctness rule

$$\frac{(I\ \textbf{and}\ c\ )\ \textbf{implies}\ V > 0, \qquad \{I\ \textbf{and}\ c\ \}\ \ z := V;\ b\ \ \{I\ \textbf{and}\ (V < z)\}}{\{I\}\ \mathit{Loop}\ (\mathit{test}: c\ ;\ \mathit{body}: b\ )\ \{I\ \ \textbf{and not}\ \ c\}}$$

Recall that I (the invariant) must be an assertion, treated here as a boolean expression, V (the variant) is an integer expression, and z is a fresh integer variable not appearing elsewhere in the program fragment considered.

The new rule handles both partial correctness and termination. As compared to [9.18], the second antecedent of the rule has a simpler postcondition: $0 \leq V$ is not needed there any more, since (because of the first antecedent):

I **and** $(V \leq 0)$ **implies** (**not** c)

so that the loop stops whenever V becomes non-positive. In essence, the condition c **implies** $(V > 0)$ has been integrated into the invariant.

This rule completes the pre-post theory of basic Graal. The treatment of other language features in this framework (arrays, pointers and procedures) will be outlined below (9.10).

9.8 THE CALCULUS OF WEAKEST PRECONDITIONS

The previous section has given the pre-post semantics of the Graal constructs. It is interesting to consider these constructs again from a complementary viewpoint: weakest precondition semantics.

9.8.1 Overview and definitions

In pre-post semantics, the assertions used to characterize an instruction are not necessarily the most "interesting" ones. For a formula of pre-post semantics:

[9.19]
$$\vdash \quad \{P\}\ a\ \{Q\}$$

the rule of consequence will also yield

[9.20]
$$\vdash \quad \{P'\}\ a\ \{Q'\}$$

for any P' stronger than P and Q' weaker than Q ("stronger" and "weaker" were defined on page 319). Thus [9.19] may be said to be more interesting than [9.20], as the latter may be derived from the former and so is less informative. More generally, we may define "more interesting than" as an order relation between formulae: the weaker the precondition, and the stronger the postcondition, the more interesting (informative) the formula.

When the aim is to prove specific properties of a given program, we often need to derive formulae which are not the most interesting among all possible ones. But when we define the semantics of a language we should look for the most interesting statements about the instructions of that language. All axioms and inference rules given in the preceding section, except for the properties of loops, are indeed "most interesting" specifications of instructions, in the sense that the given Q is the strongest possible postcondition for the given P, and P is the weakest possible precondition for Q.

Even so, we have many possible choices of pre-post pairs to characterize any particular instruction. It is legitimate to restrict the potential for arbitrary choice by fixing one of the two assertions. For example any instruction a may be characterized by the answer to either of the following questions:

- For an arbitrary assertion P, what is the strongest assertion Q such that $\{P\}\ a\ \{Q\}$?
- For an arbitrary assertion Q, what is the weakest assertion P such that $\{P\}\ a\ \{Q\}$?

In both cases, we view an instruction as an **assertion transformer**, that is to say, a mechanism that associates with a given precondition or postcondition the most interesting postcondition or precondition (respectively) which corresponds to it through the instruction.

The method to be described now, due to Dijkstra, follows this approach. Of the two questions asked, the more fruitful turns out to be the second: given an instruction and a postcondition, find the weakest precondition. This is due to both a technical and a conceptual reasons.

- The technical reason, already apparent in the above presentation of pre-post semantics, is that for common languages it is easier to express preconditions as functions of postconditions than the reverse.
- The conceptual reason has to do with the use of axiomatic techniques for program construction. A program is built to satisfy a certain goal, expressed as a property of the output results – that is to say, a certain postcondition. It is natural to construct the program by working backwards from the postcondition, choosing preconditions as the weakest possible (least committing on the input data).

The weakest precondition approach is based on these observations: it defines the semantics of a programming language through a set of rules which associate with every construct an assertion transformer, yielding for any postcondition the weakest corresponding precondition.

Three other features distinguish the theory given below from the above pre-post theory: it does not require, at least in principle, the invention of a variant and invariant; it directly handles total correctness; finally, it deals with non-deterministic constructs (which, however, could also be specified with pre-post semantics).

9.8.2 Basic definitions

The basic objects of the theory are **wp-formulae** (wp for weakest precondition) written under the general form

[9.21]

$$a \ \textbf{wp} \ Q$$

where a is an instruction and Q an assertion. Such a formula denotes, not a property which is either true or false (as was the case with a pre-post formula), but an **assertion**, defined as follows:

[9.22]

> **Definition** (Weakest Precondition): The wp-formula a **wp** Q, where a is an instruction and Q an assertion, denotes the weakest assertion P such that a is totally correct with respect to precondition P and postcondition Q

The expression "calculus of weakest preconditions" indicates the ambition of the theory: to provide a set of rules for manipulating programs and their associated assertions in a purely formal way, similar to how mathematical formulae are manipulated in ordinary arithmetic or algebra. To compute an expression such as $(x^2 - y^3)^2$, you merely apply well-defined transformation rules; these rules are defined by structural induction on the structure of expressions. Similarly, the calculus of weakest preconditions provides rules for computing a **wp** Q for a class of instructions a and assertions Q; the rules defined inductively on the structure of a and Q if a and Q are complex program objects.

Unfortunately, the calculus of programs and assertions is not as easy as elementary arithmetic; computing a **wp** Q remains a difficult or impossible endeavor as soon as a contains a loop. The theory nevertheless yields important insights and is particularly useful in connection with the constructive approach to program correctness, discussed below in 9.11.

9.8.3 True and false as postconditions

The theory relies on a set of simple axioms. The first one, called the "Law of the excluded miracle" by Dijkstra, states that no instruction can ever produce the postcondition *False*:

[9.23]

$$\vdash \quad a \ \textbf{wp} \ \mathit{False} = \mathit{False}$$

This is an axiom schema, applicable to any instruction *a*. In words: The weakest precondition that ensures satisfaction of *False* after execution of *a* is *False* itself. Since it is impossible to find an initial state for which *False* is satisfied, there is no state from which *a* will ensure *False*.

Having seen that, for consistency, *a* **wp** *False* must be *False* for any instruction *a*, you may legitimately ask what *a* **wp** *True* is. *True* is the assertion that all states satisfy. But do not conclude hastily that *a* **wp** *True* is *True* for any *a*. If an instruction is started in a state satisfying *True*, that is to say in any state, the final state will indeed satisfy *True* – provided there is a final state; in other words, provided *a* terminates. So *a* **wp** *True* is precisely the weakest precondition that will ensure termination of *a*.

This property is the first step towards establishing the calculus of weakest preconditions as a theory not just of correctness but of total correctness.

9.8.4 The rule of consequence

It is interesting to see how the rule of consequence (CONS, page 320) appears in this framework:

[9.24]

$$\frac{Q \textbf{ implies } Q'}{(a \textbf{ wp } Q) \textbf{ implies } (a \textbf{ wp } Q')}$$

In words: if *Q* is stronger than *Q'*, then any initial condition which guarantees that instruction *a* will terminate in a state satisfying *Q* also guarantees that *a* will terminate in a state satisfying *Q'*. That is to say, one may derive new properties from "more interesting" ones.

9.8.5 The rule of conjunction

The rule of conjunction has a similarly simple wp-equivalent:

[9.25]

$$\vdash \quad a \textbf{ wp } (Q \textbf{ and } Q') = (a \textbf{ wp } Q) \textbf{ and } (a \textbf{ wp } Q')$$

You are invited to study by yourself the practical meaning of this rule.

9.8.6 The rule of disjunction

The corresponding rule for boolean "or" may be written as:

[9.26]

$$\vdash \quad a \textbf{ wp } (Q \textbf{ or } Q') = (a \textbf{ wp } Q) \textbf{ or } (a \textbf{ wp } Q')$$

This rule requires more careful examination; as will turn out, it is satisfied for Graal and ordinary languages, but not for more advanced cases. At this point you are invited to ponder the meaning of this rule and decide for yourself whether it is a theorem or not. (For an answer, see 9.9.4 below.)

9.8.7 *Skip* and *Abort*

We are now ready to start studying the wp-rules for language constructs. The axiom schema for the *Skip* instruction is predictably trivial:

$$\vdash \quad \mathit{Skip}\ \mathbf{wp}\ Q = Q$$

for any assertion Q.

An instruction (not present in Graal) that would do even less than *Skip* is *Abort*, characterized by the following axiom schema:

$$\vdash \quad \mathit{Abort}\ \mathbf{wp}\ Q = \mathit{False}$$

In other words, *Abort* cannot achieve *any* postcondition Q – not even *True*, the least committing of all. Quoting from [Dijkstra 1976]:

> *This one cannot even "do nothing" in the sense of "leaving things as they are"; it really cannot do a thing.*

"Leaving things as they are" is a reference to the effect of *Skip*.

You may picture *Abort* as a non-terminating loop: by failing to yield a final state, it fails to ensure any postcondition at all. But this view, although not necessarily wrong, is overspecifying: all the rule expresses is the impossibility of proving anything of interest about *Abort*. This is another example of the already noted unobtrusiveness of the axiomatic method, where the "meaning" of a program consists solely of what you may prove about it. What *Abort* "does" practically, like looping forever or crashing the system, is irrelevant to the theory. To paraphrase Wittgenstein's famous quote: What one cannot prove about, one must not talk about.

9.8.8 Assignment

The wp-rule for assignments is:

[9.27]

$$\vdash \quad \mathit{Assignment}\,(\mathit{target}: x\ ;\ \mathit{source}: e)\ \ \mathbf{wp}\ \ Q\ =\ Q\ [x \leftarrow e\,]$$

This is the same as the corresponding pre-post rule ($A_{Assignment}$, page 323), with the supplementary information that $Q\ [x \leftarrow e\,]$ is not just one possible precondition but the most interesting – weakest.

9.8.9 Conditional

The wp-rule for conditional instructions is also close to the pre-post rule ($I_{Conditional}$, page 329):

[9.28]

$$\vdash \quad \mathit{Conditional}\ (\mathit{test}: c\ ;\ \mathit{thenbranch}: a\ ;\ \mathit{elsebranch}: b)\ \mathbf{wp}\ Q\ =$$
$$(c\ \mathbf{implies}\ (a\ \mathbf{wp}\ Q))\ \mathbf{and}\ ((\mathbf{not}\ c)\ \mathbf{implies}\ (b\ \mathbf{wp}\ Q))$$

In words: for the instruction **if** *c* **then** *a* **else** *b* **end** to terminate in a state where *Q* is satisfied, the two possible scenarios in the initial state — *c* satisfied, *c* not satisfied — must both lead to a final state where *Q* is satisfied; in other words, the initial state must satisfy both of the following properties:

- If *c* is satisfied, the condition for *a* to terminate and ensure a state where *Q* is satisfied.
- If *c* is not satisfied, the condition for *b* to terminate in a state where *Q* is satisfied.

As before, this wp-rule expresses that the precondition of the pre-post rule was weakest. Yet here it also includes something else: a termination property. The rule implies that a conditional instruction will terminate if and only if every branch terminates whenever its guard is true. Here the "guard" of a branch is the condition under which it is executed (*c* for the *thenbranch* and **not** *c* for the *elsebranch*).

9.8.10 Compound

For compounds too the wp-rules directly reflect the pre-post rules (page 331):

[9.29]

$$\vdash \quad \mathit{Compound}\ (<>)\ \mathbf{wp}\ Q\ =\ Q$$

[9.30]

$$\vdash \quad (c\ +\!+\ <a>)\ \mathbf{wp}\ Q\ =\ c\ \mathbf{wp}\ (a\ \mathbf{wp}\ Q)$$

In the second rule, *c* is an arbitrary compound and *a* an arbitrary instruction.

9.8.11 Loop

We may expect the rule for loops to be more difficult; also, it is interesting to see how the theory handles total correctness.

The basic wp-rule for loops is:

[9.31]

⊢

given

$l \triangleq Loop\ (test: c\ ;\ body: b)$ -- i.e. **while** c **do** b **end**

$G_0 \triangleq$ **not** c **and** Q ;

$G_i \triangleq c$ **and** $(b$ **wp** $G_{i-1})$ -- for $i > 0$

then

l **wp** $Q = \exists\ n: \mathbf{N} \bullet G_n$

end

The rule may be explained as follows. For the loop to terminate in a state satisfying Q, it must do so after a finite number of iterations. So the weakest precondition is of the form

G_0 **or** G_1 **or** G_2 **or** ...

where, for $i \geq 0$, G_i is the weakest precondition for the loop to terminate after exactly i iterations in a state satisfying Q. A loop started in an initial state σ terminates after i iterations in a state satisfying Q if and only if:

- For $i = 0$:
  - No iteration is performed, so σ satisfies **not** c.
  - σ satisfies Q.
- For $i > 0$:
  - One iteration is performed, so σ satisfies c.
  - This iteration brings the computation to a state from which the loop performs exactly $i-1$ further iterations and then terminates in a state satisfying Q: in other words, σ satisfies b **wp** G_{i-1}.

