Theories of Programming: Top-Down and Bottom-Up and Meeting in the Middle

C.A.R. Hoare

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD
tony.hoare@comlab.ox.ac.uk

1 Introduction

The goal of scientific research is to develop an understanding of the complexity of the world which surrounds us. There is certainly enough complexity out there to justify a wide range of specialist branches of science; and within each branch to require a wide range of investigatory styles and techniques.

For example, among the specialists in Physics, cosmologists start their speculations in the vast distances of intergalactic space, and encompass the vast time-scales of the evolution of the stars. They work methodically downward in scale, until they find an explanation of phenomena that can be observed more or less directly by the naked eye.

At the other end of the scale, particle physicists start with the primitive components of the material world, currently postulated to be quarks and gluons. They then work methodically upward in scale, to study the composition of baryons, hadrons, and leptons, clarifying the laws which govern their assembly into atoms and molecules. Eventually, they can explain the properties of materials that we touch and smell and taste in the world of every day.

In spite of the difference in scale of their starting points, and in the direction and style of their investigations, there is increasing excitement about the convergence and overlap of theories developed by cosmologists and by particle physicists. The point at which they converge is the most significant event in the whole history of the universe, the big bang with which it all started.

The same dichotomy between top-down and bottom-up styles of investigation may be found among mathematicians. For example, category theorists start at the top with a study of the most general kind of mathematical structure, as exemplified by the category of sets. They then work downward to define and classify the canonical properties that distinguish more particular example structures from each other.

Logicians on the other hand start from the bottom. They search for a minimal set of primitive concepts and notations to serve as a foundation for all of mathematics, and a minimal collection of atomic steps to define the concept of a valid proof. They then work methodically upward, to define the more familiar concepts of mathematics in terms of the primitives, and to justify the larger proof steps which mathematicians need for efficient prosecution of their work.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 1–27, 1999.

Fortunately in this case too, the top-down and the bottom-up styles of investigation both seek a common explanation of the internal structure of mathematics and clarification of the relationship between its many branches. Their ultimate goal is to extend the unreasonable power of mathematical calculation and make it more accessible to the experimental scientist and to the practicing engineer.

Computer science, like other branches of science, has as its goal the understanding of highly complex phenomena, the behaviour of computers and the software that controls them. Simple algorithms, like Euclid's method of finding the greatest common divisor, are already complex enough; a challenge on a larger scale is to understand the potential behaviour of the million-fold inter-linked operating systems of the world-wide computing network.

As in physics or in mathematics, the investigation of such a system may proceed in a choice of directions, from the top-down or from the bottom-up. In the following exposition, this dichotomy will be starkly exaggerated. In any particular scientific investigation, or on any particular engineering project, there will be a rapid alternation or mixture of the two approaches, often starting in the middle and working outward. A recommendation to this effect is made in the conclusion of the paper.

An investigation from the top-down starts with an attempt to understand the system as a whole. Since software is a man-made artifact, it is always relevant to ask first what is its purpose? Why was it built? Who is it for? What are the requirements of its users, and how are they served?

The next step is to identify the major components of the system, and ask how they are put together? How do they interact with each other? What are the protocols and conventions governing their collaboration? How are the conventions enforced, and how does their observance ensure successful achievement of the goals of the system as a whole?

A top-down theory of programming therefore starts by modelling external aspects of the behaviour of a system, such as might be observed by its user. A meaningful name is given to each observation or measurement, so that the intended behaviour of the system can be described briefly and clearly, perhaps in a user manual for a product, or perhaps even in a specification agreed with the user prior to implementation.

The set of observations is extended to include concepts needed to describe the internal interfaces between components of the system. The goal of the theory is to predict the behaviour of a complex assembly by a calculation based only on descriptions of the behaviour of its major components. The collection of formulae used for these calculations effectively constitutes a denotational semantics for the languages in which a system is specified, designed, and eventually implemented.

The programming language used for ultimate implementation is defined by simply selecting an implementable subset of the mathematically defined notations for describing program behaviour. The correctness of a program simply means that all possible observations of its behaviour under execution are included in the range defined by its specification.

The development of the theory starts from the denotational definitions and continues by formalisation and proof of theorems that express the properties of all programs written in the language. The goal is to assemble a collection of mathematical laws (usually equations and inequations) that will be useful in the top-down design of programs from their specifications, and ensure that the resulting code is correct by construction.

Investigation of a complex system from the bottom-up starts with an attempt to discover a minimum collection of primitive components from which it has been made, or in principle could have been. These are assembled into larger components by primitive combinators, selected again from a minimal set. The notations chosen to denote these primitives and combinators constitute the syntax of a simple programming language.

Since programs are intended for execution by a machine, their operational meaning is defined by enumerating the kinds of primitive step that will be taken by the machine in executing any program that is presented to it. The theory may be further developed by investigation of properties of programs that are preserved by all the possible execution steps; they are necessarily preserved throughout execution of any program.

The resulting classification of programs is presented as a set of axioms that can be used in proofs that a program enjoys the relevant property. The properties are often decidable, and the axioms can be used as a type system for the programming language, with conformity checkable by its compiler.

In favourable cases, the type system allows unique or canonical types to be inferred from an untyped program. Such inference can help in the understanding of legacy code, possibly written without any comprehensible documentation describing its structure or purpose (or worse, the original documentation often has not been kept up to date with the later changes made to the code).

The benefits of a top-down presentation of a theory are entirely complementary to those of a bottom-up presentation. The former is directly applicable to discussion and reasoning about the design of a program before it has been written, and the latter to the testing, debugging, and modification of code that has already been written. In both cases, successful application of the theory takes advantage of a collection of theorems proved for this purpose. The most useful theorems are those which take the form of algebraic laws.

The advantages of both approaches can be confidently combined, if the overlap of laws provided by both of them is sufficiently broad. The laws are a specification of the common interface where the two approaches meet in the middle. I suggest that such a convergence of laws developed by complementary approaches and applied to the same programming language should be a rigorous scientific criterion of the maturity of a theory and of a language, when deciding whether it is ready for practical implementation and widespread use.

2 Top-Down

A top-down presentation of a theory of programming starts with an account of a conceptual framework appropriate for the description of the behaviour of a running program as it may be observed by its users. For each kind of observation, an identifier is chosen serve as a variable whose exact value will be determined on each particular run of the program.

Variables whose values are measured as a result of experiment are very familiar in all branches of natural science; for example in mechanics, x is often declared to denote the displacement of a particular object from the origin along a particular axis, and $\bullet\ x$ denotes the rate of change of x. We will find that such analogies with the normal practice of scientists and engineers provide illumination and encouragement at the start as well as later in the development of theories of programming.

There are two special times at which observation of an experiment or the run of a program are especially interesting, at the very beginning and at the very end. That is why the specification language VDM introduces special superscript arrow notations: $\overleftarrow{x}$ to denote the initial value of the global program variable x, and $\overrightarrow{x}$ to denote its final value on successful termination of the program. (The Z notation uses x and x' for these purposes).

Fragments of program in different contexts will update different sets of global variables. The set of typed variables relevant to a particular program fragment is known as its *alphabet*. In the conventional sequential programming paradigm, the beginning and the end of the run of a program are the only times when it is necessary or desirable to consider the values of the global variables accessed and updated by it. We certainly want to ignore the millions of possible intermediate values calculated during its execution, and it is a goal of the theory to validate this simplification.

