A Theory of Programming: Denotational, Algebraic and Operational Semantics

C.A.R. Hoare

March 31, 1993

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 a simple theory of sequential non-deterministic programming in each of these three styles; by deriving each presentation from its predecessor, mutual consistency is assured.

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 [1]. 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 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 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 [2] to define the Calculus of Concurrent Systems (CCS); the algebraic style took precedence in the definition [3] of the Algebra of Concurrent Processes (ACP), whereas the denotational style lies at the basis of the mathematical theory [4] of Communicating Sequential Processes (CSP). Many of the detailed differences between these three process theories originate from their different styles of presentation. To obtain a synthesis based on a full understanding, it is helpful to concentrate on a single theory, and present it fully in all three styles; there is the additional hope that their complementary benefits can be exploited in practice. It is the goal of this paper to explore the relevant techniques in the case of a simple sequential programming language, thereby avoiding any controversy that surrounds the treatment of process algebra.

Not a single idea in this paper is original. The concept of denotational semantics is due to Strachey and Scott [5], and the particular choice of ordering of non-deterministic programs is due to Smyth [6]. The embedding of programs as predicates is due to Hehner [7]. The language is essentially the same as that of Dijkstra [1]. The denotational theory is taken from Tarski's calculus of relations [8]. The treatment of recursion in specifications is given by Tarski's

fixed point theorem [10] and for programs by Plotkin [9]. The algebraic treatment of the language has already been fully covered in [11]. Even the idea of consistent and complementary definitions of programming languages goes back at least to [12].

The only originality in the paper 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 operators of the language in terms of the second-order predicate calculus. These can be individually formulated and understood in isolation from each other. The algebraic laws can then be derived one by one, without danger of complex or 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 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. The phenomena of deadlock (no transitions) and divergence (an infinite sequence of transitions) are analysed, and shown to relate correctly to their algebraic interpretation.

As always in such smooth developments, the simplicity is an artefact of many laborious and less successful iterations, mercifully concealed from the reader. Another reason for the simplicity and modularity of the proofs described above is that they follow the natural progression from abstract description to concrete implementation. It is possible (and indeed more usual) to work in the other direction, starting with an operational presentation. A concept of bisimulation is then selected, permitting the proof of algebraic laws; and a model can then be derived by a standard initial algebra construction. A derivation in both directions establishes completeness as well as consistency of the three presentations. But that is the subject of another paper.

2 Observations and Predicates

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:

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

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 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 II (pronounced "skip") and defined

$$\text{II} \ =_{df} \ x' = x \wedge y' = y \wedge \ldots \wedge z' = z.$$

In words, an observation of the final state of II is the same as that of its initial state.

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.

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

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

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 which describes 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 available, because clarity of specification is the only protection we have against subsequent misunderstandings of the client's requirements, which can often lead to 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 settling 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 math-

ematical 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.

This means that we must live with the danger of proposing and accepting an unimplementable predicate as a specification. 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 correctly 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.

3 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

$$\Pi \quad =_{df} \quad x := x.$$

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

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

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 these 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.

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 parameters 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

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

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 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 exam-

ple, it is associative and has II 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.

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

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 its initial values. So if $P(x)$ is any predicate mentioning x, P is true of the final value of x 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 generalises to any expression:

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

For example

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

This convention permits a rightward distribution law for conditionals:

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

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 does not say 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 definition 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 share 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*, 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 some 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 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:

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

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 is the presumably simpler task of finding two programs P and Q which implement the two designs, i.e.,

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

Now all that remains is to 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. Note

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.

4 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 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 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 [10] 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 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

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

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

$$X = X$$

and 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

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

which starts with an infinite loop. 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. That 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.

In the case of computer programs, the anomaly 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, which are never allowed to appear in the text of the program:

st, which becomes true when the program has been started, and is false beforehand.

st', which becomes true when the program has stopped, and remains forever false in case of non-termination (and *a fortiori,* if program is never started).

While st' 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, while st is false, even the initial values are unobservable. These considerations underlie the validity of the desired zero laws.

