But first, some clean up.

Our semantics:

- $e_1 \rightarrow e'_1$
- $e_2 \rightarrow e'_2$
- $e \rightarrow e'$
- $e \rightarrow e'$

$e_1 e_2 \rightarrow e'_1 e'_2$
$v e_2 \rightarrow v e'_2$
$A(e) \rightarrow A(e')$
$B(e) \rightarrow B(e')$

$e_1 \rightarrow e'_1$
$e_2 \rightarrow e'_2$
$e \rightarrow e'$
$e \rightarrow e'$

$(e_1, e_2) \rightarrow (e'_1, e'_2)$
$(v_1, e_2) \rightarrow (v_1, e'_2)$
$e.1 \rightarrow e'.1$
$e.2 \rightarrow e'.2$

match $e$ with $Ax. e_1 | By. e_2$ → match $e'$ with $Ax. e_1 | By. e_2$

$\lambda x. e \rightarrow e[v/x]$
$(v_1, v_2).1 \rightarrow v_1$
$(v_1, v_2).2 \rightarrow v_2$

match $A(v)$ with $Ax. e_1 | By. e_2 \rightarrow e_1[v/x]$

match $B(v)$ with $Ay. e_1 | Bx. e_2 \rightarrow e_2[v/x]$
But first, some clean up.

Our semantics:
Boring rules to grind sub-expressions down:

\[
\begin{align*}
    e_1 \to e'_1 \\
    e_2 \to e'_2 \\
    e \to e' \\
    e \to e' \\
    e_1 \cdot e_2 \to e'_1 \cdot e' \quad v \cdot e_2 \to v \cdot e'_2 \\
\end{align*}
\]

\[A(e) \to A(e') \quad B(e) \to B(e')\]

\[e \to e'\]

\[E ::= \textbf{[·]} | E \cdot v | E \cdot E | (E, e) | (v, E) | E \cdot 1 | E \cdot 2 \]

\[| A(E) | B(E) | \text{(match } E \text{ with } A.x. e_1 \text{ | } B.y. e_2)\]

Evaluation contexts define where interesting work can happen:

- \(E[e]\) just means to “fill the hole” in \(E\) with \(e\).
- \(E[e]\) just means to “fill the hole” in \(E\) with \(e\).

Now we can cleanly separate our semantics:

\[
\begin{align*}
    e \to e' \text{ with 1 rule: } &
    \frac{e \to e'}{E[e] \to E[e']} \\
    e \to e' \text{ does all the “interesting work”: } &
    \frac{(\lambda x. e) \cdot v \to e'[v/x]}{(v_1, v_2).1 \to v_1} \\
    \frac{(v_1, v_2).2 \to v_2}{(v_1, v_2).2 \to v_2} \\
\end{align*}
\]

\[
\begin{align*}
    \text{match } A(v) \text{ with } A.x. e_1 | B.y. e_2 \to e_1[v/x] \\
    \text{match } B(v) \text{ with } A.y. e_1 | B.x. e_2 \to e_2[v/x] \\
\end{align*}
\]

We can do better: Separate concerns

\[
E ::= \textbf{[·]} | E \cdot v | E \cdot E | (E, e) | (v, E) | E \cdot 1 | E \cdot 2 \\
| A(E) | B(E) | \text{(match } E \text{ with } A.x. e_1 | B.y. e_2)\]

Evaluation relies on decomposition (unstapling the correct subtree)

- Given \(e\), find \(E, e_a, e'_a\) such that \(e = E[e_a]\) and \(e_a \to e'_a\)

Many possible eval contexts may match a given \(e\) ...

\[
\begin{align*}
    (\textbf{[·]})[(1, (1, (1, (1, 1)))),] & = (1, (1, (1, (1, 1)))), \\
    ((1, [\textbf{·}]})[(1, (1, (1, 1)))),] & = (1, (1, (1, (1, 1)))), \\
    ((1, (1, [\textbf{·}]})[(1, (1, 1)))),] & = (1, (1, (1, (1, 1)))), \\
    ((1, (1, (1, [\textbf{·}]})[(1, 1)))),] & = (1, (1, (1, (1, 1)))), \\
    ((1, (1, [\textbf{·}]})[(1, 1)))),] & = (1, (1, (1, (1, 1)))), \\
    \end{align*}
\]
Evaluation with evaluation contexts

\[ E ::= [·] | E e | v E | (E, e) | (v, E) | E.1 | E.2 | A(E) | B(E) | (\text{match } E \text{ with } Ax. \, e_1 | By. \, e_2) \]

Evaluation relies on decomposition (unstapling the correct subtree)

- Given \( e \), find \( E, e_a, e'_a \) such that \( e = E[e_a] \) and \( e_a \xrightarrow{P} e'_a \)

Unique Decomposition Theorem: at most one decomposition of \( e \)

- \( E \) carefully picks leftmost non-value sub-expression
- Hence eval is deterministic: at most one primitive step applies

