Chapter 0

The Challenge of Unification

The study of science has long been split into many branches, and within each branch there are many specialisations. Each specialisation concentrates on some narrowly defined natural process, and hopes to discover the laws which govern it. In the early days of a new branch of science, these laws are very specific to the outcome of a particular experiment, but as the range of experiment broadens, a collection of laws are found to be special cases of some more general theory, and in turn these theories are comprehended within some theory of yet greater generality. This constant tendency in science towards unifying theories has as ultimate goal the discovery of a clear and convincing explanation of the entire working of the natural universe.

A classical example of a unifying theory is Newton's theory of gravitation, which assimilates the motion of the moon or planets in the sky with the trajectories of apples or cannon balls falling to earth. Unification often begins more simply than that, with just a humble classification: Mendeleev's periodic table of elements classified them by their chemical properties, and these were only later explained by the unifying theory of atomic valences. A unifying theory is usually complementary to the theories that it links, and does not seek to replace them. Physicists still hope to find a Grand Unified Theory, which underlies the four known fundamental forces of nature; when found, this will further reinforce our understanding of the separate theories. It certainly will not replace them by abolishing the forces or repealing the laws that govern them.

The drive towards unification of theories that has been so successful in science has achieved equal success in revealing the clear structure of modern mathematics. For example, topology brings order to the study of continuity in all the many forms and applications discovered by analysis. Algebra classifies and generalises the many properties shared by familiar number systems, and succinctly codifies their differences. Logic and set theory formalise the common principles that govern mathematical reasoning in all branches of the subject. Category theory makes yet further abstraction from logic, set theory, algebra and topology. Computing science is a new subject, and we have not yet achieved the unification of theories that should support a proper understanding of its structure. In meeting this challenge, we may find inspiration and guidance even from quite superficial analogies with better established branches of knowledge. Our main appeal will be to thought experiments rather than physical observation; this suggests that mathematics will provide closer analogies than the physical sciences.

A proposed unification of theories occasionally receives spectacular confirmation and reward by the prediction and subsequent discovery of new planets, of new elements or of new particles. In the timespan of decades and centuries, the resulting improvement of understanding may lead to new branches of technology, with new processes and products contributing to the health and prosperity of mankind. But at the start of the research and on initial study of its results, such benefits are purely speculative, and are best left unspoken. The real driving force for the scientist is wonderment about the complexity of the world we live in, and the hope that it can be described simply enough for us to understand, and elegantly enough to admire and enjoy. Computing scientists need no excuse to indulge and cultivate their genuine curiosity about the complex world of computers, and the languages in which their programs are written.

In satisfying this curiosity, we face the challenge of building a coherent structure for the intellectual discipline of computing science, and in particular for the theory of programming. Such a comprehensive theory must include a convincing approach to the study of the range of languages in which computer programs may be expressed. It must introduce basic concepts and properties which are common to the whole range of programming methods and languages. Then it must deal separately with the additions and variations which are particular to specific groups of related programming languages. The aim throughout should be to treat each aspect and feature in the simplest possible fashion and in isolation from all the other features with which it may be combined or confused. Just as the study of chemical molecules is based upon an understanding of their constituent atoms, the study of programming languages should be based on a prior analysis of their constituent features. Simplification and isolation are the very essence of scientific method, and need no apology.

On the other hand, any practical programming language must include a great many features, together with many *ad hoc* compromises needed to reconcile them with efficient implementation and to maintain compatibility with many previously released implementations. The construction of an effective conceptual framework to understand and control the complexity of currently fashionable programming languages is a continuing challenge and stimulus to productive research. If progress is slow, this should be remedied by more rigorous isolation of the fundamental and more general issues. By concentrating on theory, the pursuit of pure science aims to convey a broader and deeper understanding of the whole range of the subject, and to contribute a foundation, a structure and an intellectual framework for further and more specialised studies of its individual branches.

In this introductory chapter we survey the immensity of the task of unification, and the general methods proposed for tackling it. Theories of programming may be classified along three independent axes.

1. Firstly, there is the vast number of programming languages already invented and yet to come. They can be classified under a smaller range of general computational paradigms, based on the structure and technology of their implementation.
2. Each of these paradigms can be described at different levels of abstraction or detail, with a corresponding trade-off between simplicity and accuracy.
3. Finally, there is a choice of mathematical technique for presenting the foundations of each theory in a simple and convincing fashion.

Unification of theories must study their variation along all these axes. To characterise a particular theory in programming (as in science generally), we need to describe the primitive concepts of the theory, and the way that these are related to the real world by observation or experiment. The technical terms chosen to denote these concepts are called the *alphabet* of the theory. Next, there is a choice of primitive statements of the theory and methods of combining them into more complex descriptions of experiments and products. The symbols chosen for this purpose are called the *signature* of the theory. Finally, the results of a theory are expressed by a set of equations or other *laws* that are both mathematically provable and useful in the design of programs and prediction of the results of their execution. Theories are unified by sharing elements of their alphabet, signature and laws. They are differentiated by what they do not share.

This introductory chapter concludes with a survey of challenging tasks which have not been completed in this book, and which are recommended as topics for future research and development.