We still maintain the convention that no observation will be made of the variables while the program is running, so we never observe that st is true and st' is false, except in the case of non-termination. This is the essential abstraction from details of execution time, which permits a separation of concerns between correctness and efficiency in reasoning about program behaviour. It also permits programs written for the IBM 704 in 1960 to run correctly on supercomputers for the present day, in spite of a vast difference in speed.

The variables st and st' 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 then be written

$$(st \wedge P) \Rightarrow (st' \wedge Q)$$

or in words "If the program components start in a state satisfying P, it will stop in a state satisfying Q."

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 designed behaviour of the subsequent 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, and the corresponding obligations undertaken by their colleagues. So it is worth while to introduce a special notation

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

This is the primitive notation used by Morgan in [14]. The clear distinction of precondition P from postcondition Q is a distinctive feature of VDM [13].

Another advantage of explicit mention of starting and stopping 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

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

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 has not started ($st = false$) 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 has terminated ($st' = true$), and the value of x' is e, and the final values of all the other variables are the same as their initial values.

Fortunately, this is the only new definition that is needed; the definition of conditionals, recursion, and sequential composition remain unchanged, and all laws (except those involving assignment) remain valid. In fact, the laws involving assignment also remain valid, provided that their variables range not over arbitrary predicates, but only over predicates expressed in programming notations. For this restricted class of predicates (hereafter called programs), we will have to prove the unit laws

$$\Pi; P \;=\; P \;=\; P; \Pi, \qquad \text{for all programs } P$$

as well as the new zero laws

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

In compensation, the zero laws give an assurance that no programs can be equal to the unimplementable predicate *false*, which does not satisfy them.

It is quite easy to check that the zero and unit laws are valid for the simple case of programs that are assignments, even when these are interpreted according to the new definition. This proof can be extended by structural induction to more complex kinds of program. A simple example of a lemma needed in this proof is:

If P and Q satisfy the laws

$$P; \mathbf{true} \;=\; \mathbf{true} \;=\; Q; \mathbf{true}$$

then so do $(P; Q)$ and $(P \lhd b \rhd Q)$.

The proof of this theorem is also quite simple:

$$\begin{aligned}(P;Q);\mathbf{true} &= P;(Q;\mathbf{true}) = P;\mathbf{true} = \mathbf{true}\\ (P \lhd b \rhd Q);\mathbf{true} &= (P;\mathbf{true}) \lhd b \rhd (Q;\mathbf{true})\\ &= \mathbf{true} \lhd b \rhd \mathbf{true} = \mathbf{true}.\end{aligned}$$

Unfortunately the required additional theorems for the left zero law and for the recursion operator μ are much more difficult to prove. The relevant mathematics is worked out in the next chapter.

5 The Algebra of Programs.

In this section 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, these have been laws which are 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

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

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[15]. 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 used than the foundationary definitions of analysis on which it is based.

<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.

Table 1 Syntax.

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*. This is an expression that 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, non-determinism, non-termination, and recursion. For each kind of normal form, we establish the zero and unit laws

$$\begin{array}{lcccl} P;\mathbf{true} & = & \mathbf{true} & = & \mathbf{true};P \\ P;\mathrm{II} & = & P & = & \mathrm{II};P \end{array}$$

for all P expressed in that normal form.

5.1 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:

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

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

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

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:

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

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.

5.2 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

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

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

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

The other operators are eliminated by distribution laws

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

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

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 5.1. Thus the occurrences of ; and $\lhd b \rhd$ in the right

hand sides of the laws given above are also eliminable. Similarly, the distribution law for composition gives an easy proof of the zero and unit laws from the same laws for their component assignments; for example

$$\begin{aligned}(\bigvee A);\ \bigvee\{\mathbf{true}\} &= \bigvee\{P:\ P \in A:\ P;\mathbf{true}\} \\ &= \bigvee\{\mathbf{true}\}\end{aligned}$$

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.