By combining these cases, we obtain the above inductive definition of G_i.

Rather than G_i, it is sometimes more convenient to use H_i, the condition for l to yield Q after **at most** i iterations (see exercise 9.17).

Rule [9.31] addresses total correctness. But it is not directly useful in practice: to check whether l **wp** Q is satisfied it would require you to check a potentially infinite number of conditions. This can only be done through a proof by induction, which in fact amounts to using an invariant and variant, as with the pre-post approach. The connection between the wp-rule and the pre-post rules is expressed by the following theorem, which reintroduces the invariant I and the variant V:

[9.32]

Theorem (Invariant and variant in wp-semantics): Let l be a loop with body b and test c, I an assertion and V an integer-valued function of the state. If for any value $z \in \mathbf{N}$

$(I$ **and** c **and** $V = z)$ **implies**

$((z > 0)$ **and** $(b$ **wp** $(I$ **and** $(0 < V < z))))$

then:

I **implies** $(l$ **wp** $(I$ **and not** $c))$

This theorem is equivalent to the inference rule IT_{Loop} (page 336). Its proof requires an appropriate model of the axiomatic theory, which will be introduced in the next chapter.

9.8.12 A concrete notation for loops

In a systematic approach to program construction, you should think of loop variants and invariants not just as "decoration" to be attached to a loop if a proof is required, but as components of the loop, conceptually as important as the body or the exit condition. Abstract and concrete syntax should reflect this role.

Whenever the rest of this chapter needs to express loops in concrete syntax, it will use the Eiffel notation for loops, which is a direct consequence of the above discussion and looks as follows:

```
from
        Compound                - - Initialization
invariant
        Assertion
variant
        Integer_expression
until
        Boolean_expression      - - Exit condition
loop
        Compound                - - Loop body
end
```

Like other uses of assertions in Eiffel, the **invariant** and **variant** clauses are optional.

The execution of such a loop consists of two parts, which we may call A and B. Part A simply executes the initialization *Compound*. Part B does nothing if the exit condition is satisfied; otherwise it proceeds with the loop body, and starts part B again.

In other words this is like a Pascal or Graal "while" loop, with its initialization included (**from** clause), and an exit test rather than a continuation test.

The reason for including the initialization follows from the axiomatic semantics of loops as studied above: every loop must have an initialization, whose aim is to ensure the initial validity of the invariant. (In some infrequent cases where the context of the loop guarantees the invariant the initialization *Compound* is empty.)

The reason for using an exit condition (**until** rather than **while**) is to make immediately visible what the outcome of the loop will be. By looking at the loop, you see right away the postcondition that will hold on loop exit:

$$G \triangleq I \textbf{ and } E$$

where I is the invariant and E the exit condition.

In the constructive approach, as discussed below (9.11), we will design loop algorithms by starting from G, the **goal** of the algorithm, and deriving I and E through various heuristics.

9.9 NON-DETERMINISM

A class of constructs enjoys a particularly simple characterization by wp-rules (although pre-post formulae would work too): non-deterministic instructions. A non-deterministic instruction is one whose effect is not entirely characterized by the state in which it is executed.

Simple examples of non-deterministic instructions are the guarded conditional and the guarded loop.

9.9.1 The guarded conditional

In concrete syntax (see exercise 9.2 for abstract syntax), the guarded conditional may be written as follows:

[9.33]

if
$c_1 : a_1 \ \square$
$c_2 : a_2 \ \square$
$\ldots \ \square$
$c_n : a_n$
end

where there are n branches ($n \geq 0$). The c_i, called guards, are boolean expressions, and the a_i are instructions.

Informally, the semantics of this construct is the following: the effect of the instruction is undefined if it is executed in a state in which none of the guards is true; otherwise, execution of the instruction is equivalent to execution of one a_i such that the corresponding guard c_i is true.

The standard **if** *c* **then** *a* **else** *b* **end** of Graal and most common languages may be expressed as a special case of this construct:

[9.34]

if
c : a □
not c : b
end

The guarded conditional has three distinctive features: first, it treats the various possible cases in a more symmetric way than the **if**...**then**... **else**... conditional; second, it is non-deterministic; third, it may fail – produce an undefined result.

The first property, symmetry, follows directly from the above informal specification. The non-determinism comes from the absence in that specification of any prescription as to which of all possible branches is selected when more than one guard is true. So the instruction

[9.35]

if
$x \geq 0$: $x := x + 1$ □
$x \leq 0$: $x := x - 1$
end

could yield $x = -1$ as well as $x = +1$ when started with $x = 0$.

This suggests the following axiom schema, which is both a generalization and a simplification of the wp-rule for the standard conditional ([9.28], page 342); *guarded_if* denotes the above construct [9.33].

[9.36]

$\vdash$
guarded_if **wp** Q =
(c_1 **or** c_2... **or** c_n) **and**
(c_1 **implies** (a_1 **wp** Q)) **and**
(c_2 **implies** (a_2 **wp** Q)) **and**
... **and**
(c_n **implies** (a_n **wp** Q))

Note how simply this axiom expresses the non-determinism of the construct's informal semantics. For *guarded_if* to ensure satisfaction of Q, there must be a branch whose guard c_i is true and whose action a_i ensures Q. There may be more than one such branch; if so, it does not matter which one is selected, as only the result, Q, counts. The axiom states this through the first **and** clause.

The axiom also captures the last of the construct's three key properties listed above: regardless of Q, the weakest precondition is *False* – that is to say, non-satisfiable by any state – if none of the c_i is true. This means that *guarded_if* is informally equivalent in this case to *Abort*, since we may not prove anything about it.

Many people are shocked by this convention when they first encounter the symmetric **if**: should *guarded_if* not behave like *Skip*, not *Abort*, when no guard is satisfied?

There are serious arguments, however, for the interpretation implied by [9.36]. One of the dangers of the **if** ... **then** ... **else** construct is that it lumps the last case with all unforeseen cases in the **else** branch. More precisely, assume a programmer has identified n cases for which a different treatment is required. The usual way to write the corresponding instruction is the following (using the Algol 68-Ada-Eiffel abbreviation **elseif** to avoid useless nesting of conditionals):

if c_1 **then** a_1

elseif c_2 **then** a_2

…

elseif c_{n-1} **then** a_{n-1}

else a_n **end**

-- No need to specify that the last branch corresponds to the case
-- c_n true, c_j false $(1 \leq j \leq n-1)$

The risk is to forget a case. When all of the c_i, including c_n, are false, the instruction executes a_n, which is almost certainly wrong; but the error may be hard to catch.

In the guarded conditional, on the other hand, every branch is explicitly preceded by its guard and executed only if the guard is true. If no guard is satisfied, a good implementation will produce an error message and stop execution, or raise an exception, or loop forever; this is better than proceeding silently with a wrong computation.

9.9.2 The guarded loop

The other basic non-deterministic construct is the guarded loop, which may be written as:

[9.37]

loop

$c_1 : a_1 \square$

$c_2 : a_2 \square$

$\ldots \square$

$c_n : a_n$

end

with the following informal semantics: if no c_i is true, the instruction does nothing; otherwise it executes one of the a_i such that c_i is true, and the process starts anew. The formal wp-rule for this construct is left for your pleasure (work from [9.31], page 342, and [9.36], page 346). The rule should make it clear, as [9.36], that it does not matter which branch is chosen when several are possible.

Here the case in which no c_i is satisfied is not an error but normal loop termination. In particular, for $n = 0$, the guarded loop is equivalent to *Skip*, not to *Abort* as with the guarded conditional.

9.9.3 Discussion

Why should one want to specify non-deterministic behavior? There are two main reasons.

The first reason is that non-deterministic programs may be useful to model non-deterministic behavior of the real world, as in real-time systems. (The non-determinism is not necessarily in the events themselves, but sometimes only in our perception of them; however the end result is the same.)

The second reason is the desire not to overspecify, mentioned at the start of this chapter: if it does not matter which branch of (say) a conditional is selected in a certain case, as both branches will lead to equally acceptable results, then the programmer need not choose explicitly. The goal here is abstraction: when a feature of the implementation is irrelevant to the specification, you should be able to leave it implicit.

How can we implement a non-deterministic construct such as the guarded conditional or loop? You must not think that such an implementation needs to use some kind of random mechanism for choosing between possible alternatives. All that the rules say is that whenever more than one c_i is satisfied, every corresponding a_i must yield the desired postcondition, and then any of the corresponding branches may be selected. Any implementation which observes this specification is correct.

Examples of correct implementations include: one that test the guards in the order in which they are written, and takes the first branch whose guard is true (as with the **if** … **then** … **elseif** …); one that starts from the other end; one that behaves like the first on even-numbered days and like the second on odd-numbered days; one that uses a random number generator to find the order in which it will evaluate the guards; one that starts n parallel processes to evaluate the guards (or asks n different nodes on a network), and chooses the branch whose guard is first (or last) computed as true; and many others.

No proof of properties of the construct is correct if it relies on knowledge about the actual policy used for choosing between competitive true guards. But as long as the proof is only based on the official, policy-independent rules, any implementation that abides by these rules is acceptable.

> One way to picture the situation is to imagine that a **demon** is in charge of choosing between acceptable branches when more than one guard is true. The demon does not have to be erratic, although he may well be; some demons are bureaucrats who always follow the same routine, others take pleasure in constantly changing their policies to defeat any attempt at second-guessing. But regardless of the individual psychology of the demon that has been assigned to us by the Central Office of Demon Services, he is in another room, and *we are not allowed to look.*

9.9.4 The rule of disjunction

The above remarks are the key to the pending issue of the rule of disjunction ([9.26], page 340). The (so far tentative) rule may be written as

[9.38]

given

$lhs \triangleq a$ **wp** $(Q$ **or** $Q')$;

$rhs \triangleq (a$ **wp** $Q)$ **or** $(a$ **wp** $Q')$

then

$lhs = rhs$

end

The rule expresses an equality between two assertions, that is to say a two-way implication: according to this rule, whenever a state satisfies *lhs*, it satisfies *rhs*, and conversely.

Look first at *rhs*. This assertion is true of states σ such that one or both of the following holds:

- a, started in σ, is guaranteed to terminate in a state satisfying Q.
- a, started in σ, is guaranteed to terminate in a state satisfying Q'.

Each of these conditions implies that a, started in σ, is guaranteed to terminate in a state satisfying Q **or** Q'; in other words, that state σ satisfies *lhs*. So *rhs* implies *lhs*.

Assume conversely that σ satisfies *lhs*. Instruction *a*, started in σ, is guaranteed to terminate in a state satisfying *Q* or satisfying *Q'*. Does this imply that σ is either guaranteed to terminate in a state satisfying *Q* or guaranteed to terminate in a state satisfying *Q'*? The answer is yes in the absence of non-deterministic constructs: since *a*, started in σ, is always executed in the same fashion, a guarantee that it ensures *Q* or *Q'* means either a guarantee that it ensures *Q* or a guarantee that it ensures *Q'*.

This is no longer true, however, if we introduce non-deterministic constructs. Assume for example that *a* is "toss a coin", *Q* is the property of getting heads and *Q'* of getting tails. Before tossing the coin you are guaranteed to get heads or tails: so *a* **wp** (*Q* **or** *Q'*) is true. But you are not guaranteed to get heads: thus *a* **wp** *Q* is false; so is *a* **wp** *Q'* and hence their **or**. Tossing a coin may be viewed as an implementation of the following program:

[*TOSS*]

if
    true : *produce_heads* □
    true : *produce_tails*
end

where

produce_heads **wp** (*result* = *heads*) = *True*
produce_heads **wp** (*result* = *tails*) = *False*
produce_tails **wp** (*result* = *heads*) = *False*
produce_tails **wp** (*result* = *tails*) = *True*

This discussion assumes a non-deterministic coin-tossing process, with an unpredictable result: the coin is tossed by a demon, who does not reveal his tossing policies. To see this formally note that

TOSS **wp** ((*result* = *heads*) **or** (*result* = *tails*))
    = *TOSS* **wp** *True*
    = *True*

but if we apply the non-deterministic conditional axiom ([9.33], page 345) we see that

TOSS **wp** (*result* = *heads*)
    = (*true* **implies** (*produce_heads* **wp** (*result* = *heads*)) **and**
      (*true* **implies** (*produce_tails* **wp** (*result* = *heads*))
    = (*produce_heads* **wp** (*result* = *heads*)) **and**
      (*produce_tails* **wp** (*result* = *heads*))
    = *True* **and** *False*
    = *False*

and *TOSS* **wp** (*result* = *tails*) is similarly *False*. By tossing a coin we are sure to get either heads or tails; but we can neither be sure to get heads nor be sure to get tails.

