Ports for Objects in Concurrent Logic Programs*

Sverker Janson Johan Montelius Seif Haridi
Swedish Institute of Computer Science†

July 13, 1993

Abstract

We introduce *ports*, an alternative to streams, as communication support for object-oriented programming in concurrent constraint logic programming languages. From a pragmatic point of view ports provide efficient many-to-one communication, object identity, means for garbage collection of objects, and opportunities for optimised compilation techniques for concurrent objects. From a semantic point of view, ports preserve the monotonicity of the constraint store which is a crucial property of all concurrent constraint languages.

We also show that the Exclusive-read, Exclusive-write PRAM model of parallel computation can be realised quite faithfully using ports in terms of space and time complexity, thus allowing arbitrary parallel programs to be written efficiently.

Ports are available in AKL, the Andorra Kernel language, a concurrent logic programming language that provides general combinations of don't know and don't care nondeterministic computations.

1 Introduction

In this paper we introduce an alternative to streams as the communication medium for object-oriented programming in concurrent logic programming languages. This alternative will be seen to provide us with efficient many-to-one communication, object identity, means for garbage collection of objects, and optimised compilation techniques for objects. It will also provide us with means for mixing freely objects and other data structures provided in concurrent logic programming languages.

We regard *objects* for concurrent logic programming languages as processes, as first proposed by Shapiro and Takeuchi [16], and later extended and refined in systems such as Vulcan, A'UM, and Polka [11, 20, 3] (Figure 1). Some of these systems are embedded languages that make restricted use of the underlying language. We are interested in full-strength combinations of the underlying language with an expressive concurrent object-oriented extension, with all the problems and opportunities this entails.

In this paper, we present an approach to communication between objects (and processes in general), which is both efficient and useful in such a general setting. In the next section, we will remind the reader of a number of problems with previous

*Also in Research Directions in Concurrent Object-Oriented Programming, MIT Press, 1993.
†Box 1263, S-164 28 KISTA, Sweden, E-mail: {sverker,jm,seif}@sics.se

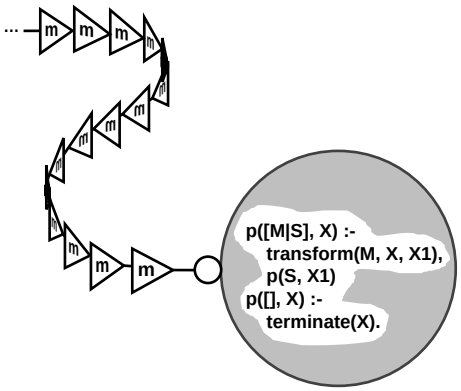

Figure 1: Objects as stream-consuming recursive processes

approaches, and then, in the following sections, we introduce *ports* as a solution to these problems, one which also adds entirely new possibilities, such as a simple approach to optimised compilation of objects.

We will also show that the Exclusive-read Exclusive-write PRAM model of parallel computation can be realised quite faithfully, in terms of both space and time complexity, using ports. This indirectly demonstrates that arbitrary parallel algorithms can be expressed quite efficiently.

2 Background

In this section we present some of the background of our work. First we examine some requirements on object-oriented systems. Then we discuss the notion of a communication medium, and review a number of proposals that do not meet our requirements. We also discuss the limitations of some existing systems that are based on these proposals.

2.1 Requirements of Object-Oriented Systems

Our starting point is a number of requirements on object-oriented languages currently in use, such as Smalltalk and C++, and we will let these guide our work. In so doing, we here only consider requirements on the object-based functionality, including requirements on the interaction between the host language and the concurrent object-oriented extension. Other aspects of object-oriented languages, such as inheritance, can be realised in this setting in many different ways (e.g., [7]).

