DOI 10.1007/s00165-012-0249-0
BCS © 2012
Formal Aspects of Computing (2012) 24: 423–431

Formal Aspects of Computing

In praise of algebra

Tony Hoare[1] and Stephan van Staden[2]

[1] Microsoft Research, Cambridge, UK
[2] ETH Zurich, Zurich, Switzerland

Abstract. We survey the well-known algebraic laws of sequential programming, and extend them with some less familiar laws for concurrent programming. We give an algebraic definition of the Hoare triple, and algebraic proofs of all the relevant laws for concurrent separation logic. We give the provable concurrency laws for Milner transitions, for the Back/Morgan refinement calculus, and for Dijkstra's weakest preconditions. We end with a section in praise of algebra, of which Carroll Morgan is such a master.

Keywords: Algebra, Refinement, Concurrency

1. Introduction

The basic ideas and content of an algebra of sequential programming are familiar [HHJ+87]; they are summarised in Table 1. The last column describes properties of the concurrency operator ($\|$).

As described in the reference, the free variables of the algebraic equations stand equally for both programs and specifications. Both of them are regarded as descriptions of events occurring in and around a computer when the program is executed. We can therefore apply programming operators to specifications and logical operators to programs. The algebra makes no distinctions. However, in informal description of the meaning of the operators it helps to distinguish the cases.

The operator (;) stands for sequential composition of programs. The implementation of $p\,;q$ is expected to obey the constraint that no event in the behaviour of p can depend on any event in the behaviour of q. When p and q are specifications, $p\,;q$ describes the behaviour of any program formed as the sequential composition of two disjoint subprograms, the first of which satisfies p, and the second satisfies q. Disjunction ($\vee$) stands for non-deterministic choice between alternative subprograms, and conjunction $p \wedge q$ stands for a program that behaves in any way that both p and q can behave. Its execution can be highly inefficient; it is even impossible if p and q are disjoint, and have no behaviour in common (in this case $p \wedge q = \bot$, and $\bot$ has no executions). Conjunction is omitted from or highly restricted in most practical programming languages.

Correspondence and offprint requests to: S. van Staden, E-mail: Stephan.vanStaden@inf.ethz.ch
Van Staden was supported by ETH Research Grant ETH-15 10-1.
Dedication: to Carroll Morgan, whose use of algebra has so often given delight to so many of his audience.

Table 1. Basic properties of the operators

	$\vee$	$\wedge$	;	$\parallel$
Commutative	Yes	Yes	No	Yes
Associative	Yes	Yes	Yes	Yes
Idempotent	Yes	Yes	No	No
Unit	$\bot$	$\top$	*skip*	*skip*
Zero	$\top$	$\bot$	$\bot$	$\bot$

The primitive *skip* describes the program that does nothing; it always terminates successfully. The bottom symbol $\bot$ (falsity) describes a program that is never executed. It signifies an error, like a syntactic or typing error, which the compiler is responsible for detecting; the compiler must then prevent subsequent execution of the program. The top symbol $\top$ (truth) describes a program that may do anything whatsoever. Think of it as a program that is susceptible to attack by a virus. There is no limit to the ways in which it may behave or misbehave. Avoidance of $\top$ is the responsibility of the programmer. The clear division of responsibility between the implementer of the programming language and its user is an important function of language design. It also plays a central role in the design of programming tools that help to discharge the user's responsibility.

The laws of sequential programming share a surprising feature with many of the laws of physics: they are invariant to reversal of the direction of time. Every axiom, and every theorem derived from the axioms, remains an axiom or a theorem when the operands of all its sequential compositions are reversed. This is because the laws do not in any way describe the means by which an implementation discovers a valid execution of a program. All the axioms for semicolon are valid when (;) is interchanged with ($\tilde{;}$) which is defined by $p \,\tilde{;}\, q = q \,;\, p$. Caution: this symmetry states that any equation is equally provable as the same equation after the interchange. It does not say that $p \,;\, q = p \,\tilde{;}\, q$. Furthermore, when a new law is added to the algebra, it is necessary to check whether the new law is invariant to time reversal or not. If not, the interchange must not be applied to an equation whose proof relies on the new law.

The partial ordering of refinement is defined as follows (see e.g. [HHJ+87]):

$$p \sqsubseteq q \quad \stackrel{\mathbf{def}}{=} \quad q = p \vee q$$

The properties of disjunction ensure that this is a partial order.