9.10 ROUTINES AND RECURSION

The presentation of language features in denotational semantics summarized the role of routines in software development (7.7.1). Now we must see what axiomatic semantics has to say about them.

This section will show how to derive properties of software elements containing routine calls, including recursive ones.

9.10.1 Routines without arguments

Consider first routines with no arguments and no results. The abstract syntax of a routine declaration is then just:

Routine $\triangleq$ *name*: *Identifier*; *body*: *Instruction*

and a routine call (new branch for the abstract syntax production describing instructions) just involves the name of the routine:

Call $\triangleq$ *called*: *Identifier*

Then a call instruction simply stands for the insertion of the corresponding routine body at the point of call. This is readily translated into an inference rule (for any routine declaration r):

I0$_{Routine}$

$$\frac{\{P\}\ r.body\ \{Q\}}{\{P\}\ Call\ (called:\ r.name)\ \{Q\}}$$

The rule expresses that any property of the body yields a similar property of the call.

9.10.2 Introducing arguments

Routines without arguments are not very exciting; let us see how arguments affect the picture. The discussion will first introduce arguments; then, as was done in 7.7 for the denotational specification, it will show how we can avoid complicating the specification by treating argument and result passing as assignment.

In many languages, the arguments to a routine may be of three kinds: "in" arguments, passed to the routine; results, also called "out" arguments, computed by the routine; and "in-out" arguments, which are both consumed and updated. In some languages such as Algol W or Ada, routine declarations qualify each argument with one of these modes.

As in 7.7.5, it is convenient to restrict the discussion to in arguments and out arguments, from now on respectively called arguments and results. Callers may still obtain the effect of in-out arguments by including variables in both the argument and result actual lists, subject to limitations given below.

A further simplification is to write every routine with exactly one (in) formal argument and one result, both being lists (finite sequences). This is not a restriction in practice since lists may have any number of elements. Proof examples given below in concrete syntax will follow the standard style, with individually identified arguments and result; but grouping arguments and results into two lists makes the theoretical presentation clearer.

Routines are commonly divided into functions, which return a result, and procedures, which do not. A procedure call stands for an instruction; a function call stands for an expression. Our routines cover both functions and procedures, but for consistency it will be preferable to treat all calls as instructions. A routine which represents a procedure will have an empty result list; a routine representing a procedure or function with no arguments will have an empty argument list.

The call instruction now has the following abstract syntax, generalized from the form without arguments given page 351:

Call ≜ *called*: *Identifier*; *input*: *Expression**; *output*: *Variable**

The elements of the actual *input* list may be arbitrary expressions, whose value will be passed to the routine; the actual *output* elements will have their value computed by the routine, so they must be variables (or, more generally, elements whose value may be changed at execution time, such as arrays or records).

For a routine f representing a function, a call using i as actual arguments and o as actual results, described in abstract syntax as

Call (*called*: f**.***name*; *input*: i; *output*: o)

corresponds to what is commonly thought of as an assignment instruction with a function call on the right-hand side:

$o := f\ (i)$

so that the discussion below will allow us to derive an axiomatic semantics for such assignments, which the earlier discussion (see page 326) had specifically excluded from the scope of the assignment axiom.

Because every routine has exactly one argument list and one result list, there is no need to make the formal argument and result explicit. We may simply keep the abstract syntax

Routine ≜ *name*: *Identifier*; *body*: *Instruction*

with the convention that every *body* accesses its argument and result lists through predefined names: *argument* and *result*. (Eiffel uses this convention for results, as seen

below.) We do not allow nesting of routine texts, so any use of these names unambiguously refers to the enclosing routine. We must of course make sure that no "normal" variable is called *argument* or *result*.

With these conventions, we may derive a first rule for routines with arguments by interpreting a call instruction

Call (*called*: *f*.*name*; *input*: *i*; *output*: *o*)

as the sequence of instructions (in mixed abstract-concrete syntax)

argument := *i*;

f.*body*;

o := *result*

(Section 7.7.6 gave a more precise equivalence, taking into account possible name clashes in block structure. The above equivalence suffices for this discussion.)

Taking this interpretation literally, assume that we know the axiomatics of the body in the form of a theorem or theorem schema

{*P*} *called*.*body* {*Q*}

Then we can use the assignment axiom to include the first instruction above, the initialization of *argument*:

{*P* [*argument* ← *i*]} *argument* := *i*; *called*.*body*; {*Q*}

It appears at first more difficult to include the final instruction, the assignment to *o*. But here *result* includes only variables, as opposed to composite expressions or constants; so the forward rule for assignment ([9.11], page 325) applies if *o* does not occur in *P*. It yields for the whole instruction sequence the pre-post formula

{*P* [*argument* ← *i*]}

argument := *i*; *called*.*body*; *o* := *result*

{*Q* [*result* ← *o*]}

In other words, taking the instruction sequence to represent the call: if the body is characterized by a precondition *P* and a postcondition *Q*, which may involve the local variables *argument* and *result* representing the arguments, then any call will be characterized by precondition *P* applied to the actual inputs *i* instead of *argument*, and postcondition *Q* applied to the actual results *o* instead of the formal *result*.

This yields the first version of the rule for routines with arguments, applicable to a non-recursive routine *r*. Some restrictions, given below, apply.

$I1_{Routine}$

$$\frac{\{P\}\ r.body\ \{Q\}}{\{P\ [argument \leftarrow i\,]\}\ Call\ (called: r.name;\ input: i;\ output: o)\ \{Q\ [result \leftarrow o\,]\}}$$

This rule admits a simple weakest precondition version:

given
$c \triangleq Call\ (called: r.name;\ input: i;\ output: o)$
then
c **wp** $Q' = (r.body$ **wp** $(Q'\ [o \leftarrow result\,]))\ [argument \leftarrow i\,]$
end

Here Q', an arbitrary assertion subject to the restrictions below, corresponds to $Q\ [result \leftarrow o\,]$ in the pre-post rule.

9.10.3 Simultaneous substitution

In rule $I1_{Routine}$ and its weakest precondition counterpart, the source and targets of substitutions are list variables, representing lists of formal and actual arguments and results. Since the original definition of substitution applied to atomic variables, we must clarify what the notation means for lists.

The generalization is straightforward: if *vl* is a list of variables and *el* a list of expressions, take

$$Q\ [vl \leftarrow el\,]$$

to be the result of replacing simultaneously in Q every occurrence of *vl* (1) with *el* (1), every occurrence of *vl* (2) with *el* (2) etc. For example:

$$(x + y)\ [<x, y> \leftarrow <3, 7>] = (3 + 7)$$

$$(x + y)\ [<x, y> \leftarrow <y, x>] = (y + x)$$

The simultaneity of substitutions is essential. In the second example, if the substitutions were executed in two steps in the order given, the first step would yield $(y + y)$, which the second would transform into $(x + x)$ – not the desired result. A formal definition of simultaneous substitution, generalizing the *subst* function for single substitutions ([9.9], page 323), is the subject of exercise 9.21.

The simultaneity requirement only makes sense if all the elements in the variable list *vl* are different: if x appeared as both $vl\ (j)$ and $vl\ (k)$ with $j \neq k$, the result would be ambiguous since you would not know whether to substitute $el\ (j)$ or $el\ (k)$ for x. The absence of duplicate variables is one of the constraints listed below on the application of $I1_{Routine}$.

The example proofs that follow list arguments individually, rather than collectively as list elements. To avoid introducing lists in such a case, it is convenient to use the notation

$$Q\ [x_1 \leftarrow a_1, ..., x_n \leftarrow a_n]$$

as a synonym for

$$Q\ [<x_1, ..., x_n> \leftarrow <a_1, ..., a_n>]$$

9.10.4 Conditions on arguments and results

The application of rule $I1_{Routine}$ assumes that the call satisfies some constraints. One, already noted, is absence of recursion; we shall see below how to make the rule useful for recursive routines. Let us first study the other seven constraints, which could be expressed as static semantic validity functions (exercise 9.19). The constraints are the following:

1 • No identifier may occur twice in the formal argument list.

2 • No identifier may occur twice in the formal result list.

3 • No identifier may occur in both the formal argument list and the formal result list.

4 • In any particular call, no variable may occur twice in the actual *output* result list.

5 • No variable local to the body of the routine may have the same name as a variable accessible to the calling program unit, unless it occurs in neither the precondition P nor the postcondition Q.

6 • No element of the *result* list may appear in P.

7 • If the postcondition Q involves any element of the *argument* list, then the corresponding element of the actual *input* list may not occur in the *output* list (that is to say, it may not be used as an in-out argument).

We already encountered the first four constraints in the denotational specification (7.7.4 and 7.7.7).

Constraints 1 and 2 follow directly from the consistency condition for simultaneous substitutions, as given above. From a more practical point of view, a duplicate identifier in the argument or result formal list would amount to a duplicate declaration; occurrences of the identifier in the routine body would then be ambiguous.

The latter observation also applies to an identifier appearing in both formal lists, justifying constraint 3.

Constraint 4 precludes any call which uses the same variable twice as actual result. Assume this constraint is violated and consider a routine

```
s (out x, y: INTEGER) is
        do              -- BODY
                x := 0;  y := 1
        end
```

whose body makes the following pre-post formula hold:

$$\vdash \quad \{true\}\ BODY\ \{x = 0 \ \textbf{and}\ y = 1\}$$

Then rule $I1_{Routine}$ could be used to deduce the contradictory result

$$\{true\}\ \textbf{call}\ s\ (a, a)\ \{a = 0 \ \textbf{and}\ a = 1\}$$

From a programmer's viewpoint, this means that the outcome would depend on the order in which the final values of the formal results (here x and y) are copied into the corresponding actual arguments on return from a call – a decision best left to the compiler writers and kept out of the language manual. Many programming language specifications indeed include constraint 4.

Nothing in constraints 1 to 4 precludes an expression from occurring more than once in the actual *input* list, or a variable from occurring in both the actual *input* and *output* lists. The latter case achieves the effect of in-out arguments.

Constraint 5 precludes sharing of local variable names between the routine and any of its callers. Such "puns" would cause incorrect application of the rule: if a local variable of r occurs in P or Q, then it will also occur in $P\ [i \leftarrow argument]$ or $Q\ [o \leftarrow result]$, and may yield an incorrect property of its namesake in the calling program. Lambda calculus raised similar problems (5.7).

A name clash of this kind, resulting from the independent choice of the same identifier in different program units, may be removed by manual renaming; more conveniently, compilers and formal proof systems can disambiguate the names statically. The denotational specification of block structure described one way of doing this in a formal system (see 7.2, especially 7.2.3). The same techniques could be applied here, removing the need for constraint 5. (This constraint had no equivalent in the denotational discussion of routines, which could afford to be more tolerant precisely because it assumed a block structure mechanism as a basis.)

Constraint 5 is not a serious impediment for programmers or program provers:

- It does not prevent routines from accessing global variables, as long as there is no name conflicts with locally declared variables. It is important to let routines access externally declared variables, especially if they use globals in the disciplined style enforced by object-oriented programming.
- If the local variable is used in neither the precondition nor the postcondition, no harm will result. In this case the computation performed by the routine uses the variable for internal purposes, but its properties do not transpire beyond the routine's boundaries. This means that constraint 5 is essentially harmless in practice, since meaningful pre-post assertions for a routine have no business referring to anything else than *argument, result* and global variables.

You may then interpret constraint 5 as a requirement on language implementers, specifying that each routine which uses a certain variable name must allocate a different variable for that name (and in the case of recursive routines, studied below, that every *call* to a routine must allocate a new instance of the variable).

Constraints 6 and 7 are in fact special cases of constraint 5, applying to the variable lists *argument* and *result*, implicitly declared in every routine, and hence raising many apparent cases of possible name clashes between routines and their callers. Constraint 6 was necessary to apply the forward assignment rule (page 353).

To avoid any harm from such clashes, we must first exclude any element of *result* from the precondition P; since *result* is to be computed by the routine, its presence in the precondition would be meaningless anyway. This is constraint 6.

The presence of *input* (or any of its elements) in Q is a problem only if the calling routine uses its own *input* (or the corresponding element of its *input*) as *result* (or part of it). Assume for example a routine with a single integer argument and a single integer result, whose body computes

result := *argument* + 1

Then with *true* as precondition P we may deduce

result = *argument* + 1

as postcondition Q. Now assume a call in which the caller's *argument* is used as both actual argument and actual result; this is expressly permitted, to allow the effect of in-out arguments. But then blind application of $I1_{Routine}$, without constraint 7, would allow us to infer, as postcondition for the call, that

argument = *argument* + 1

which is absurd. Constraint 7 specifically prevents this. It does not prohibit the presence of *argument* in Q if *argument* is not used as actual result for the call. As you are invited to check, this case does not raise any particular problem; nor does the possible presence of *argument* in P or *result* in Q.

