Logic, Algebra, and Geometry at the Foundation of Computer Science

Tony Hoare[1,2], Alexandra Mendes[3,4], and João F. Ferreira[5]

[1] Microsoft Research, Cambridge, United Kingdom
[2] Cambridge University Computing Laboratory, Cambridge, United Kingdom
[3] HASLab, INESC TEC
[4] Department of Informatics, Universidade da Beira Interior, Covilhã, Portugal
[5] INESC-ID & Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal

Abstract. This paper shows by examples how the Theory of Programming can be taught to first-year CS undergraduates. The only prerequisite is their High School acquaintance with algebra, geometry, and propositional calculus. The main purpose of teaching the subject is to support practical programming assignments and projects throughout the degree course. The aims would be to increase the student's enjoyment of programming, reduce the workload, and increase the prospect of success.

Keywords: Algebra · Logic · Geometry · Teaching Formal Methods · Unifying Theories of Programming

1 Introduction

The Theory of Programming lies at the foundation of modern development environments for software, now widely used in industry. Computer Science graduates who understand the rationale of programming tools, and who have experience of their use, are urgently needed in industry to maintain the current rate of innovations and improvements in software products installed worldwide.

We put forward the following theses:

1. The fundamental ideas of the Theory of Programming were originally formulated by great philosophers, mathematicians, geometers and logicians, dating back to antiquity.
2. These ideas can be taught as an aid to practical programming throughout a degree course in Computer Science. The desirable initial level of Math for first-year CS students is that of High School courses in Algebra, Geometry and Propositional Logic.
3. The ideas should form the basis of a student-oriented Integrated Development Environment (IDE), needed to support students in understanding requirements, in designing solutions, in coding programs, in testing them, and in diagnosing and debugging their errors.

One of the goals of this paper is to contribute to the challenge posed by Carroll Morgan in [21]:

Invariants, assertions and static reasoning should be as self-evidently part of the introductory Computer Science curriculum as are types, variables, control structures and I/O in the students' very first programming language.
Can you help to bring that about?

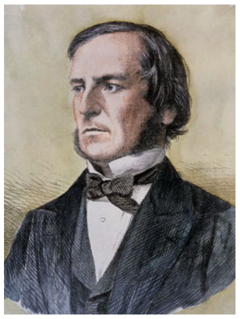
Paper structure. In this paper we provide examples of material that can be taught to first-year CS undergraduates. In Section 2, we introduce the underlying concepts of algebra and logic. These are then applied to the execution of computer programs: in Section 3 we discuss the familiar topic of sequential composition and in Section 4 we move on to concurrent composition. Section 4 includes material suitable for a more advanced and elective course in formal methods delivered at later stage in the syllabus, where we show how two familiar and widely used theories of programming can be unified. After presenting in Section 5 some related work, we conclude in Section 6, where we also briefly suggest directions for future work.

2 Algebra and Logic

This section introduces the underlying concepts of algebra and logic. The first subsection is entirely elementary, but it proves some essential theorems that will be used in later sections. The second subsection shows how familiar logical proof rules can be derived from the algebra. The third subsection introduces spatial and temporal reasoning about the execution of computer programs.

2.1 Boolean Algebra

Boolean Algebra, which is widely taught at the beginning of degree courses in Mathematics and in Philosophy, is doubly relevant in a Computer Science course, both for Hardware Design and for Program Development.


George Boole
(1815–1864)

Boolean Algebra is named for the nineteenth century mathematician George Boole (1815–1864). His father was a shoe-maker in Lincoln, where he attended primary school. His father died when he was aged 16, and he became the family breadwinner, working as a schoolmaster. At age 25 he was running a boarding school in Lincoln, where he was recognised as a local civic dignitary. He learnt mathematics from books lent to him by friendly mathematicians. At the age of 34, he was appointed as first Professor of Mathematics at the newly founded Queen's College in Cork. He published a number of articles in the humanities, and wrote several mathematical textbooks. But he is now best known for his logical investigations of the Laws of Thought [6], which he published in 1854 and where he proposed the binary algebraic operators *not*, *and*, *or*, and a binary comparison for predicates as the foundation for a deductive logic of propositions.

Disjunction. Disjunction is denoted as $\vee$ (read as 'or') and satisfies three axioms: it is associative, commutative, and idempotent. All three axioms are illustrated in the following proof.

