Unified Theories of Programming

C.A.R. Hoare
Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, UK

Abstract. Professional practice in a mature engineering discipline is based on relevant scientific theories, usually expressed in the language of mathematics. A mathematical theory of programming aims to provide a similar basis for specification, design and implementation of computer programs. The theory can be presented in a variety of styles, including

1. Denotational, relating a program to a specification of its observable properties and behaviour.
2. Algebraic, providing equations and inequations for comparison, transformation and optimisation of designs and programs.
3. Operational, describing individual steps of a possible mechanical implementation.

This paper presents simple theories of sequential non-deterministic programming in each of these three styles; by deriving each presentation from its predecessor in a cyclic fashion, mutual consistency is assured.

Keywords. Programming, programming methods, non-determinism, semantics, operational, algebraic, denotational

Chapter 1

Introduction

A scientific theory is formalised as a mathematical description of some selected class of processes in the physical world. Observable properties and behaviour of such a process can then be predicted from the theory by mathematical deduction or calculation. An engineer applies the theory in the reverse direction. A specification describes the observable properties and behaviour of some system that does not yet exist in the physical world; and the goal is to design and implement a product which can be predicted by the theory to meet the specification.

This paper proposes a mathematical treatment of computer programming in the simple non-deterministic programming language introduced by Dijkstra [**?**]. The theory is well suited for use by engineers, since it supports both stepwise development of designs from specifications and hierarchical decomposition of complex systems into simpler components which can be designed and implemented separately. Furthermore, it permits derivation of a complete set of algebraic laws to help in transformation of designs and optimisation of programs. Finally, an operational semantics is derived; this treats practical aspects of implementation and efficiency of execution.

The main goal of this monograph is to show simple ways in which the three presentations of the same language can be derived from each other by mathematical definition, calculation and proof. The denotational theory consists just of a number of separate mathematical definitions of the notations of the language as predicates describing observable aspects of program execution. These definitions can be individually formulated and understood in isolation from each other.

The individual algebraic laws can then be derived from the denotational definitions, without danger of inconsistency or other unexpected interactions. A normal form theorem gives insight into the degree of completeness of the laws, and permits additional laws to be proved without induction. An argument is given for the computability of programs expressed in the language.

An operational theory is equally easily derived from the algebraic. First an algebraic definition is given for the basic step (transition relation) of an abstract implementation; and then the individual transition rules can be proved separately and individually as algebraic theorems, again with reduced risk of complex or unexpected interactions. A concept of bisimulation then lifts the level of abstraction, to permit derivation of a minimal collection of algebraic laws. An alternative abstraction technique derives the observational presentation from the operational, by proving all its definitions as theorems.

This completes the circle: it means that any of the presentations can be accepted as a primary definition of the meaning of the language, and the others can be derived from it. Even better, different definitions can be safely

and consistently used at different times and for different purposes. It is a characteristic of the most successful theories, in mathematics as well as in natural science, that they can be presented in several apparently independent ways, which are in a useful sense provably equivalent. For example, topology can be based on a certain kind of family of sets, or on neighbourhoods, or on a closure operator; and a theory of gravity can be based on Newtonian forces acting at a distance, or on field theory, or on Einsteinian geodesics. It is to be hoped that a multifaceted theory of programming can be developed to deliver similar benefits in the practice of software engineering.

The technical material presented in this monograph serves primarily as an illustration or proof of concept. If the concept is found attractive, there is plenty of scope for future work. The most obvious direction is to extend the programming language, for example by including local variables, procedures and parameters. Each extension must be presented in all three semantic styles, with a preferably derivational proof of consistency. An essential goal is to manage each newly introduced complexity by use of as much as possible of the existing theory and algebra: starting again from scratch is forbidden. Ideally, each new feature can be defined and introduced separately, in a way that permits them to be combined without further complexities of interaction. Mature mathematical theories like topology provide an excellent model of the kind of structure needed.

More ambitious extensions may also be of interest: perhaps timing, probability and even a range of paradigms of parallel programming – shared-store, dataflow, and communicating processes. The ultimate goal would be to cover all the interesting paradigms of computing, both declarative and procedural, both in hardware and in software. Again, the challenge is to reveal and profit from their commonality, and so to isolate and control their differences. The work involved is not likely to be easier than fundamental research in any other branch of science; it can be motivated only by the same intense curiosity to find the answer to fundamental questions. This must be supported by the same intense belief in the ultimate simplicity of basic law, and even the elegance of the reasoning on which it is based. The challenge is one that will require the cooperative endeavour of a whole generation of theoretical computing scientists.

The insights described here were obtained by a study of communication and concurrency in parallel processes, where the three semantic styles have been applied individually by independent schools of research to the same class of phenomena. The operational style was used first [**?**] to define the Calculus of Communicating Systems (CCS); the algebraic style took precedence in the definition [**?**] of the Algebra of Concurrent Processes (ACP), whereas the denotational style lies at the basis of the mathematical theory [**?**] of Communicating Sequential Processes (CSP). Many of the detailed differences between these three process theories originate from their different styles of presentation. This monograph attempts a synthesis by presenting the same

programming language in all three styles. The choice of a simple sequential language may defuse some of the controversy that has accompanied research into process algebras.

Not a single idea in this paper is original. The concept of denotational semantics is due to Strachey and Scott [**?**], and the particular choice of ordering of non-deterministic programs is due to Smyth [**?**]. The embedding of programs as predicates is due to Hehner [**?**]. The language is essentially the same as that of Dijkstra [**?**]. The denotational theory is taken from Tarski's calculus of relations [**?**]. The treatment of recursion in specifications is given by Tarski's fixed point theorem [**?**] and for programs by Plotkin [**?**]. The algebraic treatment of the language has already been fully covered in [**?**]. Even the idea of consistent and complementary definitions of programming languages goes back at least to [**?**].

The intention of this monograph is to develop these familiar ideas in a smooth progression. In fact, the smoothness is the result of many laborious iterations, mercifully concealed from the reader. Even more, it is due to witting or unwitting inspiration, advice and assistance from many scientists, including E.W. Dijkstra, A.J.R.G. Milner, J.A. Bergstra, A.W. Roscoe, He Jifeng, G.D. Plotkin, J.W. de Bakker, J. Baeten, M. Hennessy, W.P. de Roever, P. Gardiner, S.A. Schneider, C.C. Morgan, A. Sampaio, B. von Karger, G. Lowe, F. Vaandrager, D.S. Scott, Joachim Parrow.

Acknowledgements

The research which is reported in this monograph was largely inspired and partially supported by the European Community under ESPRIT basic research actions CONCUR and PROCOS.

Chapter 2

Denotational Semantics

When a physical system is described by a mathematical formula, the free variables of the formula are understood to denote results of possible measurements of selected parameters of the system. For example, in the description of a mechanical assembly, it may be understood that x denotes the projection of a particular joint along the x-axis, $\dot{x}$ stands for the rate of change of x, and t denotes the time at which the measurement is taken. A particular observation can be described by giving measured values to each of these variables, for example:

$$x = 14\text{mm} \quad \wedge \quad \dot{x} = 7\text{mm/s} \quad \wedge \quad t = 1.5\text{sec}.$$

The objective of science is not to construct a list of actual observations of a particular system, but rather to describe all possible observations of all possible systems of a certain class. The required generality is embodied in mathematical equations or inequations, which will be true whenever their free variables are given values obtained by particular measurements of any particular system of that class. For example, the differential equation

$$\dot{x} = 0.5 \times x, \qquad \text{for } t \leq 3$$

describes the first three seconds of movement of a point whose velocity varies in proportion to its distance along the x axis. The equation is clearly satisfied by the observation given previously, because

$$7 = 0.5 \times 14 \quad \text{and} \quad 1.5 \leq 3.$$

In applying this insight to computer programming, we shall confine attention to programs in a high level language, which operate on a fixed collection of distinct global variables

$$x, y, \ldots z.$$

The values of these variables are observed either before the program starts or after it has terminated. To name the final values of the variables (observed after the program terminates), we place a dash on the names of the variables

$$x', y', \ldots, z'.$$

But to name the initial values of the variables (observed before the program starts), we use the variable names themselves, without decoration. So an observation of a particular run of a program might be described as a conjunction

$$x = 4 \quad \wedge \quad x' = 5 \quad \wedge \quad y' = y = 7.$$

This is just one of the possible observations of a program that adds one to the variable x, and leaves unchanged the values of y and all the other variables; or in familiar symbols, the program consisting of the single assignment

$$x := x + 1.$$

A general formula describing all possible observations of every execution of the above program is

$$x' = x + 1 \quad \wedge \quad y' = y \quad \wedge \ldots \wedge \quad z' = z.$$

Such a formula will henceforth be abbreviated by the programming notation which it exactly describes; for example, the meaning of an assignment is actually explained by the *definition*

$$x := x + 1 \quad =_{df} \quad x' = x + 1 \wedge y' = y \wedge \ldots \wedge z' = z.$$

Similarly, a program which makes no change to anything is written as $\mathbb{I}$ (pronounced "skip") and defined

$$\mathbb{I} \ =_{df} \ x' = x \wedge y' = y \wedge \ldots \wedge z' = z.$$

In words, an observation of the final state of $\mathbb{I}$ is the same as that of its initial state.

These definitions play the role of specifications, giving an observable criterion of correctness for any implementation of the language. Like all definitions, they cannot actually be proved false or even refuted by experiment: however, as we shall see later, they may fail to define the language that we want to implement and use.

Of course, high level programs are more usually (and more usefully) regarded as instructions to a computer, "*given* certain values of $x, y, \ldots, z$, *to find* values of $x', y', \ldots, z'$ that will make the predicate true". But for the purpose of our mathematical theory, there is no need to distinguish between descriptive and imperative uses of the same predicate.

2.1 Correctness and implication

In engineering practice, a project usually begins with a specification, perhaps embodied in a formal or informal contract between a customer and an implementor. A specification too is a predicate, describing the desired (or at least permitted) properties of a product that does not yet exist. For example, the predicate

$$x' > x \quad \wedge \quad y' = y$$

specifies that the value of x is to be increased, and the value of y is to remain the same. No restriction is placed on changes to any other variable. There are many programs that satisfy this specification, including the previously quoted example

$$x := x + 1.$$

Correctness of a program means that every possible observation of any run of the program will yield values which make the specification true; for example, the specification $(x' > x \wedge y' = y)$ is satisfied by the observation $(x = 4 \wedge x' = 5 \wedge y' = y = 7)$. The formal way of defining satisfaction is that the specification is implied by a description of the observation, for example

$$(x = 4 \wedge x' = 5 \wedge y' = y = 7) \Rightarrow (x' > x \wedge y' = y).$$

This implication is true for all values of the observable variables

$$\forall x, y, \ldots, y', z' :: (x = 4 \wedge x' = 5 \wedge y' = y = 7) \Rightarrow (x' > x \wedge y' = y).$$

In future, we will abbreviate such universal quantification by Dijkstra's conventional square brackets, which surround the universally qualified formula thus

$$[(x = 4 \wedge x' = 5 \wedge y = y' = 7) \Rightarrow (x' > x \wedge y' = y)].$$

In fact, the specification is satisfied not just by this single observation but by every possible observation of every possible run of the program $x := x+1$:

$$[(x := x + 1) \Rightarrow x' > x \wedge y' = y].$$

This mixture of programming with mathematical notations may seem unfamiliar; it is justified by the identification of each program with the predicate describing exactly its range of possible behaviours. Both programs and specifications are predicates over the same set of free variables; and that is why the concept of program correctness can be so simply explained as universally quantified logical implication between a program and its specification.

Logical implication is equally interesting as a relation between two programs or between two specifications. If S and T are specifications, $[S \Rightarrow T]$ means that T is a more general or abstract specification than S, and at least as easy to implement. Indeed, by transitivity of implication, any program that correctly implements S will serve as an implementation of T, though not necessarily the other way round. So a logically weaker specification is easier to implement, and the easiest of all is the predicate *true*, which can be implemented by anything.

Similarly, if P and Q are programs, $[P \Rightarrow Q]$ means that P is a more specific or determinate program than Q, and it is (in general) more useful. Indeed, by transitivity of implication, any specification met by Q will be

met by P, though not necessarily the other way round. So a logically weaker program is for any given purpose less likely to serve; and the weakest program **true** is the most useless of all.

The initial specification of a complex product is usually separated from its eventual implementation by one or more stages of development. The interface between each stage can in principle be formalised as a design document D. If this is also interpreted as a predicate, the correctness of the design is assured by the implication $[D \Rightarrow S]$ and the correctness of the later implementation P by $[P \Rightarrow D]$. The correctness of P with respect to S (and the validity of the whole idea of stepwise development) follows simply by transitivity of implication:

$$\text{If } [P \Rightarrow D] \text{ and } [D \Rightarrow S] \text{ then } [P \Rightarrow S].$$

When a predicate is used as a specification, there is no reason to restrict the mathematical notations available for its expression. Indeed, any notation with a clear meaning should be pressed into service, because clarity of specification is the only protection we have against misunderstanding of the client's requirements, which can often lead to subsequent disappointment or even rejection of a delivered product.

Particularly important aids to clarity of specification are the simple connectives of Boolean algebra, conjunction (and), disjunction (or), and negation (not). Conjunction is needed to connect individual requirements such as "Temperature must be less than 30° *and* more than 27°". Disjunction is needed to provide useful options for economic implementation: "For mixing, use either the pressure vessel *or* the evaporation tank". And negation is needed for even more important reasons: "It must *not* explode".

The freedom of notation which is appropriate for specification cannot be extended to the programming language in which the ultimate implementation is expressed. Programming notations must be selected to ensure computability, compilability, and reasonable efficiency of execution. In a given programming language, there is a limited collection of combinators available for construction of programs from their primitive components. Typical components include assignments, inputs and outputs; and typical combinations include conditionals, sequential composition, and some form of iteration or recursion. It is for good reason that most programming languages exclude the Boolean combinators and quantifiers of mathematical logic. For example, there is no programming language or compiler that would enable you to protect against disaster by writing a program that *causes* an explosion, and then avoid explosion by just negating the program before execution.

