Programming Language Theory

Evaluation Contexts, First-Class Continuations, and Continuation-Passing-Style
Type Systems Respite

Let’s spend one lecture on a somewhat different topic.

▶ How operational semantics can be defined more concisely, and how they can be related to abstract machines.
▶ How lambda-calculus can be extended with *first-class continuations*, a powerful *control operator*.
▶ Some programming idioms related to these concepts.
(Simply-Typed) Lambda Calculus with Extensions (pairs, sums, fix)

Syntax:

\[
e ::= \lambda x. e \mid x \mid e \ e \mid (e,e) \mid e.1 \mid e.2 \mid L(e) \mid R(e) \mid \text{case } e \text{ of } L(x) \Rightarrow e \mid R(y) \Rightarrow e \mid \text{fix } e
\]

\[
v ::= \lambda x. e \mid (v,v) \mid L(v) \mid R(v)
\]
(Simply-Typed) Lambda Calculus with Extensions (pairs, sums, fix)

Small-step, call-by-value (CBV), left-to-right operational semantics:

\[
\begin{align*}
\frac{e_f \rightarrow_{cbv} e'_f}{e_f \, e_a \rightarrow_{cbv} e'_f \, e_a} & \quad \frac{e_a \rightarrow_{cbv} e'_a}{v_f \, e_a \rightarrow_{cbv} v_f \, e'_a} & \quad (\lambda x. e_b) \, v_a \rightarrow_{cbv} e_b[v_a/x] \\
\frac{e_1 \rightarrow_{cbv} e'_1}{(e_1, e_2) \rightarrow_{cbv} (e'_1, e_2)} & \quad \frac{e_2 \rightarrow_{cbv} e'_2}{(v_1, e_2) \rightarrow_{cbv} (v'_1, e'_2)} \\
\frac{e_p \rightarrow_{cbv} e'_p}{e_p \cdot 1 \rightarrow_{cbv} e'_p \cdot 1} & \quad \frac{e_p \rightarrow_{cbv} e'_p}{(v_1, v_2) \cdot 1 \rightarrow_{cbv} v_1} & \quad \frac{e_p \rightarrow_{cbv} e'_p}{e_p \cdot 2 \rightarrow_{cbv} e'_p \cdot 2} & \quad (v_1, v_2) \cdot 2 \rightarrow_{cbv} v_2 \\
\frac{e_s \rightarrow_{cbv} e'_s}{\text{case } e_s \text{ of } L(x) \Rightarrow e_l | R(y) \Rightarrow e_r \rightarrow_{cbv} \text{ case } e'_s \text{ of } L(x) \Rightarrow e_l | R(y) \Rightarrow e_r} \\
\frac{e_1 \rightarrow_{cbv} e'_1}{\text{case } L(v_1) \text{ of } L(x) \Rightarrow e_l | R(y) \Rightarrow e_r \rightarrow_{cbv} e_l[v_1/x]} \\
\frac{e_2 \rightarrow_{cbv} e'_2}{\text{case } R(v_2) \text{ of } L(x) \Rightarrow e_l | R(y) \Rightarrow e_r \rightarrow_{cbv} e_r[v_2/y]} \\
\frac{e_f \rightarrow_{cbv} e'_f}{\text{fix } e_f \rightarrow_{cbv} \text{ fix } e'_f} & \quad \frac{\text{fix } (\lambda x. e_b) \rightarrow_{cbv} e_b[\text{fix } (\lambda x. e_b)/x]}{}
\end{align*}
\]
(Simply-Typed) Lambda Calculus with Extensions (pairs, sums, fix)

Small-step, *call-by-value (CBV)*, left-to-right operational semantics:

\[
\begin{align*}
\text{Small-step, call-by-value (CBV), left-to-right operational semantics:} \\
\frac{e_f \rightarrow_{cbv} e_f'}{e_f \ e_a \rightarrow_{cbv} e_f' \ e_a} & \quad \frac{e_a \rightarrow_{cbv} e_a'}{v_f \ e_a \rightarrow_{cbv} v_f \ e_a'} & \quad (\lambda x. \ e_b) \ v_a \rightarrow_{cbv} e_b[v_a/x] \\
\frac{e_1 \rightarrow_{cbv} e_1'}{(e_1, e_2) \rightarrow_{cbv} (e_1', e_2)} & \quad \frac{e_2 \rightarrow_{cbv} e_2'}{(v_1, e_2) \rightarrow_{cbv} (v_1', e_2')} \\
\frac{e_p \rightarrow_{cbv} e_p'}{e_p.1 \rightarrow_{cbv} e_p'.1} & \quad \frac{(v_1, v_2).1 \rightarrow_{cbv} v_1}{e_p \rightarrow_{cbv} e_p'} & \quad \frac{(v_1, v_2).2 \rightarrow_{cbv} v_2}{e_p \rightarrow_{cbv} e_p'} \\
\frac{e_s \rightarrow_{cbv} e_s'}{\text{case } e_s \text{ of } L(x) => e_l | R(y) => e_r \rightarrow_{cbv} \text{ case } e_s' \text{ of } L(x) => e_l | R(y) => e_r} \\
\frac{e_1 \rightarrow_{cbv} e_1'}{L(e_1) \rightarrow_{cbv} L(e_1')} & \quad \frac{e_2 \rightarrow_{cbv} e_2'}{R(e_2) \rightarrow_{cbv} R(e_2')} \\
\frac{e_f \rightarrow_{cbv} e_f'}{\text{fix } e_f \rightarrow_{cbv} \text{ fix } e_f'} & \quad \frac{\text{fix } (\lambda x. \ e_b) \rightarrow_{cbv} e_b[\text{fix } (\lambda x. \ e_b)/x]}{\text{case } L(v_1) \text{ of } L(x) => e_l | R(y) => e_r \rightarrow_{cbv} e_l[v_1/x]} \\
\quad \frac{\text{case } R(v_2) \text{ of } L(x) => e_l | R(y) => e_r \rightarrow_{cbv} e_r[v_2/y]}{}
\end{align*}
\]

Note: lots of "boring inductive rules" with some "interesting do-work rules"
(Simply-Typed) Lambda Calculus with Extensions (pairs, sums, fix)

Small-step, call-by-value (CBV), left-to-right operational semantics:

\[
\begin{align*}
& e_f \rightarrow_{\text{cbv}} e'_f \\
& e_f e_a \rightarrow_{\text{cbv}} e'_f e_a \\
& e_a \rightarrow_{\text{cbv}} e'_a \\
& v_f e_a \rightarrow_{\text{cbv}} v_f e'_a \\
& (\lambda x. e_b) v_a \rightarrow_{\text{cbv}} e_b[v_a/x]
\end{align*}
\]

\[
\begin{align*}
& e_1 \rightarrow_{\text{cbv}} e'_1 \\
& (e_1, e_2) \rightarrow_{\text{cbv}} (e'_1, e_2) \\
& e_2 \rightarrow_{\text{cbv}} e'_2 \\
& (v_1, e_2) \rightarrow_{\text{cbv}} (v_1, e'_2)
\end{align*}
\]

\[
\begin{align*}
& e_p \rightarrow_{\text{cbv}} e'_p \\
& e_p \cdot 1 \rightarrow_{\text{cbv}} e'_p \cdot 1 \\
& (v_1, v_2) \cdot 1 \rightarrow_{\text{cbv}} v_1 \\
& e_p \rightarrow_{\text{cbv}} e'_p \\
& e_p \cdot 2 \rightarrow_{\text{cbv}} e'_p \cdot 2 \\
& (v_1, v_2) \cdot 2 \rightarrow_{\text{cbv}} v_2
\end{align*}
\]

\[
\begin{align*}
& e_s \rightarrow_{\text{cbv}} e'_s \\
& \text{case } e_s \text{ of } L(x) = \Rightarrow e_l | R(y) = \Rightarrow e_r \rightarrow_{\text{cbv}} \text{case } e'_s \text{ of } L(x) = \Rightarrow e_l | R(y) = \Rightarrow e_r
\end{align*}
\]

\[
\begin{align*}
& e_1 \rightarrow_{\text{cbv}} e'_1 \\
& L(e_1) \rightarrow_{\text{cbv}} L(e'_1) \\
& e_2 \rightarrow_{\text{cbv}} e'_2 \\
& R(e_2) \rightarrow_{\text{cbv}} R(e'_2)
\end{align*}
\]

\[
\begin{align*}
& \text{case } L(v_1) \text{ of } L(x) = \Rightarrow e_l | R(y) = \Rightarrow e_r \rightarrow_{\text{cbv}} e_l[v_1/x]
\end{align*}
\]

\[
\begin{align*}
& \text{case } R(v_2) \text{ of } L(x) = \Rightarrow e_l | R(y) = \Rightarrow e_r \rightarrow_{\text{cbv}} e_r[v_2/y]
\end{align*}
\]

\[
\begin{align*}
& e_f \rightarrow_{\text{cbv}} e'_f \\
& \text{fix } e_f \rightarrow_{\text{cbv}} \text{fix } e'_f \\
& \text{fix } (\lambda x. e_b) \rightarrow_{\text{cbv}} e_b[\text{fix } (\lambda x. e_b)/x]
\end{align*}
\]

