Chapter 7

Concurrency

The reduced cost of microprocessor chips has made it economic to obtain yet higher computational performance by harnessing many processors to execute different parts of the same program concurrently. A program component intended for concurrent execution with other such components is called a *process*, and its composition with other processes is known as *parallel* composition, in analogy with the familiar *sequential* composition, which reuses a single processor to execute both components. There are two main classes of concurrent programming paradigm, based on different ways of connecting the hardware processing elements. In a *distributed* architecture, each processor has its private local memory, and communicates with other processors, usually by means of a common bus (often a ring) or by dedicated wires which connect them in pairs. In the *shared*-memory architecture, the processors each have access to the same global memory, which interleaves service requests from all of them in a fair and efficient manner. It is the second kind of paradigm that will be treated in this chapter.

The shared-memory paradigm permits considerable variation in the implementing architecture. For example, it is possible (and often desirable) for a single processing element to share its attention among several processes of a single program using interrupts to switch between them at unpredictable points. This is known as multiprogramming or time-slicing, and is common on conventional single-processor machines. A good theory must be sufficiently abstract that it permits an implementation of concurrency in this way too.

Other possible architectural variations include the provision of multiple memory banks, capable of concurrent service of many access requests. Some (or even all) of these banks may be primarily devoted to service of requests from a single processor; in particular, each processor has exclusive access to its own local bank of special hardware registers intimately connected to its arithmetic unit. It often also has its own local memory cache, providing rapid access to more or less up-to-date copies of information held in the slower global memory.

Most of the complexity of such a memory hierarchy is concealed from the programmer by the hardware, which provides a simulation of a global virtual memory, and a single homogeneous addressing range. Valiant and even expensive measures are taken in hardware design to achieve a defined level of predictability. Nevertheless, non-determinism is an almost inevitable consequence of executing programs on such an architecture. It arises because there is no way of predicting or controlling the order in which the memory will service requests from the many processing elements. Further non-determinism may arise from an unknown delay between changing the value of the local cached copy of a global variable and the transmission of the new value to the global memory, from which it can be read by the other processors. Finally, an optimising compiler or a hardware instruction pipeline may (within certain, maybe not obvious, limits) change the order of execution of assignments within a single process, in a manner which varies from one computer supplier to another, or even from one version of the same product to the next.

The goal of a theory of concurrency is to provide an effective method of reasoning to avoid or to control all such complexity and non-determinism, from whatever source it arises. In this chapter we take the simplest measures to achieve this goal. We maintain a level of abstraction that requires no consideration of the values of variables at any time except at the start and termination of the process. As a result, all finite programs using parallelism can be translated into programs that are purely sequential, and infinite programs can be reduced to the same sequential normal form. Systems which allow intermediate observations after initiation but before termination will be treated in the next chapter.

The simplest cases of parallelism are those that forbid the possibility of sharing store. Each process is allocated a disjoint region, ensuring that it never reads or writes in any region of store except its own. These restrictions are usually enforced by special hardware in a multiprogrammed computer. In our theory, the regions can be defined by partitioning the alphabet between the processes, and the observance of the restrictions can be enforced by a syntactic check. This case is described in Section 7.1.

Section 7.2 relaxes the disjointness condition to allow processes to share one or more variables. This case can be reduced to the disjoint case as follows. Each process is first executed on its *private* version of the shared variables independently. When all have terminated, their updates on the shared variables are *merged* and written back to the *global* version of the shared variables. The parallel by merge can be used to model a variety of shared-variable concurrency paradigms, such as *interleaving* and *synchronisation*; and it also obeys algebraic laws similar to those for disjoint parallelism. We also investigate the conditions on the merge relation which ensure that familiar healthiness conditions can be preserved by parallel composition.

The method described above for implementing the updates on a shared vari-

able has been introduced primarily for reasoning about the overall effect of concurrent execution of two or more processes. In practice, only one copy of the shared variable is kept, and it is updated by interleaving the atomic actions invoked by the processes which share it. The consistency of these two implementation methods must be established by proof of the relevant algebraic laws. Systematic application of these laws can even transform a parallel program into an equivalent sequential one. This often results in massive expansion to make all the non-determinism explicit. Section 7.3 establishes a range of expansion laws applicable to various kinds of merge operation.

Section 7.4 discusses the case where a large global array is shared by parallel processes. The processes are allocated disjoint regions of indices in the array, and each may change array elements in its own region and no other. The rest of the array, which is not allocated for updating by any process, may be freely read by all of them. When these rules are observed, all updates and accesses on the array will commute with each other, and this completely avoids non-determinism and its associated risks.

The disjointness conditions on shared variables deliberately prevent their use for communication between processes. The combined effect of all the updates becomes accessible only after the processes have terminated. To make safe access to intermediate results, it is essential that the processes have the capability to synchronise with each other. Synchronisation is treated in Section 7.5 in its simplest form as a global action in which all processes must engage simultaneously [123, 183].

The final section introduces the logic programming paradigm in a concurrent form. The disjunction operator is implementable by concurrent execution and interleaving as an alternative to the backtracking which characterises ordinary sequential logic programming.

7.1 Disjoint processes

The basic idea of concurrency has been described in Chapter 1, where it was explained by simple conjunction of predicates describing the concurrently operating components. This explanation works just as well for specifications as for implementations. Suppose a customer's needs are described by two separate predicates S and T, each describing a separate physical object, say a shirt and a shoe. The combination of the needs of the customer is therefore described by the conjunction $S \wedge T$. To avoid confusion, it is necessary that each variable in S (describing a property of the shirt) should be distinguishable from each variable in T (describing a property of the shoe).

Definition 7.1.1 (Disjointness)

Two predicates S and T are disjoint if their alphabets have no variables in common

$$\alpha S \cap \alpha T = \{\}$$ □

When a specification $(S \wedge T)$ is expressed as a conjunction of disjoint predicates, it is usually easiest to satisfy the specification by designing and delivering two separate products (call them P and Q), each of which satisfies just one part of the specification

$$\alpha P = \alpha S \text{ and } [P \Rightarrow S]$$
$$\alpha Q = \alpha T \text{ and } [Q \Rightarrow T]$$

The two products will usually be entirely disconnected and independent, and could be delivered at different times to different places; after delivery, they may be used separately or together, one after another, or in overlapping periods of time. That is what justifies the claim made in Chapter 1, that in the case of disjoint predicates the concurrent behaviour of many processes is accurately described by the conjunction of the predicates describing their individual behaviours.

In describing two objects of different kinds, it is reasonable to assume (because it is easy to ensure) that the alphabets are disjoint. But the customer may want two disjoint objects of the *same* kind, say two shirts. This requirement cannot be described by the conjunction of two copies of the same predicate $(S \wedge S)$: firstly, this is identical to the description of just a single object S; and secondly, it violates the rule that the alphabets of concurrent processes must be disjoint. Both of these problems are solved by assigning a different name or serial number (say 0 or 1) to each of the two objects. Then each observation x of the object with name 0 is denoted by prefixing the serial number to the observation name, to give a new double-barrelled or *qualified* observation name $0.x$, and $1.x$ denotes a distinct but similar kind of observation of the other object. In general, if m is a different serial number from n, then we assume $m.x$ is a name distinct from $n.x$; it denotes a similar kind of observation made of a different instance of the same kind of product. Product naming or numbering gives a homogeneous way of forcing the alphabets of predicates to be disjoint. It is important in practical applications, but its role in the development of theory is mainly to demonstrate that the constraint of disjointness is one that can easily be met whenever needed.

Definition 7.1.2 (Labelling)

Let p be a value used as a process name. Then

$$p.P(v) =_{df} P(p.v)$$

where $P(p.v)$ is the result of replacing every occurrence of an observation variable

x in P by an occurrence of the qualified name $p.x$. Accordingly

$$\alpha(p.P) \;=_{df}\; \{p.x \mid x \in \alpha P\}$$

It is assumed that $p.x$ is a different variable from $q.x$ whenever $p \neq q$. □

This definition permits a conjunction of any number of distinct and disjoint instances of S, whether finite or infinite, for example

$$(1.S) \wedge (2.S)$$

$$\bigwedge \{p.S \mid p \in \mathbf{PID}\}, \quad \text{where } \mathbf{PID} \text{ is a set of process names}$$

As explained above, to model concurrency by conjunction permits each process to start and finish completely independently of the other. But in parallel programming, we often want to start the processes at the same time, and allow them to operate on disjoint areas of a store that has been initialised as a whole on completion of some previous part of the program. Equally often, we want the two processes to terminate together, so that the entire store becomes available again for further processing by the following part of the program. This kind of synchronised initiation and termination of processes can be treated only in a theory which already models these phenomena, for example by the variables ok and ok'. We will therefore confine attention to predicates that are designs (Definition 3.1.1). These have distinguishable input and output alphabets, referring to the initial state before execution and the final state afterwards. If P and Q are disjoint relations, the undashed variables of their conjunction $P \wedge Q$ describe a simultaneous observation of the initial state of P and of the initial state of Q, and similarly, the final state of $P \wedge Q$ is given by simultaneous observation of the values of the dashed variables contributed by both processes. This means that execution of a concurrent composition should start with the start of *both* the processes, and similarly, its execution terminates with the termination of *both* of them. This kind of structured concurrency was introduced by Dijkstra with the notation **cobegin** ... **coend** [49]. It is different from the concurrent use of shirts and shoes which may be purchased, worn and discarded without any synchronisation.