Progress Theorem (restated): If \( e \) is well-typed, then there is a decomposition or \( e \) is a value

Evaluation contexts so far just cleanly separate the “find and plug” from the “take that step” by building an explicit \( E \)

Continuations

Now that we have defined \( E \) explicitly in our metalanguage, what if we also put it on our language

- From metalanguage to language is called reification

First-class continuations:

\[ e ::= \ldots | \text{letcc } x. \, e | \text{throw } e \, e | \text{cont } E \]
\[ v ::= \ldots | \text{cont } E \]
\[ E ::= \ldots | \text{throw } E \, e | \text{throw } v \, E \]

\[ E[\text{letcc } x. \, e] \rightarrow E[(\lambda x. \, e)(\text{cont } E)] \quad E[\text{throw } (\text{cont } E') \, v] \rightarrow E'[v] \]

- New operational rules for \( \rightarrow \) not \( \xrightarrow{P} \) because “the \( E \) matters”
- \text{letcc } x. \, e grabs the current evaluation context (“the stack”)
- \text{throw } (\text{cont } E') \, v restores old context: “jump somewhere”
- \text{cont } E not in source programs: “saved stack (value)”

Examples (exceptions-like)

\[ 1 + (\text{letcc } k. \, 2 + 3) \rightarrow^* 6 \]
\[ 1 + (\text{letcc } k. \, 2 + (\text{throw } k \, 3)) \rightarrow^* 4 \]
\[ 1 + (\text{letcc } k. \, (\text{throw } k \, (2 + 3))) \rightarrow^* 6 \]
\[ 1 + (\text{letcc } k. \, (\text{throw } k \, (\text{throw } k \, (2 + 3)))) \rightarrow^* 3 \]
Another view

If you’re confused, think call stacks:

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

- “Resume the stack’s hole” with something different or when mutable state is different
  - Else you are sure to have an infinite loop since you will later resume the stack again

Example (“time travel”)

SML/NJ has continuations. This runs and binds 10 to \( z \):

```plaintext
open SMLofNJ.Cont
val g : int cont option ref = ref NONE
val x = ref true (* avoids infinite loop *)
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
```

Is this useful?

First-class continuations are a single construct sufficient for:

- Exceptions
- Cooperative threads (including coroutines)
  - “yield” captures the continuation (the “how to resume me”) and gives it to the scheduler (implemented in the language), which then throws to another thread’s “how to resume me”
- Other crazy things
  - Often called the “goto of functional programming” — incredibly powerful, but nonstandard uses are usually inscrutable
  - Key point is that we can “jump back in” unlike boring-old exceptions

Where are we

Done:

- Redefined our operational semantics using evaluation contexts
- That made it easy to define first-class continuations
- Example uses of continuations

Now: How the heck do we implement this?

Rather than adding a powerful primitive, we can achieve the same effect via a whole-program translation into a sublanguage (source-to-source transformation)

- Every function takes extra arg: \( continuation \) says what’s next
- Never “return” — instead call current continuation w/ \texttt{result}
- \textit{Every expression becomes a continuation-accepting function}
- Will be able to reintroduce \texttt{letcc} and \texttt{throw} “for free”
CPS examples

Invariant: every function takes continuation as extra argument

let mult' x y k = ...

let mult’ x y k = k (x * y)
CPS examples

Invariant: every function takes continuation as extra argument

```ocaml
let mult' x y k = k (x * y)
let add' x y k = k (x + y)
let sub' x y k = k (x - y)
let eq' x y k = k (x = y)
```

```ocaml
let rec fact' n k = ...
```

OK, now you convert:

```ocaml
let fact n =
aux n 1
let rec aux n acc =
if n = 0 then
  acc
else
  aux (n - 1) (n * acc)
```
The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

\[ e ::= x | \lambda x. e | e e | c | e + e \]
\[ v ::= x | \lambda x. e | c \]

\[ \text{CPS}_V(v) = c \]
\[ \text{CPS}_V(x) = x \]
\[ \text{CPS}_V(\lambda x. e) = \lambda x. \lambda k. \text{CPS}_E(e) \]

To run the whole program \( e \), do \( \text{CPS}_E(e) (\lambda x. x) \)

Encoding first-class continuations

If you apply the CPS transform, then you can add \texttt{letcc} and \texttt{throw} “for free” right in the source language

\[ \text{CPS}_E(\texttt{letcc} k. e) = \lambda k. \text{CPS}_E(e) \]
\[ \text{CPS}_E(\texttt{throw} e_1 e_2) = \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \mu x_2. x_1 x_2 \]

- \texttt{letcc} gets passed the current continuation just as it needs
- \texttt{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 too...

A useful advanced programming idiom

- 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 for you
  - Gives you a “prompt-client” primitive without server-side state

- Because CPS uses only tail calls, it avoids deep call stacks when traversing recursive data structures
  - See \texttt{lec13code.ml} for this and related idioms

In short, “thinking in terms of CPS” is a powerful technique few programmers have.