The comparison $p \sqsubseteq q$ when applied to specifications means logical implication. So p is a valid design for a program that implements a specification q. When applied to programs, $p \sqsubseteq q$ means that p is the same as q, or more deterministic. So p is a valid optimisation of q. Reduction of non-determinism may happen even during execution, when some step in the execution of q requires resolution of some of the non-determinism of q; this happens, for example, when two guards in a guarded command of Dijkstra are both true. If p is a program and q is a specification, it means that p is correct: all behaviours of p when executed satisfy the specification q. Refinement therefore plays a central role in specification, design, development, optimisation and execution of programs. It has the same central role in programming as logical implication does in mathematical proof. It is wonderful to use the same algebra for all of them.

1.1. Distribution

The laws of Table 1 deal individually with each operator of the algebra. There are more interesting laws of algebra that link two or more operators. Distribution laws are the best known examples.

1. Sequential composition (and concurrent composition) distributes through choice in both directions. Thus if any component of a sequential program is non-deterministic over a number of cases, it is sufficient (and necessary) to reason about the whole program by considering each case separately. Note that the pair of distribution laws together, one for each direction, respect the reversal of time. One of them would not be enough.
2. Any operator distributive through ($\vee$) is monotonic. This permits a process of collaborative program development by divide-and-conquer (stepwise decomposition).
3. Conjunction is related to disjunction by the equivalence

$$q = p \vee q \quad \Leftrightarrow \quad p = p \wedge q$$

The most interesting law for concurrency is an exchange law, similar to the exchange law of Category Theory. It relates two operators, namely sequential and concurrent composition. When these operators appear as the major and minor connectives of an expression, their roles can be interchanged:

4. $(p \parallel q)\,;\,(p' \parallel q') \subseteq (p\,;\,p') \parallel (q\,;\,q')$

The law expresses the fact that concurrency introduces non-determinism; in fact it introduces more non-determinism when it is the major operator, as on the right hand side of the law. This law respects time reversal.

The above laws are satisfied by an implementation of concurrency as interleaving of strings, described (say) by a regular language. Interleaving is obviously commutative and associative and has the empty string as its unit. The right hand side of the exchange law describes the interleaving of two operands, each of which is a sequential composition. The left hand side describes only those interleavings in which the two semicolons on the right hand side are synchronised.

The shape of the interchange law suggests a divide-and-conquer implementation of the interleaving of two strings: split each string into two non-empty substrings; by two (perhaps concurrent!) recursive calls, generate an interleaving of the two left substrings and an interleaving of the two right substrings; return the sequential composition of the two results.

The separating conjunction of separation logic [ORY01, O'H04] also satisfies all our laws for concurrency. The frame law of separation logic depends on the 'small' exchange law $(p \parallel q)\,;\,r \subseteq p \parallel (q\,;\,r)$, which can be proved from the full exchange law by substituting *skip* for p'.

Conjunction is another operator that exchanges with sequential composition. The proof relies only on the property that (;) is monotonic and that the result of $(\wedge)$ is stronger than both its operands. In fact, exchange laws abound, and each expresses an important insight.

Because of its incrementality, it is easy to extend the algebra with an iteration construct. This unary operator is typically written as a postfix Kleene star: p^* describes the iteration where p is performed zero or more times in sequence. Iteration interacts with the other operators according to four laws from Kleene algebra [Koz94]:

5. $\mathit{skip} \vee (p\,;\,p^*) \subseteq p^*$
6. $p \vee (q\,;\,r) \subseteq r \quad \Rightarrow \quad q^*\,;\,p \subseteq r$
7. $\mathit{skip} \vee (p^*\,;\,p) \subseteq p^*$
8. $p \vee (r\,;\,q) \subseteq r \quad \Rightarrow \quad p\,;\,q^* \subseteq r$

The first law says that p^* has more behaviours than *skip*, and more behaviours than $p\,;\,p^*$. A valid implementation of an iteration can therefore start by unfolding it into two cases, one of which does no iterations, and the other of which does at least one iteration. The second law implies that iteration is the least solution of the first inequation. It permits inductive proofs of the properties of an iteration. The last two laws simply swap the arguments of (;) and thus preserve time reversal. In fact the third law need not be a postulated—it can be derived as a theorem.

