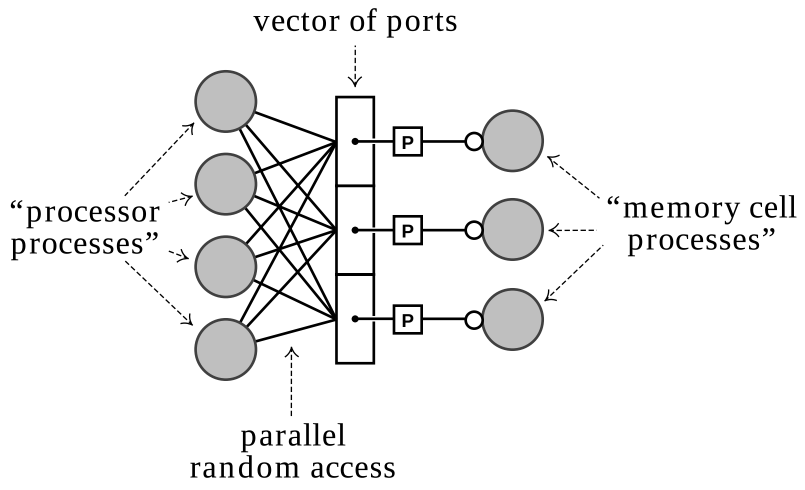

Figure 7.11. PRAM with AKL ports

```
cellproc([], _) :-
    → true.
cellproc([read(V0) | S], V) :-
    → V0 = V,
      cellproc(S, V).
cellproc([write(V) | S], _) :-
    → cellproc(S, V).
cellproc([exch(V1, V2) | S], V) :-
    → V2 = V,
      cellproc(S, V1).
```

PRAM is achieved by creating an array of ports to cells (Figure 7.11).

```
memory(M) :=
      M = m(C1, …, Cn),
      cell(C1), …, cell(Cn).
```

Any number of processes can share this array and send messages to its memory cells in parallel, updating them and reading them. The random access is achieved through the random access to slots in the array, and the fact that we can send to embedded ports without updating the array. Sequencing is achieved by the processors, using continuations as above.

7.6.2 RAM without Ports

Most logic programming languages do not even allow modelling RAM, as a consequence of the single-assignment property. Shapiro's simulator for a RAM [1986b] depends on a built-in *n*-ary stream distributor to access cell processes in constant time as above. In KL1 the MRB scheme allows a vector to be managed efficiently, as long as it is single-referenced [Chikayama and Kimura 1987].

```
memory(M) := new_vector(M, n).
```

A program may access (read) the vector using the

vector_element(*Vector*, *Position*, *Element*, *NewVector*)

operation (which preserves the single-reference property). Sequencing is achieved through continuing access on *NewVector*. Similarly, a program may modify (write) the array using the

set_vector_element(*Vector*, *Position*, *OldElement*, *NewElement*, *NewVector*)

operation (which also preserves the single-reference property). Sequencing can be achieved as above.

7.6.3 PRAM without Ports?

If MRB (or *n*-ary stream distributors) and multiway merging are available, they can be used to model PRAM, but with a significant memory overhead.

Each processor-process is given its own vector (or *n*-ary stream distributor) of streams to the memory cells. All streams referring to a single memory cell are merged. Sequencing is achieved as above. Thus we need O(*n m*) units of storage to represent a PRAM with *n* memory cells and *m* processors.

The setup of memories is correspondingly more awkward.

```
memories($M_1$, …, $M_m$) :=
        memvector($M_1$, $C_{11}$, …, $C_{1n}$),
        …,
        memvector($M_m$, $C_{m1}$, …, $C_{mn}$),
        cell($C_1$), …, cell($C_n$),
        merge($C_{11}$, …, $C_{1m}$, $C_1$),
        …,
        merge($C_{n1}$, …, $C_{nm}$, $C_n$).

cell(S) :=
        cellproc(S, 0).

memvector(M, $C_1$, …, $C_n$) :=
        new_vector($M_1$, $n$),
        set_vector_element($M_1$, 1, _, $C_1$, $M_2$),
        …,
        set_vector_element($M_n$, $n$, _, $C_n$, M).
```

Memory is accessed and modified as in a combination of the two previous models (Figure 7.12). A stream to a memory cell is accessed using the KL1 vector operations. A message for reading or writing the cell is sent on the stream, and the new stream is placed in the vector. An isomorphic structure can also be achieved using *n*-ary stream distributors.

