PFPL Supplement: Concurrent Algol, Modernized*

Robert Harper

Fall, 2019

1 Introduction

The technical formulation of Concurrent Algol (**CA**) in Chapter 40 of **PFPL** is an uncomfortable compromise of competing considerations. There is a tension between the desire to maintain the dynamics of **MA** largely intact and the desire to integrate mechanisms for allocating channels, spawning new processes, and emitting and accepting messages.

The dynamics given in **PFPL** makes use of a judgment defining the "local steps" of execution that are integrated into the "global steps" governing the program as a whole, in particular the definition of synchronization among processes. 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.

Each local step determines a global step that integrates the effects into the global state. The method works, but cannot be commended for its elegance.

Another approach is to consider that the global state is a *process* in the sense of the π-calculus. The virtue of this formulation is that all global steps are specified locally, relying on the machinery of process calculus to integrate effects as appropriate. As a bonus the same methods may be used to define the dynamics of **MA**, regarding assignable cells as "servers" that respond to get and put requests; see Harper (2020) for the particulars.

2 Reformulated Processes for **CA**

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

$$p ::= \mathbf{1} \mid p_1 \otimes p_2 \mid \mathbf{0} \mid p_1 \oplus p_2 \mid \nu\, a \sim \tau\, .\, p \mid \mathtt{emit}(\, e\,) \mid \mathtt{recv}\langle a\rangle(\, x\, .\, p\,) \mid \mathtt{run}\langle a\rangle(\, m\,)$$

There is one form of receive process, 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).

S-NULL
$$\frac{}{\Gamma \vdash_\Sigma \mathbf{0}\ \mathsf{proc}}$$

S-CHOOSE
$$\frac{\Gamma \vdash_\Sigma p_1\ \mathsf{proc} \qquad \Gamma \vdash_\Sigma p_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma p_1 \oplus p_2\ \mathsf{proc}}$$

S-EMIT
$$\frac{\Gamma \vdash_\Sigma e : \mathtt{clsfd}}{\Gamma \vdash_\Sigma \mathtt{emit}(\,e\,)\ \mathsf{proc}}$$

S-SRCV
$$\frac{\Gamma, x : \tau \vdash_{\Sigma, a\sim\tau} p\ \mathsf{proc}}{\Gamma \vdash_{\Sigma,a\sim\tau} \mathtt{recv}\langle a\rangle(\,x\,.\,p\,)\ \mathsf{proc}}$$

S-RUN
$$\frac{\Gamma \vdash_{\Sigma,a\sim\tau} m \mathrel{\dot{\sim}} \tau}{\Gamma \vdash_{\Sigma,a\sim\tau} \mathtt{run}\langle a\rangle(\,m\,)\ \mathsf{proc}}$$

Figure 1: Special Process Forms for **CA**: Statics

D-CHOOSE
$$\frac{}{p_1 \oplus p_2 \underset{\Sigma}{\longmapsto} p_1}$$

D-EMIT
$$\frac{e\ \mathsf{val}_\Sigma}{\mathtt{emit}(\,e\,) \underset{\Sigma}{\overset{e\,!}{\longmapsto}} \mathbf{1}}$$

D-SRCV
$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\mathtt{recv}\langle a\rangle(\,x\,.\,p\,) \underset{\Sigma,a\sim\tau}{\overset{a\cdot e\,?}{\longmapsto}} \{e/x\}p}$$

Figure 2: Special Process Forms for **CA**: Dynamics

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.

3 Reformulated Dynamics of CA

The dynamics of **CA** 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**. There are several notable aspects of this dynamics.

A central issue is the management of sequencing of commands. Here, in contrast to **PFPL**, the dynamics of `bnd` and `ret` is given in terms of process synchronization. Rule D-BND allocates a private channel and creates a `run` process executing the first command in the sequence, then arranges for the second to await the completion of the first by querying the private channel. Correspondingly, a execution of a RET command sends the returned value along the channel associated with it. Once the first command has completed, the send synchronizes with the receive, and continues with the appropriate instance of the second command in the sequence.