0.1 Programming paradigms

Programming languages may be classified in accordance with their basic control structures, or computational *paradigm*. The earliest and most widespread paradigm is that of conventional *imperative* programming. Design of an imperative program requires planned reuse of computer storage by assigning new values to its individual locations. Examples of imperative languages are machine code, FORTRAN, COBOL, ALGOL 60, PASCAL and C. The *functional* programming paradigm makes no reference to updatable storage. It specifies a function by a formula that describes how to compute its result from its arguments. This paradigm is embodied in the languages LISP [122], ML [128, 182] and Haskell [95]. The *logical* paradigm specifies the answer to a question by defining the predicates which the answer must satisfy; search for an answer may involve backtracking, as in the language PROLOG [115], or in more recent constraint logic languages such as CLP [100] and CHIP [53].

The *parallel* programming paradigm permits a program to exploit the power of many processing units operating concurrently and cooperating in the solution of the same problem. There are many variations of this paradigm; they correspond to the different kinds of mechanism which are implemented in hardware for the connection of separate processors, and the different kinds of channel through which they communicate and interact with each other. Many of the fastest computers are designed with multiple processing units working out of a single homogeneously addressed main store. In the study of program complexity, this is known as the PRAM model [58], and it is finding application in BSP (the Bulk Synchronous Paradigm) [123, 183] for high performance computing; its characteristic feature is an occasional global synchronisation. At a much lower level of granularity, a similar kind of lockstep progression is standard in hardware design, and its theory is embodied in SCCS [125].

The other main class of parallel programming paradigm is suited to *distributed* systems. It replaces shared storage by communication of messages, output by one process and input by another. An early example was the *actor* paradigm [8, 9], in which any message output by any process could be collected at any subsequent time by any other process, or even by the same one. A similar scheme underlies Linda [63, 65, 64]. Most subsequent message passing models require them to be directed through *channels*, which connect exactly two processes. In the *data flow* variation [33, 75, 106, 107], messages which have been output by one process will be stored in the correct sequence until the inputting process calls for them. In versions designed for *asynchronous* hardware design [105, 181], the wires have no storage, so the outputting process must undertake not to send a second message until the first has been consumed. Finally, the most widely researched variant is that of fully *synchronised* communication, where the output and input of a message occur virtually simultaneously, as in the theories CCS [125, 126], ACP [23] and CSP [88], and in the programming language occam [97]. The component processes of a distributed parallel program play the role of objects in the *object-oriented* programming paradigm [41, 44, 74, 124].

Certain language properties and features can be included or omitted from any language, independently of its underlying paradigm. For example, *non-determinism* is a property of a language by which a program leaves unspecified the exact actions to be performed, or the exact result produced [50, 83, 137]. Non-determinism tends to arise implicitly in parallel languages, but it is easier to study in isolation as an explicit choice operator, independent of the language in which it is embedded.

Another capability is that of *higher order* [154] programming, which allows a program to treat other programs as data or results. It is a common feature of a functional programming language. Data structures containing programs provide another approach to object-oriented programming, and communication of such data models the program distribution capability of Java [12]. *Timing* [10] can be introduced as a facility for synchronising with a clock, measuring either resource usage or the passage of real or simulated time, and *hybrid* systems [71] include an element of continuous change, perhaps modelling an analog computer or even the real world. A surprisingly powerful feature in programming is *probability* [54], which permits the actions of a computer to be selected by random choice with specified (or unspecified) probabilities. This is widely used in simulation studies; it also promises to solve problems of fault tolerance and self-stabilisation, particularly in distributed systems.

0.2 Levels of abstraction

This survey of the branches and specialisations in the science of programming has classified them according to their choice of paradigm, language and feature. Classification by topic of study is characteristic of any branch of science in its early stages. But as our understanding matures, there often appears an orthogonal classification, by which the same materials and phenomena are treated by different theories, at different scales and different levels of complexity or abstraction. The most mature branch of science is physics, which explains the properties of matter by theories at four (or more) levels: chromodynamics deals with the interactions of quarks, quantum theory with elementary particles, nuclear physics with atoms and molecular dynamics with molecules. Above this, the theories of chemistry begin to diversify according to the choice of material studied. At each level the theory is self-contained, and can be studied in isolation. But the most spectacular achievement of physics is the discovery that the theory at each level can in principle be fully justified by embedding it in the theory below: with the aid of plausible definitions of its concepts, its laws are provable at least as approximations to the underlying reality. The necessary calculations have been checked in detail for the case of the simpler particles and atoms, and there is no reason to doubt the scientists' faith in extrapolation of their results to the cases that are too complicated for practical computation. Clarification of the hierarchical structure of physical theories is what gives the study of physics its pride of place among all the branches of science.

A similar hierarchy of theories is evident in mathematics, where set theory is the basis of topology, which provides a foundation for analysis; in its turn, analysis derives and justifies the laws of the differential calculus, which are then applied to the solution of practical problems by engineers and scientists from a broad range of disciplines. Different concepts, notations, theorems and problem solving methods are available at each of these levels. Fortunately, the successful application of each theory does not require any knowledge of its more abstract foundations.