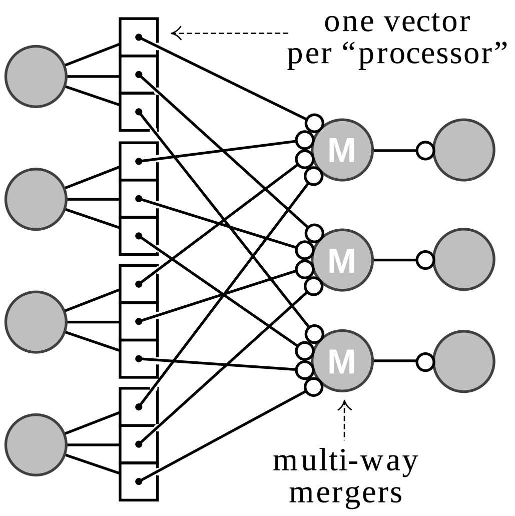

Figure 7.12. PRAM without AKL ports

7.7 EXAMPLES

The first two examples, which are due to Barth, Nikhil, and Arvind [1991], exhibit the need for *parallel random access* functionality in a parallel programming language. The two examples, histogramming a tree of samples and graph traversal, exemplify basic computation structures common to many different settings.

Barth, Nikhil, and Arvind contrast random access solutions with pure functional programs, showing clearly that the former are an improvement both in terms of the total number of computation steps and in terms of the length of the critical path (in “maximally” parallel executions). The compared programs can be expressed in AKL with and without ports, respectively. Only the parallel random access solution with ports is shown; its alternatives without ports can be expressed in many different ways.

The next two sections discuss Lisp and Id, which are functional languages extended with side effect operations, and suggest how these extensions can be simulated in AKL using ports.

The last three sections discuss concrete formalisms for concurrent programming, Actors, Linda, and Erlang, and show how their particular modes of communication can be realised in AKL using ports.

7.7.1 Histogramming a Tree of Samples

Given a binary tree in which the leaves contain numbers in the range 1, …, *n*, count the occurrences of each number.

In our solution, the counts for the numbers are in a PRAM memory of the kind defined above, which we assume has indices in the given range. The program traverses the tree, and, for each number found, increments the value in the cell with this index. The short circuit guarantees that all nodes have been counted before returning the memory.

```
hist(T, M) :-
        memory(M0),
        count(T, M0, S0, S),
        ( S0 = S → M = M0 ).

count(leaf(I), M, S0, S) :-
    → arg(I, M, C),
        send((exch(K, K+1) & S = S0), C).
count(node(L,R), M, S0, S) :-
    → count(L, M, S0, S1),
        count(R, M, S1, S).
```

7.7.2 Graph Traversal

Given a directed graph in which the nodes contain unique identifiers and numbers, compute the sum of the numbers in nodes reachable from a given node. Assume that the identifiers are numbers in the range 1, ..., *n*. Any number of computations may proceed concurrently on the same graph.

In our solution, the nodes which have been traversed are marked in a separate array. For simplicity we assume this to have the indices in the given range, whereas a better solution would employ hashing to reduce its size. Assume that the memory agent returns a memory of this size. A graph node is an expression of the form

```
        node(I, K, Ns)
```

where I is the unique node identifier, K is a number (to be summed), and Ns is a list of neighbouring nodes. (Note that cyclic structures are not a problem in the constraint system of rational trees.)

```
sum(N, G, S) :=
        memory(M),
        traverse(N, G, M, S).

traverse(node(I, K, Ns), M, S) :-
        arg(I, M, C),
        send(exch(1, X), C),
        (       X = 1
            →   S = 0
        ;       traverse_list(Ns, M, S) ).

traverse_list([], _, S) :-
    → S = 0.
traverse_list([N | Ns], M, S) :-
    → traverse(N, M, S1),
        traverse_list(N, M, S2),
        S = S1 + S2.
```

7.7.3 Lisp

Many functional languages, such as Standard ML, CommonLisp, and Scheme, support various forms of *side effect* operations. Whether or not such practices are regarded with favour by purists, they have allowed the integration of object-oriented capabilities, e.g., CLOS into CommonLisp, which substantially enhance Lisp's expressive power.

