Message of Thanks

On the receipt of the 2011 ACM SIGPLAN Distinguished Achievement Award

Tony Hoare

Principal Researcher, Microsoft Research Ltd.,
Hon. Mem. Cambridge University Computer Laboratory.

Categories and Subject Descriptors D.2.1 [*Programming Languages*]: Formal Definitions and Theory; D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Verification, Reliability, Languages, Theory, Security

Even to one who has lived long enough to receive many awards, the recognition of distinguished achievement from the scientific community in one's own research field is surprisingly welcome. I particularly treasure the SIGPLAN citation for my award, which singles out exactly the modest achievements of which I am most proud. I have known nearly all the previous winners of this award, and they number among my professional colleagues, rivals, and friends. I have derived inspiration from them all, and I am pleased to be regarded in their company.

The award prompts me to reflect again on the origins and progress of my professional career: how did I select the topics for my research? What methods did I consider appropriate for an academic researcher to pursue such research? What is the current level of maturity of research on these topics? What does the future hold for it? Why do I regard my renewed interest in unifying theories as a contribution to that future?

Computer programs

I have always regarded the computer program as a worthy topic for scientific study. Indeed, programs must surely be a central topic in computer science, both pure and applied. Having chosen this topic, I addressed myself to the fundamental questions that excite the curiosity of all scientists, no matter what their chosen topic of research.

What does it do? To answer this question, we need a conceptual framework and language for describing the externally observable properties and behaviour the program. Equally important is a description of the environment in which the program is executed, including its users. In many branches of science, separate branches of mathematics have been developed to provide effective descriptive frameworks for the topics relevant to that branch. Fortunately, logicians from around the beginning of the last century have shown that all branches of mathematics rest on a common foundation. To preserve the greatest generality in describing the behaviour of a general-purpose computer, we should exploit this common foundation. From it we get Boolean algebra, predicate calculus, and the theory of sets. When required, it is easy to define from these foundations the structures manipulated by our programs, for example functions, sums, products, types, relations, sequences, bags, etc. Experiments in application have shown that these concepts give concise and intuitive descriptions of computer systems, while preserving a level of abstraction that is appropriate for humanly comprehensible specifications.

How does it work? For this we need to look inside the program, to identify its internal components and the ways that they are connected. Again, the foundations of mathematics provide a language for defining the behaviour of each component, and the interfaces between them. In an idealised vision of rational software engineering, these interface specifications will be written in advance of the design. They will serve as a contract between the implementers of the various program components. This is an ideal that has inspired much excellent pure research in the field, aimed at the design of theories that exploit modularity (compositionality) of designs, even if it has to be sacrificed later to efficiency of the eventually executed program.

Why does it work? The answer to this deeper question must appeal to general principles, which apply not just to a particular program, but to a general range of similar programs, actual or hypothetical. The principles should support a proof that the joint working of all the components of a program will necessarily lead to the correct operation of the whole program, as described by its specification. Then we will really know both why and how the program works.

The principles of programming are often codified as a set of rules defining the semantics of the programming language in which the program is (or can be) expressed. Several forms of semantics have been developed to serve different purposes. An operational semantics provides a guide for an implementation of the programming language, and serves the programmer as a basis for diagnosing errors discovered in test. A deductive semantics gives the pattern for proving correctness of programs, which is a more difficult but more certain way of ensuring that no errors will detected in service. A denotational semantics constructs a mathematical model of the programming language, so that standard mathematical reasoning can be applied directly to it. Each of these styles of semantics

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.
ACM 978-1-4503-1083-3/12/01.

has a clearly defined role. The last section of this message expresses the hope that a full theory of programming will eventually unify semantics expressed in all the styles, and so ensure their mutual consistency.

How do we know the theory applies to the real world? This is the question of greatest concern to the scientist. It is answered by conducting a series of varied experiments, with collection and interpretation of their observed results. Every claim of discovery of a new theory, and every extension/correction to an old theory, must be supported by appeal to experiment – even a thought experiment will do. Further and larger experiments are needed to explore the limits of applicability even of already well-supported theories. For objectivity, these experiments should be conducted by scientists independent of the proposers of the theory. Often the theory needs modification or extension to extend the range of its application. A unifying theory is one whose range of application is the widest of all.

How are the results of the research transferred to engineering practice? Modern engineering depends utterly on automation of the design process. It is the computer that works out in detail the implications of every engineering design decision, and checks the serviceability and safety of a product before it comes into service. An established design automation toolset provides a rapid means of world-wide transfer of new scientific results into everyday engineering practice. When scientists agree about the strength of the evidence of a new theory, or an amendment or extension of an old theory, the implementers and suppliers of the tools will compete to incorporate any consequential improvements in the next release of the toolset for which they have developed a market share.

Current state of the art

In mature branches of science, it is the natural world that supplies the experimental observations that support, refute, or refine the theories of pure scientists. Astronomers and physicists (and most recently geneticists) have collaborated on long-term international projects to build the telescopes and reactors to conduct experiments, whose results populate the enormous data bases of the world's computers. For an engineering science, the experiments are performed on the products made by man. In computer science, the programs developed by the open source movement already give cheap and public access to experimental material, on a scale that is fortunately more than adequate for our current research needs.

The main task for modern data-based science is to interpret the enormous volumes of experimental material, by relating it to the natural laws which are believed to explain them. Scientists write computer significant computer programs to analyse the mass of data automatically in the light of current theories. This is now the only way to extract scientifically illuminating information from the data, and so to refine, extend, and even unify existing theories. Analogous tools for scientific analysis of programs are used by the software industry to help in the engineering of critical parts of widely used software. They are often used to support the experimental side of research into the principles of programming.