2. Hoare triples

We define the familiar Hoare triple in terms of sequential composition and refinement (as in [HWO09]):

$$p\,\{q\}\,r \quad \stackrel{\mathbf{def}}{=} \quad p\,;\,q \subseteq r$$

We will refer to p as the *pre* of the triple, and r as the *post*, and q as the *prog*. The definition says that if *pre* describes what has happened before *prog* starts, then *post* describes what has happened when *prog* has finished. In conventional presentations of Hoare logic (e.g. [Hoa69]), the *pre* and *post* are required to be predicates describing a single state of the executing computer before and after the execution of *prog* respectively. This is just a special case of our more general definition, because a single-state predicate may be regarded as a description of all executions which leave the computer in a state that satisfies the predicate when they terminate. Such an interpretation preserves the intuitive meaning of the triple: in every state that satisfies *pre*, every behaviour of *prog* will establish *post* if and when it terminates.

This definition allows Hoare's familiar laws for the operators of structured programming to be derived from the algebra. To contribute to the insight, the laws below are quoted in the new algebraic notation. After all, it is no longer than the Hoare triple.

1. Consequence (post):

   $$p\,;\,q \subseteq r' \quad \& \quad r' \subseteq r \quad \Rightarrow \quad p\,;\,q \subseteq r$$

   The post of a triple can be weakened. The law depends only on transitivity of $\subseteq$.
2. Consequence (pre):

   $$p \subseteq p' \quad \& \quad p'\,;\,q \subseteq r \quad \Rightarrow \quad p\,;\,q \subseteq r$$

   The other law of consequence permits the pre to be strengthened. It is nothing but a restatement of monotonicity of semicolon.
3. For *skip*:

   $$p\,;\,skip \subseteq p$$

   It depends on the fact that *skip* is the right unit of (;).
4. Sequential composition:

   $$p\,;\,q \subseteq r' \quad \& \quad r'\,;\,q' \subseteq r \quad \Rightarrow \quad p\,;\,(q\,;\,q') \subseteq r$$

   This rule depends on associativity and monotonicity of sequential composition.
5. Non-determinism:

   $$p\,;\,q \subseteq r \quad \& \quad p\,;\,q' \subseteq r \quad \Rightarrow \quad p\,;\,(q \vee q') \subseteq r$$

   This rule states that if both q and q' have the same pre and post, then so does their disjunction. It depends only on rightward distribution of (;) through ($\vee$).
6. Iteration:

   $$p\,;\,q \subseteq p \quad \Rightarrow \quad p\,;\,q^* \subseteq p$$

   Here p is the iteration invariant that q preserves. The rule is a consequence of law 8.
7. Concurrency:

   $$p\,;\,q \subseteq r \quad \& \quad p'\,;\,q' \subseteq r' \quad \Rightarrow \quad (p \parallel p')\,;\,(q \parallel q') \subseteq r \parallel r'$$

   This is the law postulated for separating conjunction in separation logic. It states that the pre of $q \parallel q'$ is the concurrent composition of the separate pre's of q and q', and similarly the post. As a result, the proof of a concurrent composition is modular: it is sufficient to find a pre and post for its operands. The rule of concurrency can be derived directly from the algebraic exchange law, and vice versa.
8. Frame:

   $$p\,;\,q \subseteq r \quad \Rightarrow \quad (p \parallel f)\,;\,q \subseteq r \parallel f$$

   The frame law of separation logic states that for any f, the pre and post of a triple remain valid when they are composed concurrently with f. This rule is derived by monotonicity and commutability of ($\parallel$) from the small exchange law; and derivation of the law from the rule is trivial.
9. Disjunction:

   $$p\,;\,q \subseteq r \quad \& \quad p'\,;\,q \subseteq r' \quad \Rightarrow \quad (p \vee p')\,;\,q \subseteq r \vee r'$$

   This rule depends on the distributivity of (;) through ($\vee$).
10. Conjunction:

    $$p\,;\,q \subseteq r \quad \& \quad p'\,;\,q' \subseteq r' \quad \Rightarrow \quad (p \wedge p')\,;\,(q \wedge q') \subseteq r \wedge r'$$

    The law for conjunction has the same form as the concurrency law, with ($\wedge$) replacing ($\parallel$). The conjunction law of Floyd is just a special case, which arises when q and q' are the same variable.

