Gimme A Break (from types)

We have more to do with type systems:
- Subtyping
- Parametric Polymorphism
- Type-And-Effect Systems

But sometimes it’s more fun to mix up the lecture schedule

This lecture: Related topics that work in typed or untyped settings:
- How operational semantics can be defined more concisely
- How lambda-calculus (or PLs) can be enriched with *first-class continuations*, a powerful *control operator*
- Cool programming idioms related to these concepts
Structural Operational Semantics (again)

The rules for structural operational semantics can be classified into two types

- **structural congruence rules**, which constrain the choice of reductions that can be performed next, thus defining both the order of evaluation and whether subexpressions are evaluated lazily (let’s call these “boring” rules)
- **reduction rules**, which describe the actual computation steps (let’s call these “interesting” rules)

For example, the CBV reduction strategy for the $\lambda$-calculus was captured in the following rules:

\[
\beta\text{-reduction: } \quad \frac{e[v/x] = e'}{(\lambda x. e) \ v \rightarrow e'} \quad \text{cool!}
\]

\[
\frac{e_1 \rightarrow e'_1}{e_1 \ e_2 \rightarrow e'_1 \ e_2} \quad \text{zzz...} \quad \frac{e_2 \rightarrow e'_2}{v \ e_2 \rightarrow v \ e'_2} \quad \text{zzz...}
\]
λ-calculus with extensions has many “boring” inductive rules:

\[
\begin{align*}
& \frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2} && \frac{e_2 \rightarrow e'_2}{v e_2 \rightarrow v e'_2} && \frac{e \rightarrow e'}{A(e) \rightarrow A(e')} && \frac{e \rightarrow e'}{B(e) \rightarrow B(e')} \\
& \frac{e_1 \rightarrow e'_1}{(e_1, e_2) \rightarrow (e'_1, e_2)} && \frac{e_2 \rightarrow e'_2}{(v_1, e_2) \rightarrow (v_1, e'_2)} && \frac{e.1 \rightarrow e'.1}{e.2 \rightarrow e'.2} \\
& e \rightarrow e' \\
\end{align*}
\]

match $e$ with $A x. e_1 | B y. e_2 \rightarrow$ match $e'$ with $A x. e_1 | B y. e_2$

And some “interesting” do-work rules:

\[
\begin{align*}
& \frac{(\lambda x. e) v \rightarrow e[v/x]}{(v_1, v_2).1 \rightarrow v_1} && \frac{(v_1, v_2).2 \rightarrow v_2}{(v_1, v_2).2 \rightarrow v_2} \\
& \frac{\text{match } A(v) \text{ with } A x. e_1 | B y. e_2 \rightarrow e_1[v/x]}{\text{match } B(v) \text{ with } A y. e_1 | B x. e_2 \rightarrow e_2[v/x]} \\
\end{align*}
\]
There typically many structural congruence ("boring") rules in real-world programming languages.

It would be nice to have a more compact way to express them.

*Evaluation contexts* provide a mechanism to do just that.
Evaluation Contexts

An evaluation context $E$, sometimes written $E[\cdot]$, is a $\lambda$-term or a metaexpression representing a family of $\lambda$-terms with a special variable $[\cdot]$ called the *hole*.

If $E[\cdot]$ is an evaluation context, then $E[e]$ represents $E$ with the term $e$ substituted for the hole.
Evaluation Contexts (cont.)

*Evaluation contexts:* expressions with one hole where “interesting work” is allowed to occur

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

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

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

Semantics: Use two judgments

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

- \( e \xrightarrow{p} e' \) with all the “interesting work”:
  \[ (\lambda x. \ e) \xrightarrow{p} e[v/x] \quad (v_1, v_2).1 \xrightarrow{p} v_1 \quad (v_1, v_2).2 \xrightarrow{p} v_2 \]

  \[
  \begin{align*}
  \text{match } A(v) \text{ with } Ax. \ e_1 & \mid By. \ e_2 \xrightarrow{p} e_1[v/x] \\
  \end{align*}
  \]
Decomposition

Evaluation relies on *decomposition* (splitting or 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
Evaluation Contexts: So what?

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

- Totally equivalent step sequence
  - (made both left-to-right call-by-value)
- Just rearranged things to be more concise: Each “boring” rule became a form of $E$
- Both “work” the same way:
  - Find the next place in the program to take a “primitive step”
  - Take that step
  - Plug the result into the rest of the program
  - Repeat (next “primitive step” could be somewhere else) until you can’t anymore (value or stuck)

Evaluation contexts so far just cleanly separate the “find and plug” from the “take that step” by building an explicit $E$
Small Detour: Control Flow

Categories based on the purpose of the constructs.

- Invocation
  - Direct calls: functions, subroutines
  - Indirect calls: function pointers, class methods, closures

- Termination of Scope
  - Structured: break, break to a label, exceptions, CPS
  - Unstructured: goto, setjmp/longjmp, exit

- Selection
  - Structured: if/then/else, match, continue, switch, case
  - Unstructured: goto, computed goto, labeled entries
Control Flow (cont.)

