From Algebra to Operational Semantics

C.A.R. Hoare He Jifeng Augusto Sampaio

Abstract

Gordon Plotkin has shown how the operational semantics of a programming language can be presented in an elegantly structured style. Robin Milner has pioneered the use of various kinds of bisumulation to obtain a range of more abstract semantics of a language in the form of algebraic laws. The laws can then be used more widely in practice to reason about specification, to transform designs and to optimise programs.

In tribute to Robin Milner, we reverse the direction of the derivation. Starting with a sufficiently powerful collection of algebraic laws, we prove the correctness of an operational interpreter of the same language. The proof takes full advantage of algebraic reasoning based on the laws. The technique is illustrated on a sequential non-deterministic language suggested by Dijkstra.

1 Introduction

Professional practice in a mature engineering discipline is based on relevant scientific theories, usually expressed in the language of mathematics. A mathematical semantics for a programming notation aims to provide a scientific basis for specification, design and implementation of computer programs expressed in that notation. The goal is to improve the cost, quality, and duration of software engineering projects. Unfortunately, in the early stages of a new science, such worthy goals are sometimes obscured by irrelevant factors such as notational variations, which divide the research community into rival schools, obscure the proper structuring of the subject into its branches, and delay the general progress of knowledge by effective collaboration of a large body of scientists.

The study of program semantics may be rationally structured into three main branches, differentiated by their style of presentation. The *denotational* style relates each program to a description of its observable properties and behaviour when executed. This style is most directly applicable to capture of requirements, and the correctness of specifications and designs, even before the program has been written. The process languages CSP and occam have been given a semantics in this style by Bill Roscoe and his colleagues at Oxford [2, 4, 5]. The *algebraic* style characterises a programming language by a somewhat complete collection of equations which are postulated to hold between programs. These can be used directly to reason about specifications, to transform designs, and to optimise programs for efficient execution. The process language ACP [1] has been defined primarily in algebraic

style by Jan Bergstra and his colleagues in Amsterdam. The *operational* style [11] gives meaning to a program by showing how its execution can be split into a sequence of atomic steps performed by a simple abstract mechanism. This gives an essential insight into the efficiency and even the computability of the language. An operational semantics is the starting point for the process notations CCS [9], designed and explored by Robin Milner and his colleagues at Edinburgh.

One of the main inspirations of scientific research is the pursuit of simplicity. And simple theories are also easier to use in engineering practice. So the notations selected to illustrate a given semantic style will naturally be chosen to simplify their semantic treatment in that style. The selection may be subtly or deeply different from a selection made to optimise definition in a different style. In natural science, such conflicts would be subjected to the decisive test of experiment. In computing science, no such test seems possible; and to a superficial view, differences in notation are taken as a symptom of the immaturity of the subject; meanwhile, practising software engineers continue to use notations with no semantics or underlying theory at all.

The best way to resolve conflicts which are not amenable to experimental test is to prove that they do not exist. For example, the physical theory of gravitation may be presented in terms of Newtonian force acting at a distance, or in field theory, or by Einsteinian geodesics; and all of these are provably equivalent. In mathematics, a topology may be defined equivalently by a family of sets, by a neighbourhood system, or by a closure operator. Scientist and mathematicians will be equally familiar with all approaches, and will use whichever presentation is easiest for the purpose in hand. (Of course, this does not prevent arguments about teaching syllabus design.)

To follow the example of mature disciplines, any programming theory recommended for practical use should be presentable in many different styles, thereby combining their individual distinctive advantages; and the theorist should guarantee that they are equivalent. Ideally, all the presentations should be equally simple, and even the proof of their consistency should be accessible and elegant. An inspiring example is the correspondence proved between CCS modulo bisimulation and the Hennessy–Milner modal μ-calculus. In tribute to Robin Milner, this paper makes a similar connection between an algebraic and an operational style of presentation.

The notation chosen is simple and non-controversial. It is essentially the sequential non-deterministic language introduced by Dijkstra to explore the Discipline of Programming [3]. It already has a denotational semantics in terms of predicate transformers. It already has an algebraic semantics very similar to that of the relational calculus. It is not difficult to write down the obviously intended operational semantics; the only challenge is to prove its correctness in an elegant fashion, one which may be generalised to more complex languages in due course.

In Section 2, we summarise a selection of the algebraic laws that are presented as the algebraic definition of the language. These laws support inequational reasoning, using a partial order, representing a refinement or improvement relation. The easiest way to derive an operational semantics is to show that the final state of each operational step is an improvement on its initial state. Because improvement is transitive, the final state of

the whole computation (i.e. its result) will be better than its initial state, the one which contains the whole program loaded for execution. This approach is worked out in Section 3.

Unfortunately this approach guarantees only partial correctness, ignoring both deadlock and non-termination. The proof of total correctness is given in Section 4, using an approach that has often been used in proof of the correctness of an implementation of functional programming languages. First, the executing mechanism is defined by means of an interpreter written in the language itself. This is an obviously isomorphic presentation of the operational semantics. Then the interpretation of the text of any program is proved to be equal to the meaning of the program itself (or perhaps an improvement of it). The proof is conducted exclusively in the higher conceptual level of the programming language, avoiding more operational modes of reasoning. A similar technique has been applied in the ProCoS project to prove the correctness of a translator to machine code [7].

2 The Language and its Algebraic Semantics

The programming language contains the following operators:

$\bot$	abort
II	skip (do nothing)
$x, y, \ldots, z := e, f, \ldots, g$	multiple assignment
$P; Q$	sequential composition
$P \sqcap Q$	non-determinism
$P \lhd b \rhd Q$	conditional: if b then P else Q
$\mu X \bullet P$	recursive program X with body P

In addition to the above, the following operators will be used for reasoning about the correctness of the operational semantics:

$\mathbf{var}\, v$	declaration introducing variable v
$\mathbf{end}\, v$	end the scope of v
$b_\bot$	assertion: if b then II else $\bot$
$\Box_{i \in S}\, b_i \rightarrow p_i$	guarded command set

Usually, guarded command sets allow the expression of arbitrary non-determinism of guarded commands: one, many or even none of the guards may be satisfied. Here we will use a guarded command set in a more restrictive way: exactly one of the guards will be satisfied at a given state. A formal way to express this fact is to require that the latter operator above satisfy

$\vee_{i \in S}\, bi \;=\; true$	conditions are exhaustive
$\forall i, j \in S \mid i \neq j \bullet bi \wedge bj = false$	conditions are pairwise disjoint

These two conditions will be assumed in any future use of the guarded command set operator.

The algebraic semantics of our simple programming language is given by a set of laws (equations and inequations). In the reminder of this section we present a selection of algebraic laws which will be used to prove the correctness of an operational semantics for our simple language. A *complete* set of laws for Dijkstra's language is given in [8][1]. To easy future references, we associate both a number and a name with each law.

skip, Abort and Sequential Composition

The execution of II always terminates and leaves everything unchanged; therefore to precede or follow a program by II does not change its effect.

