Chapter 10

# Operational Semantics

An *operational* semantics of a programming language is one that defines not the observable overall effect of a program but rather suggests a complete set of possible individual steps which may be taken in its execution. The observable effect can then be obtained by embedding the steps into an iterative loop, like that of the interpreter described in Section 6.3. Different programs which always give the same overall result under interpretation are thereby shown to be the same. Similarly, different presentations of operational semantics which always agree in the result they give to the same program are thereby shown to define the same meaning for the language.

The individual steps of an operational semantics are usually written in a special notation SOS (Structured Operational Semantics), which is due to Plotkin [151]. It is well suited to its purpose, and has been used to define many different languages [73, 126, 127]. It consists of a collection of transition rules of the form

$$m \rightarrow m'$$

where $m$ is a pattern describing the state of an executing mechanism before a typical step, and $m'$ describes the state immediately after. Frequently the interpreter is required to take some action $a$ whenever it makes the transition; this is written above the arrow

$$m \xrightarrow{a} m'$$

There are many advantages in a purely operational definition of the semantics of a programming language. Firstly, it does not require advanced mathematics to formalise the programmer's intuitive understanding of the way in which a computer executes a program. This understanding is essential when using run-time tracing and symbolic dumps for detecting errors in a program written carelessly, perhaps by someone else. The transition rules abstract nicely from the details of a particular implementing mechanisation, but they are sufficiently faithful to guide

the design of a practical implementation of the language. A detailed count of the number of steps executed is immediately available as a basis for the optimisation of algorithms and the study of their complexity. Finally, an operational semantics gives an in-built guarantee of the computability of the language defined. With so many advantages for the programmer, the debugger, the implementor and the complexity theorist, it is not surprising that operational semantics has been given as the presentation of the semantics of a great many programming languages, and sometimes it is the only available presentation.

But it can never be a complete presentation, because it does not specify the circumstances under which two differently written programs are to be regarded as equal, or one of them as better than the other. Furthermore it does not specify the way in which a selection is made between alternative steps, when more than one is possible. For example, is the choice made in advance? Or at run time? Or is it made according to the first transition in the list? Or is it necessary to explore both alternatives, with or without a later selection? Even the concept of repetition is left vague: perhaps the strongest fixed point may be used rather than the weakest. This lack of precision has been turned to advantage by theorists, who delight in exploring a wide variety of possible dialects of the same language, where the sameness is defined as sharing the same operational semantics. This kind of bottom-up classification is complementary to the top-down classification of languages, which share the same or similar alphabets, signatures and laws.

## 10.1 Derivation of the step relation

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 abstract for application to the hardware of a variety of real computers. 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 $m'$. 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 states are identified as pairs (s, P), where

- s is a *text*, defining the data state as an assignment of constants to all variables of the alphabet, as described in Section 5.8.
- P is a *program text*, representing the rest of the program that remains to be executed. When this is $\Pi$, there is no more program to be executed; the

state (s, Π) is the last state of any execution sequence that contains it, and s defines the final values of the variables.

As in Section 6.3, we have to make the vital distinction between two natures of a program: *syntactic* (its textual representation) and *semantic* (its meaning as a predicate). In the rest of this chapter, we use typewriter font to distinguish text from meaning: P is the text of a program whose meaning is the predicate $P$ and s is the text of the assignment $s$. The important point is that the step relation $\rightarrow$ is defined between the texts, rather than the meanings.

Suppose that $(s; P) \sqsubseteq t$, where s and t range over data states as represented above. As described in Section 5.8, this means that t is a possible final observation of the final data state of an execution of program P that has started in data state s. As a practical consequence, an implementation which is required to execute P in the initial state s is permitted instead to execute the much shorter program t, and this immediately defines the final state t. Similarly, $(s; P) \sqsubseteq (t; Q)$ means that an implementation of the program (s; P) is permitted instead to execute (t; Q), and the result can only be an improvement of the original. This suggests a criterion for valid stepwise execution of a program. Each step merely replaces the current machine state (s, P) with a new machine state (t, Q), which is known to be an improvement of it. This account does not yet give a fully sufficient condition for total correctness of execution, but it explains the preliminary definition of the step relation given below.

**Definition 10.1.1** (Step relation)

$$(\mathtt{s}, \mathtt{P}) \rightarrow (\mathtt{t}, \mathtt{Q}) \quad =_{df} \quad (s; P) \sqsubseteq (t; Q) \qquad \square$$

This definition allows the following transition rules numbered (1) to (7) to be derived as theorems, rather than being presented as postulates or definitions; they are easily proved from the algebraic laws of the programming language.

The effect of a total assignment v := e starting on an initial state s is to end in a final state in which the variables of the program have constant values (s; e), by which we mean the result of evaluating the list of expressions e with all variables in it replaced by their initial values in the data state s. Here we recall the simplifying assumption that expressions are everywhere defined, and state the law

(1) $(\mathtt{s}, \mathtt{v} := \mathtt{e}) \rightarrow (\mathtt{v} := (\mathtt{s}; \mathtt{e}), \Pi)$

**Proof** From 3.1**L4** (*Π*-; left unit) and 3.1**L2** (combine assignments). $\square$

A Π in front of a program Q is immediately discarded.

(2) $(\mathtt{s}, \Pi; \mathtt{Q}) \rightarrow (\mathtt{s}, \mathtt{Q})$

**Proof** From 3.1**L4** (*Π*-; left unit). $\square$

The first step of the sequential composition (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.

(3) $(\mathtt{s}, \mathtt{P};\mathtt{R}) \rightarrow (\mathtt{t}, \mathtt{Q};\mathtt{R})$, whenever $(\mathtt{s}, \mathtt{P}) \rightarrow (\mathtt{t}, \mathtt{Q})$

**Proof** From 2.2**L1** (; assoc) and 2.4**L6** (;-⊓ left distr). □

The first step of the non-deterministic choice (P ⊓ Q) is to discard either one of the components P or Q. The criterion for making the choice is completely undetermined.

(4) $(\mathtt{s}, \mathtt{P} \sqcap \mathtt{Q}) \rightarrow (\mathtt{s}, \mathtt{P})$

$(\mathtt{s}, \mathtt{P} \sqcap \mathtt{Q}) \rightarrow (\mathtt{s}, \mathtt{Q})$

**Proof** From 2.4**L7** (;-⊓ right distr). □

The first step of the conditional (P ◁ b ▷ 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 in the data state s.

