Laws of Programming: The Algebraic Unification of Theories of Concurrency

Tony Hoare

Microsoft Research (Cambridge) Ltd.

Abstract. I began my academic research career in 1968, when I moved from industrial employment as a programmer to the Chair of Computing at the Queens University in Belfast. My chosen research goal was to discover an axiomatic basis for computer programming. Originally I wanted to express the axioms as algebraic equations, like those which provide the basis of arithmetic or group theory. But I did not know how. After many intellectual vicissitudes, I have now discovered the simple secret. I would be proud of this discovery, if I were not equally ashamed at taking so long to discover it.

1 Historical Background

In 1969 [6], I reformulated Bob Floyd's assertional method of assigning meanings to programs [4] as a formal logic for conducting verification proofs. The basic judgment of the logic was expressed as a triple, often written

$$\{p\}q\{r\}.$$

The first operand of the triple (its precondition p) is an assertion, i.e., a description of the state of the computer memory before the program is executed. The middle operand (q) is the program itself, and the third operand (its postcondition r) is also an assertion, describing the state of memory after execution.

I now realise that there is no need to confine the precondition and the postcondition to be simply assertions. They can be arbitrary programs. The validity of the logic in application to programming is not affected. However the restrictions are fully justified by the resulting simplification in application of the logic to program verification.

The logic itself was specified as a collection of proof rules, similar to the system of natural deduction for reasoning in propositional logic. I illustrated the rules by the proof of correctness of a single small and over-simplified program, a very long division. This method of verification has since been used by experts in the proof of many more programs of much greater significance.

In the 1970s, my interests turned to concurrent programming, which I had failed to understand when I was manager of an operating system project for my industrial employer (the project failed [10]). To develop and confirm my understanding, I hoped to find simple proof rules for verification of concurrent programs. In fact, I regarded simplicity of proof as an objective criterion of

P. Baldan and D. Gorla (Eds.): CONCUR 2014, LNCS 8704, pp. 1–6, 2014.

the quality of any feature proposed for inclusion in a high level programming language - just as important as a possibly conflicting criterion, efficiency of implementation. As a by-product of the search for the relevant proof rules, I developed two features for shared-memory multiprogramming: a proposal for the conditional critical region [7], and later for a program structure known as the monitor [8].

At this time, the microprocessor revolution was offering an abundance of cheap computer power for programs running on multiple processors. They were connected by simple wires, and did not share a common memory. It was therefore important that communication on the wires entailed a minimum of software or hardware overhead. The criterion of efficiency led me to the proposal of a programming language structure known as Communicating Sequential Processes (CSP) [9]. The results of this research were exploited in the design of a comparatively successful microprocessor series, the INMOS transputer, and its less widely used programming language *occam* [14]. However, I was worried by the absence of a formal verification system for CSP.

In 1977 I moved to Oxford University, where Dana Scott had developed a tradition of denotational semantics for the formal definition of programming languages [20]. This tradition defines the meaning of a program in terms of all its possible behaviours when executed. I exploited the research capabilities of my Doctoral students at Oxford, Steve Brookes and Bill Roscoe; they developed a trace-based denotational semantics of CSP and proved that it satisfies a powerful and elegant set of algebraic laws [3].

Roscoe exploited the trace-based semantics in a model checking tool called *FDR* [19]. Its purpose was to explore the risk of deadlocks, non-termination and other errors in a program. On discovery of a potential failure, the trace of events leading up to the error helps the programmer to explore its possible causes, and choose which of them to correct.

2 The Origins of CONCUR

In the 1990s, I was a co-investigator on the basic research action CONCUR, funded by the European Community as an ESPRIT project. Its goal was a unification of three rival theories of concurrent programming: CSP (described above), a Calculus of Communicating Systems (CCS, due to Robin Milner) [17], and an Algebra of Communicating Processes (ACP, due to Jan Bergstra) [1,2]. These three designs differed not only in the details of their syntax, but also in the way that their semantic foundations were formalised.

Milners CCS was defined by an operational semantics. Its basic judgment is also a triple, called a transition

$$r \xrightarrow{q} p.$$

But in this case, the first operand (r) is the program being executed, the second operand (q) is a possible initial action of the program, and the third operand (p) is the program which remains to be executed after the first action has been

performed. The operational semantics is given as a set of rules for deriving transitions, similar to those for deriving triples by verification logic.