There is one unfortunate consequence of the requirement of simultaneous termination: if one of the processes fails to terminate, then so does its composition with any other process whatsoever. As a result, it is not possible to observe the final values, even of variables updated by a process that by itself would have terminated. So in practice, abortion of a component causes abortion of the whole program. But this is not what is predicted by a theory which defines parallel composition as conjunction, because obviously

$$\mathbf{true} \wedge Q \;=\; Q \qquad \text{rather than} \qquad \mathbf{true} \wedge Q \;=\; \mathbf{true}$$

The same problem arose in the original treatment of sequential composition,

which failed to satisfy the necessary zero laws, and we solve it for concurrency in a similar way. This is another reason for restricting attention to *designs*, in which the precondition for termination is made explicit. Since parallel execution involves starting *both* the processes, it is necessary to ensure that *both* their preconditions are valid to start with. Thus the precondition for successful execution of the parallel composition is the *conjunction* of their separate preconditions rather than their disjunction. The corresponding new definition of concurrency will be denoted by parallel bars $\|$.

Definition 7.1.3 (Parallel composition)

Let P and Q have disjoint alphabets, and let $P = P_0 \vdash P_1$ and $Q = Q_0 \vdash Q_1$.

$$\begin{aligned} P \| Q &=_{df} (P_0 \wedge Q_0) \vdash (P_1 \wedge Q_1) \\ \alpha(P \| Q) &=_{df} \alpha P \cup \alpha Q \end{aligned}$$

□

Example 7.1.4 (Parallel assignment)

Let x and y be distinct variables. The parallel assignment

$$(x := f(x)) \| (y := g(y))$$

has the same effect as the multiple assignment

$$\begin{aligned} & (x := f(x)) \parallel (y := g(y)) && \{\text{def of assignment}\} \\ = \ & (\mathbf{true} \vdash x' = f(x)) \| (\mathbf{true} \vdash (y' = g(y))) && \{\text{Def. 7.1.3}\} \\ = \ & (\mathbf{true} \vdash ((x' = f(x)) \wedge (y' = g(y)))) && \{\text{def of assignment}\} \\ = \ & x, y := f(x), g(y) \end{aligned}$$

□

In a strict interpretation of output disjointness, Definition 7.1.3 is invalid, because the variables ok and ok' are in the alphabet of both the component processes. The relaxation of the restriction is justified as follows. The implementation is required to start execution of the parallel composition by starting execution of both the components. Even if one or both components start with an initial delay, we do not need or want to contemplate or observe a time when one of the two processes has started but not the other. That is why only one variable ok is needed to record the synchronised start of both of them, which is the same as the start of the parallel composition. Similar reasoning based on synchronised termination applies to the output variable ok'. The reasoning will later be generalised to other synchronised events like input and output between communicating processes.

Definition 7.1.5 (Disjointness of designs)

Two designs P and Q are disjoint if

$$\alpha P \cap \alpha Q = \{ok, ok'\}$$

□

In Chapter 3, designs were classified according to their degree of healthiness, and all the operators of the language were shown to conserve the healthiness properties of their operands. Fortunately, the new definition of parallelism is equally conservative; the disjointness condition is what makes it so.

Theorem 7.1.6

If P and Q both satisfy **H3** or both satisfy **H4**, then $P \| Q$ has the same property.

Proof Direct from Definition 7.1.3. □

Although the new definition of parallelism is more complicated than conjunction, it obeys nearly all the same algebraic laws, except idempotence, which is excluded by the disjointness condition on the alphabets. In stating the laws, we will implicitly assume that all the obviously intended disjointness constraints are satisfied.

L1 $P \| Q = Q \| P$ (|| comm)

L2 $P \| (Q \| R) = (P \| Q) \| R$ (|| assoc)

L3 $I\!I_X \| I\!I_Y = I\!I_{X \cup Y}$ (||-unit)

L4 $\mathbf{true} \| P = \mathbf{true}$ (||-zero)

It is well known that simple forms of parallel composition can be implemented by (or even transformed into) sequential composition by simply interleaving execution of atomic actions from the participating processes. That is the message of the following distribution laws.

L5 $(P \lhd b \rhd Q) \| R = (P \| R) \lhd b \rhd (Q \| R)$

L6 $(P \sqcap Q) \| R = (P \| R) \sqcap (Q \| R)$

L7 $(\bigsqcup S) \| R = \bigsqcup_n (S_n \| R)$, for any descending chain $S = \{S_n \mid n \in \mathcal{N}\}$

L8 $(x := e; P) \| Q = (x := e); (P \| Q)$

Proof of **L8**

$$
\begin{array}{lll}
 & (x := e); (P \| Q) & \{\text{Def. 7.1.3}\} \\
= & (x := e); ((P_0 \wedge Q_0) \vdash (P_1 \wedge Q_1)) & \{x \notin \alpha Q\} \\
= & (P_0[e/x] \wedge Q_0) \vdash (P_1[e/x] \wedge Q_1) & \{\text{Def. 7.1.3}\} \\
= & (P_0[e/x] \vdash P_1[e/x]) \| (Q_0 \vdash Q_1) & \{\text{Theorem 3.1.4}\} \\
= & (x := e; P) \| Q & \square
\end{array}
$$

The final proof that parallelism introduces nothing new into the language is achieved by showing that parallel programs have the same normal form as shown for sequential programs in Chapter 5.

Theorem 7.1.7 (Normal form of disjoint parallel programs)

Let $S = \{S_n \mid n \in \mathcal{N}\}$ and $T = \{T_n \mid n \in \mathcal{N}\}$ be descending chains of finite normal forms. Then

$$(\sqcup S)\|(\sqcup T) = \sqcup_n (S_n\|T_n)$$

and each $(S_i\|T_i)$ can be reduced to a finite normal form.

Proof

$$\begin{array}{lll} & \sqcup_n\ (S_n\|T_n) & \{(S_i\|T_j) \sqsubseteq (S_{i+j}\|T_{i+j})\} \\ = & \sqcup_{i,j}\ (S_i\|T_j) & \{\mathbf{L7}\} \\ = & (\sqcup S)\|(\sqcup T) & \end{array}$$

Let $P = ((\sqcap_i x := e_i) \lhd b \rhd \mathbf{true})$ and $Q = ((\sqcap_j y := f_j) \lhd c \rhd \mathbf{true})$.
We are going to show how to convert $P\|Q$ into a finite normal form.

$$\begin{array}{lll} & P\|Q & \{\mathbf{L4} \text{ and } \mathbf{L5}\} \\ = & ((\sqcap_i x := e_i)\|(\sqcap_j y := f_j)) \lhd b \wedge c \rhd \mathbf{true} & \{\mathbf{L6}\} \\ = & (\sqcap_{i,j}(x := e_i\|y := f_j)) \lhd b \wedge c \rhd \mathbf{true} & \{\mathbf{L3} \text{ and } \mathbf{L8}\} \\ = & (\sqcap_{i,j}\, x := e_i; y := f_j) \lhd b \wedge c \rhd \mathbf{true} & \{5.1\mathbf{L3}\} \\ = & (\sqcap_{i,j}\, x, y := e_i, f_j) \lhd b \wedge c \rhd \mathbf{true} & \square \end{array}$$

The disjointness condition is designed to prevent any process from reading any variable which is subject to update by any concurrent process. The disjointness condition can therefore be relaxed to allow the input alphabets to share variables that are never updated at all. In fact, it is even possible to share initial values of variables that are updated by just one of the two processes and excluded from the output alphabet of the other. But this places upon an implementation the obligation to make a separate copy of the initial value for the benefit of the process that does not change it. Any update by the other process is therefore invisible: disjointness is effectively preserved, and so is determinacy. The main problem is the expense of making a fresh copy of the initial values.

Exercises 7.1.8

Prove that for disjoint processes

(1) If both P and Q satisfy **H3** and **H4** then $P\|Q = (P;Q) = (Q;P)$.

(2) If both P and Q satisfy **H4** then $(P;R)\|(Q;S) = (P\|Q);(R\|S)$.

(3) $(P; x := e)\|(Q; y := f) = (P\|Q);(x := e\|y := f)$ $\square$

7.2 Parallel by merge

The reason why disjoint parallelism is so simple to implement, and so obviously obeys all the stated laws, is that all the actions of each process commute with all the actions of any process disjoint from it. A parallel implementation can execute disjoint processes by arbitrary interleaving of their actions, or even by sequential execution of the entire processes in either order (Exercise 7.1.10(1)). In spite of this range of implementations, commutativity ensures that there is no increase in non-determinism. Furthermore, disjointness is subject to a syntactic check. In this section we first extend the treatment of parallelism to allow the processes to update a shared variable in various ways, some of them commuting and some not.

Initially, we will deal with a single shared variable at a time. Our plan is to reduce the sharing case of parallelism to the disjoint case. This will ensure that the more complex kind of parallelism obeys the same algebraic laws, and preserves the same healthiness properties. Disjointness is achieved by replacing the shared variables m' in each process by new labelled variables $0.m'$ and $1.m'$. Of course, we assume throughout that these labelled variables are not already in the alphabets of the processes involved. Otherwise we would have used different labels. Under this proviso, the undashed variables can be left unlabelled. Once the shared variables are differentiated by labelling, the processes are disjoint, and they may each produce different results in the variables $0.m'$ and $1.m'$. The final effect of parallel execution is then defined by a *merging* operation, which shows how the real final value of m is computed from these two results, and from the initial value of m. Different kinds of parallelism will use different merging operations, as illustrated by a series of simple canonical examples.