AKL models side effects using *ports*. If it is assumed that assignment is confined to variables and to explicitly declared objects, an efficient translation of such Lisp programs into AKL is possible, which apart from allowing reuse of old code may also be seen as a parallelisation technique for Lisp programs.

Assume given a translation of the functional subset of Lisp along the lines presented in the previous section. Assume further that an “object-oriented” extension provides operations of the form “(put-field *x v*)” and “(get-field *x*)” for the assignable fields of its objects. Clearly, these operations have to be serialised in the order given by sequential execution of Lisp, but other operations can remain unaffected.

The translation of a Lisp expression of the form

(r … (p …) (q …))

will yield an AKL program of the form

…, p(…, R_1), q(…, R_2), r(…, R_1, R_2, R_3), …

If we assume that p, q, and r perform side effect operations, these can be serialised by adding two new arguments, chained in the order of execution of the original Lisp program, e.g.,

…, p(…, R_1, T_0, T_1), q(…, R_2, T_1, T_2), r(…, R_1, R_2, T_2, T_3), …

These arguments are used to pass a token between side effecting operations; each of these will wait for the token, perform the operation, and then pass the token along. Assuming that assignable fields in objects are memory cells with continuation-calling ports as defined in Section 7.5.1, the put-field and get-field operations may be written as

```
put_field(record(…, X, …), V, token, T) :-
    → send((write(V) & T = token), X).

get_field(record(…, X, …), V, token, T) :-
    → send((read(V) & T = token), X).
```

It should be quite obvious how to proceed to a general translation of such features, and no space will be wasted on this exercise here.

As an aside, the concept of *monads* provides a clean explanation of some well-behaved forms of side effect programming (among many other things) [Wadler 1990]. It remains to be seen if it suffices to explain, e.g., CLOS.

7.7.4 Id

In the taxonomy of Barth, Nikhil, and Arvind [1991], there are three approaches to parallel execution of functional languages:

> *Approach A*: purely functional (strict or non-strict) with implicit parallelism or annotations for parallelism.
>
> *Approach B*: strict functional, sequential evaluation, imperative extensions, and threads and semaphores for parallelism.
>
> *Approach C*: non-strict functional, implicitly parallel evaluation, M-structures, and occasional sequentialisation.

The style of functional programming discussed in Section 3.4.1 provided parallelism in a manner adhering to Approach A. Erlang, discussed for its process-oriented features below, is reminiscent of Approach B. M-structures are available in the parallel non-strict functional language Id, which is the only instance of approach C [Barth, Nikhil, and Arvind 1991]. Note that the approach presented in the previous section does not fall within any of these categories.

M-structures give Id the ability to express arbitrary PRAM-algorithms, just as ports do for AKL. The port examples shown above were modelled upon corresponding Id programs, using techniques that do not immediately correspond to M-structures. In this section, we show that M-structures can easily be expressed in terms of ports.

An *M-field* is a value cell in a structured data type, such as a record or an array. Such a cell is either *full* or *empty*. The *take* operation atomically reads the contents of a full cell and sets its state to empty. If the cell is empty, the take operation suspends. The *put* operation atomically stores a value in an empty cell and sets its state to full. If there are suspended take operations, one is resumed. If the cell is full, the put operation suspends. A data structure that contains M-fields is called an *M-structure*.