9.10.5 A concrete notation for routines

Like invariants and variants for loops, routine preconditions and postconditions play such a key role in building, understanding and using software that they deserve to be part of the abstract and concrete syntax for routines, on a par with the argument list or the body.

When they need a concrete syntax, subsequent examples of routines will use the Eiffel notation, which results from the preceding discussion and rule $I1_{routine}$ (and supports the extended rule for recursive routines given below). An Eiffel routine is of the form

routine_name (*argument*: *TYPE*; *argument*: *TYPE*; ...): *RESULT_TYPE* **is**

-- Header comment (non-formal)

require

Precondition

do

Compound

ensure

Postconditions

end

expressing the precondition and the postcondition as part of the routine text, through the **require** and **ensure** clauses. Like other uses of assertions, these clauses are optional.

A call to the routine is correct if and only if it satisfies the *Precondition*; if the routine body is correct, the caller may then rely on the *postcondition* on routine return. This is the idea, already mentioned above (page 314), of **Design by Contract**. A routine call is a contract to perform a certain task. The caller is the "client", the called routine is the "supplier". As in every good contract, there are advantages and obligations for both parties:

- The precondition is an obligation for the client; for the supplier, it is a benefit, since, as expressed by $I1_{Routine}$, it relieves the routine body from having to care about cases not covered by the precondition.
- For the postcondition the situation is reversed.

An Eiffel routine, as given by the above form, is a function, returning a result. That result is a single element rather than a list. This is sufficient for the examples below; generalization to a list of results would be immediate. The examples below will use the Eiffel convention for computing the result of a function: any function has an implicitly declared variable called *Result*, of type *RESULT_TYPE*, to which values may be assigned in the body; its final value is the result returned to the caller.

The notation also supports procedures, which do not return a result. A routine is a procedure if its header does not include the part

: *RESULT_TYPE*

9.10.6 Recursion

Rule $I1_{Routine}$, it was said above, is not applicable to recursive routines. This is not because it is wrong in this case, but rather because it becomes useless.

The problem is that the rule only enables you to prove a formula of the form

$\vdash$ {*P*} **call** *s* (...) {*Q*}

if you can prove the corresponding property of the body b of s, with appropriate actual-formal argument substitutions. If s is recursive, however, its body will contain at least one call to s, so that proving properties of b will require proving properties of calls to s, which because of the inference rule will require proving properties of.... The proof process itself becomes infinitely recursive.

If, as with loops, we take a partial correctness approach, accepting the necessity to prove termination separately, we need not change much to $I1_{Routine}$ to make it work for recursive routines. The idea is that you should be allowed to use inductively, when trying to prove a property of b, the corresponding property of **calls** to s.

To understand this, look again at the application of the non-recursive rule. As noted, the goal is to prove

[9.39]
$$\{P'\} \quad \textbf{call } s\ (\ldots) \quad \{Q'\}$$

by proving

[9.40]
$$\{P\} \quad BODY \quad \{Q\}$$

where P' and Q' differ from P and Q by substitutions only. So the proof of [9.39] includes two steps: first prove [9.40], the corresponding property on the body; then, using $I1_{Routine}$, derive [9.39] by carrying out the appropriate substitutions.

If the same approach is applied to recursive routines, the first step in this process – the proof relative to the body – must be allowed to assume the property of the call, the very one which is the ultimate goal of the proof.

The more general rule for routine calls follows from this observation. (It is applicable to non-recursive routines as well, although in this case the former rule suffices). The restrictions of 9.10.4 apply as before.

$I2_{Routine}$

$$\frac{\{P\ [argument \leftarrow i\,]\}\ Call\ (called\colon r.name;\ input\colon i;\ output\colon o)\ \{Q\ [result \leftarrow o\,]\} \Longrightarrow \{P\ \}\ r.body\ \{Q\}}{\{P\ [argument \leftarrow i\,]\}\ Call\ (called\colon r.name;\ input\colon i;\ output\colon o)\ \{Q\ [result \leftarrow o\,]\}}$$

The premise of this rule is of the form $F \Longrightarrow G$, and its conclusion of the form H, where F, G, H are pre-post formulae. The rule means: "If you can prove that F implies G, then you may deduce H". The antecedent being an implication, its proof will often be a conditional proof. What is remarkable, of course, is that H is in fact the same as F.

With the Eiffel concrete notation for preconditions and postconditions, as introduced above, rule $I2_{Routine}$ indicates that the instructions leading to any recursive call in the body (**do** clause) must guarantee the precondition (**require** clause) before that call, and may be assumed to guarantee the postcondition (**ensure** clause) on return, with appropriate actual-formal substitutions in both cases. The rule also indicates that if you try to check the first property (precondition satisfied on call), you may recursively assume that property on routine entry.

You will have noted the use of the terms "may assume" and "must guarantee" in the preceding discussion. They reflect the client-supplier relationship as derived from the contract theory of software construction. Here the routine is its own client and supplier, and the alternating interpretations of the assertions' meaning reflect this dual role.

The article that first introduced the axiomatics of recursive routines [Hoare 1971] stressed the elegance of the recursive routine rule in particularly apt terms:

> *The solution of the infinite regress is simple and dramatic: to permit the use of the desired conclusion as a hypothesis in the proof of the body itself. Thus we are permitted to prove that the procedure body possesses a property, on the assumption that every recursive call possesses that property, and then to assert categorically that every call, recursive or otherwise, has that property. This assumption of what we want to prove before embarking on the proof explains well the aura of magic which attends a programmer's first introduction to recursive programming.*

9.10.7 Termination

In any practical call, the regress had better be finite if you hope to see the result in your lifetime. (This was apparent in the last chapter's denotational study of recursion: even though in non-trivial cases no finite number of iterations of a function chain f will yield its fixpoint f_∞, once you choose a given x you know there is a finite i such that $f\ (x)$ is $f_i\ (x)$.)

To prove termination, it suffices, as with loops, to exhibit a variant, which here is an integer expression of which you can prove that:

- Its value is non-negative for the first (outermost) call.
- If the variant's value is non-negative on entry to the routine's body, it will be at least one less, but still non-negative, for any recursive call.

9.10.8 Recursion invariants

Like loops, recursive routines have variants. Not unpredictably, they also share with loops the notion of invariant.

A *recursion invariant* is an assertion I such that the recursive routine rule, $I2_{Routine}$, will apply if you use I both as precondition (P in the rule as given above) and postcondition (Q).

Here the rule means that if you are able to prove, under the assumption that any call preserves I, that the body preserves I as well, then you may deduce that any call indeed preserves I. The proof and deduction must of course be made under the appropriate actual-formal argument substitutions.

As an example of use of a recursion invariant in a semi-formal proof, consider a procedure for printing the contents of a binary search tree:

```
print_sorted (t: BINARY_TREE) is
        -- Print node values in order
    require
        -- t is a binary search tree, in other words:
        given
            left_nodes  ≜ subtree (t.left);
            right_nodes ≜ subtree (t.right)
                -- where subtree (x) is the set of nodes
                -- in the subtree of root x
        then
            ∀ l : left_nodes, r : right_nodes • l.value ≤ t.value ≤ r.value
        end
    do
        if not t.Void then
            print_sorted (t.left);
            print (t.value);
            print_sorted (t.right)
        end
    ensure
        "All values in subtree (t) have been printed in order."
    end
```

This assumes primitives *left*, *right*, *empty* and *value* applicable to any tree node, and a predefined procedure *print* to print a value. The variant "height of subtree of root t" ensures termination.

Here the informal recursion invariant is "If any value in *subtree* (t) has been printed, then all values of *subtree* (t) have been printed in order". The invariant is trivially satisfied before the call since no node value has been printed yet. If t is a void tree, then *subtree* (t) is an empty set and the procedure preserves the invariant since it

prints nothing at all. If t is not void, then the procedure calls itself recursively on $t.left$, then prints the value attached to the root t, then calls itself recursively on $t.right$. Since t is a binary search tree (see the precondition), this preserves the invariant. Furthermore, we know that in this case at least one value, $t.value$, has been printed, so the invariant gives the desired postcondition – "All values in *subtree* (t) have been printed in order".

Transforming this semi-formal proof into a fully formal one requires developing a small axiomatic theory describing the target domain – binary trees. This is the subject of exercise 9.26; the following section, which builds such a mini-theory for another target domain, may serve as a guideline.

Since loops share the notion of invariant with recursive routines, it is natural to ask whether the two kinds of invariant are related at all, especially for recursive routines which have a simple loop equivalent. The most common examples are "linearly-recursive" routines, such as a recursive routine for computing a factorial, which may be written

```
factorial (n: INTEGER): INTEGER is
                -- Factorial of n
    require
        n ≥ 0
    do
        if n = 0 then
            Result := 1
        else
            Result := n * factorial (n − 1)
        end
    ensure
        Result = n!
    end
```

To simplify the proof, let us rely on the convention that *Result*, before explicitly receiving a value through assignment, has the default initialization value 0. If we prove that the property

Result = 0 **or** *Result* = n!

is invariant, and complement this by the trivial proof that *Result* cannot be 0 on exit, we obtain the desired postcondition *Result* = n!.

This recursive algorithm has a simple loop counterpart with an obvious invariant:

```
i: INTEGER;
from
        i := 0; Result := 1
variant
        n – i
invariant
        Result = i!
until i = n loop
        i := i + 1; Result := Result * i
end
```

As this example indicates, although there may be a relation between a recursion invariant and the corresponding loop invariants, the relation is not immediate.

The underlying reason was pointed out in the analysis of recursive methods in the previous chapter: although a recursive computation will be executed, as its loop counterpart, as a "bottom-up" computation, the recursive formulation of the algorithm describes it in top-down format (see 8.4.3). The loop and recursion invariants reflect these different views of the same computation.

9.10.9 Proving a recursive routine

To understand the recursive routine rule in detail, it is useful to write a complete proof. The object of this proof will be what is perhaps the archetypal recursive routine: the solution to the Tower of Hanoi puzzle [Lucas 1883].

In this well-known example, the aim is to transfer n disks initially stacked on a peg A to a peg B, using a third peg C as intermediate storage. The argument n is a non-negative integer. Only one operation is available, written

move (*x*, *y*)

Its effect is to transfer the disk on top of x to the top of y (x and y must each be one of A, B, C). The operation may be applied if and only if pegs x and y satisfy the following constraints:

- There is at least one disk on x; let d be the top disk.
- If there is at least one disk on y, then d was above the top disk on y in the original stack.

Another way to phrase the second constraint is to assume that the disks, all of different sizes, are originally stacked on A in order of decreasing size, and to require that *move* never transfers a disk on top of a smaller disk.

The proof will apply to the following procedure for solving this problem:

```
Hanoi (n: INTEGER; x, y, z: PEG) is
        -- Transfer n disks from peg x to peg y, using z as intermediate storage.
    do
        if n > 0 then
            Hanoi (n – 1, x, z, y);
            move (x, y);
            Hanoi (n – 1, z, y, x)
        -- else do nothing
        end
    end
```

Although based on a toy example, this is an interesting routine because it is "really" recursive: unlike simpler examples of recursive computations (such as the recursive definition of the factorial function) it does not admit a trivial non-recursive equivalent. In addition, its structure closely resembles that of many useful practical recursive algorithms such as Quicksort or binary tree traversal (see exercises 9.25 and 9.26).

The proof of termination is trivial: n is a recursion variant. What remains to prove is that if there are n disks on A and none on B or C, the call *Hanoi* (n, A, B, C) transfers the n disks on top of B, leaving no disks on A or C. The proof that disk order does not change will be sketched later.

It turns out to be easier to prove a more general property: if there are n or more disks on A, the call will transfer the top n among them on top of those of B if any, leaving C in its original state.

Much of the proof work will be preparatory: building the right model for the objects whose properties we are trying to prove. This is a general feature of proofs: often the task of **specifying** what needs to be proved is as hard as the proof proper, or harder.

Here we must find a formal way to specify piles of disks and their properties. As a simple model, consider "generalized stacks" whose elements may be pushed or popped by whole chunks, rather than just one by one as with ordinary stacks. The following operations are defined on any generalized stacks s, t, for any non-negative integer i:

```
|s|
        -- Size: an integer, the number of disks on s.
s^i
        -- Top: the generalized stack consisting of the i top elements of s,
        -- in the same order as on s. Empty if i = 0.
        -- Defined only if 0 ≤ i ≤ |s|.
```

$s - i$

-- Pop: the generalized stack consisting of the elements of s
-- except for the i top ones.
-- Defined only if $0 \le i \le |s|$.