Example 7.2.1 (Shared resource)

Consider an integer variable r (short for *resource*) used to count the total utilisation of some resource, say machine time. Each process updates the value of r by assignments

$$r := r + e \qquad \text{or} \qquad r := r + f$$

where e and f are expressions containing only local variables of the updating process. In the concurrent execution of the processes, the updates contributed by each of them are arbitrarily interleaved. Let us suppose that neither process needs to access any of the intermediate values of r. Then the same effect can be achieved by a disjointly parallel execution, in which each process updates its own private copy $0.r$ or $1.r$ of the variable r by assignments like

$$0.r := 0.r + e \qquad \text{or} \qquad 1.r := 1.r + f$$

where $0.r$ and $1.r$ are initialised to r. The actual final value of r is the same as if

it were computed from $0.r$ and $1.r$ afterwards by an assignment that merges their values

$$r := 0.r + 1.r - r$$

Here subtraction of the initial value r is needed to counteract the initial value assigned to both $0.r$ and $1.r$. □

Example 7.2.2 (Shared log)

In this example, we relax the constraint that all operations on shared variables must commute, and accept that the interleaving of actions by parallel processes can lead to a vast explosion in non-determinism. A particularly common example of sharing is by interleaving of output messages on a display or file, used as a system log. We will represent the log by a specially named variable *out*, which holds as its value the sequence of messages output so far. Each output operation appends to the sequence a new message, with value defined by an expression e

$$out := out^\frown < e >$$

The sharing of an output stream by interleaving is modelled in the same way as in Example 7.2.1. Firstly, separate streams $0.out$ and $1.out$ are declared for use by each process, and they have the global stream as their initial values. Each process then appends messages to its own output stream. When both processes have terminated, the final value of *out* will be found to contain its initial value followed by an interleaving of the sequences of messages output by the separate processes. The final action of parallel composition computes this result from the final values of $0.out$ and $1.out$, contributed by the two processes. The merging operation is therefore

$$\mathbf{true} \vdash (out' - out) \in (0.out - out) ||| (1.out - out)$$

where $s - t$ is the result of moving an initial copy of t from s, and $s|||t$ is the set of all interleavings of s and t. If either s or t is empty, this is easy to define. Otherwise, one of them must contribute the first element of the result, and the rest of the result is obtained recursively

$$<> ||| t = t ||| <> =_{df} t$$

$$(< x >^\frown s) ||| (< y >^\frown t) =_{df} \{< z >^\frown u \mid z = x \wedge u \in (s|||(< y >^\frown t)) \vee z = y \wedge u \in ((< x >^\frown s)|||t)\}$$ □

Example 7.2.3 (Shared clock)

In discrete event simulation on a computer, multiple processes of the program are used to model the concurrent activities of people and objects in the real world,

and then synchronisation in the program represents an action (like shaking hands) in which all the processes involved are engaged simultaneously. For an example of global synchronisation, consider the passage of simulated time, which always has the same (or negligibly different) value accessible to all the processes of the system. We represent this by a shared variable called *clock*, recording the value of the current simulated time, in some convenient unit. The responsibility of each process is to carry out all the actions that are supposed to take place at this current moment of time, and then to wait until the clock moves on to the next moment of time. The tick of the clock is represented within each process by the synchronised action

$$clock := clock + 1$$

In order to use disjoint parallelism, we provide each process with its own private clock $0.clock$ and $1.clock$ (initially synchronised to the public clock), which can be incremented apparently in an independent fashion. When both processes are finished, the public clock is set to the larger of the two private clocks. This represents the fact that the process that terminates earlier no longer has the obligation to update its private clock.

$$clock := max(0.clock,\ 1.clock)$$ □

Example 7.2.4 (Synchronised input)

In certain kinds of real-time application, input data are presented to the computer at regular intervals, say 30 times per second. To a first approximation, we assume that there is enough computing power to deal with input at this rate; indeed, the main purpose of parallelism is to ensure exactly this. The successive values that are input are represented by a sequence-valued input variable in, where in_c is the value that is input on the occasion that the clock has value c. On each occasion, the input is assigned to a fixed shared-memory location m; this kind of memory-mapped input is often implemented in hardware. The effect is modelled abstractly in each process by a simultaneous assignment on (labelled versions of) the variables m and c

$$m,\ c := in_c,\ c + 1$$

As in the case of the clock, these assignments are executed simultaneously in all processes, and they all assign the same value to the shared variables m and c. There is no need to merge the values of m and c at the end, provided that all processes engage in all inputs, and so terminate at the same time. This is an obligation placed upon the programmer by an assertion at the end of parallel execution

$$(0.c = 1.c)_{\perp}\ ;\ c := 0.c$$ □

Example 7.2.5 (Synchronised output)

Synchronised output is treated like synchronised input, with the aid of a counter c and an output sequence *out*. On each tick of the clock, the value assigned to out_c is defined by a local expression e of the outputting process

$$out_c,\ c := e,\ c+1$$

The values of the expressions assigned to out_c by different processes will be different, and the actual output to the environment of both processes will be a combination of the information contributed by each process. The notation $x|y$, denoting the desired combination of the value of x and y, is taken from the Synchronous Calculus of Concurrent Systems (SCCS) [125]; it is assumed to be associative and commutative. The operator is applied to every element of the output sequence. The final values of *out* and c are then defined by the merge

$$(0.c = 1.c)_\perp\ ;$$
$$\mathbf{true} \vdash (\forall i : i < 0.c \bullet out'_i = out_i \lhd i < c \rhd (0.out_i | 1.out_i)) \ \wedge \ (c' = 0.c)$$

This states that messages output before the processes start will remain unchanged, and those output after the start will be the combination of messages output at the same time by the two processes. □

In all these examples, the intended interpretation of the observation of the shared variable is what determines the choice of the appropriate merging operation. It also determines the range of possible actions that are permitted to be performed upon the shared variable. For example, the variable *clock* does not stand for a variable in the store of the computer; it stands for the global time recorded (say) by a caesium clock and transmitted by a radio signal. The intended implementation of the assignment $clock := clock + 1$ is to wait until the next tick of this clock. Such an implementation is certainly permitted, because on completion of the action, the final reading of the clock is one greater than the initial reading. But no such implementation would be possible for any assignment that *decreases* the value of *clock*. For reasons of realism, it is necessary to keep close control over the atomic actions which are allowed on each of the shared variables. We will therefore introduce the letter $\mathcal{A}$ to stand for the set of permitted actions.

A general definition of parallel composition $(P \| Q)$ requires first a definition of the transformation that is made to each of the two processes; each assignment $m := f(m)$ to the shared variable m must be replaced by the corresponding assignment to one of the two private variables

$$0.m := f(0.m) \ \text{ in } P \qquad \text{or} \qquad 1.m := f(1.m) \ \text{ in } Q$$

These modified processes (say P_0 and Q_1) must be preceded by declaration and initialisation of the private variables. The overall effect of the parallel execution is

then described

$$\textbf{var } 0.m,\ 1.m := m,\ m;(P_0 \| Q_1);\ M$$

where M is the appropriate operation for merging the values of $0.m$ and $1.m$.

The description given above ensures the processes P and Q are presented textually as programs in which textual changes can be made to the component assignments. But in the top-down design of programs, we need to apply parallel composition at an earlier stage in the design progression, that is to the predicates $P(m, m')$ and $Q(m, m')$ which specify the individual processes. Fortunately, all that is needed is to substitute m' by $0.m'$ or $1.m'$ in each predicate, giving $P(m, 0.m')$ and $Q(m, 1.m')$. The same substitution must be made in the alphabets of the two processes. It is convenient to define the required effect by a pair of simulations $U0$ and $U1$.

Definition 7.2.6 (Separating simulations)

$$\begin{aligned} U0(m) &=_{df} \textbf{var } 0.m := m;\ \textbf{end } m \\ \alpha U0(m) &=_{df} \{m,\ 0.m'\} \end{aligned}$$

$U1(m)$ and $U2(m)$ may be given a similar definition. □

We will conventionally use the letter M to stand for the predicate that merges the final values $0.m$ and $1.m$ produced by the two processes. The examples show that its input alphabet also needs to include the initial value of the shared variable m. Its output alphabet is just m'; in this way M contains an implicit end to the declarations of $0.m$ and $1.m$ which were introduced by $U0$ and $U1$.

Definition 7.2.7 (Parallel merge)

Let m be the shared variables of programs P and Q

$$\{m\} \ = \ out\alpha P \cap out\alpha Q$$

Let M be a predicate with the alphabet $\alpha P \cup \alpha Q \cup \{0.m,\ 1.m\}$.
Let $A \ = \ out\alpha P \setminus \{m\}$ and $B \ = \ out\alpha \setminus \{m\}$.

$$P \|_M Q \ =_{df} \ ((P; U0(m)_{+A}) \| (Q; U1(m)_{+B}))_{+m};\ M$$ □

Exercise 7.2.8

Show that for healthy predicates P and Q, Definition 7.1.3 is just a special case of Definition 7.2.7, where the shared variable is just ok. □

In order to satisfy 7.1**L1** to **L8**, the merge predicate M needs to be *valid* in the sense defined below.

