An implementation model of rendezvous communication

Luca Cardelli

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

This paper describes the low-level primitives necessary to implement a particular flavor or inter-process communication. It is motivated by the design of a communication subsystem for a higher-order functional language [Cardelli 84]. Here we try to abstract somewhat from the special characteristics of that language, but the model does not accommodate a wide range of communication schemes.

This communication model is intended to be used on (uniprocessor) personal computers. In this model, processes running on the same processor can share the same address space. If the underlying language is safe, a process can affect other processes only by communication, or by affecting data structures which have been explicitly transmitted. This ensures privacy and data protection even in a shared address space.

Processes running in the same address space can exchange arbitrarily complex objects very cheaply, just by passing pointers. Processes running on different processors communicate through restricted "flat" channels, e.g. character channels. In this case, complex objects have to be encoded to fit into flat channels, and decoded on the other side; the encoding activity may or may not be automatic. In any case there is a semantic difference between exchange of objects in the same address space, where objects are shared, or in different address spaces, where objects are copied.

The basic communication mechanism is rendezvous [Milner 80]: both the sender and the receiver may have to wait until the other side is ready to exchange a message. Both the sender and the receiver may *offer* communications simultaneously on different channels: when a pair of complementary offers is selected for a rendezvous, all the other simultaneous offers, on both sides, are *retracted*.

The scheduling is non-preemptive: a running process will run until it explicitly gives up control (e.g. by attempting a communication); at that point other processes will get a chance to run. We assume a cooperative universe, where no process will try to take unfair advantage of other processes, unless it has some reason for doing so.

Channels and processes can be dynamically created. Channels can be manipulated as objects and even passed through other channels [Abramsky 83, Inmos 84, Milner 82]. Processes are not denotable values; they can only be accessed through channels.

In the following sections, we use the term "pool" rather than "queue" because the latter term implies a particular scheduling policy to which we do not wish to commit ourselves. Note that the relation "being in the same pool" is non-transitive: an object can appear in different pools without implying the equivalence of all such pools.

Channel pools

A communication channel has an *output* pool and an *input* pool. The output pool records all the processes that have offered an output communication on this channel, and the respective output values. The input pool records all the processes that have requested an input communication on this channel. There can be any number of processes in either pool. In the figures, objects in the same pool are connected by a double line.

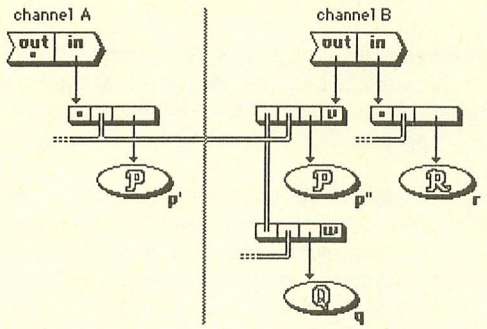


Figure 1: Channel pools

In the above figure, the process P is simultaneously waiting for input on channel A, and for output on channel B (where it is ready to communicate the value v) and possibly on more channels. The simultaneous communication offers of processes are chained across channels into *mutual exclusion* pools; one such pool connects the two instances of P above.

If P is selected for communication on channel A (which could not happen in this situation because the out pool is empty), then P will resume in state p'. If instead P is selected for communication on channel B, it will resume in state p''. In both cases, P is removed from the pool of channels A, B and all other channels in the same mutual exclusion pool.

Two communications are possible on channel B: P with R and Q with R. If Q and R are made to communicate (by exchanging the value w), both Q and R are removed from the channel B pools; Q is resumed in state q and R is resumed in state r. Again, all the simultaneous offers of Q and R are removed from the appropriate channels. Restriction: although a process can be waiting for input and output on the same channel, it cannot be made to communicate with itself because the resumption state would be ambiguous.

Scheduler Pools

A channel is *active* if it has non-empty in or out pools. At any moment, all the active channels are grouped into an *active channel* pool, which is used to select the next communication according to some scheduling policy.

The active channel pool is the primary scheduling structure: processes ready to communicate hang from it until they are activated. However, in some situations, e.g. just

after process creation, processes have not had a chance to be inserted into the active channel pool. Such processes are temporarily stored in an auxiliary pool of *ready* processes. Processes in the ready pool have a resumption state; for newly created processes this is just the initial state.

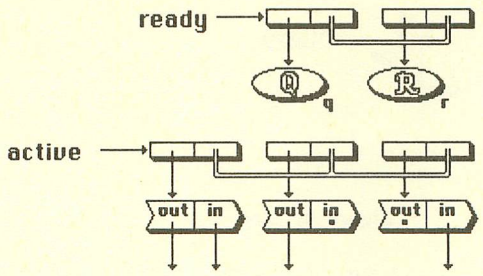


Figure 2: Scheduler pools

State Transitions

The global state of the system is determined by three quantities: the currently executing process (and its current state), the ready process pool and the active channel pool.