A full understanding of a description of program behaviour requires prior specification of its alphabet, and agreement on the way in which the value of each variable in it can be determined by experiment. To interpret the meaning of a program without knowing its alphabet is as impossible as the interpretation of a message in information theory without knowing the range of message values that might have been sent instead.

Not all the relevant parameters of program behaviour have to be directly observable from outside the computer; some may be observable only indirectly, by their effect on other programs. Actually, even the values of the program variables inside the computer are inaccessible to a user; they can be controlled or observed only with the aid of an input-output package, which is written in the same language as the program under analysis. The indirect observations are needed to make successful predictions about the behaviour of larger programs, based on knowledge of the behaviour of their components parts.

Successful termination is one of the most important properties of a program to predict, so we need a special variable (called $\overrightarrow{ok}$) which is true just if and when termination occurs. The corresponding initial variable $\overleftarrow{ok}$ indicates that the program has started. Of course a false value of $\overrightarrow{ok}$ will never be conclusively

observed; but that doesn't matter, because the intention of the theorist and the programmer alike is to ensure it that $\overrightarrow{ok}$ is necessarily true, and to prove it. Such a proof would be vacuous if the possibility of its falsity were not modelled in the theory.

In general, for serious proof of total correctness of programs, it is essential to model realistically all the ways in which a program can go wrong, even if not directly observable. In fact, the progress of science is marked by acceptance of such unobservable abstractions as force and mass and friction as though they were directly measurable quantities. As Einstein pointed out, it is the theory itself which determines what is observable.

In the interactive programming paradigm, the most important observable component of program behaviour is an interaction between the system and its environment. Each kind of interaction has a distinct name. For example, in the process algebra CCS [6] the event name *coin* may stand for the insertion of a pound coin in the slot of a vending machine, and the event name *choc* may stand for the selection and extraction of a chocolate bar by the user.

The CSP [7] variant of process algebra allows the user to record a *trace* of the sequence in which such events have occurred while the machine is running; so $\langle coin, choc, coin \rangle$ is a value of *trace* observed in the middle of the second transaction of the machine; the empty trace $\langle\rangle$ is the value when the machine is first delivered.

We also model the possibility of deadlock (hang-up) by recording the set of events currently offered by the machine's environment, but which it refuses to accept. For example, initially the machine refuses $\{choc\}$ because it has not been paid (or perhaps because it has run out of chocolates). A deadlocked machine is one that refuses all the events offered by its environment.

Practical programming of useful systems will involve a combination of interactive and imperative programming features; and the relevant alphabet must include both internal variable names and external event names. The special variable $\overrightarrow{ok}$ should be reinterpreted as successful stabilisation, or avoidance of livelock (divergence). A new special variable $\overrightarrow{wait}$ is needed to distinguish those stable states in which the program is waiting for an interaction with its environment from those in which it has successfully terminated.

An important achievement in the theory of programming has been to formalise separate models for sequential and for interactive programs, and then to combine them with only a minimum of extra complexity.

A top-down theory of programming is highly conducive to a top-down methodology for program design and development. The identifiers chosen to denote the relevant observations of the ultimate program are first used to describe the intended and permitted behaviour of a program, long before the detailed programming begins. For example, a program can be specified not to decrease the value of x by the statement

$$\overleftarrow{x} \leq \overrightarrow{x}$$

A precondition for termination of a program can be written as the antecedent of a conditional

$$(\overleftarrow{x} < 27 \ \wedge \ \overleftarrow{ok}) \ \Rightarrow \ \overrightarrow{ok}$$

The owner of a vending machine may specify that the number of *choc* events in the *trace* must never exceed the number of *coin* events. And the customer certainly requires that when the balance of coins over chocs is positive, extraction of a chocolate will not be refused.

Explicit mention of refusals is a precise way of specifying responsiveness or liveness of a process, without appeal to the concept of fairness. But there is nothing wrong with fairness: it can be treated simply by allowing traces to be infinite. A fair trace is then one that contains an infinite number of occurrences of some relevant kind of event.

It is not an objective of a programming theory to place finitary or other restrictions on the language in which specifications are written. Indeed, our goal is to place whole power of mathematics at the disposal of the engineer and scientist, who should exercise it fully in the interests of utmost clarity of specification, and utmost reliability in reasoning about correctness. We will therefore allow arbitrary mathematical statements as predicates: as in the mu-calculus, we will even allow the definition of weakest fixed points of monotonic predicate transformers.

In an observational semantics of a programming language, the meaning of an actual computer program is defined simply and directly as a mathematical predicate that is true just for all those observations that could be made of any execution of the program in any environment of use.

For example, let x, y, and z be the entire alphabet of global variables of a simple program. The assignment statement $x := x + 1$ has its meaning completely described by a predicate stating that when it is started, the value of x is incremented, and that termination occurs provided the value of x is not too large. The values of all the other global program variables remain unchanged

$$\overleftarrow{x} < max \ \wedge \ \overleftarrow{ok} \quad \Rightarrow \quad \overrightarrow{ok} \ \wedge \ \overrightarrow{x} = \overleftarrow{x} + 1 \ \wedge \ \overrightarrow{y} = \overleftarrow{y} \ \wedge \ \overrightarrow{z} = \overleftarrow{z}$$

Similarly, the behaviour of the deadlock process in a process algebra can be described purely in terms of its trace behaviour—it never engages in any event, and so the trace remains forever empty

$$trace = \langle\rangle$$

(Here and in future, we will simplify our treatment of processes by ignoring issues of divergence).

This kind of definition of programming concepts enables us to regard both specifications and programs as predicates placing constraints on the range of values for the same alphabet of observational variables; the specification restricts the range of observations to those that are permitted; and the program defines exhaustively the full range of observations to which it could potentially give rise.

As a result, we have the simplest possible explanation of the important concept of program correctness. A program P meets a specification S just if the

predicate describing P logically implies the predicate describing S. Since we can identify programs and specifications with their corresponding predicates, correctness is nothing but the familiar logical implication

$$P \Rightarrow S$$

For example, the specification of non-decreasing x is met by a program that increments x, as may be checked by a proof of the implication

$$\overleftarrow{ok} \wedge \overleftarrow{x} < max \wedge x := x + 1 \quad \Rightarrow \quad \overleftarrow{x} \leq \overrightarrow{x} \wedge \overrightarrow{ok}$$

This simple notion of correctness is obviously correct, and is completely general to all top-down theories of programming. Furthermore it validates in complete generality all the normal practices of software engineering methodology.

For example, stepwise design develops a program in two (or more) steps. On a particular step, the engineer produces a design D which describes the properties of the eventual program P in somewhat greater detail than the specification S, but leaving further details of the eventual program to be decided in later steps.

The general design method is defined and justified by the familiar *cut rule* of logic, expressing the mathematical property of transitivity of logical implication

$$\frac{D \Rightarrow S \quad P \Rightarrow D}{P \Rightarrow S}$$

In words this rule may be read: if the design is correct relative to the specification, and if the program meets its design requirement, then the program also meets its original specification.

The most useful method of constructing the specification of a large system is as the conjunction of its many requirements. Programs and designs can also be combined by conjunction, provided that they have completely disjoint alphabets. In that case, the conjunction can generally be implemented by parallel execution of its operands.

Such a parallel implementation is also possible when programs share parts of their alphabet, provided that these include observations of all the ways in which the programs can interact with each other during their execution. In these cases, the stepwise approach to implementation can be greatly strengthened if each step is accompanied by a decomposition of the design D into separately implementable parts D_1 and D_2.