Definition 7.2.9 (Valid merge)

A merge predicate $M(ok, m, 0.m, 1.m, m', ok')$ is *valid* if it is a design satisfying the following properties

(1) M is *symmetric* in its input $0.m$ and $1.m$

$$(0.m, 1.m := 1.m, 0.m); M \;=\; M$$

(2) M is *associative*

$$(0.m, 1.m, 2.m := 1.m, 2.m, 0.m); M3 \;=\; M3$$

where $M3$ is a three-way merge relation generated by M

$$M3 \;=_{df}\; \exists x, t \bullet M(ok, m, 0.m, 1.m, x, t) \wedge M(t, m, x, 2.m, m', ok')$$

(3) $(\mathbf{var}\ 0.m, 1.m := m, m;\ M) \;=\; I\!I$ □

The purpose of the definition of validity is to prove the familiar laws of disjoint parallelism.

Theorem 7.2.10 (Parallel by valid merge)

If M is valid then $\|_M$ satisfies 7.1**L1** to **L8**, where **L8** needs to include a proviso

L8 $(x := e; P)\|_M Q \;=\; (x := e); (P\|_M Q)$, if $x := e$ does not mention m

Proof of **L2**

$$\begin{aligned}
&(P\|_M Q)\|_M R && \{\text{Def. 7.2.7}\}\\
=\;& (((P\|_M Q); U0)\|(R; U1))_{+m}\,;\, M && \{\text{def of ;}\}\\
=\;& ((((P; U0)\|(Q; U1))_{+m}; M; U0)\\
&\|(R; U2))_{+m}; M[2.m/1.m] && \{\text{Exercise 7.1.8(2)}\}\\
=\;& (((P; U0)\|(Q; U1))\|(R; U2))_{+m};\\
&(M; U0)_{+\{m,2.m\}}; M[2.m/1.m] && \{\text{7.1}\mathbf{L2}\text{ and def of } M3\}\\
=\;& ((P; U0)\|(Q; U1)\|(R; U2))_{+m}; M3 && \{\text{Def. 7.2.9(2)}\}\\
=\;& ((P; U0)\|(Q; U1)\|(R; U2))_{+m};\\
&(0.m, 1.m, 2.m := 1.m, 2.m, 0.m); M3 && \{\text{def of ;}\}\\
=\;& ((P; U2)\|(Q; U0)\|(R; U1))_{+m}; M3 && \{\text{7.1}\mathbf{L2}\}\\
=\;& ((Q; U0)\|(R; U1)\|(P; U2))_{+m}; M3 && \{\text{similar argument}\}\\
=\;& (Q\|_M R)\|_M P && \{\mathbf{L1}\}\\
=\;& P\|_M(Q\|_M R) && \square
\end{aligned}$$

Examples 7.2.11 (Valid merge)

The merge operations of Examples 7.2.1 to 7.2.5 are valid. □

Examples 7.2.1 to 7.2.3 introduced three separate shared variables, and defined separate merging operators for each of them. Examples 7.2.4 and 7.2.5 each introduced a group of variables, which were updated and merged by the appropriate multiple assignments. The general definition of parallelism is formulated to apply equally well to single variables as to groups of variables. The same theory therefore applies to a language which shares all the variables treated in the examples, and maybe more. Of course, a merge operation for a long list of variables is in general more complicated than one for shorter lists. Fortunately, the complexity can often be avoided. If separate merge operations are already defined for each variable (or small group) individually, the appropriate merge operation for the group containing all the variables is just the (necessarily disjoint) parallel composition of all the separate merge operations. The validity of this result is guaranteed by the following theorem.

Theorem 7.2.12 (Composition of valid merge)

If M and N are valid merges of distinct variables m and n respectively, then $M \| N$ is also valid. □

Although this is a simple way of decomposing the complexity of sharing many variables, the effect may be somewhat unexpected. For example, here is a simple parallel program which shares both a clock and an output log

$$(out := out^\frown < 0 >; clock := clock + 1) \| (clock := clock + 1; out := out^\frown < 1 >)$$

One of them appears to output before the tick of the clock and one of them after. But in our theory, each process is simply a multiple assignment to disjoint variables, and there is no obligation to perform the assignments in any particular order. The overall effect of the parallel composition is calculated as

$$\begin{aligned} & (out, clock := out^\frown < 0 >, clock + 1) \| (out, clock := out^\frown < 1 >, clock + 1) \\ = \;& clock := clock + 1; (out := out^\frown < 0, 1 > \sqcap out := out^\frown < 1, 0 >) \end{aligned}$$

The second of the non-deterministic alternatives may be unexpected. If synchronisation were the guaranteed implementation method for the clock, the possibility of the message coming out in the wrong order $< 1, 0 >$ would be ruled out. This would also prohibit an implementation of the log by separate buffers associated with each process. If it is desired to enforce a stricter synchronisation, one method would be to record the clock time in each output message, and redefine interleaving in a way that preserves an ascending sequence of clock times. In general, synchronisation is an atomic action involving all shared variables of the system simultaneously, as described in Section 7.5.

In the following we show the healthiness conditions defined in Chapter 3 are preserved by parallel composition with sharing.

Theorem 7.2.13 (Healthiness conditions of parallel merge)

(1) $\|_M$ preserves **H1** if M satisfies **H1**.

(2) $\|_M$ preserves **H2** if M satisfies **H2**.

(3) $\|_M$ preserves **H3** if $(\Pi_{\{0.m\}} \| \Pi_{\{1.m\}}); M = M; \Pi_{\{m\}}$.

(4) $\|_M$ preserves **H4** if M satisfies **H4**.

Proof of (3)

$$
\begin{array}{lll}
 & (P \|_M Q); \Pi_{\{m\}} & \{\text{Def. 7.2.7}\} \\
= & ((P; U0) \| (Q; U1))_{+m}; M; \Pi_{\{m\}} & \{\text{assumption of } M\} \\
= & ((P; U0) \| (Q; U1))_{+m}; & \\
 & (\Pi_{\{0.m\}} \| \Pi_{\{1.m\}}); M & \{\text{Exercise 7.1.8(3)}\} \\
= & ((P; U0; \Pi_{\{0.m\}}) \| & \\
 & (Q; U1; \Pi_{\{1.m\}}))_{+m}; M & \{U0; \Pi_{\{0.m\}} = \Pi_{\{m\}}; U0\} \\
= & ((P; \Pi_{\{m\}}; U0) & \\
 & \|(Q; \Pi_{\{m\}}; U1))_{+m}; M & \{P \text{ and } Q \text{ satisfy } \mathbf{H3}\} \\
= & P \|_M Q & \square
\end{array}
$$

7.3 The spreadsheet principle

The discussion of the previous sections has suggested two rather different approaches to the implementation of parallelism. The one that more closely follows the theoretical definition requires that separate copies are taken of the initial values of any shared variable, one for each process, and each process updates its own copy. This is the method used for *forking* in the UNIX operating system. Nevertheless, for larger shared variables like arrays, copying could be expensive, and it is clearly impossible for most of the objects in the real world, including the world itself and each of its inhabitants.

A more generally practical method of implementation is to maintain just a single copy of the shared variable, and allow it to be updated by synchronisation or interleaving of the actions of the sharing processes. Care will be needed to prevent interference between interleaved actions; they must be executed serially without risk of overlap. For example, when one action has been started, any attempt by another process to initiate another update of the same variable must be delayed until the first action is complete. This property of actions is called *atomicity*; it is implemented by mutual exclusion of critical regions, but details of the mechanism need not concern us here. Sometimes, atomic actions must be executed in strict

synchronisation, but we will also postpone details of the treatment of synchronisation to the next section and the next chapter.

The main purpose of this section is to demonstrate the compatibility of the theoretical and the practical method of implementing parallelism. An implementation may even use a dynamically varying mixture of the methods to achieve the highest efficiency. To justify this we prove a collection of algebraic laws which permit an atomic action to be moved from the start of the parallel processes to the start of the whole program. These laws have the shape

$$(op; P)\|Q \sqsubseteq op; (P\|Q)$$

or $$(op1; P)\|(op2; Q) \sqsubseteq (op1\|op2); (P\|Q)$$

where *op* is called an *asynchronous* atomic action and (*op*1, *op*2) is called a *synchronising pair*. (In an operational semantics of parallelism, the $\sqsubseteq$ is usually written as $\rightarrow$.) Each such law describes just one of the ways of implementing the parallelism. A collection of laws offers a non-deterministic choice of implementations, which can be made explicit in a single, more complicated derived law, for example

$$(op1; P)\|(op2; Q) \sqsubseteq op1; (P\|(op2; Q)) \sqcap op2; ((op1; P)\|Q)$$

But this law cannot be used to prove correctness of the left hand side of the inequation. For that, the right hand side needs to enumerate *all* permitted implementation methods, so that the inequality can validly be replaced by an equation. For obvious reasons, such equations are known as *expansion* laws. Their systematic application to finite programs can eliminate the parallel combinator in favour of a non-deterministic choice of sequential programs. In an algebraic semantics, the expansion laws would be taken as a definition of parallelism, and the theorems of this section would be an approach to a proof of the consistency of the algebra with our more abstract view of program specification and correctness. The corresponding completeness property is akin to the *full abstraction* of recent models of the typed lambda calculus [5].