Law 2.1 $(II; P) = P = (P; II)$ $\langle ;\text{-}II \text{ unit}\rangle$

To precede or follow a program P by the command $\bot$ results in abortion.

Law 2.2 $(\bot; P) = \bot = (P; \bot)$ $\langle ;\text{-}\bot \text{ zero}\rangle$

Sequential composition is associative.

Law 2.3 $P; (Q; R) = (P; Q); R$ $\langle ; \text{assoc}\rangle$

Non-determinism

The operator $\sqcap$ satisfies many useful laws: it is associative, commutative, idempotent and has $\bot$ as zero. In addition, sequential composition distributes through non-determinism.

Law 2.4 $\langle ;\text{-}\sqcap \text{ dist}\rangle$
(1) $P; (Q \sqcap R) = (P; Q) \sqcap (P; R)$
(2) $(Q \sqcap R); P = (Q; P) \sqcap (R; P)$

The Ordering Relation

Equational reasoning is usually rather limited for some applications of program transformation. It has become standard practice to introduce a *refinement relation*: $P \sqsubseteq Q$ means that Q is at least as good as P in the sense that it will meet every purpose and satisfy every specification satisfied by P. Furthermore, substitution of Q for P in any context can only be an improvement.

We define $\sqsubseteq$ in terms of $\sqcap$. Informally, if the non-determinism of P and Q always yields P, one can be sure that P is worse than Q in all situations.

[1]The set of laws is complete in the sense that they are sufficient to reduce any finite (non-recursive) program to a *normal form*.

Definition 2.1 (The ordering relation)

$$P \sqsubseteq Q \stackrel{def}{=} (P \sqcap Q) = P$$

■

The relation $\sqsubseteq$ is an ω-complete partial ordering: it is reflexive, transitive and antisymmetric, and if $P_i \sqsubseteq P_{i+1}$ for all i, then the sequence has a *least upper bound* satisfying

Law 2.5 $\sqcup_i Pi \sqsubseteq Q \quad \equiv \quad \forall i \bullet P_i \sqsubseteq Q$ $\langle \sqsubseteq - \sqcup$ lub$\rangle$

Moreover, $\sqsubseteq$ has $\bot$ as its bottom element and $\sqcap$ as the *greatest lower bound.*

Law 2.6 $\bot \sqsubseteq P$ $\langle \sqsubseteq - \bot$ bottom$\rangle$

Law 2.7 $(R \sqsubseteq P \wedge R \sqsubseteq Q) \equiv R \sqsubseteq (P \sqcap Q)$ $\langle \sqsubseteq - \sqcap$ glb$\rangle$

In order to be able to use the algebraic laws to transform subcomponents of compound programs, it is crucial that $P \sqsubseteq Q$ imply $F(P) \sqsubseteq F(Q)$, for all *contexts* F (functions from programs to programs). This is equivalent to saying that F (and consequently, all the operators of our language) must be *monotonic* with respect to $\sqsubseteq$. For example:

Law 2.8 If $P \sqsubseteq Q$ then
(1) $(P \sqcap R) \sqsubseteq (Q \sqcap R)$ $\langle \sqcap$ monotonic$\rangle$
(2) $(R; P) \sqsubseteq (R; Q)$ and $(P; R) \sqsubseteq (Q; R)$ $\langle ;$ monotonic$\rangle$

A stronger property of sequential composition will be used in our proofs: it is continuous, i.e., it preserves the least upper bound of ascending chains.

Law 2.9 $P; \sqcup_i Q_i; R = \sqcup_i (P; Q_i; R)$ $\langle \sqsubseteq -;$ continuous$\rangle$

Guarded Command Set

If the same program Q is guarded by all the conditions of a guarded command set, we can be sure that P will be selected for execution. (Recall that we have enforced that the conditions of a guarded command set are exhaustive).

Law 2.10 $(\Box_{i \in S}\, b_i \rightarrow Q) = Q$ $\langle \Box$ idemp$\rangle$

Sequential composition distributes leftward through a guarded command set.

Law 2.11 $(\Box_{i \in S}\, b_i \rightarrow P_i); Q = \Box_{i \in S}\, b_i \rightarrow (P_i; Q)$ $\langle ;- \Box$ leftdist$\rangle$

Conditional

The most basic property of the conditional is that its left branch is executed if the condition holds initially; otherwise its right branch is executed. This suggests that any conditional can be written as a guarded command set.

Definition 2.2 (Conditional)

$$(P \lhd b \rhd Q) \stackrel{def}{=} (b \to P \,\Box\, \neg b \to Q)$$

■

The next two laws follow directly from the above definition and the laws of guarded command sets.

Law 2.12 $(P \lhd true \rhd Q) = P = (Q \lhd false \rhd P)$ $\langle \lhd \rhd \text{ elim} \rangle$

Law 2.13 $(P \lhd b \rhd Q); R = (P; R) \lhd b \rhd (Q; R)$ $\langle ;\!- \lhd \rhd \text{ left dist} \rangle$

Assertion

We will use the notation $b_\perp$ to stand for an *assertion*. It behaves like II if b is true; otherwise it behaves like $\perp$. Clearly, an assertion can be defined in terms of the conditional.

Definition 2.3 (Assertion)

$$b_\perp \stackrel{def}{=} (II \lhd b \rhd \perp)$$

■

We can always determine which command will be selected for execution provided we know which guard holds beforehand. This law relies on the two conditions we have imposed on a guarded command set: the guards must be exhaustive and pairwise disjoint.

Law 2.14 If $j \in S$ then
$b_{j\perp}; (\Box_{i \in S}\, b_i \to P_i) = b_{j\perp}; P_j$ $\langle \Box \text{ elim} \rangle$

If an assertion of a condition b is guarded by b itself, this assertion has no effect whatsoever.

Law 2.15 $\Box_{i \in S}\, b_i \to (b_{i\perp}; P_i) = \Box_{i \in S}\, b_i \to P_i$ $\langle \text{void assertion} \rangle$

Assignment

Obviously, the assignment of the value of a variable to itself does not change anything.

Law 2.16 $(x := x) \;=\; II$ $\langle :=$ skip$\rangle$

In fact, such a vacuous assignment can be added to any other assignment without changing its effect.

Law 2.17 $(x, y := e, y) \;=\; (x := e)$ $\langle :=$ identity$\rangle$

The list of variables and expressions may be subjected to the same permutation without changing the effect of the assignment.

Law 2.18 $(x, y, z := e, f, g) \;=\; (y, x, z := f, e, g)$ $\langle :=$ sym$\rangle$

The following abbreviation will be helpful.