By reversal of time, every triple provable from the algebra gives rise to another triple. For example, the frame rule translates to rule frame′:

$$p\,;\,q \subseteq r \quad \Rightarrow \quad p\,;\,(q \parallel f) \subseteq r \parallel f$$

3. Concurrency in other calculi

3.1. An operational calculus

One way of reading the basic judgement $p\,;\,q \subseteq r$ is backwards. It says that one of the ways of executing the program r is to execute p first, leaving the execution of the remainder q to be executed after p. This is remarkably close to the meaning of the basic judgement of the Milner calculus of transitions [Mil80]. A transition can be defined in terms of the Hoare triple:

$$r \xrightarrow{p} q \quad \stackrel{\mathbf{def}}{=} \quad p\,;\,q \subseteq r$$

According to the above interpretation of the Milner transition, it is the same as that of the Hoare triple, except for a rotation of its operands! A consequence of this discovery is that every algebraically valid rule of the Hoare calculus can be simply translated into an algebraically valid rule of the Milner calculus, and vice versa. For example, the rule frame′ translates to

$$r \xrightarrow{p} q \quad \Rightarrow \quad r \parallel f \xrightarrow{p} q \parallel f$$

This is one of Milner's concurrency rules, stating that an operand of (‖) may perform any of its possible first actions, independently of the other operand. Instantiating q with *skip* and simplifying, we get

$$r \xrightarrow{p} \mathit{skip} \quad \Rightarrow \quad r \parallel f \xrightarrow{p} f$$

Another Milner concurrency rule is remarkably similar to the Hoare rule:

$$r \xrightarrow{p} q \quad \& \quad r' \xrightarrow{p'} q' \quad \Rightarrow \quad r \parallel r' \xrightarrow{p \parallel p'} q \parallel q'$$

It states that both operands can perform their first action concurrently, and the rest of the computation is the concurrent execution of the rest of each of the operands. This rule forms the basis of communication, which achieves a rendezvous among two concurrent processes and combines their compatible actions in a single internal action. For example, if p and p' in the above rule are the CCS input and output actions a and $\overline{a}$ respectively, then the internal action τ hides the successfully synchronised $a \parallel \overline{a}$ in CCS's familiar rule:

$$r \xrightarrow{a} q \quad \& \quad r' \xrightarrow{\overline{a}} q' \quad \Rightarrow \quad r \parallel r' \xrightarrow{\tau} q \parallel q'$$

Other familiar operational rules that also hold as theorems include ones for prefixing, sequential composition, choice, and iteration:

$$r\,;\,r' \xrightarrow{r} r'$$
$$r \xrightarrow{p} q \quad \Rightarrow \quad r\,;\,r' \xrightarrow{p} q\,;\,r'$$
$$r \xrightarrow{p} q \quad \Rightarrow \quad r \vee r' \xrightarrow{p} q$$
$$r^* \xrightarrow{\mathit{skip}} \mathit{skip}$$
$$r^* \xrightarrow{\mathit{skip}} r\,;\,r^*$$

Applying time reversal to the Hoare rules and writing the results in Milner notation yields rules that are based on the top-level operator of the triple's middle argument:

$$r \xrightarrow{q'} r' \quad \& \quad r' \xrightarrow{q} p \quad \Rightarrow \quad r \xrightarrow{q';\,q} p$$
$$r \xrightarrow{q} p \quad \& \quad r \xrightarrow{q'} p \quad \Rightarrow \quad r \xrightarrow{q \vee q'} p$$
$$p \xrightarrow{q} p \quad \Rightarrow \quad p \xrightarrow{q^*} p$$

Although they look the same as Hoare rules, these rules are dual (and not equivalent) to the counterparts in Hoare logic. In fact, the rules actually used by Milner in his definition of CCS are only a subset of those that are derivable from the algebra introduced here. Milner distinguishes atomic actions of a program (such as input, output and internal actions) from composite actions. He requires that the middle operand of his triple and the first operand of a sequential composition (its prefix) must be atomic. The last three rules are consequently rejected because they use composite actions in their conclusion judgements. Except for the third-last rule, all the other operational rules in this section still hold when the distributivity law $p\,;(q \vee r) = (p\,;q) \vee (p\,;r)$ is removed from the algebra. The third-last rule does not need full distribution, but only the weaker property that (;) is monotone in its second operand.