5.3 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

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

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

$$\mathit{true} \vee \Pi.$$

The other operators can be eliminated by the laws

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

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

$$
\begin{aligned}
& [b; c \Rightarrow b] \\
\text{and } & [b; Q \equiv b].
\end{aligned}
$$

The law for testing equivalence is

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

5.4 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 m and n be non-negative integer variables in

$$
\textbf{while } m > 0 \;\wedge\; n > 0 \textbf{ do } (m := m - 1 \;\vee\; n := n - 1).
$$

Informally, the effect of this is described

$$
\begin{aligned}
& (m, n := 0, n) \vee (m, n := 0, n - 1) \vee \ldots \vee (m, n := 0, 1) \\
& \vee\; (m, n := m, 0) \vee (m, n := m - 1, 0) \vee \ldots \vee (m, n := 1, 0).
\end{aligned}
$$

But there is no *finite* set of assignments whose disjunction can replace the informal elipses (...) shown above, because the length of the disjunction depends on the initial values of m and 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 5.3; 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

$$(\bigwedge S)$$

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

$$\begin{array}{lcl}
i \ominus k & =_{df} & i \lhd k \geq i \rhd i - k \\
S_0 & = & m + n \geq 0 \\
S_1 & = & m + n \geq 1 \ \vee \ (m,n := m,0) \\
 & & \qquad \vee \ (m,n := 0,n) \\
S_2 & = & m + n \geq 2 \\
 & & \vee \ (m,n := m,0) \ \vee \ (m,n := m \ominus 1,0) \\
 & & \vee \ (m,n := 0,n) \ \vee \ (m,n := 0,n \ominus 1) \\
S_i & = & m + n \geq i \\
 & & \vee \ (\bigvee_{k<i} m,n := m \ominus k,0) \\
 & & \vee \ (\bigvee_{k<i} m,n := 0,n \ominus k).
\end{array}$$

Each S_i is in finite normal form. It describes exactly the behaviour of the program when $(m+n)$ does not initially exceed i, 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 sum of the initial values of m and n. It follows that the predicate describing the behaviour of the whole program is equal to the required infinite disjunction.

$$\bigwedge S \;=\; \bigvee_k(m,n \;:=\; m \ominus k, 0) \;\vee\; \bigvee_k(m,n \;:=\; 0, n \ominus k).$$

The laws for recursion will provide a general method for calculating the successive approximations S_i which describe the behaviour of any particular loop.

The main justification of the descending chain condition for S is that it permits distribution by all the operators of the language through intersection:

$$\begin{aligned}
(\bigwedge S) \wedge P &= \bigwedge_i (S_i \vee P) \\
(\bigwedge S) \lhd b \rhd P &= \bigwedge_i (S_i \lhd b \rhd P) \\
P \lhd b \rhd (\bigwedge S) &= \bigwedge_i (P \lhd b \rhd S_i) \\
(\bigwedge S); P &= \bigwedge_i (S_i; P) \\
P; (\bigwedge S) &= \bigwedge_i (P; S_i).
\end{aligned}$$

The last two laws require that P also should be expressed in normal form. Operators that distribute through intersections of descending chains are called *continuous*. Every combination of continuous operators is also continuous in *all* its arguments.

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

$$\bigwedge_k (\bigwedge_l S_{kl}) \;=\; \bigwedge_i S_{ii}.$$

Together with continuity, this gives the required elimination laws for the three operators of the language

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

The occurrence of the operators on the right hand side of these laws can eliminated by the laws of 5.3, 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. A recursive program is written

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

where $F.X$ contains X as its only free 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 (possibly infinite) 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 the chain is the least fixed point of F

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

For each n

$$F^n.\mathbf{true}$$

can be expressed in normal form, say

$$\bigwedge_m S_{nm}.$$

By diagonalisation, the right hand side of this equation can be expressed in normal form. First, all innermost recursions are replaced in this way and then the replacement can be repeated on the new innermost recursions, until all recursions have disappeared. The whole program is now in normal form.

The only remaining worry is that it is an infinite normal form, so the reduction can never be completed. Nevertheless, every one of the finite approximations can in principle be calculated in a finite time. From a mathematical point of view, the principle is good enough. No-one would wish to perform the calculations in practice, even with the aid of a computer.

6 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 a variety of real computers. As before, we will 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'. Each step is drawn from a very limited repertoire, within the capabilities of a simple machine. A definition of the set of all possible steps simultaneously defines the machine and all possible execution sequences that it can give rise to.