(5) $(\mathtt{s}, \mathtt{P} \lhd \mathtt{b} \rhd \mathtt{Q}) \rightarrow (\mathtt{s}, \mathtt{P})$, whenever $s; b$

$(\mathtt{s}, \mathtt{P} \lhd \mathtt{b} \rhd \mathtt{Q}) \rightarrow (\mathtt{s}, \mathtt{Q})$, whenever $s; \neg b$

**Proof** From 3.1**L3** (:=-; right distr) and 2.1**L5** (cond unit). □

Recall that s and t are total assignments of constants, so $(s; b)$ effectively reduces to *true* or *false* and $(s; \neg b)$ to *false* or *true* accordingly.

Recursion is implemented by the copy rule, whereby each recursive call within the procedure body is replaced by the whole recursive procedure.

(6) $(\mathtt{s}, \mu \mathtt{X} \bullet \mathtt{P}(\mathtt{X})) \rightarrow (\mathtt{s}, \mathtt{P}(\mu \mathtt{X} \bullet \mathtt{P}(\mathtt{X})))$

**Proof** From 2.6**L2** (fixed point). □

The worst program **true** engages in an infinite repetition of vacuous steps.

(7) (s, **true**) → (s, **true**)

**Proof** ⊑ is reflexive. □

The correctness of each of these transition rules has been proved simply and separately from the algebraic laws. But the main motive for formulating and proving these particular laws was to give a complete recipe for executing an arbitrary program expressed in the programming language, and for reasoning about such executions. In effect, the laws constitute an operational semantics whose step is *defined* as the *least* relation satisfying the transition rules (1) to (7). As so often in mathematics, the theorems of one branch have become the axioms or the definitions of some apparently distinct branch. Henceforth in this chapter, we

will adopt the operational approach. Accordingly, we will regard (1) to (7) as a *definition* of the step relation between machine states. The proof outlined above is regarded as a demonstration of the *soundness* of the operational semantics with respect to the denotational semantics for the programming language, in the sense that every final state of an execution satisfies the predicate associated with the program. This is certainly a necessary condition for correctness. But it is not a sufficient condition. There are two kinds of error that it does not guard against:

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

   $$(\mathtt{s}, \mathtt{P}) \rightarrow (\mathtt{s}, \mathtt{P})$$

   is entirely consistent since it just expresses reflexivity of $\sqsubseteq$. But its inclusion in the operational definition of the language means that every execution of every program could result in an infinite repetition of such vacuous transitions. Such infinite sequences are inevitable in the case of non-terminating iterations which results from rule (7) and sometimes from rule (6), but these should be the only cases.

The definition of the meaning of correctness for an operational semantics is therefore open to question. An adequate answer to the question should show how to derive the algebraic or denotational semantics back again from the operational. This is the topic of the next two sections.

## 10.2 Bisimulation

The operational semantics uses the actual text of programs to control the progress of the computation; as a consequence, two programs are equal if they are written in exactly the same way, so there cannot be any non-trivial algebraic equations. Instead, we have to define and use some reasonable *equivalence* relation (conventionally denoted $\sim$) between program texts. In fact, it is customary to define this relation first between complete machine states, including the data part. Two programs $\mathtt{P}$ and $\mathtt{Q}$ will then be regarded as equivalent if they are equivalent whenever they are paired with the same data state

$$\mathtt{P} \sim \mathtt{Q} \;=_{df}\; \forall \mathtt{s} \bullet (\mathtt{s}, \mathtt{P}) \sim (\mathtt{s}, \mathtt{Q})$$

Now the basic question is: What is meant by a *reasonable* equivalence between

As an example of the use of bisimulation, we will prove the commutative law of $\sqcap$.

**Lemma 10.2.1**

$\mathtt{P} \sqcap \mathtt{Q} \sim_s \mathtt{Q} \sqcap \mathtt{P}$

**Proof** The trick is to define a relation $\sim$ which makes the law true, and then prove that it has the properties of a strong bisimulation. So let us define a reflexive relation $\sim$ that relates every state of the form $(\mathtt{s}, \mathtt{P} \sqcap \mathtt{Q})$ with itself and with the state $(\mathtt{s}, \mathtt{Q} \sqcap \mathtt{P})$, and relates every other state to itself. This is clearly an equivalence relation. It vacuously satisfies bisimilarity condition (**i**). Further, if $(\mathtt{s}, \mathtt{P} \sqcap \mathtt{Q}) \rightarrow (\mathtt{t}, \mathtt{X})$ then inspection of the two transitions for $\sqcap$ reveals that $\mathtt{t} = \mathtt{s}$ and $\mathtt{X}$ is either $\mathtt{P}$ or $\mathtt{Q}$. In either case, inspection of step (4) shows that $(\mathtt{s}, \mathtt{Q} \sqcap \mathtt{P}) \rightarrow (\mathtt{t}, \mathtt{X})$. The mere existence of this bisimulation proves the bisimilarity of $\mathtt{P} \sqcap \mathtt{Q}$ with $\mathtt{Q} \sqcap \mathtt{P}$. □

An additional most important property of an equivalence relation is that it should be respected by all the operators of the programming language, for example

$$\text{If } \mathtt{P} \sim \mathtt{R} \text{ and } \mathtt{Q} \sim \mathtt{S} \text{ then } (\mathtt{P}; \mathtt{Q}) \sim (\mathtt{R}; \mathtt{S})$$

An equivalence relation with this property is called a *congruence.* It justifies the principle of substitution which underlies all algebraic calculation and reasoning: without it, the algebraic laws would be quite useless. Fortunately, the bisimularity relation $\sim_s$ happens to be a congruence for all operators mentioned in our operational semantics.

**Theorem 10.2.2**

If $\mathtt{P} \sim_s \mathtt{Q}$, then $\mathtt{F}(\mathtt{P}) \sim_s \mathtt{F}(\mathtt{Q})$ for any programming combinator $\mathtt{F}$.

