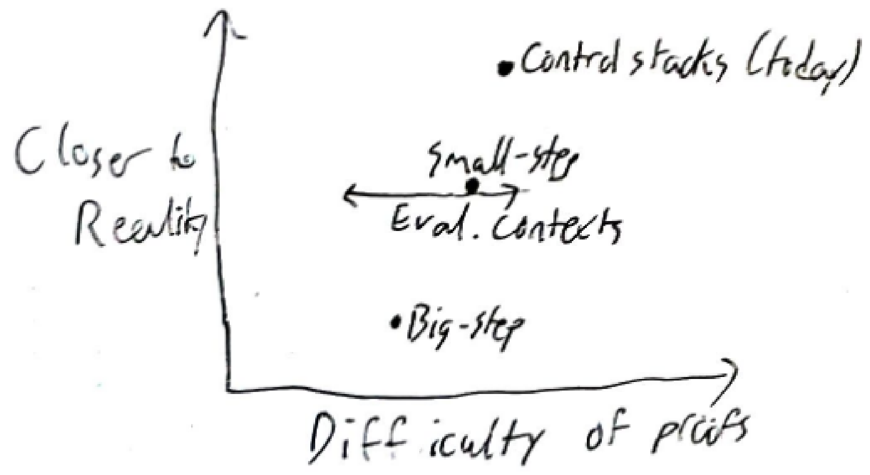
Control Stacks

Stefan Muller

CSE 5095: Types and Programming Languages, Fall 2025
Lecture 23

Today, we explore another point in the design space of dynamics semantics.



Control stacks are another way of avoiding search rules—this time, not for the purpose of sweeping them under the rug, but actually because the search rules themselves sweep some complexity under the rug. In particular, how does the computer "remember" what we still need to do with the result of evaluation? What looks like one step in a small-step semantics actually has added cost to it.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{fst}\ (((\overline{1},\overline{2}),\overline{3}),\overline{4}) \mapsto ((\overline{1},\overline{2}),\overline{3})}}{\mathsf{fst}\ \mathsf{fst}\ (((\overline{1},\overline{2}),\overline{3}),\overline{4}) \mapsto \mathsf{fst}\ ((\overline{1},\overline{2}),\overline{3})}}{\mathsf{fst}\ \mathsf{fst}\ \mathsf{fst}\ (((\overline{1},\overline{2}),\overline{3}),\overline{4}) \mapsto \mathsf{fst}\ \mathsf{fst}\ ((\overline{1},\overline{2}),\overline{3})}$$

In the computer (when a function f calls g which calls h), this kind of thing shows up in the call stack:

f
g
h

We can do the same thing with expressions that are awaiting results:

fst —
fst —
$\mathsf{fst}\ (((\overline{1},\overline{2}),\overline{3}),\overline{4})$

Types	τ	$::=$	$\mathsf{unit} \mid \tau \to \tau \mid \tau \times \tau$
Values	v	$::=$	$() \mid \lambda x : \tau.e \mid (v, v)$
Expressions	e	$::=$	$v \mid x \mid e\ e \mid (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$
Stack Frames	f	$::=$	$—\ e \mid v\ — \mid (—, e) \mid (v, —) \mid \mathsf{fst}\ — \mid \mathsf{snd}\ —$
Control Stacks	k	$::=$	$\epsilon \mid k; f$
States	s	$::=$	$k \triangleright e \mid k \triangleleft v$

