1 Introduction

The technical formulation of Concurrent Algol (CA) in Chapter 40 of PFPL is a mildly uncomfortable compromise of considerations. There is a tension between the desire to maintain the dynamics of MA largely intact (which is especially desirable when it is fleshed out into a full-scale language) and the desire to integrate mechanisms for allocating channels (classes), spawning new processes, and emitting and accepting messages (or, in the more general case, synchronizing on a disjunction of events). The compromise used in PFPL is to introduce a judgment representing the possible “local steps” of execution. A local step transforming one command into another can have one of three ancillary effects:

1. It can allocate a fresh channel.
2. It can spawn a new process.
3. It can engender a synchronization action.

These steps are integrated into the “global steps” governing the program as a whole, notably the synchronization of processes capable of undertaking complementary actions.

Another approach that avoids some of these complexities (or, more properly, moves them around) is to eliminate the idea that command execution can, in itself, engender effects. Rather, any effects are moved to the process level, where they are handled as in Chapter 39 on process calculus. Doing so avoids the need for the ad hoc mechanism of local execution steps, but requires that states be fleshed out to account for synchronization among processes. It is also necessary to introduce a choice mechanism, either an acausal choice, as is done here, or a causal choice, which is left as an exercise.

2 Alternative Dynamics of CA

The process level of the CA dynamics is changed as follows:

$$p ::= 1 | p_1 \otimes p_2 | 0 | p_1 + p_2 | \nu a.\tau \cdot p | ! e | ? a(x.p) | \text{run}[a](m)$$

There are two forms of receive process, generic and selective, and non-deterministic choice among processes. The atomic process consisting of a single command is parameterized by a channel indicating where the return value is to be sent; this is used to implement sequencing (see below).
The statics of processes is as given in Chapter 40, augmented with the rules given in Figure 1 governing choice, sending, receiving, and running a command. Messages are dynamically classified. The selective receive operation filters out those with a specified class, passing the associated value directly to the recipient process. The dynamics of processes is augmented by the rules given in Figure 2. Bear in mind that processes are identified up to structural congruence, so that, for example, choice applies to either summand implicitly, not just the left.

The dynamics of processes is given in Figure 3. Notice that commands are given dynamics only as processes; there is no need for the local action judgment used in PFPL. By convention a \texttt{ret} command sends the returned value on the channel naming the process. Then sequencing is implemented by allocating a channel, which is provided as destination to the first command in the sequence, and along which the returned value is communicated to the second command in the sequence. Spawning an encapsulated command creates a new process, with no destination, returning the unit value to the spawning process.\footnote{Alternatively, \texttt{spawn} could return a reference to the channel that names it, so that another process can await its termination for receiving on that channel.} Allocation of a new channel allocates a new channel and returns a reference to it to the allocating process. Emitting a message emits the message concurrently with the return of the unit value to the emitter. Generic and selective receive perform the indicated operations and return the resulting value to the receiving process. Synchronization on the null process transitions to the null choice, and synchronization on a binary choice transitions to a binary choice of synchronizations. Note that both choices are processes with the same name, which is sensible because only one can be chosen to be the “true” process with that name. Finally, synchronization on a wrapper transitions to a sequencing command that effects the wrapping after synchronization.

The use of acausal choice (among processes) facilitates giving the dynamics of \texttt{sync} in a compositional manner. From an implementation viewpoint acausal choice is problematic because it is not clear how to resolve the choice in such a way that progress is assured whenever possible. For
Figure 3: Revised CA Dynamics (Selected Rules)
example, consider the process
\[ p \triangleq !e \otimes ( ?a(x.p) + ?b(x.q) ). \]
The process \( p \) may spontaneously (acausally) transition to either
\[ p_a \triangleq !e \otimes ?a(x.p), \]
which commits to receiving on \( a \), or to
\[ p_b \triangleq !e \otimes ?b(x.q), \]
which commits to receiving on \( b \). But then if \( e \) is \( a \cdot e' \), then \( p_b \) is permanently blocked, or, if \( e \) is instead \( b \cdot e' \), then \( p_a \) is permanently blocked. Yet in either blocked case there would have been an alternative choice that would lead to further progress by synchronization. However, an implementation cannot know which would be the correct choice—particularly if the emitting process were created after the choice is to be made!

The solution is to consider only causal choice at the process level, in the form described in PFPL, and to modify the dynamics to make use of it. Specifically, the choice processes, \( 0 \) and \( p_1 + p_2 \), are replaced by synchronization processes of the form
\[ \$( ?a_1(x_1.p_1) + \ldots + ?a_n(x_n.p_n)) \]
where \( n \geq 0 \). This process cannot, in itself, make a transition, but communication is defined to choose among the receive events when synchronizing with an emitting process. Thus,
\[ \$( ?a_1(x_1.p_1) + \ldots + ?a_n(x_n.p_n)) \otimes !a \cdot e \xrightarrow{\varepsilon} \Sigma \left[ e/x \right] p_i, \]
with the choice being determined by the class of the emitted message.

The dynamics of \( \text{CA} \) using causal choice relies on a structural congruence on values of type \( \tau \text{event} \) such that every such value has the form of an \( n \)-ary disjunction of wrapped receive events. The critical idea is to postulate the distributivity of wrappers over other forms of event:
\[
\text{wrap} (\text{null}; x . e) \equiv \text{null} \\
\text{wrap} (\text{or} (e_1; e_2); x . e) \equiv \text{or} (\text{wrap} (e_1; x . e); \text{wrap} (e_2; x . e)) \\
\text{wrap} (\text{wrap} (e_1; x . e_2); x . e) \equiv \text{wrap} (e_1; x . \text{let} (e_2; x . e))
\]
Any event is then structurally congruent to a disjunction of events of the form \( \text{wrap} (\text{rcv}[a]; x . e) \). The dynamics of synchronization may then be given schematically as follows:

\[ \text{run}[a](\text{sync}(\text{or} (\text{wrap} (\text{rcv}[a_1]; x_1.e_1); \ldots))) \xrightarrow{\varepsilon} \Sigma \xrightarrow{\Sigma} \$( ?a_1(x_1.\text{run}[a](\text{ret}(e_1))) + \ldots \)
\]
In essence synchronization command on a \( \text{CA} \) event turns into a synchronization process among the specified receives, returning the wrapped message as result.

**Exercise 2.1.** *Extend the dynamics to account for adding exceptions as commands to \( \text{CA} \).*

**References**