Definition 2.4 (Substitution) Let $f(x)$ be an arbitrary expression possibly containing x as a free variable. We define that to precede this expression with an assignment $x := e$ results in $f(e)$, that is, f with x replaced by e:

$$(x := e; f(x)) \stackrel{def}{=} f(e)$$

■

The sequential composition of two assignments to the same variable is easily combined to a single assignment.

Law 2.19 $(x := e; x := f(x)) \;=\; (x := (x := e; f(x))) \;=\; (x := f(e))$
$\langle :=$ combination$\rangle$

Assignment distributes rightward through a conditional, replacing occurrences of the assigned variables in the condition by the corresponding expressions.

Law 2.20 $x := e; (P \lhd b \rhd Q) \;=\; (x := e; P) \lhd x := e; b \rhd (x := e; Q)$
$\langle := - \lhd \rhd$ right dist$\rangle$

An assignment of a value to a variable x may be eliminated if this assignment is preceded by an assertion that x already holds that value.

Law 2.21 $(x = e)_\perp; (x := e) \;=\; (x = e)_\perp$ $\langle$void assignment$\rangle$

Conditional expressions

We will use the notation

$$\Box_{i \in S}\, b_i \to e_i$$

to represent a *conditional expression* which behaves like the expression e_i whenever the guard b_i holds. The same restrictions imposed on guarded command sets will be assumed here: exactly one of the guards will hold at a given state.

We define a *conditional assignment* as a particular kind of guarded command set.

Definition 2.5 (Conditional assignment)

$$x := (\Box_{i \in S}\, b_i \to e_i) \stackrel{def}{=} (\Box_{i \in S}\, b_i \to (x := e_i))$$

■

The following law allows the elimination of a conditional assignment whose expressions are constant values.

Law 2.22 If x is not free in P and, for each $i \in S$, the expression c_i is a constant value, then
$(x := \Box_{i \in S}\, b_i \to c_i);\ P;\ (x := \Box_{i \in S}\, b_i \to c_i) \ =\ P;\ (x := \Box_{i \in S}\, b_i \to c_i)$
⟨void cond assignment⟩

It is useful to have means of transforming conditional expressions directly. They satisfy laws similar to those of guarded command sets. For example, conditional expressions are idempotent

Law 2.23 $(\Box_{i \in S}\, b_i \to e) \ =\ e$ ⟨**cond exp** idemp⟩

and allow the use of information about the guards in guarded expressions.

Law 2.24 If none of e_i contains x as a free variable then
$\Box_{i \in S}\, (x = e_i) \to f_i(x) \ =\ \Box_{i \in S}\, (x = e_i) \to (x := e_i;\, f_i(x)) \ =$
$\Box_{i \in S}\, (x = e_i) \to f_i(e_i)$ ⟨**cond exp** substitution⟩

Recursion and Iteration

A recursive program $\mu X \bullet F(X)$ is defined as the limit of the ascending chain $F^0(\bot) \sqsubseteq F^1(\bot) \sqsubseteq \ldots$ where:

$$F^0(X) = X \quad \text{and}$$
$$F^{n+1}(X) = F(F^n(X)), \quad \text{for all } n \geq 0.$$

Definition 2.6 (Recursion)

$$\mu X \bullet F(X) \stackrel{def}{=} \sqcup_i F^i(\bot)$$

■

Algebraically, recursion can be characterised by the following well-known fixed point laws.

Law 2.25 $\mu X \bullet F(X) = F(\mu X \bullet F(X))$ ⟨μ fixed point⟩

Law 2.26 $F(Y) \sqsubseteq Y \Rightarrow \mu X \bullet F(X) \sqsubseteq Y$ ⟨μ least fixed point⟩

Iteration can be defined as a special case of recursion:

Definition 2.7 (Iteration)

$$(b * P) \stackrel{def}{=} \mu X \bullet ((P; X) \lhd b \rhd II)$$

■

Therefore it is possible to prove (rather than just postulate) the properties of iteration given below. All these properties are special cases of more general theorems stated and proved in [7].

If we know that the condition of an iteration is true beforehand, we can unfold this iteration in the following way.

Law 2.27 If x is not free in e then
$x := e; (x = e) * P = x := e; P; (x = e) * P$ ⟨$*$ unfold⟩

If the execution of P always establishes $x = e$, then the iteration behaves like abort.

Law 2.28 If $(P; x := e) = P$ and x is not free in e then
$x := e; (x = e) * P = \bot$ ⟨$*$ abort⟩

On the other hand, if the condition is false beforehand the iteration has no effect.

Law 2.29 If x is not free in e then
$x := e; (x \neq e) * P = (x := e)$ ⟨$*$ elim⟩

The following law allows the combination of loops with the same body.

Law 2.30 $(b \wedge c) * P; (b * P) = (b * P)$ ⟨$*$ combination⟩

Variable Declaration

The declaration $\mathbf{var}\, x, y, \ldots, z$ introduces the variables $x, y, \ldots, z$ for use in the program which follows. The undeclaration $\mathbf{end}$ x,y,...,z ends the scope of variables $x, y, \ldots, z$. If P is a program and x is a variable, we say that an occurrence of x in P is *free* if it is not in the scope of any declaration of x in P, and *bound* otherwise.

It does not matter if variables are declared in one list or singly; nor does it matter in which order they are declared.

Law 2.31 ⟨**dec** comm⟩
(1) $(\mathbf{var}\, x; \mathbf{var}\, y) \;=\; \mathbf{var}\, x, y \;=\; (\mathbf{var}\, y; \mathbf{var}\, x)$
(2) $(\mathbf{end}\, x; \mathbf{end}\, y) \;=\; \mathbf{end}\, x, y \;=\; (\mathbf{end}\, y; \mathbf{end}\, x)$

The scope of a variable may be increased (or reduced) without effect, provided that this does not interfere with other variables with the same name.

Law 2.32 If x is not free in P ⟨**dec** change scope⟩
(1) $P; \mathbf{var}\, x \;=\; \mathbf{var}\, x; P$
(2) $\mathbf{end}\, x;\; P \;=\; P; \mathbf{end}\, x$

Declaration distributes through guarded command set, as long as no interference occurs with the conditions.

Law 2.33 If x is not free in b_i, for all $i \in S$
$\mathbf{var}\, x; (\Box_{i \in S}\, b_i \rightarrow P_i); \mathbf{end}\, x \;=\; \Box_{i \in S}\, b_i \rightarrow (\mathbf{var}\, x; P_i; \mathbf{end}\, x)$ ⟨**dec** $-\,\Box$ dist⟩

$\mathbf{var}\, x$ followed by $\mathbf{end}\, x$ has no effect whatsoever.

