Lecture Notes on
Shared Memory Concurrency

15-814: Types and Programming Languages
Frank Pfenning

Lecture 19
Thursday, November 7, 2019

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 [PP19].

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By the time we actually evaluate e, all the variables declared in Γ will have been replaced by values v (values, because we are in a call-by-value language, with variables for fixed point expressions representing an exception to that rule).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

$$\mathit{list}\ \alpha \cong (\texttt{Nil} : 1) + (\texttt{Cons} : \alpha \times \mathit{list}\ \alpha)$$

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

$$\mathsf{fold}\ (\texttt{Cons} \cdot \langle v_1, \mathsf{fold}\ (\texttt{Cons} \cdot \langle v_2, \mathsf{fold}\ (\texttt{Nil} \cdot \langle\,\rangle)\rangle)\rangle) : \mathit{list}\ \tau$$

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

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

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. As someone pointed out during lecture, it is not clear whether d itself might also be redundant, but we keep it for now. While P executes, the cell d does not have a value yet, since this is P's job to produce. We write such uninitialized cells as $\mathsf{cell}\ d\ \_$, and they always occur together with a process computing with destination d.

$$\mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_$$

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 d is created, P is spawned with destination d, and Q can read from d. We formalize this as

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

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 d 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 the two occurrences of the channel x whose type must match.

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

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 d precedes all the processes that may read from d or other cells that contain d. Because we do not have arbitrary mutation of store there cannot be any cycles (although we have to carefully consider 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.

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

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 $d.\langle\,\rangle$.

$$\begin{array}{l} [\![\langle\,\rangle]\!]\,d = d.\langle\,\rangle \\ \mathsf{proc}\ d\ (d.\langle\,\rangle), \mathsf{cell}\ d\ \_ \mapsto \mathsf{cell}\ d\ \langle\,\rangle \end{array}$$

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

The way we evaluate $\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')$ 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.

$$\begin{array}{rl} [\![\mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e')]\!]\,d = & d_1 \leftarrow [\![e]\!]\,d_1\ ; \\ & \mathsf{case}\ d_1\ (\langle\,\rangle \Rightarrow [\![e']\!]\,d) \end{array}$$

Note here how the process executing $[\![e]\!]\,d_1$ will write to the fresh destination d_1 and the $\mathsf{case}\ d_1$ destructor will read the value of d_1 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 $!\mathsf{cell}\ c\ W$. 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:

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

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 $[\![\langle e_1, e_2\rangle]\!]$ 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.

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

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 on the midterm the synchronization point is earlier, namely when the pair of the values of e_1 and e_2 is formed.

$$\begin{aligned}[\![\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e')]\!]\, d = {} & d_0 \leftarrow [\![e_0]\!]\, d_0 \ ; \\ & \mathsf{case}\ d_0\ (\langle x_1, x_2\rangle \Rightarrow [\![e']\!]\, d)\end{aligned}$$

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 $d.\langle d_1, d_2\rangle$ simply writes the pair $\langle d_1, d_2\rangle$ to destination d and case reads the pair from memory and matches it against the pattern $\langle x_1, x_2\rangle$.

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

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.

$$\frac{d_1 : \tau_1 \in \Gamma \quad d_2 : \tau_2 \in \Gamma}{\Gamma \vdash d.\langle d_1, d_2\rangle :: (d : \tau_1 \times \tau_2)}\ \times R^0$$

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.

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

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

We close this section with the corresponding logical rules. If we write Γ, A in the conclusion of a rule we mean that A may or may not also remain in Γ.

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

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

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.

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

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

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

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

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

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

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

9 Further Remarks and Notes

A few notes on discussions we had in class.

Garbage Collection. If there is an initial root destination d_0 one can run a garbage collector over the configuration, including processes as well as cells. This is somewhat simpler than for many other languages because in the pure fragment we have presented so far, memory will be in the shape of a tree and memory is statically typed. This will have to be reconsidered in the presence of general recursion and also when mutable references are introduced in the future.

Concurrency. The new semantics is no longer equivalent to the original one. In particular, it obviously no longer satisfies small-step determinacy. It is also unclear what might happen when in an expression such as

$$\mathsf{case}\ \langle e_1, e_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow x_1)$$

the process implementing e_2 does not terminate. One option is to simply garbage collect this process.

Call-by-Value and Call-by-Need. We have carefully written the rules so that a certain scheduling strategy will recover our previous call-by-value semantics. The idea is for an expression $x \leftarrow P\ ;\ Q$ to compute P until it writes to x and terminates, and only then continue with Q.

Call-by-need (which we haven't discussed yet in the course, but is used in Haskell) would instead postpone the evaluation of P entirely until Q tries to access the value of x. At this point we go back and run P until it has written this value and then resume Q. Note that the next time x is referenced the cell x will hold a value so P is not re-executed, which is a core idea behind call-by-need.

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.

[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.

[PP19] Klaas Pruiksma and Frank Pfenning. Back to futures. Under submission, October 2019.

[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.