PFPL Supplement: Modernized Algol, Concurrently*

Robert Harper

Summer, 2020

1 Introduction

The scoped dynamics of Modernized Algol in **PFPL** Chapter 34 is given by transitions of the form

$$\mu \parallel m \underset{\Sigma}{\longmapsto} \mu' \parallel m'$$

in which Σ determines the assignables in scope, μ determines their contents, and m is any command that is well-formed in that scope. The scope-free dynamics given in Chapter 35 is given by transitions of the form

$$\nu\,\Sigma\,\{\,\mu \parallel m\,\} \longmapsto \nu\,\Sigma'\,\{\,\mu' \parallel m'\,\}$$

which gives global scope to all assignables, and permits assignables to persist beyond the body of their declaration.

These disparate formulations may be unified by using ideas from process calculus to manage the state of a computation. The main idea is to think of the assignable cells as "servers" that interact with the main program during its execution. The deallocation of scoped assignables is expressed using principles of structural congruence to eliminate unneeded cells.

2 States as Processes

The execution states of **MA** are replaced by *processes* given by the following grammar:

$$p ::= \mathbf{1} \mid p_1 \otimes p_2 \mid \nu\, a \sim \tau\,.\,p \mid a \hookrightarrow v \mid \mathtt{run}(\,m\,) \mid \mathtt{cont}[\,\tau\,](\,x\,.\,m\,)$$

The first two process forms represent concurrent composition. The third process form represents the allocation of a channel for use within a process. Assignable cells, $a \hookrightarrow v$, are primitive forms of process, as is the running command, $\mathtt{run}(\,m\,)$, and a continuation process, $\mathtt{cont}[\,\tau\,](\,x\,.\,m\,)$. (The role of the continuation process will become evident shortly.) The appropriate form of structural congruence depends on whether assignables are free or scoped; this will be discussed in the next section.

The statics of processes is defined in Figure 1, and their dynamics is given in Figure 2.

$$\textsc{S-STOP}\ \frac{}{\Gamma \vdash_\Sigma \mathbf{1}\ \mathsf{proc}} \qquad \textsc{S-CONC}\ \frac{\Gamma \vdash_\Sigma p_1\ \mathsf{proc} \qquad \Gamma \vdash_\Sigma p_2\ \mathsf{proc}}{\Gamma \vdash_\Sigma p_1 \otimes p_2\ \mathsf{proc}} \qquad \textsc{S-NEW}\ \frac{\Gamma \vdash_{\Sigma, a\sim\tau} p\ \mathsf{proc}}{\Gamma \vdash_\Sigma \nu\, a \sim \tau\, .\, p\ \mathsf{proc}}$$

$$\textsc{S-MEM}\ \frac{\Gamma \vdash_\Sigma e : \tau \qquad \Sigma \vdash a \sim \tau}{\Gamma \vdash_\Sigma a \hookrightarrow e\ \mathsf{proc}} \qquad \textsc{S-RUN}\ \frac{\Gamma \vdash_\Sigma m \div \rho}{\Gamma \vdash_\Sigma \mathtt{run}(\, m\,)\ \mathsf{proc}} \qquad \textsc{S-CONT}\ \frac{\Gamma, x : \tau \vdash_\Sigma m \div \rho}{\Gamma \vdash_\Sigma \mathtt{cont}[\tau](\, x\, .\, m\,)\ \mathsf{proc}}$$

Figure 1: Process Forms: Statics

$$\textsc{D-CONC}\ \frac{p_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1'}{p_1 \otimes p_2 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1' \otimes p_2} \qquad \textsc{D-SYNC}\ \frac{p_1 \overset{\alpha}{\underset{\Sigma}{\longmapsto}} p_1' \qquad p_2 \overset{\overline{\alpha}}{\underset{\Sigma}{\longmapsto}} p_2'}{p_1 \otimes p_2 \underset{\Sigma}{\longmapsto} p_1' \otimes p_2'} \qquad \textsc{D-NEW}\ \frac{p \overset{\alpha}{\underset{\Sigma, a\sim\tau}{\longmapsto}} p' \qquad \vdash_\Sigma \alpha\ \mathsf{action}}{\nu\, a \sim \tau\, .\, p \overset{\alpha}{\underset{\Sigma}{\longmapsto}} \nu\, a \sim \tau\, .\, p'}$$

$$\textsc{D-GET}\ \frac{e\ \mathsf{val}_{\Sigma, a\sim\tau}}{a \hookrightarrow e \overset{a!e}{\underset{\Sigma, a\sim\tau}{\longmapsto}} a \hookrightarrow e} \qquad \textsc{D-SET}\ \frac{e'\ \mathsf{val}_{\Sigma, a\sim\tau}}{a \hookrightarrow e \overset{a?e'}{\underset{\Sigma, a\sim\tau}{\longmapsto}} a \hookrightarrow e'}$$