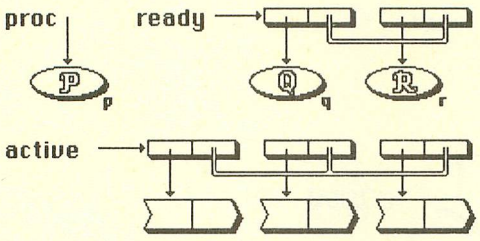


Figure 3: The state

The system evolves by state *transitions* [Plotkin 81]. There are three kind of transitions: (a) internal process transitions, which only affect the internal state of the currently executing process, (b) external process transitions, which affect the ready pool, the active pool or deactivate the current process, and (c) scheduler transitions, which take place when there is no current process. Internal process transitions are not described here (except for an operation which creates new channels), because they are largely independent of the concurrency aspects, but they can be taken to be similar to the ones in [Cardelli 83].

In the figures, a transition is represented as an "old" state, a big arrow with a transition name, and a "new" state. Those parts of the state which are not affected by a transition, are omitted.

Process transitions

There are five concurrency-related operations a process can perform. The first one is **stop**, which kills the currently executing process:

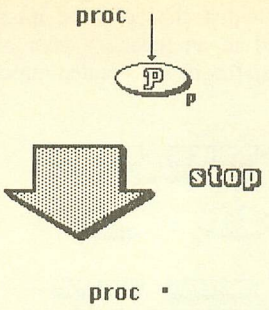


Figure 4: Stop

After a stop operation there is no current process. In this situation the scheduler takes over, as we shall see in the next section. The scheduling discipline ensures that P is not in any pool when it stops. We assume that the channels known by P will be garbage collected, if needed.

A process can create another process by a **start** operation. Given a program Q, a new process is created to execute Q and is placed in the ready pool in its initial state. The current process continues execution. The exact structure of Q is not discussed here; it could be a functional closure, as described in [Cardelli 83].

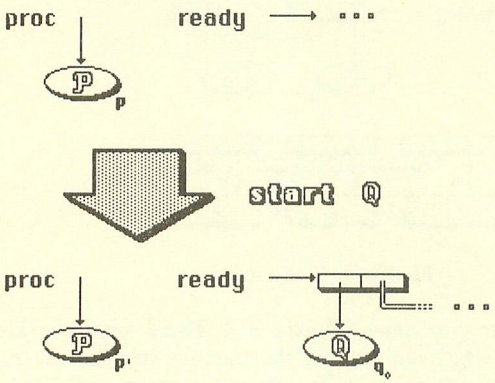


Figure 5: Start

A process can create a new channel by the **channel** operation. New channels are not active, and are placed on the execution stack of the current process.

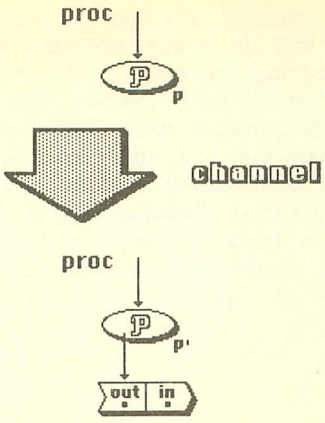


Figure 6: Channel

A process can simultaneously offer communication on a set of channels, with the intent of communicating on exactly one of them as soon as a communication is available. This is done by the **select** operation. There can be any positive number of input and/or output offers (a **select** with zero offers would be equivalent to **stop**). Each offer designates a channel (A, B and C in the example below) and a resumption state (a, b and c) for the process after a communication. Input offers are decorated with a question mark (?) and output offers with an exclamation mark (!) and a value to be communicated.

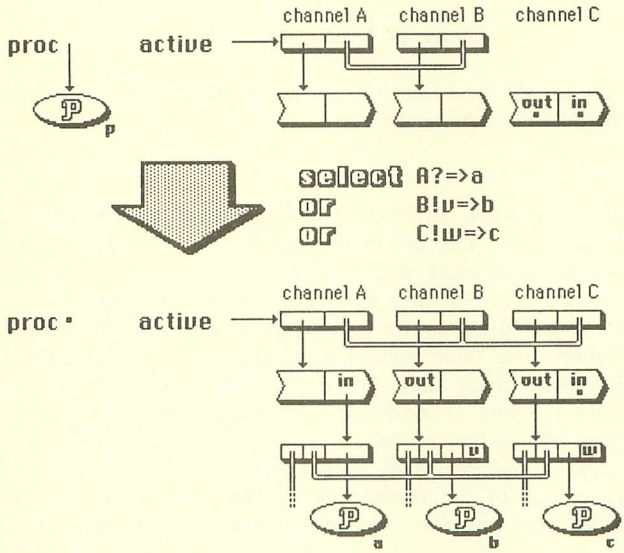


Figure 7: Select

The current process is inserted in the input or output (as appropriate) channel pools of the channels involved in the operation, with the proper resumption state and output value (if appropriate). All these offers are chained together into a mutual exclusion pool, to remember that they were done simultaneously. Some inactive channels (like C above) may become active and are then inserted into the active channel pool. After a **select** operation there is no current process. Hence, the scheduler takes over and other processes get a chance to run.

