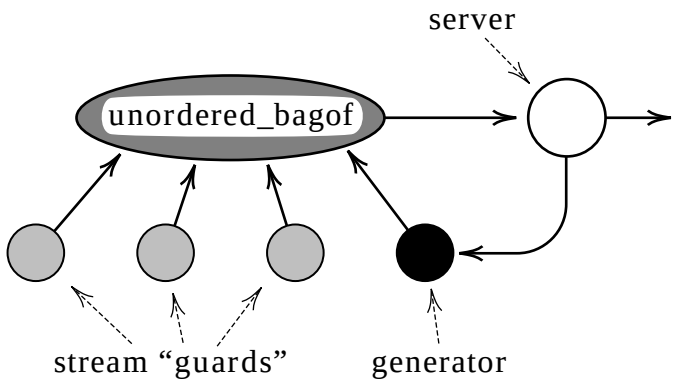

Figure 7.9. Constant delay multiway merger in AKL

```
generator(I, [S | R]) :-
    → (    true
         ? I = S,
           ( I = [] → true
           ; I = [_,_] → true
           ; I = split(_,_) → true )
       ;   true
         ? generator(I, R) ).
```

The server agent expects a stream of instantiated streams from the bagof agent. Apart from dealing with these in the above mentioned way, it also keeps track of the number of merged streams, and closes the output stream and terminates the bagof agent when none remain.

```
server([[E | R] | B], A, S, N) :-
    |   S = [E | S1],
        A = [R | A1],
        server(B, A1, S1, N).
server([split(S1,S2) | B], A, S, N) :-
    |   A = [S1,S2 | A1],
        server(B, A1, S, N+1).
server([[] | B], A, S, N) :-
    |   server(B, A, S, N-1).
server(B, A, S, 0) :-
    |   A = [],
        S = [].
```

The first clause deals with the “normal” case, when a message has been sent on the input stream. The message is added to the output stream, and the rest of the input stream is fed back to the generator. The second clause deals with the case when an input stream is split in two. Both are fed back to the generator, and the number of input streams is incremented by one. The third clause deals with the case when an input stream is closed. The number of input streams is decremented by one. The fourth clause deals with the case when there are no remain-

ing input streams. The output stream is closed, and [] is fed back into the generator to thereby terminate the bagof.

This section presented an implementation of a multiway merger in AKL, but does not provide new solutions to the problems discussed in the previous section. It can be suspected that the problems of merging and multiple-writers are inherent in streams, and people have therefore looked for alternative data types.

7.3.3 Mutual References

Shapiro and Safra [1986] introduced *mutual references* to optimise multiple writers, and as an implementation technique for constant delay multiway merging. The mental model is that of multiple writers.

A shared stream *S* is accessed indirectly through a *mutual reference Ref*, which is created by the allocate_mutual_reference(*Ref, S*) operation. Conceptually, the mutual reference *Ref* becomes an alias for the stream *S*. The message sending and stream closing operations on mutual references are provided as built-in operations. The stream_append(*X*, *Ref, New_Ref*) operation will bind the end-of-stream of *Ref* to the list pair [*X* | *S'*], returning *New_Ref* as a reference to the new end-of-stream. The stream *S* can be closed using the close_stream(*Ref*) operation, which binds the end-of-stream to the empty list [].

An advantage of this is that the mutual reference can be implemented as a pointer to the end-of-stream. When a message is appended, the pointer is advanced and returned. If a group of processes are sending messages on the same stream using mutual references, they can share the pointer, and sending a message will always be an inexpensive, constant time operation. Mutual references can be used to implement a constant delay multiway merger. Another advantage is that an inactive sender will no longer have a reference to old parts of the stream. This makes it possible to deallocate or reuse consumed parts of the stream.

A disadvantage is that we cannot exclude the possibility that the stream has been constrained from elsewhere, and that stream_append has to be prepared to advance to the real end-of-stream to provide multiple writers behaviour.

Otherwise, mutual references has the advantages of multiple writers, but the difficulty of closing the stream remains. It is also unfortunate that the mental model is still that of competition instead of co-operation.

7.3.4 Channels

Tribble et al [1987] introduced *channels* to allow multiple readers as well as multiple writers.