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 program that remains to be executed. When this is II, there is no more program to be executed; the state (t, II) is last state of any execution 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. We also introduce an improper machine state $\perp$, representing a machine that cannot be used (perhaps because some previous program never terminates); it is identified with program **true**. If s is an initial 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. It is therefore a description of all the possible final data values of any execution of P started in s. If t is any other data state

$$[(t; \mathrm{II}) \Rightarrow (s; P)]$$

means that the final state (t, II) is a permitted result of execution of P. Furthermore,

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

means that every result 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 is reached, that will be correct too.

The symbol $\rightarrow$ traditionally denotes the execution step relation. We define it by giving the necessary and sufficient condition for its correctness:

$$(s, P) \rightarrow (t, Q) \quad =_{df} \quad [(t; Q) \Rightarrow (s; P)].$$

The following theorems are now trivial

$$(s, x := e) \quad \rightarrow \quad ((x := (s; e)), \mathrm{II})$$

The effect of an assignment $x := e$ is to end in a final state, in which x is assigned a constant value $(s; e)$ i.e., the result of evaluating e with all variables in it replaced by their initial values.

$$(s, (\mathrm{II}; Q)) \quad \rightarrow \quad (s, Q)$$

A II in front of a program Q is immediately discarded.

$$(s, P; R) \quad \rightarrow \quad (t, Q; R), \qquad \text{whenever } (s, P) \rightarrow (t, Q)$$

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

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

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.

$$(s, P \lhd b \rhd Q) \quad \rightarrow \quad (s, P) \qquad \text{whenever } s; b$$
$$(s, P \lhd b \rhd Q) \quad \rightarrow \quad (s, Q) \qquad \text{whenever } s; \neg b$$

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.

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

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

$$(s, \mathbf{true}) \quad \rightarrow \quad (t, Q) \qquad \text{for any } t, Q$$
$$(\bot, P) \quad \rightarrow \quad (t, Q) \qquad \text{for any } t, Q.$$

For the undefined state $\bot$ and the undefined program **true**, any transition is possible.

There is one law that is deliberately missing from the above list, namely the vacuous law

$$(s, P) \quad \rightarrow \quad (s, P).$$

This law would permit an implementation at any time to make an infinite sequence of steps, each of which leaves the machine state unchanged. Traditionally, in the definition of operational semantics, the possibility of infinite computations is ignored.

Conclusion

This paper 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 pursued 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 judgement 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. The investigations by various schools, now labelled as CSP, CCS, ACP, Petri Nets, etc., will have contributed to the understanding which leads to their own demise.

The establishment of a proper structure of branches and subbranches 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 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.

References

[1] E.W. Dijkstra, "A Discipline of Programming", Prentice Hall, 1976.

[2] A.R.J.G. Milner, "A Calculus of Communicating Systems", LNCS 92, Springer-Verlag, 1980.

[3] J.A. Bergstra and J.W. Klop, "Algebra of Communicating Processes with Abstraction", Theoretical Computer Science 37(1), 77-121, 1985.

[4] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe, "A Theory of Communicating Sequential Processes", Journal of ACM 31(7) 560-599, 1984.

[5] D.S. Scott and C. Strachey,"Towards a Mathematical Semantics for Computer Languages", PRG-6, Oxford 1971.

[6] M.B. Smyth, "Power domains", JCSS (16) 23-26, 1978.

[7] E.C.R. Hehner, "Predicative Programming", Comm ACM 27(2), 134-143.

[8] A. Tarski, "On the Calculus of Relations", J Symbolic Logic 6; 73-89, 1941.

[9] G.D. Plotkin, "A Structural Approach to Operational Semantics", Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.

[10] A. Tarski, "A Lattice-theoretic Fixed Point Theorem and its Applications".

[11] C.A.R. Hoare et al., "The Laws of Programming", Comm ACM 30(8), 672-87.

[12] C.A.R. Hoare, R.E.Lauer, "Consistent and complementary formal theories of the semantics of programming languages", Acta Informatica 3(2), 135-153, 1974.

[13] C.B. Jones, "Systematic Software Development using VDM", Prentice Hall International, 1986.

[14] C.C. Morgan, "Programming from Specifications", Prentice Hall International, 1990.

[15] J.A. Goguen and T. Winkler, "Introducing OBJ3", Technical Report SRI-CSL-88-9, SRI International Computer Science Lab., 1988.