Since our goal is the integration of concurrent object and logic programming, we conform to the tradition of languages such as C++, where the object-oriented aspects are added to the underlying language and, in particular, allow objects and other data structures to be mingled freely. Thus, in a logic programming language, it should be possible to have objects *embedded* in a term data structure (Figure 2). Since terms can be shared freely in concurrent logic programming languages, the extra ability will allow concurrent objects to share, for instance, an array of concurrent objects. Higher-level object-oriented languages provide automatic *garbage collection* of objects that are no longer referenced (Figure 3). Programmers do not

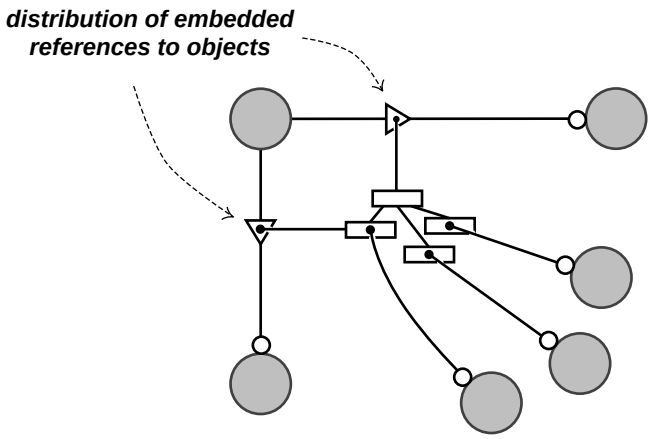

Figure 2: Objects embedded in terms

have to think about when objects are no longer in use, nor do they have to deallocate them explicitly. The high-level nature of logic programming languages makes it desirable to provide garbage collection of objects, just as of other data structures. Note that, in the goal-directed view of logic programming, an object is a goal, which

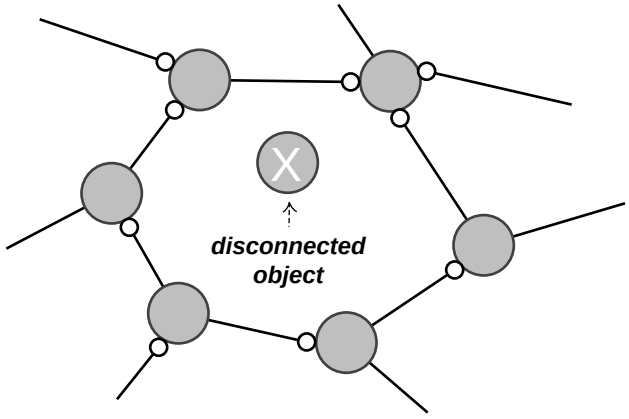

Figure 3: Garbage-collectable object

has to be proved; it cannot just be thrown away. However likely the assumption that a goal is provable without binding variables, there is no such guarantee. In a concurrent object-oriented setting, another dimension is added, in that an object may still be active, and affect its environment, although it is no longer referred to by other objects. Even if there are no incoming messages, an object may wish to perform some cleaning up before being discarded. Thus, garbage collection of objects in concurrent (logic) programs should involve notifying an object that it will no longer receive messages. It is up to the object to decide to terminate.

In addition, an implementation of objects should provide

- light-weight message sending and method invocation, the cost of which should preferably be similar to that of a procedure call,
- compact representation of objects, the size of which should be dominated by the representation of instance variables,

- memory conservative behaviour, which means that a state transition should only involve modifying relevant instance variables—objects are reused.

In this paper, we will address all but the last of the above requirements. The last requirement is generally solved in concurrent logic programming languages by providing a mechanism for detecting single references, and reusing the old instance variables.

2.2 Communication Media

A *communication medium* is a data type that carries messages between processes acting as objects (Figure 4). The medium is used as a handle to an object, and is

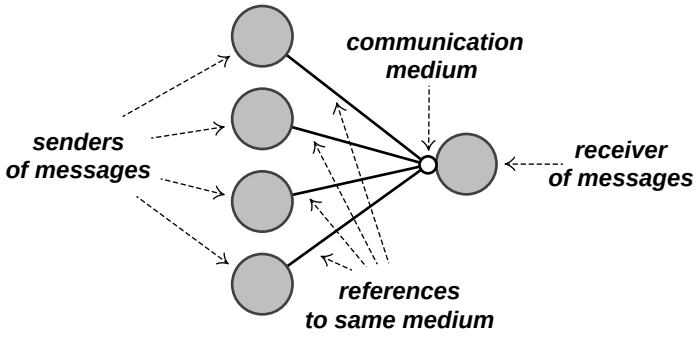

Figure 4: The communication medium

regarded as the *object identifier* from a programmer's point of view.