A channel is a partially ordered set of messages. The write(M, C_1, C_2) operation imposes a constraint that: (1) the message M is a member of the channel C_1, and (2) M precedes all messages in the channel C_2. The read(M, C_1, C_2) operation selects a minimal (first) element M of C_1, returning the remainder as the chan-

nel C_2. The empty(C) operation tests the channel C for emptiness. The close(C) operation imposes a constraint that the channel C is empty.

In the intended semantics, messages have to be labelled to preserve message multiplicity, and only minimal channels (without superfluous messages) satisfying the constraints are considered.

Channels seem to share most of the properties of multiple writers on streams. Thus, all messages have to be retained on an embedded channel, in case it will be read in the future. An inactive sender causes the same problem. Closing is just as explicit and problematic. The multiple readers ability can be achieved by other means. For example, a process can arbitrate requests for messages from a stream conceptually shared by several readers.

7.3.5 Bags

Kahn and Saraswat [1990] introduced *bags* for the languages Lucy and Janus.

Bags are multisets of messages. There is no need for user-defined merging, as this is taken care of by the Tell constraint bag-union $B = B_1 \cup \cdots \cup B_n$. A message is sent using the Tell constraint $B = \{m\}$. A combination of these two operations, $B = \{m\} \cup B'$, corresponds to sending a message on a stream, but without the order of messages being given by the stream. A message is received by the Ask constraint $B = \{m\} \cup B'$.

Note that bags can be implemented as streams, with a multiway merger as bag-union. Therefore, it is not surprising that bags have most of the disadvantages of streams with multiway merging. The host languages Lucy and Janus only allow single-referenced objects and therefore suffer less from these problems. It is possible to reuse list pairs in the multiway merger, and to deallocate (or reuse) list pairs when a bag is consumed.

7.4 PORTS

We propose *ports* as a solution to the problems with previously proposed communication media. In this section, we first define their behaviour. We then discuss their interpretation. Finally, we describe their implementation.

7.4.1 Ports Informally

A *port* is a connection between a bag of messages and a corresponding stream of these messages (Figure 7.10, next page). A bag which is connected via a port to a stream is usually identified with the port, and is referred to as a port.

The open_port(P, S) operation creates a bag P and a stream S, and connects them through a port. Thus, P becomes a port to S. The send(M, P) operation adds a message M to a port P. A message which is sent on a port is added to its associated stream with constant delay. When the port becomes garbage, its associated stream is closed. The is_port(P) operation recognises ports.

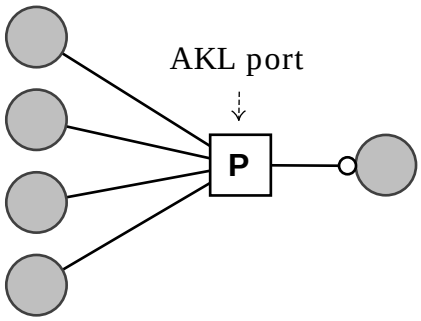

Figure 7.10. An AKL port

A first simple example follows. Given the program

```
p(S) := open_port(P, S), send(a, P), send(b, P).
```

calling

```
p(S)
```

yields the result

```
S = [a,b]
```

Here we create a port and a related stream, and send two messages. The order in which the messages appear could just as well have been reversed.

Ports solve all of the problems mentioned for streams and others.

- No merger has to be created; a port is never split.
- Several messages can be sent on the same port, which means that ports can be embedded.
- Message sending delay is constant.
- Senders cannot refer to old messages and thus prevent garbage collection.
- A port is closed automatically when there are no more potential senders.
- The transparent forwarding problem is solved, since messages can be distributed without inspection.
- Messages can be sequenced, as described in Section 7.5.

7.4.2 Ports as Constraints

We can provide a sound and intuitive interpretation of ports as follows. Ports are bags (also called multisets). The open_port and send operations on ports are constraints with the following reading. The open_port constraint states that all members in the bag are members in the stream and vice versa, with an equal number of occurrences. The send constraint states that an object is a member of the bag. The is_port constraint states that $\exists S$ open_port(*P*, *S*) for a bag *P*.