Law 2.34 $(\mathbf{var}\, x;\; \mathbf{end}\, x) \;=\; II$ ⟨**var** $-$ **end** skip⟩

The next law postulates that the sequential composition of $\mathbf{end}\, x$ with $\mathbf{var}\, x$ has no effect whenever it is followed by an assignment to x that does not rely on the previous value of x.

Law 2.35 If x is not free in e
$(\mathbf{end}\, x;\; \mathbf{var}\, x;\; x := e) \;=\; (x := e)$ ⟨**end** $-$ **var** skip⟩

An assignment to a variable just before the end of its scope is irrelevant.

Law 2.36
$(x := e;\; \mathbf{end}\, x) \;=\; \mathbf{end}\, x$ ⟨**end**$-$:= final value⟩

The following equations can be easily derived from previous laws of declaration.

Law 2.37 ⟨dec– := elim⟩

(1) $(\mathbf{var}\, x;\ x := e;\ \mathbf{end}\, x) \ =\ II$

(2) If x is not free in P

$(\mathbf{var}\, x, y;\ x, y := e, f; P\, \mathbf{end}\, x, y) \ =\ (\mathbf{var}\, y;\ y := f; P\, \mathbf{end}\, y)$

Equations such as the ones above, which combine the effect of more basic laws, allow more interesting transformations, leading to more concise proofs.

3 The Operational Semantics

It is the purpose of an operational semantics to define the relationship between a program and its possible executions by machine. For this we need a concept of execution and a design of machine which are sufficiently realistic to provide guidance for real implementation, but sufficiently abstract for application to the hardware of a variety of real computers. Furthermore it is the major aim of this paper to show that it is possible to derive this kind of semantics in such a way to guarantee its correctness (with respect to the algebraic semantics presented in the previous section).

In the most abstract view, a computation consists of a sequence of individual *steps*. Each step takes the machine from one state m to a closely similar one m'; the transition is often denoted $m \longrightarrow m'$. Each step is drawn from a very limited repertoire, within the capabilities of a simple machine. A definition of the set of all possible single steps simultaneously defines the machine and all possible execution sequences that it can give rise to in the execution of a program.

The step can be defined as a relation between the machine state before the step and the machine state after. In the case of a stored program computer, the state can be analysed as a pair (s, P), where s is the data part (ascribing actual values to the program variables $x, y, \ldots, z$), and P is a representation of the rest of the program that remains to be executed. When this is II, there is no more program to be executed; the state (t, II) is the last state of any execution sequence that contains it, and t defines the final values of the variables.

It is extremely convenient to represent the data part of the state by a total assignment

$$x, y, \ldots, z := k, l, \ldots, m$$

where $k, l, \ldots, m$ are *constant* values which the state ascribes to $x, y, \ldots, z$ respectively.

Suppose that $(s; P) \sqsubseteq t$, where s and t range over states. This means that one of the possible effects of starting program P in state s is to end in state t. Similarly, $(s; P) \sqsubseteq (t; Q)$ means that an implementation of the program $(s; P)$ is permitted instead to execute $(t; Q)$.This is what will actually happen if t is an intermediate state in the execution of $(s; P)$ and Q represents the program that remains to be executed. That explains the definition of the transition relation given below.

Definition 3.1 (Transition relation)

$$(s, P) \longrightarrow (t, Q) \quad \stackrel{def}{=} \quad (s; P) \sqsubseteq (t; Q)$$

■

Therefore the following transition rules can be regarded as theorems, rather then as definitions; they are easily proved from the algebraic laws of the programming language.

1. $(s, v := e) \longrightarrow ((v := (s; e)), II)$
   Proof: from laws $\langle ;\text{-}II \text{ unit}\rangle(2.1)$ and $\langle := \text{ combination}\rangle(2.19)$.
   The effect of a total assignment $v := e$ is to end in a final state in which the variables of the program have constant values $(s; e)$, that is, the result of evaluating the list of expressions e with all variables in it replaced by their initial values. Here we return to the simplifying assumption that expressions are everywhere defined.

2. $(s, (II; Q)) \longrightarrow (s, Q)$
   Proof: from Law $\langle ;\text{-}II \text{ unit}\rangle(2.1)$.
   A II in front of a program Q is immediately discarded.

3. $(s, (P; R)) \longrightarrow (t, (Q; R))$ whenever $(s, P) \longrightarrow (t, Q)$
   Proof: sequential composition is monotonic.
   The first step of the program $(P; R)$ is the same as the first step of P, with R saved up for execution (by the preceding rule) when P has terminated.

4. $(s, P \sqcap Q) \longrightarrow (s, P)$
   $(s, P \sqcap Q) \longrightarrow (s, Q)$
   Proof: from Law $\langle \sqsubseteq\text{-}\sqcap \text{ glb}\rangle(2.7)$.
   The first step of the program $(P \sqcap Q)$ is to discard either one of the components P or Q. The criterion for making the choice is completely undetermined.

5. $(s, P \lhd b \rhd Q) \longrightarrow (s, P)$ whenever $s; b$
   $(s, P \lhd b \rhd Q) \longrightarrow (s, Q)$ whenever $s; \neg b$
   Proof: from laws $\langle ;\text{-} \lhd \rhd \text{ leftdist}\rangle(2.13)$ and $\langle \lhd \rhd \text{ elim}\rangle(2.12)$.
   The first step of the program $(P \lhd b \rhd Q)$ is also a choice, but unlike in the previous rule the choice is made in accordance with the truth or falsity of $(s; b)$, that is, the result of evaluating b with all free variables replaced by their initial values.

6. $(s, \mu \mathtt{X} \bullet \mathtt{F(X)}) \longrightarrow (s, \mathtt{F}(\mu \mathtt{X} \bullet \mathtt{F(X)}))$
   Proof: from Law $\langle \mu \text{ fixed point}\rangle(2.25)$.
   Recursion is implemented by the copy rule, whereby each recursive call within the procedure body is replaced by the whole recursive procedure.

7. $(s, \bot) \longrightarrow (s, \bot)$
   Proof: $\sqsubseteq$ is reflexive.
   The worst program $\bot$ engages in an infinite repetition of vacuous steps.

4 Total Correctness of the Operational Semantics

The correctness of each of the transition rules presented in the previous section is certainly a necessary condition for the correctness of the operational semantics, but it is not a sufficient condition. There are two kinds of error that it does not guard against:

- There may be too few transitions (or even none at all!). An omitted transition would introduce a new and unintended class of terminal states. A more subtle error would be omission of the second of the two rules (4) for $P \sqcap Q$, thereby eliminating non-determinism from the language.
- On the other hand, there may be too many transitions. For example, the transition

  $$(s, Q) \longrightarrow (s, Q)$$

  is entirely consistent with the approach of the previous section, since it just expresses reflexivity of the refinement relation. But its inclusion in the operational definition of the language would mean that every execution of every program could result in an infinite iteration of this dump step.