$s + t$

-- Push: the generalized stack consisting of the elements
-- of t on top of those of s.

These operations satisfy a number of properties:

[9.41]

Definition (Axioms for generalized stacks):

G1	$\vdash$	$s + (t + u) = (s + t) + u$		
G2	$\vdash$	$s - 0 = s$		
G3.a	$\vdash$	$0 \le i \le \|t\|$	$\Longrightarrow$	$(s + t)^i = t^i$
G3.b	$\vdash$	$\|t\| < i \le \|s\| + \|t\|$	$\Longrightarrow$	$(s + t)^i = s^{i - \|t\|} + t$
G4	$\vdash$	$0 \le j \le i \le \|s\|$	$\Longrightarrow$	$(s^i)^j = s^j$
G5.a	$\vdash$	$0 \le i \le \|t\|$	$\Longrightarrow$	$(s + t) - i = s + (t - i)$
G5.b	$\vdash$	$\|t\| \le i \le \|s\| + \|t\|$	$\Longrightarrow$	$(s + t) - i = s - (i - \|t\|)$
G6	$\vdash$	$0 \le i \le \|s\|$	$\Longrightarrow$	$(s - i) + s^i = s$
G7	$\vdash$	$0 \le i + j \le \|s\|$	$\Longrightarrow$	$(s - i) - j = s - (i + j)$
G8	$\vdash$	$0 \le i \le \|s\|$	$\Longrightarrow$	$\|s^i\| = i$
G9	$\vdash$	$0 \le i \le \|s\|$	$\Longrightarrow$	$\|s - i\| = \|s\| - i$
G10	$\vdash$	$\|s + t\| = \|s\| + \|t\|$		

The rest of this discussion accepts these properties as the axioms of the theory of generalized stacks (also known as the specification of the corresponding "abstract data type"). Alternatively, you may wish to prove them using a *model* for generalized stacks, such as finite sequences; chapter 10 gives a more complete example of building a model for an axiomatic theory.

The axioms apply to any generalized stacks s, t, u and any integers i, j. Axioms G3.b, G5.b, G7 and G10 use "+" and "–" also as ordinary integer operators.

The associativity of "+" (property G1) will make it possible to write expressions such as $s + t + u$ without ambiguity.

The following axiom schema (for any assertion Q and any generalized stacks s and t) expresses the properties of the *move* operation:

[9.42]

$$\vdash \{|s| > 0 \textbf{ and } Q\ [s \leftarrow s - 1,\ t \leftarrow t + s^{1}\ ]\}\ \ move\ (s,\ t)\ \ \{Q\}$$

In words: the effect of *move* (*s*, *t*) is to replace s by $s - 1$ (s with its top element removed) and t by $t + s^{1}$ (t with the top element of s added on top). The first clause of the precondition expresses that s must contain at least one disk.

> To state this axiom is to interpret *move* as two assignments: the axiom rephrases $A_{Assignment}$ (page 323) applied to the generalized stack assignments
>
> $t := t + s^{1}$;
>
> $s := s - 1$
>
> where the assignments should really be carried out in parallel, although they will work in the order given (but not in the reverse order).

We must prove that the call *Hanoi* (*n*, *x*, *y*, *z*) transfers the top n elements of x onto the top of y, leaving z unchanged. Expressed as a pre-post theorem schema, for any assertion Q, any integer i and any generalized stacks s, t, u, the property to prove is

[9.43]

$$\{|s| \geq i \textbf{ and } Q\ [s \leftarrow s - i,\ t \leftarrow t + s^{i}\ ]\}\ \ Hanoi\ (i, s, t, u)\ \ \{Q\}$$

Let *BODY* be the body of routine *Hanoi* as given above. To establish [9.43], rule $I2_{Routine}$ tells us that it suffices to prove the same property applied to *BODY* with the appropriate argument substitutions:

[9.44]

$$\{|x| \geq n \textbf{ and } Q\ [x \leftarrow x - n,\ y \leftarrow y + x^{n}\ ]\}\ \ BODY\ \ \{Q\}$$

and that the proof is permitted to rely on [9.43] itself. This is the goal for the remainder of this section.

Thanks to $I_{Conditional}$ (page 329) we can dispense with the trivial case $n = 0$; for positive n, *BODY* reduces to the following (with assertions added as comments):

[9.45]

```
                        -- {Q1}
Hanoi (n – 1, x, z, y);
                        -- {Q2}
move (x, y) ;
                        -- {Q3}
Hanoi (n – 1, z, y, x)
                        -- {Q4}
```

We must prove that the above is a correct pre-post formula if Q_4 is Q and Q_1 is the precondition given in [9.44]. Since we are dealing with a compound and generalized assignments, the appropriate technique is to work from the end, starting with the postcondition Q as Q_4, and derive successive intermediate assertions Q_3, Q_2 and Q_1, such that Q_1 is the desired precondition.

To obtain Q_3, we apply [9.43] to the second recursive call; this requires substituting the actual arguments $n-1, z, y, x$ for i, s, t, u respectively. Then:

$$Q_3 \triangleq |z| \geq n-1 \textbf{ and } Q\ [z \leftarrow z-(n-1),\ y \leftarrow y+z^{n-1}]$$

Moving up one instruction, application of the *move* axiom [9.42] to Q_3, with actual arguments x and y substituted for s and t respectively, yields:

$$\begin{aligned} Q_2 &\triangleq |x| > 0 \textbf{ and } Q_3\ [x \leftarrow x-1,\ y \leftarrow y+x^1] \\ &= |x| > 0 \textbf{ and } |z| \geq n-1 \textbf{ and} \\ &\quad Q\ [x \leftarrow x-1,\ y \leftarrow y+x^1+z^{n-1},\ z \leftarrow z-(n-1)] \end{aligned}$$

The only delicate part in obtaining Q_2 is the substitution for y, derived by combining two successive substitutions; this uses the rule for composition of substitutions ([9.10], page 324), applied to identical a and b and generalized to simultaneous substitutions. (In this generalization, all substitutions apply to the tuple $<x, y, z>$, serving as both a and b.)

Finally, applying [9.43] again to the first recursive call with actual arguments $n-1$, x, z, y yields Q_1:

$$Q_1 \triangleq |x| \geq n-1 \textbf{ and } Q_2\ [x \leftarrow x-(n-1),\ z \leftarrow z+x^{n-1}]$$

so that, composing substitutions again:

[9.46]

$$\begin{aligned} Q_1 = |x| \geq n-1 &\textbf{ and } |x-(n-1)| > 0 \textbf{ and } |z+x^{n-1}| \geq n-1 \textbf{ and} \\ &Q\ [x \leftarrow x-(n-1)-1, \\ &\quad y \leftarrow y+(x-(n-1))^1+(z+x^{n-1})^{n-1} \\ &\quad z \leftarrow (z+x^{n-1})-(n-1)] \end{aligned}$$

There remains to simplify Q_1, using the various axioms for generalized stacks [9.41]. Consider the first part of Q_1 (the conditions on sizes). From axiom G9, the clause $|x-(n-1)| > 0$ is equivalent to $|x| \geq n$. From axioms G10 and G8,

$$\begin{aligned} |z+x^{n-1}| &= |z|+|x^{n-1}| \\ &= |z|+n-1 \\ &\geq n-1 \end{aligned}$$

so that the first line of the expression for Q_1 [9.46] is equivalent to just $|x| \geq n$

Now call x_{new}, y_{new} and z_{new} the replacements for x, y and z in the substitutions on Q on the next three lines. Then:

$$
\begin{aligned}
x_{new} &\triangleq (x - 1) - (n - 1) \\
&= x - n \\
&\qquad\qquad \text{-- From G7} \\
y_{new} &\triangleq y + (x - (n-1))^{1} + (z + x^{n-1})^{n-1} \\
&= y + (x - (n-1))^{1} + x^{n-1} \\
&\qquad\qquad \text{-- From G8 and G3.a} \\
&= y + (x - (n-1) + x^{n-1})^{n} \\
&\qquad\qquad \text{-- This comes from G3.b, used from right to left} \\
&\qquad\qquad \text{-- for } i \triangleq n,\ s \triangleq x - (n-1) \text{ and } t \triangleq x^{n-1}; \\
&\qquad\qquad \text{-- applicability of G3.b is deduced from G8, G9 and G10.} \\
&= y + x^{n} \\
&\qquad\qquad \text{-- From G6, with } s \triangleq x \text{ and } i \triangleq n-1 \\
z_{new} &\triangleq (z + x^{n-1}) - (n-1) \\
&= z \\
&\qquad\qquad \text{-- From G5.b, justified by G8, and G2}
\end{aligned}
$$

As a result of these simplifications, the overall precondition Q_1 obtained in [9.46] is in fact

$$Q_1 = |x| \geq n \ \textbf{and}\ Q\ [x \leftarrow x - n, y \leftarrow y + x^{n}]$$

which is the desired precondition [9.44]. □

The proof does not take into account disk ordering constraints, as represented by rules on disk sizes. Here is one way to refine the above discussion so as to remove this limitation. (The method will only be sketched; you are invited to fill in the details.) Add to the specification of generalized stacks an operation written s_i, so that s_i, an integer, is the size of the i-th disk in s from the top. Define a boolean-valued function on generalized stacks, written $s!$ and expressing that s is sorted, as:

$$s! \ \triangleq\ \forall i : 2 .. |s| \bullet s_{i-1} < s_i$$

To adapt the specification so that it will only describe sorted stacks, add to all axioms involving a subexpression of the form $s + t$ a guard (condition to the left of the $\Longrightarrow$ sign) of the form

$$|s| \geq 1 \ \wedge \ |t| \geq 1 \quad \Longrightarrow \quad t_{|t|} < s_1$$

and add to the precondition of *move* (*s, t*) a similar clause stating that if t is not empty its top disk is bigger than the top disk of s.

Then you need to prove that the property

$$x\,! \ \textbf{and} \ y\,! \ \textbf{and} \ z\,!$$

is a recursion invariant, by adding it to the postcondition Q and moving it up until it yields the precondition.

9.11 ASSERTION-GUIDED PROGRAM CONSTRUCTION

Among the uses of formal specifications listed in chapter 1, the most obvious applications of the axiomatic techniques developed in this chapter seem to be program verification and language standardization.

Perhaps less immediately apparent but equally important is the application of axiomatic techniques to the construction of reliable software. Here the goal is not to prove an existing program, but to integrate the proof with the program construction process so as to ensure the correctness of programs from the start. This may be called the **constructive approach** to software correctness.

There are several reasons why this approach deserves careful consideration:

- Unless you make the concern for correctness an integral part of program building, it is unlikely that you will be able to produce provably correct programs. Were you able to prove anything at all, the most likely outcome is a proof of *in*correctness.
- With the methods of this chapter, proofs require that the program be stuffed with assertions. The best time to write these assertions is program design time. Many of them will in fact come from the preceding phases of analysis.
- In many practical cases, you will not be able to carry out complete proofs of correctness, if only because of technical limitations such as the lack of a complete axiom system for a given programming language. But the techniques of this chapter can still go a long way toward ensuring correctness by helping you to write programs so as to pave the way for a hypothetical proof.

The rest of this chapter expands on these ideas by showing examples of how axiomatic techniques can help make the correctness concern an integral part of the software design.

A warning is in order: the techniques developed below are neither fail-safe nor universal. Formal proofs are the only way to guarantee correctness (and even they are meaningful only to the extent that you can trust the compiler, the operating system and the hardware). But an imperfect solution is better than the standard approach to program construction, where correctness concerns play a very minor role, if any role at all.

9.11.1 Assertions in programming languages

Because assertions are such a help in designing correct software, and such a good trace of the specification and design process that led to a particular software element, it seems a pity not to include them in the final software text.

Of course, you may always include assertions as comments. This is indeed highly recommended if you are using a programming language that offers no better deal. Having more formal support for assertions as part of the programming language proper offers a number of advantages:

1 • The path from specification to design and implementation becomes smoother: the first phase produces the assertions; the next ones yield instructions which satisfy the corresponding pre-post formulae.

2 • If the language includes a formal assertion sublanguage, software tools can extract the assertions from a software element automatically to produce high-level documentation about the element. This is a better approach than having programmers write software documentation as a separate effort.

3 • Even in the absence of a program proving mechanism, a compiler may have an option which will generate code for checking assertions at run-time (the next best thing to a proof). This turns out to be a remarkable debugging aid, since many bugs will manifest themselves as violations of the consistency conditions expressed by assertions.

4 • Assertions also have a direct connection with the important issue of exception handling, which, however, falls beyond the scope of the present discussion.

A number of programming languages have included some support for assertions. The first was probably Algol W, which has an instruction of the form

ASSERT (*b*)

where *b* is a boolean instruction. Depending on a compilation option, the instruction either evaluates *b* or is equivalent to a *Skip*. In the first case, program execution will terminate with an informative message if the value of *b* is false. The C language offers a similar mechanism.