To carry through this task, we will need to analyse more carefully the nature of the atomic operations, the way in which they update the shared variable, and the interaction between the update and the merge used to define parallel composition. We will introduce the theory in its simplest form by an analogy with a spreadsheet, laid out as a two-dimensional array containing at the bottom a row of column sums and on the right a column of row sums, and an overall total in the bottom right corner. The total may be computed identically in two different ways. One way is to get separate processes to compute the column sums, and when they have done so, the total is computed from these. This corresponds to the abstract definition of parallel composition by a final merge. The other method is to compute the row sums serially in the order written, and keep a running total on the right hand side. That is the method of calculation suggested by the expansion law. The equality

of the results of these two methods of calculation may be called the *spreadsheet* principle.

Example 7.3.1 (Spreadsheet)

9	7	16
4	3	7
13	10	23

Generalising to a matrix of symbolic variables, consider

p	q	$p+q$
r	s	$r+s$
$p+r$	$q+s$	

The spreadsheet principle can now be expressed algebraically as an equation between the two ways of filling the missing square

$$(p+q)+(r+s) \;=\; (p+r)+(q+s)$$

This is an immediate consequence of associativity and commutativity of $+$. □

Modern spreadsheets are not confined to addition. They allow the rightmost column to be computed by an arbitrary operator specified by the user; for our purpose, we choose $\|$. The operator used to compute the column results on the bottom row may be chosen independently; let this be sequential composition.

p	q	$p \| q$
r	s	$r \| s$
$p;r$	$q;s$	

The spreadsheet property states the uniqueness of the value for the checksquare

$$(p\|q)\,;\,(r\|s) \;=\; (p;r)\,\|\,(q;s) \qquad (*)$$

This is just the exchange law of category theory, which we met in Exercise 2.1.12. In practical use, spreadsheets are certainly not confined to four entries: they may have any number of rows and columns. How can we be sure that these larger tables still have the spreadsheet property? It is sufficient to require that both the composition operators are associative. This means that any pair of adjacent columns in a spreadsheet (excluding the rightmost) may be replaced by their horizontal composition without affecting the correctness of the spreadsheet, and similarly with vertical composition of adjacent rows. Thus any larger matrix can be reduced to a matrix with just four elements.

In its general form, the spreadsheet principle is far too strong. It is violated by many definitions of $\|_M$ which we have given so far. We therefore need to investigate the various conditions under which restricted forms of the principle are valid for particular definitions of $\|_M$. Theories of parallel programming may then be classified according to the restrictions that are needed to ensure a complete reconciliation with their more practical methods of implementation.

The first and most important restriction is to confine attention to programs that can be expressed solely by means of the set $\mathcal{A}$ of allowed actions on the shared variables. Formally, it is the smallest set which contains $\mathcal{A}$ and is closed with respect to all the operators of the programming language. Because of the normal form theorem (Theorem 7.1.7), it is sufficient to characterise the closure by the single assignments to the shared variable that are possible in the normal form. The set of such assignments will be called $\mathcal{A}^+$.

Example 7.3.2 (Resource)

For the shared resource (Example 7.2.1), the atomic actions all have the form

$$\mathcal{A} \;=\; \{r := r + e \mid e \text{ does not mention } r\}$$

The set $\mathcal{A}^+$ is exactly equal to $\mathcal{A}$, as may be proved by the reductions

$$\begin{aligned} (r := r + e); (r := r + f) &= r := r + (e + f) \\ (r := r + e) \lhd b \rhd (r := r + f) &= r := r + (e \lhd b \rhd f) \\ (r := r + e) \| (r := r + f) &= r := r + (e + f) \end{aligned}$$

In the case of the conditional, we need to take advantage of the general restriction that a condition must not mention a shared variable. The shared resource is the simplest of examples, because it satisfies the spreadsheet principle in the greatest generality. All that is needed is to restrict the variables in the law $(*)$ to the members of $\mathcal{A}^+$

$$\begin{aligned} & (r := r + e\,;\, r := r + f) \| (r := r + g\,;\, r := r + h) \\ = \;& r := r + e + f + g + h \\ = \;& (r := r + e \| r := r + g)\,;\, (r := r + f \| r := r + h) \end{aligned}$$

This simple proof extends automatically to all programs, because it holds between normal forms. □

Example 7.3.3 (Clock)

For the clock (Example 7.2.3), the atomic actions are more restricted than for the resource, consisting just of

$$clock := clock + 1$$

The closure $\mathcal{A}^+$ is similar to that for the resource, except that the expression added to the clock must be strictly positive. As a result the unit Π is not a member of $\mathcal{A}^+$, and a special law is necessary to guide implementation in this case

$$P \| \Pi \;=\; P, \quad \text{for all } P \in \mathcal{A}^+$$

Furthermore, the spreadsheet principle is severely restricted to the case where the first two operands are equal

$$(P;Q) \| (P;S) \;=\; (P \| P);(Q \| S), \quad \text{for } P \in \mathcal{A}^+$$

Implementation of equal actions in parallel takes advantage of idempotence

$$P \| P \;=\; P \qquad \square$$

Example 7.3.4 (Shared log)

Atomic updates on the shared log (see Example 7.2.2) have the form

$$out \;:=\; out^\frown < e >$$

and their closure has the form

$$out \;:=\; out^\frown s$$

where s is a sequence-valued expression not containing *out*. In this case the spreadsheet principle can simply be weakened to an inequation, indicating that parallel composition is less deterministic than sequential

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

To obtain an equation, the non-determinism must be made explicit, and the first two actions must be restricted to atomic actions

$$(p;Q) \| (r;S) \;=\; p;(Q \| (r;S)) \sqcap r;((p;Q) \| S)$$

provided that p and r are atomic actions. Since Π is not in the closure, we need also the unit law

$$P \| \Pi \;=\; P \qquad \square$$

Exercise 7.3.5

Define $\mathcal{A}$ and $\mathcal{A}^+$ for Examples 7.2.4 and 7.2.5. Formulate and prove the relevant expansion laws. $\square$

7.4 Shared array

Preservation of the principle of commutativity is an obligation that must sometimes be passed to the programmer. Consider for example a common requirement in an engineering and scientific calculation, which updates some large global array A. In each phase of the calculation, it may be possible to define certain disjoint regions within the array which need updating, and allocate each region to a separate process. Each process may change array elements in its own region and no other. The rest of the array, which is not allocated for updating to any process, may be freely read by all of them. If these rules are observed, all atomic actions on the array and all accesses to it from different processes will commute with each other.

Definition 7.4.1 (Disjoint regions)

Let W and R be subsets of the index set of the array A. The pair $r =_{df} (W, R)$ is called a *region* of A.
Regions $r0 = (W0, R0)$ and $r1 = (W1, R1)$ are *disjoint* if

$$W0 \cap (W1 \cup R1) = \{\} \quad \text{and} \quad W1 \cap (W0 \cup R0) = \{\} \qquad \square$$

In the examples of Section 7.2, the restriction to commuting operations is checkable by a scan of the processes involved. But in the case of a shared array, it is the programmer who has to discharge all the obligations of disjointness which are needed to validate our relatively simple theory of concurrency. There is no way that any general kind of static analysis could check the disjointness of the run-time values of the indices used to access the elements of the array.

In a language which permits disjoint sharing of arrays among parallel processes, we have to introduce a slight complexity into the declaration of an array, and into the definition of access and assignment to a subscripted variable. The declaration of an array with index range I and values of type T may be written

$$\textbf{array}\ A : I \to T$$

This is interpreted as introducing two additional variables $A.W$ and $A.R$, which range over subsets of I; they are initialised to the whole set

$$\textbf{var}\ A.W, A.R := I, I$$

In a parallel process, the values of the private copies of these variables will be reduced to the actual region allocated and accessible to the process. Any attempt to access A outside $A.R$ will fail, as ensured by the definition

$$\mathcal{D}(A[i]) =_{df} (i \in A.R)$$

Similar failure occurs on an attempt to assign to a subscripted variable with an index outside $A.W$

$$(A[i] := e) \ =_{df} \ (i \in A.W)_{\perp} ; \ A := (A \oplus \{i \mapsto e\})$$

The parallel composition operator ${}_r\|_s$ is now decorated with two subscripts, indicating the regions in which the two processes operate. The merge operation uses the two write regions to indicate which updates have to be written back into the final value of the array.

$$M \ =_{df} \ (A := A \oplus (r_0 \lhd 0.A) \oplus (s_0 \lhd 1.A)) ;$$
$$\textbf{end}\, 0.A,\, 1.A$$

where r_0 and s_0 are the W components of the pairs r and s, and $X \lhd A$ represents the subarray of A with its domain restricted to indices in X, and $\oplus$ is the overriding operator on functions.

The full definition of parallel composition has to adjust the value of $A.R$ and $A.W$ in the obvious way before executing each of the processes. But their global values remain unchanged.

Definition 7.4.2 (Shared array with disjoint regions)

Let r and s be disjoint regions and let M be as defined above.

$$P \,{}_r\|_s Q \ =_{df} \ (A.W,\, A.R := r_0 \cap A.W,\, r_1 \cap A.R ;\, P)$$
$$\|_M \, (A.W,\, A.R := s_0 \cap A.W,\, s_1 \cap A.R ;\, Q) \qquad \square$$

Theorem 7.4.3

The parallel composition of shared array with disjoint regions satisfies 7.2**L2** to **L8**, and the adapted commutative law

L1a $P \,{}_r\|_s Q \ = \ Q \,{}_s\|_r P$

It also obeys the following expansion law for commuting updates

L9 $(op; P) \,{}_r\|_s Q \ = \ \hat{op} ; \, (P \,{}_r\|_s Q)$

where

$$\hat{op} \ =_{df} \ (\exists AW', AR' \bullet op[A.W \cap r_0,\, A.R \cap r_1 / A.W,\, A.R])_{+\{A.W,\, A.R\}} \qquad \square$$

7.5 Synchronisation

In this section, we transfer attention from interleaving to synchronisation, which we introduce in its purest form as a single action

sync

By definition, this can take place only with simultaneous participation by all active processes. If there are no variables shared between the synchronising processes, their synchronisation is vacuous. The only purpose of synchronisation is to force a merge of the values of all the shared variables, and so to make sure that the same result is accessible to all the processes of the system. This gets round the deficiency of all the interleaving models considered so far: that no process can ever reliably access any result produced by any other concurrent process. Synchronisation is surely a necessary condition of reliable communication. Otherwise there is the grave risk of accessing a necessary result before it has actually been computed.