To guard against these dangers, we need to obtain a clear idea of what it means for an operational semantics to be correct. The purpose of an operational semantics is to define the "machine code" of an abstract machine. One of the best and most common ways of defining a machine code is to write an interpreter for it in a high level programming language, whose meaning is already known; and a language immediately available for this purpose is the one whose algebraic meaning has been given in Section 2. The criterion of total correctness of the interpreter is that its application to a textual representation of any program must be proved to be equal to the algebraic meaning of the program itself.

Therefore we need to make a distinction between two natures of a program: *syntactic* (its textual representation) and *semantic* (its meaning as given by the algebraic laws). In this setting, we must consider the transition rules of the previous section as a *definition* of the way in which programs are to be executed. In this definition, the components s and P of each state are represented concretely by their texts. The states are identified as pairs $(\mathtt{s}, \mathtt{P})$, where $\mathtt{s}$ is a *text* describing the data state, and $\mathtt{P}$ is a program *text*, defining the program state. We use typewriter font to distinguish text from meaning: $\mathtt{P}$ is the text of a program whose meaning is the predicate P. The symbol $\longrightarrow$ is a relation between the texts; it is actually *defined* inductively by the transition rules: it is the *smallest* relation satisfying just those rules.

The alphabet of global variables of our interpreter will be

s to hold the data state (as text)
p to hold the program state (as text)

They will be updated on each step of the interpreter by the assignment

$$s, p := next(s, p)$$

The definition of the *next* function operating on texts is taken directly from the individual clauses of the operational semantics:

1. $next(s, \mathtt{v} := \mathtt{e}) \;=\; ((\mathtt{v} := (s; \mathtt{e})), \mathtt{II})$
2. $next(s, (\mathtt{II}; \mathtt{Q})) \;=\; (s, \mathtt{Q})$
3. $next(s, (\mathtt{P}; \mathtt{R})) \;=\; (t, (\mathtt{Q}; \mathtt{R}))$ whenever $next(s, \mathtt{P}) \;=\; (t, \mathtt{Q})$
4. $next(s, \mathtt{P} \sqcap \mathtt{Q}) \;=\; (s, \mathtt{P})$
   $next(s, \mathtt{P} \sqcap \mathtt{Q}) \;=\; (s, \mathtt{Q})$
5. $next(s, \mathtt{P} \lhd \mathtt{b} \rhd \mathtt{Q}) \;=\; (s, \mathtt{P}) \lhd s; b \rhd (s, \mathtt{Q})$
6. $next(s, \mu\, \mathtt{X} \bullet \mathtt{F(X)}) \;=\; (s, \mathtt{F}(\mu\, \mathtt{X} \bullet \mathtt{F(X)}))$
7. $next(s, \text{“}\bot\text{”}) \;=\; (s, \text{“}\bot\text{”})$

Note that the condition on the right-hand side of rule 5 refers to the (semantic) evaluation of the expression b whose textual representation is denoted by $\mathtt{b}$. For example, if the value of s is $\mathtt{v} := \mathtt{c}$, then $s; b$ is the same as $v := c; b$: that is, b with every occurrence of v replaced by c.

For non-determinism we need two assignments

$$s, p := next1(s, p) \;\sqcap\; s, p := next2(s, p) \qquad \ldots \text{ STEP}$$

where $next1(s, \mathtt{P} \sqcap \mathtt{Q}) \;=\; (s, \mathtt{P})$ and $next2(s, \mathtt{P} \sqcap \mathtt{Q}) \;=\; (s, \mathtt{Q})$.

In all other cases $next1(s, \mathtt{P}) \;=\; next2(s, \mathtt{P}) \;=\; next(s, \mathtt{P})$.

The inner loop of the interpreter repeats this step until the program terminates:

$$(p \neq \mathtt{II}) * STEP \qquad \ldots \text{ LOOP}$$

The interpreter defined as LOOP describes the relationship between the initial and the final values of the program variables s and p. It leaves unchanged the values of the program variables $x, y, \ldots, z$. Recall that the program text $\mathtt{P}$ initially assigned to the variable p has its meaning given by P; the latter describes the way that the abstract variables $x, y, \ldots, z$ are updated. The interpreter is correct if the updates correspond to each other in some appropriate sense. The relevant sense is described exactly by the final value of the variable s, produced by the interpreter. This should describe one of the particular final values permitted by P.

The following program, denoted $\Psi(P)$, describes the interpreted execution of the program text $\mathtt{P}$.

Definition 4.1 (Interpreted execution)

$$\begin{aligned}\Psi(P) \stackrel{def}{=}\ & \textbf{var}\ s, p; \\ & \quad p, s := \mathtt{P}, \Box_{c \in Val}\, v = c \rightarrow \mathtt{v} := \mathtt{c}; \\ & \quad LOOP; \\ & \quad \Box_{c \in Val}\, s = (\mathtt{v} := \mathtt{c}) \rightarrow v := c; \\ & \textbf{end}\ s, p\end{aligned}$$

■

The first action of the interpreted execution is to introduce s and p as local variables. The initial value of p is the program text P. The initial value of s is determined by a conditional expression; this value is calculated from the initial state of the program variables v whose domain is represented by the set Val[2]. Then the interpreter LOOP is executed; this updates s and p, but not the program variables v. If and when the loop terminates, a guarded set command which converts the final value of s to that of v is executed; this will assign the desired final values to the program variables. The final values of s and p are irrelevant after the interpreter is executed, so their scope is ended.

The operational semantics is correct if and only if we can prove the following, for all programs P:

$$\Psi(P) = P$$

The proof that the above holds for every program P is performed by structural induction.

The following theorem deals with II. Its proof does not use any of the transition rules, since (s,II) was regarded as the final states (one for each possible value of s) in the operational semantics.

Lemma 4.1 (Skip)

$$\Psi(II) = II$$

Proof:

$$\begin{aligned}& \Psi(II) \\ = \quad & \{\langle * \text{ elim}\rangle(2.29) \text{ and } \langle \textbf{dec}- := \text{ elim}\rangle(2.37)\} \\ & \textbf{var}\ s;\ s := (\Box_{c \in Val}\, v = c \rightarrow \mathtt{v} := \mathtt{c});\ (\Box_{c \in Val}\, s = (\mathtt{v} := \mathtt{c}) \rightarrow v := c); \\ & \textbf{end}\ s \\ = \quad & \{\text{def cond assign}(2.5),\ \langle ;\text{-}\Box \text{ left dist}\rangle(2.11) \text{ and } \langle \Box - := \text{ elim}\rangle(2.14)\} \\ & \textbf{var}\ s;\ \Box_{c \in Val}\, v = c \rightarrow (s := (\mathtt{v} := \mathtt{c}); v := c);\ \textbf{end}\ s\end{aligned}$$

[2]It may be worth observing that the guarded command set which initialises s satisfies the two required properties: the conditions are clearly exhaustive (since Val comprises every possible value that may be assigned to v) and pairwise disjoint, since v will hold exactly one of the values of the set Val.