Such constructs, however, are mostly debugging aids – application 3 above. They are insufficient to support the full role of assertions in the software construction process, especially applications 1 and 2.

Some languages take the notion of assertion more seriously. An example is the Anna design language (see bibliographical notes). Another is Eiffel.

Eiffel's mechanism, used for the examples of assertion-guided software construction in the rest of this discussion, directly supports all four applications above. Two aspects of

the mechanism have already been described:

- The syntactic inclusion of invariant, variant and initialization clauses in loops (page 344).
- The **require** and **ensure** clauses in routines, supporting the principle of "programming by contract", as discussed on page 358.

Eiffel assertions appear in two other important contexts:

- A class may (and often does) have a **class invariant**, which expresses global properties of the class's instances. Class invariants are theoretically equivalent to clauses added to both the precondition and postcondition of every exported routine of a class, but they are better factored out at the class level.
- A check instruction, of the form **check** *Assertion* **end**, may be used at any point where you want to assert that a certain property will hold, outside of the constructs just discussed.

In the programming examples which follow, the notation **check** *Assertion* **end** will replace the Metanot braces used earlier in this chapter, as in {*Assertion*}.

More generally, the examples will rely on the Eiffel notation, slightly adapted for the circumstance: first, some assertions will include quantified expressions (∀ … and ∃ …), currently not supported by the Eiffel assertion sublanguage, which is based on boolean expressions; second, the examples do not take advantage of some specific Eiffel structures and mechanisms (the class construct, deferred routines for specification without implementation, genericity, the treatment of arrays as abstractly defined data structures and others) which have not been described in this book.

It is often necessary, in a routine postcondition, to refer to the value an expression had on routine entry. The discussion will use the Eiffel **old** notation, of which an example is given by the following routine specification:[2]

```
enter (x: T; t: table of T): BOOLEAN is
        -- Insert x into t ; increment count.
    require
        not full -- There should be room in the table
    do
        …
    ensure
        …
        count = old count + 1
    end
```

[2] Here *count* has to be externally available to the routine. In Eiffel it would usually be an attribute of the class. Also, the argument *t* would normally be implicit, routine *enter* being part of a class describing tables.

9.11.2 Embedding strategies

Among the control structures studied in this chapter, the most interesting ones, requiring invention on the part of the programmer, are routines (especially recursive ones) and loops. This discussion will focus on loops.

The inference rules for loops express the postcondition – goal – of a loop as

$$G \triangleq I \textbf{ and } E$$

where I is the invariant and E is the exit condition.

This suggests that the invariant is a **weakened version of the goal**: weak enough that you can ensure its validity on loop initialization; but strong enough to yield the desired goal when combined with the exit condition.

Figure 9.6: Embedding

Loop construction strategies, then, may be viewed as various ways to weaken the goal.

The above figure illustrates the underlying view of loops. When looking for a solution to a programming problem, you are trying to find one or more objects satisfying the goal in a certain solution space – the curve G on the figure. G corresponds to the goal. The aim is to find an element x in G. If you do not see any obvious way to hit G directly, you may try a loop solution, which is an iterative strategy working as follows:

- Embed G into a larger space, I. I represents the set of states satisfying the invariant.
- Define E so that G is the intersection of I and E. E corresponds to the exit condition.
- Choose as starting value for x some point x_0 in I. Because I is a superset of G, it will be easier in general to find this element than it would be to find an element of G right away. Element x_0 corresponds to the loop's initialization.
- At each step let V be the "distance" from x to G. V corresponds to the loop's variant.
- Apply an iterative mechanism which, at each step, determines if x is in G, in which case the iteration terminates (you have reached a solution), and otherwise computes the next element by applying to x a transformation B which must keep x in I but will decrease V. B corresponds to the loop body.

The loop is of the form

from
    $x := x_0$
invariant
    I
variant
    V
until $x \in G$ **loop**
    B
end

check I **and** $x \in G$ **end** -- (i.e. G)

We may now view loop construction as the problem of finding the best way to embed goal spaces such as G into larger "invariant" spaces I, with the associated choices for the starting point x_0, the variant V, and the body B.

The following sections study two particular embedding strategies:

- Constant relaxation.
- Uncoupling.

9.11.3 Constant relaxation

With the constant relaxation strategy, you obtain the invariant I from the goal G by substituting a variable for a constant value. The initialization will assign some trivial value to the variable to ensure I; each loop iteration gets the variable's value closer to that of the constant, while maintaining the invariant.

The simple example of linear search in a non-sorted list provides a good illustration of the idea.

Assume you have an array of elements of any type T, and an element x of the same type, and you want to determine whether x is equal to any of the elements in t.

You can write the routine as a function *has* returning a boolean value, with the postcondition

[9.47]
$$Result = (\exists\ k : 1\,..\,n \bullet x = t\ [k])$$

In other words, the result is true if and only if the array contains an element equal to x.

The function may be specified as follows (assuming n is a non-negative constant):[3]

```
has (x: T; t: array [1..n] of T): BOOLEAN is
            -- Does x appear in t?
        require
            true -- No precondition
        do
            ...
        ensure
            Result = (∃ k : 1.. n • x = t [k])     -- [9.47]
        end
```

In assertion-guided program construction, we examine the specification (the postcondition) and look for a refinement which will yield a solution (the routine body). For a loop solution, the refinement is an embedding as defined above.

To find such an embedding, we may note that any difficulty in obtaining the goal G, as given by [9.47], is the presence of the interval $1..n$. The smaller the n, the easier; with a value such as 1 or better yet 0 the answer is trivial. For 0, it suffices to use *false* for *Result*.

This yields an embedding based on the constant relaxation method: introduce a fresh variable i which will take its values in the interval $0..n$ and rewrite the goal [9.47] as

[3] In normal Eiffel usage this function would appear in a class describing some variant of the array data structure; as a consequence, the argument *t* would be implicit.

Result = (∃ *k* : 1 .. *i* • *x* = *t* [*k*]) -- *I*
and *i* = *n*

which is trivially equivalent to the original. Call *I* the condition on the first line. *I* has all the qualifications of an invariant:

- *I* is easy to ensure initially (take **false** for *Result* and 0 for *i*).
- *I* is a weakened form of the goal, since it coincides with it for *i* = *n*.
- Maintaining *I* while bringing *i* a little closer to *n* will not be too difficult (see next).

This prompts us to look for a solution of the form

from
i := 0; *Result* := **false**;
invariant
I
variant
n – *i*
until
i = *n*
loop
"Get *i* closer to *n*, maintaining the validity of *I*"
end
check [9.47] **end**

The loop body ("Get *i* ...") is easy to obtain. It must be an instruction *LB* which makes the following pre-post formula correct:

check *I* **and** *i* < *n* **end**
LB
check *I* **and** 0 ≤ *n* – *i* < **old** (*n* – *i*) **end**

The **old** notation makes it possible to refer to the value of the variant, *n* – *i*, before the loop body (although **old** usually applies to routines).

The simplest way to "Get *i* closer to *n*" is to increase it by 1. This suggests looking for an instruction *LB'* such that the following is correct:

check
Result = (∃ *k* : 1 .. *i* • *x* = *t* [*k*])
and *i* < *n*
end
i := *i* + 1; *LB'*
check *Result* = (∃ *k* : 1 .. *i* • *x* = *t* [*k*]) **end**

The postcondition is very close to the precondition; more precisely, the precondition implies that after execution of the instruction *i := i + 1* the following holds:

$$Result = (\exists\ k : 1\,..\ i-1 \bullet x = t\ [k])$$

so that the specification for *LB'* is:

check $Result = (\exists\ k : 1\,..\ i-1 \bullet x = t\ [k])$ **end**
LB'
check $Result = (\exists\ k : 1\,..\ i \bullet x = t\ [k])$ **end**

An obvious solution is to take for *LB'* the instruction

Result := Result **or else** (*t* [*i*] = *x*)

which application of the assignment rule ($A_{Assignment}$, page 323) easily shows to satisfy the specification. The **or else** could be an **or**, but we do not need to perform the test on *t* [*i*] if *Result* is already true.

This gives a correct implementation of *has*:

```
has (x: T; t: array [1..n] of T): BOOLEAN is
                -- Does x appear in t?
        require
                true -- No precondition
        local
                i: INTEGER
        do
                from
                        i := 0; Result := false
                invariant
                        Result = (∃ k : 1.. i • x = t [k])
                variant
                        n – i
                until
                        i = n
                loop
                        i := i + 1;
                        Result := Result or else (t [i] = x)
                end
        ensure
                Result = (∃ k : 1.. n • x = t [k])
        end
end
```

You are invited to investigate for yourself how to carry out the obvious improvement – stopping the loop as soon as *Result* is found to be true – in the same systematic framework.

This example is typical of the constant relaxation method, applicable when the postcondition contains a constant such as n above and you can obtain an invariant by substituting a variable such as i. The variant is the difference between the constant and the variable; the loop body gets the variable closer to the constant and re-establishes the invariant.

"For" loops of common languages support this strategy.

9.11.4 Uncoupling

Another embedding strategy, related to constant relaxation but different, is "uncoupling". It applies when the postcondition is of the form

$$p\ (i)\ \textbf{and}\ q\ (i)$$

for some variable i. In other words, the postcondition introduces a "coupling" between two clauses p and q. You may then find it fruitful to introduce a fresh variable j, rewrite the postcondition as

$$p\ (i)\ \textbf{and}\ q\ (j) \qquad \text{-- } I$$
$$\textbf{and}\ i = j$$

and use the first line as candidate loop invariant I, the variant being $|j - i|$. Because you have "uncoupled" the variables in the two conditions p and q, it may be much easier to ensure the initial validity of I. The loop body is then of the form

"Bring i and j closer, maintaining I"

which will often be done in two steps:

"Bring i and j closer";

"Re-establish I if needed"

As an example of this strategy, consider a variation on the preceding searching problem, with the extra hypothesis that T has an order relation, written $\leq$, and the array t is sorted. This assumption may be expressed as a precondition:[4]

[9.48]

$$\forall k : 2\,..\,n \bullet t\ [k-1] \leq t\ [k]$$

[4] In Eiffel's object-oriented software decomposition, such a property would normally be expressed not as the precondition of an individual routine, but as a **class invariant** for the enclosing class.

The previous version of *has* would still work, of course, but we may want to rewrite it to take advantage of t being sorted. One possibility is to write the body of the new *has* under the form

[9.49]

position := *index* (*t*, *x*);

Result := *position* $\in 1 .. n$ **and then** $x = t$ [*position*]

where the auxiliary function *index* returns a position such that x either appears at that position or does not appear in the array at all.

The precise specification of *index*'s postcondition turns out to be perhaps the most delicate part of this problem (which you are invited to try out by yourself first):

A • The specification must be satisfiable in all cases: whatever the value of x is relative to the array values, there must be at least one *Result* satisfying the postcondition.

B • To make the above algorithm [9.49] a correct implementation of *has*, the *Result* must be the index of an array position where x appears, or otherwise must enable us to determine that x does not occur at all.

The following postcondition satisfies these requirements:

[9.50]

Result $\in 0 .. n$

and $(\forall k : 1 .. \mathit{Result} \bullet t\ [k] \leq x)$ -- p (*Result*)

and $(\forall k : \mathit{Result}+1 .. n \bullet t\ [k] \geq x)$ -- q (*Result*)

To check for condition A above, note that 0 will do for *Result* if x is smaller than all array values, and n if it is larger than all array values. (Remember once again that $\forall x : E \bullet P$ is always true if E is empty.) For condition B, [9.50] implies that x appears in t if and only if

$i > 0$ **and then** t [*Result*] $= x$

Specification [9.50] is non-deterministic: if two or more (necessarily consecutive) array entries have value x, any of the corresponding indices will be an acceptable *Result*. There are several ways to change the postcondition so that it defines just one *Result* in all cases; we may for example change the last clause to read

$(\forall k : \mathit{Result}+1 .. n \bullet t\ [k] > x)$

so that, in case of multiple equal values, *Result* will be the highest adequate index. It is preferable, however, to keep the more symmetric version [9.50].

The problem, then, is to write the body for

index (*x: T*; *t:* **array** [*1..n*] **of** *T*): *INTEGER* **is**

-- Does *x* appear in *t*?

require

[9.48] -- *t* is sorted

do

…

ensure

[9.50]

end

How do we ensure the postcondition [9.50]? For more clarity let us use variable *i* instead of *Result*; the **do** clause may then end with *Result* := *i*. The postcondition is of the form

$i \in 0..n$ **and** $p\ (i)$ **and** $q\ (i)$

which the uncoupling strategy suggests rewriting as

$i, j \in 0..n$ **and** $p\ (i)$ **and** $q\ (j)$ -- *I*

and $i = j$

leading to a loop solution of the form

from

$i := i_0;\ j := j_0$

invariant

$p\ (i)$ **and** $q\ (j)$

variant

distance (*i, j*)

until

$i = j$

loop

"Bring *i* and *j* closer"

end

This solution will be correct if and only if i_0 satisfies p, j_0 satisfies q, the refinement of "Bring *i* and *j* closer" conserves the invariant $p(i)$ **and** $q(j)$, and *distance* (*i, j*) is an integer variant.