3.2. The adjoint calculi

A slight rephrasing of the basic judgement $p\,;q \subseteq r$ is that q is a program whose execution after p will lead to satisfaction of r. The weakest post specification r/p is defined in [HHJ+87] as the weakest (specification of a) program q that has this property. Any other program that satisfies this property must be stronger. Let us introduce into our algebra a least upper bound operator for arbitrary sets of programs. We will strengthen the distribution law so that semicolon distributes through arbitrary least upper bounds. We can then define

$$r/p \quad \stackrel{\mathbf{def}}{=} \quad \bigvee\{q \mid p\,;q \subseteq r\}$$

The weakest post-specification can be identified with the specification statement of Back [Bac80, Bac81] and Morgan [Mor88, Mor94]. In Morgan's work, the notation $:[p, r]$ is used instead of r/p.

A direct consequence of the definition is the equivalence

$$q \subseteq r/p \quad \Leftrightarrow \quad p\,;q \subseteq r$$

Again, a fundamental judgement of the calculus is obtained by reordering the arguments of the same judgement of the previously treated calculi. For example, the concurrency rule for weakest post-specifications is translated from the rule of Hoare logic:

$$q \subseteq r/p \quad \& \quad q' \subseteq r'/p' \quad \Rightarrow \quad q \parallel q' \subseteq (r \parallel r')/(p \parallel p')$$

From this rule, we can immediately derive the algebraic law which relates (/) to (‖):

$$(r/p) \parallel (r'/p') \subseteq (r \parallel r')/(p \parallel p')$$

It simply says that (/) and (‖) exchange.

Other laws which follow from the algebra are listed in [HHJ+87]. (/) distributes leftward through conjunctions, and has $\top$ as its left zero. It distributes rightward through disjunction, but ($\vee$) changes into ($\wedge$), i.e. $r/(p \vee q) = (r/p) \wedge (r/q)$, and it has right unit *skip*. It is monotone in the first and anti-monotone in the second argument. All of these properties of Morgan's post-specification can be translated by time reversal to the weakest pre-specification, defined in [HHJ+87] as a generalisation of Dijkstra's weakest precondition [Dij76]. It is defined by

$$p\backslash r \quad \stackrel{\mathbf{def}}{=} \quad \bigvee\{q \mid p\,\tilde{;}\,q \subseteq r\}$$

or more simply by

$$q \subseteq p\backslash r \quad \Leftrightarrow \quad p \subseteq r/q$$

By time reversal, we know that (\) and (‖) also exchange.

All theorems of this paper have been formally checked with Isabelle/HOL [NWP02]. A proof script is available online [Pro11]. Most proofs were found automatically, without interaction, using the Sledgehammer tool [BN10]. For an introduction on algebraic reasoning with Isabelle and Sledgehammer, see [FSW11].

4. In praise of algebra

The merits of algebra as an aid to reasoning were recognised by Gottfried Leibnitz in the eighteenth century. Inspired by his discovery of the infinitesimal calculus, he proposed that the validity of all mathematical proofs could be established by symbolic calculation based on algebraic foundations. In the nineteenth century, George Boole similarly formalised his Laws of Thought in terms of Boolean Algebra. Boolean algebra is now used as the basis of all reasoning, both human and mechanical, about the design, implementation, optimisation, verification and evolution of computer hardware circuits. Would it be too fanciful to suggest that an algebra of programming, and the tools which are based upon it, might be of similar benefit in software engineering?

In the early 20th century, the unifying power of algebra was critical in the exploration of the foundations of mathematics. For example, various branches of arithmetic have completely different foundations. Natural numbers are defined in terms of sets of sets, integers are defined as certain pairs of natural numbers, fractions as pairs of integers, real numbers as sets of fractions, etc. The arithmetic operations on each of these types of number inevitably have very different definitions. Fortunately, all the definitions have been shown to obey (nearly) the same algebraic laws. The differences between the various number systems are simply and clearly formalised by a choice between axioms that are satisfied by only one of them. So it is possible to reason and calculate with numbers, without necessarily specifying what type of number it is. This paper has used algebra to unify four independent (and even competing) calculi of programming: Hoare logic, operational semantics, the specification statement calculus, and weakest preconditions.

In the exploration of the theory of programming, the most useful property of algebra is its incrementality. Axioms for a new programming design pattern or a new programming language concept or structure can be easily introduced into an existing algebraically defined programming language. All the axioms of the existing language remain intact, and so do all the theorems already proved from them. Thus each innovation builds on previous work, and there is no need to start again from scratch.