The concurrency primitives are managed using the same process calculus mechanisms. Spawning an encapsulated command creates a new process, with no destination, returning the unit value to the spawning process.[1] 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. Receives perform the indicated operations and return the resulting

[1] Alternatively, `spawn` could return a reference to the channel that names it, so that another process can await its termination for receiving on that channel.

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 at a synchronization step. Finally, synchronizing on a wrapper transitions to a sequencing command that effects the wrapping.

The use of acausal choice (among processes) facilitates giving the dynamics of `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 example, consider the process

$$p \triangleq \texttt{emit}(\, e \,) \otimes (\, \texttt{recv}\langle a \rangle(\, x \,.\, p \,) + \texttt{recv}\langle b \rangle(\, x \,.\, q \,)\,).$$

The process p may spontaneously (acausally) transition to either

$$p_a \triangleq \texttt{emit}(\, e \,) \otimes \texttt{recv}\langle a \rangle(\, x \,.\, p \,),$$

which commits to receiving on a, or to

$$p_b \triangleq \texttt{emit}(\, e \,) \otimes \texttt{recv}\langle b \rangle(\, x \,.\, q \,),$$

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, $\mathbf{0}$ 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,

$$\$\,(\,?\,a_1(\, x_1 \,.\, p_1 \,) + \ldots + ?\,a_n(\, x_n \,.\, p_n \,)\,) \otimes \texttt{emit}(\, a_i \cdot e \,) \underset{\Sigma}{\longmapsto} \{e/x\}p_i,$$

with the choice being determined by the class of the emitted message; the choices not taken are thereby abandoned.

When restricted to causal choice, processes have, up to structural congruence, the form

$$\nu\, \Sigma \,.\, p_{send} \otimes p_{recv} \otimes p_{run},$$

wherein

1. p_{send} is a concurrent composition of asynchronous send processes, $!\, a \cdot v$, representing a send of a value v along channel a. There may be multiple send processes on a given channel.

2. p_{recv} is a concurrent composition of synchronization processes, each making a choice of some number of events $\texttt{recv}\langle a \rangle(\, x \,.\, p \,)$, representing a pending receive of a message on channel a. There may be many receiving events for a given channel, distributed across any number of synchronizations.

D-RET

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\tau}}{\mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,e\,)\,) \xmapsto[\Sigma,a\sim\tau]{} \mathtt{emit}(\,(\,a\cdot e\,)\,)}$$

D-BND

$$\frac{(p_1 \triangleq \mathtt{run}\langle a_1\rangle(\,m_1\,),\ p_2 \triangleq \mathtt{recv}\langle a_1\rangle(\,x\,.\,\mathtt{run}\langle a\rangle(\,m_2\,)\,))}{\mathtt{run}\langle a\rangle(\,\mathtt{bnd}(\,\mathtt{cmd}[\,\tau_1\,](\,m_1\,)\,;x\,.\,m_2\,)\,) \xmapsto[\Sigma,a\sim\tau]{} \nu\, a_1 \sim \tau_1\,.\,\{p_1 \otimes p_2\}}$$

D-SPAWN

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{spawn}(\,\mathtt{cmd}[\,\tau\,](\,m\,)\,)\,) \xmapsto[\Sigma,a\sim\mathtt{unit}]{} \nu\, b \sim \tau\,.\,\mathtt{run}\langle b\rangle(\,m\,) \otimes \mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,\langle\rangle\,)\,)}$$

D-NEWCH

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{newch}[\,\tau\,]\,) \xmapsto[\Sigma a\sim\mathtt{cls}(\,\tau\,)]{} \nu\, b \sim \tau\,.\,\mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,\&\,b\,)\,)}$$

D-EMIT

$$\frac{e\ \mathsf{val}_{\Sigma,a\sim\mathtt{unit}}}{\mathtt{run}\langle a\rangle(\,\mathtt{emit}(\,e\,)\,) \xmapsto[\Sigma,a\sim\mathtt{unit}]{} \mathtt{emit}(\,e\,) \otimes \mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,\langle\rangle\,)\,)}$$