In the concurrent constraint view that we take, the communication medium is managed (described and inspected) using constraints [13]. All constraints are added to a shared *constraint store*. To *send* a message means to impose a constraint on the medium which allows a process to detect the presence of a message, and *receive* the message by inspecting its properties. A sender of messages is a *writer* of the medium. A receiver of messages is a *reader* of the medium. A message that is received once and for all is said to be *consumed*. A medium can be *closed* by imposing constraints that disallow additional messages. A receiver can detect that a medium is closed.

An important property of the constraint store in concurrent logic programming languages is that it is monotonic. Addition of constraints will produce a new constraint store that entails all the information in the previous one. This property is important because it implies that once a process is activated by the receipt of a message, this activation condition will continue to hold until the message is consumed regardless of the actions of other processes.

2.2.1 Requirements on Communication Media

The discussion above leads us to the following requirements on our communication medium.

- Any constraint-based solution should preserve the monotonicity of the constraint store.

- The number of operations required to send a message (make it visible to an end receiver) should be constant (for all practical purposes), independent of the number of senders. All senders should be given equal opportunity, according to a first come first served principle. We call this the *constant delay property.*

- When a part of the medium that holds a message has been consumed, it should be possible to deallocate or reuse the storage it occupied, by garbage collection or otherwise.

- To provide completely automatic garbage collection of objects, it should be possible to apply the closing operation automatically (when the medium is no longer in use).

- To enable sending multiple messages to embedded objects, it should be possible to send multiple messages to the same medium.

The last requirement seems odd in conventional object-oriented systems. It is however a problem in all single-assignment languages including the current concurrent logic programming languages.

2.2.2 Streams

The list is the by far most popular communication medium in concurrent logic programming. In this context lists are usually called *streams.*

A message m is sent on the stream S by constraining S to a *list pair* S=[m|S1]. The next message is sent on the stream S1. The stream is closed by constraining it to the *empty list* S=[]. The receiver, which should be waiting for S to become constrained to either a list pair or the empty list, will then either successfully match S against a list pair [M|R], whereupon M will be bound to m and R to S1, in which case the next message can be received on S1, or match S against [], in which case no further messages can be received. By the *end-of-stream* we mean a tail of the stream that is not yet known to be a list pair or an empty list.

To achieve the effect of several senders on the same stream, there are two basic techniques: (1) Several streams, one for each sender, are interleaved into one by a process called *merging.* (2) The language provides some form of atomic "test-and-set" unification, which allows *multiple writers* to compete for the end-of-stream.

Merging is typically achieved either by a tree of binary mergers, or by a multiway merger. The binary merge tree is built by splitting a stream as necessary (Figure 5). Clearly, this technique does not have the constant delay property as the (best case) cost is $O(\log m)$ in the number of senders m. A *multiway merger* is a single merger process that allows input streams to be added and deleted dynamically (Figure 6). A *constant delay multiway merger* cannot be expressed in most concurrent logic programming languages (AKL is an exception [8]), but it is conceptually clean, and it is quite possible to provide one as a language primitive. We will assume that all multiway mergers have constant delay.

A number of disadvantages of merging follow.