Theorem 1. *Disjunction distributes through itself:*

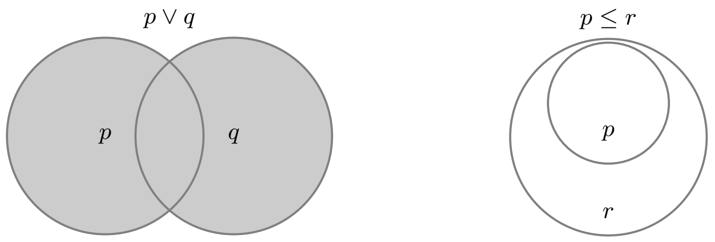
$$(p \vee q) \vee r = (p \vee r) \vee (q \vee r)$$

Proof.

$$\begin{aligned} RHS &= p \vee (r \vee (q \vee r)) && \text{by associativity} \\ &= p \vee ((q \vee r) \vee r) && \text{by commutativity} \\ &= p \vee (q \vee (r \vee r)) && \text{by associativity} \\ &= p \vee (q \vee r) && \text{by idempotence} \\ &= LHS && \text{by associativity} \end{aligned}$$

Corollary 1. *Rightward distribution (follows by commutativity).*

Geometry. Geometry is recognised in Mathematics as an excellent way of gaining intuition about the meaning and the validity of algebraic axioms, proofs, conjectures, and theorems. The relevant geometric diagrams for Boolean algebra are familiar as Venn diagrams. For example, Fig. 1a illustrates the Venn diagram for disjunction.



(a) Disjunction: p or q (b) Comparison: r is weaker than p

Fig. 1: Venn diagrams for disjunction and comparison.

Comparison (denoted as $\leq$). The most important comparison operator between terms of Boolean algebra is implication. It is written here as a simple less-than-or-equal sign ($\leq$). It is defined simply in terms of disjunction:

$$p \leq r \quad \text{is defined as} \quad r = p \vee r$$

The comparison may be read in many ways: that p implies r, or that p is stronger than r, or that r is weaker than p. The definition is illustrated by a Venn diagram showing containment of the stronger left side p by the weaker right side r (see Fig. 1b).

Disjunction is a weakening operator. An operator is defined as weakening if its result is always weaker than both of its operands. From Theorem 2 and Corollary 2 below, we conclude that the result of disjunction is always weaker than both of its operands. The proof of this again uses all three axioms.

Theorem 2. $p \ \leq \ p \vee r$

Proof.

$$\begin{aligned} p \vee r &= (p \vee p) \vee r && \text{by idempotence} \\ &= p \vee (p \vee r) && \text{by associativity} \end{aligned}$$

The theorem follows by definition of $\leq$.

Corollary 2. $p \ \leq \ r \vee p$ *(by commutativity)*

Henceforth, we omit brackets around associative operators and proofs of theorems that follow by commutativity.

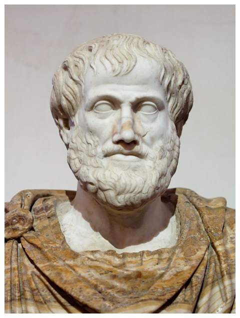
2.2 Deductive Logic

The axioms of algebra are restricted to single equations or comparisons between two algebraic terms. This makes algebraic reasoning quite simple, using only substitution of equals to deduce a new equation from two equations that have already been proved. The price of this simplicity is that proofs can get too long for comfort, and they can be quite difficult to find. To tackle these problems we need more powerful techniques, which are expressed as rules of logical deduction.

The Aristotelian Syllogism. A syllogism is a form of proof rule that has been taught for over two thousand years. It consists of two antecedents written above a line and one consequent written below the line. This says that any proof that contains both the antecedents can validly be extended by adding the consequent as its next line. A well-known example of a syllogism is:

$$\frac{\textit{All men are animals} \qquad \textit{All animals are mortal}}{\textit{All men are mortal}}$$

The use of syllogisms as a tool for reasoning can be dated back to the work of the ancient Greek philosopher Aristotle [32], who made a remarkable contribution to the history of human thought. He was the founder, director and a lecturer at a private academic institution in Athens. His lecture notes still survive. They deal with both the sciences and the humanities, and spanned almost the full range of human intellectual endeavour for the next two thousand years. The first application of syllogisms was probably in Biology, of which he is also recognised as the founding father. They are well adapted to deducing the consequences of his biological classifications.


Aristotle
(384–322 BC)

A proof rule in algebra rather than biology is given in the following theorem.

Theorem 3 (Proof by cases).

$$\frac{p \leq r \qquad q \leq r}{(p \vee q) \leq r}$$

Proof. Assuming the antecedents $r = p \vee r$ and $r = q \vee r$, we prove the consequent:

$$\begin{aligned} r &= r \vee r && \text{by idempotence} \\ &= (p \vee r) \;\vee\; (q \vee r) && \text{by substitution for each } r \\ &= (p \vee q) \;\vee\; r && \text{by Theorem 1} \end{aligned}$$

The conclusion follows by definition of $\leq$.

