Lecture Notes on Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 18
Tuesday, November 9, 2021

1 Introduction

The main objective of this lecture is to start making the role of memory explicit in a description of the dynamics of our programming language. Towards that goal, we take several steps at the same time:

1. We introduce a translation from our source language of *expressions* to an intermediate language of concurrent *processes* that act on (shared) memory. The sequential semantics of our original language can be recovered as a particular *scheduling policy* for concurrent processes.
2. We introduce a new collection of semantic objects that represent the state of processes and the shared memory they operate on. The presentation is as a *substructural operational semantics* [Pfe04, PS09, CS09]
3. We introduce *destination-passing style* [CPWW02] as a particular style of specification for the dynamics of programming languages that seems to be particularly suitable for an explicit store.

We now start to develop the ideas in a piecemeal fashion. This lecture is based on very recent work, at present under submission [DPP20, PP20].

2 Representing the Store

Our typing judgment for expressions is

$$\Gamma \vdash e : \tau$$

By the time we actually evaluate e, all the variables declared in Γ will have been replaced by values v (values, because we are in a call-by-value language, with variables for fixed point expressions representing an exception to that rule). Evaluation of *closed* expressions e proceeds as

$$e \mapsto e_1 \mapsto e_2 \mapsto \cdots \mapsto v$$

where v (if the computation is finite) represents the final outcome of the evaluation. A nice property of this formulation of the dynamics is that it does not require any semantic artifacts: we stay entirely within the language of expressions (which include values). The K Machine from Lecture 12 introduced continuations as a first dynamic artifact.

The main dynamic artifact we care about in this lecture is a representation of the *store* or *memory*, terms we use interchangeably. In our formulation, cells can hold only *small values* W (yet to be defined) and we write

$$\mathsf{cell}\ c_0\ W_0, \mathsf{cell}\ c_1\ W_1, \ldots, \mathsf{cell}\ c_n\ W_n$$

where all c_i are distinct. We read $\mathsf{cell}\ c\ W$ as "*cell c contains W*" or "*the memory at address c holds W*". We will shortly generalize this further.

As an example, before we actually see how these arise, let's consider the representation of a list. We define

$$\mathit{list}\ \alpha \cong (\mathbf{nil} : 1) + (\mathbf{cons} : \alpha \times \mathit{list}\ \alpha)$$

Then a list with two values $v_1 : \tau$ and $v_2 : \tau$ would be written as an *expression*

$$\mathsf{fold}\ (\mathbf{cons} \cdot \langle v_1, \mathsf{fold}\ (\mathbf{cons} \cdot \langle v_2, \mathsf{fold}\ (\mathbf{nil} \cdot \langle\,\rangle)\rangle)\rangle) : \mathit{list}\ \tau$$

Our representation of this in memory at some initial address c_0 would be

$$\begin{array}{l}
\mathsf{cell}\ c_8\ \langle\,\rangle \\
\mathsf{cell}\ c_7\ (\mathbf{nil} \cdot c_8), \\
\mathsf{cell}\ c_6\ (\mathsf{fold}\ c_7), \\
\mathsf{cell}\ c_5\ \langle a_2, c_6\rangle, \\
\mathsf{cell}\ c_4\ (\mathbf{cons} \cdot c_5), \\
\mathsf{cell}\ c_3\ (\mathsf{fold}\ c_4), \\
\mathsf{cell}\ c_2\ \langle a_1, c_3\rangle, \\
\mathsf{cell}\ c_1\ (\mathbf{cons} \cdot c_2), \\
\mathsf{cell}\ c_0\ (\mathsf{fold}\ c_1)
\end{array}$$

Here, we assume a_1 is the *address* of v_1 in memory, and a_2 the address of v_2. You can see a list of length n requires $3n + 3$ cells. In a lower-level representation this could presumably be optimized by compressing the information.