- A merger process has to be created whenever there is the slightest possibility that several senders will send on the same stream. For many purposes this is not a problem. Once a multiway merger is created, adding and deleting input

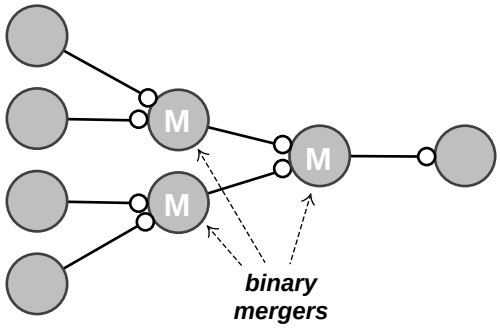

Figure 5: Binary merge network

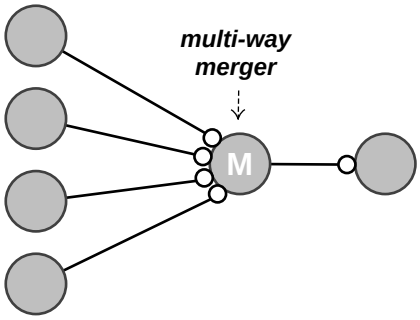

Figure 6: Multi-way merger

streams is fairly efficient. Yet, needing one feels like an overhead in an object-oriented context, where references to objects should be freely distributable.

- Explicit closing of all streams to all objects is necessary, since otherwise the program will either eventually deadlock, or at least some objects will continue to be suspended, forever occupying storage.

- The merger process itself occupies storage, it also wastes storage when creating a new merged stream, and if it uses the standard mechanisms for suspension, it is likely to be (comparatively) inefficient.

- Messages have to be sent on the current end-of-stream variable, which is changed for every message sent. To make it possible to use a stream that is stored in some other data structure, the new end-of-stream variable has to be stored in the data structure after one round of sending is completed. Usually this means copying parts of the data structure (but if some form of single reference optimisation is employed by the language implementation [2, 10, 4], this would not necessarily be the case). Even worse, to enable sharing of this data structure with another process/object, where the possibility of the other object sending messages on the embedded streams cannot be excluded, two copies of the data structure have to be created (allocating new memory for at least one). All the streams have to be split in two (by merging), one for each copy.

- A serious problem is the *transparent message-distribution problem*. A message is usually a term m(C1,..,Cn) where the Ci's are message components. Suppose

we want to implement a transparent message-distributor object, which when it receives a message, of any kind, will distribute it to a list of other objects. Without prior knowledge of the components of messages, the distributor object cannot introduce the merging required for stream components.

An advantage with merging is that it allows list pairs to be reused in the merger and deallocated by the receiver as soon as a message has been consumed. In some cases, in a system with fairly static object-structure, explicit closing of all streams as a means for controlled termination of objects can be considered an advantage. Another general advantage of all stream communicating systems is the implicit sequencing of messages from a source object to a destination. This simplifies synchronisation in many applications.

Multiple writers can only be expressed in some languages with atomic test-and-unify. The drawbacks of multiple writers are summarised as follows:

- The cost for multiple writers is typically $O(m)$ per message, when there are m senders, and is therefore even further from the constant delay property than merging.
- The delay is proportional to the number of messages that have been sent.
- An inactive sender may hold a reference to parts of the stream that have already been consumed by the intended receiver, making deallocation impossible.
- It is difficult to close a stream. Some distributed termination detection technique, such as short-circuits or the like, has to be used. In practice, this outweighs the advantages of multiple writers.

An advantage with multiple writers is that no merger has to be created. Moreover, several messages can be sent on the same stream, and not only by having explicit access to the end-of-stream variable.

Neither merging nor multiple writers provide a general solution to the problem of automatic garbage collection of objects. There are special cases, as exemplified by A'UM (see Section 2.3), where (acyclic structures of) objects are terminated automatically.

It can be suspected that many of the problems described above are inherent in the use of streams as communication channels. Thus, people have looked for alternative data types.

2.2.3 Mutual References

Shapiro and Safra [15] 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|S1], and New_Ref is returned 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 the 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 bound 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 have 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 cooperation.

2.2.4 Channels

Tribble et al [17] introduced *channels* to allow multiple readers as well as multiple writers.

A channel is a partially ordered set of messages. The write(M, C1, C2) operation imposes a constraint that: (1) the message M is a member of the channel C1, and (2) M precedes all messages in channel C2. The read(M, C1, C2) operation selects a minimal (first) element M of C1, returning the remainder in a channel C2. The empty(C) operation detects that a channel C is empty. The close(C) operation imposes the constraint that a channel C is empty.

