Lecture Notes on Memory Safety

15-814: Types and Programming Languages
Frank Pfenning

Lecture 21
Thursday, November 14, 2019

1 Introduction

In the previous lecture we have presented a translation from expressions in our sequential functional language to processes. The processes access shared memory, but in a highly disciplined manner: each process has the responsibility to *write* to a particular location, while multiple processes may *read* from that location. Each location is only written to once, so the reader must check that the cell has indeed been initialized. This is a form of synchronization between concurrent processes. When processes are scheduled to run in a stack-like fashion, then we recover the usual sequential semantics. Every location will be written before any other process can try to access it.

Why is all of this safe? We have an obligation to prove properties of our concurrent operational model that are analogous to progress and preservation (together called *type soundness*).

2 Refactoring the Statics and Dynamics

There are many patterns and symmetries in the our process language we now exploit to simplify the presentation.

$$
\begin{array}{llllll}
\text{Processes} & P & ::= & x \leftarrow P \,;\, Q & & \text{allocate/spawn} \\
 & & \mid & x^W \leftarrow y^R & & \text{copy} \\
 & & \mid & x^W.\langle\,\rangle & \mid \mathsf{case}\ x^R\ (\langle\,\rangle \Rightarrow P) & (1) \\
 & & \mid & x^W.\langle y, z\rangle & \mid \mathsf{case}\ x^R\ (\langle y, z\rangle \Rightarrow P) & (\times) \\
 & & \mid & x^W.j(y) & \mid \mathsf{case}\ x^R\ (i(y) \Rightarrow P_i)_{i\in I} & (+) \\
 & & \mid & x^W.\mathsf{fold}(y) & \mid \mathsf{case}\ x^R\ (\mathsf{fold}(y) \Rightarrow P) & (\rho) \\
\hline
 & & \mid & x^R.\langle y, z\rangle & \mid \mathsf{case}\ x^W\ (\langle y, z\rangle \Rightarrow P) & (\rightarrow) \\
\\
\text{Cell Contents} & W & ::= & \langle\,\rangle \mid \langle d_1, d_2\rangle \mid j(d) \mid \mathsf{fold}(d) & & \\
 & & \mid & (\langle x, y\rangle \Rightarrow P) & & \\
\\
\text{Configurations} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}_1, \mathcal{C}_2 \mid \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ \mid \mathsf{!cell}\ c\ W & &
\end{array}
$$

This table is partially incomplete, for example, omitting lazy records $\&_{i\in I}(i : \tau_i)$ which is a generalization of lazy pairs and dual to (eager) sums (see Exercises L20.2–4).

We now refactor the syntax by dividing the cell contents into small values V and continuations K (different from the continuations k used to define the K machine).

$$
\begin{array}{lllll}
\text{Values} & V & ::= & \langle\,\rangle \mid \langle y, z\rangle \mid j(y) \mid \mathsf{fold}(y) & \\
\text{Continuations} & K & ::= & (\langle\,\rangle \Rightarrow P) \mid (\langle y, z\rangle \Rightarrow P) \mid (i(y) \Rightarrow P_i)_{i\in I} \mid (\mathsf{fold}(y) \Rightarrow P) & \\
\text{Processes} & P & ::= & x \leftarrow P \,;\, Q & \text{allocate/spawn} \\
 & & \mid & x^W \leftarrow y^R & \text{copy} \\
 & & \mid & x^W.V \mid \mathsf{case}\ x^R\ K & \text{positive types} \\
 & & \mid & x^R.V \mid \mathsf{case}\ x^W\ K & \text{negative types} \\
\text{Cell Contents} & W & ::= & V \mid K & \\
\text{Configurations} & \mathcal{C} & ::= & \cdot \mid \mathcal{C}, \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ \mid \mathcal{C}, \mathsf{!cell}\ c\ W &
\end{array}
$$

This syntax allows some forms (e.g., lazy folds) for which we do not have corresponding types, although we could (see Exercises L20.2–4).