Note: lots of “boring inductive rules”
(Simply-Typed) Lambda Calculus with Extensions (pairs, sums, fix)

Small-step, *call-by-value (CBV)*, left-to-right operational semantics:

\[
\begin{align*}
\text{ef} & \to_{cbv} e_f' \\
\text{ef ea} & \to_{cbv} e_f' e_a \\
\text{ea} & \to_{cbv} e_a' \\
\text{vf ea} & \to_{cbv} v_f e_a' \\
(\lambda x. \text{eb}) v_a & \to_{cbv} \text{eb}[v_a/x] \\
\text{e1} & \to_{cbv} e_1' \\
(\text{e1}, \text{e2}) & \to_{cbv} (e_1', e_2') \\
\text{e2} & \to_{cbv} e_2' \\
(\text{v1}, \text{e2}) & \to_{cbv} (v_1, e_2') \\
\text{ep} & \to_{cbv} e_p' \\
\text{ep.1} & \to_{cbv} e_p'.1 \\
(\text{v1}, \text{v2}).1 & \to_{cbv} v_1 \\
\text{ep.2} & \to_{cbv} e_p'.2 \\
(\text{v1}, \text{v2}).2 & \to_{cbv} v_2 \\
\text{es} & \to_{cbv} e_s' \\
\text{case es of L(x) => e_l | R(y) => e_r} & \to_{cbv} \text{case e_s' of L(x) => e_l | R(y) => e_r} \\
\text{e1} & \to_{cbv} e_1' \\
\text{L(e1)} & \to_{cbv} \text{L(e1')} \\
\text{e2} & \to_{cbv} e_2' \\
\text{R(e2)} & \to_{cbv} \text{R(e2')} \\
\text{fix ef} & \to_{cbv} \text{fix e_f'} \\
\text{fix (\lambda x. \text{eb})} & \to_{cbv} \text{eb[fix (\lambda x. \text{eb})/x]} \\
\end{align*}
\]

Note: lots of “boring inductive rules” with some “interesting do-work rules”
Rethinking Small-Step Operational Semantics

Every $e \rightarrow_{\text{cbv}} e'$ derivation uses some "boring inductive rules" and one "interesting do-work rule".

Therefore, executing a program works like:

- Find the next “primitive step”
  - (function application, pair selection, case dispatch, recursion unrolling)
- Perform that “primitive step”
- Plug the result back into the rest of the program
- Repeat (next “primitive step” could be at a new place)
- Until program is a value (or is “stuck”)

Move the first step out and produce a data structure that describes where the next “primitive step” occurs.
Evaluation Contexts

Define evaluation contexts:

- expressions with one “hole” where the “interesting work” may occur

\[
E ::= \cdot | E \ e | v \ E | (E, e) | (v, E) | E.1 | E.2 | L(E) | R(E) | \text{case } E \text{ of } L(x) \Rightarrow e | R(y) \Rightarrow e | \text{fix } E
\]

Define “filling the hole” \( E[e] \) in the obvious way.

- A metafunction of type EvalContext \( \rightarrow \) Exp \( \rightarrow \) Exp

Semantics now uses two judgements \( e \rightarrow_{cbvc} e' \) and \( e \xrightarrow{p}_{cbvc} e' \), but the former has only 1 rule and the latter has just the “interesting work”.
Evaluation Contexts

\[ e \rightarrow_{\text{cbvc}} e' \]

\[ e = E[e_a] \quad e_a \xrightarrow{p}_{\text{cbvc}} e_a' \quad E[e_a'] = e' \]
\[ e \rightarrow_{\text{cbvc}} e' \]

\[ (\lambda x. e_b) \ x \xrightarrow{p}_{\text{cbvc}} e_b[v_a/x] \]

\[ (v_1, v_2).1 \xrightarrow{p}_{\text{cbvc}} v_1 \]
\[ (v_1, v_2).2 \xrightarrow{p}_{\text{cbvc}} v_2 \]

case L(v_1) of L(x) \Rightarrow e_l | R(y) \Rightarrow e_r \xrightarrow{p}_{\text{cbvc}} e_l[v_1/x] \]

\[ \text{case R}(v_2) \text{ of } L(x) \Rightarrow e_l | R(y) \Rightarrow e_r \xrightarrow{p}_{\text{cbvc}} e_r[v_2/y] \]

\[ \text{fix } (\lambda x. e_b) \xrightarrow{p}_{\text{cbvc}} e_b[\text{fix } (\lambda x. e_b)/x] \]
Evaluation Contexts

