Unifying Semantics for Concurrent Programming

Tony Hoare

Microsoft Research
Cambridge
United Kingdom

Abstract. Four well-known methods for presenting semantics of a programming language are: denotational, deductive, operational, and algebraic. This essay presents algebraic laws for the structural features of a class of imperative programming languages which provide both sequential and concurrent composition; and it illustrates the way in which the laws are consistent with the other three semantic presentations of the same language. The exposition combines simplicity with generality by postponing consideration of the possibly more complex basic commands of particular programming languages. The proofs are given only as hints, but they are easily reconstructed, even with the aid of a machine.

Essay in celebration of Samson Abramsky's sixtieth birthday.

1 Introduction

Well-known methods for presenting semantics of a programming language are: denotational [1], deductive [2, 3], operational [4, 5] and algebraic [6, 7]. Each presentation is useful as a formal specification of a different Software Engineering tool.

1. The denotational semantics defines a program in terms of all its legitimate behaviours, when executed on any occasion, and in any possible external environment (including other programs). It provides a theoretical foundation for implementation and use of program test environments, which assist in location, diagnosis and correction of unintended effects in the execution of a program.
2. The deductive semantics (originally called axiomatic), provides a set of proof rules, capable of verifying general properties of all possible executions of a particular program. It is the theoretical foundation for program analysis tools (extended static checkers), and also for semi-automatic verifiers, that assist in finding and checking proofs of program correctness.
3. The operational semantics is a set of rules for running a particular program to produce just one of its possible executions. It is the theoretical foundation of any implementation of the language, by a combination of interpreters, compilers and run-time libraries.

B. Coecke, L. Ong, and P. Panangaden (Eds.): Abramsky Festschrift, LNCS 7860, pp. 139–149, 2013.

4. The algebraic semantics (which is more directly axiomatic) has the simplest and most elegant presentation. It helps directly in efficient reasoning about a program, and in optimising its implementation. An additional role of algebra is to help establish relevant aspects of the mutual consistency of the other forms of semantics. In application, the algebra could contribute to the definition of consistent interfaces between the major components of a Design Automation toolset for Concurrent Software Engineering.

The unification of the four semantic presentations proceeds in four steps. (1) The denotational model is shown to satisfy each law of the algebra. (2) A selection of the laws of the algebra is given an equivalent presentation as a proof rule, or as a pair of rules. (3) The basic triple of Hoare logic is given an algebraic definition, which translates (in both directions) between each structural proof rule of Hoare logic and a rule from (2). (4) The basic triple (transition) of Milner [4, 5] is similarly defined, and shown to translate between each rule of an operational semantics and another of the rules from (2). Thus the selected laws of the algebra are equivalent to the conjunction of the rules of the other two semantics. The proofs are highly modular, and they are presented separately for each operator.

There are many simple and useful, algebraic laws which are valid for the denotational model, but which have no direct counterpart in the deductive or the operational semantics. This suggests that the algebraic method will have much to contribute to the exploration of the principles of programming, and the formalisation of its discoveries for application at the interfaces between software engineering tools.

2 Denotations

Let Act be a set, whose elements are interpreted as individual occurrences of basic atomic actions of a program. It includes actions occurring both inside and in the environment of a computer system that is executing the program. Let Dep be a relation between elements of Act. The statement that $x\ Dep\ y$ is interpreted as saying that x is necessary to enable y , in the sense that action y depends on (prior) occurrence of all the actions that it depends on. In game semantics, Dep might serve as a justification relation. A dependency is often drawn as an arrow between two actions, where the actions are drawn as nodes in a graphical diagram of program execution. Examples of such a diagrams include: (1) message sequence charts, where the vertical arrows represent control dependency in a process or thread, and horizontal arrows represent communications or calls between threads; (2) a Petri occurrence net, where the actions are drawn as transitions, and there is an implicit place for token storage on each arrow; or (3)a hardware waveform diagram, where the actions are rising or falling edges of voltage level on a wire, and arrows between the level changes represent dependencies enforced elsewhere in the circuit. All these examples show actions as the points, and the arrows as the line segments, of a discrete non-metric geometry of interaction.