A goal for a unified theory of programming is to suggest a very similar hierarchical approach to software engineering. Even for a single programming paradigm, a unified theory should link a family of related subtheories at various levels of abstraction. A subtheory at a macroscopic level of granularity and at a high level of abstraction will be useful for capture and analysis of the requirements of the eventual user of a software product. A theory at an intermediate level will help in the definition of the components of the product itself, and the interfaces between its subassemblies and parts. At the lowest level, a theory must fully explain the behaviour of programs written in a particular programming language. The links between all the theories at these different levels must be based on mathematical calculations and proof; without that, it is impossible to establish with confidence that the delivered program will meet the originally specified requirements.

0.3 Varieties of presentation

Even confining attention to a single theory defining a single class of phenomenon at a single level of abstraction, there is scope for wide variation in the manner in which the theory is presented. For example, the theory of gravitation may be presented in its original form as governing the effect of forces acting at a distance. A more modern presentation is in terms of field theory; yet another uses Einsteinian geodesics. All these presentations may be proved to be equally valid, because they are formally equivalent. A branch of mathematics often enjoys a similar range of styles of definition. For example, a particular topology may be defined as a family of open sets, subject to certain conditions. Alternatively it can be specified as a closure operation mapping any set onto its smallest containing closed set. Or it may be specified as a collection of neighbourhoods. Each presentation may be suitable for a different purpose; because they are known to be equivalent, an experienced mathematician will move effortlessly between them as required to solve the current problem. Understanding the relationship between the presentations ensures that the diversity is only beneficial; it is an excellent indicator of the value and maturity of the theory as a whole.

A similar diversity of presentation is seen in a theory of programming, which has to explain the meaning of the notations of a programming language. The methods of presenting such a semantic definition may be classified under three headings. The *denotational* [155, 164, 167, 172, 176] method defines each notation and formula of the language as denoting some value in a mathematical domain which is understood independently, say as a function, or as a set of trajectories, or in general as some kind of observation of the properties and behaviour of the program when executed. The *algebraic* [82, 83, 93, 137, 138, 161] style is more subtle and abstract. It does not say what programs actually mean, but if two differently written programs happen to mean the same thing, this can be proved from the equations of an algebraic presentation. An *operational* [73, 126, 151] presentation describes how a program can be executed by a series of steps of some abstract mathematical machine. As in the hardware of current general-purpose stored-program computers, the text of the program itself is often taken as part of the state of the machine.

The denotational style of definition is closest to that used most normally in mathematics, for example, to define complex numbers or matrices and operations upon them. In the case of programs and other engineering products, we can relate the definitions immediately to more or less direct observations of the execution of the program. A specification too is nothing but a description of the observations of the product which the customer will regard as acceptable. This gives an extraordinarily simple definition of the central concept of the theory, namely program correctness. To be correct, a program must be just a subset of the observations permitted by the specification. The definition of a non-deterministic union of two programs is equally simple – just the union of all the observations that might be made of either of the alternatives. Other connectives of propositional logic and predicate calculus also play an important role.

The great merit of algebra is as a powerful tool for exploring family relationships over a wide range of different theories. For example, study of the foundations of mathematics has given denotations to a wide variety of number systems – integers, reals, complex, etc. Deep distinctions are revealed in the structure and content of each kind of number so defined. It is only their algebraic properties that emphasise the family likenesses across the range of number systems. That is why we are justified in calling them all numbers, and using the same symbols for all the arithmetic operators. There are practical advantages too: the same theorems can be reused without proof in all branches of mathematics which share the same axioms. And algebra is well suited for direct use by engineers in symbolic calculation of parameters and structure of an optimal design. Algebraic proofs by term rewriting are the most promising way in which computers can assist in the process of reliable design.

The operational style of definition of a programming language is distinctive to the study of theoretical computing science, and it also plays an essential practical role. For example, the search for program efficiency and the study of abstract complexity are wholly dependent on counting the number of steps in program execution. In analysing the faults of an incorrect program, it is common to obtain information dumped from an intermediate step of the running program, and this can be interpreted only in the light of an understanding of operational semantics.

Furthermore, the existence (or at least the possibility) of implementation is the only reason for taking an interest in a particular set of notations, or dignifying them with the title of a programming language.

Each of these three styles of presentation has its distinctive advantages for a study of the theory of programming. To combine these advantages, a comprehensive theory of programming treats a programming language in all three styles, and proves that the definitions are consistent with each other in the appropriate sense. In this book, the denotational definition is given first; it provides a basis for proof of the laws needed in the algebraic presentation. At a certain stage, the laws are sufficiently powerful to derive and prove correctness of the step (transition relation) of an operational semantics. This is a natural and fairly easy progression, from abstract definitions through mathematical proof to one or more concrete implementations. But many excellent treatments of semantics [184, 73] proceed in the opposite direction, from the concrete to the abstract. Starting with an operational semantics, one can derive from it a collection of valid algebraic laws and even a denotational semantics. The derivations in this direction use methods based on the concept of simulation in automata theory. These have been further developed by computing scientists under the name *bisimulation*; this has been very successfully applied in the context of CCS [126], and can be extended to other languages.

Caution: The original presentations of the denotational semantics of a programming language made clear a notational distinction between its syntax and its semantics, and the semantics was wholly presented within the mathematical domain of partial functions. These characteristics came to be regarded by later authors as definitive of the nature of denotational semantics. The style which we call denotational has been given many alternative names: "predicative" [77, 90], "specification-oriented" [139], "application-oriented", "observational" or even "relational" [16, 117]. A pioneer of our direct style of denotational definition is Mosses [132]. The decision to return to the original term "denotational" is justified by appeal to its original significance [172, 173]:

1. Each component of the program has a meaning which is independent of its text or the manner of its execution.
2. The meaning of a larger program can be determined as a mathematical function of the *meaning* of its syntactic constituents, not of their syntactic form.