In the intended semantics, messages have to be labeled to preserve message multiplicity. Also, since the constraints do not specify which messages are not in the channels, only minimal channels 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 someone might read it. 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.

2.2.5 Bags

Kahn and Saraswat [10] introduced *bags* for the languages Lucy and Janus.

Bags are multi-sets of messages. They are like streams in that subsequent messages are sent on subsequent bags, but there is no need for user-defined merging, as this is taken care of by the Tell constraint bag-union $B = B1 \cup \cdots \cup Bn$. A message is sent using the Tell constraint $B = \{m\}$. A combination of these two operations, $B = \{m\} \cup B1$, corresponds to sending a message on a stream, but without the order of messages given by the stream. A message is received by the Ask constraint $B = \{m\} \cup B1$.

Note that bags can be implemented as streams, with a multiway merger as bag-union. The single-reference property of Janus then makes it possible to reuse list pairs in the multiway merger, and to deallocate (or reuse) list pairs when a bag is consumed.

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.

2.3 Object-Oriented Concurrent Logic Programming

Most proposals for object-oriented embedded languages in, and object-oriented extensions of, concurrent logic programming languages, are limited by their communication medium, streams.

Vulcan is a pure object-oriented embedded language [11]. It apparently does not allow embedding objects in terms, has no special provisions for termination, and suffers from the transparent message distributor problem.

A'UM is also a pure object-oriented embedded language [20]. It provides automatic termination, by reference counting of objects implemented in the host language, and solves the transparent message distributor problem by restricting all components of a message to be streams to other objects. This, of course, restricts the language.

Polka is a language extension of Parlog [3]. It does apparently allow arbitrary mixtures of objects and terms, in a style not entirely unlike what we are aiming for, but it does not solve any of the problems of stream merging.

3 Ports

We propose *ports* as a solution to the problems with previously proposed communication media.

3.1 Ports Informally

A *port* is a connection between a bag of messages and a corresponding stream of these messages (Figure 7). 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(P, M) 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 automatically closed. The port(P) operation recognises ports. A first simple example follows.

```
?- open_port(P, S), send(P, a), send(P, b).

P = <some printed representation of a port>,
S = [a,b] ?
```

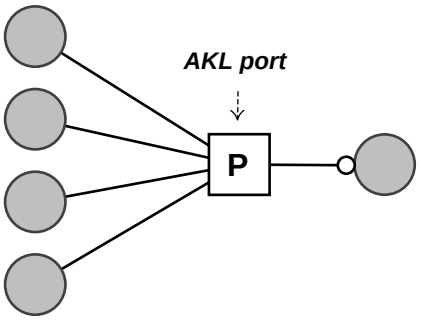

Figure 7: AKL port

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, thus notifying the consumer of messages.
- The transparent forwarding problem is solved, since messages can be distributed without inspection.
- Messages can be sequenced, as described in Section 4.

3.2 Ports as Constraints

We can provide a sound and intuitive interpretation of ports as constraints as follows. Ports are bags, in other words multi-sets. 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.

Our method for finding a solution to these constraints is don't care nondeterministic. Like the commit operation in concurrent logic programming languages, we are happy with any solution. 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 [6]. The don't care nondeterministic and resource sensitive behaviour of ports can easily be captured by LL. The automatic closing requires much more machinery. If such an exercise would aid our understanding remains to be seen.

3.3 Solving Port Constraints

We now define the operational semantics of ports in terms of rewrites on (parts of) a constraint store.

Observe that the rewrite rules will strictly accumulate information in the constraint store; the right hand side always implies the left hand side. The first rule adds an annotation only. The second and third rules add constraints on the stream. Thus, we preserve the desirable monotonicity property.