**Proof** We define a relation $\sim$ that is clearly weaker than $\sim_s$, and which trivially satisfies $F(P) \sim F(Q)$ whenever $P \sim_s Q$. We then prove that $\sim$ is a bisimulation, and therefore *equal* to the weakest one $\sim_s$.

$$\begin{aligned} \sim \quad =_{df} \quad & \sim_s \ \cup \ \{(\mathtt{s}, \mathtt{F}(\mathtt{X})),\ (\mathtt{s}, \mathtt{F}(\mathtt{Y})) \mid \mathtt{X} \sim_s \mathtt{Y}\} \\ & \cup \ \{(\mathtt{s}, \mathtt{X}; \mathtt{R}),\ (\mathtt{t}, \mathtt{Y}; \mathtt{R}) \mid (\mathtt{s}, \mathtt{X}) \sim_s (\mathtt{t}, \mathtt{Y})\} \end{aligned}$$

(**i**) is trivially satisfied, because $(\mathtt{s}, \mathtt{F}(\mathtt{X}))$ is never a terminal state. Since $\sim$ is symmetric, we only need to concentrate on (**ii**). The proof proceeds by case analysis.

**Case 1** $\mathtt{F}(\mathtt{X}) = (\mathtt{X} \sqcap \mathtt{R})$.

The antecedent of (**ii**)

$$(\mathtt{s}, \mathtt{P} \sqcap \mathtt{R}) \sim (\mathtt{t}, \mathtt{Q} \sqcap \mathtt{R}) \text{ and } (\mathtt{t}, \mathtt{Q} \sqcap \mathtt{R}) \rightarrow (\mathtt{v}, \mathtt{S})$$

by step (4) implies that $\mathtt{t} = \mathtt{v}$ and $\mathtt{S}$ is either $\mathtt{Q}$ or $\mathtt{R}$. By the definition of $\sim$ it follows that either $\mathtt{P} \sqcap \mathtt{R} \sim_s \mathtt{Q} \sqcap \mathtt{R}$ (which completes the proof) or $\mathtt{s} = \mathtt{t}$.

**Case 1a** $S = Q$.

$$(s, P \sqcap R) \sim (s, Q \sqcap R) \wedge (s, Q \sqcap R) \rightarrow (s, Q) \qquad \{\text{step (4)}\}$$
$$\Rightarrow (s, P \sqcap R) \rightarrow (s, P) \wedge (s, P) \sim_s (s, Q) \qquad \{\text{def of } \sim\}$$
$$\Rightarrow (s, P \sqcap R) \rightarrow (s, P) \wedge (s, P) \sim (s, Q)$$

**Case 1b** $S = R$.

$$(s, P \sqcap R) \sim (s, Q \sqcap R) \wedge (s, Q \sqcap R) \rightarrow (s, R) \qquad \{\text{step (4) and } \sim_s \subseteq \sim\}$$
$$\Rightarrow (s, P \sqcap R) \rightarrow (s, R) \wedge (s, R) \sim (s, R)$$

**Case 2** $F(X) = (X \lhd b \rhd R)$.

Similar to **Case 1**.

**Case 3** $F(X) = (X; R)$.

First consider the case when $Q \neq \Pi$.

def $\sim$ and step (3)

$$\Rightarrow (s, P; R) \sim (s, Q; R) \wedge (s, Q; R) \rightarrow (v, Q'; R) \qquad \{\text{step (3)}\}$$
$$\Rightarrow (s, Q) \rightarrow (v, Q') \qquad \{P \sim_s Q\}$$
$$\Rightarrow (s, P) \rightarrow (u, P') \wedge (u, P') \sim_s (v, Q') \qquad \{\text{def } \sim, \text{ step (3)}\}$$
$$\Rightarrow (s, P; R) \rightarrow (u, P'; R) \wedge (u, P'; R) \sim (v, Q'; R)$$

If $Q = \Pi$, from condition (i) of a strong bisimulation it follows that $P = \Pi$.

**Case 4** $F(X) = (R; X)$.

Similar to **Case 3**.

**Case 5** $F(X) = \mu Y \bullet G(X, Y)$.

def $\sim$ and step (6)

$$\Rightarrow (s, \mu Y \bullet G(P, Y)) \sim (s, \mu Y \bullet G(Q, Y))$$
$$\wedge (s, \mu Y \bullet G(Q, Y)) \rightarrow (s, G(Q, \mu Y \bullet G(Q, Y))) \qquad \{\text{def of } \sim \text{ and step (6)}\}$$
$$\Rightarrow (s, \mu Y \bullet G(P, Y)) \rightarrow (s, G(P, \mu Y \bullet G(P, Y)))$$
$$\wedge (s, G(P, \mu Y \bullet G(P, Y))) \sim (s, G(Q, \mu Y \bullet G(Q, Y)))$$

From the above analysis we conclude that $\sim$ is a strong bisimulation. □

But there is no strong bisimulation that would enable one to prove the idempotence law for $\sqcap$. For example, let $\sim$ be a relation such that

$$(s, \Pi) \sim (s, \Pi \sqcap \Pi)$$

Clearly $(\mathtt{s},\ \mathrm{II} \sqcap \mathrm{II}) \rightarrow (\mathtt{s},\ \mathrm{II})$. However, the operational semantics deliberately excludes any $(\mathtt{t},\ \mathtt{X})$ such that $(\mathtt{s},\ \mathrm{II}) \rightarrow (\mathtt{t},\ \mathtt{X})$; condition **(ii)** for bisimulation is therefore violated. In fact, this condition is so strong that it requires any two bisimilar programs to terminate in exactly the same number of steps. Since one of the main motives for exploring equivalence of programs is to replace a program by one that can be executed in fewer steps, strong bisimilarity is far too strong a relation for this purpose.

Milner's solution to this problem is to define a weak form of bisimilarity (which we denote $\approx$) for which the first two conditions are weakened to

**(i)** if $(\mathtt{s},\mathtt{P}) \approx (\mathtt{t},\mathtt{Q})$ and $(\mathtt{t},\mathtt{Q}) \not\rightarrow$

then there is a state $(\mathtt{t},\mathtt{R})$ such that

$$(\mathtt{s},\mathtt{P}) \stackrel{*}{\rightarrow} (\mathtt{t},\mathtt{R}) \quad \text{and} \quad (\mathtt{t},\mathtt{R}) \not\rightarrow$$

where $\stackrel{*}{\rightarrow}$ is defined formally in the next section as the *reflexive transitive closure* of the relation $\rightarrow$.

**(ii)** $(\approx;\rightarrow) \subseteq (\stackrel{*}{\rightarrow};\approx)$

Condition **(ii)** is very similar to the confluence condition used in the proof of the Church–Rosser property of a set of algebraic transformations. Weak bisimilarity $\approx_w$ is defined from weak bisimulation in the same way as for strong bisimilarity, and it is also a congruence.