A result of these practical restrictions is that, although we can interpret all programs as predicates, the converse is obviously invalid: not every predicate describes the behaviour of a program. For example, consider the extreme predicate *false*. No observation satisfies this predicate, so the only object that it could correctly describe is one that gives rise to no observation whatsoever.

From a scientific viewpoint, such an object does not exist and could never be constructed. The notations of a programming language must therefore be defined to ensure that they can never express the predicate *false*, or any other wholly unimplementable predicate.

However, we must live with the danger of proposing and accepting an unimplementable predicate for specifications. Indeed, any general notational restriction that ensures computability (or even just satisfiability) could seriously impact clarity and conciseness of specification, and so increase the much greater risk of failure to capture the true requirements and goals of the project. Once these have been successfully formalised, a check on implementability (and on efficiency of implementation) may be made separately with the aid of mathematics or good engineering judgement; and this will be confirmed in the end by successful delivery of an actual product which meets the specification. There is fortunately no danger whatsoever of delivering an implementation of an unimplementable specification.

2.2 The programming language

In this section we shall give a denotational semantics of our simple sequential programming language in terms of predicates describing the behaviour of any program expressed in that language. As explained earlier, the variables $x, y, \ldots, z$ stand for the initial values of the like-named global variables of the program, and $x', y' \ldots, z'$ stand for the final values.

Let $e, f, \ldots, g$ stand for expressions such as $x + 1,\ 3 \times y + z, \ldots$ that can feature on the right hand side of an assignment. Clearly, their free variables are confined to the undashed variables of the program; and for simplicity, we assume that all expressions always evaluate successfully to a determinate result. Generalising an example given earlier, we define a simple assignment,

$$(x := e) \quad =_{df} \quad (x' = e \wedge y' = y \wedge \ldots \wedge z' = z).$$

The program which makes no change is just a special case

$$\mathbb{I} \quad =_{df} \quad x := x.$$

A multiple assignment has a list of distinct variables on the left hand side, and a list of the same number of expressions on the right; it is defined

$$(x, y := e, f) \ =_{df} \ (x' = e \wedge y' = f \wedge \ldots \wedge z' = z).$$

A clear consequence of the definition is that an implementation must evaluate all the expressions on the right hand side before assigning any of the resulting values to a variable on the left hand side.

Other consequences can be simply formulated as algebraic laws; they have very simple proofs. For example

$$\begin{array}{rcl} x := e & = & x, y := e, y \\ x, y := e, f & = & y, x := f, e. \end{array}$$

All the definitions and laws extend to lists of more than two variables, for example

$$(z, y := g, f) \;=\; (x, y, \ldots, z := x, f, \ldots, g).$$

In fact every assignment may be transformed by such laws to a *total* assignment

$$x, y, \ldots, z \;:=\; e, f, \ldots, g$$

where the left hand side is a list of *all* the free variables of the program, in some standard order. In future we will abbreviate this to

$$v := f(v),$$

where v is the vector $(x, y, \ldots, z)$ of program variables, and f is a total function from vectors to vectors. Predicates will be similarly abbreviated

$$P(v, v') \;\text{ instead of }\; P(x, y, \ldots, z, x', y', \ldots, z').$$

Any non-trivial program is composed from its primitive components by the combining notations (combinators) of the programming language. The run-time behaviour of a composite program is obtained by actual execution of its components — all, some, or sometimes even none of them. Consequently, at a more abstract level, a predicate describing this composite behaviour can be defined by an appropriate composition of predicates describing the individual behaviours of the components. So a combinator on programs is defined as a combinator on the corresponding predicates. That is part of what it means for a semantics to be denotational.

The first combinator we consider is the *conditional*. Let b be a program expression, containing only undashed variables and always producing a Boolean result (true or false); and let P and Q be predicates describing two fragments of program. A conditional with these components describes a program which behaves like P if b is initially true, and like Q if b is initially false. It may therefore be defined as a simple truthfunction

$$P \lhd b \rhd Q \;=_{df}\; (b \wedge P) \vee (\neg b \wedge Q).$$

A more usual notation for a conditional is

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q \;\text{ instead of }\; P \lhd b \rhd Q.$$

The reason for the change to infix notation is that it simplifies the expression of algebraic laws:

$$
\begin{aligned}
P \lhd\ b \rhd P &= P \\
P \lhd\ b \rhd Q &= Q \lhd \neg b \rhd P \\
(P \lhd\ b \rhd Q) \lhd\ b \rhd R &= P \lhd\ b \rhd (Q \lhd\ b \rhd R) \\
&= P \lhd\ b \rhd R \\
P \lhd\ b \rhd (Q \lhd c \rhd R) &= (P \lhd\ b \rhd Q) \lhd c \rhd (P \lhd\ b \rhd R).
\end{aligned}
$$

The first law expresses idempotence, the second gives a form of skew symmetry, the third is an associative law, and the fourth states the distribution of any conditional operator $\lhd b \rhd$ through the conditional $\lhd c \rhd$, for any condition c. All the laws may be proved by propositional calculus; the easiest way is to consider separately the cases when b is true and when it is false. In the first case, replace $P \lhd\ b \rhd Q$ by P and in the second case by Q. The purpose of the algebraic laws is to help in mathematical reasoning, without such tedious case analyses.

The most characteristic combinator of a sequential programming language is sequential composition, often denoted by semicolon. $(P\,;\,Q)$ may be executed by first executing P and then Q. Its initial state is that of P, and its final state is that of Q. The final state of P is passed on as the initial state of Q; but this is only an intermediate state of $(P\,;\,Q)$, and it cannot be directly observed. All we know is that it exists. The formal definition therefore uses existential quantification to hide the intermediate observation, and to remove the variables which record it from the list of free variables of the predicate.

$$P(v, v')\,;\, Q(v, v') \ =_{df}\ \exists v^0 :: P(v, v^0) \wedge Q(v^0, v').$$

Here, the vector variable v^0 stands for the correspondingly decorated list of bound variables

$$(x^0, y^0, \ldots, z^0).$$

These record the intermediate values of the program variables

$$(x, y, \ldots, z),$$

and so represent the intermediate state as control passes between P and Q. But this operational explanation is far more detailed than necessary. A clever implementation is allowed to achieve the defined effect by more direct means, without ever passing through any of the possible intermediate states. That is the whole purpose of a more abstract definition of the programming language.

In spite of the complexity of its definition, sequential composition obeys some simple, familiar and obvious algebraic laws. For example, it is associative and has $\mathbb{I}$ as its left and right unit. Finally, sequential composition distributes leftward (but not rightward) over the conditional. This asymmetry arises because the condition b is allowed to mention only the initial values of the variables, and not the final (dashed) variables.

$$
\begin{array}{c}
(P\,;\,Q)\,;\,R = P\,;\,(Q\,;\,R) \\
\mathbb{I}\!\mathbb{I}\,;\,P \;=\; P \;=\; P\,;\,\mathbb{I}\!\mathbb{I} \\
(P \lhd\, b \,\rhd Q)\,;\,R \;=\; (P\,;\,R) \lhd\, b \,\rhd (Q\,;\,R).
\end{array}
$$

If e is any expression (only mentioning undashed variables), the assignment

$$x := e$$

changes the value of x so that its final value is the same as the initial value of e, obtained by evaluating e with all its variables taking their initial values. So if $P(x)$ is any predicate mentioning x, P is true of the value of x after the assignment in just the case that P is true of e, i.e.,

$$
\begin{array}{rcl}
x := e\,;\,P(x) & = & (\exists x^0 : x^0 = e : P(x_0)) \\
& = & P(e).
\end{array}
$$

But $P(e)$ is just P with x substituted by e. This substitution effect is defined to generalise to any expression:

$$(x := e\,;\,f(x)) \;=\; f(e).$$

For example

$$(x := x + 1\,;\,(3 \times x + y < z)) \;=\; (3 \times (x + 1) + y < z).$$

Other common notations for substitution are f^x_e, $f[e/x]$ and $f[x/e]$. Substitution permits a rightward distribution law for conditionals:

$$x := e\,;\,(P \lhd\, b \,\rhd Q) \;=\; (x := e\,;\,P) \lhd x := e\,;\,b \rhd (x := e\,;\,Q).$$

Non-determinism is the programming concept that we define next. Let P and Q be predicates describing the behaviour of programs. Their disjunction $(P \;\vee\; Q)$ describes the behaviour of a program which may behave like P or like Q, but with no indication which it will be. As an operator of our programming language, disjunction may be easily implemented by arbitrary selection of either of the operands; and the selection may be made at any time, either before or after the program is compiled or even after it starts execution. Disjunction is an extremely simple explanation of the traditionally obscure phenomenon of non-determinism in computing science; and its simplicity provides additional justification for the interpretation and manipulation of programs as predicates.

All the program combinators defined so far distribute through disjunction. This means that separate consideration of each case is adequate for all reasoning about non-determinism. Curiously, disjunction also distributes through itself and through the conditional

$$
\begin{array}{lcccl}
P \lhd b \rhd (Q \vee R) & = & (P \lhd b \rhd Q) & \vee & (P \lhd b \rhd R) \\
P \,;\, (Q \vee R) & = & (P \,;\, Q) & \vee & (P \,;\, R) \\
(Q \vee R) \,;\, P & = & (Q \,;\, P) & \vee & (R \,;\, P) \\
P \vee (Q \vee R) & = & (P \vee Q) & \vee & (P \vee R) \\
P \vee (Q \lhd b \rhd R) & = & (P \vee Q) & \lhd b \rhd & (P \vee R).
\end{array}
$$

As a consequence of distribution through disjunction, all program combinators also enjoy the property of monotonicity. A function f is said to be *monotonic* if it preserves the relevant ordering, in this case implication. More formally

$$[f.X \Rightarrow f.Y] \text{ whenever } [X \Rightarrow Y].$$

(Here, X and Y are mathematical variables ranging over *predicates* with the same given alphabet, and the line displayed above is true, no matter what predicates take the place of X and Y). All program combinators defined so far are monotonic in all arguments; for example

$$[X \,;\, Y \Rightarrow X' \,;\, Y'] \text{ whenever } [X \Rightarrow X'] \text{ and } [Y \Rightarrow Y'].$$

Monotonicity is a very important principle in engineering. Consider an assembly which tolerates a given range of variation in its working environment. Consider also one of its components, which also has a certain tolerance t. The tolerance of the whole assembly can be expressed as some function f of the component tolerance t. The engineer usually assumes that f is a monotonic function, so that if the component is replaced by one with a broader tolerance t', then the tolerance of the whole assembly will in general also be broader, or at worst, the same:

$$[t \leq t' \Rightarrow f(t) \leq f(t')],$$

where $\leq$ is a partial ordering for comparison of breadth of tolerance. Problems arising from violation of monotonicity are in practice the most difficult to diagnose and rectify, because they invalidate the whole theory upon which design of the assembly has been based.

When faced with the task of implementing a complex specification S, it is usual to make an early decision on the general structure of the product, for example as the sequential composition of two program components. To formalise and communicate this decision, each of these components is going to need separate specifications, say D and E. The correctness of these specifications can be checked before implementation by proof of the implication

$$[(D \,;\, E) \Rightarrow S], \qquad (*)$$

where the sequential composition between specifications has the same definition as between programs considered as predicates. Now what remains are the presumably simpler tasks of finding two programs P and Q which

implement the two designs, i.e.,

$$[P \Rightarrow D] \text{ and } [Q \Rightarrow E].$$

Now just deliver the product $(P\,;\,Q)$. By monotonicity of sequential composition

$$[(P\,;\,Q) \Rightarrow (D\,;\,E)],$$

and the fact that

$$[(P\,;\,Q) \Rightarrow S]$$

follows by transivity from a proof of the correctness of the design step $(*)$. What is more, this proof was completed before the start of implementation of P or Q. The technique can be repeated on the components P and Q; and because of monotonicity it extends to all other program combinators. Their monotonicity is essential to the general engineering method of stepwise design decomposition.

But this account of stepwise design requires a simultaneous guess of both the component designs D and E. This is like trying to factorise a given whole number S by simultaneously guessing both the factors D and E, and then checking the guess by multiplication:

$$D \times E = S.$$

If one of the factors E is already known, there is a much more certain way of calculating D by long division

$$D = S \div E,$$

where $\div$ is an approximate inverse of $\times$, in the sense that

$$D \leq S \div E \;\text{ if and only if }\; D \times E \leq S.$$

Such an inverse is called a Galois connection, and is very useful when exact inverses are not available.

It turns out that sequential composition also has a Galois connection (which we also write as $\div$). So if E specifies some program already designed or already available in a library, it is possible to calculate $S \div E$ as the weakest specification which satisfies the original intention

$$(S \div E); E \Rightarrow S.$$

Such Galois connections exist for any operator that distributes through disjunction (through empty and infinite disjunctions as well). However, the symbolic calculations required to simplify the predicate $S \div E$ may be heavy; and the result will be unimplementable in the case that there is no way of

using E as proposed to implement S. For example

$$(x' \text{ is odd }) \ \div \ (x := 2 \times x) \quad = \quad \mathit{false}.$$

All the symbolic calculations described above require that designs are expressed in a mixture of programming notations (for decisions that have already been taken) and more general predicates (for the parts that are specified but still need to be designed). This is yet another advantage of the philosophy of expressing both programs and specifications in the same logical space of predicates.

2.3 Recursion

A final advantage of monotonicity is that it permits a simple treatment of the important programming concept of recursion, and of its important special case, iteration; without this, no program can take significantly longer to execute than to input. Predicates over a given set of observational variables may be regarded as a complete lattice under implication ordering, with universal quantification as meet and existential as join. The bottom of the lattice is the strongest predicate *false* and the top is **true**. Here we will use bold font to distinguish **true** (considered as a program predicate over free variables v, v') from italic *true*, which is a possible value of a Boolean expression b (containing only undashed free variables v).