The initialization is trivial: we choose i_0 to be 0 and j_0 to be n; $p\ (0)$ and $q\ (n)$ are true since they are $\forall$ properties on empty sets.

With these initializations it appears reasonable to maintain *i* no greater than *j* throughout the loop. This suggests a reinforced invariant:

$p\ (i)$ **and** $q\ (j)$ **and** $0 \le i \le j \le n$

The most obvious way to "Bring i and j closer" is to increment i by 1, or alternatively decrement j by 1, and see what it takes to keep the invariant true. Since the problem is symmetric in i and j, we should treat both possibilities equally.

Assuming the invariant is satisfied and $i < j$ (the exit condition is not met yet), under what conditions may we increment i or decrement j?

Clearly, the instruction

$$i := i + 1$$

will preserve the invariant if and only if $p\ (i + 1)$ is true, and

$$j := j - 1$$

if and only if $q\ (j - 1)$ is true.

Look first at the i part. By definition,

$$p\ (i) = (\forall\, k : 1 .. i \bullet t\ [k] \le x)$$

so that if $t\ [i + 1]$ is defined (in other words, for $i < n$):

$$p\ (i + 1) = p\ (i) \quad \textbf{and} \quad t\ [i + 1] \le x$$

Starting from a state where $p\ (i)$ is satisfied, then, incrementing i by 1 will preserve the invariant if and only if

$$i < n \ \textbf{and then}\ t\ [i + 1] \le x$$

With respect to the j part

$$q\ (j) = (\forall\, k : j+1 .. n \bullet t\ [k] \ge x)$$

we may decrease j by 1 if and only if

$$j > 0 \ \textbf{and then}\ t\ [j] \ge x$$

In spite of appearances, the symmetry between the conditions on i and j is perfect; simply, because in the original postcondition [9.50] p involves *Result* and q involves *Result* + 1, it is in fact a symmetry between i and $j + 1$.

The guards $i < n$ and $j > 0$ are in fact superfluous: the invariant includes $0 \le i \le j \le n$, and $i > j$ will hold as long as the exit condition is not satisfied; so whenever the loop body is executed $i+1$ and j belong to the interval $1 .. n$. So we may try as loop body:

if
$\quad t\ [i + 1] \le x\ :\ i := i + 1$ □
$\quad t\ [j] \ge x\ :\ j := j - 1$
end

Because symmetry is so strong in this problem, the solution uses the guarded conditional (see [9.33], page 345). We must be careful, however: a guarded conditional will only

execute properly if, in all possible cases, at least one of the guards is true. Fortunately, here this is the case: because the array is sorted and $i < j$ is a precondition for the loop body, if the first guard is false, that is to say $t\ [i + 1] > x$, then the second guard, $t\ [j] \geq x$, is true.

The guarded conditional yields a non-deterministic instruction: if $t\ [i + 1] \leq x \leq t\ [j]$, then the instruction may execute either of its two branches. Using the standard conditional instruction removes the non-determinism:

[9.51]

```
if t [i + 1] ≤ x then
        i := i + 1
else
        j := j – 1
end
```

Either form yields a simple and correct version of *index:*

```
index (x: T; t: array [1..n] of T): INTEGER is
                -- Does x appear in t?
        require
                [9.48] -- t is sorted
        local
                i, j: INTEGER
        do
                from
                        i := 0; j := n
                invariant
                        p (i) and q (j) and 0 ≤ i ≤ j ≤ n
                variant
                        j – i
                until
                        i = j
                loop
                                -- This could use the if … then … else form instead
                        if
                                t [i + 1] ≤ x :  i := i + 1 □
                                t [j] ≥ x :  j := j – 1
                        end
                end
        ensure
                [9.50] -- (cf. page 378)
        end
```

As you will have noted, this is not the way most people usually write sequential search. The standard form will follow from an efficiency improvement that we should carry out as systematically as the above development. The price to pay for this improvement is the removal of the esthetically pleasant symmetry. Whenever the first guard is false, in other words $t\ [i + 1] > x$, then assigning to j the value of i (rather than just $j - 1$) will still preserve the invariant. This suggests rewriting the conditional as

```
if t [i + 1] ≤ x then
        i := i + 1
else
        j := i
end
```

(Of course, the symmetric change would also work.) As a result we may dispense with variable j altogether by noting that loop termination occurs when either $i = n$ or $t\ [i + 1] > x$, yielding the more usual form for sequential search:

```
from
        i:= 0
invariant … variant …
until i = n or else t[i+1] > x loop
        i := i+1
end
```

You should complete the **invariant** and **variant** clauses of this loop.

9.11.5 Binary search

Removing the symmetry between i and $j-1$ at best yielded a marginal efficiency improvement. A more promising avenue for improving the performance of sorted table searching is based on the property that t is an array, meaning constant-time access to any element whose index is known. This suggests "Bringing i and j closer" faster than by increments of +1 or –1.

The idea of **binary search** is to aim for the middle of the interval $i..j$. As Knuth noted in the volume on searching of his *Art of Computer Programming* [Knuth 1973]:

> *Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried.*

If you doubt this, it should suffice to take a look at exercise 9.12, which shows four innocent-looking versions – all wrong. For each version there is a case in which the algorithm fails to terminate, exceeds the array bounds, or yields a wrong answer. Before you read further, it is a good idea to try to come up with a correct version of binary search by yourself.

Both binary search and the above version of function *index* belong to a more general class of solutions based on the same uncoupling of the postcondition, where the loop body finds an element m in $i..j$ and assigns the value of m (or a neighboring value) to i or j.

In the version seen above m is $i + 1$ or $j - 1$; for binary search it will be approximately $(i + j)$ **div** 2, so that we can expect a maximum number of iterations roughly equal to $\log_2 n$ rather than n. (The operator **div** denotes integer division.)

This must be done carefully, however. In the general case we aim for a loop body of the form

m := *"Some value in* $1..n$ *such that* $i < m \leq j$*"*;
if
    $t\,[m] \leq x$: $i := m$ □
    $t\,[m] \geq x$: $j := m - 1$
end

which, in ordinary programming languages, will be written deterministically:

m := *"Some value in* $1..n$ *such that* $i < m \leq j$*"*;
if $t\,[m] \leq x$ **then**
    $i := m$
else
    $j := m - 1$
end

Whether the conditional instruction is deterministic or not, it is essential to get all the details right (and easy to get some wrong):

1 • The instruction must always decrease the variant $j - i$, by increasing i or decreasing j. If the the definition of m specified just $i \leq m$ rather than $i < m$, the first branch would not meet this goal.

2 • This does not transpose directly to j: requiring $i < m < j$ would lead to an impossibility when $j - i$ is equal to 1. So we accept $m \leq j$ but then we must take $m - 1$, not m, as the new value of j in the second branch.

3 • The conditional's guards are tests on $t\,[m]$, so m must always be in the interval $1..n$. This follows from the clause $0 \leq i \leq j \leq n$ which is part of the invariant.

4 • If this clause is satisfied, then $m \leq n$ and $m - 1 \geq 0$, so the conditional instruction indeed leaves this clause invariant.

5 • You are invited to check that both branches of the conditional also preserve the rest of the invariant, $p\ (i)$ **and** $q\ (j)$.

Any policy for choosing m is acceptable if it conforms to the above scheme. Two simple choices are $i + 1$ and j; they lead to variants of the above sequential search algorithm.

For binary search, m will be roughly equal to the average of i and j,

$$midpoint \triangleq (i + j) \textbf{ div } 2$$

The value of *midpoint* itself is not acceptable for m, however, because it might not satisfy requirement 1 above. Choosing *midpoint* + 1 will, however, satisfy all the above requirements.

This yields the following new version of *index*, using binary search:

```
index (x: T; t: array [1..n] of T): INTEGER is
                    -- Does x appear in t?
        require
            [9.48] -- t is sorted
        local
            i, j, m: INTEGER
        do
            from
                i := 0; j := n
            invariant
                p (i) and q (j) and 0 ≤ i ≤ j ≤ n
            variant
                j – i
            until
                i = j
            loop
                m := (i + j) div 2 + 1;
                if t [m] ≤ x then
                    i := m
                else
                    j := m – 1
                end
            end
        ensure
            [9.50] -- cf. page 378
end
```

We can check that the loop will be executed at most $\lfloor \log_2 n \rfloor$ times by proving that $\lfloor \log_2 (j - i) \rfloor$ is a variant. ($\lfloor x \rfloor$ is the largest integer no greater than x.) For any real numbers a and b, $\lfloor \log_2 (a) \rfloor < \lfloor \log_2 (b) \rfloor$ if $a \leq b / 2$. Here, $j - i$ is indeed at least divided by 2 in both possible cases in the loop, since whenever $i < j$ (i and j being integers):

$$j - ((i + j) \textbf{ div } 2 + 1) \leq (j - i) \textbf{ div } 2$$

$$i - ((i + j) \textbf{ div } 2) \leq (j - i) \textbf{ div } 2$$

(To check this, consider separately the cases $i + j$ odd and $i + j$ even).

Of course, the version of binary search obtained here is not the only possible one; you may wish to obtain others through variants of the uncoupling strategy.

9.11.6 An assessment

Although they apply to well-known and relatively simple algorithms, the above examples provide a good illustration of the constructive approach:

- The same framework served to derive two classes of algorithms (sequential and binary search). In the sorted array case, it is only at the last step (choosing how to "bring i and j closer") that different design choices lead to different computing methods.
- The heuristics used, constant relaxation and uncoupling, are quite general. One of the exercises (9.24) asks you to apply uncoupling to a completely different problem, sequence or array partitioning.
- We have built all versions so as to convince ourselves that they are correct and to know why they are.

Given the human capacity for error and self-deception, it would be absurd to characterize the methods illustrated here as sure recipes to obtain correct programs, or to claim that they make other correctness techniques (such as testing) obsolete. Perfect or universal they are not; more modestly, they constitute an important tool, among others, in the battle for software reliability. This suffices to make them one of the most valuable application of axiomatic semantics.

9.12 BIBLIOGRAPHICAL NOTES

The basis of the axiomatic method is mathematical logic, based on classical rhetoric but made considerably more rigorous in this century as an attempt to solve the crisis of mathematics that followed the development of set theory at the turn of the century. There are many good introductions to mathematical logic, such as [Kleene 1967] or [Mendelson 1964]. [Copi 1973] presents the notions of truth, validity, proof, axiom, inference etc. in a particularly clear fashion. [Manna 1985] is especially geared towards computer scientists.

Work presenting mathematical foundations for axiomatic theories is usually rooted in denotational semantics; this is the area of "complementary semantics", discussed in the next chapter. See the bibliographical notes to that chapter.

The first article on program proving using techniques based on assertions was [Floyd 1967], with a suggestive title: "Assigning Meanings to Programs". The paper also introduced the notion of loop invariant, called "inductive assertion".

Floyd's techniques were refined and improved in [Hoare 1969], which expressed them as a system of axioms and inference rules associated with programming language constructs. The approach was then applied to further language constructs such as routines

[Hoare 1971] and jumps [Clint 1972], and to the specification of a large part of the Pascal language [Hoare 1973b]. A comprehensive survey of Hoare semantics is given in [Apt 1981].

The weakest precondition approach was developed by Dijkstra in an article [Dijkstra 1975] and a book [Dijkstra 1976].

These publications by Dijkstra also pioneered the "constructive approach" to software correctness (9.11). [Gries 1981] is a very readable presentation of this approach. [Alagić 1978] and [Dromey 1983] apply similar ideas to teaching program design, algorithms and data structures. [Jones 1986] also emphasizes the use of program proving techniques for software development. See also work by the author [Meyer 1978, 1980] and in collaboration [Bossavit 1981], the latter describing the systematic construction of vector algorithms for supercomputers. For an account of how the spirit of the axiomatic method may be applied to the construction and proofs of algorithms in the very difficult area of concurrent programming, see the article on "on-the-fly garbage collection" [Dijkstra 1978].

The assertion mechanism of Anna is described in [Luckham 1985]. The assertion mechanism of Eiffel and its application to the construction of reliable software components are described in [Meyer 1988]. A further discussion of these topics, and the theory of "Design by Contract", may be found in [Meyer 1991b].