In this proof, the assumption of the antecedents of the rule is justified by the general embargo which forbids use of the rule until the antecedents have already been proved.

A validated proof rule can also be used backwards to suggest a structure and strategy for a proof of a desired conjecture which matches the conclusion of the rule. Then the task of proof can be split into subtasks, one for each of the antecedents. Success of this strategy requires that each antecedent is in some way simpler than the conclusion. For example in the rule for proof by cases, the conclusion has a disjunction $p \vee q$ where the antecedents only contain a single operand, either p or q. The backward use is widely adopted in the search for proofs by computer.

Partial orders. The well-known properties of an ordering in mathematics are usually defined by means of proof rules. The rules shown in the proof of Theorem 4 define the concept of a *partial order*. Each rule is proved by only one of the three axioms of disjunction. The first line shows how an axiom itself can be written as the consequent of a proof rule with no antecedents.

Theorem 4. *Comparison ($\leq$) is a partial order.*

Proof. Comparison is:

reflexive: $$\frac{}{p \leq p}$$ (by idempotence)

transitive: $$\frac{p \leq q \quad q \leq r}{p \leq r}$$ (by associativity)

antisymmetric: $$\frac{p \leq q \quad q \leq p}{p = q}$$ (by commutativity)

Covariance (monotonicity) of disjunction. Covariance is the property of an operator that if either of its operands is strengthened, its result is also strengthened (or stays the same). Such an operator is said to respect the ordering of its operands. Covariance justifies the use of the comparison operator $\leq$ for substitution of one formula in another, just like the familiar rule of substitution of equals.

Theorem 5. *Disjunction is covariant (monotonic) with respect to $\leq$, that is:*

$$\frac{p \leq q}{p \vee r \leq q \vee r}$$

Proof. From the antecedent, transitivity of $\leq$, and weakening of disjunction, we have:

$$p \leq q \leq q \vee r \quad \textit{and} \quad r \leq q \vee r$$

The consequent follows by the proof rule by cases.

Covariance is also a formal statement of a common principle of engineering reasoning. Suppose you replace a component in a product by one that has the same behaviour, but is claimed to be more reliable. The principle says that the product as a whole will be made more reliable by the replacement; or at least it will remain equally reliable. If the product is found in use to be less reliable than it was before the replacement, then the claimed extra reliability of the component is disproved.

2.3 Spatio-Temporal Logic

A theorem of Boolean algebra is used to state an universal truth, which remains true everywhere and forever. The ideas of temporal logic were explored by Aristotle and his successors, for reasoning about what may be true only during a certain interval of time (its duration), and in a certain area of space (its extent). A proposition describes all significant events occurring within its given duration and within its given extent. However, the logic does not allow any mention of a numeric measurement of the instant time or the point in space at which an event occurs. Thus a proposition in the logic can be true of many different regions of space and time.

Temporal logic was widely explored by philosophers and theologians in the middle ages. William of Occam (1287–1347), a Franciscan friar studying philosophy at Oxford, is considered to be one of the major figures of medieval thought. Unfortunately he got involved in church politics. He antagonised the pope in Rome, and was excommunicated from the Church in 1328. This was believed to condemn him to an eternity in hell. Fortunately, he was reprieved thirty years later. Occam's book on Logic, *Summa Logicae* (1323) included familiar operators of Boolean Algebra, augmented by operators that apply to propositions of spatial and temporal logic [25]. They include sequential composition p *then* q, written here with semicolon $(p; q)$, and p *while* q, written here with a single vertical bar $(p \mid q)$.

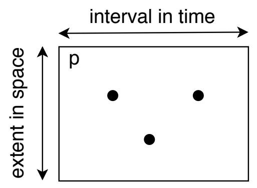
William of Occam (1287-1347)

Geometric Diagrams. The propositions of Occam's spatio-temporal logic are best illustrated by two-dimensional geometric diagrams, with one axis representing time and the other representing space. As shown in Fig. 2a, the region described by a proposition p is represented by a rectangular box with the name p written in the top left corner. The box contains a finite set of discrete points, representing all the events that occurred in the region. The horizontal edges of the box represent the interval of time within which those events occur. The vertical edges represent the locations in space where the events occur. Fig. 2b illustrates these two dimensions.

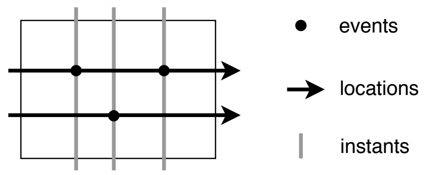
In Cartesian plane geometry, each point lies at the intersection of a vertical coordinate, shown here in gray, and a horizontal coordinate shown as a black arrow (Fig. 2b). Each point can therefore be identified by a pairing of a horizontal coordinate with a vertical coordinate. But the geometry shown here differs from this in that not all coordinate positions are occupied by a point. This is because in the description of the real world many or most coordinates are occupied by no event. Our diagrams are comparable to the output of a multiple pen recorder, for example the seismograms of geology and the cardiograms. Each horizontal line is the output of a single pen recording the value given by sensors in different locations. The events record significant changes in the value of the sensor.