Moving to a second-order predicate calculus, we introduce a variable X to stand for an arbitrary predicate over the standard set of first-order variables. Fortunately, all the combinators of our programming language are monotonic, and any formula constructed by monotonic functions is monotonic in all its free variables. Let $G.X$ be a predicate constructed solely by monotonic operators and containing X as its only free predicate variable. Tarski's theorem [?] guarantees that the equation

$$X = G.X$$

has a solution for X; and this is called a fixed point of the function G. Indeed, among all the fixed points, there is a weakest one in the implication ordering. This will be denoted by

$$(\mu X :: G.X).$$

It can be implemented as a single non-recursive call of a parameterless procedure with name X and with body $(G.X)$. Occurrences of X within $(G.X)$ are implemented as recursive calls on the same procedure.

The mathematical definition of recursion is given by Tarski's construction:

$$(\mu X :: G.X) =_{df} \bigvee\{X : [X \Rightarrow G.X] : X\}$$

where $\bigvee$ is the lattice join applied to the set of all solutions of $[X \Rightarrow G.X]$. The following laws state that the join is indeed a fixed point of G, and that it is the weakest such:

$$[G.(\mu X :: G.X) \equiv (\mu X :: G.X)]$$
$$[Y \Rightarrow \mu X :: G.X] \text{ whenever } [Y \Rightarrow G.Y].$$

A simple common case of recursion is the iteration or while loop. If b is a condition,

$$\textbf{while } b \textbf{ do } P$$

repeats the program P for as long as b is true before each iteration. More formally, it can be defined as the recursion

$$(\mu X :: (P\,;\, X) \lhd b \rhd \mathbb{I}).$$

An even simpler example (but hopefully less common) is the infinite recursion which never terminates

$$\mu X.X.$$

This is the weakest solution of the trivial equation

$$X = X;$$

it is therefore the weakest of all predicates, namely **true**. In engineering practice, a non-terminating program is the worst of all programs, and must be carefully avoided by any responsible engineer. That will have to suffice as justification for practical use of a theory which equates any non-terminating program with a totally unpredictable one, which is the weakest in the lattice ordering.

Consider now the program which starts with an infinite loop:

$$(\mu X :: X)\,;\, x, y, \ldots, z := 3, 12, \ldots, 17.$$

In any normal implementation, this would fail to terminate, and so be equal to $(\mu X :: X)$. Unfortunately, our theory gives the unexpected result

$$x' = 3 \ \wedge \ y' = 12 \ \wedge \ldots \wedge \ z' = 17,$$

the same as if the prior non-terminating program had been omitted. To achieve this result, an implementation would have to execute the program backwards, starting with the assignment, and stopping as soon as the values of the variables are known. While backward execution is not impossible (indeed, it is standard for lazy functional languages), it is certainly not efficient for normal procedural languages. Since we want to allow the conventional forward execution, we are forced to accept the practical consequence that the

program

$$(\mu X :: X)\,;\, P$$

will fail to terminate for any program P; and the same is true of

$$P\,;\,(\mu X :: X).$$

Substituting $(\mu X :: X)$ by its value **true** we observe in practice of all programs P that

$$\begin{aligned}&\mathbf{true}\,;\, P = \mathbf{true}\\ &P\,;\,\mathbf{true} = \mathbf{true}.\end{aligned}$$

These laws state that **true** is a zero for sequential composition.

But these laws are certainly not valid for an arbitrary predicate P. As always in science, if a theory makes an incorrect prediction of the behaviour of an actual system, it is the theory that must be adapted; and this usually involves an increase in complication. The violation of plausible simpler laws is what requires and justifies introduction of new concepts and variables, which cannot perhaps be directly observed or controlled, but which are needed to explain what would otherwise be anomalies in more directly observable quantities. All the discoveries of fundamental forces and particles in modern physics have been made in this way. Of course, the old theory is not actually refuted: it would still apply to a lazy implementation. But we are not interested in such an inefficient kind of implementation. It is not the one that we wanted.

2.4 Preconditions and postconditions

In the case of computer programs, the anomaly described in the previous section is resolved by investigating more closely the phenomena of starting and stopping of programs. The collection of free variables describing programs is enlarged to include two new Boolean variables:

ok, which is true of a program that has started in a fully defined state.

ok', which is true of a program that has stopped in a fully defined state.

If ok' is false, the final values of the program variables are unobservable, and the predicate describing the program should make no prediction about these values. Similarly, if ok is false, even the initial values are unobservable. These considerations underlie the validity of the desired zero laws.

The new variables ok and ok' must never appear in the text of any program. However, they *do* appear in the list of existentially quantified variables

in the definition of sequential composition, and in the list of universally quantified variables which are abbreviated by the square brackets.

The variables ok and ok' are useful also in specifications of components of larger programs. The correctness and even the termination of a component with specification Q is often dependent on some assumed properties of the initial values of the variables. This assumption is described by a *precondition* P, which will be true before the program starts. The specification can therefore be written

$$(ok \wedge P) \Rightarrow (ok' \wedge Q),$$

or in words "If the program component starts in a state satisfying P, it stops in a state satisfying Q." For historical reasons, Q is known as a *postcondition*.

The responsibility for ensuring that P is true at the start is thereby delegated to the preceding part of the program. If the assumption is violated, no constraint whatsoever is placed on the behaviour of the program; it may even fail to terminate. Successful teamwork in a large engineering project always depends on appropriately selected assumptions made by the individual designers engaged on a particular sub-task, and the corresponding obligations undertaken by their colleagues. So it is worth while to introduce a special notation

$$(P, Q) =_{df} (ok \wedge P \Rightarrow ok' \wedge Q).$$

This is the primitive notation used by Morgan in [**?**]. The clear distinction of precondition P from postcondition Q is also an advantageous feature of VDM [**?**].

In the interpretation of programs as single predicates, the concepts of correctness and refinement are identified with the implication ordering. The same ordering can be defined for pairs of predicates, as shown by the theorem

$$[(P_1, Q_1) \Rightarrow (P_2, Q_2)] \text{ iff } [P_2 \Rightarrow P_1] \text{ and } [P_2 \ \& \ Q_1 \Rightarrow Q_2].$$

Here, (P_1, Q_1) is better because it has a weaker precondition P_1, and so it can be used more widely; furthermore, in all circumstances (P_2) where (P_2, Q_2) can be used, (P_1, Q_1) has a stronger postcondition, so its behaviour can be more readily predicted and controlled.

Equivalence of predicate pairs can be defined by mutual implication:

$$[(P_1, Q_1) \equiv (P_2, Q_2)] \text{ iff } [(P_1, Q_1) \Rightarrow (P_2, Q_2)] \ \text{ and } \ [(P_1, Q_1) \Rightarrow (P_2, Q_2)].$$

It follows that many equivalent predicate pairs actually denote the same predicate, for example:

$$[(P, Q) \equiv (P, P \wedge Q)] \quad \text{and} \quad [(P, Q) \equiv (P, P \Rightarrow Q)].$$

It is convenient to regard the form $(P, P \Rightarrow Q)$ as a standard form for writing the predicate pair.

The definition of a predicate pair shows how any pair of predicates can be converted to a single predicate by introducing two new variables ok and ok' into its alphabet. The resulting predicate $R(ok, ok')$ is monotonic in ok' and antistrict in ok, in the sense that

$$[R(ok, false) \Rightarrow R(ok, true)] \quad \text{and} \quad [R(false, ok')].$$

Conversely, any predicate R with the above properties can be written as a predicate pair (R_0, R'), where

$$R_0 = \neg R(true, false) \quad \text{and} \quad R' = R(true, true).$$

The definition establishes a bijection between predicate pairs and single predicates with the required monotonic and antistrict properties. All the notations of our programming language (including the refinement ordering on programs) can be applied to predicate pairs, as will be shown in the rest of this section. Consequently the bijection is an isomorphism. For all purposes of mathematical calculation, one may use either interpretation of the meaning of programs.

A significant advantage of explicit mention of preconditions and postconditions is a solution of the postponed problem of undefined expressions in assignments. For each expression e of a reasonable programming language, it is possible to calculate a condition $\mathcal{D}e$ which is $true$ in just those circumstances in which e can be successfully evaluated. For example

$$\begin{array}{lcl} \mathcal{D}17 & = & \mathcal{D}x \ = \ true \\ \mathcal{D}(e+f) & = & \mathcal{D}e \wedge \mathcal{D}f \\ \mathcal{D}(e/f) & = & \mathcal{D}e \wedge \mathcal{D}f \wedge (f \neq 0). \end{array}$$

Successful execution of an assignment relies on the assumption that the expression will be successfully evaluated, so we formulate a *new* definition of assignment

$$x := e \ =_{df} (\mathcal{D}e \ , \ x' = e \ \ \wedge \ \ y' = y \ \ \wedge \ \ \ldots \wedge z' = z).$$

Expressed in words, this definition states that

either the program never starts ($ok = false$) and nothing can be said about its initial and final values,

or the initial values of the variables are such that evaluation of e fails ($\neg \mathcal{D}e$), and nothing can be said about the final values,

or the program terminates ($ok' = true$), and the value of x' is e, and the final values of all the other variables are the same as their initial values.

The definition of the conditional also needs to be modified to take into account the possibility that evaluation of the condition is undefined

$$P \lhd b \rhd Q = (\mathcal{D}b \Rightarrow (b \wedge P \vee \neg b \wedge Q)).$$

However, in future we will maintain the simplifying assumption that all program expressions are everywhere defined.

The normal combinators of the programming language can be defined directly in terms of predicate pairs as shown by the theorems:

$$\begin{array}{lcl}
(P_1, Q_1) \vee (P_2, Q_2) & = & (P_1 \wedge P_2\,,\; Q_1 \vee Q_2) \\
(P_1, Q_1) \lhd b \rhd (P_2, Q_2) & = & (P_1 \lhd b \rhd P_2\,,\; Q_1 \lhd b \rhd Q_2) \\
(P_1, Q_1); (P_2, Q_2) & = & (\neg(\neg P_1; \mathit{true}) \;\&\; \neg(Q_1; \neg P_2)\,,\; Q_1; Q_2).
\end{array}$$

These theorems show that all the combinators of the programming language map pairs of predicates to pairs of predicates. The functions are all monotonic, and pairs of predicates form a complete lattice under implication ordering:

$$\begin{array}{lcl}
\bigvee_i (P_i, Q_i) & = & ((\bigwedge_i P_i), (\bigvee_i Q_i)) \\
\bigwedge_i (P_i, Q_i) & = & ((\bigvee_i P_i), \bigwedge_i (P_i \Rightarrow Q_i)).
\end{array}$$

It follows that the least fixed point of such a function is also expressible as a predicate pair: this property of all programs is therefore maintained even in the presence of recursion.

To be more specific, any function of predicate pairs can be analysed as a pair of functions applied to (P, Q), for example

$$(F(P,Q), \quad G(P,Q)).$$

Here, F is monotonic in P and antimonotonic in Q, whereas for G it is the other way round. The least fixed point is calculated over the complete lattice of predicate pairs, and gives the mutually recursive formula

$$\mu(X,Y) :: (F(X,Y), G(X,Y)) = (P(Q), Q))$$

$$\begin{array}{lcl}
\text{where } P(Y) & = & \nu X :: F(X,Y) \\
Q & = & \mu Y :: (P(Y) \Rightarrow G(P(Y), Y)).
\end{array}$$

Here, νx denotes the strongest fixed point. It may be calculated from the weakest fixed point by the law:

$$(\nu x :: g.x) = (\neg \mu x :: \neg g(\neg x)).$$

The treatment given above applies to arbitrary predicate pairs (P, Q), provided that they do not contain ok or ok'. In particular, the precondition P is even allowed to mention the dashed final values of the program.

Morgan's refinement calculus has shown that this freedom may be useful in the early stages of specification and design. But programs with such preconditions would be expensive or impossible to execute. In fact, all predicate pairs expressible in the notations of the programming language satisfy the restriction that their preconditions do not mention final values. This permits simplification of the precondition for sequential composition. This fact will be proved and used in the next chapter.

The preference of many researchers is to define the predicate pair as the *meaning* of the program, and use the theorems given above as *definitions* of the combinators of the programming language. This avoids the introduction of the "unobservable" variables ok and ok', widely regarded as a distasteful coding trick. However, the coding trick is still useful in revealing the isomorphism between single-predicate and two-predicate definitions of the same language. The programmer may use whichever is most convenient for the immediate purpose, and change freely between them when pursuing different purposes. That is the practical benefit of unifying theories of programming.

For the proof of general algebraic laws which apply equally to specifications as to programs, there is no doubt that the single predicate formulation is the most convenient. However, there are a few important laws which are not valid for general predicates, but only for those that are expressible as predicate pairs. These include the unit laws for composition

$$\mathbb{I};\ (P, Q) = (P, Q) = (P, Q)\ ;\ \mathbb{I}$$

which are valid for the new definition of $\mathbb{I}$ as

$$(\mathit{true}, x' = x \wedge y' = y \wedge \ldots \wedge z' = z).$$

Even more important is the first zero law, which originally motivated introduction of the predicate pairs, which is now trivial

$$\mathbf{true}\ ;\ (P, Q) = \mathbf{true}.$$

The second law zero is

$$P; \mathbf{true} = \mathbf{true}, \quad \text{for all programs } P.$$

This can be proved easily for assignments; and a simple induction extends the proof to all programs written without recursion. However, proof of the recursive case uses methods developed only in the next chapter.

2.5 Predicate transformers

The definition of our programming language was originally given by Dijkstra in terms of predicate transformers. For any program Q, its weakest precondition $P = wp(Q, R)$ maps a postcondition R describing the final state after its

execution onto a predicate P. This is the weakest precondition that describes all initial states in which execution of the program is guaranteed to terminate in a state satisfying the given postcondition. For a *liberal* precondition (wlp), termination is not guaranteed; but if it occurs, the postcondition will be satisfied.

