Toward Evaluation Contexts

λ-calculus with extensions has many "boring inductive rules":

\[ e_1 \rightarrow e'_1 \]
\[ e_2 \rightarrow e'_2 \]
\[ e \rightarrow e' \]
\[ e \rightarrow e' \]

\[ (e_1, e_2) \rightarrow (e'_1, e'_2) \]
\[ (v_1, e_2) \rightarrow (v_1, e'_2) \]
\[ e \rightarrow e' \]
\[ e \rightarrow e' \]
\[ e \rightarrow e' \]
\[ e \rightarrow e' \]

\[ \text{match } e \text{ with } Ax. e_1 \mid By. e_2 \rightarrow \text{match } e' \text{ with } Ax. e_1 \mid By. e_2 \]

And some "interesting do-work rules":

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

\[ \text{match } A(v) \text{ with } Ax. e_1 \mid By. e_2 \rightarrow e_1[v/x] \]

\[ \text{match } B(v) \text{ with } Ay. e_1 \mid Bx. e_2 \rightarrow e_2[v/x] \]
Evaluation Contexts

Define *evaluation contexts*, which are expressions with one hole where “interesting work” is allowed to occur:

\[ E ::= \mathbb{E} \mid E \, e \mid v \, E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \mid A(E) \mid B(E) \mid \text{match } E \text{ with } A. \, e_1 \mid B. \, e_2 \]

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

- A metafunction of type \( \text{EvalContext} \rightarrow \text{Exp} \rightarrow \text{Exp} \)

Semantics: Use two judgments

- \( e \rightarrow e' \) with 1 rule:
  \[
  \frac{e \xrightarrow{p} e'}{E[e] \rightarrow E[e']}
  \]

- \( e \xrightarrow{p} e' \) with all the “interesting work”:
  \[
  \frac{E[e] \rightarrow E[e']}{E[E[e]]}
  \]
Decomposition

Evaluation relies on decomposition (unstapling the correct subtree)
- Given \( e \), find \( E, e_a, e'_a \) such that \( e = E[e_a] \) and \( e_a \xrightarrow{P} e'_a \)

Theorem (Unique Decomposition): There is at most one decomposition of \( e \)
- Hence evaluation is deterministic since at most one primitive step can apply to any expression

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

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 in one slide:

\[
e ::= \cdots \mid \text{letcc } x.\ e \mid \text{throw } e\ e \mid \text{cont } E
\]

\[
v ::= \cdots \mid \text{cont } E
\]

\[
E ::= \cdots \mid \text{throw } e\ e \mid \text{throw } v\ E
\]

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

- New operational rules for \( \xrightarrow{P} \) 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)

\[
1 + (\text{letcc } k. \ 2 + 3) \rightarrow^* 6
\]

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

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

\[
1 + (\text{letcc } k. \ (\text{throw } k \ (\text{throw } k \ 2))) \rightarrow^* 3
\]

Note: Breaks the Church-Rosser property. Under full reduction:

\[
\text{letcc } k. \ (\text{throw } k \ 1) + (\text{throw } k \ 2)) \rightarrow^* 1
\]

\[
\text{letcc } k. \ (\text{throw } k \ 1) + (\text{throw } k \ 2)) \rightarrow^* 2
\]

Example ("time travel")

Caml doesn’t have first-class continuations, but if it did:

```ocaml
let valOf x = match x with None-> failwith "" |Some x-> x
let x = ref true (* avoids infinite loop *)
let g = ref None
let y = ref (1 + 2 + (letcc k. (g := Some k); 3))
let z = if !x then (x := false; throw (valOf (!g)) 7)
then (x := false; throw (valOf (!g)) 7)
else !y
```

SML/NJ does: This runs and binds 10 to z:

```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
```

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

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
The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

\[
\begin{align*}
  e & ::= x | \lambda x. e | e + e | c | e + e \\
  v & ::= x | \lambda x. e | c
\end{align*}
\]

\[
\begin{align*}
  \text{CPS}_E(v) &= \lambda k. \text{CPS}_V(v) \\
  \text{CPS}_E(e_1 + e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. k (x_1 + x_2) \\
  \text{CPS}_E(e_1 e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda f. \text{CPS}_E(e_2) \lambda x. f x k
\end{align*}
\]

\[
\begin{align*}
  \text{CPS}_V(e) &= c \\
  \text{CPS}_V(x) &= x \\
  \text{CPS}_V(\lambda x. e) &= \lambda x. \lambda k. \text{CPS}_E(e) k
\end{align*}
\]

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

Result of the CPS transformation

- Correctness: \( e \) is equivalent to \( \text{CPS}_E(e) \lambda x. x \)
- 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 a variable (helpful for compiler writers)
- For all \( e \), evaluation of \( \text{CPS}_E(e) \) stays in this sublanguage:

\[
\begin{align*}
  e & ::= v | v v | v v v | v (v + v) \\
  v & ::= x | \lambda x. e | c
\end{align*}
\]

- Hence 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, etc.

Encoding first-class continuations

If you apply the CPS transform, then \texttt{letcc} and \texttt{throw} can become \( O(1) \) operations encodable in the source language

\[
\begin{align*}
  \text{CPS}_E(\texttt{letcc } k. e) &= \lambda k. \text{CPS}_E(e) k \\
  \text{CPS}_E(\texttt{throw } e_1 e_2) &= \lambda k. \text{CPS}_E(e_1) \lambda x_1. \text{CPS}_E(e_2) \lambda x_2. \lambda x_1 x_2 \text{ or just } x_1
\end{align*}
\]

- \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 lec13code.ml for this and related idioms

In short, "thinking in terms of CPS" is a powerful technique few programmers have