**Theorem 10.2.3**

If $\mathtt{P} \approx_w \mathtt{Q}$ then $\mathtt{F}(\mathtt{P}) \approx_w \mathtt{F}(\mathtt{Q})$ for all programming combinators $\mathtt{F}$.

**Proof** Analogous to Theorem 10.2.2. □

Using the weak bisimilarity we can prove disjunction is idempotent.

**Lemma 10.2.4**

$\mathtt{P} \sqcap \mathtt{P} \approx_w \mathtt{P}$

**Proof** Let $\approx$ relate every $(\mathtt{s},\ \mathtt{P})$ just to itself and to $(\mathtt{s}, \mathtt{P} \sqcap \mathtt{P})$ and vice versa. In the operational semantics $(\mathtt{s}, \mathtt{P} \sqcap \mathtt{P})$ is related by $\rightarrow$ only to $(\mathtt{s},\ \mathtt{P})$; fortunately $(\mathtt{s},\mathtt{P}) \stackrel{*}{\rightarrow} (\mathtt{s},\ \mathtt{P})$ since $\stackrel{*}{\rightarrow}$ is reflexive. Conversely, whenever $(\mathtt{s},\ \mathtt{P}) \rightarrow (\mathtt{s}',\ \mathtt{P}')$ then $(\mathtt{s},\ \mathtt{P} \sqcap \mathtt{P}) \rightarrow (\mathtt{s},\ \mathtt{P}) \rightarrow (\mathtt{s}',\ \mathtt{P}')$, so equality is restored after two steps. This therefore is the bisimulation that shows the weak bisimilarity

$$(\mathtt{P} \sqcap \mathtt{P}) \approx_w \mathtt{P}$$ □

Unfortunately, we still cannot prove the associative law for disjunction. The three simple states $(\mathtt{s},\ \mathtt{x} := 1), (\mathtt{s},\ \mathtt{x} := 2)$ and $(\mathtt{s}, \mathtt{x} := 3)$ end in three distinct

final states, and by condition (i), none of them is bisimilar to any other. The state $(\mathbf{s}, (\mathbf{x} := 1 \sqcap \mathbf{x} := 2))$ and $(\mathbf{s}, (\mathbf{x} := 2 \sqcap \mathbf{x} := 3))$ are also distinct, because each of them has a transition to a state (i.e. $(\mathbf{s}, \mathbf{x} := 1)$ and $(\mathbf{s}, \mathbf{x} := 3)$ respectively) which cannot be reached in any number of steps by the other. For the same reason $(\mathbf{s}, (\mathbf{x} := 1 \sqcap \mathbf{x} := 2) \sqcap \mathbf{x} := 3)$ is necessarily distinct from $(\mathbf{s}, \mathbf{x} := 1 \sqcap (\mathbf{x} := 2 \sqcap \mathbf{x} := 3))$. The associative law for disjunction is thereby violated.

In fact, there is a perfectly reasonable sense in which it is possible to observe the operational distinctness between the two sides of an associative equation

$$(\mathtt{P} \sqcap \mathtt{Q}) \sqcap \mathtt{R} \;=\; \mathtt{P} \sqcap (\mathtt{Q} \sqcap \mathtt{R})$$

Let us suppose that at any time it is possible to split the machine state into two identical copies, and examine the behaviour of each member of the pair independently (some card games offer a similar option). The left hand side of the associative equation may follow any of the paths of the tree of Figure 10.2.5 (where the data state is unchanged, and therefore omitted).

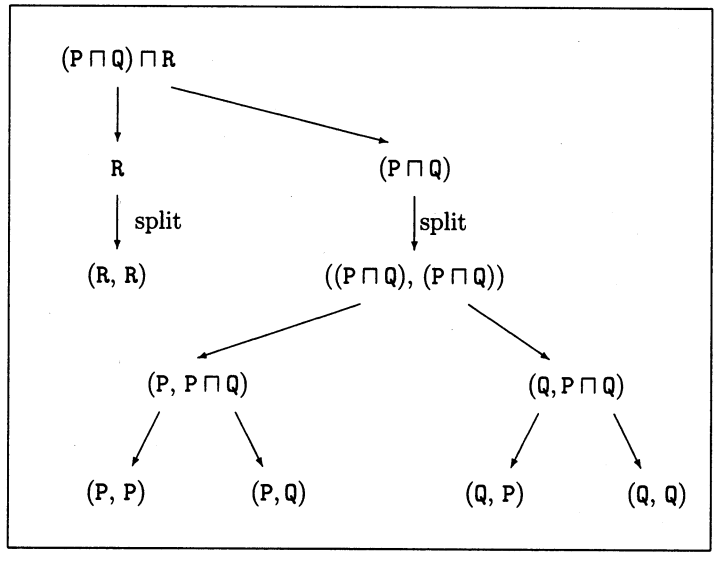


**Figure 10.2.5** The game tree of $(\mathtt{P} \sqcap \mathtt{Q}) \sqcap \mathtt{R}$

The leaves of this tree are the pairs of games

$$(\mathtt{R}, \mathtt{R}),\quad (\mathtt{P}, \mathtt{P}),\quad (\mathtt{P}, \mathtt{Q}),\quad (\mathtt{Q}, \mathtt{P}) \text{ and } (\mathtt{Q}, \mathtt{Q})$$

A similar tree for the bracketing $P \sqcap (Q \sqcap R)$ is given in Figure 10.2.6.

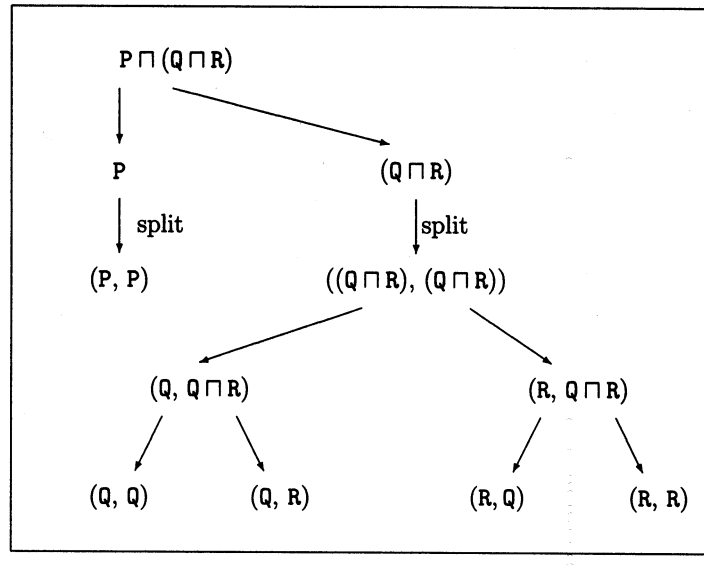


**Figure 10.2.6** The game tree of $P \sqcap (Q \sqcap R)$

It has the leaves

$$(P, P),\quad (Q, Q),\quad (Q, R),\quad (R, Q) \text{ and } (R, R)$$

Observation of a pair of terminal states from $(P, Q)$ will distinguish the left bracketing from the right bracketing. If we want to explore a theory applicable to systems that can be arbitrarily duplicated, we have to abandon associativity of non-determinism.