The purpose of the Act and Dep parameters is to allow formalisation of the meaning of the basic commands of a programming language. For example, the following axioms can be postulated to apply to objects, variables, and communication channels: the 0th action of any object is its allocation, and the last action is its disposal. When the n^{th} action is not a disposal:

- the n^{th} action of an object enables the $(n+1)^{st}$ action;
- the n^{th} action of an output port enables the n^{th} action of the input port of that channel;
- on a single-buffered channel, the n^{th} input enables the $(n+1)^{st}$ output;
- the n^{th} assignment to a variable enables all the reads of the n^{th} assignment;
- the $(n+1)^{st}$ assignment depends on all reads of n^{th} assignment.

The last clause merely says that every read of a variable reads the value most recently assigned to it. This axiom applies to familiar strong memory models. Its violation is characteristic of weaker memory models, where fences have to be inserted to achieve a similar effect.

Let Tra be the powerset of Act. An element of Tra (denoted $p, q, r, \ldots$) is interpreted as a complete and successful trace of the execution of a particular program on a particular occasion, whether in the past, the present or the future. Let Prog be the powerset of Tra. An element of Prog (denoted $P, Q, R, \ldots$) is interpreted as a complete description of all possible and successful traces of the execution of a program, in all possible environments of use. If no trace of a program can be successfully completed, the program is represented by the empty set of traces. This is an error like a syntax error: ideally, a compiler should be able to detect it, and consequently ensure that the program is never executed. Such non-execution must be clearly distinguished from the empty trace, which contains no actions, and is easy to execute rather than impossible.

The user specification of a program is its most general description, describing only those traces which will be acceptable to the end users. The program itself is the most specific description: it will describe only those traces which record a possible complete behaviour of the program when executed. Correctness is inclusion ($\subseteq$) of the latter in the former. Even a single assertion, describing a set of possible states of the machine before or after execution of a program, can be interpreted as a set of traces, namely those which end (alternatively, which begin) in one of the states described by the assertion. A single state is described by an assertion which only that state satisfies. We exploit these interpretations to obtain a simple homogeneous algebra, with just a single sort.

Of course, there is good reason why distinct sorts, with different notations, are often used for these different kinds of description. For example, programs need to be expressed in some notation (a computable programming language), for which there exists an implementation of adequate efficiency and stability on the hardware available to execute the program. Specifications, on the other hand, are normally allowed to exploit the full expressive power of mathematics, including scientific concepts relevant to the real world environment of a particular application. A single state is often expressed as a tuple, describing the structure of the state of an

executing machine. Assertions are often restricted to Boolean terms that can be evaluated in the current state of the machine at run time. The simpler distinct notations generally used for each of these special cases are shorter than descriptions of general traces; they are more useful, more comprehensible and easier to reason with. But distinct notations tend to obscure or even accentuate differences between languages and theories. Disregard of syntactic distinctions is an essential preliminary to unification of the underlying semantics.

A denotational semantics is formalised as a collection of definitions of the basic commands of a programming language, and of the operators by which they are composed into programs. Each action occurrence in a trace is the execution of a basic command. There is also a basic command Id, interpreted as a program that has no action, because it does nothing. There are three binary operators: semicolon (;) standing for sequential composition, star ($*$) standing for concurrent composition, and $\vee$ standing for disjunction or choice. These form the signature of the algebra. In many cases, there are several definitions that will satisfy the same algebra, and some interesting alternatives will occasionally be indicated in the text. The main thread of development presents a model that is (where necessary) a compromise between simplicity of exposition and authenticity to application.

2.1 Basic Commands

$$\mathsf{Id} = \{\{\}\}$$

This is the command that performs no action.

Each of the other basic commands of the programming language is also defined as a set of traces. Each trace is a unit set, containing just a single occurrence of the action of executing the command. We will assume that there is an infinite set of possible executions of the command, which can occur in different programs, in different contexts and on different occasions. As a result, it is trivial to model resource allocation and disposal: each allocated resource is distinct, because its allocation was distinct from all allocations of all other resources.