The first rule annotates the open_port constraint, for control purposes. We need an index to keep track of where in the stream S we are putting new messages.

$$\mathsf{open\_port}(P, S) \Rightarrow \mathsf{open\_port}(P, S)_S$$

The second 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.

$$\begin{array}{l}\mathsf{open\_port}(P, S)_{S'}, \mathsf{send}(P, m) \Rightarrow \\ \quad S' = [m \mid S''], \mathsf{open\_port}(P, S)_{S''}\end{array}$$

The third and final rule closes the associated stream when the port P satisfies the *garbage condition* with respect to the computation state (see below).

$$\mathsf{open\_port}(P, S)_{S'} \Rightarrow S' = [], \mathsf{open\_port}(P, S)_{S'}$$

The garbage condition for a port holds in a computation state if there exists a *garbage collected* state in which the port only occurs in a single open_port constraint and in consumed send constraints.

We will attempt to give a definition of a kind of a garbage collected state, which, although quite an intuitive notion, is fairly operational in nature, and therefore difficult to fit into the notion of a constraint store.

Let us assume, to keep our definition simple, that our host language is a determinate concurrent constraint language, with ports and with syntactic equality constraints of the form $\mathsf{X} = \mathsf{Y}$ or $\mathsf{X} = \mathsf{f}(\mathsf{Y}_1, \ldots, \mathsf{Y}_n)$, where variables may be ports. Note that this combination of ports and equality is not accounted for by the constraint solving rules, and that the system presented here is incomplete in this regard.

A *computation state* consists of a conjunction of constraints and agents. The idea is that we may throw away constraints that no longer restrict variables reachable from agents, because the resulting constraint store is logically equivalent.

A computation state A is *garbage collected* wrt another computation state B if (1) they have the same agents, (2) the constraints in A are a subset of the constraints in B, and (3) the conjunction σ of constraints in A is equivalent to the conjunction θ of constraints in B, modulo the existential quantification of variables $v_1, \ldots, v_n$ occurring in constraints but not in agents, i.e.

$$\exists v_1 \cdots \exists v_n . \sigma \equiv \exists v_1 \cdots \exists v_n . \theta$$

For example, if "$\mathsf{p(X)}$, $\mathsf{X = f(Y)}$, $\mathsf{Z = g(W, X)}$" is a state, then "$\mathsf{p(X)}$, $\mathsf{X = f(Y)}$" is garbage collected wrt that state, since

$$\exists \mathsf{Y} \exists \mathsf{Z} \exists \mathsf{W}(\mathsf{X = f(Y)} \;\&\; \mathsf{Z = g(W, X)}) \equiv \exists \mathsf{Y} \exists \mathsf{Z} \exists \mathsf{W}(\mathsf{X = f(Y)})$$

This also corresponds to our intuition. Note that for some constraint systems a more involved notion of simplification than the deletion of individual constraints is necessary.

3.4 Implementation

There are advantages in the implementation of ports, and of objects based on ports, which we first discuss in this section and then return to in Section 4.

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 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 recognize a 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 rewrite 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 multi-processor 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 Section 4 we discuss compilation techniques 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 and compile-time GC are also of limited value [2, 4], as we often want ports to be multiply referenced.

4 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.

4.1 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 (which is written in AKL, using the conditional (if-then-else) guard operator "→").

```
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 which happens in the continuation will come after the message. For example, it can be used as follows.

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

The procedure p/2 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(P, (m & C)) is compiled as (send(P, m), 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(P0, m, P) :-
        send(P0, (m & P = P0)).
```

which is useful if several messages are to be sent in sequence. If this is very common, a send-list operation can be useful.

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

4.2 Objects based on Ports

If messages are consumed one at a time by the object message-handler, and the input stream is not manipulated in other ways, as can be guaranteed by an object-oriented linguistic extension, then it is possible to compile the message handler using message oriented scheduling [19]. 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.

Object-types 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 interfaces to objects written in foreign languages.

Ports and built-in objects based on ports are available in the AKL Programming System (AKL/PS) [9]. An interesting example is that an AKL engine is provided as a built-in object. A user program can start a computation, inspect its results, ask for more solutions (AKL is don't know nondeterministic), 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 AKL/PS top-level and some programs with graphical interaction. In the future, this facility will also be used for debugging of AKL programs and for meta-level control of problem solving.