The correctness of the decomposition can be checked before implementation starts by proof of the implication

$$D_1 \wedge D_2 \Rightarrow D$$

Further implementation of the designs D_1 and D_2 can be progressed independently and even simultaneously to deliver components P_1 and P_2. When the components are put together they certainly will meet the requirements of the original design D.

The proof principle that justifies the method of design by parts is just the expression of the monotonicity of conjunction with respect to implication

$$\frac{P_1 \Rightarrow D_1 \quad P_2 \Rightarrow D_2}{P_1 \wedge P_2 \;\Rightarrow\; D_1 \wedge D_2}$$

An even more powerful principle is that which justifies the reuse of a previously written library component, which has been fully described by the specification L. We want to implement a program P which uses L to help achieve a specification S. What is the most general description of a design for P that will achieve this goal in the easiest way? The answer is just $S \vee \overline{L}$, as described by the proof rule

$$\frac{P \;\Rightarrow\; S \vee \overline{L}}{P \wedge L \;\Rightarrow\; S}$$

The Boolean term $S \vee \overline{L}$ is often written as an implication (e.g., $L \supset S$); indeed, the above law, together with the inference in the opposite direction, is used in intuitionistic logic to define implication as an approximate inverse (Galois connection) of conjunction. An implication is always a predicate, but since it is antimonotonic in its first argument, it will rarely be a program.

The identification of programs with more abstract descriptions of their behaviour offers a very simple and general explanation of a number of important programming concepts. For example, a non-deterministic program can be constructed from two more deterministic programs P and Q by simply stating that you do not care which one of them is selected for execution on each occasion. The strongest assertion you can make about any resulting observation is that it must have arisen either from P or from Q.

So the concept of non-determinism is simply and completely captured by the disjunction $P \vee Q$, describing the set union of their observations. And the proof rule for correctness is just the familiar rule for disjunction, defining it as the least upper bound of the implication ordering

$$\frac{P_1 \Rightarrow D \quad P_2 \Rightarrow D}{P_1 \vee P_2 \;\Rightarrow\; D}$$

In words, if you want a non-deterministic program to be correct, you have to prove correctness of both alternatives. This extra labour permits the most general (demonic) interpretation of non-determinism, offering the greatest opportunities for subsequent development and optimisation.

Existential quantification in the predicate calculus provides a means of concealing the value of a variable, simultaneously removing the variable itself from the alphabet of the predicate. In programming theory, quantification allows new variables local to a particular fragment of program to be introduced and then eliminated.

In a process algebra, local declaration of event names ensures that the internal interactions between components of an assembly are concealed, as it were in a black box, before delivery to a customer. Observations of such interactions are

denoted by some free variable, say x occurring in the formula P_x ; on each execution of P_x this variable must have some value, but we do not know or care what it is. The value and even the existence of the variable can be concealed by using it as the dummy variable of the quantification $(\exists\, x.P_x)$.

An important example of concealment is that which occurs when a program component P is sequentially composed with the component Q, with the effect that Q does not start until P has successfully terminated. The assembly (denoted $P;\ Q$) has the same initial observations as P, and the same final observations as Q. Furthermore, we know that the initial values of the variables of Q are the same as the final values of the variables of P.

But in normal sequential programs we definitely do not want to observe these intermediate values on each occasion that execution of the program passes a semicolon. Concealment by existential quantification makes the definition of sequential composition the same as that of composition in the relational calculus

$$(P;\ Q) =_{df} \exists\, x.P(\overleftarrow{x}, x) \wedge Q(x, \overrightarrow{x})$$

Here we have written x and its superscripted variants to stand for the whole list of global variables in the alphabet of P and Q. In a procedural programming language sequential composition is the commonest method of assembling small components. The definition given above shows that the properties of the assembly can be calculated from a knowledge of its components, just as they can for conjunction.

Surprisingly, sequential composition is like conjunction also in admitting an approximate inverse, —a generalisation of Dijkstra's weakest precondition [3]. $L \setminus S$ is defined as the weakest specification [4] of a program P such that $P;\ L$ is guaranteed to meet specification S. There is also a postspecification, similarly defined. Such inverses can be invaluable in calculating the properties of a design, even though they are not available in the eventual target programming language.

In the explanation of stepwise composition of designs, we used conjunction to represent assembly of components. Conjunction of program components is not an operator that is generally available in a programming language. The reason is that it is too easy to conjoin inconsistent component descriptions, to produce a description that is logically impossible to implement, for example,

$$(x := x + 1) \wedge (x := x + 2), \qquad \text{which equals } \mathbf{false}$$

So a practical programming language must concentrate on operators like sequential composition, which are carefully defined by conjunction and concealment to ensure implementability. Negation must also be avoided, because it turns **true**, which is implementable, to **false**, which is not.

That is why prespecifications, which are antimonotonic in their first argument, cannot be allowed in a programming language. But there is a compensation. Any operator defined without direct or indirect appeal to negation will be monotonic, and the programmer can use for the newly defined operator the same rules for stepwise decomposition that we have described for conjunction. The whole process of software engineering may be described as the stepwise

replacement of logical and mathematical operators used in specifications and designs by the implementable operators of an actual programming language.

Ideally, each step should be small and its correctness should be obvious. But in many interesting and important cases, the structure of the implementation has to differ radically from the usual conjunctive structure of the specification, and the validity of the step must be checked by a more substantial proof. You do not expect to build an engine that is fast, eco-friendly, and cheap from three simpler components, each of which enjoy only one of these properties. A mismatch with implementation structure can throw into question the value of prior specification. But it should not; indeed, the value of specification to the user is greatest just when it is fundamentally and structurally simpler than its delivered implementation.

The simplest implementable operator to define is the conditional, in which the choice between components P and Q depends on the truth or falsity of a boolean expression b, which is evaluated in the initial state. So b can be interpreted as a predicate $\overleftarrow{b}$, in which all variables are replaced by their initial values.

$$\textbf{if } b \textbf{ then } P \textbf{ else } Q =_{df} \overleftarrow{b} \wedge P \vee \neg \overleftarrow{b} \wedge Q$$

All the mathematical properties of the conditional follow directly from this definition by purely propositional reasoning.

The most important feature of a programming language is that which permits the same portion of program to be executed repeatedly as many times as desired; and the most general way of specifying repetition is by recursion.

Let X be the name of a parameterless procedure, and let $F(X)$ be the body of the procedure, written in the given programming language, and containing recursive calls on X itself. Since F is monotonic in the inclusion ordering of the sets of observations desribed by predicates, and since these sets can be regarded as a complete lattice, we can use Tarski's fixed point theorem to define the meaning of each call of X as the weakest possible solution of the implication $X \Rightarrow F(X)$.

This definition applies also to recursively defined specifications. Incidentally, if F is expressed wholly in programming notations, it will be a continuous function, and an equivalent definition can be given as the intersection of a descending chain of iterates of F applied to **true**.

A non-terminating recursion can all too easily be specified as a procedure whose body consists of nothing but a recursive call upon itself. Our choice of the weakest fixed point says that such a program has the meaning **true**, a predicate satisfied by all observations whatsoever. The programmer's error has been punished in the most fitting way: no matter what the specification was (unless it was also trivally **true**), it will be impossible to prove that the product is correct.