2.2 Sequential Composition

$$P; Q = \{p \cup q \mid p \in P \text{ and } q \in Q \text{ and } p \times q \subseteq \mathsf{Seq}\}$$

where $\mathsf{Seq} = (\neg(\mathsf{Dep}^*))^\cup$, and $p \times q$ is the Cartesian product, Dep^* is the reflexive transitive closure of Dep and $(\cdot)^\cup$ denotes relational converse.

This definition states that a trace of $P; Q$ is the union of all the actions of some trace p of P with all the actions of another trace q of Q. Furthermore, there is no dependency (direct or indirect) of any action occurrence in p on any action occurrence in q. Such a dependency would certainly prevent the completion of p before the execution of q starts; and this should surely be an allowed method of implementation. But our definition also allows many other implementations, in which actions of q can occur concurrently with (or before) those of p , subject only to the given dependency constraint. This freedom of implementation

is widely exploited by the optimisations performed by the main compilers for today's programming languages. The strongest reasonable definition of sequential composition would replace the above definition of Seq by simply Dep^*. This requires that all actions of p precede all actions of q.

2.3 Concurrent Composition

$$P * Q = \{p \cup q \mid p \in P \text{ and } q \in Q \text{ and } p \times q \subseteq \mathsf{Par}\}$$

where $Par = \neg(\mathsf{Dep}^*) \cup (\neg(\mathsf{Dep}^*))^{\cup}$. Again, this is the weakest reasonable definition of concurrent composition. The condition on $p \times q$ rules out a class of impossible traces, where an event in p depends on an event in q, and vice-versa; such a dependency cycle would in practice end in deadlock, which would prevent the successful completion of the trace.

2.4 Choice

$$P \vee Q = P \cup Q$$

This is simplest possible definition of choice: $P \vee Q$ is executed either by executing P or by executing Q. The criterion of selection is left indeterminate. Other definitions of choice can be implemented by an operator (e.g., a conditional) that controls the choice by specifying when one of the operands will have no traces. The details are omitted here.

2.5 Galois Inverses

Galois inverses (of functions that distribute through big unions) are defined in the usual way for a complete lattice of sets.

1. $Q \mathbin{\leftarrow\!\!\!\circ} R = \bigcup(P \mid P; Q \subseteq R)$
2. $P \mathbin{\circ\!\!\!\rightarrow} R = \bigcup(Q \mid P; Q \subseteq R)$
3. $P \mathbin{—\!\!*} R = \bigcup(\{Q \mid P * Q \subseteq R\})$

The first of these inverses is a generalisation of Dijkstra's [8] weakest precondition $wp(Q, R)$. It is a description of the most general program P whose execution before Q will satisfy the overall specification R. The second is a version of the Back/Morgan specification statement [9, 10], the most general description of the program Q whose execution after P will satisfy the specification R. The third is very similar, except that P and Q are executed concurrently rather than sequentially. It is called "magic wand" in separation logic.

2.6 Iteration

$$P^* = \bigcup\{X \mid \mathsf{Id} \cup P \cup (X*; X*) \subseteq X\}.$$

This is Pratt's definition [11], using the Knaster-Tarski fixed-point theorem. The same technique can be used to define the more general concept of recursion, under the condition of monotonicity of the function involved.

3 Algebra and Logic

This section explains the algebraic properties of the listed operators, and gives hints why they are satisfied by the denotational definitions of the previous sections. The properties are mostly expressed as the familiar laws of associativity, commutativity, and distribution. The less familiar laws are those applicable to concurrency. In this section, the elements of the algebra are all sets; and it is convenient to denote them by lower case letters $p, q, r, \ldots$.

This section also relates the algebra to the familiar rules of a deductive and an operational semantics. It selects a subset of the equational and inequational algebraic laws, and expresses them in the form of proof rules, which can be derived from them, and from which the laws can themselves be derived. It gives algebraic definitions of the basic judgements of a deductive and of an operational semantics. It then shows how the selected laws can be derived from the semantic rules, and vice-versa. This is the same method that is used to show that natural deduction is just a logical form of presentation for the laws of Boolean algebra.