We have shown that even weak bisimilarity is not weak enough to give one of the laws that we quite reasonably require. Unfortunately, it is also too weak for our purposes: it gives rise to algebraic laws that we definitely do *not* want. For example, consider the program

$$\mu X \bullet (\Pi \sqcap X)$$

This could lead to an infinite computation (if the second disjunct $X$ is always selected), or it could terminate (if the first disjunct $\Pi$ is ever selected, even only once). Weak bisimilarity ignores the non-terminating case, and equates the program to $\Pi$. However, in our theory it is equated to **true**, the weakest fixed point of the equation

$$X = \Pi \sqcap X$$

Our weaker interpretation **true** permits a wider range of implementations: for example, the "wrong" choice may be infinitely often selected at run time; indeed, the "right" choice can even be eliminated at compile time! For a theory based on bisimilarity, neither of these implementations is allowed. A non-deterministic construction $(\mathtt{P} \sqcap \mathtt{Q})$ is expected to be implemented *fairly*: in any infinite sequence of choices, each alternative must be chosen infinitely often. Weak bisimilarity is a very neat way of imposing this obligation, which at one time was thought essential to the successful use of non-determinism.

Unfortunately, the requirement of fairness violates the basic principle of monotonicity, on which so much of engineering design depends. The program $(\mathtt{X} \sqcap \Pi)$ is necessarily less deterministic than $\mathtt{X}$, so $(\mu \mathtt{X} \bullet \mathtt{X} \sqcap \Pi)$ should (by monotonicity) be less deterministic than $(\mu \mathtt{X} \bullet \mathtt{X})$, which is the worst program of all. However, weak bisimulation identifies it with the completely deterministic program $\Pi$. It would therefore be unwise to base a calculus of design on weak bisimulation.

But that was never the intention; bisimulation was originally proposed by Milner as the strongest equivalence that can reasonably be postulated between programs, and one that could be efficiently tested by computer, without any consideration of any possible meaning of the texts being manipulated. It was used primarily to explore the algebra of communication and concurrency. It was not intended for application to a non-deterministic sequential programming language, and the problems discussed in this section suggest it would be a mistake to do so. A great many alternative definitions of program equivalence based on operational semantics have been explored by subsequent research. The one that serves our purposes best is described in the next section.

## 10.3 From operations to algebra

In this section we will define a concept of simulation which succeeds in reconstructing the algebraic semantics of the programming language on the basis of its operational semantics. Firstly, we define $\xrightarrow{*}$ as the reflexive transitive closure of $\rightarrow$

$$\xrightarrow{*} \quad =_{df} \quad \nu X \bullet (id \vee (\rightarrow; X))$$

where $id$ stands for the identity relation on machine states.

Secondly, we define the concept of *divergence*, being a state that can lead to an infinite execution

$$(\mathtt{s}, \mathtt{P}) \uparrow \quad =_{df} \quad \forall n, \exists \mathtt{t}, \mathtt{Q} \bullet (\mathtt{s}, \mathtt{P}) \rightarrow^n (\mathtt{t}, \mathtt{Q})$$

where $\rightarrow^1 \quad =_{df} \quad \rightarrow$

and $\rightarrow^{n+1} \quad =_{df} \quad \rightarrow^1; \rightarrow^n$

Now, we define an ordering relation $\sqsubseteq$ between states. One state is *better* in this ordering than another if any result given by the better state is also possibly given by the worse and, furthermore, a state that can fail to terminate is worse than any other

$$(\mathtt{s}, \mathtt{P}) \sqsubseteq (\mathtt{t}, \mathtt{Q}) \;=_{df}\; \neg(\mathtt{s}, \mathtt{P})\uparrow$$
$$\Rightarrow\; (\neg(\mathtt{t}, \mathtt{Q})\uparrow \;\wedge\; (\forall \mathtt{u} \bullet (\mathtt{t}, \mathtt{Q}) \xrightarrow{*} (\mathtt{u}, \Pi) \Rightarrow (\mathtt{s}, \mathtt{P}) \xrightarrow{*} (\mathtt{u}, \Pi)))$$

One program is better than another if it is better in all data states

$$\mathtt{P} \sqsubseteq \mathtt{Q} \quad \textbf{iff} \quad \forall \mathtt{s} \bullet (\mathtt{s}, \mathtt{P}) \sqsubseteq (\mathtt{s}, \mathtt{Q})$$

Our task in this section is to show that the $\sqsubseteq$ relation defined between machine states by the operational semantics corresponds exactly with the refinement ordering relation of the algebraic semantics, which is already in correspondence with implication between predicates of the denotational semantics (see Section 5.10).

**Theorem 10.3.1**

The syntactically defined $\sqsubseteq$ relation is a preorder. □

The following theorem states that the operational semantics *faithfully* implements the algebraic semantics in the following sense:

- a state $(\mathtt{s}, \mathtt{P})$ is divergent **iff** the algebra agrees that $P$ when started in the initial state $s$ fails to terminate, and
- a state $(\mathtt{s}, \mathtt{P})$ can lead to a termination state $(\mathtt{t}, \Pi)$ **iff** the algebra agrees that $P$ when started in the initial state $s$ may deliver $t$ as the final state.

**Theorem 10.3.2**

(1) $(\mathtt{s}, \mathtt{P})\uparrow$ **iff** $(s; P) = \mathbf{true}$ is algebraically provable.

(2) $(\mathtt{s}, \mathtt{P})\uparrow$ or $(\mathtt{s}, \mathtt{P}) \xrightarrow{*} (\mathtt{t}, \Pi)$ **iff** $(s; P) \sqsubseteq t$ is algebraically provable.

**Proof** We prove the conclusion by structural induction on P.

**Case 1** $\mathtt{P} = \mathbf{true}$.

The conclusion follows from 5.3**L5** and the fact that $(\mathtt{s}, \mathbf{true})\uparrow$.