5 Modelling PRAM with Ports

Shapiro [14] 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 multi-processor 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.

5.1 PRAM with Ports

A memory cell is easily modelled as an object.

```
cell(P) :-
        open_cc_port(P, S),
        cellproc(S, 0)

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 8).

```
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.

5.2 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 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 [2].

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

A program may access (read) the vector using the

```
vector_element(Vector, Position, ^Element, ^NewVector)
```

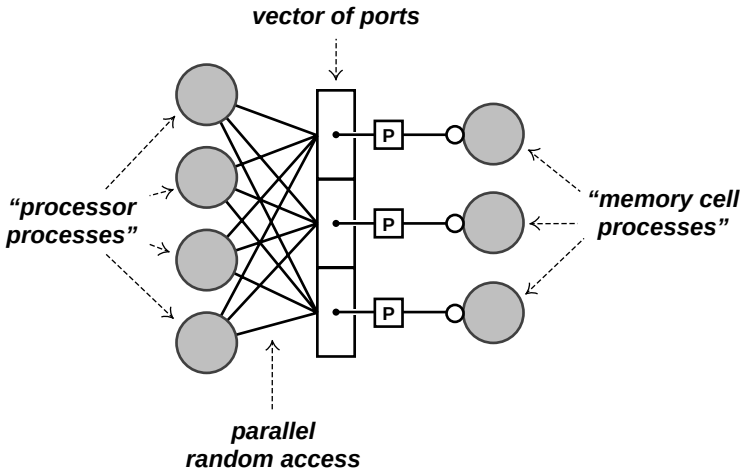

Figure 8: PRAM with AKL ports

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.

5.3 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 of streams to the memory cells. All streams referring to a single memory cell are merged. Sequencing is achieved as above. Thus we need $O(nm)$ units of storage to represent a PRAM with n memory cells and m processors.

The setup of memories is correspondingly more awkward.

```
memories(M1, ..., Mm) :-
        memvector(M1, C11, ..., C1n),
        memvector(M2, C21, ..., C2n),
        ...,
        memvector(Mm, Cm1, ..., Cmn),
        cell(C1), ..., cell(Cn),
        merge(C11, ..., C1m, C1),
        merge(C21, ..., C2m, C2),
        ...,
        merge(Cn1, ..., Cnm, Cn).

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

memvector(M, C1, ..., Cn) :-
        new_vector(M1, n),
        set_vector_element(M1, 1, _, C1, M2),
```

```
set_vector_element(M2, 2, _, C2, M3),
...,
set_vector_element(Mn, n, _, Cn, M).
```

Memory is accessed and modified as in a combination of the two previous models (Figure 9). 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.

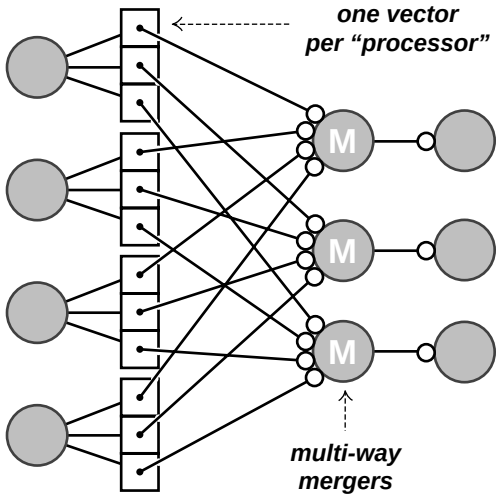

Figure 9: PRAM without AKL ports

6 Examples

In the following, examples are shown, due to Barth, Nikhil, and Arvind [1], which 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.

6.1 Histogramming a Tree of Samples

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

In our solution, the number of occurrences are collected in a table of counters, with indices in the given range. Assume that the memory agent defined in the previous section returns a memory of this size. The program traverses the tree,

incrementing the counter corresponding to a number. To guarantee that all nodes have been counted, the program performs the computation in a guard, and returns the table upon its completion.

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

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

6.2 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 $1, \ldots, n$. Any number of computations on the same graph can be done concurrently.