Such an operational semantics is most directly useful in the design and implementation of interpreters and compilers for the language. In fact, the restriction of q to a single atomic action is motivated by this application. Relaxation of the restriction leads to a 'big step' semantics, which is equally valid for describing concurrent programs, but less useful for describing implementations.

The semantics of ACP was expressed as a set of algebraic equations and inequations, of just the kind I originally wanted in 1969. Equations between pairs of operands are inherently simpler and more comprehensible than triples, and algebraic substitution is a simpler and more powerful method of reasoning than that described by proof rules. Thus algebra is directly useful in all forms of reasoning about programs, including their optimisation for efficient execution on available hardware.

Unfortunately, we did not exploit this power of algebra to achieve the unification between theories that was the goal of the CONCUR project. In spite of the excellent research of the participants, this goal eluded us. I explained the failure as ultimately due to the three different methods of describing the semantics. I saw them as rivals, rather than complementary methods, useful for different purposes.

Inspired by this failure, in the 1990s I worked with my close colleague He Jifeng on a book entitled *"Unifying Theories of Programming"* (published in 1998) [5]. It was based on a model in which programs are relations between the initial and final states of their execution. To represent errors like non-termination, the relations were required to satisfy certain 'healthiness' constraints. Unfortunately, we could not find a simple and realistic model for concurrency and communication in a relational framework.

3 The Laws of Programming [11]

In the 1980s, the members of the Programming Research Group at Oxford were pursuing several lines of research in the theory of programming. There were many discussions of our apparently competing approaches. However, we all agreed on a set of algebraic laws covering sequential programming. The laws stated that the operator of sequential composition (**;**) is associative, has a unit (**skip**), and distributes through non-deterministic choice ($\sqcup$). This choice operator is associative, commutative and idempotent. I now recommend introduction of concurrent composition as a new and independent operator ($\|$). It shares all the algebraic properties of sequential composition and in addition it is commutative.

These algebraic properties are very familiar. They are widely taught in secondary schools. They are satisfied by many different number systems in arithmetic. And their application to computer programs commands almost immediate assent from experienced programmers.

A less familiar idea in the algebra of programming is a fundamental refinement ordering ($p < q$), which holds between similar or comparable programs.

It means that q can do everything that p can do, but maybe more. Thus p is a more determinate program than q; in all circumstances, it is therefore a valid implementation of q. Furthermore, if q has no errors, then neither has p. The algebraic principle of substitution of sub-terms within a term is strengthened to state that replacement of any sub-term of p by a sub-term that refines it will lead to a refinement of the original term p. This property is often formalised by requiring all the operators of the algebra to be monotonic with respect to the refinement ordering. Equality, and the substitution of equals, is just an extreme special case of refinement.

The most important new law governing concurrency is called the exchange law [12,13]. I happened upon it in 2007, and explored and developed it in collaboration with Ian Wehrman, then an intern with me at Microsoft Research. The law has the form of a refinement, expressing a sort of mutual distribution between sequential composition and concurrent composition. It is modelled after the interchange law, which is part of the mathematical definition of a two-category [16]. Although the law has four operands, it is similar in shape to other familiar laws of arithmetic:

$$(p \parallel q); (p \parallel q) < (p; p) \parallel (q; q)$$

The exchange law can be interpreted as expressing the validity of interleaving of threads as an implementation their concurrent composition. Such an interleaving is still widely used in time-sharing a limited number of processing units among a larger number of threads. But the law does not exclude the possibility of true concurrency, whereby actions from different threads occur simultaneously. As a result, the law applies both to shared-memory concurrency with conditional critical regions, as well as to communicating process concurrency, with either synchronous or buffered communication. Such a combination of programming idioms occurs widely in practical applications of concurrent systems.

4 Unification of Theories

This small collection of algebraic laws also plays a central role in the unification of other theories of concurrency, and other methods of presenting its semantics. For example, the deductive rules of Hoare logic can themselves be proved from the laws by elementary algebraic reasoning, just as the rules of natural deduction are proved from the Boolean Algebra of propositions. The proofs are based on a simple algebraic definition of the Hoare triple:

$$\{p\}q\{r\} \triangleq p\,;q < r$$

Hoare logic has more recently been extended by John Reynolds and Peter O'Hearn to include separation logic [15,18], which provides methods for reasoning about object orientation as well as concurrency. It thereby fills two serious gaps in the power of the original Hoare logic. The two new rules of concurrent separation logic can be simply proved from the single exchange law. And vice-versa: the exchange law can be proved from the rules of separation logic.