3 From Expressions to Processes

We translate expressions e to processes P. Instead of returning a value v, a process P executes and writes the result of computation to a *destination* d which is the address of a cell in memory. So we write the translation as

$$[\![e]\!]\, d = P$$

which means that expression e translates to a process P that computes with destination d. Given an expression

$$\Gamma \vdash e : \tau$$

its translation $P = [\![e]\!]\, d$ will be typed as

$$\Gamma \vdash P :: (d : \tau)$$

In this typing judgment we have made the destination d of the computation explicit. But the reinterpretation does not end there: we also no longer substitute values for the variables in Γ. Instead, we substitute *addresses*, so the process P can *read* from memory at the addresses in Γ and must *write* to the destination d (unless it does not terminate). We will also arrange that after writing to destination d the process P will immediately terminate. Explicitly:

$$\underbrace{c_1 : \tau_1, \ldots, c_n : \tau_n}_{\text{read from}} \vdash P :: \underbrace{(d : \tau)}_{\text{write to}}$$

Because at the moment we are only interested in modeling our pure functional language and *not* arbitrary mutation of memory, we require that all the c_i and d are distinct.

For each process P that is executing we have a semantic object

$$\mathsf{proc}\ d\ P$$

which means that P is executing with destination d. We do not make the cells that P may read from explicit because it would introduce unnecessary clutter.

4 Allocation and Spawn

Given the logic explained in the preceding sections, there is a single construct in our language of processes that accomplishes two things: (a) it allocates a

new cell in memory, and (b) it spawns a process whose job it is to write to this cell. We may also have a single initial cell c_0 to hold the outcome of the overall computation. We write this as

$$\text{Process} \quad P \quad ::= \quad x \leftarrow P \,;\, Q \mid \ldots$$

where the scope of x includes both P and Q. More specifically, a new destination c is created, P is spawned with destination c, and Q can read from c (once its value has been written. We formalize this as

$$\mathcal{C}, \mathsf{proc}\ d\ (x \leftarrow P\ ;\ Q) \mapsto \mathcal{C}, \mathsf{proc}\ c\ ([c/x]P), \mathsf{proc}\ d\ ([c/x]Q) \quad (c\ \text{fresh})$$

Here $\mathcal{C}$ represents the remaining *configuration*, which includes the representation of memory and other processes that may be executing. The freshly allocated cell at address c is uninitialized to start with. It represents a point of synchronization between P and Q, because Q can only read from it after P has written to it. Except for this synchronization point, P and Q can now evolve independently.

From a typing perspective, we can see that the type of two occurrences of the cell x must match.

$$\frac{\Gamma \vdash P :: (x : \tau) \quad \Gamma, x : \tau \vdash Q :: (d : \sigma)}{\Gamma \vdash x \leftarrow P \,;\, Q :: (d : \sigma)}\ \mathsf{cut}$$

This rule is called *cut* because of this name for the corresponding logical rule in the *sequent calculus*

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash C}{\Gamma \vdash C}\ \mathsf{cut}$$

where A acts as a lemma in the proof of C from Γ.

The configuration is not intrinsically ordered, so the process with destination d can occur anywhere in a configuration. Nevertheless, we follow a convention writing a configuration (or part of a configuration) so that a cell c precedes all the processes that may read from c or other cells that contain c. Because we do not have arbitrary mutation of store there cannot be any cycles (although we have to carefully reconsider this point when we consider fixed point expressions).

Since all of our rules only operate locally on a small part of the configuration, we generally omit $\mathcal{C}$ to stand for the remainder of the configuration. But we always have to remember that we *remove* the part of the configuration matching the left-hand side of a transition rule and then we *add in* the right-hand side.

5 Copying

Before we get into the constructors and destructors for specific types in our source language of expressions, let's consider the translation of variables. We write

$$[\![x]\!]\, d = d \leftarrow x$$

The intuitive meaning of the process expression $d \leftarrow x$ is that it copies the contents of the cell at address x to address d. Thereby, this process has written to its destination d and terminates.

$$\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (d \leftarrow c) \mapsto \mathsf{cell}\ c\ W, \mathsf{cell}\ d\ W$$

In this rule the cell c should have been written to already, and we just copy its value (which is small) to d.