This interpretation of divergence does not place any obligation on an implementor of the programming language actually to exhibit the full range of allowable observations. On the contrary, the implementor may assume that the programmer never intended the divergence, and on this assumption may validly perform many useful optimisations on the program before executing it.

$$
\begin{aligned}
P \vee Q &= Q \vee P \\
P \vee (Q \vee R) &= (P \vee Q) \vee R \\
P \vee \mathbf{false} &= P \\
P \vee \mathbf{true} &= \mathbf{true} \\
P \wedge (Q \vee R) &= (P \wedge Q) \vee (P \wedge R) \\
P;\ (Q \vee R) &= (P;\ Q) \vee (P;\ R) \\
(Q \vee R);\ P &= (Q;\ P) \vee (R;\ P)
\end{aligned}
$$

Table 1. Basic algebra of non-determinism

As a result of such optimisations, the program may even terminate, for example,

$$\mathbf{while}\ x \leq 0\ \mathbf{do}\ x := x - 1;\ x := abs(x)$$

can be optimised to nothing, because the optimiser assumes that the intention of the while loop was to terminate, which only happens when x starts positive. The anomalous terminating behaviour of the optimised program for negative x is allowed by the semantics, and is entirely attributed to the fault of the programmer. Our theory of programming, whose objective is to avoid non-termination, can afford to treat all instances of non-termination as equally bad; and the whole theory can often be simplified just by regarding them as equal.

After definition of the relevant programming concepts, the next stage in the top-down exploration of the theory of programming is the formalisation and proof of the mathematical properties of programs. The simplest proprieties are those that can be expressed as algebraic laws, either equations or inequations; they are often pleasingly similar to algebraic properties proved of the familiar operators of the arithmetic of numbers, which are taught at school.

For example, it is well known that disjunction—used to define non-determinism in programming—is like multiplication: it is associative and commutative, with **false** serving as its unit and **true** as its zero. Furthermore, conjunction distributes through disjunction, and so do most simple programming combinators, including sequential composition: see Table 1. Laws are the basis for algebraic reasoning and calculation, in which professional engineers often develop considerable skill.

The same principles of programming language definition apply to process algebras, which have the observational variable *trace* in their alphabet. One of the risks of interactive programming is deadlock; and the worst deadlock is the process that never engages in any recordable action, no matter what events the environment may offer to engage in at the same time. As a result, its trace is always empty

$$\mathbf{0} =_{df} trace = \langle\rangle$$

This definition is equally applicable to the process $STOP$ in CSP.

A fundamental operation of a process algebra is external choice $P + Q$, which allows the environment to choose between its operands by appropriate selection of the first event to occur. It has an astonishingly simple definition

$$P + Q =_{df} (P \wedge Q \wedge \mathbf{0}) \vee (\overline{\mathbf{0}} \wedge (P \vee Q))$$

While the trace is empty, an event can be refused by $P + Q$ just if it can be refused by both of them. When the trace is non-empty, the subsequent behaviour is determined by either P or Q, whichever is consistent with the first event in the trace. If both are, the result is non-deterministic.

As in the case of the conditional, the algebraic properties of this simple definition can be simply verified by truth tables. External choice is commutative, idempotent and associative, with unit $\mathbf{0}$; and it is mutually distributive with non-deterministic union. The corresponding operator $\Box$ in CSP has the same properties, but its definition has been made a little more complicated, to take account of the risk of divergence of one of its two operands. The top-down approach to both theories helps to elucidate exactly how two very similar theories may in some ways be subtly different.

The aim of the top-down method of system development is to deliver programs that are correct. Assurance of correctness is obtained not just by testing or debugging the code, but by the quality of the reasoning that has gone into its construction. This top-down philosophy of correctness by construction is based on the premise that every specification and every design and every program can be interpreted as a description of some subset of a mathematically defined space of observations. But the converse is certainly not true. Not every subset of the observation space is expressible as a program. For example, the empty predicate **false** represents a specification that no physical object could ever implement: if it did, the object described would be irretrievably unobservable.

The question therefore arises, what are the additional characteristics of those subsets of observations that are in fact definable in the restricted notations of a particular programming language? The answer would help us to distinguish the feasible specifications that can be implemented by program from the infeasible ones that cannot.

The distinguishing characteristics of implementable specifications have been called *healthiness conditions* [Dijkstra]. They act like conservation laws or symmetry principles in physics, which enable the scientist quickly to dismiss impossible experiments and implausible theories; and similarly they can protect the engineer from many a wild-goose chase. As in the natural sciences, healthiness conditions can be justified by appeal to the real-world meaning of the variables in the alphabet. Analysis of termination gives a good example.

A characteristic feature of a program in any programming language is that if its first part fails to terminate, any fragment of program which is written to be executed afterwards will never be started, and the whole program will also fail to terminate. In our top-down theory, the non-terminating program is represented by the predicate **true**; so the relevant healthiness condition can be

neatly expressed as an algebraic law, stating that **true** is a left zero for sequential composition

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

This law is certainly not true for all *predicates* P; for example, when P is false, we have

$$\mathbf{true};\ \mathbf{false}\ =\ \mathbf{false}$$

This just means that the healthiness condition is succeeding in its primary purpose of showing that unimplementable predicates like **false** can never be expressed as a program.

The majority of simple algebraic laws that are applicable to programs can be proved once-for-all as mathematical theorems about sets of observations; and they can be applied equally to designs and even to specifications. But healthiness conditions, as we have seen, are just not true for arbitrary sets: they cannot be proved and they must not be applied to specifications. Their scope is mainly confined to reasoning about programs, including program transformation and optimisation. It is therefore an obligation on a programming theorist to prove that each healthiness condition holds at least for all programs expressible in the restricted notations of the programming language, and perhaps to certain design notations as well.

The method of proof is essentially inductive on the syntax of the language. All the primitive components of a program must be proved to satisfy the healthiness condition; furthermore, all the operators of the language (including recursion) must be proved to preserve the health of their operands. Here is a proof that union and sequential composition preserve the healthiness condition that they respect non-termination: for union,

$$\begin{array}{ll}
\mathbf{true};\ (P \vee Q) & \\
\quad = (\mathbf{true};\ P) \vee (\mathbf{true};\ Q) & \\
\qquad\qquad & \text{relational composition distributes through disjunction} \\
\quad = \mathbf{true} \vee \mathbf{true} & \text{by induction hypothesis, } P \text{ and } Q \text{ are healthy} \\
\quad = \mathbf{true} & \vee \text{ is idempotent}
\end{array}$$

and for sequential composition,

$$\begin{array}{ll}
\mathbf{true};\ (P;\ Q) & \\
\quad = (\mathbf{true};\ P);\ Q & \text{composition is associative} \\
\quad = \mathbf{true};\ Q & \text{by inductive hypothesis, } P \text{ is healthy} \\
\quad = \mathbf{true} & \text{by inductive hypothesis, } Q \text{ is healthy}
\end{array}$$

Algebraic laws are so useful in reasoning about programs, and in transforming them for purposes of optimisation, that we want to have as many laws as possible, provided of course that they are valid.

How can we know that a list of proven laws is complete in some appropriate sense? One possible sense of completeness is given by a normal form theorem, which shows that every program in the language can be reduced (or rather expanded) to a normal form (not necessarily expressible in the programming language).

A normal form should be designed so that the identity of meaning of non-identical normal forms is quite easy to decide, for example, merely by rearranging their sub-terms. Furthermore, if two normal forms are unequal, it should always be possible to find an observation described by one of them but not the other.