The Hoare triple $p\{q\}r$ says that if q is executed after p, then the overall effect will satisfy the specification r. It is therefore defined as the algebraic inequation $p; q \subseteq r$. The logical implication $p \Rightarrow p'$ between assertions is defined by the inequation $p; \mathsf{Id} \subseteq r$. Similarly, the Milner transition $r \xrightarrow{p} q$ means that one of the ways of executing r is to execute p first and then q. This is simply defined as exactly the same inequation $p; q \subseteq r$! The silent action τ is defined as Id. As a result, the basic operational rule of CCS ($p; q \xrightarrow{p} q$) follows just from the reflexivity of $\subseteq$. From these definitions, it is easy to show that the main structural rules of Hoare and Milner calculi are the same as rules of deduction, which are interderivable with a subset of the axioms in the algebra. Note that our rules for operational semantics are based on "big steps", and in general they differ from those used by Milner in his definition of CCS [5].

Our definitions also ignore the restrictions that are normally placed on the judgements of the two rule-based semantics. Hoare logic usually restricts p and r to be assertions, and Milner transitions (in a small-step semantics) usually restrict q to be a basic action. Furthermore, p and r are usually machine states, often represented by displaying the structure of the program that remains to be executed in the future (the continuation). These restrictions are fully justified by their contribution to simpler reasoning about programs and to greater efficiency of their implementation. However, the denotational model shows that the restrictions are semantically unnecessary, and we shall ignore them.

In summary, a significant part of the algebraic semantics, including the novel laws for concurrency, can be derived from the rules of the two rule-based semantics. Additional evidence is thereby given of the realism and applicability of the algebraic laws, and of their conformity to an already widely shared understanding of the fundamental concepts of programming.

3.1 Monotonicity

Theorem 1. *Sequential composition is monotonic with respect to set inclusion* $\subseteq$.

This is a simple consequence of the implicit existential quantifier in the definition of this operator (and of others).

Monotonicity of a binary operator is normally expressed as two proof rules

$$(a)\quad \frac{p \subseteq p'}{p;q \subseteq p';q} \qquad (b)\quad \frac{q \subseteq q'}{p;q \subseteq p;q'}$$

Using the properties of partial ordering, these rules are interderivable with the single rule

$$(c)\quad \frac{p \subseteq p' \quad p';q \subseteq r' \quad r' \subseteq r}{p;q \subseteq r}$$

When translated to Hoare triples, (a) and (b) give the two clauses of the familiar Rule of Consequence. When translated to Milner transitions, (c) gives a stronger version of the structural equivalence rule of process algebra. The strengthening consists in replacement of $=$ by $\subseteq$.

3.2 Sequential Composition

Theorem 2. ; *is associative and has unit* Id.

Proof of associativity does not depend on the particular definition of Seq, which can be an arbitrary relation between actions. It depends only on the fact that Cartesian product distributes through set union.

Associativity can be expressed in two complementary axioms

$$p;(q;r) \subseteq (p;q);r \qquad (p;q);r \subseteq p;(q;r).$$

Using monotonicity of ; and antisymmetry of $\subseteq$, the first of these can be expressed as a proof rule

$$(a)\quad \frac{p;q \subseteq s \quad s;r \subseteq r'}{p;(q;r) \subseteq r'}$$

Similarly, the second axiom is expressible as

$$(b)\quad \frac{p;s \subseteq r' \quad q;r \subseteq s}{(p;q);r \subseteq r'}$$

When translated to Hoare triples, (a) is the familiar rule of sequential composition. When translated to transitions, (b) gives a (less familiar, large-step) rule for sequential composition.