Incrementality encourages an axiomatic style in the formalisation of programming calculi and programming languages. Each axiom may say very little about the properties of the operators that it contains. Further independent properties can always be added separately. Design decisions about the language can be made one property at a time. This contrasts with the more usual approach to formalisation, in which each operator is introduced by a formal definition or a rule, and only one such definition is allowed. This definition must therefore be sufficiently complex that all the desired properties of the operator can be deduced from it. Of course, when the complete algebra has been specified, there is no further need for incrementality. Then it is reasonable to give a definition of each operator of the algebra in terms of a model, and prove that the definitions satisfy the laws of the algebra.

Finally, when the time comes to put the algebra to practical use, we can enlist the assistance of proof tools, which are already extremely effective at algebraic reasoning. Even the educated general public understands algebra, and many scientists and engineers are good at algebraic calculation. Those with mathematical inclinations even appreciate the elegance of an algebraic proof.

The disadvantage of algebra is its high degree of abstraction. If you are interested in what you are talking about, the algebra will never tell you what it is! As a result, it is hard to tell when there are enough axioms, or whether there are maybe too many. There may even be so many axioms that every formula of the algebra can be proved equal to every other formula. That would not be a useful algebra, because there is only one value in the universe to which the algebra applies, and only one operator, namely the one which always delivers this only value as its result.

The final problem is that algebra will never tell you that a conjectured equation is false. For this reason, the algebraist will normally accompany the postulation of axioms by exploration of a wide range of disparate mathematical models which satisfy these axioms. The scientist or engineer can then choose a model, or construct a new one, that corresponds to some known reality in the real world. Again, the search for counterexamples to a conjecture can exploit the power of modern model checking tools.

An alternative, which is more congenial to the pure mathematician, is to prove a normal form theorem. A normal form is an expression which follows a particular pattern. A normal form theorem gives valid rules which enable every expression of the algebra to be transformed into just one of these normal forms. So if two expressions are reduced to different normal forms, there can be no proof that they are equal. The set of all normal forms is itself an 'initial' model of the algebra.

In summary, the study of algebra (together with its models) has made an immense contribution to the advancement of mathematics; and there is now a good prospect that it may assist in reasoning about computer programs, and in improving confidence in their correctness.

5. Conclusion

Our praise of algebra is supported by evidence drawn from many referenced resources, including doctoral theses [Bac78], technical reports [Bac80], proceedings of conferences [Bac79, HMSW09, HHM+11], scientific monographs [Dij76, Mil80], learned journals [HMSW11], letters [WHO09], and keynote addresses [O'H04]. Each of these publications was written (as this one is too) for a special occasion and for a special audience. The main contribution of this paper is to collect relevant material from all these sources, and to present it briefly, in a coherent, comprehensible and comprehensive manner, to broaden the insight and enlarge the understanding of the readership of a general scientific Journal.

In comparison with its sources, the contributions of this paper are neither numerous nor deep. Indeed their triviality only strengthens the evidence in favour of algebra as an effective research tool. There are no original formulae. Again, discovery that two well-known formulae are in fact the same is sometimes as valuable as the discovery of a new formula. It prevents duplication (and even potential errors) in the planning and implementation of a design automation suite of tools that use both the formulae.

The main original insight of the paper, which surprised and delighted its authors, was the discovery that all the laws satisfy the symmetry of time reversal. Since they also satisfy the order-reversal symmetry of lattices, symmetry provides a great many 'theorems for free'. The authors were also pleased to find an algebraic proof that that both the Dijkstra and the Back and Morgan calculi satisfy a version of the exchange law, and so does conjunction.

We conjecture that the main significance of the algebra reported in this paper is in the convergence of deductive and operational semantics of programming, and their extension to both sequential and concurrent programming. The two symmetries of the algebra show that each style of semantics is just a backwards and upside-down presentation of the other! Furthermore, the two kinds of semantics are consistent, in the sense that any model of the algebra is simultaneously a model of the deductive and of the operational rules for a programming language. This is the basis of a proof of correctness of an operational implementation of the language with respect to its deductive semantics (or vice versa). A traditional proof of the consistency of two complementary calculi can be more arduous. It often uses induction (on the structure of syntax) and co-induction (on the steps of the computation). Such proofs are notoriously fragile in face of extensions or variations in the features of the language. We hope that algebra will make it easier to build new languages with new features on the basis of the results already achieved by research on earlier languages.

Acknowledgments

We are grateful to the anonymous referees of this paper, who made many suggestions for its improvement.