The typing rule just requires that c and d have the same type (otherwise copying would violate type preservation).

$$\frac{}{\Gamma, c : \tau \vdash (d \leftarrow c) :: (d : \tau)}\ \mathsf{id}$$

From a logical perspective, it explains that the antecedent A entails the succedent A in the sequent calculus, usually called the identity rule.

$$\frac{}{\Gamma, A \vdash A}\ \mathsf{id}$$

6 The Unit Type

Recall the constructor and destructor for the unit type 1.

$$\text{Expressions} \quad e \quad ::= \quad \langle\rangle \mid \mathsf{case}\ e\ (\langle\rangle \Rightarrow e') \mid \ldots$$

The unit element is already a small value, so it can be written directly to memory. Our notation for this is $d.\langle\rangle$.

$$[\![\langle\rangle]\!]\, d = d.\langle\rangle$$
$$\mathsf{proc}\ d\ (d.\langle\rangle) \mapsto \mathsf{cell}\ d\ \langle\rangle$$

$$\frac{}{\Gamma \vdash d.\langle\rangle :: (d : 1)}\ 1R$$

The way we evaluate case e $(\langle\,\rangle \Rightarrow e')$ is to first evaluate e and then match the resulting value against the pattern $\langle\,\rangle$. Actually, we know by typing this will be the only possibility.

$$[\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\ d = x \leftarrow [\![e]\!]\ x\ ;\ \mathsf{case}\ x\ (\langle\,\rangle \Rightarrow [\![e']\!]\ d)$$

Note here how the process executing $[\![e]\!]\ x$ will write to a fresh destination c (substituted for x) and the case c destructor will read the value of c from memory when it becomes available. We then continue with the evaluation of e' to fill the original destination d.

$$\mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ P$$

We see here that we need to replicate the cell c that we read on the right-hand side of the rule because there may be other processes that may want to read c. Because this is a frequent pattern, we mark cells that have a value as *persistent* by writing !cell $c\ W$. It means this object, once created, persists from then on. In particular, if it occurs on the left-hand side of a transition rule it is *not* removed from the configuration. We now rewrite our rules with this notation:

$$\mathsf{proc}\ d\ (d.\langle\,\rangle) \mapsto \mathsf{!cell}\ d\ \langle\,\rangle$$
$$\mathsf{!cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ P$$

The typing rule for this case construct is straightforward.

$$\frac{c : 1 \in \Gamma \quad \Gamma \vdash P :: (d : \tau)}{\Gamma \vdash \mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P) :: (d : \tau)}\ 1L$$

We name these rules $1R$ (the type 1 occurring in the succedent) and $1L$ (the type 1 occurring among the antecedents) according to the traditions of the sequent calculus.

7 Eager Pairs

Eager pairs are another positive type and therefore quite analogous to the unit type. To evaluate an eager pair $\langle e_1, e_2\rangle$ we have to evaluate e_1 and e_2 and then form the pair of their values. The corresponding process $[\![\langle e_1, e_2\rangle]\!]\ d$ allocates two new destinations, d_1 and d_2 and launches two new processes, one to compute and write the value of e_1 to d_1 and the other to write the value of e_2 to d_2. Without waiting for these two finish, we already can form the pair $\langle d_1, d_2\rangle$ and write it to the original destination d.

$$\begin{aligned}[\![\langle e_1, e_2\rangle]\!]\, d = {} & x_1 \leftarrow [\![e_1]\!]\, d_1 \ ; \\ & x_2 \leftarrow [\![e_2]\!]\, d_2 \ ; \\ & d.\langle x_1, x_2\rangle\end{aligned}$$

There is a lot of parallelism in this translation: not only can the translations of e_1 and e_2 can proceed in parallel (without possibility of interference), but any process waiting for a value in the cell d will be able to proceed immediately, before either of these two finish. In the previously introduced parallel pairs[1] the synchronization point is earlier, namely when the pair of the values of e_1 and e_2 is formed.

$$\begin{aligned}[\![\mathsf{case}\ e\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\, d = {} & x \leftarrow [\![e]\!]\, x \ ; \\ & \mathsf{case}\ x\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\, d)\end{aligned}$$

In the rule just above we note that the occurrences of x_1 and x_2 in e' will be translated using the rule for variables.