The state $k \triangleright e$ represents evaluating e with the stack k. Eventually, it'll be a value v and we'll return it to the stack, represented $k \triangleleft v$. We can represent

$$\mathsf{fst}\ \mathsf{fst}\ \mathsf{fst}\ (((\overline{1}, \overline{2}), \overline{3}), \overline{4})$$

as

$$\mathsf{fst}\ —; \mathsf{fst}\ —; \mathsf{fst}\ — \triangleright (((\overline{1}, \overline{2}), \overline{3}), \overline{4})$$

$$\begin{array}{lcll}
k \triangleright v & \mapsto & k \triangleleft v & (1) \\
k \triangleright e_1\ e_2 & \mapsto & k; —\ e_2 \triangleright e_1 & (2) \\
k; —\ e_2 \triangleleft v & \mapsto & k; v\ — \triangleright e_2 & (3) \\
k; \lambda x : \tau.e_1\ — \triangleleft v & \mapsto & k \triangleright [v/x]e_1 & (4) \\
k \triangleright (e_1, e_2) & \mapsto & k; (—, e_2) \triangleright e_1 & (5) \\
k; (—, e_2) \triangleleft v_1 & \mapsto & k; (v_1, —) \triangleright e_2 & (6) \\
k; (v_1, —) \triangleleft v_2 & \mapsto & k \triangleleft (v_1, v_2) & (7) \\
k \triangleright \mathsf{fst}\ e & \mapsto & k; \mathsf{fst}\ — \triangleright e & (8) \\
k; \mathsf{fst}\ — \triangleleft (v_1, v_2) & \mapsto & k \triangleleft v_1 & (9) \\
k \triangleright \mathsf{snd}\ e & \mapsto & k; \mathsf{snd}\ — \triangleright e & (10) \\
k; \mathsf{snd}\ — \triangleleft (v_1, v_2) & \mapsto & k \triangleleft v_2 & (11)
\end{array}$$

1 Type Safety

The typing judgment for stack frames is $f : \tau_1 \rightsquigarrow \tau_2$ and says that f expects a τ_1 and returns a τ_2.

$$\frac{\bullet \vdash e : \tau_1}{—\ e : (\tau_1 \to \tau_2) \rightsquigarrow \tau_2}\ (\text{TypeFFun}) \qquad \frac{\bullet \vdash v : \tau_1 \to \tau_2}{v\ — : \tau_1 \rightsquigarrow \tau_2}\ (\text{TypeFArg})$$

$$\frac{\bullet \vdash e_2 : \tau_2}{(—, e_2) : \tau_1 \rightsquigarrow \tau_1 \times \tau_2}\ (\text{TypeFPairL}) \qquad \frac{\bullet \vdash e_1 : \tau_1}{(e_1, —) : \tau_2 \rightsquigarrow \tau_1 \times \tau_2}\ (\text{TypeFPairR})$$

$$\frac{}{\mathsf{fst}\ — : (\tau_1 \times \tau_2) \rightsquigarrow \tau_1}\ (\text{TypeFFst}) \qquad \frac{}{\mathsf{snd}\ — : (\tau_1 \times \tau_2) \rightsquigarrow \tau_2}\ (\text{TypeFSnd})$$

$k \leftsquigarrow \tau$ says that the stack k expects a τ.

$$\frac{}{\epsilon \leftsquigarrow \tau}\ (\text{TypeSEmpty}) \qquad \frac{k \leftsquigarrow \tau' \qquad f : \tau \rightsquigarrow \tau'}{k; f \leftsquigarrow \tau}\ (\text{TypeSFrame})$$

Finally, states are OK if the type of the expression or value matches what the stack is expecting.

$$\frac{k \leftsquigarrow \tau \qquad \bullet \vdash e : \tau}{k \triangleright e\ \mathsf{ok}}\ (\text{OKEval}) \qquad \frac{k \leftsquigarrow \tau \qquad \bullet \vdash v : \tau}{k \triangleleft v\ \mathsf{ok}}\ (\text{OKRet})$$

Lemma 1 (Preservation)**.** *If s ok and $s \mapsto s'$ then s' ok.*

Proof. By "induction" (really just inversion) on the derivation of $s \mapsto s'$.

1. Then $s = k \triangleright v$ and $s' = k \triangleleft v$. By inversion on OKEval, we have $k \leftsquigarrow \tau$ and $\bullet \vdash v : \tau$. Apply rule OKRet.