Our method for finding a solution to these constraints is don't care nondeterministic. Any solution will do. Therefore, the interpretation in terms of constraints is not a complete characterisation of the behaviour of ports, just as

Horn clauses do not completely characterise the behaviour of commit guarded clauses. In particular, it does not account for message multiplicity, nor for their "relevance", i.e. it does not "minimise" the ports to the messages that appear in a computation.

A logic with resources could possibly help, e.g., Linear Logic [Girard 1987]. It can easily capture the don't care nondeterministic and resource sensitive behaviour of ports. The automatic closing requires much more machinery. If such an exercise would aid our understanding remains to be seen.

The method is also incomplete. A goal of the form

$$\text{open\_port}(P_1, S_1), \text{open\_port}(P_2, S_2), P_1 = P_2$$

or of the form

$$\text{open\_port}(P_1, S_1), (\text{open\_port}(P_2, S_2), P_1 = P_2 \rightarrow \ldots)$$

cannot be solved. With the constraint interpretation, this would amount to unifying S_1 and S_2, but to be able to do this it is necessary to keep the whole list of messages sent throughout the lifetime of a port. Instead, this situation may be regarded as a runtime error.

Goals of the form

$$\text{open\_port}(P, S), (\text{send}(M, P) \rightarrow \ldots)$$

or of the form

$$\text{send}(M, P), (\text{open\_port}(P, S) \rightarrow \ldots)$$

can also not be solved. These situations do not make sense pragmatically, and are left unsolved. Both cases can be detected in an implementation, and could be reported as a runtime errors.

The AKL computation model does not, however, in its present form support don't care nondeterministic or incomplete constraint solving. An alternative approach is to describe the behaviour of port operations by *port reduction rules*, which assign unique identifiers to ports, and only make use of constraints for which it is reasonable for constraint solving to be complete [Franzén 1994]. This corresponds to the behaviour of the AGENTS implementation of ports, and is also the approach taken in the following formalisation.

7.4.3 Port Reduction Rules

Following Franzén [1994], we define the operational semantics of ports in terms of rewrites on port operations and on (parts of) the local constraint store of an and-box, to which correspond D-mode transitions in the AKL computation model. The syntactic categories are extended so that port operations may occur in the position of a program atom in programs and in configurations.

In the following, the letter *n* stands for a natural number.

A constraint

$$\text{port}(v, n, w, w')$$

means that v is a port with identifier n, associated list w, and w' is a tail of w. (To conform exactly with the syntax for constraints in the computation model, the parameter n would have to be expressed as a variable constrained to be equal to a number, but this is relaxed here.)

Observe that the following rules strictly accumulate information in the constraint store; the right hand side always implies the left hand side.

The first rule opens a port.

$$\textbf{and}(\text{R}, \text{open\_port}(v, w), \text{S})_{\text{V}}^{\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{\text{port}(v, n, w, w)\wedge\sigma}$$

where n is a closed term that does not occur in R, S, or χ.

The second rule enters a constraint that recognises ports.

$$\textbf{and}(\text{R}, \text{is\_port}(v), \text{S})_{\text{V}}^{\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{\exists u \exists w \exists w' \text{ port}(v, u, w, w')\wedge\sigma}$$

The third rule consumes a send to a port, moving the message to its associated stream. Observe that this rule monotonically adds information to the constraint store. Although the send constraint is removed, it is still implied by the presence of the message in the stream. Observe that the constraint store is composed by the associative and commutative conjunction operator ($\wedge$), and that constraint atoms may be reordered as appropriate for the application of rules.

$$\textbf{and}(\text{R}, \text{send}(u, v'), \text{S})_{\text{V}}^{\text{port}(v, n, w, w')\wedge\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{\text{port}(v, n, w, w'')\wedge w'=[u \mid w'']\wedge\sigma}$$

if $\sigma\wedge\text{env}(\chi)$ entails $v = v'$.

The fourth and final rule closes the associated stream when the port only occurs in a single port constraint.

$$\textbf{and}(\text{R})_{\text{V}}^{\text{port}(v, n, w, w')\wedge\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{w'=[]\wedge\sigma}$$

For this to become (nontrivially) applicable, it is necessary to have "garbage collection" rules. A simplification rule is given for a constraint theory **TC** with port constraints and rational tree equality constraints of the form $X = Y$ or $X = f(Y_1, \ldots Y_n)$, where variables may be ports. (For a discussion about such combinations of theories, see [Franzen 1994].)