In computer applications, the horizontal lines stand uniquely for a variable held in the memory of the computer. The events on a line represent assignments of potentially new values to the variables. The vertical lines are often drawn in later to explain a group of significant changes made simultaneously in many variables.

The sequential composition of p and q, denoted as p;q, starts with the start of p and ends with the end of q. Furthermore, q starts only when p ends. Fig. 3a shows a diagram of the sequential composition of p and q. As before, the box is named by the term written in the top left corner. Every event in the composition is inside exactly one of the two operands. The vertical line between p and q is shared by both of them. It shows that time intervals of the two operands are immediately adjacent in time. The interval for the result is the set union of the interval for p and the interval for q .

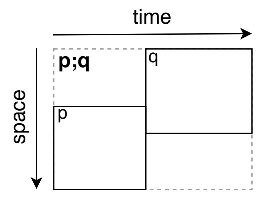


(a) Region describing p, with points representing events.

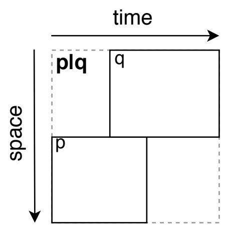


(b) Events occur within an interval in a given location in space.

Fig. 2: Propositions as two-dimensional geometric diagrams.



(a) Sequential composition.



(b) Concurrent composition.

Fig. 3: Spatio-temporal diagrams for sequential composition and concurrent composition.

The boxes with dotted edges at the corners of the $p; q$ contain no events. They are padding, needed to draw the result of composition as a box. To represent this padding we introduce an algebraic constant $\square$, read as '*skip*'.

The concurrent composition of p and q, denoted as $p \mid q$ and read as p *while* q, starts with the start of both p and q and ends with the end of both of them. Its duration is the maximum of their durations. Fig. 3b shows a diagram of the concurrent composition of p and q. Its extent in space is the disjoint union of the extents of the operands. This means that no location can be shared by the concurrent components This embargo is the characteristic of O'Hearn's separation logic [23, 24, 30], which protects against the well-known problem of races in concurrent programs.

3 Sequential Composition

The algebraic axioms for sequential composition are:

- Sequential composition is associative and has the unit $\square$
- Sequential composition distributes through disjunction (both leftward and rightward):

$$p; (q \vee q') = p; q \vee p; q' \qquad \text{and} \qquad (q \vee q'); p = q; p \vee q'; p$$

Distribution justifies giving sequential composition a stronger precedence than disjunction. The associativity of sequential composition is evident from its diagram, and so is the unit law.

We now show how the algebraic axioms can be used to prove some rules.

Theorem 6 (Proof rule for sequential composition).

$$\frac{p; q \leq m \qquad m; r \leq t}{p; q; r \leq t}$$

Proof. Assuming the antecedents (1) $m = p;q \vee m$ and (2) $t = m;r \vee t$, we prove the consequent:

$$\begin{array}{ll} t = (p;q \vee m);r \vee t & \text{substitute (1) in (2)} \\ t = p;q;r \vee (m;r \vee t) & ;\text{ distributes through } \vee \\ t = p;q;r \vee t & \text{substitute back by (2)} \end{array}$$

This proof rule is used for decomposing its consequent into two parts, each of which has only three operands instead of four. Each antecedent is in this way simpler than the consequent, whose proof can therefore be constructed by divide and conquer.

Rules of Consequence. The following corollaries are a consequence of Theorem 6.

Corollary 3.

$$\frac{p \le m \qquad m;r \le t}{p;r \le t}$$

Corollary 4.

$$\frac{p;q \le m \qquad m \le t}{p;q \le t}$$

Proof. Corollary 3: by substitution of q by $\square$. Corollary 4: by substitution of r by $\square$.

3.1 Hoare Triples

Consider the proposition $p;q \le r$. It means that if p describes the interval from the start of r to the start of q, and q describes the interval from the end of p to the end of r, then r correctly describes the whole of $p;q$. This is the intended meaning of the Hoare triple [14]. Therefore, we define:

$$\{p\}\ q\ \{r\} \overset{\text{def}}{=} p;q \le r$$

This definition allows p and r to be arbitrary programs — a generalisation of the original formulation of Hoare logic, in which p and r are required to be assertions.