0.4 Alphabets

Our primary subject of study is a new branch of science, the science of computer programming. Like other sciences, its study requires a specialised language for describing the class of relevant phenomena arising from a well-conducted experiment, and introduces a formal framework for deducing consequences from these descriptions. In well-established branches of natural science, the observable results of an experiment are described by a collection of equations or inequations or other mathematical relations. These descriptions will be called by the general logical term *predicate.* The theory of programming uses predicates in the same way as a scientific theory, to describe the observable behaviour of a program when it is executed by computer. In fact, we will define the *meaning* of a program as a predicate which describes as exactly as possible the full range of its possible behaviour, when executed in any possible environment of its use.

Scientific predicates contain many mathematical symbols whose meaning has been defined by pure mathematicians; the laws of reasoning that govern them are ultimately based on the axioms of set theory and logic. But any scientifically meaningful predicate also contains free variables like $x, y, v, \dot{v}$, standing for possible results of measurements taken from an experiment in the real world; for example, the position of some particle, its velocity and acceleration. The relationship between these free variables and the method of observing their values can never be formalised: an understanding can be conveyed only by informal description and practical demonstration. Theories developed for individual branches of science are differentiated by their selection of relevant observations, measurements and naming conventions; the chosen collection of names will be called the *alphabet* of the theory. It is by their alphabets that we shall relate our various theories of programming. Obviously, we will choose the same names for observations that are the same in all the different theories.

Predicates are used in engineering not just to describe the behaviour of an existing product that has already been designed, produced, delivered and put into service. They are also used to specify the requirements on a new product that has not yet even been designed. Such a specification will use the agreed alphabet of free variables to describe the desired behaviour of the eventual product. It may also use any other concept that has a clearly defined or axiomatised mathematical meaning; in software engineering, we may include even the notations of the eventual programming language. The eventual program is, of course, wholly restricted to these notations. The program is *correct* if its interpretation as a predicate describing its behaviour on execution logically implies its original specification. That will ensure that no execution of the program can ever give rise to an observation that violates the specification.

Observations of an experiment may be made at various times during its progress; but the most important time is at the beginning, when the initial settings are made for the controlled variables. We adopt the convention that any observation made at this time will be denoted by an *undecorated* variable $(x, y, ok, trace)$, whereas observations made on later occasions will be *decorated*. For example, the variable x may stand for the initial value of a global variable updated by the program, and the final value of that variable on termination will be denoted x'.

Once an experiment has started, it is usual to wait for some initial transient behaviour to stabilise before making any further observation, and under certain initial conditions, this may never happen. To represent this possibility, we introduce a Boolean variable *ok* (and its decorated version *ok*′), which takes the value *true* just when the program has reached a stable and therefore observable state. Consequently, *ok* is true of any program that has started, and *ok*′ is true of any program that has successfully terminated. In a simple theory of sequential programming, initiation and termination are the only occasions on which the state of executing mechanism may be observed.

A programming language is called *reactive* if the behaviour of its programs can be observed or even altered at stable states intermediate between initiation and termination. To distinguish intermediate states from final ones, we introduce another Boolean variable *wait*, which is true when the program is in a stable intermediate state, and which is false if the program has terminated; after termination, further intermediate observations are of course impossible. A sequential language does not need a *wait* variable, because there are no intermediate observations, and so it would always be false.

We take a view shared with quantum theory that each intermediate observation is an event that may change the subsequent behaviour both of the experiment and of the observer. We introduce the name $\mathcal{A}$ to stand for the set of all possible events that may occur at intermediate stages of the program execution, together with any values that may be measured or observed on that occasion. A typical event may be the exchange of some message with the environment within which the program is running. The value of the message will be observed and recorded, together with the identity of the channel along which it is communicated. $\mathcal{A}$ contains or consists of the set of all such records of communication events.

We use the name *trace* to stand for a cumulative record of all observations that have been made of a program so far. This may be organised as a finite sequence of observations from it, corresponding to the temporal order of occurrence of these events: simultaneous events have to be recorded in arbitrary order. In theories of so-called *true concurrency*, strict interleaving is not needed because the trace is regarded as partially rather than totally ordered. For reasoning about fairness, infinite traces also have to be admitted.

Programming languages of the reactive class offer a facility for synchronisation with the observer on the occasion of each observation. When the program has reached a stable intermediate state (i.e. *wait* is true), it will remain in this state until the environment actually makes the relevant observation, for example by accepting an output message or providing an input message. But not all observations are always possible: on any given occasion, a certain subset of events will be *refused*, even if the environment is ready and willing to engage in them. Such a subset of events is known as a *refusal*. Constraints on the refusal set permit an analysis of the responsiveness or liveness of a distributed system.