An M-field can be modelled in AKL as follows.

```
make_mfield(P) :=
        open_port(P, S),
        separate(S, Ps, Ts),
        match(Ps, Ts).

separate([], Ps, Ts) :-
    → Ps = [], Ts = [].
separate([put(U) | S], Ps, Ts) :-
    → Ps = [U | Ps1],
        separate(S, Ps1, Ts).
separate([take(V) | S], Ps, Ts) :-
    → Ts = [V | Ts1],
        separate(S, Ps, Ts1).
```

```
match([], _) :-
    |  true.
match(_, []) :-
    |  true.
match([U | Ps], [V | Ts]) :-
    |  V = U,
       match(Ps, Ts).
```

The state of the M-field (*full* or *empty*) is implicit in the program. If there are available put requests, then it is full, otherwise it is empty.

Since Id provides implicit parallelism, the order of take and put operations is quite undefined. The synchronisation provided by these operations is often sufficient, but sometimes general *sequencing* is desirable and Id provides a sequencing operator for this purpose. Here short-circuiting would be appropriate for a general translation of Id, whereas *continuations* (Section 7.5.1) are sufficient for most programming purposes.

Finally, note that the implementation scheme for ports described in Section 7.4.4 allows simple low-level optimisations, by which the efficiency of the implementation of a port-based object such as an M-structure can be radically improved. In this case, it would be possible to use the implementation techniques proposed for Id.

7.7.5 Actors

The notion of Actors, as the intuitive notion of the essence of a process, was introduced by Hewitt and Baker [1977], and was further developed by Clinger [1981] and Agha [1986]. It was a source of inspiration for the conception of the concurrent logic programming languages. A mapping from Actors to AKL is shown here to make clear the ease with which the actor style of concurrency and “mailbox” communication can be expressed AKL.

A pure view of Actors is given. Any language based on Actors will be augmented by the power of the chosen host formalism, such as Scheme in the Act family of actor languages [Agha and Hewitt 1987], but the process-oriented component remains the same.

An *actor* has an *address* to which it may receive *communications*. We assume that communications have *identifiers* and *arguments*, which are the addresses of other actors. It has *acquaintances* which are other actors whose addresses it knows. Upon receiving a communication

1) communications may be sent in response, to its acquaintances or to arguments of the communication,
2) new actors may be created, with acquaintances chosen from the acquaintances of the actor, the arguments of the communication, and other new actors,
3) one of these actors may be appointed to receive further incoming communications on the original address.

An actor is defined by specifying its behaviour for each relevant type of communication.

It is easy to map the above notion of an actor into AKL using *ports* as follows.

The definition of an actor A is mapped to a definition of an AKL agent A. The address of an actor is a port (P), which is created for each new actor, and to which communications are sent. The acquaintances (F_1, …, F_k) of A are supplied in the other arguments of A.

```
A(P, F_1, …, F_k) :=
        open_port(P, S),
        A_behaviour(S, F_1, …, F_k).
```

The agent A_behaviour consumes the corresponding stream (S) of communications, which are constructor expressions of the form m(X_1, …, X_l), where X_i are ports. Its definition contains one clause of the following form for each possible type of incoming communication.

```
A_behaviour([m(X_1, …, X_l) | S], F_1, …, F_k) :-
    → send(M_1, …), …, send(M_m, …),
        new_N_1(P_1, …), …, new_N_n(P_n, …).
        A′_behaviour(S, …).
```

The communications sent are M_1, …, M_m. The new actors created are calls to new_N_1, …, new_N_n, and A′ receives further communications on S. The communications sent, their arguments, and the remaining arguments to the atoms representing new actors, are chosen from X_1, …, X_l, F_1, …, F_k, P_1, …, P_n, as given by the original agent definition.

Each definition also contains a clause

```
A_behaviour([], F_1, …, F_k) :- | true.
```

which terminates the actor when its address, the port, is no longer accessible, and has been closed automatically.

A major deficiency of actor languages in comparison with AKL, and other concurrent (constraint) logic programming languages, is that the elegance of implicitly concurrent programs such as quicksort is not readily reconstructed in an actor-based language, with its explicit and strictly unordered form of communication.