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 given 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 Discussion

We have argued that the communication medium *ports* solves a number of problems with the interpretation of processes as objects. It provides efficient many to one communication, object identity, means for garbage collection of objects, and opportunities for optimised compilation techniques for objects.

Ports are available in the AKL Programming System being developed at SICS. The prototype is available without charge for research purposes. (Contact any of the authors.)

The Andorra Kernel Language (AKL) is a general combination of search-oriented nondeterministic languages, such as Prolog, and the process-oriented committed-choice and concurrent constraint languages [18, 12, 13]. For an introduction to the language from this perspective, see [8]. For a formal treatment, see [5].

In AKL interesting combinations of object-oriented and constraint solving programs are possible, partly thanks to ports, but that is the topic of another paper.

Acknowledgements

This work was in part sponsored by ESPRIT Project 2471 ("PEPMA"). SICS is a non-profit research foundation, sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK), Asea Brown Boveri AB, Ericsson Group, IBM Svenska AB, NobelTech Systems AB, the Swedish Defence Material Administration (FMV), and Swedish Telecom.

References

[1] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: extending a parallel, non-strict, functional language with state. In *Functional Programming and Computer Architecture '91*, 1991.

[2] T. Chikayama and Y. Kimura. Multiple reference management in flat GHC. In *Proceedings of the Fourth International Conference on Logic Programming*, volume 2, pages 276–293. MIT Press, 1987.

[3] Andrew Davison. *POLKA: A Parlog Object-Oriented Language*. PhD thesis, Department of Computing, Imperial College, London, May 1989.

[4] Ian Foster and Will Winsborough. Copy avoidance through compile-time analysis and local reuse. In *Logic Programming: Proceedings of the 1991 International Symposium*, San Diego, California, October 1991. MIT Press.

[5] Torkel Franzén. Logical aspects of the Andorra Kernel Language. SICS Research Report R91:12, Swedish Institute of Computer Science, October 1991.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[7] Yaron Goldberg and Ehud Shapiro. Logic programs with inheritance. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*. Omsha Ltd, June 1992.

[8] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*, San Diego, California, October 1991. MIT Press.

[9] Sverker Janson and Johan Montelius. Design of a sequential prototype implementation of AKL. SICS research report, Swedish Institute of Computer Science, 1992.

[10] Kenneth M. Kahn and Vijay A. Saraswat. Actors as a special case of concurrent constraint programming. In Norman Meyrowitz, editor, *OOPSLA/ECOOP '90 Conference Proceedings*. ACM/SIGPPLAN, 1990.

[11] Kenneth M. Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. Vulcan: Logical concurrent objects. In P. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

[12] Michael J. Maher. Logic semantics for a class of committed choice programs. In *Proceedings of the Fourth International Conference on Logic Programming*. MIT Press, 1987.

[13] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1990.

[14] Ehud Shapiro. A test for the adequacy of a language for an architecture. In *Concurrent Prolog: Collected Papers*. MIT Press, 1987.

[15] Ehud Shapiro and Shmuel Safra. Multiway merge with constant delay in Concurrent Prolog. *Journal of New Generation Computing*, 4(3):211–216, 1986.

[16] Ehud Shapiro and Akikazu Takeuchi. Object-oriented programming in Concurrent Prolog. *Journal of New Generation Computing*, 1(1):25–49, 1983.

[17] Eric Dean Tribble, Mark S. Miller, Kenneth Kahn, Daniel G. Bobrow, Curtis Abbott, and Ehud Shapiro. Channels: A generalisation of streams. In *Proceedings of the Fourth International Conference on Logic Programming*. MIT Press, 1987.

[18] Kazunori Ueda. Guarded horn clauses. Technical Report TR-103, ICOT, June 1985.

[19] Kazunori Ueda and Masao Morita. A new implementation technique for flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*. MIT Press, 1990.

[20] K. Yoshida and T. Chikayama. A'UM—a stream-based concurrent object-oriented language. In *Proceedings of FGCS'88*, ICOT, Tokyo, 1988.