PFPL Supplement: Concurrent Algol, Modernized*

Robert Harper

Spring, 2024

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 Processes for CA

The language of processes for **CA** is essentially that given in **PFPL**, but with the change that all messages are of type `clsfd`, with receive events matching against the class of the message to determine whether they are eligible to process that message. For clarity, the statics of the `emit` primitive and for a running command are given in Figure 1, along with the labelled transition for emitting a value of type `clsfd`, and selectively receiving such a value according to an event with matching class.

$$\text{S-EMIT}\quad \frac{\vdash_\Sigma e : \mathtt{clsfd}}{\vdash_\Sigma \mathtt{emit}(\,e\,)\ \mathsf{proc}} \qquad \text{S-RUN}\quad \frac{\vdash_\Sigma m \div \tau \qquad \Sigma \vdash a \sim \tau}{\vdash_\Sigma \mathtt{run}[\,a\,](\,m\,)\ \mathsf{proc}}$$

$$\text{D-EMIT}\quad \frac{e\ \mathsf{val}_\Sigma}{\mathtt{emit}(\,e\,) \underset{\Sigma}{\overset{e!}{\longmapsto}} \mathbf{1}} \qquad \text{D-AWAIT}\quad \frac{}{\mathtt{await}(\,\mathtt{rcv}[\,a\,](\,x\,.\,P\,) + \ldots\,) \underset{\Sigma}{\overset{a\cdot v\,?}{\longmapsto}} \{v/x\}P}$$

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

$$\text{D-RET}\quad \frac{e \Downarrow_\Sigma v}{\mathtt{run}[\,a\,](\,\mathtt{ret}(\,e\,)\,) \underset{\Sigma}{\longmapsto} \mathtt{emit}(\,a \cdot v\,)}$$

$$\text{D-BND}\quad \frac{e_1 \Downarrow_\Sigma \mathtt{cmd}[\,\tau_1\,](\,m_1\,)}{\mathtt{run}[\,a\,](\,\mathtt{bnd}(\,e_1\,;x\,.\,m_2\,)\,) \underset{\Sigma}{\longmapsto} \nu\, a_1 \sim \tau_1\,.\,\mathtt{run}[\,a_1\,](\,m_1\,) \otimes \mathtt{await}(\,\mathtt{rcv}[\,a_1\,](\,x\,.\,\mathtt{run}[\,a\,](\,m_2\,)\,)\,)}$$

$$\text{D-NEWCH}\quad \frac{}{\mathtt{run}[\,a\,](\,\mathtt{newch}[\,\tau\,]\,) \underset{\Sigma}{\longmapsto} \nu\, b \sim \tau\,.\,\mathtt{run}[\,a\,](\,\mathtt{ret}(\,\&\,b\,)\,)}$$

$$\text{D-SPAWN}\quad \frac{e \Downarrow_\Sigma \mathtt{cmd}[\,\tau\,](\,m\,)}{\mathtt{run}[\,a\,](\,\mathtt{spawn}(\,e\,)\,) \underset{\Sigma}{\longmapsto} \nu\, b \sim \tau\,.\,\mathtt{run}[\,b\,](\,m\,) \otimes \mathtt{run}[\,a\,](\,\mathtt{ret}(\,\langle\rangle\,)\,)}$$

$$\text{D-SYNC}\quad \frac{e \Downarrow_\Sigma v \qquad v \equiv \mathtt{or}(\,\mathtt{rcv}[\,a_1\,](\,x\,.\,e_1\,)\,;\mathtt{or}(\,\ldots\,;\mathtt{rcv}[\,a_n\,](\,x\,.\,e_n\,)\,)\,)}{\mathtt{run}[\,a\,](\,\mathtt{sync}(\,e\,)\,) \underset{\Sigma}{\longmapsto} \mathtt{await}(\,\mathtt{rcv}[\,a_1\,](\,x\,.\,\mathtt{run}[\,a\,](\,\mathtt{ret}(\,e_1\,)\,)\,) + \ldots + \mathtt{rcv}[\,a_n\,](\,x\,.\,\mathtt{run}[\,a\,](\,\mathtt{ret}(\,e_n\,)\,)\,)\,)}$$

Figure 2: **CA** Dynamics (Selected Rules)

3 Revised Dynamics of CA

As a technical convenience, the receive event of **CA** is herein replaced by a more general form, written `rcv`$[\,a\,](\,x\,.\,e\,)$, which indicates willingness to receive a message with class a, binding its associated value to x within e to determine the resulting value of that event. By choosing e to be x, the original form of receive event is obtained. It is then a simple exercise to define `wrap`$(\,e_1\,;\,x\,.\,e_2\,)$ in terms of the other event constructs (nullary and binary choice, the new form of receive). Importantly, **CA** event values are taken modulo the commutativity, associativity, and unit laws for the nullary and binary disjunctions, so that any value of event type is, essentially, a disjunction of receive events of the form just described.

With this in mind the dynamics of **CA** is given in Figure 2. There are several notable aspects of this dynamics. First, 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 corresponding 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. Synchronizations turns into process synchronizations, albeit with the continuation being a process that returns the provided value.

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`.

[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.