Let Q be a predicate describing the behaviour of a program. Its weakest liberal precondition can be defined

$$wlp(Q, R) =_{df} \neg(Q; \neg R).$$

This means that execution of Q cannot end in a state that fails to satisfy R. From this definition, we can derive theorems that characterise the notations of the programming language, for example,

$$\begin{array}{lcl} wlp(Q_1 \vee Q_2, R) & = & wlp(Q_1, R) \wedge wlp(Q_2, R) \\ wlp(Q_1 \lhd b \rhd Q_2, R) & = & wlp(Q_1, R) \lhd b \rhd wlp(Q_2, R) \\ wlp(Q_1; Q_2, R) & = & wlp(Q_1, wlp(Q_2, R)). \end{array}$$

The simplicity of the last equation is impressive.

The wlp function satisfies the main *healthiness* condition required by Dijkstra, that it distributes through arbitrary universal quantification

$$wlp(Q, \forall i :: R_i) = \forall i :: wlp(Q, R_i).$$

Conversely, let f be any function with this distributive property. Then there is a unique Q such that

$$wlp(Q, R) = f.R, \quad \text{for all } R.$$

The Q is defined by

$$Q = f.\neg \mathbb{I}.$$

An isomorphism has been established between a language defined in terms of single predicates (containing dashed and undashed variables), and one defined in terms of universally distributive predicate transformers. The original and continuing attraction of predicate transformers is that there is no need to use dashed variables: if the postcondition does not contain them, neither does the precondition. Yet another coding trick is eliminated.

The weakest liberal precondition suffers from the same problem with nontermination as the single-predicate theory of programming described in section **??**, for example

$$wlp(\mathbf{true}, \mathit{true}) = \mathit{true}$$

To obtain the full strength of Dijkstra weakest precondition in guaranteeing termination, the two-predicate theory is better. Let (P, Q) be a pair of predi-

cates describing the behaviour of a program, as described in section **??**. This can be converted to a predicate transformer by

$$wp((P,Q),R) =_{df} P \ \& \ wlp(Q,R).$$

Like *wlp*, this transformer is also universally distributive, and satisfies the same laws that characterise the combinators of the programming language. However, non-termination is treated in a more realistic fashion:

$$wp(\mathbf{true}, true) = wp((false, true), true) = false.$$

Chapter 3

The Algebra of Programs.

In this chapter we confine attention to that subset of predicates which are expressible solely in the limited notations of a simple programming language, defined syntactically in table 1. The semantic definitions have been given in the previous section, and provide the basis for proof of a number of algebraic laws. Hitherto, most of these have been valid for arbitrary predicates; but now we can prove additional laws, valid only for predicates which are programs. To emphasize the algebraic orientation, we shall use normal equality between programs in place of the previous universally quantified equivalence

$$P = Q \qquad \text{for} \qquad [P \equiv Q].$$

Such laws are valid for all P and Q ranging over programs. Of course, P and Q themselves are predicates which contain free variables in the appropriate alphabet. Capital letters are used to distinguish these "second order" variables from the lower case variables which they contain as predicates.

Table 1. Syntax.

<program> :: = **true**
| <variable list> := <expression list>
| <program> $\lhd$ <Boolean expression> $\rhd$ <program>
| <program> ; <program>
| <program> $\vee$ <program>
| <recursive identifier>
| μ <recursive identifier> :: <program>
In the form $(\mu X :: P)$, X must be the only free recursive identifier in P.

Algebraic laws in the form of equations and inequations have many advantages in practical engineering. As in more traditional forms of calculus, they are useful in calculating parameters and other design details from more general structural decisions made by engineering judgement. There are good prospects of delegating part of the symbolic calculation to a mechanised term rewriting system like OBJ3 [**?**]. And finally, a theory presented as a set of equations is often easier to teach and to learn than one presented as a mathematical model. Differential calculus is much more widely taught, and more widely used by scientists and engineers, than the foundationary definitions of analysis on which pure mathematicians have shown it to be based.

That is why each of the formal definitions given in the previous section has been followed by a statement of its most important algebraic properties. Proof of these properties is rightly the responsibility of a mathematician; that is the best way of helping engineers, whose skill lies in calculation rather than proof. The goal is to compile a complete collection of laws, so that any other true law that may be needed can be derived by symbolic calculation from the original collection, without ever again expanding the definition of the notations involved.

A valuable aid to the achievement of completeness is the definition of a *normal form*. A normal form uses only a subset of the primitives and combinators of the language, and only in a certain standard order. For example, the conjunctive normal form of Boolean Algebra has conjunction as its outermost combinator, disjunction next, and negation as its innermost operator. The algebraic laws must be sufficient to ensure that every program in the language can be transformed by calculation using just these laws to a program expressed in normal form. There is often a simple test of equality between normal forms; so reduction to normal form generalises this test to arbitrary expressions of the language.

The laws may be classified according to the role that they play in the reduction to normal form.

1. Elimination laws remove operators which are not allowed in the normal form. Such laws contain more occurrences of the operator on one side of the equation than on the other.

2. Distribution laws ensure that the remaining operators can be rearranged to a standard order of nesting.

3. Association and commutation laws are needed to determine equality of normal forms which unavoidably admit variation in their written representation.

For purposes of exposition, we will define a series of normal forms, of increasing complexity and generality, dealing successively with assignment, nondeterminism, non-termination, and recursion.

3.1 Assignment normal form

The first in our series of a normal forms is the total assignment, in which all the variables of the program appear on the left hand side in some standard order:

$$x, y, \ldots, z \ := \ e, f, \ldots, g.$$

Any non-total assignment can be transformed to a total assignment by vacuous extension of the list, for example:

$$(x, y \ := \ e, f) \quad = \quad (x, y, \ldots, z \ := \ e, f, \ldots, z).$$

As mentioned before, we abbreviate the entire list of variables $(x, y, \ldots, z)$ by the simple vector variable v, and the entire list of expressions by the vector expressions $g(v)$ or $h(v)$; these will usually be abbreviated to g or h. Thus the normal form will be written

$$v \ := \ g \quad \text{or} \quad v \ := \ h(v).$$

The law that eliminates sequential composition between normal forms is

$$(v \ := \ g \,; \ v \ := \ h(v)) \quad = \quad (v \ := \ h(g)).$$

The expression $h(g)$ is easily calculated by substituting the expressions in the list g for the corresponding variables in the list v. For example

$$\begin{aligned}(x \,, \ y \ &:= \ x + 1, y - 1 \ ; \ x, y \ := \ y, x) \\ &= (x, y \ := \ y - 1, x + 1).\end{aligned}$$

We now need to assume that our programming language allows conditional expressions on the right hand side of an assignment. Such an expression is defined mathematically

$$\begin{aligned} e \lhd c \rhd f \quad &= \quad e \qquad\qquad \text{if } c \\ &= \quad f \qquad\qquad \text{if } \neg c. \end{aligned}$$

The definition can be extended to lists, for example

$$(e1, e2) \lhd c \rhd (f1, f2) \quad = \quad ((e1 \lhd c \rhd f1), (e2 \lhd c \rhd f2)).$$

Now the elimination law for conditionals is

$$((v \ := \ g) \lhd c \rhd (v \ := \ h)) \quad = \quad v \ := (g \lhd c \rhd h).$$

Finally, we need a law that determines when two differently written normal forms are equal. For this, the right hand sides of the two assignments must be equal:

$$(v \ := \ g) \quad = \quad (v \ := \ h) \quad \text{iff} \quad [g = h].$$

Of course, if g and h are expressions of an undecidable calculus, the algebra of programs will be equally incomplete. This means that a kind of relative completeness has to be accepted as the best that can be achieved in a calculus of programming.

3.2 Non-determinism

Disjunction between two semantically distinct assignments cannot be reduced to a single assignment, which is necessarily deterministic. We therefore move to a more complicated normal form, in which the disjunction operator connects a finite non-empty *set* of total assignments

$$(v := f) \vee (v := g) \vee \ldots \vee (v := h).$$

Let A and B be such sets; we will write the normal form as $\bigvee A$ and $\bigvee B$. All the previous normal forms can be trivially expressed in the new form as a disjunction over the unit set

$$v := g \quad = \quad \bigvee\{v := g\}.$$

The easiest operator to eliminate is disjunction itself; it just forms the union of the two sets:

$$(\bigvee A) \vee (\bigvee B) \quad = \quad \bigvee(A \cup B).$$

The other operators are eliminated by distribution laws

$$(\bigvee A) \lhd b \rhd (\bigvee B) \quad = \quad \bigvee\{P, Q : P \in A \wedge Q \in B : (P \lhd b \rhd Q)\}$$

$$(\bigvee A); (\bigvee B) \quad = \quad \bigvee\{P, Q : P \in A \wedge Q \in B : (P; Q)\}.$$

The right hand sides of these laws are disjunctions of terms formed by applying the relevant operator to total assignments P and Q, which have been selected in all possible ways from A and B. Each of these terms can therefore be reduced to a total assignment, using the laws of **??**. Thus the occurrences of ; and $\lhd b \rhd$ in the right hand sides of the laws given above are also eliminable.

The laws which permit comparison of disjunctions are

$$[(\bigvee A) \Rightarrow R] \text{ iff } \forall P : P \in A : [P \Rightarrow R]$$

$$[v := f \Rightarrow (v := g \vee \ldots \vee v := h)] \text{ iff } [f \in \{g, \ldots, h\}].$$

The first law is a tautology; it enables a disjunction in the antecedent to be split into its component assignments, which are then decided individually by the second law.

3.3 Non-termination

The program constant **true** is not an assignment, and cannot in general be expressed as a finite disjunction of assignments. Its introduction into the language requires a new normal form

$$\textbf{true} \lhd b \rhd P$$

where P is in the previous normal form. It is more convenient to write this as a disjunction

$$b \vee P.$$

Any unconditional normal form P can be expressed as

$$\mathit{false} \vee P$$

and the constant **true** as

$$\mathit{true} \vee \mathbb{I}.$$

The other operators between the new normal forms can be eliminated by the laws

$$\begin{array}{lcl} (b \vee P) \vee (c \vee Q) & = & (b \vee c) \vee (P \vee Q) \\ (b \vee P) \lhd\ d\ \rhd (c \vee Q) & = & (b \lhd\ d\ \rhd c) \vee (P \lhd\ d\ \rhd Q) \\ (b \vee P); (c \vee Q) & = & (b \vee (P; c)) \vee (P; Q). \end{array}$$

The third law relies on the fact that b and c are conditions (not mentioning dashed variables), and P and Q are disjunctions of assignments; from this, it follows that

$$[b; c \Rightarrow b] \text{ and } [b; Q \equiv b]. \qquad (*)$$

We also need a right distribution law

$$x := e; (P \lhd\ b\ \rhd Q) = (x := e; P) \lhd\ x := e; b\ \rhd (x := e; Q)$$

if P and Q are disjunctions of assignments. The law for testing implication is

$$[(b \vee P) \quad \Rightarrow \quad (c \vee Q)] \ \text{ iff } \ [b \Rightarrow c] \ \text{ and } \ [P \Rightarrow c \vee Q].$$

3.4 Recursion

The introduction of recursion into the languages permits construction of a program whose degree of non-determinism depends on the initial state. For example, let n be a non-negative integer variable in

$$\textbf{while } n \text{ is odd } \textbf{do } (n := n \ominus 1 \vee n := n \ominus 2)$$

where $n \ominus k = 0 \lhd k \geq n \rhd n - k$. The effect of this is clearly described by the predicate

$$n' \leq n \;\&\; n' \text{ is even } \&\; (n \text{ is even} \Rightarrow n' = n).$$

Informally, this can be expressed as a disjunction of assignments:

$$\begin{array}{rcl} n & := & (n \lhd n \text{ is even } \rhd n-1) \\ \vee\, n & := & (n \lhd n \text{ is even } \rhd n-3) \\ & & \cdots \\ \vee\, n & := & (n \lhd n \text{ is even } \rhd 0) \end{array}$$

But there is no *finite* set of assignments whose disjunction can replace the informal ellipses (...) shown above, because the length of the disjunction depends on the initial value of n.

The solution is to represent the behaviour as an *infinite* sequence of expressions

$$S = \{i : i \in \mathcal{N} : S_i\}.$$

Each S_i is a finite normal form, as defined in **??**; it correctly describes all the possible behaviours of the program, but maybe some impossible ones as well. So we arrange that each S_{i+1} is potentially stronger and therefore a more accurate description than its predecessor S_i:

$$[S_{i+1} \Rightarrow S_i], \qquad \text{for all } i.$$

This is called the descending chain condition. It allows the later members of the sequence to exclude more and more of the impossible behaviours; and in the limit, every impossible behaviour is excluded by some S_i, provided that i is large enough. Thus the exact behaviour of the program is captured by the intersection of the whole sequence, written $(\bigwedge_i S_i)$, or more briefly $(\bigwedge S)$.

For the example shown above, we define the infinite sequence S as follows

$$\begin{array}{rcl} S_0 & = & \mathit{true} \\ S_1 & = & n' = n \lhd n \text{ is even} \rhd n \geq 1 \\ S_2 & = & n' = n \lhd n \text{ is even} \rhd (n \geq 3 \vee n' = 0) \\ S_3 & = & n' = n \lhd n \text{ is even} \rhd (n \geq 5 \vee n' \in \{0, 2\}) \\ & \vdots & \\ S_i & = & n' = n \lhd n \text{ is even} \rhd (n \geq 2i-1 \vee n' < i \;\&\; n' \text{ even}). \end{array}$$

Each S_i is expressible in finite normal form. It describes exactly the behaviour of the program when n is initially less than $2i$, so that the number of iterations is bounded by i. The exact behaviour of the program is described by any S_i with i greater than the initial value of n. It follows that the predicate describing the behaviour of the whole program is equal to the required infinite conjunction $\bigwedge_i S_i$. The laws for recursion given below will provide a

general method for calculating the successive approximations S_i describing the behaviour of any particular loop.