$=$ {⟨void assertion⟩(2.15) and⟨void assignment⟩(2.21)}
var s; $\Box_{c \in Val}\, v = c \rightarrow s := (\mathsf{v} := \mathsf{c})$; **end** s
$=$ {def cond assign(2.5) and ⟨**dec**– := elim⟩(2.37)}
II

■

Lemma 4.2 (Assignment)

$$\Psi(v := e) \;=\; (v := e)$$

Proof:

$\Psi(v := e)$
$=$ {⟨∗ unfold⟩(2.27) def of *next* and ⟨:= combination⟩(2.19)}
var $s, p; p, s := II, \Box_{c \in Val}\, v = c \rightarrow \mathsf{v} := ((\mathsf{v} := \mathsf{c}); \mathsf{e})$; $LOOP$;
$(\Box_{c \in Val}\, s = (\mathsf{v} := \mathsf{c}) \rightarrow v := c)$; **end** s, p
$=$ {⟨∗ elim⟩(2.29) and ⟨**dec**– := elim⟩(2.37)}
var s; $s := \Box_{c \in Val}\, v = c \rightarrow \mathsf{v} := ((\mathsf{v} := \mathsf{c}); \mathsf{e})$;
$(\Box_{c \in Val}\, s = (\mathsf{v} := \mathsf{c}) \rightarrow v := c)$; **end** s
$=$ {def cond assign(2.5), ⟨;–□ left dist⟩(2.11) and ⟨□–:= elim⟩(2.14)}
var s; $\Box_{c \in Val}\, v = c \rightarrow (s := (\mathsf{v} := ((\mathsf{v} := \mathsf{c}); \mathsf{e}))$; $v := ((v := c); e))$;
end s
$=$ {⟨:= combination⟩(2.19), ⟨void assertion⟩(2.15) and
⟨void assignment⟩(2.21)}
var s; $\Box_{c \in Val}\, v = c \rightarrow (s := (\mathsf{v} := ((\mathsf{v} := \mathsf{c}); \mathsf{e}))$; $v := e)$; **end** s
$=$ {⟨;–□ left dist⟩(2.11) and ⟨**dec**– := change scope⟩(2.32)}
var s; $(\Box_{c \in Val}\, v = c \rightarrow s := (\mathsf{v} := ((\mathsf{v} := \mathsf{c}); \mathsf{e})))$; **end** s; $v := e$
$=$ {def cond assign(2.5) and ⟨**dec**– := elim⟩(2.37)}
$v := e$

■

Lemma 4.3 (Abort)

$$\Psi(\bot) \;=\; \bot$$

Proof: directly from the definition of the function *next* and laws ⟨∗ abort⟩(2.28) and ⟨;–⊥ zero⟩(2.2). ■

Now we will show that Ψ distributes through each operator of our programming language.

Lemma 4.4 (Non-determinism)

$$\Psi(P \sqcap Q) \;=\; \Psi(P) \sqcap \Psi(Q)$$

Proof: let $S = \Box_{c \in Val}\, v = c \rightarrow \mathtt{v} := \mathtt{c}$ and $T = \Box_{c \in Val}\, s = (\mathtt{v} := \mathtt{c}) \rightarrow v := c$.

$\Psi(P \sqcap Q)$

$=$ $\{\langle *\text{ unfold}\rangle(2.27),\ \text{def. of } \mathit{next} \text{ and } \langle :=\ \text{combination}\rangle(2.19)\}$

$\mathbf{var}\, s, p;\ (p, s := \mathtt{P}, S \ \sqcap\ p, s := \mathtt{Q}, S);\ \mathit{LOOP};\ T;\ \mathbf{end}\, s, p$

$=$ $\{\langle ;- \sqcap \text{ dist}\rangle(2.4)\}$

$(\mathbf{var}\, s, p;\ p, s := \mathtt{P}, S;\ \mathit{LOOP};\ T;\ \mathbf{end}\, s, p) \ \sqcap$

$(\mathbf{var}\, s, p;\ p, s := \mathtt{Q}, S;\ \mathit{LOOP};\ T;\ \mathbf{end}\, s, p)$

$=$ $\{\text{def of } \Psi\}$

$\Psi(P) \sqcap \Psi(Q)$

■

The proof that Ψ distributes through the conditional uses the following definition and its corollary.

Definition 4.2 Let the value of the program variable s be $\Box_{c \in Val}\, v = c \rightarrow \mathtt{v} := \mathtt{c}$. Then

$$(s;\ b) \;\stackrel{def}{=}\; \Box_{c \in Val}\, v = c \rightarrow (v := c;\ b)$$

■

Corollary 4.1 $(s;\ b) \;=\; b$
Proof: from laws $\langle$**cond exp** substitution$\rangle$(2.24) and $\langle$**cond exp** idemp$\rangle$(2.23).■

Lemma 4.5 (Conditional)

$$\Psi(P \lhd b \rhd Q) \;=\; \Psi(P) \lhd b \rhd \Psi(Q)$$

Proof: let $S = \Box_{c \in Val}\, v = c \rightarrow \mathtt{v} := \mathtt{c}$ and $T = \Box_{c \in Val}\, s = (\mathtt{v} := \mathtt{c}) \rightarrow v := c$.

$\Psi(P \lhd b \rhd Q)$

$=$ $\{\langle *\text{ unfold}\rangle(2.27),\ \text{def of } \mathit{next} \text{ and Corollary (4.1)}\}$

$\mathbf{var}\, s, p;\ p, s := ((\mathtt{P}, S) \lhd b \rhd (\mathtt{Q}, S));\ \mathit{LOOP};\ T;\ \mathbf{end}\, s, p$

$=$ $\{\text{def cond assign},\ \langle ;- \lhd \rhd \text{ left dist}\rangle(2.13) \text{ and } \langle \mathbf{dec}{-}\Box \text{ dist}\rangle(2.33)\}$

$(\mathbf{var}\, s, p;\ p, s := \mathtt{P}, S;\ \mathit{LOOP};\ T;\ \mathbf{end}\, s, p) \ \lhd b \rhd$

$(\mathbf{var}\, s, p;\ p, s := \mathtt{Q}, S;\ \mathit{LOOP};\ T;\ \mathbf{end}\, s, p)$

$=$ $\{\text{def of } \Psi\}$

$\Psi(P) \lhd b \rhd \Psi(Q)$

■

The proof that Ψ distributes through sequential composition is slightly more difficult. It uses the following two lemmas. The first one establishes a simple fact about one iteration of the interpreter.