\[ e \xrightarrow{cbvc} e' \]

\[
\frac{e_a \xrightarrow{p} e'_a}{E[e_a] \xrightarrow{cbvc} E[e'_a]}
\]

\[ e \xrightarrow{p} e' \]

\[
\frac{(\lambda x. e_b) v_a \xrightarrow{p} e_b[v_a/x]}{(v_1, v_2).1 \xrightarrow{p} v_1}
\]

\[
\frac{(v_1, v_2).2 \xrightarrow{p} v_2}{\text{case } L(v_1) \text{ of } L(x) \Rightarrow e_l \mid R(y) \Rightarrow e_r \xrightarrow{p} e_l[v_1/x]}
\]

\[
\frac{\text{case } R(v_2) \text{ of } L(x) \Rightarrow e_l \mid R(y) \Rightarrow e_r \xrightarrow{p} e_r[v_2/y]}{\text{fix } (\lambda x. e_b) \xrightarrow{p} e_b[\text{fix } (\lambda x. e_b)/x]}
\]
Evaluation Contexts: So what?

Thus far, all we have done is rearrange our semantics to be more concise.
▶ Each boring rule becomes a form of $E$

Evaluation relies on decomposition:
▶ Given $e$, find an $E$, $e_a$, $e'_a$ such that $e = E[e_a]$ and $e_a \xrightarrow{p}_{cbvc} e'_a$.

Theorem (Unique Decomposition): For all $e$, there is at most one decomposition of $e$ into an $E$ and $e_a$ (such that $e_a \xrightarrow{p}_{cbvc} e'_a$).
▶ When is there no decomposition?
Evaluation Contexts: So what?

Thus far, all we have done is rearrange our semantics to be more concise.

- Each boring rule becomes a form of $E$

Evaluation relies on decomposition:

- Given $e$, find an $E$, $e_a$, $e'_a$ such that $e = E[e_a]$ and $e_a \xrightarrow{p}_{\text{cbvc}} e'_a$.

Theorem (Unique Decomposition): For all $e$, there is at most one decomposition of $e$ into an $E$ and $e_a$ (such that $e_a \xrightarrow{p}_{\text{cbvc}} e'_a$).

- When is there no decomposition?

Unique Decomposition means that

- Evaluation is deterministic

- Progress (restated): If $e$ is well-typed, then $e$ is a value or there is a decomposition of $e$.
Evaluation Contexts

In fact, don’t even need two judgements:

\[ e \rightarrow_{cbvc} e' \]

\[
E[(\lambda x. e_b) v_a] \rightarrow_{cbvc} E[e_b[v_a/x]]
\]

\[
E[(v_1, v_2).1] \rightarrow_{cbvc} E[v_1]
\]

\[
E[(v_1, v_2).2] \rightarrow_{cbvc} E[v_2]
\]

\[
E[\text{case } L(v_1) \text{ of } L(x) => e_l \mid R(y) => e_r] \rightarrow_{cbvc} E[e_l[v_1/x]]
\]

\[
E[\text{case } R(v_2) \text{ of } L(x) => e_l \mid R(y) => e_r] \rightarrow_{cbvc} E[e_r[v_2/y]]
\]

\[
E[\text{fix } (\lambda x. e_b)] \rightarrow_{cbvc} E[e_b[\text{fix } (\lambda x. e_b)/x]]
\]
Evaluation Contexts: So what?

Small-step semantics (old) and evaluation-context semantics (new) are very similar:

- $e \rightarrow_{\text{cbv}} e'$ if and only if $e \rightarrow_{\text{cbVC}} e'$. (total equivalence of $\rightarrow_{\text{cbv}}$ and $\rightarrow_{\text{cbVC}}$ semantics)

- Just rearranged things to be more concise:
  each boring rule became a form of $E$
  (Proofs aren't necessarily any easier; will often need induction on $E$.)

- Both “work” the same way:
  - Find the next “primitive step”
    (function application, pair selection, case dispatch, recursion unrolling)
  - Perform that “primitive step”
  - Plug the result back into the rest of the program
  - Repeat (next “primitive step” could be at a new place)
  - Until program is a value (or is “stuck”)

Evaluation contexts so far just cleanly separate the “find and plug” from the “perform that primitive step” by building an explicit $E$. 
But, now that we have defined $E$ explicitly in our metalanguage, what happens if we allow $E$ in our language:

- Moving from metalanguage to language is called *reification*
- Programs (in language) might save and restore evaluation contexts

Sufficient for

- Exceptions
- Cooperative threads / coroutines / iterators
- “Time travel” with stacks
- *setjmp/longjmp*
First-class Continuations