The lowest level of programming language is known as *machine code*, because its programs are directly executed by a particular brand of computing machinery. Execution of the program is controlled by the value of a program pointer held in a special hardware sequence control register. This will be denoted by the special global variable *control*, ranging over values in a set αl. This set contains the location numbers of those memory cells which actually store the binary instructions of a given segment of the machine code program. By convention, termination of the program results whenever *control* first takes a value outside αl. The *control* variable is also relevant in a higher level language with jumps and labels. In this case, αl contains the symbolic names of all the labels placed within the program text. We use the control constants *start* and *finish* to stand for implicit labels placed before the beginning and after the end of the program. They are the initial value and final value taken by *control* when the program starts and finishes smoothly, without a jump.

One of the main goals of programming theory is to abstract from the notion of real time; this ensures that the correctness of a program will be unaffected by running it on a faster computer or even a slower one. For many important programs known as *real-time* programs, this abstraction is impossible, because the speed of response is part of the very specification of the program. For describing the behaviour of these programs, we use the variable *clock* (or c) to stand for the real global time at which an observation is made. Real time is distinguished from *resource* time, which keeps a record of the utilisation of processor cycles during the execution of a program; in general, it does not increase while the program is waiting. In a multiprocessor implementation *resource* is a vector quantity, recording utilisation of the various resources available.

That concludes our survey of the main common naming conventions for observations that can be made before, during and after execution of a program. Just as in other branches of science, a specialised theory will select as its alphabet only a subset of all possible observations of the world, as summarised in Appendix 0. At the same time specialisation restricts the range of experimental designs to ensure that all the relevant phenomena can still be predicted within its deliberately limited conceptual framework. In the case of a programming language, a successful restriction will guarantee that programs and their components can be safely specified and designed and proved correct within the restricted alphabet, in confidence that the delivered program will not fail as a result of factors left out of consideration. The benefit of ignoring irrelevant concerns is an obvious gain in simplicity of reasoning but it may also bring benefits in efficiency of implementation of the program and of the more restricted programming language.

The remaining sections of this chapter summarise the methods and conclusions of this book for the benefit of readers who already have some exposure to the mathematical treatment of programming. Other readers may prefer to return to them at a later stage, say after Chapter 4.

0.5 Signatures

Suppose a fixed alphabet has been selected for investigation of a particular programming language. The first task of the theory is to define the meaning of every program in the language as a predicate, with free variables restricted to the alphabet of the language. The predicate should describe as accurately as desired the entire range of observations that could be made of the program when executed. The primitive statements of the language are defined directly by such a predicate. A composite program is built from simpler components by means of the various operators of the language. Each programming operator is therefore defined as an operator on the predicates that describe its operands; this delivers a predicate that describes, again with all appropriate accuracy, the observations of an execution of the larger composite program; each such observation is usually an amalgam of single observations derived from each of the component programs.

The set of operators and atomic components (constants) of a programming theory are known as its *signature*. Essentially, the signature defines the *syntax* of a simple programming language, though a practical language based on the theory may have a much more elaborate syntax, with convenient abbreviations for a number of common idioms. There is a close correlation between the choice of a signature of a theory and its alphabet. Choice of a smaller alphabet restricts the choice of operators to those that can be defined and explained by predicates using only the restricted alphabet.

The concept of a signature is one that is familiar to students of algebra. The signature is the first part of the definition of which branch of algebra is selected for study. For example, the first four rows of Table 0.5.1 contain the operators and constants relevant to lattice theory. The columns of the table give alternative notations which are used in different applications of the algebra, for example propositional calculus or set theory. (Note that the square operators are the other way up from the angular and round ones.) These examples happen also to be Boolean algebras, in which negation is also a valid operation (row 5). A complete lattice is one in which even infinite sets have bounds; they are denoted by the limit operators of rows 6 and 7. They correspond to existential and universal quantification in the predicate calculus. The last three rows of the table extend the signature to that of a relational algebra. Relational algebra lies at the basis of our unification of theories of programming, and its mathematical properties are common to all branches.

1	greatest lower bound	$\sqcap$	$\vee$	$\cup$	union
2	least upper bound	$\sqcup$	$\wedge$	$\cap$	intersection
3	bottom	$\bot$	*true*	U	universe
4	top	$\top$	*false*	$\{\}$	empty
5	negation		$\neg$	overbar	complement
6	lower limit	$\bigsqcap$	$\exists$	$\bigcup$	join
7	upper limit	$\bigsqcup$	$\forall$	$\bigcap$	meet
8	composition	;			product
9	converse	$\smile$			
10	unit, skip	Π	$1'$		identity

Table 0.5.1 Signatures for relations/predicates/sets

A predicate used for specification may be structured with the aid of any of these operators, and indeed any other operator definable in mathematics. Any mathematically sound proof technique may be applied to reasoning about specifications, but there is considerable advantage in restricting predicates to those expressed in smaller sets of notations. This is because notationally restricted predicates are susceptible to a more powerful range of simpler proof techniques. These include familiar methods of symbolic calculation using just the algebraic properties of the chosen operators. Sometimes the algebra permits reduction to a *normal form*; perhaps there is even a decision procedure that can be implemented on a computer. The need for a proof is strongest in the design phase of a project. A language intended for this phase will therefore omit some of the symbols of a specification language. Negation is usually the first operator to go, together with the infinitary limit operators. At the same time, a design calculus introduces other operators which begin to model the global structure of the eventual target program, and thereby provide facilities for programming-in-the-large. These new operators can be defined with the aid of the more abstract operators, even ones that have been deliberately excluded from the signature. However, the excluded operators are used in carefully disciplined ways that do not invalidate the more powerful proof techniques of the design calculus.