Lemma 4.6

$$(p \neq \mathtt{II})_\perp;\ STEP;\ (p := p; \mathtt{Q}) \ = \ (p \neq \mathtt{II})_\perp;\ (p := p; \mathtt{Q});\ STEP$$

Proof: from the definition of *next* and Law ⟨:= combination⟩(2.19). ■

The following lemma is based on the notion of *data refinement.*

Lemma 4.7

$$(p \neq \mathtt{II}) * STEP;\ (p := \mathtt{II}; \mathtt{Q}) \ =$$
$$(p := p; \mathtt{Q});\ (p \neq \mathtt{II} \wedge p \neq \mathtt{II}; \mathtt{Q}) * STEP$$

Proof: We will use the following abbreviations.

$$I(X) \stackrel{def}{=} (STEP;\ X) \lhd p \neq \mathtt{II} \rhd II$$
$$R(X) \stackrel{def}{=} (STEP;\ X) \lhd p \neq \mathtt{II} \wedge p \neq \mathtt{II}; \mathtt{Q} \rhd II$$

We will show that, for $n \geq 0$:

$$I^n(\perp);\ (p := \mathtt{II}; \mathtt{Q}) \ = \ (p := p; \mathtt{Q});\ R^n(\perp)$$

For $n = 0$ we have

$$\begin{array}{ll} & I^0(\perp);\ (p := \mathtt{II}; \mathtt{Q}) \\ = & \quad \{I^0(\perp) = \perp \text{ and } \langle ;\!-\perp \text{ zero}\rangle(2.2)\} \\ & \perp \\ = & \quad \{R^0(\perp) = \perp \text{ and } \langle ;\!-\perp \text{ zero}\rangle(2.2)\} \\ & (p := p; \mathtt{Q});\ R^0(\perp) \end{array}$$

By induction we have

$$\begin{array}{ll} & I^{n+1}(\perp);\ (p := \mathtt{II}; \mathtt{Q}) \\ = & \quad \{I^{n+1}(X) = I(I^n(X))\} \\ & ((STEP;\ I^n(\perp)) \lhd p \neq \mathtt{II} \rhd II);\ (p := \mathtt{II}; \mathtt{Q}) \\ = & \quad \{\langle ;\!- \lhd \rhd \text{ left dist}\rangle(2.13) \text{ and induction hypothesis}\} \\ & (STEP;\ (p := p; \mathtt{Q});\ R^n(\perp)) \lhd p \neq \mathtt{II} \rhd (p := \mathtt{II}; \mathtt{Q}) \\ = & \quad \{\langle \text{void assertion}\rangle(2.15) \text{ and Lemma 4.6}\} \\ & ((p := p; \mathtt{Q});\ STEP;\ R^n(\perp)) \lhd p \neq \mathtt{II} \rhd (p := \mathtt{II}; \mathtt{Q}) \\ = & \quad \{\langle := -\lhd \rhd \text{right dist}\rangle(2.20)\} \\ & (p := p; \mathtt{Q});\ ((STEP;\ R^n(\perp)) \lhd (p \neq \mathtt{II}) \wedge (p \neq \mathtt{II}; \mathtt{Q}) \rhd II) \\ = & \quad \{R^{n+1}(X) = R(R^n(X))\} \\ & (p := p; \mathtt{Q});\ R^{n+1}(\perp) \end{array}$$

■

Now we show that Ψ distributes through sequential composition.

Lemma 4.8 (Sequential composition)

$$\Psi(P;Q) \;=\; \Psi(P);\, \Psi(Q)$$

Proof: let $S = \Box_{c\in Val}\, v = c \to \mathtt{v} := \mathtt{c}$ and $T = \Box_{c\in Val}\, s = (\mathtt{v} := \mathtt{c}) \to v := c$.

$\Psi(P;Q)$

$=$ {⟨∗combination⟩(2.30)}

var $s, p;\ p, s := (\mathtt{P};\mathtt{Q}), S;\ (p \neq \mathtt{II} \wedge p \neq \mathtt{II};\mathtt{Q}) * STEP;\ LOOP;\ T;$ **end** s, p

$=$ {⟨:= combination⟩(2.19)}

var $s, p;\ p, s := \mathtt{P}, S;\ (p := p;\mathtt{Q});\ (p \neq \mathtt{II} \wedge p \neq \mathtt{II};\mathtt{Q}) * STEP;$ $LOOP;\ T;$ **end** s, p

$=$ {Lemma 4.7}

var $s, p;\ p, s := \mathtt{P}, S;\ LOOP;\ (p := \mathtt{II};\mathtt{Q});\ LOOP;\ T;$ **end** s, p

$=$ {def of $next(s, \mathtt{II};\mathtt{Q})$}

var $s, p;\ p, s := \mathtt{P}, S;\ LOOP;\ (p := \mathtt{Q});\ LOOP;\ T;$ **end** s, p

$=$ {⟨void cond assign⟩(2.22)}

var $s, p;\ p, s := \mathtt{P}, S;\ LOOP;\ T;\ (p := \mathtt{Q});\ LOOP;\ T;$ **end** s, p

$=$ {$(T;\ s := S) = T$, from ⟨;–□ left dist⟩(2.11) and ⟨□– := elim⟩(2.14)}

var $s, p;\ p, s := \mathtt{P}, S;\ LOOP;\ T;\ p, s := \mathtt{Q}, S;\ LOOP;\ T;$ **end** s, p

$=$ {⟨**end** – **var** skip⟩(2.35)}

(**var** $s, p;\ p, s := \mathtt{P}, S;\ LOOP;\ T;$ **end** s, p);
(**var** $s, p;\ p, s := \mathtt{Q}, S;\ LOOP;\ T;$ **end** s, p)

$=$ {def of Ψ}

$\Psi(P);\ \Psi(Q)$

■

Therefore we have shown that Ψ is distributive; this result is summarised by the following lemma.

Lemma 4.9 (Ψ distributive)
Let $F(X_1, \ldots, X_n)$ stand for an arbitrary operator (F) of our programming language with its arguments $(X_1, \ldots, X_n)$. Then

$$\Psi(F(X_1, \ldots, X_n)) \;=\; F(\Psi(X_1), \ldots, \Psi(X_n))$$

Proof: from lemmas 4.4, 4.5 and 4.8. ■

The following corollary of the above lemma establishes one half of the proof thatΨ distributes through recursion.

Corollary 4.2

$$\mu X \bullet F(X) \sqsubseteq \Psi(\mu X \bullet F(X))$$

Proof:

$$\begin{array}{ll} & \Psi(\mu X \bullet F(X)) \\ = & \quad \{\text{def of } next(s, \mu \mathtt{X} \bullet \mathtt{F(X)})\} \\ & \Psi(F(\mu X \bullet F(X))) \\ = & \quad \{\text{Lemma 4.9}\} \\ & F(\Psi(\mu X \bullet F(X))) \end{array}$$

The final result follows from $\langle \mu$ least fixed point$\rangle$(2.26). ■

The other half of the proof will require a few more lemmas and the following abbreviations.