First-class continuations in one slide:

\[
\begin{align*}
e & ::= \ldots \mid \text{letcc } x \cdot e \mid \text{throw } e e \mid \text{cont } E \\
v & ::= \ldots \mid \text{cont } E \\
E & ::= \ldots \mid \text{throw } E e \mid \text{throw } v E
\end{align*}
\]

\[
E[\text{letcc } x \cdot e] \rightarrow_{\text{cbvc}} E[e[\text{cont } E/x]]
\]

\[
E[\text{throw } (\text{cont } E') v] \rightarrow_{\text{cbvc}} E'[v]
\]

- letcc \( x \cdot e \) gets the current evaluation context ("grab the stack")
- throw \( (\text{cont } E') v \) restores an old evaluation context ("jump somewhere")
- cont \( E \) stores an evaluation context as a value ("saved stack")
  - cont \( E \) shouldn’t appear in source programs
Examples: Exception-like

\[
\text{letcc } k. (\text{throw } k \ 3) \rightarrow_{\text{cbvc}} ^* \\
\text{letcc } k. \ 3 \rightarrow_{\text{cbvc}} ^* \\
1 + (\text{letcc } k. (\text{throw } k \ (3 + 5))) \rightarrow_{\text{cbvc}} ^* \\
1 + (\text{letcc } k. (3 + \text{throw } k \ 5)) \rightarrow_{\text{cbvc}} ^* \\
1 + (\text{letcc } k. (\text{throw } k \ (\text{throw } k \ (\text{throw } k \ 3)))) \rightarrow_{\text{cbvc}} ^*
\]
Examples: Exception-like

\[ \text{letcc } k. (\text{throw } k \ 3) \rightarrow_{\text{cbvc}}^* 3 \]

\[ \text{letcc } k. \ 3 \rightarrow_{\text{cbvc}}^* \]

\[ 1 + (\text{letcc } k. (\text{throw } k \ (3 + 5))) \rightarrow_{\text{cbvc}}^* \]

\[ 1 + (\text{letcc } k. (3 + \text{throw } k \ 5)) \rightarrow_{\text{cbvc}}^* \]

\[ 1 + (\text{letcc } k. (\text{throw } k \ (\text{throw } k \ (\text{throw } k \ 3)))) \rightarrow_{\text{cbvc}}^* \]
Examples: Exception-like

\[
\text{letcc } k. (\text{throw } k \; 3) \rightarrow_{\text{cbvc}} 3
\]

\[
\text{letcc } k. \; 3 \rightarrow_{\text{cbvc}} 3
\]

\[
1 + (\text{letcc } k. \; (\text{throw } k \; (3 + 5))) \rightarrow_{\text{cbvc}}
\]

\[
1 + (\text{letcc } k. \; (3 + \text{throw } k \; 5)) \rightarrow_{\text{cbvc}}
\]

\[
1 + (\text{letcc } k. \; (\text{throw } k \; (\text{throw } k \; (\text{throw } k \; 3)))) \rightarrow_{\text{cbvc}}
\]
Examples: Exception-like

\[
\text{letcc } k. (\text{throw } k \ 3) \rightarrow_{\text{cbvc}} 3
\]

\[
\text{letcc } k. 3 \rightarrow_{\text{cbvc}} 3
\]

\[
1 + (\text{letcc } k. (\text{throw } k \ (3 + 5))) \rightarrow_{\text{cbvc}} 9
\]

\[
1 + (\text{letcc } k. (3 + \text{throw } k \ 5)) \rightarrow_{\text{cbvc}}
\]

\[
1 + (\text{letcc } k. (\text{throw } k \ (\text{throw } k \ (\text{throw } k \ 3)))) \rightarrow_{\text{cbvc}}
\]
Examples: Exception-like

\[
\text{letcc } k. (\text{throw } k \ 3) \rightarrow_{\text{cbvc}}^* 3
\]

\[
\text{letcc } k. \ 3 \rightarrow_{\text{cbvc}}^* 3
\]

\[
1 + (\text{letcc } k. (\text{throw } k \ (3 + 5))) \rightarrow_{\text{cbvc}}^* 9
\]

\[
1 + (\text{letcc } k. (3 + \text{throw } k \ 5)) \rightarrow_{\text{cbvc}}^* 6
\]

\[
1 + (\text{letcc } k. (\text{throw } k \ (\text{throw } k \ (\text{throw } k \ 3)))) \rightarrow_{\text{cbvc}}^* 4
\]
Examples: Exception-like

\[
\text{letcc } k. \ (\text{throw } k \ 3) \rightarrow_{\text{cbvc}} 3
\]

\[
\text{letcc } k. \ 3 \rightarrow_{\text{cbvc}} 3
\]

\[
1 + (\text{letcc } k. \ (\text{throw } k \ (3 + 5))) \rightarrow_{\text{cbvc}} 9
\]

\[
1 + (\text{letcc } k. \ (3 + \text{throw } k \ 5)) \rightarrow_{\text{cbvc}} 6
\]

\[
1 + (\text{letcc } k. \ (\text{throw } k \ (\text{throw } k \ (\text{throw } k \ 3)))) \rightarrow_{\text{cbvc}} 4
\]
Example: “Time travel”-like

