COSE212: Programming Languages

Lecture 10 — Design and Implementation of PLs (6) Exceptions

Hakjoo Oh
2026 Fall

Review: The PROC Language

Syntax

$$
\begin{array}{rcl}
\boldsymbol{P} & \rightarrow & \boldsymbol{E} \\
\boldsymbol{E} & \rightarrow & \boldsymbol{n} \\
 & | & \boldsymbol{x} \\
 & | & \boldsymbol{E} + \boldsymbol{E} \\
 & | & \texttt{iszero } \boldsymbol{E} \\
 & | & \texttt{if } \boldsymbol{E} \texttt{ then } \boldsymbol{E} \texttt{ else } \boldsymbol{E} \\
 & | & \texttt{let } \boldsymbol{x} = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
 & | & \texttt{letrec } \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
 & | & \texttt{proc } \boldsymbol{x}\ \boldsymbol{E} \\
 & | & \boldsymbol{E}\ \boldsymbol{E}
\end{array}
$$

Review

Semantics

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)} \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \texttt{iszero}\ E \Rightarrow true} \qquad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \texttt{iszero}\ E \Rightarrow false}\ n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow true \qquad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow false \qquad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v} \qquad \frac{[f \mapsto (f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

$$\frac{}{\rho \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \qquad [x \mapsto v]\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad [x \mapsto v, f \mapsto (f, x, E, \rho')]\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

This Lecture: Adding Exceptions

- So far, evaluating an expression either produces a value or diverges.
- But often we need to handle an exceptional situation that should abort the current computation and jump back to a designated handler.
- Two new constructs:
  - `raise` $\boldsymbol{E}$: raise a value as an exception.
  - `try` $\boldsymbol{E_1}$ `catch` $(\boldsymbol{x})$ $\boldsymbol{E_2}$: evaluate $\boldsymbol{E_1}$; if it raises a value $\boldsymbol{v}$, bind $\boldsymbol{x}$ to $\boldsymbol{v}$ and evaluate $\boldsymbol{E_2}$ instead.
- Implementing this cleanly requires a new technique: continuations.

Example 1

```
let f = proc (x) if iszero(x) then raise 99
                 else x
in try (f(0) + 10) catch (y) y
```

Example 2

```
try (1 + (1 + (1 + raise 100))) catch (z) z
```

Making the Control Context Explicit

When we are about to evaluate `raise` $\mathbf{100}$ in Example 2, the surrounding program is

$$\texttt{try}\ (\mathbf{1} + (\mathbf{1} + (\mathbf{1} + \square)))\ \texttt{catch}\ (\boldsymbol{z})\ \boldsymbol{z}$$

where $\square$ denotes the sub-expression in focus. Everything outside $\square$ is the **control context** of `raise` $\mathbf{100}$: it records what is left to do with the value that $\square$ will produce ("add to $\mathbf{1}$, then add to $\mathbf{1}$, then add to $\mathbf{1}$, then check the `try`").

- Our rules $\boldsymbol{\rho} \vdash \boldsymbol{E} \Rightarrow \boldsymbol{v}$ have no access to this control context: each rule sees only its sub-derivations, not the surrounding context.
- `raise` needs the context so it can abandon the pending additions and jump to the enclosing `try`.
- Fix: thread the control context explicitly as a continuation $\boldsymbol{k}$. Push a frame onto $\boldsymbol{k}$ for each step we descend; `raise` then walks $\boldsymbol{k}$ looking for a `try`.

Making the Control Context Explicit

In Example 2, as we descend into the deepest sub-expression `raise` $\mathbf{100}$, the pending work waiting outside is:

add result to $\mathbf{1}$
add result to $\mathbf{1}$
add result to $\mathbf{1}$
hand result to the `try` handler

(top of the stack is the next thing to do; the `try` entry is the bottom.)

When `raise` $\mathbf{100}$ fires:

- Throw away the three pending additions.
- Walk down the stack until the `try` entry is found.
- Run the handler with $z = \mathbf{100}$.