The new process construct $d.\langle c_1, c_2\rangle$ simply writes the pair $\langle c_1, c_2\rangle$ to destination d and case reads the pair from memory and matches it against the pattern $\langle x_1, x_2\rangle$.

$$\begin{array}{l}\mathsf{proc}\ c\ (c.\langle c_1, c_2\rangle) \mapsto\ !\mathsf{cell}\ c\ \langle c_1, c_2\rangle \\ !\mathsf{cell}\ c\ \langle c_1, c_2\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle x_1, x_2\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ ([c_1/x_1, c_2/x_2]P)\end{array}$$

Typing rules generalize the unit types in interesting ways. We start with $d.\langle d_1, d_2\rangle$. This writes to d, which must therefore have type $\tau_1 \times \tau_2$. It must be able to read destinations d_1 and d_2 which must have types τ_1 and τ_2, respectively.

$$\frac{c_1 : \tau_1 \in \Gamma \quad c_2 : \tau_2 \in \Gamma}{\Gamma \vdash d.\langle c_1, c_2\rangle :: (d : \tau_1 \times \tau_2)}\ \times R^0$$

We use the superscript 0 because this is a nonstandard rule—the usual rule of the sequent calculus has 2 premises, while this rule only checks membership in the typing context. Note that c_1 and c_2 could be equal if $\tau_1 = \tau_2$.

The rule for the new case construct mirrors the usual rule for expressions, but using destinations.

$$\frac{c : \tau_1 \times \tau_2 \in \Gamma \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash P :: (d : \sigma)}{\Gamma \vdash \mathsf{case}\ c\ (\langle x_1, x_2\rangle \Rightarrow P) :: (d : \sigma)}\ \times L$$

[1] in the midterm exam

We close this section with the corresponding logical rules.

$$\frac{}{\Gamma \vdash 1}\ 1R^0 \qquad \frac{1 \in \Gamma \quad \Gamma \vdash C}{\Gamma \vdash C}\ 1L$$

$$\frac{A, B \in \Gamma}{\Gamma \vdash A \times B}\ \times R^0 \qquad \frac{A \times B \in \Gamma \quad \Gamma, A, B \vdash C}{\Gamma \vdash C}\ \times L$$

All the types considered in this lecture are *positive types*, so they are "eager" in the sense that a value only contains other values and that the destructors are case constructs.

8 Summary

Since we have changed our notation a few times, we summarize the translation and the transition rules.

$[\![x]\!]\, d = d \leftarrow x$

$[\![\langle\,\rangle]\!]\, d = d.\langle\,\rangle$
$[\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\, d = d_1 \leftarrow [\![e]\!]\, d_1\ ;$
$\qquad\qquad \mathsf{case}\ d_1\ (\langle\,\rangle \Rightarrow [\![e']\!]\, d)$

$[\![\langle e_1, e_2\rangle]\!]\, d = d_1 \leftarrow [\![e_1]\!]\, d_1\ ;$
$\qquad\qquad d_2 \leftarrow [\![e_2]\!]\, d_2\ ;$
$\qquad\qquad d.\langle d_1, d_2\rangle$
$[\![\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\, d = d_0 \leftarrow [\![e_0]\!]\, d_0\ ;$
$\qquad\qquad\qquad\qquad \mathsf{case}\ d_0\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\, d)$

$\mathsf{proc}\ d'\ (x \leftarrow P\ ;\ Q) \mapsto \mathsf{proc}\ d\ ([d/x]P), \mathsf{proc}\ d'\ ([d/x]Q) \quad (d\ \text{fresh})$ (alloc/spawn)

$!\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (d \leftarrow c), \mapsto \mathsf{cell}\ d\ W$ (copy)

$\mathsf{proc}\ d\ (d.\langle\,\rangle), \mapsto\ !\mathsf{cell}\ d\ \langle\,\rangle$ $(1R^0)$
$!\mathsf{cell}\ c\ \langle\,\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle\,\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ P$ $(1L)$