3.3 Concurrent Composition

Theorem 3. $*$ *is associative and commutative and has unit* Id. *Furthermore, it is related to sequential composition by the laws*

$(a1)$ $p; q \subseteq p * q$
$(a2)$ $q; p \subseteq p * q$
$(b1)$ $p; (q * r) \subseteq (p; q) * r$
$(b2)$ $(p * q); r \subseteq p * (q; r)$
(c) $(p * q); (p' * q') \subseteq (p; p') * (q; q')$

The proof of these four laws depends only on the fact that the Seq relation is included in Par, and that Par is commutative. When $*$ is commutative, $(a1)$ and $(a2)$ are equivalent. When ; and $*$ share the same unit, all the laws follow from (c). They are listed separately, to cater for possible alternative models.

These laws are known as exchange laws, by analogy with the similar interchange law of two-categories. They permit the interchange of ; and $*$ as major and minor connectives of a term. In category theory, the law is a weak equality, and holds only when both sides are defined. In our case, the law is a strong inequality, and the right hand side always includes the left.

The exchange laws formalise the principle of sequential consistency. They allow any formula containing only basic actions, connected by sequential and concurrent composition, to be reduced to a set of stronger forms, in which all concurrent compositions have been eliminated. Furthermore, any pair of basic commands, which appear directly or indirectly separated by ; in the original formula, appear in the same order in all the stronger interleavings. Of course many of the interleavings could turn out to be empty, because they violate dependency ordering; for example, the left hand side of $(a1)$ is empty if any action of p depends (indirectly, perhaps) on any action of q. The full strength of the principle of sequential consistency would require another axiom: that every formula is equal to the union of all the stronger and more interleaved forms derived from it. This would unfortunately be an infinite axiom set.

$(b2)$ is interderivable with the principle of local reasoning, the fundamental contribution of separation logic [3]. This is called the frame rule; it serves the same role as the rule of adaptation in Hoare logic; but it is more powerful and much simpler.

$$\frac{p; q \subseteq r}{(p * f); q \subseteq r * f.}$$

$(b1)$ is interderivable with one of the rules given in the operational semantics of CCS [5].

$$\frac{r \xrightarrow{p} q}{r * f \xrightarrow{p} q * f}$$

This rule is interpreted as follows. Suppose r is a process that can first do p and then behave as the continuation q. Then when r is executed concurrently

with a process f , the combination can also do p first, and then behave like the continuation q running concurrently with the whole of f.

(c) is interderivable with the main concurrency rule of concurrent separation logic (which is usually expressed with the operator $\|$ in place of $*$ between q and q'):

$$\frac{p; q \subseteq r \quad p'; q' \subseteq r'}{(p * p'); (q * q') \subseteq r * r'.}$$

This law expresses a principle of modular reasoning. A concurrent composition can be proved correct by proving properties of its operands separately.

When translated to transitions, (c) is the main concurrency rule of a process algebra like CCS. In a small-step semantics like that of CCS, $q * q'$ is defined as a basic action iff q and q' are an input and an output on the same channel; and it is then equal to Id (which we have identified with τ).

3.4 Units

Theorem 4. *Id is the unit of both sequential and parallel composition.*

From this, two weaker properties are selected for translation into rules.

(a) $p; Id \subseteq p$
(b) $Id; p \subseteq p.$

The rule derived from (a) is the Hoare rule for Id, and (b) is an operational rule for Id, which is not accepted as a rule of a small-step semantics.

3.5 Choice

Theorem 5. $\vee$ *is associative, commutative and idempotent. It is monotonic and increasing in both arguments. It also admits distribution in both directions by concurrent and by sequential composition. This last property can be expressed*

(a) $p; (q \vee q') \subseteq p; q \vee p; q'$
(b) $(p \vee p'); q \subseteq p; q \vee p'; q.$

As in previous examples, these properties are interderivable with standard rules. The rule derived from (a) gives the standard rule in Hoare logic for non-deterministic choice. (b) gives a very similar Milner rule, expressed in terms of big-step transitions.