In the design of a programming language there is an even stronger reason for restricting the signature of permitted operators yet further: all expressible programs must be computable in the sense of Turing and Church, and preferably they should be implemented with reasonable efficiency on available computing equipment. This requires the exclusion not only of negation but also of conjunction and **false**, for **false** cannot correctly describe any system whatsoever.

The signature of a programming theory at the lowest level of abstraction must obviously include all the notations and operators for the target programming language. Table 0.5.2 gives the signature of basic notations common to nearly all languages treated in this book. Again, they can all be defined in terms of the more abstract operators that have been excluded from the language, but again they use them in such a disciplined way that computability is preserved.

$x := e$	assignment of the value of expression e to the variable x
$P; Q$	sequential composition: Q is executed after P has terminated
$P \lhd b \rhd Q$	conditional: P is executed if b is true initially, otherwise Q
$P \sqcap Q$	non-determinism: P or Q is executed, but it is not specified which
$\|$	parallel execution of processes with disjoint alphabets
var x	introduces a new variable x
end x	terminates the scope of the variable x
$\mu X \bullet F(X)$	call a recursive procedure which has name X and body $F(X)$

Table 0.5.2 Signature of a programming language

In general programming theory, each program and each part of a program may have its own different alphabet selected from the alphabet of the theory. For example, we have already seen that each block of program has its own local variables; each process in a distributed system has a different set $\mathcal{A}$ of events in which it can participate; and in machine code, each segment of the program *must* have a disjoint set αl of locations in which its instructions are stored. In principle, each constant and operator of the signature should be subscripted by the alphabets of its operands and of its results, giving a *heterogeneous* or multi-sorted algebra [42]. In practice the alphabets are omitted, but in a way that permits a compiler to restore them automatically from the context. In exploring the theory, a knowledge of the alphabet is sometimes essential to a definition of the meaning, especially of primitive components like assignments.

Parallel programming languages are those that provide some mechanism for executing two or more programs at the same time, in a way that permits them to interact by sharing one or more global variables in their alphabet. But there is considerable variation in the ways in which the components P and Q may be connected together for mutual interaction. We therefore give a general definition of parallel composition as a *ternary* operator $(P\|_M Q)$, where M is a third predicate describing the way in which observations of P and Q are merged to give an observation of their parallel execution. For example, a particular choice of M will give the independent interleaving operator ($|||$ of CSP), and others will give the more tightly coupled parallel composition ($|$) of CCS or ACP. A reactive parallel language will also provide an external choice operator $\mathbin{\rule{0.15em}{0.8em}}$ (or $+$), which enables the choice between two alternative courses of action to be taken by other processes running in parallel.

$\Vert_M$	parallel composition of type M	$\vert, \Vert$
$\mathbin{\rule{0.15em}{0.8em}}$	external choice	$+$
$c?x$	input a new value of x from channel c	$c.x$
$c!e$	output the value of e on channel x	$\bar{c}.e$
$\vert\vert\vert$	parallelism with interleaving of actions	

Table 0.5.3 Parallelism and communication in CSP and CCS

A selection from the signatures of these more complex programming languages is given in Table 0.5.3. Each of these new operators is definable in terms of the more basic notations of Table 0.5.2. The definitions involve reference and assignment to special observational variables, such as *trace*, *refusal* or *wait*, which are included in the alphabet of the language. It is essential, both to the theory and to its implementation, that these variables are manipulated *only* by means of the operators of the signature. The programmer cannot be allowed to access or change these variables by arbitrary assignment, because it would be clearly impossible to implement (for example) an assignment that moves time backwards or cancels an event that has already occurred. The more complex languages, which need a larger alphabet of concepts to capture and reason about specifications, also need a larger signature of operators to conceal this alphabet from the programmer. But when the definitions of the operators are expanded, it turns out that the complex programs are just a subset of programs expressible in the simple language, in the same way that programs are just a subset of the predicates expressible in a design language or a specification.

0.6 Laws

The main purpose of the mathematical definition of a programming operator is to deduce its interesting mathematical properties. These are most elegantly expressed as algebraic laws – equations usually, but sometimes inequations, with implication between predicates rather than equivalence. For a newly defined binary operator, the first questions are: Is it associative or commutative? Does it have a unit or a zero? And how does it distribute through other operators? Sequential composition clearly should be associative; it has Π as its unit and distributes through disjunction. Many forms of parallel composition share these properties, and are also commutative.

The primary goal of theorists at this stage is to prove a collection of laws which is sufficiently comprehensive that any other true inequation between programs can be derived from the laws alone by algebraic reasoning, without ever again expanding the definition of the operators. This is achieved by defining a highly restricted subset of the programming language, known as a *normal form*; such a form may be defined by excluding many of the operators of the language, and requiring a fixed order of application of the others. Then a proof is given that every program in the language can be reduced by the laws to a normal form (though in practice it is more usually an expansion).