As mentioned on page 348, Dijkstra's non-deterministic choice and loop instructions have direct applications to concurrent programming. Hoare's CSP (Communicating Sequential Processes) approach to parallelism is based in part on these ideas [Hoare 1978, 1985].

The axiomatic theory of expression typing in lambda calculus (9.3) comes from [Cardelli 1984a], where it is applied to the more general problem of inferring proper types in a language (Milner's ML) where types, instead of being declared explicitly by programmers, are determined by the system from the context and the types attached to predefined identifiers. Cardelli's system also handles genericity: in other words, some of the types may include "free type identifiers" standing for arbitrary types. For example, the type of the identity function *Id* will be $\alpha \rightarrow \alpha$, where α stands for an arbitrary type.

EXERCISES

9.1 Integers in mathematics and on computers

Write an axiomatic theory of integers, starting from the standard Peano axioms (see [Suppes 1972], page 121, or any other text on axiomatic set theory). Then adapt the theory to account for the size limits imposed by number representation on computers.

9.2 Symmetric if instruction

Modify the abstract syntax and the denotational semantics of Graal, as given in chapter 5, to replace the classical **if**...**then**...**else**… conditional instruction by the guarded conditional (9.9.4).

9.3 Assignment and sequencing

Prove the following pre-post formula:

$\vdash$ $\{x = a$ **and** $y = b\}$

$t := x;\ x := x + y;\ y := t$

$\{x = a + b$ **and** $y = a\}$

9.4 Non-deterministic conditional

Compute A **wp** Q, for any assertion Q, where A is the instruction

if

$x \leq 0$: $a := -x$ □

$x \geq 0$: $a := x$

end

9.5 Weakest precondition

Show that the value of *guarded_if* **wp** Q (page 346) may also be expressed as:

guarded_if **wp** R =

$(c_1$ **or** c_2 **or** ... **or** $c_n)$ **and**

$(c_1$ **implies** $(A_1$ **wp** $R))$ **and**
$(c_2$ **implies** $(A_2$ **wp** $R))$ **and**
…
$(c_n$ **implies** $(A_n$ **wp** $R))$

9.6 Simple proofs

Prove that the following pre-post formulae, involving integer variables only and assuming perfect integer arithmetic, are theorems.

1 $\{z * x^y = K\}\ z := z * x\ \{z * x^{y-1} = K\}$

2 $\{z * x^y = K\}\ y := y - 1; z := z * x\ \{z * x^y = K\}$

3 $\{y\ even$ **and** $z * x^y = K\}\ y := y / 2; x := x^2\ \{z * x^y = K\}$

4 $\{z * x^y = K\}$

if

$y\ odd$: $y := y - 1; z := z * x$ □

$y\ even$: $y := y / 2; x := x^2$

end

$\{z * x^y = K\}$

9.7 Proving a loop

Let m and n be integers such that $m > 0$, $n \geq 0$. From the answers to exercise 9.6, determine the result of the following program; prove your answer.

x, y, z: INTEGER;

from

$x := m; y := n; z := 1$

until $y = 0$ **loop**

if

$y\ odd$: $y := y - 1; z := z * x$ □

$y\ even$: $y := y / 2; x := x^2$

end

end

9.8 Permutability of instructions

Applications such as parallel programming and the adaptation of programs to run on parallel or vector processors ("parallelization" or "vectorization"), often require to determine whether the order of two instructions may be reversed. This exercise investigates such permutability criteria.

Define two instructions A and B to be **equivalent**, and write

$A \equiv B$

if and only if for any assertion Q

$$A \textbf{ wp } Q = B \textbf{ wp } Q$$

Define that the instructions *permute*, written A **perm** B, if and only if

$$A; B \equiv B; A$$

1 – Consider assignment instructions A and B:

- A is $x := e$
- B is $y := f$

where e, f are expressions (none of the expressions considered in this exercise may contain function calls). Let V_e and V_f be the sets of variables occurring in e and f respectively. Give a sufficient condition on V_e and V_f for

$$A \textbf{ perm } B$$

to hold. Prove the result using the rules for assignment and sequence.

2 – Assume A is of the form

$$x := x \oplus e$$

and B is of the form

$$x := x \oplus f$$

where e and f are expressions, none of which contains x, and $\oplus$ is an operation which is both commutative and associative. Prove that it is true in this case that

$$A \textbf{ perm } B$$

9.9 Another assignment rule

Can you imagine a "forward rule" for assignment (see page 325) ? (**Hint**: Introduce explicitly the value that the variable being assigned to had before the assignment. The rule uses an existential quantifier.)

9.10 Array assignment

Prove theorems [9.14] (page 328). **Hint**: Remember to treat the assignment as an operation on the whole array.

9.11 Simple loop construction from invariants

Write loops to compute the following values for any n by finding first the appropriate invariants, using assertion-guided techniques (9.11).

1 $f = n!$ (factorial of n)

2 F_n, the n-th Fibonacci number, defined by

$$F_0 = 0$$
$$F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2} \quad \text{for } i > 1$$

9.12 Binary search: failed attempts

The figure on the adjacent page shows four attempts at writing a program for binary search. Each program should set *Result* to *true* if and only if the value x appears in the real array t, assumed to be sorted in increasing order.

The programs use **div** for integer division. Variable *found*, where used, is of type *BOOLEAN*.

Show that all of these programs are erroneous; it suffices for this to show that for each purported solution there exist values of the array t and the element x that produce an incorrect solution (*Result* being set to **true** although x does not appear in t or conversely) or will result in abnormal behavior at execution (out-of-bounds memory reference or infinite loop).

9.13 Indexed loops

Most languages provide a "do" or "for" loop structure in which the iteration is controlled by an index ranging over a finite range (usually an arithmetic progression over the integers, although it could be any finite set, such as a finite, sequential data structure like a linear list). In the simplest case, looping over a contiguous integer interval, the loop will be written as something like

for i: $a..b$ **loop** A_i **end**

where A_i is some instruction, usually dependent on the value of i.

Give a proof rule for such an instruction.

Figure 9.7: Four (wrong) programs for binary search

(**P1**)

```
from
        i := 1; j := n
until i = j loop
        m := (i + j) div 2;
        if t [m] ≤ x then
                i := m
        else
                j := m
        end
end;
Result := (x = t [i])
```

(**P2**)

```
from
        i := 1; j := n; found := false
until i = j and not found loop
        m := (i + j) div 2;
        if t [m] < x then
                i := m + 1
        elseif t [m] = x then
                found := true
        else
                j := m – 1
        end
end;
Result := found
```

(**P3**)

```
from
        i := 0; j := n
until i = j loop
        m := (i + j + 1) div 2;
        if t [m] ≤ x then
                i := m + 1
        else
                j := m
        end
end;
if i ≥ 1 and i ≤ n then
        Result := (x = t [i])
else
        Result := false
end
```

(**P4**)

```
from
        i := 0; j := n + 1
until i = j loop
        m := (i + j) div 2;
        if t [m] ≤ x then
                i := m + 1
        else
                j := m
        end
end;
if i ≥ 1 and i ≤ n then
        Result := (x = t [i])
else
        Result := false
end
```

9.14 Repeat... until

Give a proof rule for the Pascal

repeat … **until** …

instruction. You may use the observation that such a loop is readily expressed in terms of the **while** loop.

9.15 Equivalences between loops

Consider two loops of the following forms:

1. **while** c **loop**
   while c **and** c_1 **loop** A_1 **end**;
   while c **and** c_2 **loop** A_2 **end**
   end

2. **while** c **loop**
   if
   $c_1 : A_1 \ \square$
   $c_2 : A_2$
   end
   end

Prove that any invariant of loop 2 is also an invariant of loop 1.

Can you give an intuitive reason why an invariant of loop 1 might not be invariant for loop 2?

9.16 Precise requirements on variants

Consider a loop with continuation condition c and variant V (in the sense that the antecedents of rule IT_{Loop}, page 336, are satisfied). Show that

$\vdash \quad V = 0$ **implies** **not** c

(**Hint**: Proof by contradiction).

9.17 Weakest preconditions for loops

Define H_i, the necessary and sufficient condition for loop l to yield postcondition Q after at most i iterations ($i \geq 0$). You should first find independently an inductive definition of H_i in the manner of the definition of G_i ([9.31], page 342), and then prove its consistency with the definition of G_i.

9.18 Keeping track of the clock

(Due to Paul Eggert.) Consider Graal extended with two notions: clock counter and non-deterministic choice from integer intervals. This means two new instructions, with possible concrete syntax

- **clock** *t*
- **choose** *t* **by** *e*

In both instructions t is an integer variable; in the second, e is an integer expression.

The **clock** instruction assigns to t the current value of the machine clock. The machine clock is positive, never has the same value twice, and is always increasing.

The **choose** instruction assigns to t an integer value in the interval $0 .. |e-1|$. The implementation is free to use any value in that interval.

1 – Write axiomatic semantic definitions for these two instructions.

2 – Use your semantics to prove that the following loop always terminates:

```
from
        clock i
until i = 0 loop
        clock j;
        if i < j then
                choose i by i
        end
end
```

9.19 Restrictions on routines

Express the restrictions on routines for rule $\text{I1}_{Routine}$ (9.10.4) as static semantic constraints by defining a $V_{Routine}$ validity function (see 6.2).

9.20 Composition of substitutions

Prove the rule for composing substitutions ([9.10], page 324), using the definition of function *subst*. **Hint**: use structural induction on the structure of Q.

9.21 Simultaneous substitution

Define formally a function

simultaneous (*Q*: *Expression* ; *el*: *Expression** ; *il*: *Identifier**)

which specifies multiple simultaneous substitution (page 354) in a manner similar to function *subst* for single substitution ([9.9], page 323). **Hint**: it is not appropriate to use a list **over** … **apply** … expression on *el* or *il*; why?

9.22 In-out arguments

Extend rule $I1_{Routine}$ (page 353) to deal explicitly with in-out arguments.

9.23 Loops as recursive procedures

A loop of the form

while *c* **loop** *a* **end**

may also be written as

call *s*

where *s* is a procedure with the following body:

if *c* **then** *a* ; **call** *s* **end**

Using this definition, prove the loop rule (9.7.6) from the recursive routine rule (9.10.6).

9.24 Partitioning a sequence

Various algorithms require partitioning a sequence. This operation is used in particular for sorting arrays (next exercise) and for producing "order statistics".

Partitioning is applicable if a total order relation exists on sequence elements. Partitioning s means rearranging the order of its elements to put it in the form $t +\!\!+ <p> +\!\!+ u$, where any element of t is less than or equal to p, and any element of u is greater than or equal

to p. Value p, the **pivot**, is a sequence element, chosen arbitrarily; for the purpose of this exercise the pivot will be the element that initially appeared at the leftmost position. The only two permitted operations are *swap* (i, j), which exchanges the elements at positions i and j, and the test $s(i) \le s(j)$, which compares the values of the elements at positions i and j.

A general method for partitioning is to "burn the candle from both ends", the candle being the sequence deprived of its first element (the pivot). Maintain two integer cursors, "left" and "right", initialized to the leftmost and rightmost positions. At each step, increase the left cursor until it is under an element greater than the pivot, and decrease the right cursor until it is under an element lesser than the pivot. The two elements found are out of order, so swap them; then start the next step. The process ends when the two cursors meet; then you can swap the first sequence element (the pivot) with the element at cursor position.

Starting from this informal description, derive a correct algorithm for sequence partitioning, using the constructive methods described in 9.11.

Hint: The "candle-burning" process follows from the strategy of uncoupling, as discussed in 9.11.4 and 9.11.5.

9.25 Quicksort

A well-known algorithm for sorting an array is Quicksort, for which a routine may be written (in a simplified form applicable to sequences) as:

```
sorted (s: X*): X* is
        -- Produce a sorted permutation of s.
        -- (There must be a total order relation on X.)
    local
        t, u: X*
    do
        if s.LENGTH ≤ 1 then
            Result := s
        else
            <t, u> := partitioned (s);
            t := sorted (t);
            u := sorted (u);
            Result := t ++ u
        end
    ensure
                    -- Result is sorted, in other words:
        ∀ i : 2 .. Result .LENGTH • Result (i − 1) ≤ Result (i)
    end
```

Here *partitioned* is a routine, derived from the previous exercise, which given a sequence s of length 2 or more returns two non-empty sequences t and u such that $t +\!\!+ u$ is a permutation of s, and all elements of t are less than or equal to all elements of u.

After putting this routine in a form suitable for application of the recursive routine rule ($I2_{Routine}$, page 359), prove its correctness.

Hint: You may follow the example of the proof for routine *Hanoi* (9.10.9).

9.26 Inorder traversal of binary search trees

Prove rigorously the routine for printing the contents of a binary search tree in order (page 361). You will need to adapt the routine so that it produces a list of values as output.