3.6 Galois Adjoints

The algebraic properties of Galois inverses are well-known. For example, they are monotonic, and give an approximate inverse to an operator :

$$(q \leftharpoonup r); q \subseteq r \qquad p \subseteq q \leftharpoonup (p; q)$$

The equations that Dijkstra [8] gives as definitions of the operators of a programming language can all be derived as theorems. For example: the definition of sequential composition is

$$(p; q) \leftarrow\!\!\!\circ\ r = p \leftarrow\!\!\!\circ\ (q \leftarrow\!\!\!\circ\ r).$$

However, there seems to be no exact finite equational characterisation of concurrent composition in terms of its adjoint.

3.7 Iteration

Pratt [11] has given a complete axiomatisation of iteration by just three elegant algebraic equations. (1) is the familiar fixed point property. (2) is monotonicity. And (3) is $(p \rightarrow\!\!\!\circ\ p)^* = p \rightarrow\!\!\!\circ\ p$.

An obvious conclusion from this section is that an algebraic semantics of a programming language can be more powerful than a combination of both its deductive and its operational presentations, and simpler, by objective criteria than each of them separately. It is more powerful in the expression of more laws, and in their applicability to an unrestricted class of operands. To supply the missing laws, a rule-based semantics usually defines a concept of by equivalence, for example, in terms of contextual congruence or bisimulation. The equivalence theorems are often proved by induction over the terms of the language, sometimes jointly with induction over the length of a computation. The inductions are often simple and satisfying. The disadvantage of inductive proofs in general is that any extension to the language (or to its axioms) has to be validated by adding new clauses to every theorem previously proved by induction. This reduces the modularity of the proofs, and makes extension of the programming language more difficult than it is by the direct addition of new algebraic axioms. Of course, in an algebraic presentation, it is still highly desirable to prove consistency of the new axioms with the original model. Alternatively, a whole new model, may be required, and a new proof of all the original axioms. To avoid this, a general parameterised model may be helpful.

4 Conclusion

As well as enabling greater modularity, extensibility and reusability of semantic reasoning, the algebraic laws seem simpler (by objective criteria) than the rules of a deductive semantics, and also simpler than an operational semantics. The obvious conclusion is that algebra could play an expanded role in the exploration of the principles of programming.

I hope that this message will appeal to the hearts and stimulate the minds of Samson Abramsky and his many colleagues and admirers. They have made earlier and far deeper contributions to the unification of the semantics of programming than those reported here. The best outcome of this essay would be to make their results more widely accessible and more widely applicable to the practical problems of software engineering.

Acknowledgements. This essay reports the results of research conducted with many collaborators. Many of these results have been previously published [12–18].

References

1. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press (1977)
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
3. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)
4. Plotkin, G.D.: A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University (1981)
5. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
6. Hennessy, M.: Algebraic Theory of Processes. MIT Press (1988)
7. Baeten, J., Basten, T., Reniers, M.A.: Process Algebra: Equational Theories of Communicating Processes. Cambridge Tracts in Theoretical Computer Science, vol. 50. Cambridge University Press (2009)
8. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs (1976)
9. Back, R.J., Wright, J.: Refinement calculus: a systematic introduction. Springer (1998)
10. Morgan, C.: Programming from specifications. Prentice-Hall, Inc. (1990)
11. Pratt, V.R.: Action logic and pure induction. In: van Eijck, J. (ed.) JELIA 1990. LNCS, vol. 478, pp. 97–120. Springer, Heidelberg (1991)
12. 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. Commun. ACM 30(8), 672–686 (1987)
13. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice Hall (1998)
14. Wehrman, I., Hoare, C.A.R., O'Hearn, P.W.: Graphical models of separation logic. Inf. Process. Lett. 109(17), 1001–1004 (2009)
15. Hoare, T., Wickerson, J.: Unifying models of data flow. In: Software and Systems Safety - Specification and Verification, pp. 211–230 (2011)
16. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P.W., Petersen, R.L., Struth, G.: On locality and the exchange law for concurrent processes. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 250–264. Springer, Heidelberg (2011)
17. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program. 80(6), 266–296 (2011)
18. Hoare, T., van Staden, S.: The laws of programming unify process calculi. In: Gibbons, J., Nogueira, P. (eds.) MPC 2012. LNCS, vol. 7342, pp. 7–22. Springer, Heidelberg (2012)