We have presented configurations in a slightly different style as a list, which is equivalent to the previous form since we treat the comma operator as commutative and associative.

To type the contents of cells directly, we have two judgments $\Gamma \vdash V : \tau$ and $\Gamma \vdash K : \tau$.

$$\frac{}{\Gamma \vdash \langle\,\rangle : 1}\ \mathsf{val/unit} \qquad \frac{y : \tau \in \Gamma \qquad z : \sigma \in \Gamma}{\Gamma \vdash \langle y, z\rangle : \tau \times \sigma}\ \mathsf{val/prod} \qquad \frac{y : \tau_j \in \Gamma}{\Gamma \vdash j(y) : \sum_{i\in I}(i : \tau_i)}\ \mathsf{val/sum}$$

$$\frac{y : [\rho\alpha.\,\tau/\alpha]\tau \in \Gamma}{\Gamma \vdash \mathsf{fold}(y) : \rho\alpha.\,\tau}\ \mathsf{val/fold} \qquad \frac{\Gamma, y : \tau \vdash P :: (z : \sigma)}{\Gamma \vdash (\langle y, z\rangle \Rightarrow P) : \tau \to \sigma}\ \mathsf{cont/fun}$$

Process typing for the positives is now unified, but we still separate out the negative with some special-purpose rules.

$$\frac{\Gamma \vdash V : \tau}{\Gamma \vdash x^W.V :: (x : \tau)}\ \mathsf{pos/write} \qquad \frac{x : \tau \in \Gamma \qquad \Gamma \vdash \tau \triangleright K :: (z : \sigma)}{\Gamma \vdash \mathsf{case}\ x^R\ K :: (z : \sigma)}\ \mathsf{pos/read}$$

$$\frac{x : \tau \to \sigma \in \Gamma \quad y : \tau \in \Gamma}{\Gamma \vdash x^R.\langle y, z\rangle :: (z : \sigma)}\ \mathsf{neg/read} \qquad \frac{\Gamma \vdash K : \sigma}{\Gamma \vdash \mathsf{case}\ x^W\ K :: (x : \sigma)}\ \mathsf{neg/write}$$

The neg/read rule is somewhat overly specific, but because our only negative type is $\tau \to \sigma$ we did not generalize it further.

For positive types τ we also have a judgment to verify that a value $V : \tau$ is matched against a suitable continuation, $\Gamma \vdash \tau \triangleright K :: (z : \sigma)$.

$$\frac{\Gamma \vdash P :: (z : \sigma)}{\Gamma \vdash 1 \triangleright (\langle\,\rangle \Rightarrow P) :: (z : \sigma)} \qquad \frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash P :: (z : \sigma)}{\Gamma \vdash \tau_1 \times \tau_2 \triangleright (\langle x_1, x_2\rangle \Rightarrow P) : (z : \sigma)}$$

$$\frac{(\text{for all } i \in I) \quad \Gamma, y : \tau_i \vdash P_i :: (z : \sigma)}{\Gamma \vdash \sum_{i\in I}(i : \tau_i) \triangleright (i(y) \Rightarrow P_i) :: (z : \sigma)} \qquad \frac{\Gamma, y : [\rho\alpha.\,\tau/\alpha]\tau \vdash P :: (z : \sigma)}{\Gamma \vdash \rho\alpha.\,\tau \triangleright (\mathsf{fold}(y) \Rightarrow P) :: (z : \sigma)}$$

We could refactor the rules for the positive types further by using the idea behind pattern matching as introduced in Lecture 10.