$\mathsf{proc}\ d\ (d.\langle c_1, c_2\rangle), \mapsto\ !\mathsf{cell}\ d\ \langle c_1, c_2\rangle$ $(\times R^0)$
$!\mathsf{cell}\ c\ \langle c_1, c_2\rangle, \mathsf{proc}\ d\ (\mathsf{case}\ c\ (\langle x_1, x_2\rangle \Rightarrow P)) \mapsto \mathsf{proc}\ d\ ([c_1/x_1, c_2/x_2]P)$ $(\times L)$

9 Streamlining the Positive Types

In the presentation of this lecture we notice commonality between the cases and we can refactor it so all positive (eager) types are treated uniformly. We define (omitting $\exists\alpha.\,\tau$ for simplicity):

$$
\begin{array}{llll}
\text{Positive types} & \tau & ::= & 1 \mid \tau_1 \times \tau_2 \mid \sum_{i\in I}(i:\tau_i) \mid \mu\alpha.\,\tau \\
\text{Small values} & V & ::= & \langle\,\rangle \mid \langle a_1, a_2\rangle \mid i \cdot a \mid \mathsf{fold}\ a \\
\text{Continuations} & K & ::= & (\langle\,\rangle \Rightarrow P) \mid (\langle x_1, x_2\rangle \Rightarrow P) \mid (i \cdot x_i \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}\ x \Rightarrow P) \\
\text{Processes} & P & ::= & x \leftarrow P\ ;\ Q \qquad (allocate/spawn) \\
 & & \mid & c \leftarrow d \qquad (copy) \\
 & & \mid & d.V \qquad (write) \\
 & & \mid & \mathsf{case}\ c\ K \qquad (read/match) \\
\text{Configurations} & \mathcal{C} & ::= & \mathsf{proc}\ d\ P \mid \mathsf{!cell}\ c\ V \mid \cdot \mid \mathcal{C}_1, \mathcal{C}_2
\end{array}
$$

We only have four transition rules for configurations, in addition to explaining how values are matched against continuations.

$$
\begin{array}{rlcl}
 & \mathsf{proc}\ d\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ c\ ([c/x]P), \mathsf{proc}\ d\ ([c/x]Q) \\
\mathsf{!cell}\ c\ V, & \mathsf{proc}\ d\ (d \leftarrow c) & \mapsto & \mathsf{!cell}\ d\ V \\
 & \mathsf{proc}\ d\ (d.V) & \mapsto & \mathsf{!cell}\ d\ V \\
\mathsf{!cell}\ c\ V, & \mathsf{proc}\ d\ (\mathsf{case}\ c\ K) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K)
\end{array}
$$

$$
\begin{array}{rcll}
\langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = P \\
\langle c_1, c_2\rangle & \triangleright & (\langle x_1, x_2\rangle \Rightarrow P) & = [c_1/x_1, c_2/x_2]P \\
k \cdot c & \triangleright & (i \cdot x_i \Rightarrow P_i)_{i\in I} & = [c/x_k]P_k \\
\mathsf{fold}\ c & \triangleright & (\mathsf{fold}\ x \Rightarrow P) & = [c/x]P
\end{array}
$$

10 Example: Writing a Value

A closed value in the our language of expressions is translated to a *program* that will create a representation of this value in memory. As such, memory and the contents of the its cells is observable since it represents the outcome of the computation. As an example, consider

$bin = \mu bin.\,(\mathbf{b0} : bin) + (\mathbf{b1} : bin) + (\mathbf{e} : bin)$
$one = \mathsf{fold}\ \mathbf{b1} \cdot \mathsf{fold}\ \mathbf{e} \cdot \langle\,\rangle$

We work out the translation of $[\![one]\!]$ in stages.