SML/NJ has first-class continuations:

```plaintext
open SMLofNJ.Cont
val x = ref true (* avoids infinite loop *)
val g : int cont option ref = ref NONE
val y = ref (1 + 2 + (callcc (fn k => ((g := SOME k); 3))))
val z = if !x then (x := false; throw (valOf (!g)) 7) else !y
val _ = print ("z = " ^ (Int.toString z))
```

What would happen if we didn’t use the `x` mutable reference?
Example: “Time travel”-like

SML/NJ has first-class continuations:

```sml
open SMLofNJ.Cont
val x = ref true (* avoids infinite loop *)
val g : int cont option ref = ref NONE
val y = ref (1 + 2 + (callcc (fn k => ((g := SOME k); 3)))))
val z = if !x then (x := false; throw (valOf (!g)) 7) else !y
val _ = print ("z = " ^ (Int.toString z))
```

```
z = 10
```

What would happen if we didn’t use the `x` mutable reference?
Are Continuations Useful?

- **Exceptions**
  - `letcc x. e` for `e` handle `_` => `e'`
  - `throw x e'` for `raise in e`
  - (the `x` thrown to must be the `x` captured; simpler with a global reference)

- **Coroutines**
  - `yield` captures the continuation (the “how to resume me”) and throws it to the other’s “how to resume me”

- **Cooperative threads**
  - Generalize coroutines; each `yield` is to thread scheduler (but thread scheduler implemented in language, not runtime system)

- **Other crazy things**
  - The “goto of functional programming” — incredibly powerful, but non-standard uses are usually inscrutable
  - Key point is that we can “jump back”, unlike exceptions
  - Close connections with recent research on “algebraic effects” (almost all computational effects can be implemented with continuations)
Another View

If you’re confused, think call stacks:

- What if your favorite language had these operations:
  - Store current stack in \( x \)
  - Replace current stack with stack in \( x \)

- Need to “resume the stack’s hole” with something different or when mutable state is different
- (else, you will have an infinite loop)
First-class Continuations

First-class continuations in one slide:

\[
\begin{align*}
  e & ::= \cdots \mid \text{letcc } x.\ e \mid \text{throw } e\ e \mid \text{cont } E \\
  v & ::= \cdots \mid \text{cont } E \\
  E & ::= \cdots \mid \text{throw } E\ e \mid \text{throw } v\ E
\end{align*}
\]

\[
\begin{align*}
  E[\text{letcc } x.\ e] & \rightarrow_{\text{cbvc}} E[\text{e[cont } E/x]] \\
  E[\text{throw } (\text{cont } E')\ v] & \rightarrow_{\text{cbvc}} E'[v]
\end{align*}
\]

We’ve extended the syntax and operational semantics, now it’s time to extend the type system.
First-class Continuations

First-class continuations in one slide:

\[
\begin{align*}
e & ::= \cdots | \text{letcc } x. \ e | \text{throw } e \ e | \text{cont } E \\
v & ::= \cdots | \text{cont } E \\
E & ::= \cdots | \text{throw } E \ e | \text{throw } v \ E \\
\tau & ::= \cdots | \text{cont } \tau
\end{align*}
\]
First-class Continuations

First-class continuations in one slide:

\[
\begin{align*}
e & ::= \cdots \mid \text{letcc } x.\ e \mid \text{throw } e\ e \mid \text{cont } E \\
v & ::= \cdots \mid \text{cont } E \\
E & ::= \cdots \mid \text{throw } E\ e \mid \text{throw } v\ E \\
\tau & ::= \cdots \mid \text{cont } \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \text{cont } \tau_a \vdash e_b : \tau_a & \quad \Gamma \vdash e_k : \text{cont } \tau_a \quad \Gamma \vdash e_a : \tau_a \\
\Gamma \vdash \text{letcc } x.\ e_b : \tau_a & \quad \Gamma \vdash \text{throw } e_k\ e_a : \tau \\
x \notin FV(E) & \quad \cdot, x : \tau_a \vdash E[x] : \tau \\
\Gamma \vdash \text{cont } E : \text{cont } \tau_a
\end{align*}
\]
Connection to Interpreters

A “real” (efficient, natural) interpreter for Lambda Calculus (or ML) would not be like our small-step operational semantics

▶ Would decompose/plug the whole program for each step!