Unfortunately, there may be *no* finite set of algebraic laws that exactly characterises all true facts about the programming language. For example, even the simple relational calculus has no complete finite axiomatisation. One interpretation of the Church-Turing hypothesis states that no top-down analysis can ever exactly characterise those sets of observations that are computable by programs from those that are not. It is only by modelling computation steps of some kind of machine that we can distinguish the computable from the incomputable.

Specifications are inherently incomputable. It is their negations that are recursively enumerable: and they need to be, because we want to be able to prove by counterexample that a program is *not* correct. If your chief worry is accidental description of something that is incomputable or even contradictory, top-down theory development does not immediately address this concern. Complete protection can be obtained only by starting again from the bottom and working upward.

3 Bottom-Up

A bottom-up presentation of a theory of programming starts with a definition of the notations and syntactic structure of a particular programming language. Ideally, this should be rather a small language, with a minimum provision of primitive features; the hope is that these will be sufficiently expressive to define the additional features of more complex languages.

As an example language, we choose a subset of the pi-calculus at about the level of CCS. Figure 1 expresses its syntax in the traditional Backus-Naur form, and Figure 2 gives an informal specification of the meaning.

The traditional first example of a process expressed in a new process algebra is the simple vending machine VM: see Figure 3. It serves an indefinite series of customers by alternately accepting a *coin* and emitting a chocolate. The expected behaviour of a single customer engaging in a single transaction is

$$cust =_{df} \overline{coin}.choc.\mathbf{0}$$

The behaviour of the whole population of customers is modelled by the unbounded set $!cust$. This population can insert an indefinite number of coins; and at any time a lesser number of chocolates can be extracted.

But we plan to install a simple VM that can serve only one customer at a time. To implement this sequentialisation, we need an internal control signal nx,

by which the machine signals to itself its own readiness for the next customer; a complete definition of the vending machine is given in Figure 3.

$$\begin{aligned}\langle event\rangle &::= \langle identifier\rangle \mid \overline{\langle identifier\rangle}\\ \langle process\rangle &::= \mathbf{0} \mid \langle event\rangle.\langle process\rangle\\ &\mid\ (\langle process\rangle \mid \langle process\rangle) \mid !\langle process\rangle\\ &\mid\ (\mathbf{new}\langle identifier\rangle)\ \langle process\rangle\end{aligned}$$

Fig. 1. Syntax

- $\mathbf{0}$ is the deadlock process: it does nothing.
- $coin.P$ is a process that first accepts a $coin$ and then behaves like P.
- $\overline{nx}.\mathbf{0}$ first emits a control signal nx and then stops.
- $nx.Q$ first accepts a control signal nx and then behaves as Q.
- $P \mid Q$ executes P and Q in parallel. Signals emitted by one may be accepted by the other.
- $!P$ denotes parallel execution of an unbounded number of copies of P: $P \mid P \mid P \mid \ldots$
- (**new** e) P declares that e is a local event used only for interactions within its scope P.

Fig. 2. Explanation

$$\begin{aligned}one &=_{df}\ nx.coin.\overline{choc}.\overline{nx}.\mathbf{0}\\ many &=_{df}\ (!one) \mid (\overline{nx}.\mathbf{0})\\ VM &=_{df}\ (\mathbf{new}\ \ nx)\ many\end{aligned}$$

Fig. 3. Vending machine

The operational semantics of the programming language is presented as a collection of formulae, describing all the permitted steps that can be taken in the execution of a complete program. Each kind of step is described by a transition rule written in the form $P \to Q$, where P gives a pattern to be matched against the current state of the program, and Q describes how the program is changed after the step. For example, the rule

$$(e.P) \mid (\overline{e}.Q) \to (P \mid Q)$$

describes an execution step in which one process accepts a signal on e which is sent by the other. Emission and acceptance of the signal are synchronised, and their simultaneous occurrence is concealed; the subsequent behaviour is defined as parallel execution of the rest of the two processes involved.

The reduction shown above can be applied directly to a complete program consisting of a pair of adjacent processes written in the order shown and separated by the parallel operator |. But we also want to apply the reduction to processes written in the opposite order, to processes which are embedded in a larger network, and to pairs that are not even written adjacently in that network.

Such reductions can be described by a larger collection of formulae: e.g.,

$$\begin{aligned}(\overline{e}.Q) \mid (e.P) &\rightarrow Q \mid P\\ ((\overline{e}.Q) \mid (e.P)) \mid R &\rightarrow (Q \mid P) \mid R\\ (e.Q \mid R) \mid (\overline{e}.P) &\rightarrow (Q \mid R) \mid P\end{aligned}$$

But even this is only a small subset of the number of transition rules that would be needed to achieve communication in all circumstances. A much easier way to deal with all cases is to just postulate that | is a commutative operator, that it is associative, and that it has unit **0**.

$$\begin{aligned}P \mid \mathbf{0} &= P\\ P \mid Q &= Q \mid P\\ P \mid (Q \mid R) &= P \mid (Q \mid R)\end{aligned}$$

(These equations are more usually written with equivalence ($\equiv$) in place of equality, which is reserved for syntactic identity of two texts. They are called structural *congruences*, because they justify substitution in the same way as equality.)

These are called *structural* laws in a process calculus. They represent the *mobility* of process, because the implementation may use the equations for substitution in either direction, and so move a process around until it reaches a neighbour capable of an interaction with it. In a bottom-up presentation, these laws are just postulated as axioms that define a particular calculus; they can be used in the proof of other theorems, but they themselves are not susceptible of proof, because there is no semantic basis from which such a proof could be constructed.

The laws governing reduction apply only to complete programs; and they need to be extended to allow reduction steps to take place locally within a larger context. For example a local reduction can occur anywhere within a larger parallel network, as stated by the rule

$$\text{if } P \rightarrow P' \text{ then } (P \mid Q) \rightarrow (P' \mid Q)$$

A similar law applies to hiding.

$$\text{if } P \rightarrow P' \text{ then } (\textbf{new } e)P \rightarrow (\textbf{new } e)P'$$

But there is no similar rule for $e.P$. A reduction of P is not permitted until after e has happened. It is only this omission of a rule that permits terminating programs to be distinguished from non-termination.

One of the main objectives of a theory of programming is to model the behaviour of computing systems that exist in the world today. The world-wide network of interconnected computers is obviously the largest and most important such system.

Any of the connected computers can emit a communication into the network at any time. There is reasonable assurance that the message will be delivered at some later time at some destination that is willing to accept it (if any). But the exact order of delivery does not necessarily reflect the order of sending: messages in the net can overtake each other.

This aspect of reality is very simply encoded in the calculus by adding a single new reduction step. This just detaches a message from its sender, and allows the message to proceed independently in its own timescale through the network to its destination.

$$\overline{e}.P \rightarrow (\overline{e}.0) \mid P$$

This means that the sender P is not delayed if the receiver is unready at the time of sending. The subsequent actions of the sender proceed in parallel with the journey undertaken by its message.

A calculus with such a reduction rule is called *asynchronous*, and output prefixing is usually omitted from its syntax. That is why the algebraic laws for an asynchronous calculus are different from those of a synchronous one.