The calculation depends critically on the descending chain condition for S, because it is this that permits distribution of all the operators of the language through intersection:

$$\begin{array}{lcll}
(\bigwedge S) \vee P & = & \bigwedge_i (S_i \vee P) & \\
(\bigwedge S) \lhd b \rhd P & = & \bigwedge_i (S_i \lhd b \rhd P) & \\
P \lhd b \rhd (\bigwedge S) & = & \bigwedge_i (P \lhd b \rhd S_i) & \\
(\bigwedge S); P & = & \bigwedge_i (S_i; P) & \\
P; (\bigwedge S) & = & \bigwedge_i (P; S_i) & \text{provided that } P \text{ is in finite normal form.}
\end{array}$$

Operators that distribute through intersections of descending chains are called *continuous*. Every combination of continuous operators is also continuous in *all* its arguments. This permits formulation of a continuity law for recursion:

$$\mu X :: \bigwedge_i S_i.X = \bigwedge_i \mu X :: S_i.X$$

provided that S_i is continuous for all i, and it forms a descending chain for all X, i.e.,

$$[S_{i+1}.X \Rightarrow S_i.X].$$

Another advantage of the descending chain condition is that a descending chain of descending chains can be reduced to a single descending chain by diagonalisation

$$\bigwedge_k (\bigwedge_l S_{kl}) \quad = \quad \bigwedge_i S_{ii},$$

provided that $\forall k, l, i.\ S_{ki} \leq S_{ki+1}\ \&\ S_{il} \leq S_{i+1l}$. This ensures that a function F, continuous in each of its arguments separately, is also continuous in its arguments taken together

$$F(\bigwedge S, \bigwedge T) = \bigwedge_i F(S_i, T_i).$$

In turn, this gives the required elimination laws for the three operators of the language

$$\begin{array}{lcl}
(\bigwedge S) \vee (\bigwedge T) & = & \bigwedge_i (S_i \vee T_i) \\
(\bigwedge S) \lhd b \rhd (\bigwedge T) & = & \bigwedge_i (S_i \lhd b \rhd T_i) \\
(\bigwedge S); (\bigwedge T) & = & \bigwedge_i (S_i; T_i).
\end{array}$$

The occurrence of the operators on the right hand side of these laws can eliminated by the laws of **??**, since each S_i and T_i is finite.

The continuity laws ensure that descending chains constitute a valid normal form for all the combinators of the language; and the stage is set for treatment of recursion. Consider first an innermost recursive program (containing no other recursions)

$$\mu X :: F.X,$$

where $F.X$ contains X as its only free recursive identifier. The recursive identifier X is certainly not in normal form, and this makes it impossible to express $F.X$ in normal form. However, all the other components of $F.X$ are expressible in normal form, and all its combinators permit reduction to normal form. So, if X were replaced by a normal form (say **true**), (F.**true**) can be reduced to finite normal form, and so can $F.(F.\mathbf{true})$, $F.(F.(F.\mathbf{true}))$,... Furthermore, because F is monotonic, this constitutes a descending chain of normal forms. Since F is continuous, by Kleene's famous recursion theorem, the limit of this chain is the least fixed point of F

$$(\mu X :: F.X) \quad = \quad \bigwedge_n F^n.\mathbf{true}.$$

This reduction can be applied first to replace all the innermost recursions in the program by limits of descending chains. The remaining innermost recursions now have the form

$$\mu Y :: H(\bigwedge_m F^m.\mathbf{true}, \bigwedge_m G^m.\mathbf{true}, \ldots, Y).$$

By continuity of H, this transforms to

$$\mu Y :: \bigwedge_m H_m.Y$$

where $H_m.Y = H.(F^m.\mathbf{true}, G^m.\mathbf{true}, \ldots, Y)$, which is (for fixed Y) a descending chain in m. By continuity of μ, this equals

$$\bigwedge_m \mu Y :: H_m.Y.$$

and by Kleene's theorem

$$\bigwedge_m \bigwedge_n (H_m^n.\mathbf{true}).$$

Because this is descending in both n and m, we get

$$\bigwedge_n H_n.\mathbf{true}.$$

Thus the next innermost recursions are converted to normal form; by repeating the process, the whole program can be converted to normal form

$$\bigwedge_n S_n.$$

Another way of describing the same conversion is that S_n is the result of replacing *every* recursion $(\mu X :: F.X)$ in the program by the n^{th} element of

its approximating series, i.e, F^n. **true**.

There is no direct algebraic way of testing equality between limits of descending chains. A partial solution to this problem is to relax the descending chain condition, and represent a program as the conjunction of *all* finite normal forms which it implies. For all programs P,

$$P = \bigwedge\{X : X \text{ is a finite normal form and } [P \Rightarrow X] : X\}.$$

This means that if P and Q are different programs, there exists a finite normal form that is implied by one of them and not the other.

The proof of this fact is not trivial, and relies on the continuity of all the operators in the programming language. It also suggests another normal form for programs as intersections of arbitrary sets of finite forms. Each operator F of the language is applied pointwise to the finite forms in their operands

$$F.S = \{P : \ P \in S : \ F.P\}.$$

Recursion is explained as the greatest fixed point in the inclusion ordering for sets of finite forms. The original semantics of the language should be recoverable by applying the $\bigwedge$ operator to all the sets.

But this construction will not work for arbitrary sets of finite forms: they have to satisfy certain additional *closure* properties. These are properties shared by any set generated from a program P by taking all its finite approximations:

$$S = \{p : [P \Rightarrow p] : p\}.$$

1. If $p \in S$ and $p \Rightarrow q$ then $q \in S$.

2. If $T \subseteq S$ and $\bigwedge T$ is expressible in finite form then $\bigwedge T \in S$.

The calculus of programs should be extensible to intersections of arbitrary sets which are closed in this sense.

The finite normal forms play a role similar to that of rational numbers among the reals. Firstly, there is only a countable number of them. A second similarity is that every real is the limit of a descending chain of rationals. Finally, the rationals are dense, in the sense that any two distinct real numbers can be shown to be so by a rational number which separates them. The application of these insights to computer programs is the contribution of Scott's theory of continuous domains.

3.5 Computability

The algebraic laws given in **??**, **??** and **??** permit every finite program (one that does not use recursion) to be reduced to finite normal form. The reduction rules are nothing but simple algebraic transformations, of the kind that

can be readily mechanised on a computer, and therefore even on a Turing machine. The infinite normal form $(\bigwedge_i S_i)$ of section **??** can never be computed in its entirety; however, for each n, the finite normal form S_n can be readily computed; for example by replacing each internal recursion $(\mu X :: F.X)$ by $(F^n.\ \mathbf{true})$.

This suggests a highly impractical method of executing a program, starting in a known initial state s, in which Boolean conditions can be evaluated to *true* or *false*. The machine calculates the series S_n of finite normal forms from the program. Each of these is a disjunction $(b_n \vee P_n)$. If $(s; b_n)$ evaluates to *true*, the machine continues to calculate the next S_{n+1}. If *all* the $(s; b_n)$ are *true*, this machine never terminates; but that is the right answer, because in this case the original program, when started in the given initial state s, contains an infinite recursion or loop. But as soon as a false $(s; b_n)$ is encountered, the corresponding P_n is executed, by selecting and executing an arbitrary one of its constituent assignments. We want to prove that the resulting state will be related to the initial state by the predicate $(\bigwedge_i S_i)$. Unfortunately, this will not be so if the selected assignment is not represented in S_m, for some m greater than n.

The validity of this method of execution depends on an additional property of the normal form, that once n is high enough for b_n to be false, all the assignments P_m remain the same as P_n, for all m greater than n. This can be formalised:

$$[(b_n \vee P_n) \equiv (b_n \vee P_{n+k})],\ \text{for all}\ \ n, k.$$

Let us therefore define a new ordering relation $\leq$ between normal forms, one that is stronger than the familiar implication ordering. For finite normal forms, this requires that if the condition of the weaker program is false, its effect is exactly the same as that of the stronger program

$$(b \vee P) \leq (c \vee Q)\ \ \text{iff}\ \ [b \Rightarrow c]\ \ \text{and}\ \ [\neg c \Rightarrow (P \equiv Q)].$$

This is clearly a preorder, with weakest element $(\mathit{true} \vee \mathbf{true})$. What is more, it is respected by all the combinators of the programming language. If $F.X$ is a program, it follows that $\{i :: F^i.\mathbf{true}\}$ is a descending chain in this new ordering. This shows that all innermost recursions enjoy the property that we are trying to prove.

Furthermore, because of monotonicity, any program combinator H preserves this property:

$$H(\textstyle\bigwedge_i S_i) = \bigwedge_i (H.S_i).$$

For nested recursions, the proof uses the same construction as given at the end of the previous section. All the chains involved are descending in the new ordering as well.

3.6 Completeness

A reduction to normal form gives a method of testing the truth of any proposed implication between any pair of programs: reduce both of them to normal form, and test whether the inequation satisfies the simpler conditions laid down for implication of normal forms. If so, it holds also between the original programs. This is because the reduction laws only substitute equals for equals and each of the tests for implication between normal forms has been proved as a theorem.

For the algebra of programs, the converse conclusion can also be drawn: if the test for implication fails for the normal forms, then the implication does not hold between the original programs. The reason is that the tests give both necessary and sufficient conditions for the validity of implication between normal forms. For this reason, the algebraic laws are said to be *complete*. Of course, since the normal form is infinite, there cannot be any general decision procedure.

Completeness is a significant achievement for a theory of programming. Each of the laws requires a non-trivial proof, involving full expansion of the definitions of all the operators in the formulae, followed by reasoning in the rather weak framework of the predicate calculus. But after a complete set of laws have been proved in this more laborious way, proof of any additional laws can be achieved by purely algebraic reasoning; it will never be necessary again to expand the definitions.

For example, we have to prove the right zero law

$$P\,;\,\mathbf{true} = \mathbf{true}.$$

Since P is a program, it can be reduced to normal form $\bigwedge S$.

$$\begin{array}{rcl} \bigwedge S;\,\mathbf{true} & = & \bigwedge_i (b_i \vee P_i);\mathbf{true} \\ & = & \bigwedge_i (b_i \vee (\bigvee_j (v := e_j)\,;\,\mathbf{true})) \\ & = & \bigwedge_i b_i \vee (\bigvee_j (v := e_j\,;\,\mathbf{true})) \\ & = & \bigwedge_i b_i \vee (\bigvee_j \mathbf{true}) \\ & = & \mathbf{true}. \end{array}$$

Apart from the practical advantages, completeness of the laws has an important theoretical consequence in characterising the nature of the programming language. For each semantically distinct program there is a normal form with the same meaning, and this can be calculated by application of the laws. It is therefore possible to regard the normal form itself as a definition of the meaning of the program, and to regard the algebraic laws as a definition of the meaning of the programming language, quite independent of the interpretation of programs as predicates describing observations. This is the philosophy of "initial algebra" semantics for abstract data types.

There are many advantages in this purely algebraic approach. Firstly, algebraic reasoning is much easier in practical use than the predicate calculus. It is often quite easy to decide what laws (like the zero laws) are needed or wanted for a programming language; and then it is much easier just to postulate them than to prove them. And there is no need to indulge in curious coding tricks, like the introduction of ok and ok'. Finally, most algebraic laws are valid for many different programming languages, just as most of conventional schoolroom algebra holds for many different number systems. Even the differences between the systems are most clearly described and understood by examining the relatively simple differences in their algebraic presentations, rather than the widely differing definitions which they are given in the foundations of mathematics.

The real and substantial benefits of algebra are achieved by completely abstracting from the observational meaning of the variables and operators occurring in the formulae. Full advantage should be taken of the benefits of this abstraction, and for as long as possible. But if the algebra is ever going to be applied, either in engineering or in science (or even in mathematics itself), the question arises whether the laws are actually true in the application domain.

To answer this question, it is necessary to give an independent meaning to the variables and operators of the algebra, and then to prove the laws as theorems. It is a matter of personal choice whether the investigation of algebra precedes the search for possible meanings, or the other way round (as in this monograph). The experienced mathematician probably explores both approaches at the same time. When the task is complete, the practising engineer or programmer has a secure intellectual platform for understanding complex phenomena and a full set of calculation methods for the reliable design of useful products. And that is the ultimate, if not the only, goal of the investigations.

Chapter 4

Operational Semantics

The previous chapters have explored mathematical methods of reasoning about specifications and programs and the relationships between them. But the most important feature of a program is that it can be automatically executed by a computing machine, and that the result of the computation will satisfy the specification. It is the purpose of an operational semantics to define the relation between a program and its possible executions by machine. For this we need a concept of execution and a design of machine which are sufficiently realistic to provide guidance for real implementations, but sufficiently abstract for application to the hardware of a variety of real computers. As before, we will try to derive this new kind of semantics in such a way as to guarantee its correctness.

In the most abstract view, a computation consists of a sequence of individual *steps*. Each step takes the machine from one state m to a closely similar one m'; the transition is often denoted $m \to m'$. Each step is drawn from a very limited repertoire, within the capabilities of a simple machine. A definition of the set of all possible single steps simultaneously defines the machine and all possible execution sequences that it can give rise to in the execution of a program.

The step can be defined as a relation between the machine state before the step and the machine state after. In the case of a stored program computer, the state can be analysed as a pair (s, P), where s is the data part (ascribing actual values to the variables $x, y, \ldots, z$), and P is a representation of the rest of the program that remains to be executed. When this is $\mathbb{I}$, there is no more program to be executed; the state $(t, \mathbb{I})$ is the last state of any execution sequence that contains it, and t defines the final values of the variables.

It is extremely convenient to represent the data part of the state by a total assignment

$$x, y. \ldots, z \ := \ k, l, \ldots, m,$$

where $k, l, \ldots, m$ are *constant* values which the state ascribes to $x, y, \ldots, z$ respectively. If s is an initial data state interpreted as an assignment, and if P is any program interpreted as a predicate, $(s; P)$ is a predicate like P, except that all occurrences of undashed program variables have been replaced by their initial values $(k, l, \ldots, m)$. If this is the identically **true** predicate, execution of P started in s may fail to terminate. Otherwise it is a description of all the possible final data values v' of any execution of P started in s. If t is any other data state,

$$[(t; \mathbb{I}) \Rightarrow (s; P)]$$

means that the final state $(t, \mathbb{I})$ is one of the possible results of starting execution of P in state s. Furthermore,

$$[t; Q \Rightarrow s; P]$$

means that every result (including non-termination) of executing Q starting from data state t is a permitted result of executing P from state s. If this implication holds whenever the machine makes a step from (s, P) to (t, Q), the step will be correct in the sense that it does not increase the set of final states that result from the execution; and if ever a final state $(t, \mathbb{I})$ is reached by a series of such steps, that will be correct too.