Next: make this stack an explicit object we can manipulate.

Continuations

A continuation k records **what is left to do** after the current sub-expression returns a value.

We make it explicit by extending the judgment from

$$\rho \vdash E \Rightarrow v \qquad \text{to} \qquad \rho, k \vdash E \Rightarrow v.$$

The next two slides answer the two natural questions:

1. What shapes can k take? (the frames)
2. How does k change during evaluation? (eval/k and apply-cont rules)

Continuation Frames

While we evaluate some sub-expression, k's top frame stores "what to do once that sub-expression returns a value."

$k \in \mathit{Cont}$	::=	$\texttt{end}$	nothing left to do
	|	$\texttt{plus}_1(E_2, \rho, k)$	evaluate E_2 next, then add
	|	$\texttt{plus}_2(v_1, k)$	add v_1 to the arriving value
	|	$\texttt{iszero}_1(k)$	test arriving value against 0
	|	$\texttt{if}_1(E_2, E_3, \rho, k)$	branch on *true*/*false*
	|	$\texttt{let}_1(x, E_2, \rho, k)$	bind x to value, then eval E_2
	|	$\texttt{call}_1(E_2, \rho, k)$	value is a closure; eval E_2 next
	|	$\texttt{call}_2(v, k)$	call closure v on arriving value

For example, $\texttt{plus}_1(E_2, \rho, k)$ sits on top of k and represents the control context of E_1 in $E_1 + E_2$; once E_1's value arrives, the frame tells us to evaluate E_2 next and add the two results.

PROC in Continuation-Passing Form

Two mutually recursive judgments:

$$\begin{array}{ll} \text{eval/k:} & \boldsymbol{\rho, k \vdash E \Rightarrow v} \\ \text{apply-cont:} & \boldsymbol{k \vdash_a v \Rightarrow v'} \end{array}$$

- $\boldsymbol{\rho, k \vdash E \Rightarrow v}$: the result of evaluating $\boldsymbol{E}$ under $\boldsymbol{\rho}$ and then applying $\boldsymbol{k}$ to that value is $\boldsymbol{v}$.
- $\boldsymbol{k \vdash_a v \Rightarrow v'}$: the result of applying $\boldsymbol{k}$ to $\boldsymbol{v}$ is $\boldsymbol{v'}$.

A program $\boldsymbol{P = E}$ runs with the empty environment and the terminal continuation: $\boldsymbol{[], \texttt{end} \vdash E \Rightarrow v}$.

Eval/k: Atomic Expressions

Atomic expressions finish immediately and hand their value to k:

$$\frac{k \vdash_a n \Rightarrow v}{\rho, k \vdash n \Rightarrow v}\ \textsc{Num}$$

$$\frac{k \vdash_a \rho(x) \Rightarrow v}{\rho, k \vdash x \Rightarrow v}\ \textsc{Var}$$

$$\frac{k \vdash_a (x, E, \rho) \Rightarrow v}{\rho, k \vdash \texttt{proc}\ x\ E \Rightarrow v}\ \textsc{Proc}$$

`letrec` also pushes no frame:

$$\frac{[f \mapsto (f, x, E_1, \rho)]\rho,\ k \vdash E_2 \Rightarrow v}{\rho, k \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v}\ \textsc{Letrec}$$

Eval/k: Compound Expressions

A compound expression evaluates its first sub-expression and pushes a frame describing what to do next with the result:

$$\frac{\rho,\ \texttt{plus}_1(E_2, \rho, k) \vdash E_1 \Rightarrow v}{\rho, k \vdash E_1 + E_2 \Rightarrow v}\ \textsc{Plus}$$

$$\frac{\rho,\ \texttt{iszero}_1(k) \vdash E \Rightarrow v}{\rho, k \vdash \texttt{iszero}\ E \Rightarrow v}\ \textsc{Iszero}$$

$$\frac{\rho,\ \texttt{if}_1(E_2, E_3, \rho, k) \vdash E_1 \Rightarrow v}{\rho, k \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v}\ \textsc{If}$$

$$\frac{\rho,\ \texttt{let}_1(x, E_2, \rho, k) \vdash E_1 \Rightarrow v}{\rho, k \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v}\ \textsc{Let}$$

$$\frac{\rho,\ \texttt{call}_1(E_2, \rho, k) \vdash E_1 \Rightarrow v}{\rho, k \vdash E_1\ E_2 \Rightarrow v}\ \textsc{Call}$$