2. Then $s = k \triangleright e_1\ e_2$ and $s' = k; \text{—}\ e_2 \triangleright e_1$. By inversion, $k \leftsquigarrow \tau$ and $\bullet \vdash e_1\ e_2 : \tau$. By inversion on $\to$E, $\bullet \vdash e_1 : \tau_1 \to \tau$ and $\bullet \vdash e_2 : \tau_1$. By TypeFFun and TypeSFrame, $k; \text{—}\ e_2 \leftsquigarrow (\tau_1 \to \tau)\tau$. Apply OKEval.

3. Then $s = k; \text{—}\ e_2 \triangleleft v$ and $s' = k; v\ \text{—} \triangleright e_2$. By inversion, $k \leftsquigarrow \tau$ and $\text{—}\ e_2 : (\tau_1 \to \tau) \rightsquigarrow \tau$ and $\bullet \vdash v : \tau_1 \to \tau$. By inversion, $\bullet \vdash e_2 : \tau_1$. By TypeFArg, $v\ \text{—} : \tau_1 \rightsquigarrow \tau$. By TypeSFrame, $k; v\ \text{—} \leftsquigarrow \tau_1$. Apply OKEval.

4. Then $s = k; \lambda x : \tau_1.e_1\ \text{—} \triangleleft v$ and $s' = k \triangleright [v/x]e_1$. By inversion, $k \leftsquigarrow \tau$ and $\lambda x : \tau_1.e_1\ \text{—} : \tau_1 \rightsquigarrow \tau$ and $\bullet \vdash v : \tau_1$. By inversion, $\bullet \vdash \lambda x : \tau_1.e_1 : \tau_1 \to \tau$. By inversion on $\to$I, $x : \tau_1 \vdash e_1 : \tau$. By substitution, $\bullet \vdash [v/x]e_1 : \tau$. Apply OKEval.

□

Lemma 2 (Progress)**.** *If s ok then either $s = \epsilon \triangleleft v$ or $s \mapsto s'$.*

Proof. By induction on the derivation of s ok.

- OKEval. Then $s = k \triangleright e$ and $\bullet \vdash e : \tau$. By inversion on the derivation of $\bullet \vdash e : \tau$, e is v or $e_1\ e_2$ or (e_1, e_2) or fst e or snd e (it can't be a variable because the context is empty). In those cases, s steps by rule (1), (2), (5), (8), or (10), respectively.

- OKRet. Then $s = k \triangleleft v$ and $k \leftsquigarrow \tau$ and $\bullet \vdash v : \tau$. Proceed by induction on the derivation of $k \leftsquigarrow \tau$. If $k = \epsilon$ then $s = \epsilon \triangleleft v$. Otherwise, $k = k'; f$ and $k' \leftsquigarrow \tau'$ and $f : \tau \rightsquigarrow \tau'$. Proceed by cases on the derivation of $f : \tau \rightsquigarrow \tau'$.

  – TypeFFun. Then $f = \text{—}\ e_2$ and $s \mapsto k'; v\ \text{—} \triangleright e_2$ by (3).
  – TypeFArg. Then $f = v_1\ \text{—}$ and $\bullet \vdash v : \tau \to \tau'$. By canonical forms, $v_1 = \lambda x : \tau.e$ and $s \mapsto k \triangleright [v/x]e$ by (4).
  – TypeFPairL. Then $f = (\text{—}, e_2)$ and $s \mapsto k'; (v, \text{—}) \triangleright e_2$ by (6).
  – TypeFPairR. Then $f = (v_1, \text{—})$ and $s \mapsto k \triangleleft (v_1, v)$ by (7).
  – TypeFFst. Then $f = \mathsf{fst}\ \text{—}$ and $\tau = \tau' \times \tau_2$. By canonical forms, $v = (v_1, v_2)$ and $s \mapsto k \triangleleft v_1$ by (9).
  – TypeFSnd. Then $f = \mathsf{snd}\ \text{—}$ and $\tau = \tau_1 \times \tau'$. By canonical forms, $v = (v_1, v_2)$ and $s \mapsto k \triangleleft v_2$ by (11).

□