The concurrency rules for the transitions of Milners CCS can be similarly derived from the exchange law. Again, the proof is reversible. The definition of the Milner transition is remarkably similar to that of the Hoare triple:

$$r \xrightarrow{q} p \triangleq q\,;p < r$$

As a consequence, every theorem of Hoare logic can be translated to a theorem of Milner semantics by changing the order of the operands. And vice versa.

Additional operational rules that govern transitions for sequential composition can also be proved algebraically. The derivation from the same algebraic laws of two distinct (and even rival) systems for reasoning about programs is good evidence for the validity of the laws, and for their usefulness in application to programs.

Finally, a denotational semantics in terms of traces has an important role in defining a mathematical model for the laws. The model is realistic to the actual internal behaviour of a program when it is executed. It therefore provides an effective way of describing the events leading up to an error in the program, and in helping its diagnosis and correction.

5 Prospects

The main initial value of the unification of theories in the natural sciences is to enable experts to agree on the foundation, and collaborate in development of different aspects and different applications of it. To persuade a sceptical engineer (or manager) to adopt a theory for application on their next project, agreement among experts is an essential prerequisite. It is far too risky to apply a theory on which experts disagree.

In the longer term, the full value of a theory of programming will only be realised when their use by programmers is supported by a modern program development toolset. Such a toolset will contain a variety of sophisticated tools, based on different presentations of the same underlying theory. For example, program analysers and verification aids are based on deductive logic. Programming language interpreters and compilers are based on operational semantics. Program generators and optimisers are based on algebraic transformations. Finally, debugging aids will be based on a denotational model of program behaviour.

The question then arises: how do we know that all these different tools are fully consistent with each other? This is established by proof of the consistency of the theories on which the separate tools have been based. Mutual derivation of the theories is the strongest and simplest form of consistency: it establishes in principle the mutual consistency of tools that are based on the separate theories. What is more, the consistency is established by a proof that can be given even in advance of the detailed design of the toolset.

Acknowledgements. My sincere thanks are due to Daniele Gorla and Paolo Baldan for their encouragement and assistance in the preparation of this contribution.

References

1. Bergstra, J., Klop, J.: Fixed point semantics in process algebra. Technical Report IW 208, Mathematical Centre, Amsterdam (1982)
2. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Information and Control 60(1-3), 109–137 (1984)
3. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the ACM 31(3), 560–599 (1984)
4. Floyd, R.: Assigning meanings to programs. In: Proceedings of Symposium on Applied Mathematics, vol. 19, pp. 19–32 (1967)
5. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall International Series in Computer Science (1998)
6. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
7. Hoare, C.A.R.: Towards a theory of parrallel programming. In: Hoare, C.A.R., Perrott, R.H. (eds.) Operating Systems Techniques, Proceedings of Seminar at Queen's University, Belfast, Northern Ireland, pp. 61–71. Academic Press (1972)
8. Hoare, C.A.R.: Monitors: An operating system structuring concept. Communications of the ACM 17(10), 549–557 (1974)
9. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (1978)
10. Hoare, C.A.R.: The emperor's old clothes. Communications of the ACM 24(2), 75–83 (1981)
11. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C., Roscoe, A.W., Sanders, J.W., Sørensen, I.H., Spivey, J.M., Sufrin, B.: Laws of programming. Communications of the ACM 30(8), 672–686 (1987)
12. Hoare, C.A.R., Möller, B., Struth, G., Wehrman, I.: Concurrent kleene algebra. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 399–414. Springer, Heidelberg (2009)
13. Hoare, C.A.R., Wehrman, I., O'Hearn, P.W.: Graphical models of separation logic. In: Engineering Methods and Tools for Software Safety and Security. IOS Press (2009)
14. INMOS. occam Programming Manual. Prentice Hall (1984)
15. Ishtiaq, S.S., O'Hearn, P.W.: BI as an assertion language for mutable data structures. In: Proc. of POPL, pp. 14–26 (2001)
16. Mac Lane, S.: Categories for the working mathematician, 2nd edn. Springer, Heidelberg (1998)
17. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. of LICS, pp. 55–74 (2002)
19. Roscoe, A.W.: Model-checking CSP. In: A Classical Mind: Essays in Honour of C.A.R. Hoare. Prentice Hall International (UK) Ltd. (1994)
20. Scott, D., Strachey, C.: Toward a mathematical semantics for computer languages. Oxford Programming Research Group Technical Monograph, PRG-6 (1971)