Apply-cont (1)

Each rule fires when a value arrives at a particular frame on top of k:

$$\frac{}{\texttt{end} \vdash_a v \Rightarrow v}\ \textsc{End}$$

$$\frac{\rho,\ \texttt{plus}_2(v_1, k) \vdash E_2 \Rightarrow v}{\texttt{plus}_1(E_2, \rho, k) \vdash_a v_1 \Rightarrow v}\ \textsc{Plus}_1 \qquad \frac{k \vdash_a n_1 + n_2 \Rightarrow v}{\texttt{plus}_2(n_1, k) \vdash_a n_2 \Rightarrow v}\ \textsc{Plus}_2$$

$$\frac{k \vdash_a \mathit{true} \Rightarrow v}{\texttt{iszero}_1(k) \vdash_a 0 \Rightarrow v}\ \textsc{IsZero-T} \qquad \frac{k \vdash_a \mathit{false} \Rightarrow v}{\texttt{iszero}_1(k) \vdash_a n \Rightarrow v}\ \textsc{IsZero-F}\ (n \neq 0)$$

$$\frac{\rho, k \vdash E_2 \Rightarrow v}{\texttt{if}_1(E_2, E_3, \rho, k) \vdash_a \mathit{true} \Rightarrow v}\ \textsc{If-T} \qquad \frac{\rho, k \vdash E_3 \Rightarrow v}{\texttt{if}_1(E_2, E_3, \rho, k) \vdash_a \mathit{false} \Rightarrow v}\ \textsc{If-F}$$

$$\frac{[x \mapsto v_1]\rho,\ k \vdash E_2 \Rightarrow v}{\texttt{let}_1(x, E_2, \rho, k) \vdash_a v_1 \Rightarrow v}\ \textsc{Let}_1$$

Apply-cont (2)

$$\frac{\rho,\ \texttt{call}_2((x, E, \rho'), k) \vdash E_2 \Rightarrow v}{\texttt{call}_1(E_2, \rho, k) \vdash_a (x, E, \rho') \Rightarrow v}\ \textsc{Call}_1$$

$$\frac{\rho,\ \texttt{call}_2((f, x, E, \rho'), k) \vdash E_2 \Rightarrow v}{\texttt{call}_1(E_2, \rho, k) \vdash_a (f, x, E, \rho') \Rightarrow v}\ \textsc{Call}_1^{\mathbf{r}}$$

$$\frac{[x \mapsto v_1]\rho',\ k \vdash E \Rightarrow v}{\texttt{call}_2((x, E, \rho'), k) \vdash_a v_1 \Rightarrow v}\ \textsc{Call}_2$$

$$\frac{[x \mapsto v_1,\ f \mapsto (f, x, E, \rho')]\rho',\ k \vdash E \Rightarrow v}{\texttt{call}_2((f, x, E, \rho'), k) \vdash_a v_1 \Rightarrow v}\ \textsc{Call}_2^{\mathbf{r}}$$

