Processes for Poly and ML

David C.J. Matthews

October 12, 1987

This document describes the process mechanism as it is currently implemented in Poly and ML. Though implementing processes in functional languages is interesting in itself the main motivation was the need to be able to handle asynchronous activities in a window system.

The system is largely based on the ideas of CCS [2] and the PFL implementation [1]. The main difference is in the *choice* function.

1 Channels

Poly

```
channel:
   PROC(t: TYPE END)
      TYPE
      assign: PROC(t);
      content: PROC()t
      END
```

ML

```
type 'a channel
val channel: unit -> 'a channel
val send:    'a channel * 'a -> unit
val receive: 'a channel -> 'a
```

Processes communicate over *channels* which are unbuffered. When a process wishes to send data it must wait until a second process is ready to receive it.

The data is transferred and both processes can proceed. Similarly a process wanting to receive is blocked until a process is prepared to send.

In both Poly and ML channels are created by a call to *channel.* The result however is slightly different because of differences in the type systems. In Poly a channel behaves exactly like a variable since it is a **type** with *assign* and *content* procedures. Values are sent by assigning to the channel and received by taking its value.

The type of a channel in ML is more complicated. There is a type, also called *channel,* which is the type of values returned from the *channel* function. Values are sent and received using the *send* and *receive* functions. A problem with this is that a channel is very similar to a reference and must be constrained to be a monotype if the security of the type system is to be preserved. This means there is a restriction that the result of the *channel* function must be a monotype which will almost certainly require an explicit type constraint.

2 Processes

Poly

```
process:
   TYPE
   fork: PROC(PROC());
   console: PROC(PROC())PROC();
   choice: PROC(PROC(); PROC())
   END
```

ML

```
val fork:    (unit->unit)->unit
val console: (unit->unit)->(unit->unit)
val choice:  (unit->unit)*(unit->unit)->unit
```

The simplest way of starting a process is with *fork.* This takes a procedure as its argument and runs it as a separate process. The process calling *fork* continues unaffected.

console is similar to *fork* but creates a process which will receive an exception if a console interrupt is raised. It also returns a procedure as its result which can be called to simulate a console interrupt in that process only. This provides a way of interrupting programs which have gone out of control.

choice forks two processes and returns immediately. However unlike calls to *fork* only one of the processes is allowed to send or receive on a channel. If both processes try to send or receive only one of them will be successful and the other will be blocked for ever. If the first action of both processes is to receive or send on a channel then *choice* behaves exactly like the choice primitive of CCS. Either of the processes created by *choice* can create more processes before sending or receiving. In that case the choice is between the processes created by the two alternatives. If any of the processes created by one alternative makes a successful transfer (either sends or receives data) then none of the processes created by the other alternative will be able to. In this way *choice* is associative and commutative since if the first action on either alternative is either to send or receive or to create a process with *choice* which behaves in this way, then only one of the choices will be taken, and the order of the choices is irrelevent.

References

[1] S. Holmström, *PFL: A Functional Language for Parallel Programming*, Proceedings of the Declarative Programming Workshop, University College London, 1983.

[2] R. Milner, *A Calculus for Communicating Systems*, Lecture Notes in Computer Science, no 81, Springer Verlag, 1981.