Structural laws are also used in addition to reductions to formalise the intended meaning of the constructions of the language. For example, the repetition operator ($!P$) denotes an unbounded set of parallel instances of the same process P. The addition of a new instance of P therefore makes no difference, as stated in the unfolding law

$$!P = P \mid (!P)$$

This law can be applied any number of times

$$P \mid (!P) = P \mid P \mid (!P) \; = \; \ldots$$

If each application allows a reduction, we can generate an infinite reduction sequence leading to potential non-termination. Consider for example the process P that reduces in one step to the empty process

$$P =_{df} \; (e.\mathbf{0} \mid \overline{e}.\mathbf{0}) \; \rightarrow \; \mathbf{0} \mid \mathbf{0} \; = \; \mathbf{0}$$

This can be put into a repetition

$$!P = (e.\mathbf{0} \mid \overline{e}.\mathbf{0}) \mid !P \; \rightarrow \; \mathbf{0} \mid !P \; = !P$$

It follows that $!P$ can be subjected to an infinite series of reductions, without ever engaging in a useful interaction with its environment. This is just what is

known as divergence or livelock, and it is clearly and obviously definable on the basis of an operational semantics. A top-down presentation cannot give such an obviously appropriate *definition* of non-termination, and has to postulate that the artificial variable $\overrightarrow{ok}$ is mysteriously allowed to take the value **false** whenever there is a risk of divergence.

Another useful definition in operational semantics is that of a labelled transition, in which the transition relation $\rightarrow$ is labelled by a trace of interactions with the environment that can occur during the evolution of the process.

$$P \xrightarrow{\langle\rangle} Q =_{df} P \xrightarrow{*} Q$$

$$P \xrightarrow{\langle e\rangle^\frown s} Q =_{df} \exists P'.P \xrightarrow{*} e.P' \wedge P' \xrightarrow{s} Q$$

where $\xrightarrow{*}$ is the reflexive transitive closure of $\rightarrow$.

Now we can trace the evolution of our vending machine, using a few structural laws which seem reasonable

$$\begin{array}{lll}
many & & \\
\quad = & \overline{nx}.\mathbf{0} \mid !one & \\
\quad = & \overline{nx}.\mathbf{0} \mid (nx.coin.\overline{choc}.\overline{nx}.\mathbf{0}) \mid !one & \text{expanding } ! \\
\quad \rightarrow & \mathbf{0} \mid (coin.\overline{choc}.\overline{nx}.\mathbf{0}) \mid !one & \text{reduction} \\
\quad = & coin.\overline{choc}.\overline{nx}.\mathbf{0} \mid !one & \mathbf{0} \text{ is unit of } ! \\
\quad \rightarrow & coin.(\overline{choc}.\overline{nx}.\mathbf{0} \mid !one & \text{reduction} \\
\quad \xrightarrow{coin} & \overline{choc}.\overline{nx}.\mathbf{0} \mid !one & \text{def } \xrightarrow{coin} \\
\quad \xrightarrow{\overline{choc}} & \overline{nx}.\mathbf{0} \mid !one & \text{similarly}
\end{array}$$

$$\therefore \quad many \xrightarrow{\langle coin, \overline{choc}\rangle} many \qquad \text{local reduction}$$

$$\therefore \quad VM = ((\mathbf{new}\ \ nx)\ many \xrightarrow{\langle coin, \overline{choc}\rangle} VM)$$

This mathematical derivation is a close simulation of the execution of the program. But does it prove that the program is correct? And what does correctness mean for a programming language that has been defined only in terms of its internal execution rather than what can be observed from outside?

The usual answer to the more general question is that a program is adequately specified in the programming language itself by displaying the equations that it should satisfy. For example, perhaps what we really want to prove about the vending machine is

$$VM = coin.\overline{choc}.VM$$

(In a more abstract language like CCS or CSP, such an equation would be permitted as a recursive definition of VM).

In constructing the proof of such equations, free use may be made of all the structural laws of the calculus. But in general the structural laws are deliberately restricted to transformations on the static shape of a formula, and they do not give enough information about equality of dynamically evolving behaviour. Such reasoning would require a set of laws much larger than those postulated by the calculus. What laws should they be? And how are they justified?

The solution to this problem is of startling originality. The user of the calculus is allowed to extend its set of laws by any new equations that may be desired, provided that this does not lead to a contradiction. A contradiction is defined as the proof of an equation between processes that obviously ought to be different, like a divergent process and a non-divergent one. For example, an equation $P = Q$ leads to contradiction if you can find some program $C[P]$ containing P which does not diverge, but when P is replaced by Q, $C[Q]$ actually can diverge.

Finer discriminations may be imposed by defining a function $obs(P)$, which maps a program P to some simple set of observations that may be made of it. For example, $obs(P)$ might map P onto the set of environments in which it might deadlock. In each case, one might observe the set of events offered by the environment but refused by the process. Then a contradiction is defined as a law that equates two programs with different observable refusal sets. An equation established in this way is called a *contextual equivalence*.

Proving that a proposed law $P = Q$ leads to contradiction is quite a challenge, because the context $C[]$ that reveals it may be very large. But proving that something is *not* a contradiction can be even harder, because it involves consideration of the infinite set of all possible contexts that can be written around P and Q; such a universal hypothesis requires an inductive case analysis over all the combinators of the calculus. Sometimes, only a reduced set of contexts is sufficient; this fact is established by proof of a *context lemma*.

As a result of their syntactic orientation, proofs by induction tend to be specific to a particular calculus, and care is needed in extending their results to calculi with a slightly different syntax, different reductions, or different structural laws. For this reason, each new variation of a familiar calculus is usually presented from scratch.

Heavy reliance on induction certainly provides a strong motive for keeping down the number of notations in the original syntax to an inescapable core of primitives, even if this makes the language less expressive or efficient in practical use. The pursuit of minimality tends to favour the design of a language at a relatively low level of abstraction. The power of such a language matches that of machine code, which offers enormous power, including the opportunity for each instruction to interfere with the effect of any other.

In the presence of multi-threading or non-determinacy, understanding of the behaviour of an arbitrary program becomes rapidly impossible. And there are few general theorems applicable to arbitrary programs that can aid the understanding, or permit optimising transformations that preserve behavioural equivalence. The solution to this problem is to confine attention to programs that follow defined protocols and restrictive conventions to limit their mutual interaction.

The meaning of conformity with the convention is defined by means of a *type system*, which is also usually presented in a bottom-up fashion. The syntax gives a notation for expressing all the types that will be needed. Then a collection of axioms and proof rules provide a means of deducing which parts of each program can be judged to belong to a particular type. In a higher level programming language the programmer may be required or allowed to provide adequate type information for variables and parameters; but most of the labour of type checking or even type inference can be delegated to a compiler.

The consistency of the typing system is established by showing that pairs of programs equated by the structural laws have the same type, and that each reduction step in execution preserves the proper typing of its operand. This is called a *subject reduction* theorem. Subsequent development of the theory can then be confined to properly typed programs.

A type system based on an operational model may be designed to supply information that is highly relevant to program optimisation. For example, it can detect dead code that will never be executed, and code that will be executed at most once. Other type systems can guarantee absence of certain kinds of programming error such as deadlock. If it is known that no type can be deduced for such an erroneous program, then type consistency ensures that the error can never occur a run time.

Because type systems enforce disciplined interaction, well-typed programs often obey additional laws, useful both for comprehension and for optimisation. Type systems thereby raise the level of abstraction of an operationally defined programming language; their role in the bottom-up development of a theory is complementary to that of healthiness conditions, which in a top-down development bring abstract denotational specifications closer to implementable reality.