3.2 Verification Rules for Sequential Composition

By substitution of the definition of triple into the proof rule for sequential composition (Theorem 6), we obtain the Hoare rule for sequential composition:

$$\frac{\{p\}\ q\ \{m\} \qquad \{m\}\ r\ \{t\}}{\{p\}\ q\ ;\ r\ \{t\}}$$

From Corollaries 3 and 4, we obtain the Hoare Rules of Consequence:

$$\frac{p \le m \qquad \{m\}\ r\ \{t\}}{\{p\}\ r\ \{t\}} \qquad\qquad \frac{\{p\}\ q\ \{m\} \qquad m \le t}{\{p\}\ q\ \{t\}}$$

3.3 Milner Transition

Robin Milner defined CCS [19], a theory of programming which is now widely used in specifying how an implementation should generate a single execution of a given program r. The Milner transition defined here, and denoted $r \xrightarrow{p} q$, states that r can be executed by executing p first, saving q as a continuation for subsequent execution. (Other executions may begin with an initial step different from p). But this is exactly the meaning of the same comparison that we used to define the Hoare triple. We thus define:

$$r \xrightarrow{p} q \overset{\text{def}}{=} p; q \leq r$$

Thus the two calculi are identical, and all theorems of one can be translated letter by letter from the corresponding theorem of the other. For example, in Milner's notation the rule for sequential composition and its corollaries are

$$\frac{r \xrightarrow{p} m \qquad m \xrightarrow{q} t}{r \xrightarrow{p;q} t}$$

$$\frac{m \leq r \qquad m \xrightarrow{q} t}{r \xrightarrow{q} t} \qquad\qquad \frac{r \xrightarrow{p} m \qquad t \leq m}{r \xrightarrow{p} t}$$

These corollaries play the role of the structural equivalence, which Milner introduced into the definition of concurrent programming languages (with $\equiv$ replaced by $\leq$) [20].

4 Concurrent Composition

Concurrent composition has the same laws as sequential composition. An additional *interchange axiom* permits a concurrent program to be executed sequentially by interleaving. The algebraic axioms are:

- Concurrent composition is associative and has unit $\Box$
- Concurrent composition distributes through disjunction
- Interchange axiom: $(p \mid q); (p' \mid q') \leq (p; p') \mid (q; q')$

We omit the commonly cited commutativity law for concurrency since it can be introduced later, whenever needed. The interchange law gets its name because it interchanges operators and variables when passing from one side of the comparison to the other. Note how the RHS and LHS differ by interchange of operators (; interchanged with $|$) and of operands (p' with q).

4.1 Interchange

The two following elementary corollaries of interchange show that a concurrent composition can be strengthened by sequential execution of its operands in either order:

$$\begin{array}{ll} p; q' \leq p \mid q' & \text{by interchange with } p' = q = \Box \\ q; p' \leq p' \mid q & \text{similarly, with } q' = p = \Box \end{array}$$

From these two properties and the proof rule by cases, we obtain:

$$p; q \vee q; p \;\leq\; p \mid q$$

This means that concurrent composition is weaker than the disjunction of these alternative orderings. We will now show by example that the interchange law generalises this interleaving to operands containing any number of operators.

We start with what are known as small interchange laws.

Theorem 7 (Small interchange laws).

$$\begin{array}{ll} p; (p' \mid q') \leq (p; p') \mid q' & q \;= \Box \\ q; (p' \mid q') \leq p' \mid (q; q') & p \;= \Box \\ (p \mid q); q' \leq p \mid (q; q') & p' = \Box \\ (p \mid q); p' \leq (p; p') \mid q & q' = \Box \end{array}$$

Proof. All four are proved from the interchange axiom, by substitution of $\Box$ for a different variable.

The above six corollaries are called frame laws in separation logic. They adapt the interchange law to cases with just two or three operands. Successive application of the frame laws can strengthen any term with two or three operands to a form not containing any concurrency. The following is an example derivation:

$$p; q; q' \leq (p \mid q); q' \leq p \mid (q; q')$$

4.2 Basic Principle of Concurrent Programming

We now show how to interleave longer strings. Let x,y,z,w,a,b,c,d be characters representing single events. Let us omit ";" in strings except for emphasis. Thus:

$$xyzw = x; y; z; w$$

The interchange law itself extends this principle to arbitrary terms, with many concurrent compositions, as the following example shows:

$abcd \mid xyzw$	is the RHS of interchange
$\geq (a; bcd) \mid (xy; zw)$	associativity (twice)
$\geq (a \mid xy); (bcd \mid zw)$	interchange
$\geq (a \mid x; y); (b; cd \mid zw)$	associativity (twice)
$\geq (a \mid x); y; (b \mid zw); cd$	frame laws (twice)
$\geq x\, a\, yz\, b\, w\, cd$	similarly

In the first line of this derivation, the characters of the left operand of concurrency have been highlighted; and the same characters are highlighted in subsequent lines. This conveys the important intuition that the order of characters in each sequential substring is preserved throughout. The same applies to the original right operand. Furthermore, each line splits some of the substrings of the previous line into two substrings. When all the highlighted substrings are of length 1, the first corollary can eliminate the concurrency. This shows that any chain of calculation using the interchange law must terminate.