A *garbage collection rule* for this theory is

$$\textbf{and}(\text{R})_{\text{V}}^{\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R})_{\text{V}}^{\theta}$$

where the constraint atoms in θ is a strict subset of those in σ such that

$$\textbf{TC} \vdash \text{env}(\chi) \supset (\exists \text{W}\sigma \equiv \exists \text{W}\theta)$$

where W contains all variables in U not occurring in R, and where the set V contains all variables in U that occur in θ or R. This rule is clearly terminating, since there is a finite number of constraint atoms in a local constraint store.

For example, if χ is λ, R is p(X), σ is X = f(Y) ∧ Z = g(W,X), and U is {W, X, Y, Z}, then θ is X = f(Y) and V is {X,Y}, since

TC ∃W∃Y∃Z (X = f(Y), Z = g(W, X)) ≡ ∃W∃Y∃Z (X = f(Y))

This should correspond to our intuition for garbage collection.

7.4.4 Implementation of Ports

The implementation of ports, and of objects based on ports, gives us other advantages, which are first discussed in this section and then returned to in Section 7.5.2.

The implementation of ports can rely on the fact that a port is only read by the open_port/2 operation, and that the writers only use the send/2 and is_port/1 operations on ports, which are both independent of previous messages.

Therefore, there is no need to store the messages in the port itself. It is only necessary that the implementation can recognise a port, and add a message sent on a port to its associated stream. This can be achieved simply by letting the representation of a port point to the stream being constructed. In accordance with the port reduction rules, adding a message to a port then involves getting the stream, unifying the stream with a list pair of the message and a new "end-of-stream", and updating the pointer to refer to the new end-of-stream. Closing the port means unifying its stream with the empty list. Note that the destructive update is possible only because the port is "write only".

In this respect, ports are similar to mutual references. But, for ports there is conceptually no such notion as *advancing* the pointer to the end-of-stream. We are constructing a list of elements in the bag, and if the list is already given, it is unified with what we construct.

Other implementations of message sending are conceivable, e.g. for distributed memory multiprocessor architectures.

That a port has become garbage is detected by garbage collection, as suggested by the definition. If a copying garbage collector is used, it is only necessary to make an extra pass over the ports in the old area after garbage collection, checking which have become garbage (i.e., were not copied). Their corresponding streams are then closed.

From the object-oriented point of view, this is not optimal, as an object cannot be deallocated in the first garbage collection after the port becomes garbage, which means that it survives the first generation in a two-generation generational garbage collector. Note that for some types of objects, this is still acceptable, as their termination might involve performing some tasks, e.g. closing files. For other objects, it is not. In the next section we discuss compilation tech-

niques based on ports that allow us to differentiate between these two classes of objects, and treat them appropriately.

Reference counting is more incremental, and is therefore seemingly nicer for our purposes, but the technique is inefficient, it does not rhyme well with parallelism, and it does not reclaim cyclic structures. MRB [Chikayama and Kimura 1987] and compile-time GC (e.g., [Foster and Winsborough 1991]) are also of limited value, as we often want ports to be multiply referenced.

7.5 CONCURRENT OBJECTS

Returning to our main objective, object-oriented programming, we develop some programming techniques for ports, and discuss implementation techniques for objects based on ports.

Given ports, it is natural to retain the by now familiar way of expressing an object as a consumer of a message stream, and use a port connected to this stream as the object identifier.

```
create_object(P, Initial) :=
        open_port(P, S),
        object_handler(S, Initial).
```

In the next two sections we address the issues of synchronisation idioms, and of compilation of objects based on ports, as above.

7.5.1 Synchronisation Idioms

We need some synchronisation idioms. How do we guarantee that messages arrive in a given order? We can use continuations, as in Actor languages.