An important goal of a normal form is to help in a test whether two arbitrary programs are equal (or related by implication). Suppose a simple test is available for comparing normal forms. Then the test may be applied to any pair of programs, by first reducing them both to normal form. The reduction may be within the capability of a computer. For human benefit, the task of understanding the whole theory is simplified by separately understanding the simple normal form and the laws by which it may be derived.

The laws for a language may be powerful enough for reduction to several different normal forms, each of them useful for a different purpose. One important purpose of an algebraic transformation is to match the structure of a program to the architecture of the computer on which it will be executed. This is done by a compiler for the language whenever it translates a high level program to machine code. All the control structures of the program are translated to jumps, the data structures are replaced by single-dimensional array references, the local data are held in machine registers, and the expressions and assignments are translated to sequences of commands selected from the very limited range of available machine instructions. Defining this as the normal form, a proof that every program can be transformed to it is simultaneously a proof of correctness of a compiling algorithm that carries out the transformation.

A final advantage to be derived from a normal form is an easy proof of certain additional algebraic properties that are true for all normal forms, and therefore for all programs reducible to that kind of normal form. But such laws are not true for all predicates in general. A predicate that happens to satisfy such a law is called *healthy*, and the law is called a *healthiness condition*. Predicates expressed in intermediate design languages tend to satisfy many but not all of the healthiness conditions of the eventual target programs.

There are often sound physical reasons why programs will always satisfy a given healthiness condition. For example, no program can ever make time go backwards or change the history of what happens before it starts. Let B be a predicate describing all physically possible observations, for example

$$B \;=\; (clock \leq clock') \;\wedge\; (trace \leq trace') \;\wedge \ldots$$

No physically realisable program P can ever give rise to an observation that violates this, a fact that is expressed by the healthiness condition

$$P \Rightarrow B \quad \text{(or equivalently } P \;=\; P \wedge B\text{)}, \qquad \text{for all programs } P$$

At the other extreme, there are certain observations that a given theory regards as irrelevant. They are perfectly feasible, and nothing can prevent such observations being made: they just violate the rules of experimentation that make the theory applicable. Typical examples could be observations made before the program starts, or observations taken when the program is still in a transient or unstable state. Let T be a predicate describing all such improper observations. Since they cannot be prevented, every program must allow them, as expressed in the healthiness condition

$$T \Rightarrow \; P \quad \text{(or equivalently } P = T \vee P\text{)}, \qquad \text{for all programs } P$$

Clearly T and B are a new top and bottom of the lattice of those predicates which satisfy both conditions.

Similarly, let J be a description of the way a program should be initialised, and let K be a description of the way in which the final observation of the program should be taken. Let us suppose (not unreasonably) that repeated initialisation or finalisation has the same effect as just once, that is

$$J = J; J \quad \text{and} \quad K = K; K$$

Since every program P should have been properly initialised and finalised, it must satisfy the healthiness conditions

$$P = J; P \quad \text{and} \quad P = P; K$$

Healthiness conditions play much the same role in programming theory as principles of symmetry and conservation in science. They are not themselves testable by experiment; they are accepted because they are preserved by all known theories that do predict testable results. A well-established principle is then used as a preliminary screen to prevent waste of time considering a theory that violates it. Similarly, a healthiness condition can be used to test a specification or design for *feasibility*, and reject it if it makes implementation demonstrably impossible in the target programming language.

All the healthiness conditions described above have the same general shape

$$P = P \odot X \qquad (\text{or } P = X \odot P)$$

where $\odot$ is an associative operator (e.g. $\cap$, $\cup$, ;) and X itself satisfies the same healthiness condition

$$X = X \odot X$$

As a consequence of this idempotence principle for X, an arbitrary predicate P can be made more healthy by application of the function

$$\phi(P) =_{df} P \odot X$$

Such a function is called a *coercion*, and it plays an important role in linking theories, which may be ranked according to the healthiness conditions that they satisfy. At a level closer to specification they satisfy few; at a level close to the program they satisfy all. Coercions are potentially helpful in making the transition between the stages of a design project, to transform a design document from one level to the next in the hierarchy.

Healthiness conditions are summarised in Appendix 3. They have an important role in unification of theories of programming. They are used to differentiate programming paradigms, to classify them in families, and to clarify the choices that should be made by a programming language designer. The observance of healthiness conditions is the main cause and motivation for complexity in the definitions of a programming theory, and their isolation for independent study is a vital contribution to mastering this complexity.

0.7 Challenges that remain

Unifying theories of programming is an activity that has dominated the authors' research for over ten years. This book concentrates on those theories that have been found most amenable to unification. The study is far from complete. So far, it uses only the simplest methods to treat the most elementary aspects of the basic programming paradigms. To complete the study will require the dedicated cooperation of many theorists, exploring more deeply and far more widely the topics in which they have specialist skills and interests. To bring the results of the study to bear upon the practical problems of software engineering will require long term investment from the builders and suppliers of design automation tools. This section suggests a number of the important challenges that remain.