In order to derive an operational semantics, let us start by regarding the machine step relation $\rightarrow$ as just another (backwards) way of writing implication:

$$(s, P) \rightarrow (t, Q) \quad \text{instead of} \quad [(t; Q) \Rightarrow (s; P)].$$

The following theorems are now trivial.

1. $\quad (s, v := e) \quad \rightarrow \quad ((v := (s; e)), \mathbb{I}).$

The effect of a total assignment $v := e$ is to end in a final state, in which the variables of the program have constant values $(s; e)$, i.e., the result of evaluating the list of expressions e with all variables in it replaced by their initial values. Here, we return to the simplifying assumption that expressions are everywhere defined.

2. $\quad (s, (\mathbb{I}; Q)) \quad \rightarrow \quad (s, Q).$

A $\mathbb{I}$ in front of a program Q is immediately discarded.

3. $\quad (s, (P; R)) \quad \rightarrow \quad (t, (Q; R)), \qquad$ whenever $(s, P) \rightarrow (t, Q)$

The first step of the program $(P; R)$ is the same as the first step of P, with R saved up for execution (by the preceding law) when P has terminated.

4. $\quad (s, P \vee Q) \quad \rightarrow \quad (s, P)$
$\quad\quad (s, P \vee Q) \quad \rightarrow \quad (s, Q)$

The first step of the program $(P \vee Q)$ is to discard either one of the components P or Q. The criterion for making the choice is completely undetermined.

5. $\quad (s, P \lhd b \rhd Q) \quad \rightarrow \quad (s, P) \qquad$ whenever $s; b$
$\quad\quad (s, P \lhd b \rhd Q) \quad \rightarrow \quad (s, Q) \qquad$ whenever $s; \neg b$

The first step of the program $(P \lhd b \rhd Q)$ is similarly to select one of P or Q, but the choice is made in accordance with the truth or falsity of $(s; b)$,

i.e., the result of evaluating b with all free variables replaced by their initial values.

6. $(s, \mu X :: F.X) \quad \rightarrow \quad (s, F.(\mu X :: F.X))$

Recursion is implemented by the copy rule, whereby each recursive call within the procedure body is replaced by the whole recursive procedure.

7. $(s, \mathbf{true}) \qquad \rightarrow \qquad (s, \mathbf{true})$.

The worst program **true** engages in an infinite repetition of vacuous steps.

The formulae 1-7 have been presented above as theorems, easily provable from the algebraic laws of the programming language. But it is even easier to present these laws as a *definition* of the way in which the programs are to be executed. In this definition, the components s and P of each state are represented concretely by their texts. The states are identified as pairs (`s`, `P`), where `s` is a *text* describing the data state, and `P` is a program *text*, defining the program state. We use typewriter font to distinguish text from meaning: `P` is the text of a program whose meaning is the predicate P. The symbol $\rightarrow$ is a relation between the texts; it is actually *defined* inductively by the list of laws given above: it is the *smallest* relation satisfying just those laws.

A purely operational presentation of a programming language has many advantages. Firstly, it is surprisingly short and simple; furthermore it closely reflects the programmer's understanding (sometimes called "intuition") of how programs are actually executed. This is especially important in the diagnosis of unexpected behaviour of a program, carelessly written perhaps by someone else. Finally, the clear correspondence to an executing mechanism gives an immediate guarantee against the danger of incomputability, without the long proof of section **??**. Many researchers into the theory of programming regard an operational semantics as the standard or even the only way of starting research in the subject.

The disadvantages of an operational presentation arise from its extremely low level of abstraction. Mathematical reasoning based on this presentation has to appeal constantly to the completeness clause, i.e., to the absence of any transitions other than those postulated. Small additions or changes in the laws can have unexpectedly gross (or even rather subtle) effects on the theory. What is worse, the assumption that the whole program text is observable means that there can be no non-trivial algebraic equations between programs. Finally, since operational reasoning starts with the text of the program, it is absolutely useless for direct reasoning about specification and design *before* the program is written. In the later sections of this chapter, we will investigate methods of raising the level of abstraction of operational semantics to that of the algebraic or even denotational semantics.

4.1 Total correctness

In the previous section, we used the implication relation $\Rightarrow$ to justify the seven clauses of the operational semantics of the language. The intention was to guarantee the consistency of the operational semantics, in the sense that every final state of an execution satisfies the predicate associated with the program. This is certainly a necessary condition for correctness. But it is not a sufficient condition. There are two kinds of error that it does not guard against:

(i) There may be too few transitions (or even none at all!). An omitted transition would introduce a new and unintended class of terminal state. A more subtle error would be omission of the second of the two laws (4) for $(P \vee Q)$, thereby eliminating non-determinism from the language.

(ii) There may be too many transitions. For example, the transition

$$(s, Q) \rightarrow (s, Q)$$

is entirely consistent with the approach of the previous section, since it just expresses reflexivity of implication. But its inclusion in the operational definition of the language would mean that every execution of every program could result in an infinite iteration of this dumb step.

To guard against these dangers, we need to obtain a clear idea of what it means for an operational semantics to be correct. The purpose of an operational semantics is to define the "machine code" of an abstract machine. One of the best and most common ways of defining a machine code is to write an interpreter for it in a high level programming language, whose meaning is already known; and a language immediately available for this purpose is the one whose observational meaning has been given in Chapter 2. The criterion of total correctness of the interpreter is that its application to a textual representation of the program `P` gives rise to exactly the same observations as are described by P, when given its meaning as a predicate.

The alphabet of global variables of our interpreter will be

s to hold the data state (as text)

p to hold the program state (as text).

They will be updated on each step of the interpreter by the assignment

$$s, p := \mathit{next}\ (s, p).$$

The definition of the *next* function operating on texts is taken directly from the individual clauses of the operational semantics; for example:

$$\mathit{next}\ (s, \mathtt{I}; \mathtt{Q}) = (s, \mathtt{Q}), \quad \text{for all } s.$$

For non-determinism we need two assignments

$$s, p := \ next1\,(s, p) \vee s, p := \ next2\,(s, p) \qquad \ldots \text{ STEP}$$

where $next1\,(s, \mathtt{P} \vee \mathtt{Q}) = (s, \mathtt{P})$ and $next2\,(s, \mathtt{P} \vee \mathtt{Q}) = (s, \mathtt{Q})$. In all other cases $next1\,(s, \mathtt{P}) = next2\,(s, \mathtt{P}) = next\ (s, \mathtt{P})$. The inner loop of the interpreter repeats this step until the program terminates

$$(p \neq \text{`}\mathbb{I}\text{'}) * \text{ STEP} \qquad \ldots \text{ LOOP.}$$

The interpreter defined as LOOP is a predicate describing the relationship between the initial and final values of the program variables s and p. It leaves unchanged the values of the program variables $x, y, \ldots, z$. The predicate P, corresponding to the initial value of p, describes the way that the abstract variables $x, y, \ldots, z$ are updated. The interpreter is correct if the updates correspond with each other in some appropriate sense. The relevant sense is described exactly by the final value of the variable s, produced by the interpreter. This should describe one of the particular final values permitted by the predicate P.

So there are two ways of describing the execution of a program P.

(i) First initialise the values of s and p to $\mathtt{S}$ and $\mathtt{P}$. Then execute the interpreter LOOP. This updates s and p, but not the program variables $x, y, \ldots, z$. If and when the loop terminates, execute the final value of s, which will assign the desired final values to the program variables. The program variables s and p are local to the interpreter, and we are no longer interested in their final values. So they may as well be reinitialised. In summary, the effect of this method of execution is described by the predicate

$$s, p := \mathtt{S},\ \mathtt{P}\ ;\ \text{LOOP};\ s\ ;\ s, p := \mathtt{S},\ \mathtt{P}$$

where the programming notations have the meaning described in Chapter 2.

(ii) First execute the initial value $\mathtt{S}$, thereby assigning to the program variables the same values as they have in the interpreted execution (i). Then execute the program P, which updates the program variables, hopefully to the same final state as that produced by the interpreter. The variables s and p can then be updated to their same final state as in (i). This execution method is described by the predicate

$$S\ ;\ P\ ;\ s, p := \mathtt{s},\ \mathtt{P}.$$

It may be worth a reminder that $\mathtt{S}$ and $\mathtt{P}$ are constant program texts, S and P give their meaning as predicates, whereas s and p are local variables of the interpreter, and their values are changed by the LOOP.

The hope that the two execution methods are the same can be justified by proof of the equation between the two predicates displayed under (i) and (ii). If this can be proved entirely from the algebraic laws, it establishes the desired correspondence between the operational and the algebraic semantics. If the theorem is weakened to an implication, the operational semantics is still correct, but may be more deterministic than the denotational.

Note that both the statement of the theorem and its proof depend utterly on acceptance of the priority of the denotational or algebraic definition as a *specification* of the language. Indeed, unless there is such an independent specification, the question of correctness of an operational semantics cannot arise.

In the absence of an independent meaning for the programming language, the operational semantics may be the only one acceptable or available as a foundation for a theory of programming. In that case, other methods are needed for escaping from its deplorably low level of abstraction. Such methods are discussed in the remaining sections of this chapter. They will be of only secondary interest to readers who recognise that the essential task of proving equivalence of denotational, algebraic and operational semantics has already been completed.

4.2 Bisimulation

As mentioned above, the operational semantics uses the actual text of programs to control the progress of the computation; in principle, two programs are equal only if they are written in exactly the same way, so there cannot be any non-trivial algebraic equations. Instead, we have to define and use some reasonable *equivalence* relation (conventionally denoted $\sim$) between program texts. In fact, it is customary to define this relation between complete machine states, including the data part. Two programs `P` and `Q` will then be regarded as equivalent if they are equivalent whenever they are paired with the same data state:

$$\mathtt{P} \sim \mathtt{Q} =_{df} \forall \mathtt{s} :: (\mathtt{s}, \mathtt{P}) \sim (\mathtt{s}, \mathtt{Q}).$$

Now the basic question is: what is meant by a "reasonable" equivalence between states? The weakest equivalence is the universal relation, and the strongest is textual equality; clearly we need something between these two extremes. There are a great many possible answers to these questions; but we shall concentrate on two of the first and most influential of them, which are due to Milner and Park [**?**].

To exclude the universal relation, it is sufficient to impose an obligation on a proposed equivalence relation $\sim$ that it should preserve the distinctness of a certain minimum of "obviously" distinguishable states. For example, the terminal states of each computation are intended to be recognisable as such, and their data parts are intended to be directly observable. We define the

terminal states:

$$(\mathtt{s},\mathtt{P}) \not\rightarrow =_{df} \neg\exists(\mathtt{t},\mathtt{Q}) :: (\mathtt{s},\mathtt{P}) \rightarrow (\mathtt{t},\mathtt{Q}),$$

and require that

(i) if $(\mathtt{s},\mathtt{P}) \sim (\mathtt{t},\mathtt{Q})$ and $(\mathtt{t},\mathtt{Q}) \not\rightarrow$, then $\mathtt{s} = \mathtt{t}$ and $(\mathtt{s},\mathtt{P}) \not\rightarrow$.

The second condition on a reasonable equivalence relation between states is that it should respect the transition rules of the operational semantics; or more formally

(ii) If $(\mathtt{s},\ \mathtt{P}) \sim (\mathtt{t},\ \mathtt{Q})$ and $(\mathtt{t},\ \mathtt{Q}) \rightarrow (\mathtt{v},\ \mathtt{S})$ then there is a state $(\mathtt{u},\ \mathtt{R})$ such that

$$(\mathtt{s},\ \mathtt{P}) \rightarrow (\mathtt{u},\ \mathtt{R}) \text{ and } (\mathtt{u},\ \mathtt{R}) \sim (\mathtt{v},\ \mathtt{S}).$$

(iii) Since $\sim$ is an equivalence relation, the same must hold for the converse of $\sim$.

The condition (ii) may be expanded to a weak commuting diagram:

$$\begin{array}{ccc} (\mathtt{u},\mathtt{R}) & \sim & (\mathtt{v},\mathtt{S}) \\ \uparrow & \supseteq & \uparrow \\ (\mathtt{s},\mathtt{P}) & \sim & (\mathtt{t},\mathtt{Q}) \end{array}$$

Or it may be contracted to a simple inequation in the calculus of relations:

$$(\sim;\rightarrow) \subseteq (\rightarrow;\sim).$$

A relation $\sim$ which satisfies these three reasonable conditions is called a strong *bisimulation*. Two states are defined to be strongly *bisimilar* if there exists *any* strong bisimulation between them. So bisimilarity is a relation defined as the union of *all* bisimulations. Fortunately, distribution of relational composition through such unions means that bisimilarity is itself a bisimulation, in fact the weakest relation satisfying the three reasonable conditions listed above.

An additional most important property of an equivalence relation is that it should be respected by all of the operators of the programming language, for example:

$$\text{If } \mathtt{P} \sim \mathtt{P}' \text{ and } \mathtt{Q} \sim \mathtt{Q}' \text{ then } (\mathtt{P};\mathtt{Q}) \sim (\mathtt{P}';\mathtt{Q}').$$

An equivalence relation with this property is called a *congruence*. It justifies the principle of substitution which underlies all algebraic calculation and reasoning: without it, the algebraic laws would be quite useless. Fortunately, the bisimilarity relation happens to be a congruence for all the operators mentioned in our operational semantics; this claim needs to be checked by

mathematical proof.