A key operation in the operational semantics is passing a value to a continuation, written as $V \triangleright K$. It is defined as follows:

$$\begin{array}{llll}
\langle\,\rangle & \triangleright & (\langle\,\rangle \Rightarrow P) & = & P \\
\langle d_1, d_2\rangle & \triangleright & (\langle x_1, x_2\rangle \Rightarrow P) & = & [d_1/x_1, d_2/x_2]P \\
j(d) & \triangleright & (i(y) \Rightarrow P_i)_{i\in I} & = & [d/y]P_j \\
\mathsf{fold}(d) & \triangleright & (\mathsf{fold}(y) \Rightarrow P) & = & [d/y]P
\end{array}$$

The operational rules can be reduced to the following:

$$
\begin{array}{lcll}
\mathsf{proc}\ d\ (x \leftarrow P\ ;\ Q) & \mapsto & \mathsf{proc}\ c\ ([c/x]P), \mathsf{cell}\ c\ \_, \mathsf{proc}\ d\ ([c/x]Q) & \\
 & & & \text{(alloc/spawn)} \\
!\mathsf{cell}\ c\ W, \mathsf{proc}\ d\ (d^W \leftarrow c^R), \mathsf{cell}\ d\ \_ & \mapsto & !\mathsf{cell}\ d\ W & \text{(copy)} \\
\mathsf{proc}\ d\ (d^W.V), \mathsf{cell}\ d\ \_ & \mapsto & !\mathsf{cell}\ d\ V & (1, \times, +, \rho)R^0 \\
!\mathsf{cell}\ c\ V, \mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K) & (1, \times, +, \rho)L \\
\mathsf{proc}\ d\ (\mathsf{case}\ d^W\ K), \mathsf{cell}\ d\ \_ & \mapsto & !\mathsf{cell}\ d\ K & (\rightarrow)R \\
!\mathsf{cell}\ c\ K, \mathsf{proc}\ d\ (c^R.V) & \mapsto & \mathsf{proc}\ d\ (V \triangleright K) & (\rightarrow)L^0
\end{array}
$$

3 Typing Configurations

It is relatively straightforward to preservation in analogy with the usual preservation theorem, so we focus here on *progress*. Intuitively, progress comes down to two factors. The first is the canonical form theorem. In this context it guarantees, for example, that a cell $!\mathsf{cell}\ c\ W$ with $c : \tau_1 \times \tau_2$ must contain a pair $W = \langle c_1, c_2 \rangle$ (where $c_1 : \tau_1$ and $c_2 : \tau_2$). The second factor is that there no cyclic dependencies among the process that would form a deadlock. The simplest case of that would be where process P with destination d waits for cell c to be filled by Q which in turns waits for d, but there could be multiple processes in a cycle. Such a deadlock might look like

$$
\begin{array}{l}
\mathsf{proc}\ d\ (\mathsf{case}\ c^R\ K), \mathsf{cell}\ d\ \_, \\
\mathsf{proc}\ c\ (\mathsf{case}\ d^R\ K'), \mathsf{cell}\ c\ \_
\end{array}
$$

where $P = (\mathsf{case}\ c^R\ K)$ and $Q = (\mathsf{case}\ d^R\ K')$.

It turns out that if we start with a single well-typed initial process any configuration we might reach can progress and is therefore deadlock-free (no cyclic dependencies). This is ensured by an intrinsic ordering among the destinations which is acyclic, defined as follows:

> $c < d$ if either c occurs in a process that has destination d
> or c occurs in the contents of the cell d.

In our typing rules this dependency relation is emergent rather than postulated a priori. In other words, when we can type a configuration we can read off this relation from the typing derivation.

Our typing judgment has the form $\vdash \mathcal{C} :: \Gamma$ which means that for each address $c : \tau$ in Γ, $\mathcal{C}$ either has a cell $!\mathsf{cell}\ c\ W$, or $\mathsf{proc}\ c\ P, \mathsf{cell}\ c\ \_$. We define it with the following rules, referring back to the typing of processes, values, and continuations.

$$\frac{}{\vdash (\cdot) :: (\cdot)} \qquad \frac{\vdash \mathcal{C} :: \Gamma \quad \Gamma \vdash P :: (c : \tau)}{\vdash (\mathcal{C}, \mathsf{proc}\ c\ P, \mathsf{cell}\ c\ \_) :: (\Gamma, c : \tau)} \qquad \frac{\vdash \mathcal{C} :: \Gamma \quad \Gamma \vdash W : \tau}{\vdash (\mathcal{C}, !\mathsf{cell}\ c\ W) :: (\Gamma, c : \tau)}$$

In the second and third rules, the context $\Gamma, c : \tau$ must be well-formed, so c must be fresh. This judgment never has an antecedent, since we only type complete configurations to get the simplest progress and preservation theorems.