The intended effect of synchronisation is most clearly explained by an algebraic law. Let M be the merge operation for *all* the shared variables m of the system. Let P describe the behaviour of one process up to its first **sync** action, and let Q be the initial non-synchronising behaviour of the other process. Then the **sync** action invokes the merge operation M to consolidate the results of P and Q in their global store, so that results computed separately in m by each of them are available subsequently to both of them. The synchronisation action is retained to deal with the possibility that there are three or more processes involved. This informal account is summarised in the following expansion law

L9 $(P; \mathbf{sync}; R) \|_{\tilde{M}} (Q; \mathbf{sync}; S) \;=\; (P \|_M Q); \mathbf{sync}; (R \|_{\tilde{M}} S)$

provided that P and Q do not contain **sync**

Here the tilde over the M is meant to indicate that M is executed not just once at the end of parallel execution but also at all the intermediate synchronisation points. The main goal of this section is to define a meaning for the tilde and the **sync**, so as to ensure validity of this law. But first some examples:

Example 7.5.1 (Clocked hardware)

In the synchronous paradigm of hardware design, a clock signal is distributed regularly to all storage elements (registers). Between clock signals, the combinational circuitry performs its intended calculation with maximal concurrency. But the results of the calculation are not stored back into the registers until the clock signal arrives. The stored results become available to all combinational circuits only at the beginning of the next cycle. The length of the cycle is adjusted to ensure there is time for all the combinational circuits to stabilise before the next clock signal.□

Example 7.5.2 (BSP)

Section 7.4 describes how an array may be updated in parallel by splitting it into disjoint regions. In a scientific simulation, it is usually necessary to iterate a series of such updates, and use a different split into regions on each update. This is what permits information to propagate (perhaps more slowly than one would like) from one region of the array to other more remote regions. The treatment of regions in Section 7.4 requires all processes to terminate before any reallocation can be made. But in this section it becomes possible to perform the reallocation on each occasion of global synchronisation. This pattern of parallel programming is known as the Bulk Synchronous Paradigm (BSP) [183, 123], and it offers two advantages in high performance computing.

1. It is relatively easy to predict and optimise the performance of the system as a whole.
2. The effectiveness of the optimisation is relatively independent of the details of the implementing configuration and architecture. □

Example 7.5.3 (Cache memories)

In a modern microprocessor, there is a fast cache memory interposed between the arithmetic unit and the slower external main store. Any value read from a location in main store is held in the cache for a while, so that any later read operation from the same location may be satisfied more rapidly than by repeated access to main store. Any value written to memory is also held in the cache until the main store is ready to accept it. An operation is provided in the machine code that causes the processor to wait until all cached writes are written back through to main store. Only then is it known that the contents of the cache are in complete agreement with the main store.

When a single main store is shared among many processors, its workload increases proportionally. To balance the increase, each processor needs its own large and independent cache. The exact timings of reading from and writing to the shared main store become even more unpredictable. To achieve reliable communication between the processors, the waiting operation must act like **sync**. It must be obeyed simultaneously by all the processors, and it must also empty all the caches. This certainly ensures that the cache is up-to-date with the main store: it is the only reasonable way of doing so. It is only after the synchronisation that changes made by other processes can be reliably accessed from the shared main store. □

The formal definition of parallelism by synchronisation uses a combination of the techniques of Examples 7.2.4 and 7.2.5, though the input and output involved is purely conceptual. A sequence variable *out* is used by each process to record the value of its own copy of the shared variable m on each occasion of synchronisation. These values are combined by the merge operator M, playing the role of

the SCCS | [125]. The merged result is recorded in the input sequence *in*, and so made available to all the contributing processes simultaneously. As before, a count is maintained of the serial number of the synchronisation.

Definition 7.5.4 (Sharing with synchronisation)

The atomic operation **sync** is defined as a simultaneous memory-mapped input and output

$$\mathbf{sync} \;=_{df}\; (c,\, out_c,\, m \;:=\; c+1,\, m,\, in_c)$$

The final merge $\widetilde{M}$ simply applies M to every element of the output sequences. Assuming M is feasible, the final merge is also implementable.

$$\begin{aligned}\widetilde{M} \;=_{df}\;& ((0.c = 1.c)_{\perp}\,;\; (c := 0.c)\\ &\|\; M(m,\, 0.m,\, 1.m,\, m')\\ &\|\; \{M((m \lhd i = c \rhd in_{i-1}),\, 0.out_i,\, 1.out_i,\, out'_i) \mid c \le i < 0.c\}\\ &\|\; \{\mathit{II}_{\{out_i\}} \mid i < c\})\;;\\ &\mathbf{end}\ 0.c,\, 1.c,\, 0.out,\, 1.out\end{aligned}$$

Here $\| \{P_i \mid l \le i < n\}$ stands for the parallel program $(P_l \| P_{l+1} \| \ldots \| P_{n-1})$. □

The first line of this definition of $\widetilde{M}$ deals with the clock in the usual way. The second line says that the final value of the shared variable m is computed by the merge operation M. The third line states that on each occurrence of **sync**, the value of *output* is computed on the assumption that m had the value input on the previous **sync** (if any).

Theorem 7.5.5

If M is valid so is $\widetilde{M}$, and $\|_{\widetilde{M}}$ satisfies **L9**, and 7.1**L1** to **L8**.

Proof of **L9** Consider the case where

$$P \;=_{df}\; (m := e_0(m))$$

$$Q \;=_{df}\; (m := f_0(m))$$

and

$$R \;=_{df}\; m,\, out,\, c \;:=\; e_1(m),\, out\hat{}s,\, c+i$$

$$S \;=_{df}\; m,\, out,\, c \;:=\; f_1(m),\, out\hat{}t,\, c+j$$

Then we have

$$(P; \mathbf{sync}; R) \;=\; (m,\, out,\, c \;:=\; e_1(in_c),\, out\hat{} < e_0(m) >\hat{}s,\, c+i+1) \qquad (\dagger)$$

$$(Q; \mathbf{sync}; S) \;=\; (m,\, out,\, c \;:=\; f_1(in_c),\, out\hat{} < f_0(m) >\hat{}t,\, c+j+1) \qquad (\ddagger)$$

$$
\begin{aligned}
& (P \|_M Q); \mathbf{sync}; (R \|_{\widetilde{M}} S) && \{\text{def of } P,\ Q \text{ and } \|_{\widetilde{M}}\} \\
= \ & M(m,\ e_0(m),\ f_0(m),\ m')\,;\ \mathbf{sync}; \\
& \widetilde{M}((c,\ out,\ m),\ (c+i,\ out^\frown s,\ e_1(m)), \\
& (c+j,\ out^\frown t,\ f_1(m)),\ (c',\ out',\ m')) && \{\text{def of } \widetilde{M} \text{ and } \mathbf{sync}\} \\
= \ & \widetilde{M}((c,\ out,\ m), \\
& (c+i+1,\ out^\frown < e_0(m) >^\frown s,\ e_1(in_c)), \\
& (c+j+1,\ out^\frown < f_0(m) >^\frown t,\ f_1(in_c)), \\
& (c',\ out',\ m')) && \{(\dagger) \text{ and } (\ddagger)\} \\
= \ & (P; \mathbf{sync}; R) \|_{\widetilde{M}} (Q; \mathbf{sync}; S) && \square
\end{aligned}
$$

All the definitions given so far would be entirely applicable to a system like those of Examples 7.2.4 and 7.2.5, which engages in real input and real output with the external environment. The only extra constraint is that input and output take place simultaneously. In particular, the values from the sequence *in* are completely unconstrained. But for our present proposes we want all the communication and synchronisation to be internal to the system itself. This is done by declaring the clock c and the sequences *out* and *in* to be local variables of the program which uses them. They are declared implicitly, together with a declaration of the variable m, whose sharing is achieved by synchronisation. But most important of all, we want the values of the input messages not just to be arbitrary values plucked from the environment; instead, they must be exactly the same messages that have been jointly output by all the synchronising processes on exactly the same cycle! This apparently instantaneous feedback is characteristic of hardware description languages (e.g. Esterel [24, 25], Signal [72]). It is easily specified: the initial value of the *in* stream must be the same as the final value of the *out* stream. The implementation would be easy too, if it were able to guess magically in advance what the final value of the output stream was going to be.

Definition 7.5.6

$$
\begin{aligned}
(\mathbf{shared}\ m;\ P;\ \mathbf{end}\ m) \ =_{df} \ & \mathbf{var}\ in,\ out,\ c\,;\ (out,\ c := <>,\ 0); \\
& (P \wedge (in = out'))\,;\ \mathbf{end}\ in,\ out,\ c \qquad \square
\end{aligned}
$$

To show how this works, consider the following example where P is used to compute the maximal delay.