$$
\begin{aligned}
[\![\mathit{one}]\!]\, c_0 &= x_1 \leftarrow [\![\mathbf{b1} \cdot \mathsf{fold}\ \mathbf{e} \cdot \langle\,\rangle]\!]\, x_1\ ; \\
&\qquad x_0.(\mathsf{fold}\ b1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow [\![\mathsf{fold}\ \mathbf{e} \cdot \langle\,\rangle]\!]\, x_2\ ; \\
&\qquad\qquad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\qquad c_0.(\mathsf{fold}\ x_1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow (\ x_3 \leftarrow [\![\mathbf{e} \cdot \langle\,\rangle]\!]\, x_3\ ; \\
&\qquad\qquad\qquad x_2.(\mathsf{fold}\ x_3)\ )\ ; \\
&\qquad\qquad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\qquad c_0.(\mathsf{fold}\ x_1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow (\ x_3 \leftarrow (\ x_4 \leftarrow [\![\langle\,\rangle]\!]\, x_4\ ; \\
&\qquad\qquad\qquad\qquad x_3.(\mathbf{e} \cdot x_4)\ )\ ; \\
&\qquad\qquad\qquad x_2.(\mathsf{fold}\ x_3)\ )\ ; \\
&\qquad\qquad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\qquad c_0.(\mathsf{fold}\ x_1) \\
&= x_1 \leftarrow (\ x_2 \leftarrow (\ x_3 \leftarrow (\ x_4 \leftarrow x_4.\langle\,\rangle\ ; \\
&\qquad\qquad\qquad\qquad x_3.(\mathbf{e} \cdot x_4)\ )\ ; \\
&\qquad\qquad\qquad x_2.(\mathsf{fold}\ x_3)\ )\ ; \\
&\qquad\qquad x_1.(\mathbf{b_1} \cdot x_2)\ )\ ; \\
&\qquad c_0.(\mathsf{fold}\ x_1)
\end{aligned}
$$

This program will allocate four fresh cells, say, $c_1, \ldots, c_4$, for $x_1, \ldots, x_4$ and fill them with the indicated small values, in no particular order. The resulting final configuration will be

$$
\begin{aligned}
&\mathsf{proc}\ c_0\ ([\![\mathit{one}]\!]\, c_0) \\
&\mapsto^*\ !\mathsf{cell}\ c_4\ \langle\,\rangle, !\mathsf{cell}\ c_3\ (\mathbf{e} \cdot c_4), !\mathsf{cell}\ c_2\ (\mathsf{fold} \cdot c_3), !\mathsf{cell}\ c_1\ (\mathbf{b1} \cdot\ c_2), !\mathsf{cell}\ c_0\ (\mathsf{fold} \cdot c_1)
\end{aligned}
$$

The following law of *associativity* (not justified here, because we do not have a simple theory of process equivalence) allows us to rewrite this process into a more readable form. In order to apply the equivalence, some restrictions need to be placed on variable occurrences, so we indicate permissible references to variables for each process in parentheses.

$$x \leftarrow (y \leftarrow P(y)\ ;\ Q(x, y))\ ;\ R(x) \equiv y \leftarrow P(y)\ ;\ (x \leftarrow Q(x, y)\ ;\ R(x))$$

Applying this multiple times to re-associate the cuts to the left we get

$$
\begin{aligned}
[\![\mathit{one}]\!]\, c_0 &= x_4 \leftarrow x_4.\langle\,\rangle\ ; \\
&\qquad x_3 \leftarrow x_3.(\mathbf{e} \cdot x_4)\ ; \\
&\qquad x_2 \leftarrow x_2.(\mathsf{fold}\ x_3)\ ; \\
&\qquad x_1 \leftarrow x_1.(\mathbf{b_1} \cdot x_2)\ ; \\
&\qquad c_0.(\mathsf{fold}\ x_1)
\end{aligned}
$$

11 Example: Reversing a Pair

Another example that also involves reading from memory is a small program

$$p : \tau \times \sigma \vdash \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow \langle y, x\rangle) : \sigma \times \tau$$