References

[Bac78] Back R-J (1978) On the correctness of refinement steps in program development. PhD thesis, Åbo Akademi, Department of Computer Science. Helsinki, Finland. Report A-1978-4

[Bac79] Back R-J (1979) On the notion of correct refinement of programs. In: 5th Scandinavian logic symposium, Aalborg, Denmark. Aalborg University Press, Denmark

[Bac80] Back R-J (1980) Correctness preserving program refinements: proof theory and applications. Mathematical center tracts, vol 131. Mathematical Centre, Amsterdam

[Bac81] Back R-J (1981) Proving total correctness of nondeterministic programs in infinitary logic. Acta Informatica 15:233–249

[BN10] Böhme S, Nipkow T (2010) Sledgehammer: judgement day. In: Proceedings of the 5th international conference on automated reasoning, IJCAR'10. Springer, Berlin, pp 107–121

[Dij76] Dijkstra EW (1976) A discipline of programming. Prentice-Hall, Englewood Cliffs

[FSW11] Foster S, Struth G, Weber T (2011) Automated engineering of relational and algebraic methods in Isabelle/HOL. In: Proceedings of the 12th international conference on relational and algebraic methods in computer science, RAMICS'11. Springer, Berlin, pp 52–67

[HHJ+87] Hoare CAR, Hayes IJ, Jifeng He, Morgan CC, Roscoe AW, Sanders JW, Sorensen IH, Spivey JM, Sufrin BA (1987) Laws of programming. Commun ACM 30:672–686

[HHM+11] Hoare CAR, Hussain A, Möller B, O'Hearn P, Petersen R, Struth G (2011) On locality and the exchange law for concurrent processes. In: Joost-Pieter K, Barbara König (eds) CONCUR 2011 concurrency theory. Lecture Notes in Computer Science, vol 6901. Springer, Berlin, pp 250–264

[HMSW09] Hoare C, Möller B, Struth G, Wehrman I (2009) Concurrent Kleene algebra. In: Mario B, Gianluigi Z (eds) CONCUR 2009—concurrency theory. Lecture Notes in Computer Science, vol 5710. Springer, Berlin, pp 399–414

[HMSW11] Hoare T, Möller B, Struth G, Wehrman I (2011) Concurrent Kleene algebra and its foundations. J Logic Algebraic Program, 80(6):266–296
[Hoa69] Hoare CAR (1969) An axiomatic basis for computer programming. Commun ACM 12:576–580
[HWO09] Hoare CAR, Wehrman I, O'Hearn PW (2009) Graphical models of separation logic. In: Manfred B, Wassiou S, Tony H (eds) Proceedings of the 2008 Marktoberdorf Summer School on engineering methods and tools for software safety and security. IOS Press, Amsterdam
[Koz94] Kozen D (1994) A completeness theorem for Kleene algebras and the algebra of regular events. Inf Comput 110:366–390
[Mil80] Milner R (1980) A calculus of communicating systems. Lecture Notes in Computer Science, vol 92. Springer, Berlin
[Mor88] Morgan C (1988) The specification statement. ACM Trans Program Lang Syst 10:403–419
[Mor94] Morgan C (1994) Programming from specifications, 2nd edn. Prentice Hall International (UK) Ltd., Hertfordshire
[NWP02] Nipkow T, Wenzel M, Paulson LC (2002) Isabelle/HOL: a proof assistant for higher-order logic. Springer, Berlin
[O'H04] O'Hearn P (2004) Resources, concurrency and local reasoning. In: Philippa G, Nobuko Y (eds) CONCUR 2004—concurrency theory. Lecture Notes in Computer Science, vol 3170. Springer, Berlin, pp 49–67
[ORY01] O'Hearn PW, Reynolds JC, Yang H (2001) Local reasoning about programs that alter data structures. In: CSL '01, of LNCS, vol 2142. Springer, Berlin, pp 1–19
[Pro11] Isabelle/HOL (2011) Proofs. http://se.inf.ethz.ch/people/vanstaden/InPraiseOfAlgebra.thy,
[WHO09] Wehrman I, Hoare CAR, O'Hearn PW (2009) Graphical models of separation logic. Inf Process Lett 109(17):1001–1004

Received 20 December 2011
Accepted in revised form 25 May 2012 by Peter Höfner, Robert van Glabbeek and Ian Hayes
Published online 2 July 2012