Example: $(\texttt{proc}\ x\ (x+1))\ 5 \Rightarrow 6$

$$
\begin{array}{lll}
[], \texttt{end} \vdash (\texttt{proc}\ x\ (x+1))\ 5 & \xrightarrow{\textsc{Call}} & [], \texttt{call}_1(5, [], \texttt{end}) \vdash \texttt{proc}\ x\ (x+1) \\
& \xrightarrow{\textsc{Proc}} & \texttt{call}_1(5, [], \texttt{end}) \vdash_a (x, x+1, []) \\
& \xrightarrow{\textsc{Call}_1} & [], \texttt{call}_2((x, x+1, []), \texttt{end}) \vdash 5 \\
& \xrightarrow{\textsc{Num}} & \texttt{call}_2((x, x+1, []), \texttt{end}) \vdash_a 5 \\
& \xrightarrow{\textsc{Call}_2} & [x \mapsto 5], \texttt{end} \vdash x+1 \\
& \xrightarrow{\textsc{Plus}} & [x \mapsto 5], \texttt{plus}_1(1, [x \mapsto 5], \texttt{end}) \vdash x \\
& \xrightarrow{\textsc{Var}} & \texttt{plus}_1(1, [x \mapsto 5], \texttt{end}) \vdash_a 5 \\
& \xrightarrow{\textsc{Plus}_1} & [x \mapsto 5], \texttt{plus}_2(5, \texttt{end}) \vdash 1 \\
& \xrightarrow{\textsc{Num}} & \texttt{plus}_2(5, \texttt{end}) \vdash_a 1 \\
& \xrightarrow{\textsc{Plus}_2} & \texttt{end} \vdash_a 6 \\
& \xrightarrow{\textsc{End}} & 6
\end{array}
$$

Example: $(\texttt{let}\ f = \texttt{proc}\ x\ (x+1)\ \texttt{in}\ f)\ 5 \Rightarrow 6$

$[], \texttt{end} \vdash (\texttt{let}\ f = \texttt{proc}\ x\ (x+1)\ \texttt{in}\ f)\ 5$

$\xrightarrow{\textsc{Call}}\ [], \texttt{call}_1(5, [], \texttt{end}) \vdash \texttt{let}\ f = \texttt{proc}\ x\ (x+1)\ \texttt{in}\ f$

$\xrightarrow{\textsc{Let}}\ [], \texttt{let}_1(f, f, [], \texttt{call}_1(5, [], \texttt{end})) \vdash \texttt{proc}\ x\ (x+1)$

$\xrightarrow{\textsc{Proc}}\ \texttt{let}_1(f, f, [], \texttt{call}_1(5, [], \texttt{end})) \vdash_a (x, x+1, [])$

$\xrightarrow{\textsc{Let}_1}\ [f \mapsto (x, x+1, [])], \texttt{call}_1(5, [], \texttt{end}) \vdash f$

$\xrightarrow{\textsc{Var}}\ \texttt{call}_1(5, [], \texttt{end}) \vdash_a (x, x+1, [])$

$\vdots$ (same as previous slide from step 2 onward)

$\longrightarrow\ 6$

Adding Exceptions to PROC

Two new expressions:

$$E ::= \cdots \mid \texttt{try}\ E_1\ \texttt{catch}\ (x)\ E_2 \mid \texttt{raise}\ E$$

Continuation frames:

$$k ::= \cdots \mid \texttt{try}(x, E_h, \rho, k) \mid \texttt{raise}_1(k)$$

- $\texttt{try}(x, E_h, \rho, k)$: an installed handler. On a normal value, pass it through to k; findable by `raise`.
- $\texttt{raise}_1(k)$: intercepts the arriving value and searches k for an enclosing `try` to hand it to.

Rules: `try` and `raise`

`try` installs a handler frame; `raise` evaluates its argument under a $\texttt{raise}_1$ frame:

$$\frac{\rho,\ \texttt{try}(x, E_2, \rho, k) \vdash E_1 \Rightarrow v}{\rho, k \vdash \texttt{try}\ E_1\ \texttt{catch}\ (x)\ E_2 \Rightarrow v}\ \textsc{Try}$$

$$\frac{\rho,\ \texttt{raise}_1(k) \vdash E \Rightarrow v}{\rho, k \vdash \texttt{raise}\ E \Rightarrow v}\ \textsc{Raise}$$