A basic principle of concurrent programming states that every concurrent program can be simulated by a sequential program. Without this principle, it would have been impossible to exploit concurrency in general-purpose libraries and class declarations. The principle was proved for Turing machines by the design of a normal sequential Turing machine that could interpret any program run by multiple machines [27]. Our result is that any concurrent program can be translated by algebraic transformations for execution by a purely sequential machine. A direct algebraic proof is much simpler than a proof by interpretation. The result is also more useful because it can be applied to arbitrary sub-terms of a term. Thus the explosive increase in length of most reductions to normal form can generally be avoided.

4.3 Unifying Theories of Concurrency

The basic concurrency rule of separation logic was formulated by Peter O'Hearn in Hoare Logic. When translated to our algebraic notation it gives the following proof rule.

Interchange Rule (O'Hearn).

$$\frac{p;q \le r \qquad p';q' \le r'}{(p \mid p');(q \mid q') \le (r \mid r')}$$

His frame rule similarly translates to one of the frame laws of Theorem 7.

Just as the sequential rule is derived from the sequential axioms in section 3, the Interchange Rule is derivable from the Interchange Axiom.

Theorem 8. *The Interchange Axiom implies the Interchange Rule.*

Proof. Assume the antecedents of the interchange rule:

$$p;q \le r \quad \text{and} \quad p';q' \le r'$$

$$(p \mid p');(q \mid q') \le (p;q) \mid (p';q')$$

Covariance of $\mid$ twice: $(p;q) \mid (p';q') \le (r \mid r')$ and transitivity of $\le$

$$(p \mid p');(q \mid q') \le (r \mid r')$$

Conclusion: $$\frac{p;q \le r \qquad p';q' \le r'}{(p \mid p');(q \mid q') \le (r \mid r')}$$ the interchange rule

Surprisingly, the implication also holds in the reverse direction.

Theorem 9. *The Interchange Rule implies the Interchange Axiom.*

Proof. We start by assuming the interchange rule. Since it is a general rule, we can replace consistently all occurrences of each of its variables by anything we like.

$$\frac{p;q \leq r \qquad p';q' \leq r'}{(p \mid p');(q \mid q') \leq (r \mid r')} \qquad \text{replace } r \text{ by } p;q \text{ and } r' \text{ by } p';q'$$

$$\frac{p;q \leq p;q \qquad p';q' \leq p';q'}{(p \mid p');(q \mid q') \leq (p;q \mid p';q')} \qquad \text{both antecedents are true by reflexivity of } \leq$$

Conclusion: $(p \mid p');(q \mid q') \leq (p;q) \mid (p';q')$ the interchange axiom

Summary. We have extended to concurrency the unification between Hoare Triples and Milner Transitions that was achieved for sequentiality in section 3.

Theorem 10. *The following three rules are logically equivalent.*

$$\frac{p;q \leq r \qquad p';q' \leq r'}{(p \mid p');(q \mid q') \leq (r \mid r')} \qquad \textit{The Interchange Rule}$$

$$\frac{\{p\}\ q\ \{r\} \qquad \{p'\}\ q'\ \{r'\}}{\{(p \mid p')\}\ q \mid q'\ \{(r \mid r')\}} \qquad \textit{Translated to Hoare Triples}$$

$$\frac{r \xrightarrow{p} q \qquad r' \xrightarrow{p'} q'}{(r \mid r') \xrightarrow{(p|p')} (q \mid q')} \qquad \textit{Translated to Milner transitions}$$

The third rule is just the rule for concurrency in Milner's CCS, as formulated in the so-called 'big-step' version of operational semantics. It is interpreted as stating:

> To execute a concurrent composition of two sequential operands, split each operand into two sequential parts. Then start by executing the first part of both operands concurrently, and conclude by executing the second parts.

The unification of two widely accepted theories of programming is presented as strong evidence that our algebraic axioms are actually applicable to familiar programming languages implemented on computers of the present day. Many interpreters and compilers for programming languages are specified by an operational semantics expressed as Milner Transitions. Most program analysers and proof tools for sequential languages follow a verification semantics expressed as Hoare Triples. Many papers in the Theory of Programming prove the consistency between these two 'rival' theories for particular languages. Algebra unifies the theories, by proofs which could be understood or even discovered (under guidance) by CS students in their practical programming courses.