4 Preservation and Progress

We only state preservation here, since its proof is somewhat tedious but not essentially different from prior proofs.

Theorem 1 (Preservation) *If* $\vdash \mathcal{C} :: \Gamma$ *and* $\mathcal{C} \mapsto \mathcal{C}'$ *then* $\vdash \mathcal{C}' :: \Gamma'$ *for* $\Gamma' \supseteq \Gamma$.

The reason we have to allow for Γ' to be a superset of Γ is that the allocate/spawn step may allocate a fresh cell which then becomes visible not only to its immediate client, but also shows up in the type of the whole configuration.

To state the progress theorem we need to characterize *final states* that correspond to values in a functional language. Fortunately, that's easy: a state is final exactly if it consists only of memory cells of the form $!\mathsf{cell}\ c\ W$! We usually start with a "main" process P with $\cdot \vdash P :: (d_0 : \tau)$, encapsulated in the initial configuration as $\mathcal{C}_0 = (\mathsf{proc}\ d_0\ P, \mathsf{cell}\ d_0\ \_)$. Following the given typing rules, this means we have $\vdash \mathcal{C}_0 :: (d_0 : \tau)$.

Theorem 2 (Progress) *If* $\vdash \mathcal{C} :: \Gamma$ *then either* $\mathcal{C} \mapsto \mathcal{C}'$ *for esome* $\mathcal{C}'$ *or* $\mathcal{C}$ *is* final.

Proof: By induction on the structure of the given typing derivation.

Case:

$$\frac{}{\vdash (\cdot) :: (\cdot)}$$

where $\mathcal{C} = (\cdot)$ and $\Gamma = (\cdot)$. Then $\mathcal{C}$ *final*.

Case:

$$\frac{\vdash \mathcal{C}_1 :: \Gamma_1 \qquad \Gamma_1 \vdash W : \tau}{\vdash \mathcal{C}_1, \mathsf{!cell}\ c\ W :: (\Gamma_1, c : \tau)}$$

where $\mathcal{C} = (\mathcal{C}_1, \mathsf{!cell}\ c\ W)$ and $\Gamma = (\Gamma_1, c : \tau)$.

Either $\mathcal{C}_1 \mapsto \mathcal{C}_1'$ for some $\mathcal{C}_1'$ or $\mathcal{C}_1$ *final* — By i.h.
If $\mathcal{C}_1 \mapsto \mathcal{C}_1'$ then also $(\mathcal{C}_1, \mathsf{!cell}\ c\ W) \mapsto (\mathcal{C}_1', \mathsf{!cell}\ c\ W)$ — First subcase
If $\mathcal{C}_1$ *final* then also $(\mathcal{C}_1, \mathsf{!cell}\ c\ W)$ *final* — Second subcase

Case:

$$\frac{\vdash \mathcal{C}_1 :: \Gamma_1 \qquad \Gamma_1 \vdash P :: (d : \tau)}{\vdash \mathcal{C}_1, \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ :: (\Gamma_1, d : \tau)}$$

where $\mathcal{C} = (\mathcal{C}_1, \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_)$ and $\Gamma = (\Gamma_1, d : \tau)$.