As an example of the use of bisimulation, we will prove the commutative law for disjunction. The trick is to define a relation $\sim$ which makes the law true, and then prove that it has the properties of a bisimulation. So let us define a reflexive relation $\sim$ that relates every state of the form $(\mathtt{s}, \mathtt{P} \vee \mathtt{Q})$ with itself and with the state $(\mathtt{s}, \mathtt{Q} \vee \mathtt{P})$, and relates every other state only to itself. This is clearly an equivalence relation. It vacuously satisfies bisimilarity condition (i). Further if $(\mathtt{s}, \mathtt{P} \vee \mathtt{Q}) \rightarrow (\mathtt{t}, \mathtt{X})$ then inspection of the two transitions for $\vee$ reveals that $\mathtt{t} = \mathtt{s}$ and $\mathtt{X}$ is either $\mathtt{P}$ or $\mathtt{Q}$. In either case, inspection of the laws shows that $(\mathtt{s}, \mathtt{Q} \vee \mathtt{P}) \rightarrow (\mathtt{s},\mathtt{X})$. The mere existence of this bisimulation proves the bisimilarity of $(\mathtt{P} \vee \mathtt{Q})$ with $(\mathtt{Q} \vee \mathtt{P})$.

But there is no bisimulation that would enable one to prove the idempotence laws for disjunction. For example, let $\sim$ be a relation such that

$$(\mathtt{s}, \mathbb{I}) \sim (\mathtt{s}, \mathbb{I} \vee \mathbb{I}).$$

Clearly $(\mathtt{s}, \mathbb{I} \vee \mathbb{I}) \rightarrow (\mathtt{s}, \mathbb{I})$. However, the operational semantics deliberately excludes any $(\mathtt{t}, \mathtt{X})$ such that $(\mathtt{s}, \mathbb{I}) \rightarrow (\mathtt{t}, \mathtt{X})$; condition (ii) for bisimulation is therefore violated. In fact, this condition is so strong that it requires any two bisimilar programs to terminate in exactly the same number of steps. Since one of the main motives for exploring equivalence of programs is to replace a program by one that can be executed in fewer steps, strong bisimilarity is far too strong a relation for this purpose.

Milner's solution to this problem is to define a weak form of bisimilarity (which we denote $\approx$) for which the three conditions are weakened to

(i) if $(\mathtt{s}, \mathtt{P}) \approx (\mathtt{t}, \mathtt{Q})$ and $(\mathtt{t}, \mathtt{Q}) \not\rightarrow$

then there is a state $(\mathtt{t}, \mathtt{R})$ such that $(\mathtt{s}, \mathtt{P}) \stackrel{*}{\rightarrow} (\mathtt{t}, \mathtt{R})$ and $(\mathtt{t}, \mathtt{R}) \not\rightarrow$.

(ii) $(\approx; \rightarrow) \subseteq (\stackrel{*}{\rightarrow}; \approx)$,

(iii) $\approx$ is symmetric

where $\stackrel{*}{\rightarrow}$ is the reflexive transitive closure of $\rightarrow$. (This is very similar to the confluence condition, used in the proof of the Church-Rosser property of a set of algebraic transformations). Weak bisimilarity is defined from weak bisimulation in the same way as for strong bisimilarity; and a similar check has to be made for the congruence property.

Now we can prove idempotence of disjunction. Let $\approx$ relate every $(\mathtt{s}, \mathtt{P})$ just to itself and to $(\mathtt{s}, \mathtt{P} \vee \mathtt{P})$ and vice versa. In the operational semantics $(\mathtt{s}, \mathtt{P} \vee \mathtt{P})$ is related by $\rightarrow$ only to $(\mathtt{s}, \mathtt{P})$; fortunately $(\mathtt{s}, \mathtt{P}) \stackrel{*}{\rightarrow} (\mathtt{s}, \mathtt{P})$ since $\stackrel{*}{\rightarrow}$ is reflexive. Conversely, whenever $(\mathtt{s}, \mathtt{P}) \rightarrow (\mathtt{s}', \mathtt{P}')$ then $(\mathtt{s}, \mathtt{P} \vee \mathtt{P}) \rightarrow (\mathtt{s}, \mathtt{P}) \rightarrow (\mathtt{s}', \mathtt{P}')$, so equality is restored after two steps. This therefore is the weak bisimulation that shows the bisimilarity

$$(\mathtt{P} \vee \mathtt{P}) \approx \mathtt{P}.$$

Unfortunately, we still cannot prove the associative law for disjunction. The three simple states $(\mathtt{s}, \mathtt{x} := 1), (\mathtt{s}, \mathtt{x} := 2)$ and $(\mathtt{s}, \mathtt{x} := 3)$ end in three distinct final states; and by condition (i), none of them is bisimilar to any other. The state $(\mathtt{s}, (\mathtt{x} := 1 \vee \mathtt{x} := 2))$ and $(\mathtt{s}, (\mathtt{x} := 2 \vee \mathtt{x} := 3))$ are also distinct, because each of them has a transition to a state (i.e., $(\mathtt{s}, \mathtt{x} := 1)$ and $(\mathtt{s}, \mathtt{x} := 3)$ respectively) which cannot be reached in any number of steps by the other. For the same reason $(\mathtt{s}, (\mathtt{x} := 1 \vee \mathtt{x} := 2) \vee \mathtt{x} := 3)$ is necessarily distinct from $(\mathtt{s}, \mathtt{x} := 1 \vee (\mathtt{x} := 2 \vee \mathtt{x} := 3))$. The associative law for disjunction is thereby violated.

In fact, there is a perfectly reasonable sense in which it is possible to observe the operational distinctness between the two sides of an associative equation

$$(\mathtt{P} \vee \mathtt{Q}) \vee \mathtt{R} = \mathtt{P} \vee (\mathtt{Q} \vee \mathtt{R}).$$

Just take *two* copies of the result of executing the first step. In the case of the left hand side, this will give either of the two pairs.

$$(\mathtt{R}, \mathtt{R}) \text{ and } ((\mathtt{P} \vee \mathtt{Q}), (\mathtt{P} \vee \mathtt{Q})).$$

Now each copy is run independently. There are now five possibilities for the subsequent behaviours of the pair:

$$(\mathtt{R}, \mathtt{R}), (\mathtt{P}, \mathtt{P}), (\mathtt{P}, \mathtt{Q}), (\mathtt{Q}, \mathtt{P}) \text{ and } (\mathtt{Q}, \mathtt{Q}).$$

But if the same procedure is applied to the right hand side of the associative equation, the five possibilities are

$$(\mathtt{P}, \mathtt{P}), (\mathtt{Q}, \mathtt{Q}), (\mathtt{Q}, \mathtt{R}), (\mathtt{R}, \mathtt{Q}), \text{ and } (\mathtt{R}, \mathtt{R}).$$

The third and fourth possibilities in each case are sufficient to distinguish the two different bracketings. If copying the state is allowed after any step of the operational semantics, it is possible to make a pair of observations of the two copied programs that may tell which way the original program was bracketed. Such an observation would invalidate the associative law.

So even weak bisimilarity is not weak enough to give one of the laws that we quite reasonably require. Unfortunately, it is also too weak for our purposes: it gives rise to algebraic laws that we definitely do *not* want. For example, consider the program

$$\mu\mathtt{X} :: (\mathbb{I} \vee \mathtt{X}).$$

This could lead to an infinite computation (if the second disjunct $\mathtt{X}$ is always selected); or it could terminate (if the first disjunct $\mathbb{I}$ is ever selected, even only once). Weak bisimilarity ignores the non-terminating case, and equates

the program to $\mathbb{I}$. However, in our theory it is equated to **true**, the weakest fixed point of the equation

$$\mathtt{X} = (\mathbb{I} \vee \mathtt{X}).$$

Our weaker interpretation **true** permits a wider range of implementations: for example, the "wrong" choice may be infinitely often selected at run time; indeed, the "right" choice can even be eliminated at compile time! For a theory based on bisimilarity, neither of these implementations is allowed. A non-deterministic construction $(\mathtt{P} \vee \mathtt{Q})$ is expected to be implemented *fairly*: in any infinite sequence of choices, each alternative must be chosen infinitely often. Weak bisimilarity is a very neat way of imposing this obligation, which at one time was thought essential to the successful use of non-determinism.

Unfortunately, the requirement of fairness violates the basic principle of monotonicity, on which so much of engineering design depends. The program $(\mathtt{X} \vee \mathbb{I})$ is necessarily less deterministic than X, so $(\mu\mathtt{X} :: \mathtt{X} \vee \mathbb{I})$ should (by monotonicity) be less deterministic than $(\mu\mathtt{X} :: \mathtt{X})$, which is the least deterministic program of all. However, weak bisimulation identifies it with the completely deterministic program $\mathbb{I}$. It would therefore be unwise to base a calculus of design on weak bisimulation.

But that was never the intention; bisimulation was originally designed by Milner as the strongest equivalence that can reasonably be postulated between programs, and one that could be efficiently tested by computer, without any consideration of any possible meaning of the texts being manipulated. It was used primarily to explore the algebra of communication and concurrency. It was not designed for application to a non-deterministic sequential programming language; and the problems discussed in this section suggest it would be a mistake to do so.

A great many alternative definitions of program equivalence based on operational semantics have been explored by subsequent research. One of them is described in the next section. In the final section of this chapter there is yet another solution: to derive an observational semantics from the operational. The algebraic laws can then be proved from the observations, as described in Chapter 3.

4.3 From operations to algebra

In this section we will define a concept of simulation which succeeds in reconstructing the algebraic semantics of our chosen language on the basis of its operational semantics. First we define $\stackrel{*}{\rightarrow}$ as the reflexive transitive closure of $\rightarrow$:

$$\begin{array}{rcl} (\mathtt{s},\mathtt{P}) \rightarrow^{\circ} (\mathtt{t},\mathtt{Q}) & \text{iff} & \mathtt{s} = \mathtt{t} \text{ and } \mathtt{P} = \mathtt{Q} \\ (\mathtt{s},\mathtt{P}) \rightarrow^{n+1} (\mathtt{u},\mathtt{R}) & \text{iff} & \exists \mathtt{t},\mathtt{Q} :: (\mathtt{s},\mathtt{P}) \rightarrow (\mathtt{t},\mathtt{Q}) \text{ and } (\mathtt{t},\mathtt{Q}) \rightarrow^{n} (\mathtt{u},\mathtt{R}) \end{array}$$

$$(\mathtt{s},\mathtt{P}) \rightarrow^* (\mathtt{t},\mathtt{Q}) \quad \text{iff} \quad \exists n :: (\mathtt{s},\mathtt{P}) \rightarrow^n (\mathtt{t},\mathtt{Q}).$$

Secondly, we define the concept of *divergence*, being a state that can lead to an infinite execution:

$$(\mathtt{s},\mathtt{P}) \uparrow \;\; =_{df} \;\; \forall n :: \exists \mathtt{t},\mathtt{Q} :: (\mathtt{s},\mathtt{P}) \rightarrow^n (\mathtt{t},\mathtt{Q}).$$

Thirdly, we define an ordering relation $\sqsubseteq$ between states. One state is *better* in this ordering than another if any result given by the better state is also possibly given by the worse; and furthermore, a state that can fail to terminate is worse than any other:

$$\begin{aligned}(\mathtt{s},\mathtt{P}) \sqsubseteq (\mathtt{t},\mathtt{Q}) \equiv_{df} \quad & (\mathtt{s},\mathtt{P}) \uparrow \text{ or } \neg(\mathtt{t},\mathtt{Q}) \uparrow \\ & \text{and } (\forall \mathtt{u} :: (\mathtt{t},\mathtt{Q}) \stackrel{*}{\rightarrow} (\mathtt{u},\mathbb{I}) \Rightarrow (\mathtt{s},\mathtt{P}) \stackrel{*}{\rightarrow} (\mathtt{u},\mathbb{I})).\end{aligned}$$

One program is better than another if it is better in all data states

$$\mathtt{P} \sqsubseteq \mathtt{Q} \quad \text{iff} \quad \forall \mathtt{s} :: (\mathtt{s},\mathtt{P}) \sqsubseteq (\mathtt{s},\mathtt{Q}).$$

The $\sqsubseteq$ relation is often called *refinement* or *simulation* of the worse program by the better.

It is easy to see that the syntactically defined $\sqsubseteq$ relation is transitive and reflexive, i.e., a preorder. As a result the relation

$$\mathtt{P} \sim \mathtt{Q} =_{df} \quad \mathtt{P} \sqsubseteq \mathtt{Q} \text{ and } \mathtt{Q} \sqsubseteq \mathtt{P}$$

is an equivalence. Since it is also a congruence, it can be used in exactly the same way as proposed for the bisimilarity relation in the previous section: one can thereby derive a full collection of algebraic laws for the programming language from its operational semantics. For example, associativity of $\vee$ follows from the two lemmas

$$\begin{aligned}&(s, P \vee (Q \vee R)) \stackrel{*}{\rightarrow} (t,\mathbb{I}) \\ &\qquad\qquad \text{iff } (s,P) \stackrel{*}{\rightarrow} (t,\mathbb{I}) \text{ or } (s,Q) \stackrel{*}{\rightarrow} (t,\mathbb{I}) \text{ or } (s,R) \stackrel{*}{\rightarrow} (t,\mathbb{I}) \\ &(s, P \vee (Q \vee R)) \uparrow \text{ iff } (s,P) \uparrow \text{ or } (s,Q) \uparrow \text{ or } (s,R) \uparrow .\end{aligned}$$

The same holds for the other bracketing as well.

But this is rather a laborious way of proving the rather large collection of laws. Each of these laws of Chapter 3 is a theorem of the form $[P \equiv Q]$. They can all be automatically lifted to simulation laws by the single theorem:

$$\mathtt{P} \sim \mathtt{Q} \quad \text{if} \quad [P \equiv Q].$$

In fact, the above theorem can be strengthened to an equivalence, so that the laws proved by simulation are exactly those of the algebraic semantics. The algebraic semantics is isomorphic to the operational, when abstracted by this particular notion of simulation.

4.4 From operations to observations

An operational semantics is in essence an inductive definition of all possible sequences of states that can arise from any execution of any program expressed in the notations of the language. This can be turned directly into an isomorphic observational semantics by just assuming that the whole execution sequence generated by each program can be observed. We will use the free variable e to stand for such an execution sequence. Of course, the level of abstraction is exactly the same as that of the operational semantics. To hide the excessive degree of detail, we need to define a predicate which relates each execution sequence onto an observation just of its initial and final states. Using this relation as a coordinate transformation, the definitions of the observational semantics given in Chapter 2 can be proved as theorems. The proofs are wholly based on the definition of the operational semantics, thereby completing the cyclic proof of mutual consistency of all three theories of programming.