- **Iteration**
  - Precomputed iteration space: do, foreach
  - Dynamic iteration space: for, while, recursion

- **Concurrency**
  - Manual: processes, threads, futures, coroutines
  - Automatic: constructs in concurrent/parallel frameworks for reductions
  - Communication and synchronization techniques are critical
Continuations\textsuperscript{1}

Question:

Can we use functions to represent the control flow of a program?

\textsuperscript{1}Includes material based on lecture notes by Mark Hills, Mattox Beckman, Vikram Adve, Gul Agha, and Elsa Gunter (UIUC).
Continuations

Yes, by using the concept of a continuation.

- We will augment each procedure with an additional argument — a function to which it will pass the current computational result.
- The outer procedure “returns” no result — it will be kept in the function argument
- This function argument, receiving the result, will be called the continuation.
- At its core, the continuation is just “the rest of the computation” — it tells us what we have left to do.
- Continuations can be used to model many control flow constructs
First-class Continuations

First-class continuation are a language’s ability to completely control the execution order of instructions.

They can be used to jump:

- to a function that produced the call to the current function
- or to a function that has previous exited.

You can think of them saving the state of the program, however, first-class continuations do not save program data, just the execution context.
“Say you’re in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there’s a sandwich on the counter, and all the materials used to make it are gone. So you eat it. :-)

Continuation Passing Style

Writing procedures so that they can take a continuation to which they pass on the computation result, and which return no result is called **continuation passing style** (CPS).

CPS provides a programming technique for all forms of “non-local” control flow:
- exceptions
- GOTO
- generators (e.g., yield in python)
- async (C#)

CPS turns all non-tail calls into tail calls.
- Essentially a higher order functional GOTO
Continuation Passing Style

- CPS also acts as a compilation technique to implement non-local control flow.
- Especially useful in interpreters
- Also acts as a formalization of non-local control flow in denotational semantics.
CPS Terminology

- A function is in **direct style** when it returns its result back to the caller.
- A **tail call** occurs when a function returns the result of another function call without any more computations (like in tail recursion, but not restricted to just recursive calls).
- A function is in **continuation passing style** when it passes its result to another function instead of back to its caller – essentially we pass the result *forward*, not *backward*.
Example

A simple reporting continuation:

```ml
let report x = (print_int x; print_newline( ));
```

And a function that uses it

```ml
let plusk a b k = k (a + b);; plusk 20 22 report;;
```
Example: Factorial

(* First, the non-CPS version: *)
let rec factorial n =
  if n = 0 then 1 else n * factorial (n - 1);
factorial 4;;

(* Now, define factorial with continuations *)
let rec factorial_k n k =
  if n = 0
  then k 1
  else factorial_k (n - 1) (fun m -> k (n * m));;
factorial_k 4 print_int;;

Note that the time axis does not reflect differences or similarities in the run time of the different versions.
Example: Exceptions

```ocaml
# exception Zero;;
exception Zero
# let rec list_mult_aux list =
match list with
[ ] -> 1
| x :: xs -> if x = 0 then raise Zero
else x * list_mult_aux xs;;
val list_mult_aux : int list -> int = <fun>
# let rec list_mult list =
try list_mult_aux list with Zero -> 0;;
val list_mult : int list -> int = <fun>
# list_mult [3;4;2];;
- : int = 24
# list_mult [7;4;0];;
- : int = 0
# list_mult_aux
```

Boyana Norris
CIS 624 2013, Lecture 13
Exceptions in OCaml

- The current computation is aborted;
- Control is “thrown” back up the call stack until a matching handler is found
- all intermediate calls waiting for a return value are thrown away.
Continuations as Exceptions

# let multkp m n k =
let r = m * n in
  (print_string "product result: ";
    print_int r; print_newline ();
    k r);
val multkp : int -> int -> (int -> 'a) -> 'a = <fun>
# let rec list_multk_aux list k kexcp =
  match list with
    [ ] -> k 1
  | x :: xs -> if x = 0 then kexcp 0
    else list_multk_aux xs
        (fun r -> multkp x r k) kexcp;
val list_multk_aux : int list -> (int -> 'a)
                        -> (int -> 'a) -> 'a = <fun>
# let rec list_multk list k = list_multk_aux list k k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
Exceptions, Part 2

# list_multk [3;4;2] report;;
product result: 2
product result: 8
product result: 24
24
- : unit = ()

# list_multk [7;4;0] report;;
0
- : unit = ()
Continuations in our CBV $\lambda$-Calculus

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:

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

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

- New operational rules for $\rightarrow$ not $\rightarrow_P$ 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)”
Continuations

There are two basic constructs
Examples (exceptions-like)

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

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

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

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

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

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

\[
\text{letcc } k. \ (\text{throw } k \ 1) + (\text{throw } k \ 2)) \rightarrow^* 2
\]
Refresher: Church-Rosser Theorem

When applying reduction rules to terms in the lambda calculus, the ordering in which the reductions are chosen does not make a difference to the eventual result.
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
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: Rather than adding a powerful primitive, we can achieve the same effect via a whole-program translation into a sublanguage (source-to-source transformation)

- No expressions with nontrivial evaluation contexts
- Every expression becomes a continuation-accepting function
- Never “return” — instead call the current continuation
- Will be able to reintroduce letcc and throw as $O(1)$ operations
The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

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

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

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

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

- 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

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

\[
\text{or just } x_1
\]

▶ \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