The last decade has seen an enormous increase in the power of these programming tools. Moore's law predicts every decade a roughly a thousand fold increase in the space-time performance of commodity computer chips, often accompanied by a reduction in price. This rate of progress has been compounded by a comparable increase in the algorithmic efficiency of SAT and SMT solvers and model checkers. The advance in software tools has been driven by regular scientific competitions. These are regularly organised and refereed by independent scientists, and the whole experimental community collaborates in the assembly of realistic challenge material for conduct of the competitions.

A second fillip for program analysis has been the totally unexpected phenomenon of the computer virus. Viruses or other malware exploit a programming error to damage or even take control of (perhaps millions of) computers which run the erroneous program. That is why leading software manufacturers are continuing to invest heavily in the development of program analysis tools. These are based on the best available current theories of programming, and the best available SMT solvers and model checkers. The tools are now applied routinely to many millions of lines of commercial software before release.

The future?

I therefore predict an exciting future for further academic research on the principles of programming, and for further exploitation of its research results. The research will take advantage of the most advanced available industrial program analysis tools to perform experiments, at ever increasing scales, on real and realistic software. The tools themselves will evolve by exploiting experience of their use, both by scientists and by software developers. Continuous interaction of theorists, tool-builders and experimentalists will lead to an exponential increase in the rate of scientific progress; perhaps computer science will match the recent spurt in the progress of physics, astronomy, and more recently biology, which has been achieved by integrating computerised tools into the scientific method and culture.

Much of this progress will be made by collaboration between academic researchers and industrial software developers, who already welcome the opportunity of using program analysis tools to reduce the costs and the risks of programming error. At the same time, industry will continue to pour their resources into more immediately applicable tools, which concentrate on test case generation of early detection of programming errors.

Academic research should not be confined to competing with the better funded research of industrial users. It should also continue to pursue higher and longer term ideals, because this is the only way of ensuring a continued stream of new ideas and even breakthroughs to advance the state of the art. Ideals such as accuracy of measurement or purity of materials are the driving force of science. Even if the theory itself says that they can only be approximated, the approximations can be indefinitely refined. In computer science, the relevant scientific ideal is total correctness of computer programs, guaranteed by proofs conducted with the assistance of computers during their design and implementation. It is for the engineer to decide later in each case how far the ideals must be compromised to meet engineering constraints of cost and timescale.

Unification

My own personal research has recently reverted to pursuit of a scientific ideal, namely the unification of theories of programming. Since this is mentioned in my citation for the distinguished achievement award, I will devote this last section of my message of thanks to explaining why I believe that unification will make a contribution to the future described in the previous section.

In the natural sciences, the quest for a unifying theory is an integral part of the scientific culture. The aim is to show that a single theory applies to a wide range of highly disparate phenomena. For example, the gravitational theory of Isaac Newton applies very accurately both to apples falling towards the earth and to planets falling towards the sun. In many cases, a more homogeneous subset of the phenomena is already covered by a more specialised scientific theory. In these cases, the specialised theory must be derived mathematically from the claimed unified theory. For example, Newton's theory of gravitation unifies the elegant planetary theory

of Kepler, as well as the less elegant Ptolemaic theories of astronomy.

The scientific benefit of a unified theory is that it is supported by all the evidence that has already been accumulated for all of the previous theories separately. Furthermore, each of the previous theories then inherits the support given by the total sum of evidence contributed by all the other theories.

The practicing engineer has different concerns from the scientist, including deadlines and budgets for the current project. The engineer will therefore continue to use familiar more specialised theories that have been found from experience to be well adapted to the particular features of the current project, or the needs of the current client. Indeed, the innovative engineer will often specialise the theory even further, adapting it so closely to current needs that there will never be an opportunity for repeated use. That is why the separate theories that are subsumed by a unifying theory often retain all their practical value, and they are in no way belittled or superseded by the unification.

The real practical value of unification lies in its contribution to the transfer of the results of scientific research into engineering practice. One of the main factors that inhibit the engineer (and the sensible manager) from adopting a scientific theory is that the scientists do not yet agree what that theory should be. Fortunately, there is an agreed method of resolving a scientific dispute. An experiment is designed whose result is predicted differently by all the theories that are party to the dispute. The engineer can then have increased confidence in the winner.

But sometimes, no such decisive experiment can be discovered. This may be because, in spite of differences in their presentation, the theories are in fact entirely consistent. In this case, the only way of resolving the issue is to find a theory that unifies them all. Quantum theory provides an example. Three separate mathematical presentations of quantum theory were put forward by Heisenberg, Schroedinger and Dirac. Then Dirac showed that they were all derivable from a single unified theory. This is what enabled the award of a Nobel prize to all three of them. And quantum theory is now accepted as the nearest to a theory of everything that physics has to offer.

A second contribution of a unified theory to the practicing engineer is in the design and use of a suite of software tools that assist in automation of the design process. Since every major engineering enterprise today combines a range of technologies, it is important that all the specialised members of the tool suite should be based on a common theory, so that they can communicate consistently among each other on standard interfaces which are based upon the unification. The standards also facilitate competition among the tools, and permit independent evolution of separate tools for joint use in a design automation toolset.

Finally, the education of the general scientist and engineer will surely be facilitated by reducing the number of independently developed theories to a single theory, presented in a single coherent framework and notation. That in itself is sufficient justification for conduct by academics of research into unification of theories.