Operational presentations of semantics are particularly appropriate for analysis of security aspects of communication in an open distributed network, where co-operation between a known group of agents is subject at any time to accidental or deliberate interference by an outsider. The main role of the language is to define and limit the capabilities of the outsider.

For example, the localisation operator (**new** e) enables an agent in the system to invent an arbitrary secret code or a *nonce*, and the structural laws of the language are designed to ensure that it remains secret except to those who have received it in an explicit communication. It is then the responsibility of an implementation of the language to enforce this level of secrecy by choice of an appropriate cryptographic method. Furthermore, if the proof of security of a protocol depends on observance of type constraints, it is essential at run time to check the types of any code written by an outsider before executing it in a sensitive environment.

The purpose of a secure protocol can often be described most clearly by an equation $P = Q$, where P describes the situation before some desired interaction takes place, and Q describes the desired result afterwards. For example, we might use the equation

$$(e.P \mid \overline{e}.Q) \;=\; P \mid Q$$

to describe the intended effect of transmission of a signal e from Q to P. But this is not a valid equation in the calculus, because it is not secure against interference by an outsider R, which can intercept and divert the signal, as permitted by the reduction

$$e.P \mid \overline{e}.Q \mid e.R \;\rightarrow\; e.P \mid Q \mid R$$

This reduction will be inhibited if the name e is kept secret from the outside, so it is valid to equate

$$(\textbf{new}\ \ e)\ (e.P \mid \overline{e}.Q) \;=\; (\textbf{new}\ \ e)\ (P \mid Q)$$

Since it is assumed that the outsider is limited to the capabilities of the programming language, an arbitrary attack can be modelled by a context $C[]$ placed around both sides of the equation. A standard proof of contextual equivalence would be sufficient to show that there is no such context. That is exactly what is needed to show that no outsider can detect or affect the desired outcome described by the equation.

As in this example, the required protection is often achieved with the aid of the **new** operator, which prevents an outsider from detecting or communicating a signal with the new name. It is much more difficult to design a top-down theory for application to problems of security, privacy and authentication. A top-down theory has to start with a decision of exactly what an intruder could observe of another agent in the system, and what attacks are possible upon it. But this understanding is exactly what the security analysis seeks to develop; it cannot be postulated in advance.

A great deal of research effort has been expended on designing proof techniques that are simpler to apply and more efficient to mechanise than proof by non-contradiction. Many of these methods use a variation of the technique of bisimulation [6]. A bisimulation is a postulated equivalence between programs that is respected by the individual steps of the operational semantics of the language, i.e., if two programs belong to the same equivalence class before the step, they belong to the same equivalence class afterwards.

For particular calculi and for particular kinds of bisimulation, theorists have proved that the postulation of the bisimulation as an equality will not lead to a contradiction. Then that kind of bisimulation may safely be used to prove equality of arbitrary programs in the language. For a well-explored calculus, there may be a whole range of bisimulations of varying strength, some suitable for mechanisation, and some suitable for quick disproof. They are all approximations to the truly intended notion of equality, which is defined by the more elusive concept of contextual equivalence.

As described above, much of the effort in a bottom-up theory goes into the determination of when two programs are equal. This is absolutely no problem in a top-down theory, where normal mathematical equality of sets of observations is used throughout. Conversely, much of the effort of a top-down theory goes into determination of which subsets of observations correspond to implementations. This is absolutely no problem in a bottom-up theory, where programs are always

by definition computable. In each case the theorist approaches the target by a series of approximations. In the happy circumstance that they are working on the same language and the same theory, top-down and bottom-up will eventually meet in the middle.

4 Meeting in the Middle

A brief summary of the merits and deficiencies of top-down and bottom-up presentations show that they are entirely complementary.

- A top-down presentation of a theory of programming gives excellent support for top-down development of programs, with justifiable confidence that they are correct by construction.

  By starting with observable system properties and behaviour, it permits and encourages the advance specification of a system yet to be implemented, and the careful design of the interfaces between its major components. It provides concepts, notations and theorems that can be used throughout the design and implementation of software systems of any size and complexity.

  On the other hand, an abstract denotational semantics gives no help at all in the debugging of incorrect programs. It is therefore useless in the analysis of legacy systems, many of which have been written and frequently changed without regard to general design principles, clarity of structure, or correctness of code.

- A bottom-up presentation of a theory of programming gives excellent support for reasoning about the execution of programs that have already been written.

  By starting with a definition of the individual steps of execution, it models directly the run-time efficiency of programs. Execution traces provide the primary diagnostic information on debugging runs of incorrect programs.

  On the other hand, an operational semantics gives no help at all in relating a program to its intended purpose. It is therefore useless in reasoning about programs before they have been written in the notations of a particular programming language.

If programming theory is ever to make its full contribution to the practice of programming, we must offer all the benefits of both styles, and none of the deficiencies. Neither approach could be recommended by itself. It is clearly foolish to provide a conceptual framework for program design if there is no way of executing the resulting program step by step on a computer. It would be equally unsatisfactory to present an operationally defined theory if there is no way of describing what a program is intended to do.

In the natural sciences, it is a necessary condition of acceptability of a theory that it should agree with experiment. Experiments are equally important in

validation of theories of programming. They test the efficiency with which a new programming concept can be implemented and the convenience with which it can be used. An experiment which requires the design, implementation and use of a completely new programming language is prohibitively time-consuming.

For rapid scientific progress, it is preferable just to add a single new feature to an existing programming language, its compiler and its run time system. The first trial applications may be conducted by a group of experimental programmers, who have accepted the risk that the new feature may soon be changed or withdrawn. Even such a limited experiment is expensive; and worse, it is difficult to interpret the results, because of uncontrollable factors such as the skill and the experience of the people involved.

That is why it is advisable to restrict experimentation to test only theories that have shown the highest initial promise. The promise of a theory is not judged by its popularity or by its novelty or by its profitability in competition with rival theories. Quite the reverse: it is by its coherence and close agreement with other theories that a new theory can be most strongly recommended for test. Such agreement is much more impressive if the theories are presented in radically differing styles.

From the practical point of view, it is the stylistic differences that ensure complementarity of the benefits to the user of the programming language. And the results of the experiment are much more convincing if the implementors and trial users are completely independent of the original theorists, as they usually are in more mature branches of Science.

The unification of theories is not a goal that is easy to achieve, and it often requires a succession of adjustments to the details of the theories, and in the way they are tested. The development of an abstract denotational model to match a given operational semantics is known as the problem of full abstraction. It took many years to discover fully abstract models for PCF, a simple typed functional programming language that was presented by an operational semantics.

A recent model [1, 5] represents a type of a programming language by the rules of a two-person game, and a function by a strategy for playing the game. A large and complex collection of healthiness conditions is imposed on the games and strategies to ensure that every strategy that satisfies them can be denoted by a program expressed in the syntax of PCF.

It is generally considered sufficient to prove this just for finite games, which correspond to programs that do not use recursion or any other form of unbounded iteration. That is the best that can be done, because it is impossible within an abstract model to formulate healthiness conditions that will select exactly those sets of observations that are implementable as iterative programs.

In the practical development and analysis of programs, it is quite uncommon to make a direct appeal to the definition of the programming language, whether denotational or operational. Much more useful are theorems that have been based on those definitions; many of these take the form of algebraic laws, either proven from definitions or postulated as healthiness conditions. The importance of laws is recognised both by top-downers and by bottom-uppers, who measure

progress in their chosen direction by accumulation of larger collections of useful laws.