Either $\mathcal{C}_1 \mapsto \mathcal{C}_1'$ for some $\mathcal{C}_1'$ or $\mathcal{C}_1$ *final* — By i.h.
If $\mathcal{C}_1 \mapsto \mathcal{C}_1'$ then also $(\mathcal{C}_1, \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_) \mapsto (\mathcal{C}_1', \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_)$
First subcase

Assume $\mathcal{C}_1$ *final* — Second subcase

Now we distinguish cases on the typing of P. If $P = d^W.V$ or $P = \mathsf{case}\ d^W\ K$ then $\mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_ \mapsto \mathsf{!cell}\ d\ W$ (for $W = V$ or $W = K$) and $\mathcal{C} \mapsto \mathcal{C}_1, \mathsf{!cell}\ d\ W$.

Subcase:

$$\frac{\Gamma_1 \vdash P_1 :: (x : \sigma) \quad \Gamma_1, x : \sigma \vdash P_2 :: (d : \tau)}{\Gamma_1 \vdash x \leftarrow P_1 \ ; P_2 :: (d : \tau)}$$

for some σ where $P = (x \leftarrow P_1 \ ; P_2)$. Then

$\mathcal{C} = (\mathcal{C}_1, \mathsf{proc}\ d\ (x \leftarrow P_1 \ ; P_2), \mathsf{cell}\ d\ \_$
$\mapsto (\mathcal{C}_1, \mathsf{proc}\ c\ ([c/x]P_1), \mathsf{cell}\ c\ \_, \mathsf{proc}\ d\ ([c/x]P_2), \mathsf{cell}\ d\ \_) = \mathcal{C}'$.

Subcase:

$$\frac{c : \tau \in \Gamma_1}{\Gamma_1 \vdash d \leftarrow c :: (d : \tau)}$$

where $P = d \leftarrow c$. Because $c : \tau \in \Gamma_1$ and $\mathcal{C}_1$ is final, there must be a $\mathsf{!cell}\ c\ W \in \mathcal{C}_1$. Therefore

$$\mathcal{C} = (\mathcal{C}_1, \mathsf{proc}\ d\ (d \leftarrow c), \mathsf{cell}\ d\ \_) \mapsto (\mathcal{C}_1, !\mathsf{cell}\ d\ W) = \mathcal{C}'$$

Subcase:

$$\frac{c : \sigma \in \Gamma_1 \quad \Gamma_1 \vdash \tau \triangleright K :: (d : \tau)}{\Gamma_1 \vdash \mathsf{case}\ c^R\ K :: (d : \tau)}$$

where $P = \mathsf{case}\ c^R\ K$. Because $c : \sigma \in \Gamma_1$ and $\mathcal{C}_1$ is final, there must be a $!\mathsf{cell}\ c\ V \in \mathcal{C}_1$ with $\Gamma_2 \vdash V : \sigma$ where $\Gamma_2 \subset \Gamma_1$. By weakening, also $\Gamma_1 \vdash V : \sigma$. By Lemma 3 (stated below) we know that $V \triangleright K$ is defined and $\mathcal{C} = (\mathcal{C}_1, \mathsf{proc}\ d\ P, \mathsf{cell}\ d\ \_) \mapsto (\mathcal{C}_1, \mathsf{proc}\ d\ (V \triangleright K), \mathsf{cell}\ d\ \_) = \mathcal{C}'$.

Subcase:

$$\frac{c : \sigma \to \tau \in \Gamma_1 \quad c_1 : \sigma \in \Gamma_1}{\Gamma_1 \vdash c^R.\langle c_1, c_2 \rangle :: (c_2 : \tau)}$$

where $P = c^R.\langle c_1, c_2 \rangle$ and $d = c_2$. Then we reason by inversion in a similar way to the previous case, identifying a cell $!\mathsf{cell}\ c\ K$ that must contain a K of the right form so that $\langle c_1, c_2 \rangle \triangleright K$ is defined.

□

Lemma 3 *If* $\Gamma \vdash V : \sigma$ *and* $\Gamma \vdash \sigma \triangleright K :: (d : \tau)$ *then* $V \triangleright K = Q$ *for some* Q.

Proof: By inversion on the given typing derivations. □

In lecture we also began a discussion of mutable store, the notes on which we added to the beginning of the notes for the next lecture.