Theories of Programming:

top-down and bottom-up and meeting in the middle

Toulouse,

C.A.R.Hoare

March 1999

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 by telescopes of the present day. 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 can 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 all the more familiar concepts of mathematics in terms of the primitives, and to justify the larger proof steps with which mathematicians are more comfortable. Fortunately in this case too, the top-down and the bottom-up styles of investigation have as their common goal an explanation of the internal structure of mathematics and clarification of the relationship between its many branches.

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.

Whatever its scale, an investigation from the top-down starts with an attempt to understand the system as a whole. Since software is 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 the user of a system. A conventional 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, 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 calculation based on descriptions of the behaviour of its major components. A formula is provided for each of the assembly methods available for the purpose. The collection of formulae constitutes a denotational semantics for the notations in which a system can be specified, designed, and eventually implemented. The programming language used for 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 are included in the set defined in 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 programs written in the language. The goal is to assemble a collection of mathematical laws, 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, its operation needs to be defined as a collection of primitive steps that will be taken 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 can be 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 at a higher level of abstraction (or worse, the original documentation 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. One of them is directly applicable to discussion and reasoning about the design of a program before it has been written, and the other 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 criterion of the maturity of a theory when deciding whether it is ready for practical implementation and 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 to 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 $\dot{x}$ denotes the rate of change of x. Such examples drawn from the normal practice of scientists 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). 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. We certainly want to ignore the millions of possible intermediate values, and it is a goal of the theory to validate this simplification. 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.*

Proper understanding of a program 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. The relevant parameters of program behaviour do not have to be directly observable from outside the computer. For example, even the values of the program variables are inaccessible to a user; they can be controlled or observed only with the aid of an input-output package, which ironically may even be written in the same language as the program under analysis. Nevertheless, the theory needs to refer directly to initial and final values of program variables. Indirect observa-

tions 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 $\vec{ok}$) which is true just if and when termination occurs. The corresponding initial variable $\overleftarrow{ok}$ indicates that the program has started. Of course a negative value of $\vec{ok}$ 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 $\vec{ok}$ is necessarily true, and to prove it. Such a proof would be vacuous if the possibility of 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.

A mathematical theory for an interactive programming paradigm, assumes that each interaction between a program and its environment can be observed, and each of them has a distinct name. For example, in the process algebra CCS[Calculus of Communicating Systems] 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[Communicating Sequential Process] 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 $\{choc\}$, which it always refuses when it has run out of chocolates. A deadlocked machine refuses all the events offered by its environment.

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

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

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. There is no need for a programming theory to restrict the language in which such specifications are written. The whole power of mathematics is placed at the disposal of the engineer and scientist, and should be exercised fully in the interests of utmost clarity of specification, and utmost reliability in reasoning about it.

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 the predicate that states the value of x is incremented, and the values of all the other global program variables remain unchanged

$$\overrightarrow{x} = \overleftarrow{x} + 1 \wedge \overrightarrow{y} = \overleftarrow{y} \wedge \overrightarrow{z} = \overleftarrow{z}$$

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

$$trace = <>$$

Thus we can 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

$$x := x + 1 \Rightarrow \overleftarrow{x} \leq \overrightarrow{x}$$

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 stepwise design method of engineering 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 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

$$D_1 \wedge D_2 \Rightarrow D$$

The descriptions D_1 and D_2 should formalise in sufficient detail all the ways in which the components of the eventual product will interact with each other across the interface which separates them. This usually requires extension of the alphabet to include interactions that are not visible to the user, and that are not mentioned in D, or in any earlier design or specification. By making all interactions explicit, the behaviour of an assembly of components can be described exactly by the conjunction of the description of its components. In this case, the 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 original design requirement D. The proof principle that justifies the methods 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

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

This law is often used to define implication as an approximate inverse (Galois connection) of conjunction.

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.

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 will ensure 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 i occurring in the formula P_i; on each execution of P_i 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 $\exists i.P_i$.

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 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.

Surprisingly, sequential composition is like conjunction in admitting an approximate inverse, – a generalisation of Dijkstra's weakest precondition. $L \setminus S$ is defined as the weakest specification 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,

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

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 cannot be allowed in a programming language. 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 gradual replacement of logical and mathematical operators of specifications and designs by the implementable operators of an actual programming language.

The simplest 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, and since predicates 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 $X \Rightarrow F(X)$.

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