**Case 2** $\mathtt{P} = (\mathtt{v} := \mathtt{e})$.

The conclusion follows from 5.1**L3** and step (1)

$$(\mathtt{s}, \mathtt{v} := \mathtt{e}) \rightarrow (\mathtt{v} := (\mathtt{s}; \mathtt{e}), \Pi)$$

**Case 3** $\mathtt{P} = (\mathtt{P1} \sqcap \mathtt{P2})$.

The proof is based on the fact that all results from a disjunction, whether

terminating or not, must arise from one of the disjuncts.

$(\sqcap -1)$ $(\mathtt{s}, \mathtt{P1} \sqcap \mathtt{P2}) \uparrow$ **iff** $(\mathtt{s}, \mathtt{P1}) \uparrow$ or $(\mathtt{s}, \mathtt{P2}) \uparrow$

$(\sqcap -2)$ $(\mathtt{s}, \mathtt{P1} \sqcap \mathtt{P2}) \xrightarrow{*} (\mathtt{t}, \Pi)$ **iff** $(\mathtt{s}, \mathtt{P1}) \xrightarrow{*} (\mathtt{t}, \Pi)$ or $(\mathtt{s}, \mathtt{P2}) \xrightarrow{*} (\mathtt{t}, \Pi)$

$$
\begin{aligned}
& (\mathtt{s}, \mathtt{P}) \uparrow && \{(\sqcap -1)\} \\
\equiv\ & (\mathtt{s}, \mathtt{P1}) \uparrow \ \vee\ (\mathtt{s}, \mathtt{P2}) \uparrow && \{\text{induction hypothesis}\} \\
\equiv\ & (s; P1 = \mathbf{true}) \vee (s; P_2 = \mathbf{true}) && \{(s;b1) \vee (s;b2) = \mathbf{true} \text{ iff} \\
& && \ s;b1 = \mathbf{true} \text{ or } s;b2 = \mathbf{true}\} \\
\equiv\ & s;(P1 \sqcap P2) = \mathbf{true}
\end{aligned}
$$

**Case 4** $\mathtt{P} = (\mathtt{P1} \lhd b \rhd \mathtt{P2})$.

Similar to **Case 1**.

**Case 5** $\mathtt{X} = \mathtt{P1};\mathtt{P2}$.

In the proof we will use the fact that the second component of sequential composition can start its execution only after termination of the first one.

$(;-1)$ $(\mathtt{s}, \mathtt{P1};\mathtt{P2}) \uparrow$ **iff** $(\mathtt{s}, \mathtt{P1}) \uparrow$ or $\exists \mathtt{u} \bullet ((\mathtt{s}, \mathtt{P1}) \xrightarrow{*} (\mathtt{u}, \Pi) \ \wedge\ (\mathtt{u}, \mathtt{P2}) \uparrow)$

$(;-2)$ $(\mathtt{s}, (\mathtt{P1};\mathtt{P2})) \xrightarrow{*} (\mathtt{u}, \Pi)$ **iff** $\exists \mathtt{t} \bullet ((\mathtt{s}, \mathtt{P1}) \xrightarrow{*} (\mathtt{t}, \Pi)$ and $(\mathtt{t}, \mathtt{P2}) \xrightarrow{*} (\mathtt{u}, \Pi))$

$$
\begin{aligned}
& (\mathtt{s}, \mathtt{P}) \uparrow \ \vee\ (\mathtt{s}, \mathtt{P}) \xrightarrow{*} (\mathtt{t}, \Pi) && \{(;-1) \text{ and } (;-2)\} \\
\equiv\ & (\mathtt{s}, \mathtt{P1}) \uparrow \ \vee \\
& \exists \mathtt{u} \bullet ((\mathtt{s}, \mathtt{P1}) \xrightarrow{*} (\mathtt{u}, \Pi) \ \wedge\ (\mathtt{u}, \mathtt{P2})) \uparrow \ \vee \\
& \exists \mathtt{u} \bullet ((\mathtt{s}, \mathtt{P1}) \xrightarrow{*} (\mathtt{u}, \Pi) \ \wedge\ (\mathtt{u}, \mathtt{P2}) \xrightarrow{*} (\mathtt{t}, \Pi)) && \{\text{simplification}\} \\
\equiv\ & (\mathtt{s}, \mathtt{P1}) \uparrow \ \vee \\
& \exists \mathtt{u} \bullet ((\mathtt{s}, \mathtt{P1}) \xrightarrow{*} (\mathtt{u}, \Pi) \ \wedge \\
& (\mathtt{u}, \mathtt{P2}) \uparrow \ \vee\ (\mathtt{u}, \mathtt{P2}) \xrightarrow{*} (\mathtt{t}, \Pi)) && \{\text{induction hypothesis}\} \\
\equiv\ & \exists \mathtt{u} \bullet \ ((s; P1) \sqsubseteq u) \ \wedge\ ((u; P2) \sqsubseteq t) && \{5.3\mathbf{L3} \text{ and } 5.3\mathbf{L4}\} \\
\equiv\ & (s; P1; P2) \sqsubseteq t
\end{aligned}
$$

**Case 6** $\mathtt{P} = \mu \mathtt{X} \bullet \mathtt{F}(\mathtt{X})$.

The proof is based on the following facts:

$(\mu -1)$ $(\mathtt{s}, \mu \mathtt{X} \bullet \mathtt{F}(\mathtt{X})) \uparrow$ **iff** $\forall n \bullet (\mathtt{s}, \mathtt{F}^{(n)}(\mathbf{true})) \uparrow$