Figure 2: Process Forms: Dynamics

3 Concurrent Dynamics of MA

The dynamics of **MA** is given in Figure 3 in terms of the process forms given in Section 2.[1] The main feature of the dynamics is that each rule is given in "local form," in the sense that it need not specify the entire state, but only those parts that are relevant to each command. This is the prime advantage afforded by the use of process calculus for execution states.

An initial state is a command to be executed, $\mathtt{run}(\, m\,)$. A final state is one in which the running command is returning a value; it may be surrounded by any number of allocated cells, but not by another running command or a continuation.

The return and bind commands are managed using the continuation process and a designated channel, `ret`, on which to communicate results. Execution of a bind executes the encapsulated command, and spawns a continuation process expecting its result. Execution of a return transmits the return value along channel `ret`, and the continuation process is prepared to receive that value and continue by passing that value to the body of the bind that created it.

The declaration command allocates a new assignable and spawns a cell process governing it. Cell processes synchronize with the running process along the assignable. A cell process is always prepared to send its current value to the running command, and to receive a new value from the running command. Correspondingly, the get and set commands are executed by posting the appropriate request, to obtain the contents of a cell or to update its contents, respectively. It is

[1] Bind and return are handled as in Harper (2019), using process names that are disjoint from assignables to avoid confusion.

impossible for a cell to interact with itself, because processes cannot be duplicated.

The choice of structural congruence for processes is influenced by whether assignables are scoped or free. The allocation of assignables is formulated in Rule D-DCL, which allocates a fresh assignable and associates a cell process with it. Structural congruence rules are used to manage their scope and their deallocation.

When assignables are scoped, the mobility restriction on the type of their contents ensures that they may only occur within the running command, and not within its returned value or within any other cell. Bearing this in mind, the stack allocation of scoped assignables is expressed by the following rule of structural congruence for processes:

$$\textsc{pop}\quad \frac{v\ \mathsf{val}_{\Sigma,a\sim\tau} \qquad e\ \mathsf{val}_{\Sigma}}{\nu\, a \sim \tau\,.\, a \hookrightarrow v \otimes \mathtt{run}(\,\mathtt{ret}[\,\tau\,](\,e\,)\,) \equiv_{\Sigma} \mathtt{run}(\,\mathtt{ret}[\,\tau\,](\,e\,)\,)}$$

No other principles of structural congruence are necessary. Rule D-DCL may then be read as allocating a frame on the data stack that is then deallocated by Rule POP.

When assignables are scope-free, they may occur within the contents of any cell, including its own, and may escape the scope of the declaration that introduces it. To account for the global scope of assignables, it is necessary to postulate the following principle of *scope extrusion* for channels:

$$\textsc{scope}\quad \frac{(a \notin p_2)}{\nu\, a \sim \tau\,.\, \{p_1 \otimes p_2\} \equiv_{\Sigma} \{\nu\, a \sim \tau\,.\, p_1\} \otimes p_2}$$

When read from right-to-left, Rule SCOPE states that the scope of an assignable may be *widened* to include any other active cell, thereby permitting cycles as would arise by backpatching, or in any other form of self-referential data structure such as a circularly linked list.

When read from left-to-right, the same rule states that the scope can be *narrowed* to exlucde any process (command, continuation, or cell) that does not mention it. This is important for deallocation of free assignables. For example, the following rule states that any isolatable cell may be reclaimed:

$$\textsc{dealloc-one}\quad \frac{v\ \mathsf{val}_{\emptyset}}{\nu\, a \sim \tau\,.\, a \hookrightarrow v \equiv_{\Sigma} \mathbf{1}}.$$

A cell process may be isolated using Rule SCOPE to narrow its scope to the extent that it is not used in the running command or continuation, nor in any other cell.