An execution is formally defined as an empty, finite or infinite sequence of states in which every adjacent pair of states is related by the operational transition $\rightarrow$

$$E = \{e \mid \forall i : 0 < i < \sharp e : e_{i-1} \rightarrow e_i\},$$

where $\sharp$ gives the length of a finite sequence, or ∞ for an infinite one. The execution sequences generated by an initial state $(\mathtt{s}, \mathtt{P})$ are those that begin with this state

$$E(\mathtt{s}, \mathtt{P}) = \{e \mid e \in E \ \&\ (e = <> \vee e_0 = (\mathtt{s}, \mathtt{P}))\}$$

The observations of a program $\mathtt{P}$ are those in which $\mathtt{P}$ is the initial state of the stored program

$$E(\mathtt{P}) = \bigcup_{\mathtt{s}} E(\mathtt{s}, \mathtt{P}).$$

The function E defines an observational semantics of each program text $\mathtt{P}$. The definition can be rewritten into an equivalent recursive form

$$E(\mathtt{s}, \mathtt{P}) = (\mathtt{s}, \mathtt{P})^\wedge\{e \mid \exists \mathtt{t}, \mathtt{Q} : (\mathtt{s}, \mathtt{P}) \rightarrow (\mathtt{t}, \mathtt{Q}) \ \&\ e \in E(\mathtt{t}, \mathtt{Q})\},$$

where $x^\wedge X =_{df} \{<>\} \cup \{< x > e \mid e \in X\}$. As in Chapter 2, the recursion here is understood to define the weakest (i.e. largest) fixed point, which will include all the infinite sequences as well.

But the definition lacks the important denotational property, requiring that the semantics of each combinator is defined in terms of the semantics of its components. Fortunately, the problem is solved by proof of the following theorems, which closely follow the structure of the operational semantics

$$
\begin{array}{lcll}
E(\mathtt{s}, \mathbb{I}) & = & \{< (\mathtt{s}, \mathbb{I}) >, <>\} & \\
E(\mathtt{s}, \mathtt{v} := \mathtt{f}) & = & (\mathtt{s}, \mathtt{v} := \mathtt{f})^\wedge\{<>, (\mathtt{v} := (s; \mathtt{f}), \mathbb{I})\} & \\
E(\mathtt{s}, \mathtt{P} \vee \mathtt{Q}) & = & E(\mathtt{s}, \mathtt{P}) \cup E(\mathtt{s}, \mathtt{Q}) & \\
E(\mathtt{s}, \mathtt{P} \lhd \mathtt{b} \rhd \mathtt{Q}) & = & (\mathtt{s}, \mathtt{P} \lhd \mathtt{b} \rhd \mathtt{Q})^\wedge E(\mathtt{s}, \mathtt{P}) & \text{if } s; \mathtt{b} \\
 & = & (\mathtt{s}, \mathtt{P} \lhd \mathtt{b} \rhd \mathtt{Q})^\wedge E(\mathtt{s}, \mathtt{Q}) & \text{if } \neg s; \mathtt{b} \\
E(\mathtt{s}, \mathtt{P}; \mathtt{Q}) & = & \mathrm{add}Q(E(\mathtt{s}, \mathtt{P})) & \\
 & & \cup\{ef \mid e \in \mathrm{add}Q(E(\mathtt{s}, \mathtt{P})) \ \&\ f \in E(\mathtt{Q}) & \\
 & & \&\ \exists \mathtt{t} : e \text{ ends in } (\mathtt{t}, \mathbb{I}; \mathtt{Q}) \text{ and } f \text{ begins with } (\mathtt{t}, \mathtt{Q})\} &
\end{array}
$$

where $\mathrm{add}Q(e)_i = (\mathtt{s}, \mathtt{P}; \mathtt{Q})$ whenever $e_i = (\mathtt{s}, \mathtt{P})$,

$E(\mathtt{s}, \mu\mathtt{X} :: \mathtt{F}.\mathtt{X}) = (\mathtt{s}, \mu\mathtt{X} :: \mathtt{F}.\mathtt{X})^\wedge E(\mathtt{s}, \mathtt{F}.(\mu\mathtt{X} :: \mathtt{F}.\mathtt{X}))$.

If desired, these equations could be presented as the definition of the operational semantics of the language. Although they do not mention the step relation $\rightarrow$, they define exactly the same execution sequences. In fact, the step relation may be defined afterwards as that which holds between the first and second members of any execution sequence

$$(\mathtt{s}, \mathtt{P}) \rightarrow (\mathtt{t}, \mathtt{Q}) \ \text{ iff } \ \exists e :: e \in E(\mathtt{s}, \mathtt{P}) \ \&\ e_2 = (\mathtt{t}, \mathtt{Q}).$$

The proof of this theorem establishes an isomorphism between the traditional presentation of the operational semantics, given at the beginning of this chapter, and its denotational presentation, given by the above definition of E.

Our original presentation of a denotational semantics in Chapter 2 mapped each program text onto a predicate describing its observations. The definition of E maps each program text onto a *set* of observations, i.e., its execution sequences. Such a set can easily be timed into a predicate $e \in E(\mathtt{P})$, which uses the free variable e to stand for an observed value of the execution sequence generated by $\mathtt{P}$. Similarly, any predicate P, with free variables ranging over known sets, can be turned into a set of observations by simple comprehension, for example $\{e \mid P\}$. The distinction between these two presentations as sets and as predicates is entirely ignorable. So the equations shown above could equally well be regarded as an *observational* semantics of the programming language, in exactly the style of Chapter 2. In future, let us use the italic $P(e)$ to stand for the predicate $e \in E(\mathtt{P})$.

But of course, the level of abstraction of this new observational semantics is identical to that of the operational semantics. To raise the level, we use a standard technique of data refinement. This requires us to specify which of the more concrete observations we are interested in, and which parts of them we want to observe. These decisions can be expressed as a predicate containing *both* sets of free variables, ones standing for the concrete observations, as well as ones standing for the abstract.

Our original abstract observations were selected on the assumption that

we want to observe only a very small part of the information contained in each execution, namely the initial and final data states. Furthermore, we choose never to observe a finite non-empty execution that has not yet terminated. So we are interested only in minimal or maximal executions – those which cannot be reduced or extended

$$e = <> \vee\ e \text{ ends in } \mathbb{I} \ \vee\ \ e \text{ is infinite.}$$

The distinction between empty, terminated and infinite execution sequences is captured in the Boolean variables ok and ok':

$$ok = (e \neq <>) \text{ and } \ ok \Rightarrow (ok' = (e \text{ ends in } \mathbb{I})).$$

For non-empty sequences, we can observe the initial data state v

$$ok \Rightarrow \exists \mathtt{s}, \mathtt{P} :: e_0 = (\mathtt{s}, \mathtt{P}) \ \&\ \mathit{init}(\mathtt{s})$$

where *init* is a function that maps the text `v := k` to the predicate $v = k$. Similarly, for terminating executions, we can observe the final states

$$ok' \Rightarrow \exists \mathtt{s} :: e \text{ ends in } (\mathtt{s}, \mathbb{I}) \ \&\ \mathit{final}(\mathtt{s})$$

where *final* is a function that maps the text `v := k` to the predicate $v' = k$.

Let ABS be the conjunction of the predicates displayed above. It has free variables ok, ok', v, v' and e. It describes all the ways in which an execution e can give rise to particular values of the other more abstract variables. So the abstract observations obtainable from any possible execution of `P` are described by the predicate

$$abs(P(e)) =_{df} \exists e : ABS \wedge P(e).$$

The function *abs* maps predicates describing execution sequences (denoted by the free variable e) to predicates describing observations of the initial and final states of a subset of those sequences (denoted by the free variables ok, ok', v, v'). This latter class of predicates is exactly the one used to give the original denotational semantics of the programming language in Chapter 2. We wish to claim that the semantics defined above by the functions *abs* is the same as that of Chapter 2. The claim is substantiated by showing that *abs* is an isomorphism between the relevant subsets of the two classes of predicate, namely those predicates that can be expressed as programs.

The predicate ABS is often called a linking invariant, because it relates observations of the same system at two levels of abstraction. It is used to define the function *abs*, which translates a low level predicate, whose free variables denote details of an implementation, to a predicate describing the same system at a higher level of abstraction. The predicate $abs(\mathtt{P})$ is the strongest specification expressible at this higher level which is met by any concrete implementation described by P. But in a top-down design, we

would prefer to translate in the opposite direction; given a specification S with free variables v, v', ok, ok', what is the weakest description of the low-level observations that will satisfy the specification? The answer is given by the definition

$$abs^{-1}(S) = (\forall v, v', ok, ok' : \text{ABS} \Rightarrow S).$$

The transformations in each direction are linked by the Galois connection

$$[abs(P) \Rightarrow S] \quad \text{iff} \quad [P \Rightarrow abs^{-1}(S)].$$

From this it follows that abs is monotonic with respect to implication, and

$$P \Rightarrow abs^{-1}(abs(P)) \text{ and } abs(abs^{-1}(S)) \Rightarrow S.$$

Even more important are the equations

$$abs(P) = abs(abs^{-1}(abs(P))) \quad \text{and} \quad abs^{-1}(S) = abs^{-1}(abs(abs^{-1}(S))).$$

This means that $abs \circ abs^{-1}$ is the identity function over the range of abs, and similarly for $abs^{-1} \circ abs$. In our case, this range includes *all* the predicates expressible in the programming language. So abs is a bijection between predicates with alphabet $\{e\}$ describing execution sequences and predicates with alphabet $\{v, v', ok, ok'\}$ describing initial and final states. In both cases the predicates both arise from program texts.

Our last task is to show that abs is a homomorphism in the usual algebraic sense, that it is respected by all the combinators of the programming language. The required theorems look very like a denotational definition of the abs function.

$$\begin{aligned}
&abs(\mathbf{true}) = \mathbf{true} \\
&abs(\mathbb{I}) = \neg ok \vee (ok' \wedge v' = v) \\
&abs(v := f) = \neg ok \vee (ok' \wedge v' = f) \\
&abs(P \vee Q) = abs(P) \vee abs(Q) \\
&abs(P \lhd b \rhd Q) = (abs(P) \wedge b) \vee (abs(Q) \wedge \neg b) \\
&abs(P; Q) = abs(P); abs(Q) \\
\\
&abs(\mu X :: F.X) = \mu Y :: F'.Y \\
&\qquad \text{where } F'.Y = abs(\mathtt{F}(abs^{-1}Y)).
\end{aligned}$$

In the last clause, the functions abs and abs^{-1} are needed to translate F from a function over predicates with e as their only free variable to a function F' over predicates with the more abstract alphabet.

The form of these definitions is exactly the same as those of the original observational semantics of the language in Section **??**. Indeed, on omission of

occurrences of the function abs and abs^{-1}, the two definitions are the same. More formally, the theorems show that abs is an isomorphism between two different interpretations of the notations of the same programming language: one as a description of execution sequences derived from the operational semantics, and one as a description of a relationship between initial and final values of variables v, ok.

This completes the task of unifying observational, algebraic and operational theories of the same programming language. In each case the basic definitions or axioms of each theory have been derived as theorems in its preceding theory, in a cyclic fashion. It is therefore a matter of choice which theory is presented first. My preference is to start with the most abstract, because this gives the most help in specification, design and development of programs. Furthermore, derivation of algebraic laws is quite straightforward, using standard mathematical proof methods. Finally, proof of the properties of the operational semantics can take advantage of previously established theorems. In general, a top-down approach seems easier than starting at a low level of abstraction and working upwards. But the operational semantics has considerable attraction, and is currently quite fashionable among theorists investigating the foundations of Computing Science.

Chapter 5

Conclusion

This monograph has recommended three distinct approaches to the construction of theories relevant to computing — the operational, the algebraic, and the observational. They have each an important distinctive role, which can and should be studied independently by specialists. But the full benefits of theory are obtained by a clear and consistent combination of the benefits of all three approaches. The method of consistent combination has been illustrated by application to a very simple programming language for expression of sequential algorithms with possible non-determinism. This is only a small part of the total task of clarifying the foundations of Computing Science.

We will need to build up a large collection of models and algebras, covering a wide range of computational paradigms, appropriate for implementation either in hardware or in software, either of the present day or of some possible future. But even this is not enough. What is needed is a deep understanding of the relationships between the different models and theories, and a sound judgment of the most appropriate area of application of each of them. Of particular importance are the methods by which one abstract theory may be embedded by translation or interpretation in another theory at a lower level of abstraction. In traditional mathematics, the relations between the various branches of the subject have been well understood for over a century, and the division of the subject into its branches is based on the depth of this understanding. When the mathematics of computation is equally well understood, it is very unlikely that its branches will have the same labels that they have today. Quoting from Andreski [?], "the contours of truth never coincide with the frontiers between embattled parties and cliques". So we must hope that the investigations by various schools of programming theory will contribute to the understanding which leads to their own demise.

The establishment of a proper structure of branches and sub-branches is essential to the progress of science. Firstly, it is essential to the efficient education of a new generation of scientists, who will push forward the frontiers in new directions with new methods unimagined by those who taught them. Secondly, it enables individual scientists to select a narrow specialisation for intensive study in a manner which assists the work of other scientists in related branches, rather than just competing with them. It is only the small but complementary and cumulative contributions made by many thousands of scientists that has led to the achievements of the established branches of modern science. But until the framework of complementarity is well understood, it is impossible to avoid gaps and duplication, and achieve rational collaboration in place of unscientific competition and strife.

Quoting again from Andreski

> "...the reason why human understanding has been able to advance in the past, and may do so in the future, is that true insights are cumulative and retain their value regardless of what happens to their discoverers; while fads and stunts may bring an immediate profit to the impresarios, but lead nowhere in the long run, cancel each other out, and are dropped as soon as their promoters are no longer there (or have lost the power) to direct the show. Anyway let us not despair."