5 Related Work

This section surveys evidence for the validity of the three theses listed in the Introduction.

1. The biographies in this paper of Aristotle, Boole, and Occam are only a small selection of those who have contributed to the basic ideas of Computer Science, long before computers were available to put them into practice. Further examples are Euclid and Descartes for Geometry, Al-Khawarismi and Leibniz for Algebra, and Russel and Gödel for Logic. Their biographies may be found in Wikipedia. More recent pioneers are treated in [8].
2. Considerable experience has been accumulated of the effectiveness of teaching the Theory of Programming as part of practical degree courses in Computer Science. For example, in [29], the authors show how teaching concurrency and verification together can reinforce each other and enable deeper understanding and application. They suggest that concurrency should be taught as early as possible and they introduce a new workflow methodology that is based on existing concurrency models (CSP, π-calculus), on the model checker FDR that generates counter-example traces that show causes of errors, and on programming languages/libraries (occam-π, Go, JCSP, ProcessJ) that allow executable systems within these models.
   Another interesting example is the experimental course in "(In-)Formal Methods" [21], where invariants, assertions, and static reasoning are introduced. The author argues that the ideal place for an informal-methods course is the second half of first year, because at that point students already understand that "programming is easy, but programming correctly is very hard".
   Further proposals to introduce invariants and assertions as part of the introductory Computer Science curriculum, even at pre-university level, are presented in [10] and [11]. In [10], a programme focused on algorithmic problem solving and calculational reasoning is proposed. In [11], an experiment is presented where students specify algorithmic problems in Alloy [17] and reason about problems in an algebraic and calculational way. It has been argued that students seem to prefer and understand better calculational proofs [9]. Calculational proofs are commonly used in the functional programming community to demonstrate algorithm correctness [4, 16]. Recent tool support shows that this style can have impact in practical functional

programming [33]. An application of relational calculation to software verification is presented in [26], illustrated with a case study on developing a reliable FLASH filesystem for in-flight software. It combines the pragmatism of Alloy [17] with the Algebra of Programming presented in [5].

3. The introduction of formal methods in practical programming has accelerated in recent years. Regarding practical verification, there have been several attempts at building languages and systems that support verification, providing the ability to specify preconditions, postconditions, assertions, and invariants. ESC/Java [12] and Spec# [3] build on existing languages, Java and C#, respectively. Dafny [18] is a programming language with built-in specification constructs. The Dafny static program verifier can be used to verify the functional correctness of programs. Dafny has been extensively used in teaching. Whiley [28] is a programming language designed from scratch in conjunction with a verifying compiler. SOCOS [2] is a programming environment that applies Invariant Based Programming [1], a visual and practical program construction and verification methodology. The Java+ITP [31] was used as a teaching tool at the University of Illinois at Urbana-Champaign to teach graduate students and seniors the essential ideas of algebraic semantics and Hoare logic. A recent case of success in industry is Infer[6] [7], a static analyzer based on separation logic [30] adopted and being developed by Facebook. Infer has been used in a 4th-year MEng and MSc course on separation logic at the Department of Computing, Imperial College London[7].

6 Conclusion

We hope that this article has contributed to the challenge posed by Carroll Morgan that we mentioned in the Introduction. We also hope to have made the case that current achievements in teaching sequential programming can be extended to concurrent programming.

The theory has been further extended to object oriented programming in [15]. These extensions will require new textbooks and extension and combination of existing tools. The creation of an environment that effectively combines the experience and tools already available is an open challenge. Ideally, the environment should allow students to work at different levels of abstraction and should unify interfaces and techniques from existing tools, such as Alloy Analyzer [17] and Isabelle/UTP [13]. Since this environment is to be used in a teaching environment, we do not have the problem of scale; however, feedback must be given quickly to students (and preferably in a graphical form). The approach described in [29] is an excellent example of how a model-checker for concurrency can be integrated with a testing tool. We believe it would be fruitful if tool-builders and users adopted a similar approach, integrating their tools and ideas into this system and other rival verification platforms. Tools such as the theorem prover Lean [22] seem to provide a promising basis for further developments.

[6] Infer static analyzer website: `https://fbinfer.com`

[7] Course link: `https://vtss.doc.ic.ac.uk/teaching/InferLab.html`

References

[1] Ralph-Johan Back. “Invariant based programming: basic approach and teaching experiences”. In: *Formal Aspects of Computing* 21.3 (2009), pp. 227–244.

[2] Ralph-Johan Back, Johannes Eriksson, and Linda Mannila. “Teaching the construction of correct programs using invariant based programming”. In: *Proc. of the 3rd South-East European Workshop on Formal Methods.* 2007.

[3] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. “The Spec# programming system: An overview”. In: *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices.* Springer. 2004, pp. 49–69.