Let's translate the expression to a process:

$$\begin{aligned}
[\![\mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow \langle y, x\rangle)]\!]\, d_0 &= d_1 \leftarrow [\![p]\!]\, d_1\ ; \\
&\quad\ \mathsf{case}\ d_1\ (\langle x, y\rangle \Rightarrow [\![\langle y, x\rangle]\!]\, d_0) \\
&= d_1 \leftarrow (d_1 \leftarrow p)\ ; \\
&\quad\ \mathsf{case}\ d_1\ (\langle x, y\rangle \Rightarrow d_2 \leftarrow [\![y]\!]\, d_2\ ; \\
&\qquad\qquad\qquad\ d_3 \leftarrow [\![x]\!]\, d_3\ ; \\
&\qquad\qquad\qquad\ d_0.\langle d_2, d_3\rangle) \\
&= d_1 \leftarrow (d_1 \leftarrow p)\ ; \\
&\quad\ \mathsf{case}\ d_1\ (\langle x, y\rangle \Rightarrow d_2 \leftarrow (d_2 \leftarrow y)\ ; \\
&\qquad\qquad\qquad\ d_3 \leftarrow (d_3 \leftarrow x)\ ; \\
&\qquad\qquad\qquad\ d_0.\langle d_2, d_3\rangle)
\end{aligned}$$

At this point we have completed the translation and we can notice some redundancy: this code will allocate three new cells (d_1, d_2, and d_3) and copy the contents of p, y, and x into them. We can consider optimizing the pattern

$$x \leftarrow (x \leftarrow c)\ ;\ P \quad \rightsquigarrow \quad [c/x]P$$

We would then obtain

$$[\![\mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow \langle y, x\rangle)]\!]\, d_0 \quad \rightsquigarrow \quad \mathsf{case}\ p\ (\langle x, y\rangle \Rightarrow d_0.\langle y, x\rangle)$$

which is a simple and intuitive translation of the original expression, given that the destination of it operation is d_0: we read the addresses of the components from the cell p and write them into d_0 in the opposite order.

How can we justify this optimization? We can trace the execution of both sides and compare the results.

$\mathsf{proc}\ d\ (x \leftarrow (x \leftarrow c)\ ;\ P)$
$\mapsto \mathsf{proc}\ a\ (a \leftarrow c), \mathsf{proc}\ d\ [a/x]P$

At this point we are a stuck become $\mathsf{proc}\ a\ (a \leftarrow c)$ cannot proceed until the cell c has been written. If it has, we obtain

$!\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (x \leftarrow (x \leftarrow c)\ ;\ P)$
$\mapsto\ !\mathsf{cell}\ c\ W, \mathsf{proc}\ a\ (a \leftarrow c), \mathsf{proc}\ d\ [a/x]P$
$\mapsto\ !\mathsf{cell}\ c\ W, !\mathsf{cell}\ a\ W, \mathsf{proc}\ d\ [a/x]P$

versus the right-hand side

!cell $c\ W$, proc $d\ [c/x]P$

Intuitively, these two should be equivalent: P cannot depend on the actual address of a or c, only on what the cell contains, which is W in both cases. This is a simple form of a parametricity result: for any relation R between the names that respects cell contents in a configuration, processes related by R will produce related final configurations. For this to apply we would also have to somehow know that it is okay to assume that the cell c has already been written to. This looks like a form of bisimulation between the computation of the two configurations would be required. We do not develop such a theory here further.

12 Preservation and Progress

We postpone a discussion of preservation and progress until the next lecture, when we develop a more complete picture of how to type configuration.

References

[CPWW02] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[CS09] Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Information and Computation*, 207(10):1044–1077, October 2009.

[DPP20] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 29:1–29:22, Paris, France, June 2020. LIPIcs 167.

[Pfe04] Frank Pfenning. Substructural operational semantics and linear destination-passing style. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems*

(APLAS'04), page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302. Abstract of invited talk.

[PP20] Klaas Pruiksma and Frank Pfenning. Back to futures. *CoRR*, abs/2002.04607, February 2020.

[PS09] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proceedings of the 24th Annual Symposium on Logic in Computer Science (LICS 2009)*, pages 101–110, Los Angeles, California, August 2009. IEEE Computer Society Press.