When both theories have been adequately developed, I suggest that an appropriate measure of successful meeting in the middle is provided by the overlap of the two collections of laws. Adjustments can (and should) then be made to the details of both theories, until the overlap is sufficiently broad to meet all the needs of practice. If the practitioner uses just the appropriate laws in the appropriate circumstances, the merits of both approaches can be safely combined.

In a perfect meeting, the laws derived from the top-down and from the bottom-up would be exactly the same. In fact, this is not necessary. All that is needed is that the operationally defined axioms and laws should be a subset of those provable from the denotational definitions. Then all the remaining laws proveable from the denotations will be contextual equivalences. The existence of the denotational model guarantees their consistency, even without the need for an exhaustive inductive argument on contexts.

Identity of differently derived theories is not the only goal; and when applying the same derivational techniques to different programming paradigms, differences in the algebraic laws are to be expected and even welcomed. It turns out that a great many algebraic laws are common to nearly all paradigms, but it is the laws that they do not share that are even more interesting. The fundamental property that distinguishes two paradigms is often very neatly expressed by an algebraic law, free of all the clutter of detail involved in a formal definition, and unaltered when the detail changes.

For example, functional programming languages are classified as lazy or non-lazy. In a non-lazy language, each function evaluates its arguments first, so if an argument aborts, so does the function call. As a consequence, functional composition (denoted by semicolon) has abortion as its left zero:

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

However, a lazy functional language does not satisfy this law. It allows a constant function K to deliver its answer without even looking at its argument:

$$\mathbf{true};\ K = K$$

However, a lazy language still satisfies a right zero law:

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

So does a non-lazy language, unless it allows an argument E to raise an exception or jump. In this case the aborting function does not get the chance to start, so $E;\ \mathbf{true} = E$.

Discussion of such laws is highly relevant to the selection and design of a programming language, as well as its implementation and optimisation. Future texts on comparative programming languages will surely exploit the power of algebra to explain the fundamental principles of the subject.

Fascination with the elegance and expressive power of laws was what inspired the development of abstract algebra as a branch of modern mathematics. Since

the earliest days, mathematics has been primarily concerned with the concept of number. Its progress has been marked by the discovery of new and surprising varieties. Starting with positive integers, even the discovery of zero was a major advance. Then come negative numbers, fractions, reals, and complex numbers.

In modern times, study of the foundations of mathematics has given a denotational semantics to each of these different kinds of number. Natural numbers are defined as sets, integers and fractions as pairs, and reals as sequences. Correspondingly different definitions are given for the arithmetic operations that are performed on the different kinds of number.

In each case, algebraic laws are proved on the basis of the definitions. In spite of the fact that the definitions are so different, most of the laws turn out to be the same. It is the sharing of laws that justifies the use of the same arithmetic operator to denote operations with such radically different definitions. The laws have then inspired the development of other interesting mathematical structures, like quaternions and matrices, for which algebraically similar operations can be defined. Algebra, among all branches of mathematics, is the one that takes reusability as its primary goal.

Computing Science makes progress by discovery of new patterns and paradigms of programming. These are embodied in new programming languages, and subjected to the test of trial implementation and use. The procedural paradigm was among the earliest, and still has the widest application. Now there is also a declarative paradigm, which already splits into two major branches, the logical paradigm which permits backtracking, and the functional paradigm that does not. The functional paradigm splits into lazy and non-lazy varieties.

The advent of multiprocessors and networking has introduced a new paradigm of distributed computing, with even more variations. Some of them are based on sharing of global random access memory, and others on explicit communication. Communications may be ordered or unordered; they may be global or directed along channels; and they may be synchronised or buffered.

In addition to notations traditionally recognised in the community as programming languages, we should consider the languages used for database queries, spreadsheets, menu generators, and other complex interfaces that are coming into wide-spread use. A significant challenge for programming theory is to bring some order into this growing range of tools, and develop an understanding to assist in the selection of an appropriate tool for each purpose, and for using them in combination when necessary.

For purposes of classification, comparison, and combination, both denotational and operational semantics have far too much detail to convey the desired understanding and illumination. It is only the algebra that captures the essence of the concepts at an appropriately high level of abstraction. It is perhaps for the same reason that algebraic laws are also the most useful in practice for engineering calculation.

The primary role of algebraic laws is recognised in the most abstract of branches of algebra, namely category theory. Categories provide an excellent source of elegant laws for programming. Its objects nicely represent the types of

a programming language, and its basic operation of composition of arrows is a model for the combination of actions evoked by parts of a program.

Additional important operators and their types are specified entirely by the algebraic laws that they satisfy. The specification of an operator is often accompanied by a proof that there is only one operator that satisfies it – at least up to isomorphism. This gives assurance that the stated laws are complete: no more are needed, because all other categorial properties of the operator can be proved from them. Finally, a wide range of differing categories can be explored and classified simply by listing the operators which they make available and the laws which they satisfy.

These considerations suggest a third approach to the development of programming theory, one that starts with a collection of algebraic laws as a definitive presentation of the semantics of a programming language [2]. The theory then develops by working outwards in all directions. Working upwards explores the range of denotational models for languages which satisfy the laws. Working downwards explores the range of correct implementations for these languages. And working sideways explores the range of similar theories and languages that might have been chosen instead.

The work of the theorist is not complete until the consequences of theory have been fully developed in all relevant directions. Such an ambitious programme can be achieved only by collaboration and accumulation of results by members of different research traditions, each of whom shares an appreciation of the complementary contributions made by all the others.

5 Acknowledgements

The views put forward in this paper evolved during a long collaboration with He Jifeng on research into unifying theories of programming. They contribute towards goals pursued by the partners in the EC Basic Research Project CONCUR; and they have been subjected to test in the EC Basic Research Project PROCOS. They are more fully expounded in [4], which contains a list of 188 further references. Significant contributors to this work at Oxford include Carroll Morgan, Jeff Sanders, Oege de Moor, Mike Spivey, Jeff Sanders, Annabelle McIver, Guy McCusker, and Luke Ong.

Other crucial clarifications and insights were obtained during a sabbatical visit to Cambridge in conversations with Robin Milner, Andy Pitts, Martin Hyland, Philippa Gardner, Peter Sewell, Jamey Leifer, and many others. I am also grateful to Microsoft Research Limited for supporting my visit to Cambridge, and to researchers at Microsoft who have devoted their time to my further education, including Luca Cardelli, Andy Gordon, Cedric Fournet, Nick Benton and Simon Peyton-Jones.

The organisers of POPL 1999 in San Antonio invited me to present my thoughts there, and the participants gave useful encouragement and feedback. Krzysztof Apt and John Reynolds have suggested many improvements that have been made since an earlier draft of the paper, and more that could have been.

References

[1] S. Abramsky, R. Jagadeesan and P. Malacaria. *Full abstraction for PCF*. To appear in *Information and Computation*.
[2] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. CUP 1990, ISBN 0521 400430.
[3] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall 1976, ISBN 013 215871X.
[4] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall 1998, ISBN 0-13-458761-8.
[5] J. M. E. Hyland and C. H. L. Ong. *On Full Abstraction for PCF: I, II and III*. To appear in *Informatics and Computation*.
[6] R. Milner *Communication and Concurrency*. Prentice Hall 1989, ISBN 013 1150073.
[7] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall 1998, ISBN 013 6744095.