Exercise 3.1. *Rule (3) permits the deallocation of a single isolatable cell. Such a cell may be self-referential (construct an example!), but larger cycles are not isolatable in this sense, and hence not collectible. Generalize this rule to permit deallocation of cycles in memory using a similar principle of structural congruence.*

The dynamics enjoys a local form of safety analogous to that for Concurrent Algol:

Theorem 3.1 (Weak Safety)**.** *If* $\vdash_{\Sigma} p\ \mathsf{proc}$*, then either* $p\ \mathsf{final}_{\Sigma}$*, or there exists unique* $\vdash_{\Sigma} \alpha\ \mathsf{action}$ *and* $\vdash_{\Sigma} p'\ \mathsf{proc}$ *such that* $p \xmapsto[\Sigma]{\alpha} p'$*.*

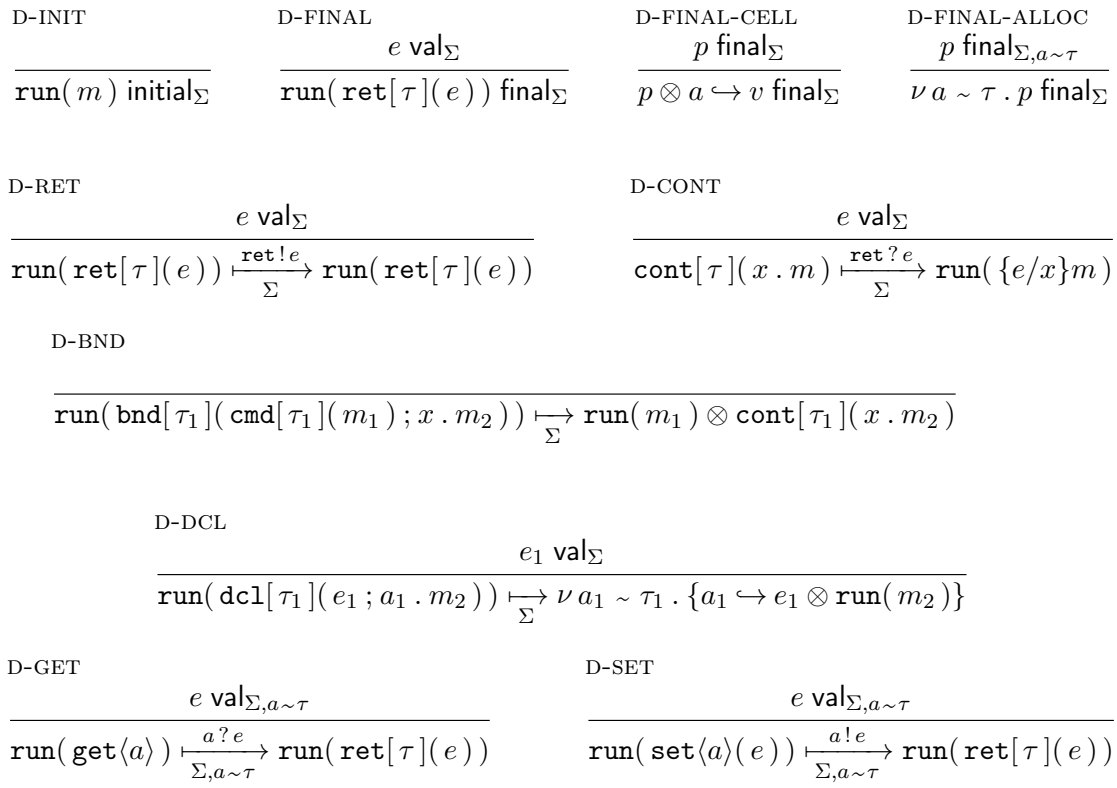

Figure 3: Concurrent Dynamics of **MA** (Selected Rules)

Exercise 3.2. *Prove Theorem 3.1.*

Theorem 3.1 permits states (well-formed processes) that cannot make an unlabelled transition (true progress). It may be strengthened to exclude this possibility by observing that the dynamics given in Figure 3 ensures the following additional properties of the state that are not captured by the statics alone:[2]

1. There is exactly one running command.

2. There is exactly one cell process associated to each assignable.

Note, in particular, that Rules D-RET and D-CONT enable an interaction that hands off between a running process and a continuation to maintain these invariants.

Exercise 3.3. *State and prove a strengthening of Theorem 3.1 that ensures progress, as would be expected for* ***MA****.*

[2]When sequencing is accounted as in Harper (2019), there can be any number of continuation processes, and neither any of these nor the running process can interact with cells.

References

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

Robert Harper. Concurrent Algol, Modernized. Supplement to Harper (2016), Fall 2019. URL `https://www.cs.cmu.edu/~rwh/pfpl/supplements/mca.pdf`.