Instead, maintain the decomposition incrementally

▶ With a stack \((S)\) of frames \((F)\) to remember “what to work on next”!

\[
F ::= \cdot \ e \mid v \cdot \mid (\cdot, e) \mid (v, \cdot) \mid \cdot \cdot \cdot \cdot 1 \mid \cdot \cdot \cdot \cdot 2 \mid \ \\
L(\cdot) \mid R(\cdot) \mid \text{case } \cdot \text{ of } L(x) \Rightarrow e \mid R(y) \Rightarrow e
\]

\[
S ::= [] \mid F :: S
\]

\[
e; S \rightarrow e'; S'
\]

The CK machine; one of very many “abstract machines”.
Now individual frames are explicit; one can do really weird things (but we won’t).
Living without `letcc x. e, throw e e, and cont E`

Remember, the (Untyped) Lambda Calculus could *encode* all of the features of the (Untyped) Lambda Calculus with Extensions (pairs, sums, fix).

So, Lambda Calculus w/ Extensions (without `letcc x. e, throw e e, and cont E`) isn’t any more powerful than Lambda Calculus.

Is Lambda Calculus w/ Extensions *with* `letcc x. e, throw e e, and cont E` more powerful than Lambda Calculus?
Living without `letcc x. e, throw e e, and cont E`

Remember, the (Untyped) Lambda Calculus could *encode* all of the features of the (Untyped) Lambda Calculus with Extensions (pairs, sums, fix).

So, Lambda Calculus w/ Extensions (without `letcc x. e, throw e e, and cont E`) isn’t any more powerful than Lambda Calculus.

Is Lambda Calculus w/ Extensions *with* `letcc x. e, throw e e, and cont E` more powerful than Lambda Calculus?

» Couldn’t be — Lambda Calculus is *Turing complete*
Living without letcc $x. e$, throw $e e$, and cont $E$

Remember, the (Untyped) Lambda Calculus could encode all of the features of the (Untyped) Lambda Calculus with Extensions (pairs, sums, fix).

So, Lambda Calculus w/ Extensions (without letcc $x. e$, throw $e e$, and cont $E$) isn’t any more powerful than Lambda Calculus.

Is Lambda Calculus w/ Extensions with letcc $x. e$, throw $e e$, and cont $E$ more powerful than Lambda Calculus?

- Couldn’t be — Lambda Calculus is *Turing complete*

Can we encode first-class continuations?
Living without \texttt{letcc} \(x\). \(e\), \texttt{throw} \(e\) \(e\), and \texttt{cont} \(E\)

Can we \textit{encode} first-class continuations?

Yes: Rather than adding a powerful feature, we can achieve the same effect via a \textit{whole-program translation} from the Lambda Calculus into a sublanguage (source-to-source transformation).

- No expressions with non-trivial evaluation contexts
- Every expression becomes a continuation-accepting function
- Never “return” — instead, call the current continuation
- \texttt{(Re)Introduce} \texttt{letcc} \(x\). \(e\) and \texttt{throw} \(e\) \(e\) as \(O(1)\) operations
Continuation-Passing-Style Transformation

Intuition:

- Pass the current continuation to every expression
- Represent the current continuation as a function \((\lambda z. E[z])\)
  - The initial continuation is the identity function \((\lambda z. z)\)
- To return a value, apply the current continuation to the value
  - Functions must take an explicit continuation argument
    - Function result sent to the current cont. of the function application
- \texttt{letcc k. e} and \texttt{throw e e} translated away
  - “Funny” manipulations of continuation functions

The target of the transformation is Lambda Calculus w/ Extensions, which we can further encode down to Lambda Calculus.
Continuation-Passing-Style (CPS) Transformation

A metafunction from expressions to expressions.