$(\mu -2)$ $\neg(\mathtt{s}, \mathtt{F}^n(\mathbf{true})) \uparrow \ \Rightarrow ((\mathtt{s}, \mathtt{F}^n(\mathbf{true})) \xrightarrow{*} (\mathtt{t}, \Pi) \equiv (\mathtt{s}, \mu \mathtt{X} \bullet \mathtt{F}(\mathtt{X})) \xrightarrow{*} (\mathtt{t}, \Pi))$

$$
\begin{array}{lll}
& (\mathtt{s}, \mathtt{P})\uparrow \ \vee \ (\mathtt{s}, \mathtt{P}) \xrightarrow{*} (\mathtt{t}, \Pi) & \{(\mu-1) \text{ and } (\mu-2)\} \\
\equiv & \forall n \bullet (\mathtt{s}, \mathtt{F}^n(\mathbf{true}))\uparrow \ \vee & \\
& \exists n \bullet \neg(\mathtt{s}, \mathtt{F}^n(\mathbf{true}))\uparrow \ \wedge & \\
& (\mathtt{s}, \mathtt{F}^n(\mathbf{true})) \xrightarrow{*} (\mathtt{t}, \Pi) & \{\text{induction hypothesis}\} \\
\equiv & \forall n \bullet (s; F^n(\mathbf{true}) = \mathbf{true}) \vee & \\
& \exists n \bullet (s; F^n(\mathbf{true}) \neq \mathbf{true}) \wedge & \{\text{Theorem 5.4.1 and} \\
& \quad (s; F^n(\mathbf{true}) \sqsubseteq t) & \text{strong ordering of Section 5.6}\} \\
\equiv & (s; P = \mathbf{true}) \ \vee \ ((s; P) \sqsubseteq t) & \{\mathbf{true} \sqsubseteq t\} \\
\equiv & (s; P) \sqsubseteq t & \square
\end{array}
$$

We have now proved a close correspondence between the syntactic ordering defined operationally between program texts and the semantic ordering relation used in an algebraic presentation. The correspondence can be strengthened by a standard construction to operational equivalence and algebraic equality

$$\mathtt{P} \sim \mathtt{Q} =_{df} (\mathtt{P} \sqsubseteq \mathtt{Q}) \wedge (\mathtt{Q} \sqsubseteq \mathtt{P})$$

Because $\sqsubseteq$ is a preorder, the relation $\sim$ is an equivalence. The algebraic semantics is isomorphic to the operational, when abstracted by this particular notion of bisimulation.

**Theorem 10.3.3**

$\mathtt{P} \sim \mathtt{Q}$ **iff** $P = Q$ is algebraically provable.

**Proof** From Theorem 10.3.2 and the fact that

$$
\begin{array}{ll}
& P = Q \quad \text{is algebraically provable} \\
\textbf{iff} & \forall \mathtt{s} \bullet (s; P = \mathbf{true}) \equiv (s; Q = \mathbf{true}) \wedge \forall \mathtt{s}, \mathtt{t} \bullet ((s; P) \sqsubseteq t) \equiv ((s; Q) \sqsubseteq t) \\
\textbf{iff} & \forall \mathtt{s} \bullet (\mathtt{s}, \mathtt{P})\uparrow \equiv (\mathtt{s}, \mathtt{Q})\uparrow \\
& \wedge \forall \mathtt{s}, \mathtt{t} \bullet (\mathtt{s}, \mathtt{P})\uparrow \vee (\mathtt{s}, \mathtt{P} \xrightarrow{*} (\mathtt{t}, \Pi)) \equiv (\mathtt{s}, \mathtt{Q})\uparrow \vee (\mathtt{s}, \mathtt{Q} \xrightarrow{*} (\mathtt{t}, \Pi)) \\
\textbf{iff} & \mathtt{P} \sim \mathtt{Q} \qquad \square
\end{array}
$$

## 10.4* From operational to denotational semantics

In Chapter 5 we derived a complete algebraic presentation of the theory of programming from a denotational definition of the semantics of the programming language. In Section 5.8, we showed how to reverse the direction of derivation, and

obtain a denotational semantics from the algebraic laws. It is encouraging that the same theory is obtained in all cases. In this section we will make a similar reversal of direction. Having derived an operational semantics from an algebraic, we complete the cycle by showing how to derive a denotational semantics from an operational.

The technique is the same as that of Section 5.8. We define a function $\mathcal{O}$ which maps any program text to a relation, and then prove that $\mathcal{O}$ satisfies the equations (given in Section 5.8) which defines the denotational semantics $\mathcal{R}$. Of course $\mathcal{O}$ must be defined wholly in terms of the step relation $\rightarrow$, which is given as the primitive in the operational presentation.

**Definition 10.4.1** (denotation from operations)

$$\mathcal{O}(\mathtt{P}) \;=_{df}\; pre(v) \vdash post(v, v')$$

where

$$\begin{aligned} pre(v_0) \quad &=_{df} \quad \neg((\mathtt{v} := \mathtt{v}_0, \mathtt{P})\uparrow) \\ post(v_0, v_1) \quad &=_{df} \quad (\mathtt{v} := \mathtt{v}_0, \mathtt{P})\uparrow \;\vee\; (\mathtt{v} := \mathtt{v}_0, \mathtt{P}) \xrightarrow{*} (\mathtt{v} := \mathtt{v}_1, \mathtt{II}) \end{aligned}$$ □

**Exercise 10.4.2**

Prove that $\mathcal{O}$ satisfies the equations defining $\mathcal{R}$. □

## 10.5* Operational semantics of CSP

This section illustrates how to derive an operational semantics for CSP from its algebraic laws in the same way that Section 10.1 did for a sequential language. We will treat prefixing, non-determinism, choice, parallel and sequential composition, hiding and recursion.

The empty transition of an operational semantics is identified with the resolution of possible internal non-determinism of a process. The result can only be equally or more deterministic than the original. As in Section 10.1 we define

$$P \rightarrow Q \;=_{df}\; P \sqsubseteq Q$$

The action transition relation of an operational semantics is written

$$P \xrightarrow{a} Q$$

It means that $a$ is a possible initial event for $P$, and $Q$ is a description of possible behaviour after $a$. Consequently, the range of behaviours of $P$ does not increase when the possible behaviour of $a \rightarrow Q$ is added as an explicit alternative to $P$. We therefore define

**Definition 10.5.1** (Action transition relation)

$$P \xrightarrow{a} Q \;=_{df}\; P \sqsubseteq (a \to Q)\|P \qquad \square$$

The inequality is required because $P$ may resolve internal non-determinism in a way that may even prevent it from engaging in an initial event $a$, or from behaving like $Q$ afterwards. The defining inequality may be rewritten as an equation

$$P = ((a \to Q)\|P) \sqcap P$$

which in turn can be rewritten to

$$P = (a \to Q)\|(x \to P)\backslash\{x\}$$

where $x$ is an event name occurring only as shown, and nowhere else. In CCS such an event is encoded as $\tau$; thus

$$P = a.Q + \tau.P$$

The main algebraic laws for the proofs have been given in Section 8.2. The following four lemmas are also required.