On the apply-cont side: a successful `try` frame passes the value through; a $\texttt{raise}_1$ frame triggers handler search:

$$\frac{k \vdash_a v \Rightarrow v'}{\texttt{try}(x, E_h, \rho, k) \vdash_a v \Rightarrow v'}\ \textsc{Try-Ok}$$

$$\frac{\textbf{find-handler}(k,\ v) \Rightarrow v'}{\texttt{raise}_1(k) \vdash_a v \Rightarrow v'}\ \textsc{Raise}_1$$

Searching for a Handler

find-handler(k, v) walks the continuation, looking for the nearest `try` frame. Non-handler frames are discarded:

$$\frac{[x \mapsto v]\rho,\ k \vdash E_h \Rightarrow v'}{\textbf{find-handler}(\texttt{try}(x, E_h, \rho, k),\ v) \Rightarrow v'}\ \textsc{Fh-Match}$$

$$\frac{F \neq \texttt{try} \quad \textbf{find-handler}(k,\ v) \Rightarrow v'}{\textbf{find-handler}(F(\ldots, k),\ v) \Rightarrow v'}\ \textsc{Fh-Skip}$$

- All frames between the `raise` and the matching `try` ($\texttt{plus}_2$, $\texttt{call}_2$, $\texttt{let}_1$, ...) are discarded.
- The handler E_h runs in the original environment of the `try`, extended with $x \mapsto v$, and continues with k (the continuation that was waiting outside the `try`).
- If k contains no `try` frame, no rule applies at `end`: a runtime error (uncaught exception).

Example

Trace of $\texttt{try}\ (\mathbf{1} + \texttt{raise}\ \mathbf{5})\ \texttt{catch}\ (x)\ x$. Let $K = \texttt{try}(x, x, [], \texttt{end})$.

$$
\begin{array}{lll}
[], \texttt{end} \vdash \texttt{try}\ (\mathbf{1} + \texttt{raise}\ \mathbf{5})\ \texttt{catch}\ (x)\ x & \xrightarrow{\textsc{Try}} & [], K \vdash \mathbf{1} + \texttt{raise}\ \mathbf{5} \\
 & \xrightarrow{\textsc{Plus}} & [], \texttt{plus}_1(\texttt{raise}\ \mathbf{5}, [], K) \vdash \mathbf{1} \\
 & \xrightarrow{\textsc{Num}} & \texttt{plus}_1(\texttt{raise}\ \mathbf{5}, [], K) \vdash_a \mathbf{1} \\
 & \xrightarrow{\textsc{Plus}_1} & [], \texttt{plus}_2(\mathbf{1}, K) \vdash \texttt{raise}\ \mathbf{5} \\
 & \xrightarrow{\textsc{Raise}} & [], \texttt{raise}_1(\texttt{plus}_2(\mathbf{1}, K)) \vdash \mathbf{5} \\
 & \xrightarrow{\textsc{Num}} & \texttt{raise}_1(\texttt{plus}_2(\mathbf{1}, K)) \vdash_a \mathbf{5} \\
 & \xrightarrow{\textsc{Raise}_1} & \textbf{find-handler}(\texttt{plus}_2(\mathbf{1}, K), \mathbf{5}) \\
 & \xrightarrow{\textsc{Fh-Skip}} & \textbf{find-handler}(K, \mathbf{5}) \\
 & \xrightarrow{\textsc{Fh-Match}} & [x \mapsto \mathbf{5}], \texttt{end} \vdash x \\
 & \xrightarrow{\textsc{Var}} & \texttt{end} \vdash_a \mathbf{5} \\
 & \xrightarrow{\textsc{End}} & \mathbf{5}
\end{array}
$$

Summary

We extended PROC with exceptions:

- Rewrote the semantics in continuation-passing style.
- Added `try` $\boldsymbol{E}$ `catch` $(\boldsymbol{x})$ $\boldsymbol{E}$ and `raise` $\boldsymbol{E}$.