D-SRCV

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,\mathtt{rcv}\langle b\rangle\,)\,) \xmapsto[\Sigma,a\sim\tau]{} \mathtt{recv}\langle b\rangle(\,x\,.\,\mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,x\,)\,)\,)}$$

D-NULL

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,\mathtt{null}\,)\,) \xmapsto[\Sigma,a\sim\tau]{} \mathbf{0}}$$

D-CHOOSE

$$\frac{e_1\ \mathsf{val}_{\Sigma,a\sim\tau} \qquad e_2\ \mathsf{val}_{\Sigma,a\sim\tau}}{\mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,\mathtt{or}(\,e_1\,;e_2\,)\,)\,) \xmapsto[\Sigma,a\sim\tau]{} \mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,e_1\,)\,) \oplus \mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,e_2\,)\,)}$$

D-WRAP

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,\mathtt{wrap}(\,e_1\,;x\,.\,e_2\,)\,)\,) \xmapsto[\Sigma,a\sim\tau]{} \mathtt{run}\langle a\rangle(\,\mathtt{bnd}(\,\mathtt{cmd}(\,\mathtt{sync}(\,e_1\,)\,)\,;x\,.\,\mathtt{ret}(\,e_2\,)\,)\,)}$$

Figure 3: Revised **CA** Dynamics (Selected Rules)

3. p_{run} is a concurrent composition of run processes of the form $\mathtt{run}\langle a\rangle(\, m\,)$, representing an executing thread. No two run processes will have the same process name.

In implementation terms the sending processes represent a multiset of values that have been sent along each active channel, serving as a "buffer" for pending messages that have not yet been received. Correspondingly, the synchronizing processes constitute a "registry" of interest in a choice of receive events, with repetitions among channels allowed. The running processes represent the "thread pool" of active processes capable of executing a command. A scheduler chooses an active thread to execute, updating the send buffers and receive registry as necessary; moreover, it ensures that pending messages synchronize with active receivers to make a (causal!) choice and activate the receiver.

The dynamics of **CA** using causal choice relies on a structural congruence on values of type $\tau\,\mathtt{event}$ 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:

$$\begin{aligned}\mathtt{wrap}(\,\mathtt{null}\,;x\,.\,e\,) &\equiv \mathtt{null}\\ \mathtt{wrap}(\,\mathtt{or}(\,e_1\,;e_2\,)\,;x\,.\,e\,) &\equiv \mathtt{or}(\,\mathtt{wrap}(\,e_1\,;x\,.\,e\,)\,;\mathtt{wrap}(\,e_2\,;x\,.\,e\,)\,)\\ \mathtt{wrap}(\,\mathtt{wrap}(\,e_1\,;x\,.\,e_2\,)\,;x\,.\,e\,) &\equiv \mathtt{wrap}(\,e_1\,;x\,.\,\mathtt{let}(\,e_2\,;x\,.\,e\,)\,)\end{aligned}$$

Any event is then structurally congruent to a disjunction of events of the form $\mathtt{wrap}(\,\mathtt{rcv}\langle a\rangle\,;x\,.\,e\,)$. The dynamics of synchronization may then be given schematically as follows:

D-SYNC

$$\frac{}{\mathtt{run}\langle a\rangle(\,\mathtt{sync}(\,\mathtt{or}(\,\mathtt{wrap}(\,\mathtt{rcv}\langle a_1\rangle\,;x_1\,.\,e_1\,)\,;\ldots\,)\,)\,)\underset{\Sigma}{\longmapsto}\$\,(\,\mathtt{recv}\langle a_1\rangle(\,x_1\,.\,\mathtt{run}\langle a\rangle(\,\mathtt{ret}(\,e_1\,)\,)\,)\oplus\ldots\,)}$$

In essence synchronization command on a **CA** event turns into a synchronization process among the specified receives, returning the wrapped message as result.

Exercise 3.1. *Extend the dynamics to account for adding exceptions as commands to* ***CA****.*

References

Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Modernized Algol, Concurrently. Supplement to Harper (2016), December 2020. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/cma.pdf`.