7.7.6 Linda

Linda is a general model for (asynchronous) communication over a conceptually shared data structure, the *tuple space* [Gelernter et al. 1985; Carriero and Gelernter 1989]. In a tuple space, *tuples* are stored, which are similar to constructor expressions in AKL, records in Pascal, and the like. There are three operations which access the tuple space. In the following *x* stands for an expression, *v* for a variable, and the notation {X | Y} for either X or Y.

out(x_1, …, x_n)

stores a tuple in the tuple space,

in({var v_1 | x_1}, …, {var v_n | x_n})

retrieves (and deletes) a matching tuple from the tuple space, suspending if none was found, and

read({var v_1 | x_1}, …, {var v_n | x_n})

locates (but leaves in place) a matching tuple in the tuple space. The notation “var v” means that the variable is constrained by the matching operation; the other arguments should be equal to the corresponding slots in the matching tuple.

A tuple space can be modelled as an AKL agent. Processes access the tuple space using a port, the corresponding stream of which is consumed by the tuple space agent. The tuple space is represented as a list of tuples. Tuples may be arbitrary objects, which are selected using arbitrary conditions, not only plain matching. Note the use of the n-ary call primitive for this purpose. A condition is a constructor expression of the form

$p(x_1, …, x_m)$

When “applied to” an argument using ternary *call*

call($p(x_1, …, x_m), x, y$)

it becomes the agent

$p(x_1, …, x_m, x, y)$

which in the following context is expected to return a continuation on y if x satisfies the condition, otherwise it should fail. In general, all n-ary call operations may be regarded as implicitly defined by clauses

call(p(X_1, …, X_m), Y_1, …, Y_{n-1}) :- → p(X_1, …, X_m, Y_1, …, Y_{n-1}).

for all p / n type (program and constraint) atoms in a given program.

The *tuple space server* accepts a stream of “in” and “out” requests. (“read” is omitted, being a restricted form of “in”.)

```
tuple_space([], _, []) :-
    → true.
tuple_space([in(P) | R], TS, S) :-
    → in(P, TS, S, TS1, S1),
      tuple_space(R, TS1, S1).
tuple_space([out(T) | R], TS, S) :-
    → out(T, R, TS, S, TS1, S1),
      tuple_space(R, TS1, S1).
```

The *in request* is served by scanning through the list of tuples looking for a matching tuple. If one is found, the continuation is called, otherwise the request is stored in the list of suspended requests.

```
in(P, [], S, TS1, S1) :-
    → TS1 = [],
      S1 = [P | S].
in(P, [T | TS], S, TS1, S1) :-
      call(P, T, C)
    → call(C),
      TS1 = TS,
      S1 = S.
in(P, [T | TS], S, TS1, S1) :-
    → TS1 = [T | TS2],
      in(P, TS, S, TS2, S1).
```

The *out request* is served by scanning through the list of suspended in requests looking for a matching request. If one is found, its continuation is called, otherwise the tuple is stored in the list of tuples.

```
out(T, TS, [], TS1, S1) :-
    → S1 = [],
      TS1 = [T | TS].
out(T, TS, [P | S], TS1, S1) :-
      call(P, T, C),
    → call(C),
      TS1 = TS,
      S1 = S.
out(P, TS, [M | S], TS1, S1) :-
    → S1 = [M | S2],
      out(P, TS, S, TS1, S2).
```

As a simple example, consider the following condition

```
integer(Y, X, C) :-
      integer(X)
    | C = (Y = X).
```

and the call

```
      tuple_space([in(integer(Y)), out(a), out(1)], [], [])
```

which returns

```
      Y = 1
```

after first suspending the in(integer(Y)) request, then trying it for out(a), which fails and is stored in the tuple space, then for out(1), which succeeds.

Of course, the above simple scheme can be improved. As it stands, the single tuple space server is a bottle neck, which unnecessarily serialises all accesses. This can be remedied, e.g., by distributing requests to different tuple-spaces using a hashing function, but the generality of the representation of conditions and tuples in the above program would have to be sacrificed to allow for a suitable hashing function.