Example 7.5.7

Let $P \ = \ (time := time + r1\,;\ \mathbf{sync}\,;\ time := time + r2) \|_{\widetilde{M}}$
$\qquad\qquad (time := time + s1\,;\ \mathbf{sync}\,;\ time := time + s2)$

where $M \ =_{df} \ time := max(0.time,\ 1.time)$.

We are going to show that this has the expected effect

$$\textbf{shared } time; P; \textbf{ end } time \;=\; time := time + max(r1,\, s1) + max(r2,\, s2)$$

$$\begin{array}{lll}
 & P & \{\text{def of } \textbf{sync} \text{ and } 3.1\textbf{L2}\} \\
= & (c,\, out_c,\, time \;:=\; c+1,\, time + r1,\, in_c + r2)\|_{\widetilde{M}} & \\
 & (c,\, out_c,\, time \;:=\; c+1,\, time + s1,\, in_c + s2) & \{\text{Def. } 7.5.4\} \\
= & c,\, out_c,\, time \;:= & \\
 & c+1,\, time + max(r1,\, s1),\, in_c + max(r2,\, s2) & (\dagger)
\end{array}$$

The assumption that $in_c = out_c$ ensures that the variable *time* gets the expected final value. Everything else is then hidden.

$$\begin{array}{lll}
 & \textbf{shared } time;\, P;\, \textbf{end } time & \{\text{Def. } 7.5.6\} \\
= & \textbf{var } in,\, out,\, c;\, (out,\, c := <>,\, 0); & \\
 & ((in = out') \wedge P); \textbf{end } in,\, out,\, c & \{(\dagger) \text{ and } 3.1\textbf{L2}\} \\
= & \textbf{var } in,\, out,\, c; & \\
 & ((in_0 = time + max(r1,\, s1)) \;\wedge & \\
 & (c,\, out_0,\, time := 1,\, time + max(r1,\, s1),\, in_0 & \\
 & +max(r2,\, s2))); \textbf{end } in,\, out,\, c & \{\text{def of assignment}\} \\
= & \textbf{var } in,\, out,\, c; & \\
 & c,\, out_0,\, time \;:=\; 1, m + max(r1,\, s1),\, time + max(r1,\, s1) & \\
 & +max(r2,\, s2); \textbf{end } in,\, out,\, c & \{2.9\textbf{L6} \text{ and } 2.9\textbf{L8}\} \\
= & time \;:=\; time + max(r1,\, s1) + max(r2,\, s2) & \square
\end{array}$$

Exercise 7.5.8

Prove that if P is a program that does not contain **sync** then

$$\textbf{shared } m;\, P; \textbf{sync}; Q\,; \textbf{ end } m \;=\; P\,; \textbf{ shared } m;\, Q; \textbf{ end } m \qquad \square$$

7.6* Concurrent logic programming

The conventional logic programming paradigm is one which allows the program to give not just a single result of a computation, but rather a whole sequence of results, depending on how many are wanted on each occasion. So the desired behaviour can be specified as a relation L with alphabet $\{q, a'\}$, where q stands for the initial observation (usually called a question), and a' stands for the resulting

sequence of answers. We use J, K and L to range over predicates with such an alphabet. Note that these are non-homogeneous relations: the input and output variables have different types and different names. As a result, sequential composition cannot be used directly between logic programs, though with care it can be used in reasoning about them.

The simplest logic program is the one that always gives an empty sequence of answers. This is used to indicate that the question is in face unanswerable; it is therefore used to represent the answer **no**.

Definition 7.6.1 (no)

$$\mathbf{no} \;=_{df}\; a :=<> \qquad \square$$

The opposite answer is **yes**; it is given to a question that is so simple that it serves as its own answer.

Definition 7.6.2 (yes)

$$\mathbf{yes} \;=_{df}\; a :=< q > \qquad \square$$

Any homogeneous relation $Q(q, q')$ can easily be transformed to a logic program $LQ(q, a')$ which gives q' as its only answer. This is done by composition with **yes**

$$LQ(q, < q' >) \;=_{df}\; Q; \mathbf{yes}$$

Other primitive operations of a logic programming language can be defined in this simple way, provided that they give just a single answer to any question.

In order to construct longer lists of answers, a concurrent logic language uses the interleaving operator $|||$, which merges the lists of answers given by its two operands.

Definition 7.6.3 (Parallel or)

Let K and L be logic programs. Define

$$K|||L \;=_{df}\; K \underset{Por}{\|} L$$

where

$$Por \;=_{df}\; \mathbf{true} \vdash a' \in (0.a|||1.a)$$

Because $|||$ permits a concurrent implementation, the language is said to be or-parallel. $\square$

This operation is known as disjunction, and it shares many of the algebraic properties of the logical operator with the same name (except idempotence).

Theorem 7.6.4

$|||$ is commutative, associative, and has **no** as its unit. It is also disjunctive and has **true** as its zero. □

A sequential logic programming language like PROLOG has a version of disjunction which can be defined as an extreme special case of interleaving. This is obtained by simple concatenation, where all the answers produced by the first operand come before any produced by the second. This reduces the benefit of concurrent implementation, because if the second operand produces answers before the first, they have to be saved up until after the first has finished.

Definition 7.6.5 (Sequential or)

$$K \textbf{ or } L =_{df} K \underset{Sor}{\|} L$$

where the merge operation Sor is defined by

$$Sor =_{df} \textbf{true} \vdash a' = 0.a \frown 1.a$$ □

This operator enjoys all the properties of parallel or except that it is no longer commutative. It also differs slightly from the true PROLOG disjunction, because it has **true** as its right zero. In PROLOG this is not so. As a result, PROLOG allows infinite sequences of answers. For example, the PROLOG program

$$(\mu X \bullet \textbf{yes or } X)$$

will give as its answers an infinite sequence of copies of its question. The above definition gives a much worse response, namely abortion.

Example 7.6.6 (Factorisation)

Questions and answers are just positive integers. $F0$ is defined to respond to a question by giving one of its proper factors (say the largest one that does not exceed the square root), and $F1$ gives the other proper factor. If the number is prime, both $F0$ and $F1$ say no. Then $(F0|||F1)$ also gives a no answer to a prime; otherwise it gives two answers whose product equals the original question. The order of the answers is arbitrary. For example,

$$(q := 13; F0) = \textbf{no}$$

$$(q := 14; F0) = a :=< 2 >$$

$$(q := 14; F1) \;=\; a :=< 7 >$$

$$(q := 14; (F0 \,|||\, F1)) \;=\; (a :=< 2,7 >) \sqcap (a :=< 7,2 >) \qquad \square$$

Any logic program $L(q, a')$ can be converted to a homogeneous relation $L^*(a, a')$ on sequences of answers. This is done by applying L to each of the questions in the input sequence a, and concatenating the sequences of all the answers that result.

Definition 7.6.7 (Star)

$$L^* \;=_{df}\; \mu X \bullet \mathbf{no} \lhd a =<> \rhd (q := head(a); L)\ \mathbf{or}\ (a := tail(a); X) \qquad \square$$

In fact L^* is the unique fixed point of the above equation. The recursion conceals the simplicity of the idea, which is revealed better in the following algebraic laws.

Theorem 7.6.8

L1 $\mathbf{yes}^* \;=\; \Pi_{\{a\}}$

L2 $\mathbf{no}^* \;=\; (a :=<>)$

L3 $\mathbf{true}^* \;=\; (a =<>)_\perp$

L4 $\mathbf{no};\, L^* \;=\; \mathbf{no}$

L5 $\mathbf{yes};\, L^* \;=\; L$

L6 $a := (a_0 \,\hat{}\, a_1);\, L^* \;=\; (a := a_0;\, L^*)\ \mathbf{or}\ (a := a_1;\, L^*)$

L7 $(K; L^*)^* \;=\; K^*; L^*$

Proof of L7

$$\begin{aligned}
& K^*; L^* && \{\text{Def. 7.6.7}\}\\
&= (\mathbf{no} \lhd a =<> \rhd ((q := head(a); K) \\
&\qquad \mathbf{or}\ (a := tail(a); K^*)));\, L^* && \{2.2\mathbf{L2}\}\\
&= (\mathbf{no}; L^*) \lhd a =<> \rhd (((q := head(a); K) \\
&\qquad \mathbf{or}\ (a := tail(a); K^*));\, L^*) && \{\mathbf{L4}\ \text{and}\ \mathbf{L6}\}\\
&= (\mathbf{no} \lhd a =<> \rhd ((q := head(a); K; L^*) \\
&\qquad \mathbf{or}\ (a := tail(a); K^*; L^*)) && \{\mathbf{L6}\}
\end{aligned}$$

which together with the uniqueness of the fixed point of the defining equation of the star operator implies **L7**. $\square$

The star helps in defining the other main operator of logic programming, which is called conjunction (**and**). This is akin to sequential composition, in that the answers produced by the first operand are fed as questions to the second operand. The final result is a concatenation of all the sequences of answers produced by the second operand.

Definition 7.6.9 (Conjunction)

$$K \textbf{ and } L \;=_{df}\; K; L^*$$ □

The algebraic properties of this operator reveal its close similarity with sequential composition between homogeneous relations.