**Lemma 10.5.2**

$(P\|Q);R \;\sqsubseteq\; (P;R)\|(Q;R)$ $\square$

**Lemma 10.5.3**

$(P\|Q)\backslash E \;\sqsubseteq\; (P\backslash E)\|(Q\backslash E)$ $\square$

**Lemma 10.5.4**

Let $P = ((a \to R)\|S)$ and $a \notin \mathcal{A}(Q)$. Then

$$P\|_{CSP}Q \;=\; (a \to (R\|_{CSP}Q))\|(P\|_{CSP}Q) \qquad \square$$

**Lemma 10.5.5**

Let $P = ((a \to R)\|S)$ and $Q = ((a \to U)\|W)$. Then

$$P\|_{CSP}Q \;=\; (a \to (R\|_{CSP}U))\,\|\,(P\|_{CSP}Q) \qquad \square$$

Using the definition given above, we now proceed to derive the standard operational semantics of CSP [158, 160].

The first step of the process $a \to P$ is to engage in the event $a$. Afterwards, it behaves like the process $P$.

(1) $(a \to P) \xrightarrow{a} P$

**Proof** From idempotence of $\|$. □

Let $E$ be any set of events, and let $Q(x)$ be an expression defining subsequent behaviour for each different $x$ in $E$. The process $(x : E \to Q(x))$ first offers a choice of any event $a$ in $E$, and then behaves like $Q(a)$.

(2) $(x : E \to Q(x)) \xrightarrow{a} Q(a), \quad \text{if } a \in E$

**Proof** From associativity and idempotency of $\|$. □

$P \sqcap Q$ behaves like $P$ or like $Q$, where the selection between them is arbitrary, without the control of the external environment.

(3) $P \sqcap Q \to P$

$P \sqcap Q \to Q$

**Proof** $\sqcap$ is the greatest lower bound operator with respect to $\sqsubseteq$. □

If one of the operands of $\|$ proceeds by resolving non-deterministic choice, then the external choice construct performs the same step, and postpones the selection on its alternatives.

(4) $P \| Q \to P' \| Q, \quad \text{whenever} \quad P \to P'$

$P \| Q \to P \| Q', \quad \text{whenever} \quad Q \to Q'$

**Proof** From monotonicity of $\|$. □

If $P$ and $Q$ can perform an action, then the environment of $P \| Q$ can control which of $P$ or $Q$ will be selected, solely based on the very first action.

(5) $P \| Q \xrightarrow{a} P', \quad \text{whenever} \quad P \xrightarrow{a} P'$

$P \| Q \xrightarrow{a} Q', \quad \text{whenever} \quad Q \xrightarrow{a} Q'$

**Proof** From associativity and monotonicity of $\|$. □

The first step of the sequential composition $P; Q$ is the same as the first step of its component $P$.

(6) $P; Q \to P'; Q, \quad \text{whenever} \quad P \to P'$

**Proof** From monotonicity of ;. □

(7) $P; Q \xrightarrow{a} P'; Q, \quad \text{whenever} \quad P \xrightarrow{a} P'$

**Proof** From Lemma 10.5.2 and law $(a \to P'); Q = a \to (P'; Q)$. □

The first step of $SKIP; Q$ is to discard its first component $SKIP$.

(8) $SKIP; Q \to Q$

**Proof** $SKIP$ is the unit of sequential composition. □

$P\backslash E$ behaves like $P$ except that all occurrences of actions internal to $P$ are concealed, and so is any event in $E$.

(9) $P\backslash E \to P'\backslash E$, whenever $P \to P'$

**Proof** From monotonicity of hiding. □

(10) $P\backslash E \to P'\backslash E$, whenever $P \xrightarrow{a} P'$ and $a \in E$

**Proof** From law

$$((a \to P) \| Q)\backslash(E \cup \{a\}) \sqsubseteq P\backslash(E \cup \{a\})$$ □

(11) $P\backslash E \xrightarrow{a} P'\backslash E$, whenever $P \xrightarrow{a} P'$ and $a \notin E$

**Proof** From Lemma 10.5.3 and 8.2**L8**

$$(a \to Q)\backslash E = a \to (Q\backslash E), \quad \text{if } a \notin E$$ □

When $P$ and $Q$ are assembled to run concurrently, events that are in both their alphabets require simultaneous participation of $P$ and $Q$.

(12) $P \|_{CSP} Q \xrightarrow{a} P' \|_{CSP} Q'$, whenever $P \xrightarrow{a} P'$ and $Q \xrightarrow{a} Q'$

**Proof** From Lemma 10.5.5. □

However, internal events, and events in the alphabet of $P$ but not in the alphabet of $Q$, are of no concern to $Q$.

(13) $P \|_{CSP} Q \to P' \|_{CSP} Q$, whenever $P \to P'$
$P \|_{CSP} Q \to P \|_{CSP} Q'$, whenever $Q \to Q'$

**Proof** From monotonicity of $\|_{CSP}$. □

(14) $P \|_{CSP} Q \xrightarrow{a} P' \|_{CSP} Q$, whenever $P \xrightarrow{a} P'$ and $a \in \mathcal{A}(P) \setminus \mathcal{A}(Q)$
$P \|_{CSP} Q \xrightarrow{a} P \|_{CSP} Q'$, whenever $Q \xrightarrow{a} Q'$ and $a \in \mathcal{A}(Q) \setminus \mathcal{A}(P)$

**Proof** From Lemma 10.5.4. □

(15) $\mu X \bullet P(X) \to P(\mu X \bullet P(X))$

**Proof** $\mu X \bullet P(X)$ is a solution of the recursive equation $X = P(X)$. □

This operational semantics of CSP is susceptible to analysis by strong or by weak bisimulation. They are both congruences, but they fail to validate the full set of algebraic laws. Solutions to all these problems are given in [158, 160].

We have now given separate operational semantics for a process algebra and for sequential programming. A practical programming language like occam [97] has to combine both sequential and concurrent features. An operational semantics for a combined language consists mainly of a combination of the operators of the

two component sublanguages, and of the steps that define how they are to be executed. But unfortunately, it is not so easy to combine theories of programming that have been based on operational semantics. This is because any results that are dependent on induction may also have to be reconsidered. The fragility of induction when languages are extended is a discouragement to the choice of operational semantics as a starting point for unifying theories of programming. In the denotational presentation, all that was necessary was to combine the alphabets of the predicates, and in an algebraic presentation, it is the laws and the healthiness conditions that are combined. Nothing has to be withdrawn and nothing has to be redefined and nothing has to be reproved. That is why this book starts with denotational and algebraic presentations, and ends here with the operational.