The basic sequencing idiom is best expressed by a program. In the following, the call / 1 agent is regarded as implicitly defined by clauses

$$call(p(X_1, \ldots, X_n)) :- \rightarrow p(X_1, \ldots, X_n).$$

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

```
open_cc_port(P, S) :=
        open_port(P, S0),
        call_cont(S0, S).

call_cont([], S) :-
    → S = [].
call_cont([(M & C) | S0], S) :-
    → S = [M | S1],
        call(C),
        call_cont(S0, S1).
call_cont([M | S0], S) :-
    → S = [M | S1],
        call_cont(S0, S1).
```

The (*Message* & *Continuation*) operator guarantees that messages sent because of something that happens in the continuation will come after the message. For example, it can be used as follows.

```
open_cc_port(P, S), send((a & Flag = ok), P), p(Flag, P).
```

The agent p may then choose to wait for the token before attempting to send new messages on the port P.

The above synchronisation technique using continuations can be implemented entirely on the sender side, with very little overhead. A goal send((*m* & *C*), *P*) is compiled as (send(*m*, *P*), *C*), with the extra condition that *C* should only begin execution after the message has been added to the stream associated with the port. It should be obvious that this is trivial, even in a parallel implementation.

Another useful idiom is the three-argument send, defined as follows.

```
send(M, P0, P) :=
    send((M & P = P0), P0).
```

which is useful if several messages are to be sent in sequence. If this is very common, the send_list operation can be useful.

```
send_list([], P0, P) :-
  → P = P0.
send_list([M | R], P0, P) :-
  → send(M, P0, P1),
    send_list(R, P1, P).
```

Both, of course, assume the use of the above continuation calling definition.

7.5.2 Objects based on Ports

If the object message-handler consumes one message at a time, feeding the rest of the message stream to a recursive call, e.g., as guaranteed by an object-oriented linguistic extension, then it is possible to compile the message handler using message oriented scheduling [Ueda and Morita 1992]. Instead of letting messages take the indirect route through the stream, this path can be shortcut by letting the message handling process pose as a special kind of port, which can consume its messages directly. There is then no need to save messages to preserve stream semantics. It is also easy to avoid creating the “top-level structure” of the message, with suitable parameter passing conventions. The optimisation is completely local to the compilation of the object.

Looking also at the implementation of ports from an object-oriented point of view, an object compiled this way poses as a port with a customised send-method. This view can be taken further by also providing customised garbage collection methods that are invoked when a port is found to have become garbage. If the object needs cleaning up, it will survive the garbage collection to perform this duty, otherwise the GC method can discard the object immediately.

Objects in common use, such as arrays, can be implemented as built-in types of ports, with a corresponding built-in treatment of messages. This may allow an efficient implementation of mutable data-structures. Ports can also serve as an interface to objects written in foreign languages.

Ports and built-in objects based on ports are available in AGENTS [Janson et al 1994]. An interesting example is that it provides an AKL engine is as a built-in object. A user program can start a computation, inspect its results, ask for more solutions, and, in particular, reflect on the failure or suspension of this computation. This facility is especially useful in programs with a reactive part and a (don't know nondeterministic) transformational part, where the interaction with the environment in the reactive part should not be affected by nondeterminism or failure, as exemplified by the AGENTS top-level and some programs with graphical interaction. In the future, this facility will also be used for debugging of programs and for metalevel control of problem solving.

In AGENTS, the open_port/2 operation is a specialised open_cc_port/2, which does not accept general continuations, but supports the send/3 operation.

7.6 PRAM

Shapiro [1986b] discusses the adequacy of concurrent logic programming for two families of computer architectures by simulating a RAM (Random Access Machine) and a network of RAMs in FCP (Flat Concurrent Prolog). However, a simulator for shared memory multiprocessor architectures, PRAMs (Parallel RAMs), is not given.

We conjecture that PRAMs cannot be simulated in concurrent logic programming languages without ports or a similar construct. This limitation could, among other things, mean that array-bound parallel algorithms, such as many numerical algorithms, cannot always be realised with their expected efficiency in these languages.

In the following we will show the essence of a simulator for an Exclusive-read Exclusive-write PRAM in AKL using ports.

7.6.1 PRAM with Ports

A memory cell is easily modelled as an object.

```
cell(P) :=
        open_cc_port(P, S),
        cellproc(S, 0).
```