The following **deactivate** operation deactivates the current process and gives control to the scheduler. It can be useful in some special scheduling policies. For example a **deactivate** can be inserted in loops and recursive function calls to prevent looping processes from locking out everybody else. It can also be used by concurrently running processes which do not need to communicate, to periodically give up control.

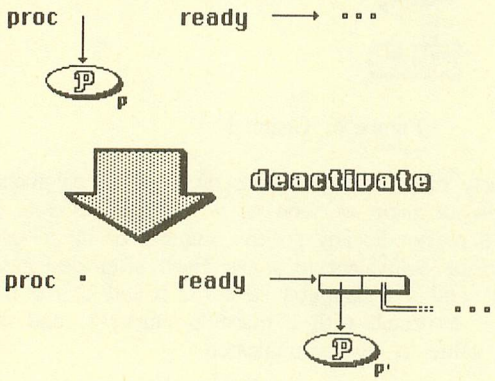


Figure 8: Deactivate

This operation is not strictly necessary; it could be simulated by a **select** on some channel which is always ready to communicate.

Scheduler transitions

When there is no current process, an entity called the *scheduler* gets control and decides who should run next. The scheduler has two major options: it can take a process from the ready pool (if one is available) and make it the current one, or it can select a channel for a communication (if one is possible) and give control to one of the processes involved in the communication. The choice between these two options is a matter of scheduling policy, and depends on the implementation. The same holds for other choices the scheduler has to make: which process to select from the ready pool, which channel to select from the active pool, which communication to fire on that channel, etc. All these possibilities are left open; the scheduling policy is not even required to be fair.

By an **activate** operation, the scheduler can select a ready process to become the current one. Processes are placed in the ready pool by **start** operations (for new processes) and by **rendezvous** operations (for processes involved in output communications, as explained later).

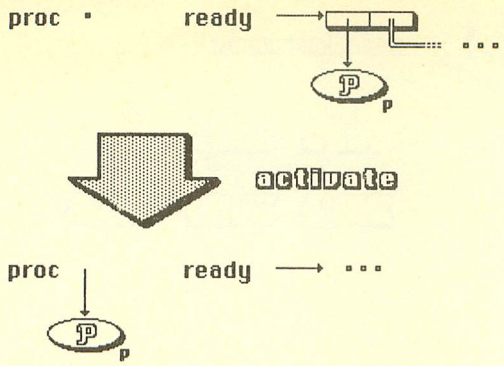
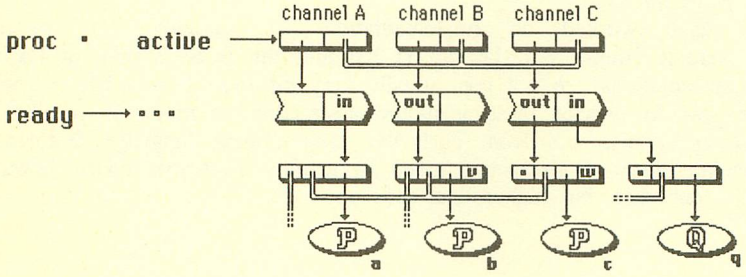


Figure 9: Activate

A **rendezvous** operation causes a communication to happen between two distinct processes, one waiting for input and the other one waiting for output on the same channel (P and Q on channel C in the example below). The process waiting for input (Q, with resumption q) becomes the current process, and the value being communicated (w) is placed on its execution stack. The process waiting for output (P, with resumption c) is placed in the ready pool. These communication offers, and all the ones linked in the same mutual exclusion pools (Pa, Pb and Pc below, and all the ones linked with Qq), are removed from the respective channel pools. Some channels may become inactive (e.g. C), and are then removed from the active pool.