\[
\begin{align*}
\text{CPS} [e] & \equiv \text{CPS}_{e}[e] (\lambda z. z) \\
\text{CPS}_{e}[x] & \equiv \lambda k. k \ x \\
\text{CPS}_{e}[\lambda x. e] & \equiv \lambda k. k (\lambda k'. \lambda x. \text{CPS}_{e}[e] k') \\
\text{CPS}_{e}[e_1 \ e_2] & \equiv \lambda k. \text{CPS}_{e}[e_1] (\lambda f. \text{CPS}_{e}[e_2] (\lambda z. f \ k \ z)) \\
\text{CPS}_{e}[(e_1, e_2)] & \equiv \\
& \lambda k. \text{CPS}_{e}[e_1] (\lambda z_1. \text{CPS}_{e}[e_2] (\lambda z_2. k (z_1, z_2))) \\
\text{CPS}_{e}[e.1] & \equiv \lambda k. \text{CPS}_{e}[e] (\lambda z. k (z.1)) \\
\text{CPS}_{e}[e.2] & \equiv \lambda k. \text{CPS}_{e}[e] (\lambda z. k (z.2))
\end{align*}
\]
Properties of the CPS Transformation

- Correctness: $e$ is totally equivalent to $\text{CPS}[e] \equiv \text{CPS}_e[e]$ ($\lambda z. z$)
- If whole program has type $\tau_P$ and $e$ has type $\tau$, then $\text{CPS}_e[e]$ has type $(\tau \rightarrow \tau_P) \rightarrow \tau_P$
- Fixes evaluation order: $\text{CPS}_e[e]$ will evaluate $e$ in left-to-right call-by-value
- Other similar transformations encode other evaluation orders
- Every intermediate computation is bound to / named by a variable (helpful for compiler writers)
- For all $e$, evaluation of $\text{CPS}_e[e]$ stays in this sublanguage:

  $e ::= v | vv | vv | v(v.1) | v(v.2)$

  $v ::= x | \lambda x. e | (v, v)$

- No need for a call-stack: every call is a tail-call
  - Now the program is maintaining the evaluation context via a closure that has the next “link” in its environment that has the next “link” in its environment, ...
Encoding First-class Continuations

With the CPS transformation, `letcc x. e` and `throw e e` can become $O(1)$ operations.

\[
\begin{align*}
\text{CPS}_{e} [\text{letcc } k. e] & \equiv \lambda k. \text{CPS}_{e}[e] \ k \\
\text{CPS}_{e} [\text{throw } e_1 e_2] & \equiv \\
& \quad \lambda k. \text{CPS}_{e}[e_1] (\lambda k'. \text{CPS}_{e}[e_2] (\lambda z. k' z))
\end{align*}
\]

- `letcc` gets passed the current continuation (just as it needs)
- `throw` ignores the current continuation (just as it should)

You can also manually program in this style (fully or partially)

- Has other uses as a programming idiom . . .
CPS as an Advanced Programming Idiom

- Because CPS uses only tail calls, it avoid deep call stacks when traversing recursive data structures
  - Recall the `iter` function from HW2
- A first-class continuation can “reify session state” in a client-server interaction
  - If the continuation is passed to the client, which returns it later, then the server can be stateless.
  - Suggests CPS for web programming
- Better: tools that do the CPS transformation
  - Gives you a “prompt-client” primitive without server-side state

“Thinking in terms of CPS” is a powerful technique.
Curry-Howard Isomorphism

Recall the typing rules for \texttt{letcc} and \texttt{throw}:

\[
\begin{align*}
\Gamma, x : \text{cont } \tau_a &\vdash e_b : \tau_a \\
\Gamma &\vdash \text{letcc } x.\ e_b : \tau_a \\
\Gamma &\vdash e_k : \text{cont } \tau_a \\
\Gamma &\vdash e_a : \tau_a \\
\Gamma &\vdash \text{throw } e\ e : \tau
\end{align*}
\]

- letcc: from a \text{cont } \tau_a assumption produce a \tau_a, produce \tau_a
- throw: from a \text{cont } \tau_a and a \tau_a, produce (any) \tau

But, STLC w/ \texttt{letcc} and \texttt{throw} (and w/o \texttt{fix}) is terminating.

- Not (necessarily) inconsistent.
Curry-Howard Isomorphism

Recall the typing rules for \texttt{letcc} and \texttt{throw}:

\[
\begin{align*}
\Gamma, x : \text{cont} \; \tau_a & \vdash e_b : \tau_a \\
\Gamma & \vdash \text{letcc} \; x. \; e_b : \tau_a
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e_k : \text{cont} \; \tau_a \\
\Gamma & \vdash e_a : \tau_a \\
\Gamma & \vdash \text{throw} \; e \; e : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma, \neg p & \vdash p \\
\Gamma & \vdash p
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash \neg p \\
\Gamma & \vdash p \\
\Gamma & \vdash q
\end{align*}
\]

- \texttt{letcc} is a form of proof-by-contradiction
- \texttt{throw} is law of non-contradiction

CPS transformation corresponds to double-negation translation that maps classical proofs to intuitionistic proofs.