Unification of Theories: a Challenge for Computing Science

Tony Hoare
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

November 12, 1999

Abstract

Unification of theories is the long-standing goal of the natural sciences; and modern physics offers a spectacular paradigm of its achievement. The structure of modern mathematics has also been determined by its great unifying theories - topology, algebra and the like. The same ideals and goals are shared by researchers and students of theoretical computing science.

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 cannonballs 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.

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 their constituent atoms, the study of complex programming languages should be based on a prior analysis of their constituent features. This is a prerequisite to understanding, reducing and controlling the complexity of their interactions.

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. But one must not be discouraged by the complexity of the initial results. If progress is slow, this should be solved 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.

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. Execution of such a language requires planned reuse of storage by assigning new values to its individual locations. Examples of imperative languages are machine code, FORTRAN, COBOL, 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, ML and Haskell. 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, or in more recent constraint logic languages CPL and CHIP. 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 mechanisms 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; and it is finding application in the Bulk Synchronous Paradigm, which requires 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.

The other main class of parallel programming paradigm replaces shared storage by communication of messages, output by one process and input by another. An early example was the *actor* paradigm, in which any message output by any process could be collected at any subsequent time by any other process, or the same one. A similar scheme underlies Linda. Most subsequent message passing models require them to be directed through *channels*, which connect exactly two processes. In the *dataflow* variation, 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, 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, ACP, and CSP and the programming language occam.

Certain language properties and features can be included or omitted from any language, independently of its underlying paradigm. For example, *nondeterminism* is a property of a language by which a program leaves unspecified the exact actions to be performed, or the exact result produced.

Non-determinism tends to arise implicitly in parallel languages; but it is easier to study in isolation by means of some explicit choice operator, which is independent of the language in which it is embedded.

Another capability is that of *higher order* programming, which allows a program to treat other programs as data or results. It is a common feature of a functional programming language. *Timing* can be introduced as a facility for synchronising with a clock, measuring either real or simulated time; and *hybrid* systems include an element of continuous change, perhaps modelling an analogue computer or even the real world. A surprisingly powerful feature in programming is *probability*, 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.

There are many important 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 grave 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. There remain broad areas of basic research to underpin and unify even the existing technologies, quite apart from those that may gain currency in the future.

Levels

This survey of the branches and specialisations in the science of programming has classified them according to their choice of paradigm, language and feature; this kind of 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 its 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. Fortunately, the successful application of each theory does not require any knowledge of its more abstract foundations.

The same multiplicity of theories can play a useful role in the understanding even of a single programming paradigm. A theory at a macroscopic level of granularity and at a high level of abstraction may be useful for capture and analysis of the requirements of the eventual user of a software product. A theory at an intermediate level may 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 thoroughly understood; without that, it is impossible to reason with confidence that the delivered program will meet the originally specified requirements.

Finally, even confining attention to a single theory defining a single class of phenomenon at a single level of abstraction, there is scope for wide varia-

tion 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; and 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; and, 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 itself.

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* 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 a description of some more general kind of observation. The *algebraic* 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* 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 running of the program. A specification too is nothing but a description of the observations which the customer will regard as acceptable. This gives an extraordinarily simple definition of the central concept of 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.

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 their 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 the process.

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 an operational semantics. Furthermore, the existence or at least the possibility of implementation is the best or 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, the theory of programming should treat each programming language in all three styles, and prove that the definitions are consistent in the appropriate sense. The denotational definition can be 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 traditional and fairly easy progression, from abstract definitions through mathematical proof to one or more concrete implementations. But it is also possible to proceed

in the opposite direction, from the concrete to the abstract. Starting with an operational semantics, we can derive from it a collection of valid algebraic laws and even a denotational semantics. The derivations in this direction use new methods developed by computing scientists under the name *simulation* or *bisimulation*, which has been very successfully explored in the context of CCS, and can be extended to other languages.

Acknowledgement

The ideas of this talk have been developed with the aid of discussions with members of IFIP WG 2.3.

Introductory Bibliography

Here is a rather arbitrary collection of book titles that may help to start on the process of broadening the interests and generalising the understanding of the reader who has been inspired by the goal of unifying theories.

References

[1] Baeten, J.C.M. and Weijland, W.P. *Process Algebra.* Cambridge University Press, 1990.

[2] Barr, M., Wells, C. *Category Theory for Computing Science.* Prentice Hall, second edition, 1995.

[3] Barrow, John D. *Theories of Everything. The Quest for Ultimate Explanation.* Oxford University Press, 1991.

[4] Dijkstra, Edsger W., Scholten, Carel S. *Predicate Calculus and Program Semantics.* Springer Verlag, 1990.

[5] Hennessy, M.C. *Algebraic Theory of Processes.* MIT Press, 1988.

[6] Hoare, C.A.R. *Unified Theories of Programming.* Oxford University Computing Laboratory, 1994.
ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Tony.Hoare/theory94.ps.Z.

[7] Hoare, C.A.R. *Communicating Sequential Processes.* Prentice Hall, 1985.

[8] Lloyd, J.W. *Foundations of Logic Programming.* Springer Verlag, second edition, 1987.

[9] Hentenryck, P. Van. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[10] Milner, A.J.R.G. *Communication and Concurrency.* Prentice Hall, 1989.

[11] Vickers, S. *Topology via Logic.* Cambridge University Press, 1989.