The first challenge is to extend the range of programming paradigms and features defined and studied within the unifying network. The omissions of this book include: the actor paradigm for parallel programming [8, 9], the process algebra CCS [126], and all varieties of temporal logic [113, 118, 136, 152]. *True concurrency* [146, 147, 148] and *fairness* [60, 144] have been unfairly neglected. The treatment of time and object orientation [10, 41, 44, 73, 74, 84] is only cursory, and the probabilistic paradigm [54] is altogether omitted. No attempt has been made to pursue analogies with computer hardware design, which offers an excellent example of a design hierarchy, ranging over transistor switching circuits, asynchronous circuits, combinational logic, and clocked sequential circuits. A good test for a unifying theory would be to formalise and validate all these interfaces. A more demanding test is to integrate them with the underlying continuous universe at one end [61], and the discrete world of programming at the other [62]. The results could be beneficial to reliable design of hybrid systems, involving a mixture of hardware, software and real-world components.

There are also many newer programming languages, practices and concepts which have not yet been investigated by programming theory. These include languages designed for more specific tasks, such as the calculation and display of spreadsheets, the control of graphical interfaces, the generation of menus, or the maintenance and interrogation of large scale data bases. Many critical computing systems are already implemented in these languages, possibly in combination with each other or with some general-purpose language. As in other branches of engineering, it is such combination of technologies that can present the gravest problems of design and maintenance. The responsible engineer needs to understand the science which underlies each of the pure technologies, as well as that which explains the possible interactions across their interfaces, because the interfaces provide a breeding ground for the most elusive, costly and persistent errors. Avoidance of such errors may be a long term benefit from study of the common theory which underlies all the technologies involved.

The level of exposition in this book is essentially introductory. Each new feature and concept is treated in isolation and in the simplest possible approach, ignoring many known complexities and perhaps some unknown ones as well. These complexities will be discovered, studied and hopefully remedied by those who take up the challenge of applying the theories in combination to complete programming languages. Priority should be given to languages that are already supported by recognised design methods and widely used support tools. Examples from hardware are Verilog [141] and VHDL [163]. On the software side, similar support is given by state charts for embedded systems, SDL [36, 37] for telecommunications, and UML (Unified Modeling Language) [59] for general data processing.

The mathematical methods used in the book are taken from logic, algebra and discrete mathematics. The only real novelty is that these branches of pure mathematics have been turned into applied mathematics. This has been accompanied by a shift in emphasis away from the pure functions that feature so strongly in mathematical tradition. Their primary role has been taken over by the more general concept of a relation. But for deeper investigation of programming theory, mathematicians and computing scientists have developed a range of more sophisticated techniques. Examples from this book include operational semantics, bisimulation and predicate transformers. Further contributions to unifying theories may be expected from the π-calculus [127], from type theory [39, 166], and from game-theoretic modelling [4]. Of particular promise are the classificatory concepts of abstract algebra and category theory [22, 96, 116, 170].

The final and most critical challenge is to bring the contributions of the theory of programming to the aid of programmers engaged in the almost impossible task of reliable design, development and maintenance of computer systems. Such aid is especially needed for very large programs, and for applications in which the consequences of design error could be critical. But first, the methods recommended by the theory should be widely tested on much smaller case studies. These should still be large enough that it is realistic to formalise a specification at a higher level of abstraction than the eventual implementation. The real benefit of a unifying theory will be most appropriately tested if there is more than one stage of design, or more than one implementation paradigm, selected as alternatives or even used in combination. Such case studies are completely absent from this book. The largest example program is about three lines. Fortunately, this does not detract from the value of the theory, which is inherently scalable. Its results are expressed as formulae with variables ranging over all programs, with no limit to their size. The glory of all of mathematics is that its truths are independent of the magnitude and accuracy of numbers, the dimensions of matrices, or the complexity of functions.

But when mathematics first finds application in industry, the problem of scale cannot be ignored. With increasing complexity of case studies, and even more in live application, the size of the programs, formulae and proofs are such that computer assistance is needed to manage them. In hardware design, the use of a range of design automation tools is now considered essential to the production of ever more complex devices at ever shortening intervals, with ever increasing requirement for accuracy. Software design is more complex and less generally well understood, and the state of the art is well behind that of hardware. Nevertheless, there are very promising developments. A good example is provided by symbolic algebra systems (Maple [3], Mathematica [186]), which are routinely used for continuous mathematical symbol processing in science and engineering. In hardware design, new and improved model-checkers (SPIN [94], FDR [159]) are capable of detecting errors in high level algorithms well in advance of implementation. Term rewriting systems (OBJ3 [67]) can reliably calculate the details of a transition from one level of abstraction to another, or reliably optimise a design at a single level. Decision procedures (PVS [143]) can check the validity of the individual steps of a design at the time that they are taken. More difficult tasks can be delegated to a proof search engine [56]. Increased hardware speeds and main store sizes are reinforcing the benefits of improved algorithms for symbol manipulation, and the rate of progress is not decreasing.

At present, the main available mechanised mathematical tools are programmed for use in isolation, and many of them are targeted towards general use in logical and mathematical proof. To extend them to meet the needs of software engineering, it will be necessary to build within each tool a structured library of programming design aids which take the advantage of the particular strengths of that tool. To ensure that the tools may safely be used in combination, it is essential that these theories be unified. In the long run, the tools also should be unified. Only then will we overcome the main barrier to industrial acceptance of both tools and theories: that there are too many of them and they all compete for attention by individual claims of universal applicability, exclusive of all the others. Achievement of a proper balance of healthy competition with eager cooperation of specialist researchers and schools has always been necessary for progress in science, and it has often been the result of unifying previously unconnected theories. Let it be so for programming too.