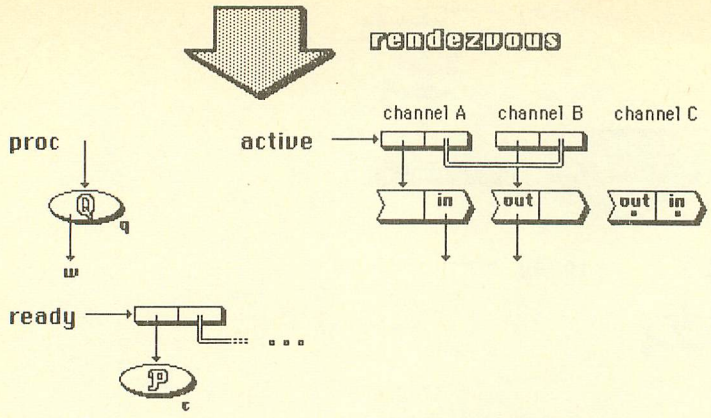


Figure 10: Rendezvous

Hence, the scheduler passes control to a process waiting for input, and eventually will give control to the corresponding process waiting for output, which is temporarily stored in the ready pool.

Conclusions

We have informally presented a set of concurrency-related primitives. They can be used as a basis for the implementation of rendezvous communication mechanisms in programming languages. The method used to describe these primitives can be formalized following [Plotkin 81], as was done in [Cardelli 83].

Several people have worked on implementations of, roughly, this model of communication, for example [Inmos 84], [Abramski 83] and the BCSP system at Oxford. I am confident that their solutions are not conceptually very different from what is described here. The aim here was to present these ideas by abstracting as much as possible from low-level implementation details, and from particular programming language features. The result is this intermediate level of abstraction given by a state transition model, lower than programming language semantics, but higher than program code.

References

[**Abramski 83**] S.Abramsky, R.Bornat: *Pascal-m: a language for loosely coupled distributed systems*. In *Distributed Computing Systems; Synchronization, Control and Communication*. Ed. Y.Parker, J.P.Verjus, Academic Press, 1983.

[**Cardelli 83**] L.Cardelli: *The functional abstract machine*. Bell Labs Technical Report TR-107, 1983.

[**Cardelli 84**] L.Cardelli: *Amber*. To appear.

[**Inmos 84**] INMOS Ltd.: *occam programming manual*. Prentice Hall, 1984

[**Milner 80**] R.Milner: *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, n. 92, Springer-Verlag, 1980.

[**Milner 82**] R.Milner: *Four combinators for concurrency*. ACM Sigact-Sigops Symposium on Principles of Distributed Computing, Ottawa, Canada, 1982.

[**Plotkin 81**] G.D.Plotkin: *A structural approach to Operational Semantics*. Internal Report DAIMI FN-19, Computer Science Department, Aarhus University, September 1981.

A FULLY ABSTRACT MODEL OF FAIR ASYNCHRONY

- extended abstract -

Philippe DARONDEAU
IRISA
Campus de Beaulieu
F 35042 Rennes Cedex
France

Assuming strong fairness for a CCS-like language, we construct a fully abstract model of the implementation preorder : $p \sqsubseteq q$ iff for every context $\mathcal{C}$ and for every program r, no computation of r in $(\mathcal{C}(p) | r)$ allows to recognize that q has been replaced by p.

1. THE LANGUAGE

1.1. THE SYNTAX

The language studied here evolved from the pure version of the asynchronous CCS of Milner [M. 1] . The main alterations are as follows : polyadic guarding operators replace the sum operator, and recursive definitions have been obliterated so that every term is a fully defined program. The limitations due to the latter feature are balanced by the supply of iterators for obtaining cyclic agents and unbounded networks of communicating agents, used much the same way as in pure CCS to simulate Turing machines.

In the sequel, we assume given disjoint sets of complementary action names Δ and $\overline{\Delta}$, ranged over by α resp. $\overline{\alpha}$, such that there exist reciprocal bijections $\alpha \rightarrow \overline{\alpha} \rightarrow \alpha$. We let Ren, ranged over by π, denote the set of domain finite injections over $\Lambda = \Delta \cup \overline{\Delta}$ such that :

$(\forall \lambda)\ (\pi\ (\lambda)\ \text{defined} \Rightarrow \pi\ (\overline{\lambda}) = \overline{\pi\ (\lambda)})\ \&$

$(\exists n)\ (\forall \lambda)\ (\pi^n\ (\lambda)\ \text{undefined})$.

Our language is the free term algebra T_Σ for the signature $\Sigma = \bigcup \Sigma_n$, $n \geq 0$ given by:

$\Sigma_0 = \{NIL\} \cup \{<\lambda \Rightarrow \lambda_1 : \lambda'_1, \ldots, \lambda_n : \lambda'_n> \mid \lambda, \lambda_i, \lambda'_i \in \Lambda\}$

$\Sigma_1 = \{<\lambda> \mid \lambda \in \Lambda\} \cup \{[\pi] \mid \pi \in Ren\} \cup \{[\lambda \triangleright \pi] \mid \lambda \in \Lambda, \pi \in Ren\}$

$\Sigma_2 = \{|\} \cup \{<\lambda_1, \lambda_2> \mid \lambda_i \in \Lambda\}$