[4] Richard Bird. *Pearls of functional algorithm design.* Cambridge Univ. Press, 2010.

[5] Richard Bird and Oege De Moor. “The algebra of programming”. In: *NATO ASI DPD.* 1996, pp. 167–203.

[6] George Boole. *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities.* Dover Publications, 1854.

[7] Cristiano Calcagno and Dino Distefano. “Infer: An automatic program verifier for memory safety of C programs”. In: *NASA Formal Methods Symposium.* Springer. 2011, pp. 459–465.

[8] Martin Davis. *Engines of Logic: Mathematicians and the Origin of the Computer.* WW Norton & Co., Inc., 2001.

[9] João F. Ferreira and Alexandra Mendes. “Students’ feedback on teaching mathematics through the calculational method”. In: *2009 39th IEEE Frontiers in Education Conference.* IEEE. 2009, pp. 1–6.

[10] João F. Ferreira, Alexandra Mendes, Roland Backhouse, and Luís S Barbosa. “Which mathematics for the information society?” In: *International Conference on Technical Formal Methods.* Springer. 2009, pp. 39–56.

[11] João F. Ferreira, Alexandra Mendes, Alcino Cunha, Carlos Baquero, Paulo Silva, Luís Soares Barbosa, and José Nuno Oliveira. “Logic training through algorithmic problem solving”. In: *International Congress on Tools for Teaching Logic.* Springer. 2011, pp. 62–69.

[12] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. “Extended static checking for Java”. In: *ACM Sigplan Notices* 37.5 (2002), pp. 234–245.

[13] Simon Foster, Frank Zeyda, and Jim Woodcock. “Isabelle/UTP: A mechanised theory engineering framework”. In: *International Symposium on Unifying Theories of Programming.* Springer. 2014, pp. 21–41.

[14] Charles Antony Richard Hoare. “An axiomatic basis for computer programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580.

[15] Tony Hoare and Jim Woodcock. “A calculus of space, time and causation: its Algebra, Geometry, and Logic”. In: *International Symposium on Unifying Theories of Programming.* To be submitted. Springer, 2019.

[16] Graham Hutton. *Programming in Haskell.* Cambridge Univ. Press, 2016.

[17] Daniel Jackson. “Alloy: a lightweight object modelling notation”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.2 (2002), pp. 256–290.

[18] K Rustan M Leino. “Dafny: An automatic program verifier for functional correctness”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning.* Springer. 2010, pp. 348–370.

[19] Robin Milner. “A calculus of communicating systems”. In: *LNCS* 92 (1980).

[20] Robin Milner. *Communicating and mobile systems: the pi calculus.* Cambridge Univ. Press, 1999.

[21] Carroll Morgan. “(In-) Formal Methods: The Lost Art”. In: *Engineering Trustworthy Software Systems.* Springer, 2016, pp. 1–79.

[22] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. “The Lean theorem prover (system description)”. In: *International Conference on Automated Deduction.* Springer. 2015, pp. 378–388.

[23] Peter O’Hearn. “Resources, concurrency, and local reasoning”. In: *Theoretical computer science* 375.1-3 (2007), pp. 271–307.

[24] Peter O’Hearn, John Reynolds, and Hongseok Yang. “Local reasoning about programs that alter data structures”. In: *International Workshop on Computer Science Logic.* Springer. 2001, pp. 1–19.

[25] William of Ockham. *Ockham’s theory of propositions : part II of the Summa logicae, translated by Alfred J. Freddoso and Henry Schuurman.* University of Notre Dame Press, 1980.

[26] José N Oliveira and Miguel A Ferreira. “Alloy meets the algebra of programming: A case study”. In: *IEEE Transactions on Software Engineering* 39.3 (2012), pp. 305–326.

[27] Christos H Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[28] David J Pearce and Lindsay Groves. “Whiley: a platform for research in software verification”. In: *International Conference on Software Language Engineering.* Springer. 2013, pp. 238–248.

[29] Jan B Pedersen and Peter H Welch. “The symbiosis of concurrency and verification: teaching and case studies”. In: *Formal Aspects of Computing* 30.2 (2018), pp. 239–277.

[30] John C Reynolds. “Separation logic: A logic for shared mutable data structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* IEEE. 2002, pp. 55–74.

[31] Ralf Sasse and José Meseguer. “Java+ ITP: A verification tool based on Hoare logic and algebraic semantics”. In: *Electronic Notes in Theoretical Computer Science* 176.4 (2007), pp. 29–46.

[32] Robin Smith. *Prior analytics.* Hackett Publishing, 1989.

[33] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. “Theorem proving for all: equational reasoning in liquid Haskell (functional pearl)”. In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell.* ACM. 2018, pp. 132–144.