Definition 4.3

$$\begin{array}{lll} \Psi^n(P) & \stackrel{def}{=} & \mathbf{var}\, s, p; \\ & & \quad p, s := \mathtt{P}, \Box_{c \in Val}\, v = c \rightarrow \mathtt{v} := \mathtt{c}; \\ & & \quad I^n(\bot); \\ & & \quad \Box_{c \in Val}\, s = (\mathtt{v} := \mathtt{c}) \rightarrow v := c; \\ & & \mathbf{end}\, s, p \end{array}$$

where

$$\begin{array}{lll} I(X) & \stackrel{def}{=} & (STEP;\, X) \lhd p \neq \mathtt{II} \rhd II, \\ I^0(X) & \stackrel{def}{=} & X \qquad \text{and} \\ I^{n+1}(X) & \stackrel{def}{=} & I(I^n(X)) \qquad \text{for } n \geq 0. \end{array}$$

■

Lemma 4.10

$$\Psi(P) = \sqcup_i \Psi^i(P)$$

Proof: from the fact that $LOOP = \sqcup_i I^i(\bot)$ and the continuity of sequential composition. ■

Similar to Lemma 4.9 (Ψ distributive) it is possible to show that Ψ is *sub-distributive.*

Lemma 4.11 (Ψ sub-distributive)

$$\Psi^n(F(X_1,\ldots,X_k)) \sqsubseteq F(\Psi^n(X_1),\ldots,\Psi^n(X_k))$$

■

Then we have the following corollary.

Corollary 4.3

$$\Psi^n(\mu X \bullet F(X)) \sqsubseteq F^n(\bot)$$

Proof: First we consider the base case.

$$\begin{array}{ll} & \Psi^0(\mu X \bullet F(X)) \\ = & \quad \{I^0(\bot) = \bot \text{ and } \langle ;\!\!-\bot \text{ zero}\rangle(2.2)\} \\ & \bot \\ = & \quad \{F^0(X) = X\} \\ & F^0(\bot) \end{array}$$

Then by induction we have

$$\begin{array}{ll} & \Psi^{n+1}(\mu X \bullet F(X)) \\ = & \quad \{\text{def of } next(s, \mu\,\mathtt{X} \bullet \mathtt{F(X)})\} \\ & \Psi^n(F(\mu X \bullet F(X))) \\ \sqsubseteq & \quad \{\text{Lemma 4.11}\} \\ & F(\Psi^n(\mu X \bullet F(X))) \\ \sqsubseteq & \quad \{\text{Linduction hypothesis}\} \\ & F(F^n(\bot)) \\ = & \quad \{F^{n+1}(X) = F(F^n(X))\} \\ & F^{n+1}(\bot) \end{array}$$

■

The following lemma establishes that Ψ distributes through recursion.

Lemma 4.12

$$\Psi(\mu X \bullet F(X)) \;=\; \mu X \bullet F(X)$$

Proof:

$$
\begin{array}{ll}
 & \mu X \bullet F(X) \\
\sqsubseteq & \quad \{\text{Corollary 4.2}\} \\
 & \Psi(\mu X \bullet F(X)) \\
= & \quad \{\text{Lemma 4.10}\} \\
 & \sqcup_i \Psi^i(\mu X \bullet F(X)) \\
\sqsubseteq & \quad \{\text{Corollary 4.3}\} \\
 & \sqcup_i F^i(\bot) \\
= & \quad \{\text{def of recursion}\} \\
 & \mu X \bullet F(X)
\end{array}
$$

■

Finally comes the most important result of this paper: the theorem which establishes the total correctness of the operational semantics.

Theorem 4.1 (Total correctness)

$$\Psi(P) \;=\; P$$

Proof: from lemmas 4.1, 4.2, 4.3, 4.9 and 4.12. ■

5 Conclusion

This paper has presented an approach to deriving an operational from an algebraic semantics of a programming language. The *consistence* of the operational semantics has been easily proved by showing how each rule can be derived from the algebraic laws. A complementary task was to prove that the rules are *complete* (not in the logical sense, but) in the sense that there are no omitted transitions which would introduce a new and unintended class of terminal states. Furthermore, our approach also prevents the inclusion of unnecessary transitions which could, for example, lead to non-termination. The approach has been illustrated by application to a very simple programming language for expression of sequential algorithms with possible non-determinism. It is our hope that it will generalise to more complex languages.

Both the statement of the correctness theorem and its proof depend utterly on acceptance of the priority of the algebraic semantics as the *specification* of the language. Indeed, unless there is such an independent specification, the question of correctness of an operational semantics cannot arise. But of course it is a matter of choice which semantic style should be presented first. An operational semantics has considerable attraction, and is currently quite fashionable among theorists investigating the foundations of Computing Science. In the absence of an independent meaning for the programming language, the operational

semantics may be the only one acceptable or available as a foundation for a theory of programming. In that case, other methods are needed for escaping from its deplorably low level of abstraction. An example of such methods is *bisimulation* [10], as used by Milner to obtain a more abstract semantics in the form of algebraic laws.

Starting with an algebraic semantics (as we have here) has the danger of inconsistency or other unexpected interactions between the postulated laws. Our work is actually a small part of a more general approach to Unified Theories of Programming [6]. This approach starts with a *denotational semantics* (relating a program to a specification of its observable properties and behaviour) from which the algebraic laws are derived. Then it is suggested how an operational theory is derived from the algebraic (the link which has been more deeply investigated here). To complete the cycle which establishes the equivalence of the three presentations, it is shown how the observational presentation can be derived from the operational. Although the order of presentation of these theories is a matter of choice, it is suggested that one should start with the most abstract, because this gives the most help in specification, design and the development and optimisation of programs.

References

[1] J. A. Bergstra and J. W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[2] S. D. Brooks, C. A. R. Hoare, and A. W. Rorcoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31:560–599, 1984.

[3] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, 1976.

[4] M. Goldsmith, A. W. Roscoe, and B. G. O. Scott. Denotational Semantics for occam2, Part1. *Transputer Communications*, 1(2):65–91, 1993.

[5] M. Goldsmith, A. W. Roscoe, and B. G. O. Scott. Denotational Semantics for occam2, Part2. *Transputer Communications*, 2(1):25–67, 1993.

[6] C. A. R. Hoare. Unified Theories of Programming. Technical report, Oxford University Computing Laboratory, 1994.

[7] C. A. R. Hoare, J. He, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.

[8] C. A. R. Hoare *et al.* Laws of Programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[9] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Lecture Note in Computer Science 92, 1980.

[10] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[11] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, DAIMI-FN-19, Aarhus University, Denmark, 1981.