Theorem 7.6.10

and is associative and has **yes** as its unit. As left zeros it has both **true** and **no**. It distributes leftward through sequential or. □

But **true** is not a right zero of **and**, because when the first operand gives an empty answer the second operand is never started

$$\textbf{no}; \textbf{true}^* \;=\; \textbf{no}$$

Conjunction **and** does not distribute rightward through **or**, because the right operand may have to be executed many times, once for each of the answers produced by the left operand.

The following distribution laws are useful in computing the result of a logic program.

Theorem 7.6.11

L8 $q := e; (K \,|||\, L) \;=\; (q := e; K) \,|||\, (q := e; L)$

L9 $q := e; (K \textbf{ or } L) \;=\; (q := e; K) \textbf{ or } (q := e; L)$

L10 $Q; (K \textbf{ and } L) \;=\; (Q; K) \textbf{ and } L$ □

Example 7.6.12 (Complete factorisation)

Using conjunction and recursion, it is now possible to write the program FAC, producing a complete list of prime factors of its question. $F0$ and $F1$ do the real work, together with a primality test to terminate the recursion.

$$FAC \;=_{df}\; \mu X \bullet \textbf{yes} \lhd prime(q) \rhd ((F0 \,|||\, F1) \textbf{ and } X)$$

An example of a simple calculation shows how this works

$$\begin{aligned}
& q := 182 \;;\; FAC && \{\text{def of } FAC\} \\
= \;& q := 182 \;;\; ((F0 \,|||\, F1) \textbf{ and } FAC) && \{\textbf{L8} \text{ and } \textbf{L10}\} \\
= \;& ((q := 182 \;;\; F0) \,|||\, (q := 182 \;;\; F1)) \textbf{ and } FAC && \{\text{def of } F0, F1\} \\
= \;& (a :=< 13 > \,|||\, a :=< 14 >) \textbf{ and } FAC && \{\text{Def. 7.6.3}\} \\
= \;& (a :=< 13, 14 > \sqcap a :=< 14, 13 >) \textbf{ and } FAC && \{2.4\textbf{L6}\}
\end{aligned}$$

$$
\begin{array}{lll}
= & (a :=< 13, 14 > \textbf{ and } FAC) \sqcap & \\
 & (a :=< 14, 13 > \textbf{ and } FAC) & \{\textbf{L6}\} \\
= & (a :=< 13 >; FAC^* \textbf{ or } a :=< 14 >; FAC^*) \sqcap & \\
 & (a :=< 14 >; FAC^* \textbf{ or } a :=< 13 >; FAC^*) & \{\text{def of } FAC\} \\
= & a :=< 13, 2, 7 > \ \sqcap\ a :=< 13, 7, 2 > \sqcap & \\
 & a :=< 2, 7, 13 > \ \sqcap\ a :=< 7, 2, 13 > & \square
\end{array}
$$

On occasion, it is known that no more than one answer will ever be required for a question. Clearly it is a waste of time to compute unwanted answers, so PROLOG provides a *cut* operation to guide a more efficient implementation. The cut has often been criticised as illogical or worse. However, it presents no difficulty in our theory. It is a deterministic homogeneous relation on sequences of answers.

Definition 7.6.13 (Cut)

$$! =_{df} a := (< head(a) > \lhd a \neq <> \rhd <>) \qquad \square$$

Theorem 7.6.14

L11 $\textbf{no}; ! = \textbf{no} \qquad \textbf{yes}; ! = \textbf{yes}$

L12 $!;! = !$

L13 $(K \textbf{ or } L);! = ((K;!) \textbf{ or } (L;!));!$

L14 $(K \textbf{ and } L);! = (K \textbf{ and } (L;!));!$

L15 $(K|||L);! = ((K;!)|||(L;!));!$

Proof of **L13** Define

$$
\begin{array}{rl}
0.! & =_{df} \ 0.a := (< head(0.a) > \lhd 0.a \neq <> \rhd <>) \\
1.! & =_{df} \ 1.a := (< head(1.a) > \lhd 1.a \neq <> \rhd <>) \\
e & =_{df} \ (< head(0.a), head(1.a) > \lhd 1.a \neq <> \rhd < head(0.a) >) \\
 & \qquad \lhd 0.a \neq <> \rhd (< head(1.a) > \lhd 1.a \neq <> \rhd <>)
\end{array}
$$

$$
\begin{array}{lll}
 & Sor;! & \{\text{Def. 7.6.5}\} \\
= & (a := 0.a \frown 1.a);! & \{\text{Def. 7.6.13 and 3.1}\textbf{L2}\} \\
= & a := head(0.a) \lhd 0.a \neq <> \rhd & \\
 & (head(1.a) \lhd 1.a \neq <> \rhd <>) & \{\text{def of } e \text{ and 3.1}\textbf{L2}\} \\
= & a := e;! & \{\text{def 0.!, 1.! and 3.1}\textbf{L2}\} \\
= & (0.!||1.!); Sor;! &
\end{array}
$$

which implies

$$
\begin{aligned}
& (K \textbf{ or } L);! && \{\text{Def. 7.6.5}\} \\
= \ & ((K;U0(a))\|(L;U1(a)));Sor;! && \{Sor;! = (0.!\|1.!);Sor;!\} \\
= \ & ((K;U0(a))\|(L;U1(a)));(0.!\|1.!);Sor;! && \{\text{Exercise 7.1.8(2)}\} \\
= \ & ((K;U0(a);0.!)\|(L;U1(a);1.!));Sor;! && \{U0(a);0.! = !;U0(a)\} \\
= \ & ((K;!) \textbf{ or } (L;!));! && \square
\end{aligned}
$$

The last three distributive laws are useful in optimising execution by inhibiting answers that can never be used.

The most controversial of all PROLOG operators is negation. It turns an empty list of answers into **yes** and any non-empty list into **no**.

Definition 7.6.15 (Negation)

$$\neg L =_{df} L;(\textbf{yes} \lhd a =<> \rhd \textbf{no}) \qquad \square$$

$\neg$ is disjunctive and distributes through conditional. It also has fixed point **true**.

Theorem 7.6.16

L16 $\neg\textbf{yes} = \textbf{no} \qquad \neg\textbf{no} = \textbf{yes} \qquad \neg\textbf{true} = \textbf{true}$

L17 $\neg\neg\neg L = \neg L$

L18 $\neg(K \lhd b \rhd L) = (\neg K) \lhd b \rhd (\neg L)$

L19 $\neg(L;!) = (\neg L) = (\neg L);!$

L20 $\neg(K|||L) = (\neg K) \textbf{ and } (\neg L)$, provided that $L;\top = \top$ $\square$

Negation provides a means of replacing the conditional. For example, $\neg F0$ gives the answer **yes** to any prime, so the factorising program could be rewritten

$$\mu X :: \neg F0|||((F0|||F1) \textbf{ and } X)$$

The termination of this recursion depends crucially on **L4**: **no and** L = **no**.

In this section, most of the operators of the logic programming language have been defined in terms of operators which are known to reduce to the normal form of Chapter 5. The only exception is negation, for which we must prove

Theorem 7.6.17

Let $S = \{S_n \mid n \in \mathcal{N}\}$ be a descending chain of finite normal forms. Then

$$\neg(\sqcup S) = \sqcup_n (\neg S_n)$$

where $\neg S_n$ can be converted into a finite normal form.

Proof
$$
\begin{aligned}
& \neg(\sqcup S) && \{;\text{ is continuous and Def. 7.6.15}\} \\
= \ & \sqcup_n \neg S_n
\end{aligned}
$$

In the following we show how to convert $\neg S_n$ into a finite normal form.

$$\begin{aligned}
& \neg(\sqcap_i a := f_i(q) \lhd b \rhd \mathbf{true}) && \{\mathbf{L18}\} \\
= \; & (\neg \sqcap_i a := f_i(q)) \lhd b \rhd \neg\mathbf{true} && \{\neg \text{ is disjunctive}\} \\
= \; & \sqcap_i(\neg a := f_i(q)) \lhd b \rhd \neg\mathbf{true} && \{\text{Def. 7.6.15}\} \\
= \; & (\sqcap_i a := (<> \lhd f_i(q) \neq <> \rhd < q >)) \lhd b \rhd \mathbf{true} && \square
\end{aligned}$$

There is nothing in the theory which prevents combined use of the parallel and the sequential form of disjunction. The advantage of the sequential form is that it gives closer control over the order of evaluation, which is very important if the later answers are going to be cut. Indeed, successful use of parallel or sequential searches may depend on the programmer giving hints about the sensible degree of priority allocated to the two branches. The disadvantage of the parallel **or** is that it introduces a high degree of non-determinism. This can be very easily eliminated by regarding the results of the program as a *bag* (or even as a set) rather than a sequence. Then the interleaving in the definition of disjunction is replaced by the deterministic operation of bag union. All the non-determinism is now concentrated in the cut operation, which selects an arbitrary member of the bag. Even this can be eliminated if the programmer promises never to apply cut except to a bag with only one member. This promise can be enforced by redefining cut to result in abortion when the promise fails

$$! =_{df} (\# a \leq 1)_\perp$$

This treatment of logic programming completely ignores one of the most delightful features of PROLOG: that it makes no fixed distinction between its input and output variables. For example, the same predicate

$$a = b \hat{} c$$

can be used to test the truth of the equation, to compute a by concatenation of b and c, to compute b from a and c, or c from a and b, or even to compute from a the list of all pairs (b, c) satisfying the equation.

Exercise 7.6.18

Write a specification of each of the four modes of use of the PROLOG program for catenation described above. Using the notations of this section, write a program that implements the fourth of them. □