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:

\[
\begin{align*}
\beta\text{-reduction:} & \quad 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...} \\
\frac{e_2 \rightarrow e'_2}{v \ e_2 \rightarrow v \ e'_2} & \quad \text{zzz...}
\end{align*}
\]
\(\lambda\text{-calculus with extensions has many “boring” inductive rules:}\)

\[
\frac{e_1 \rightarrow e_1'}{e_1 e_2 \rightarrow e_1' e_2} \quad \frac{e_2 \rightarrow e_2'}{v e_2 \rightarrow v e_2'} \quad \frac{e \rightarrow e'}{A(e) \rightarrow A(e')} \quad \frac{e \rightarrow e'}{B(e) \rightarrow B(e')}
\]

\[
\frac{e_1 \rightarrow e_1'}{(e_1, e_2) \rightarrow (e_1', e_2)} \quad \frac{e_2 \rightarrow e_2'}{(v_1, e_2) \rightarrow (v_1, e_2')} \quad \frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \quad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}
\]

\[
\frac{e \rightarrow e'}{\text{match } e \text{ with } A x. e_1 | B y. e_2 \rightarrow \text{match } e' \text{ with } A x. e_1 | B y. e_2}
\]

And some “interesting” do-work rules:

\[
\frac{(\lambda x. e) v \rightarrow e[v/x]}{(v_1, v_2).1 \rightarrow v_1} \quad \frac{(v_1, v_2).2 \rightarrow v_2}{e \rightarrow e'}
\]

\[
\frac{\text{match } A(v) \text{ with } A x. e_1 | B y. e_2 \rightarrow e_1[v/x]}{\text{match } B(v) \text{ with } A y. e_1 | B x. e_2 \rightarrow e_2[v/x]}
\]
Toward Evaluation Contexts (cont.)

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:
  \[
  \dfrac{e \xrightarrow{p} e'}{E[e] \rightarrow E[e']}
  \]

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

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

Evaluation relies on decomposition (splitting or unstacking 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
Question:

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

\[^1\]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 continuations 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 previously 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.
The Continuation Sandwich

“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:

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

And a function that uses it

```ocaml
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 factorialk n k =
  if n = 0
    then k 1
    else factorialk (n - 1) (fun m -> k (n * m));;
factorialk 4 print_int;;
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 factorialk n k =
    if n = 0
    then k 1
    else factorialk (n - 1) (fun m -> k (n * m));;
factorialk 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
```
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

```ocaml
# 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 kexcp;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

Boyana Norris
CIS 624 2015, Lecture 13
Exceptions, Part 2

```ocaml
# 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:

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

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

- New operational rules for $\rightarrow$ not $\rightarrow^p$ because “the $E$ matters”
- letcc $x. \ e$ grabs the current evaluation context (“the stack”)
- throw (cont $E'$) $v$ restores old context: “jump somewhere”
- cont $E$ not in source programs: “saved stack (value)”
Examples (exceptions-like)

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

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

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

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

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

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

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

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

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

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

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:

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

- 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

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

- \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 (make concrete or real) 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
Continue with

- JavaScript CPS examples
- Review of evaluation contexts
- Formal definition of evaluation contexts and first-class continuations
- Continuation-passing style as a programming idiom

Introduce efficient $\lambda$-Calculus interpreters.
Continuation Passing Style: Simple Example

Factorial example in last lecture.

Also in JavaScript: http://matt.might.net/articles/by-example-continuation-passing-style/
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
Recall: 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.

Reduction semantics with evaluation contexts (RSEC) is a variant of small-step structural operational semantics (SOS) where the evaluation context may appear explicit in the term being reduced.
RSEC relies on a parsing mechanism that takes a program or a fragment $p$ and decomposes it into a context $E$ and a subprogram or fragment $e$, called a redex such that $p = E[e]$.

The inverse process, composing a redex $e$ and a context $E$ into a program or fragment $p$ is called plugging or stapling (of $e$ into $E$).
Example: IMP

<table>
<thead>
<tr>
<th>IMP evaluation contexts syntax</th>
<th>IMP language syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E ::= [\cdot] )</td>
<td>( e ::= c )</td>
</tr>
<tr>
<td>( E + e )</td>
<td>( e + e )</td>
</tr>
<tr>
<td>( E \ast e )</td>
<td>( e \ast e )</td>
</tr>
<tr>
<td>( x := E )</td>
<td>( s ::= x := e )</td>
</tr>
<tr>
<td>( E; s )</td>
<td>( s; s )</td>
</tr>
<tr>
<td>( \text{if } E \ s \ s )</td>
<td>( \text{if } e \ s \ s )</td>
</tr>
<tr>
<td>( \text{while } E \ s )</td>
<td>( \text{while } e \ s )</td>
</tr>
<tr>
<td>( \text{skip} )</td>
<td></td>
</tr>
</tbody>
</table>
Example: IMP (cont.)

Examples of correct evaluation contexts for the IMP grammar

\[ 3 + \Box \]
\[ \Box + 3 \]
\[ \Box; x := 4, \text{ where } x \text{ is any variable} \]
\[ \text{if } \Box s_1 \ s_2, \text{ where } s_1 \text{ and } s_2 \text{ are any well-formed statements} \]

Examples of incorrect evaluation contexts for the IMP grammar

\[ \Box + \Box \quad \text{– a context can have only one hole} \]
\[ x + 3 \quad \text{– a context must contain a hole} \]
\[ x := 4; \Box \quad \text{– the hole can only appear in the first statement in a seq} \]
\[ \Box := 4 \quad \text{– the hole cannot appear as the first argument of } := \]
\[ \text{if } 2 \Box x := 4 \quad \text{– the hole is only allowed in the if condition} \]
Example: IMP (cont.)

Examples of decompositions of syntactic terms into a context and a redex (here we enclose evaluation contexts in parentheses for clarity):

\[
7 = (\Box)[7] \\
3 + x = (3 + \Box)[x] = (\Box + x)[3] = (\Box)[3 + x] \\
3 + 2 \times x + 7 = (3 + \Box + 7)[2 \times x] = (\Box + 2 \times x + 7)[3] = \ldots
\]

Contexts can have various types depending on the types of their holes of their result.
Recall \( e \xrightarrow{p} e' \), where \( e; e' \) are well-formed fragments and \( E \) is any appropriate evaluation context (i.e., such that \( E[e] \) and \( E[e'] \) are well-formed programs or fragments of program).

This rule is called the *characteristic rule* of RSEC. When this rule is applied, we say that \( e \) *reduces to* \( e' \) *in context* \( E \).
Continuations

Recall that a continuation is a value that encapsulates a piece of an expression’s evaluation context.
First-Class Continuations

Revisiting exceptions\(^2\) – the semantics for exceptions reifies the control stack.

- Represents the control stack as an ordinary value.
- Saves control stack on the handler stack.
- Replaces the control stack with the saved stack.

This is cheap because every saved stack is a prefix of the control stack.

- Save a “finger” or “bookmark” on the stack. Pop back to the finger on restore.
- Similar to setjmp and longjmp in C.

\(^2\)Based on slides by David Walker, Princeton
Similar to What?

Many modern programming languages (C++, Java, C#, etc) support exceptions explicitly with a try-throw-catch statement.

ANSI-C does not. See [http://www.di.unipi.it/~nids/docs/longjump_try_trow_catch.html](http://www.di.unipi.it/~nids/docs/longjump_try_trow_catch.html).

- `int setjmp(jmp_buf env);` Returns 0 after saving a limited environment (only the stack pointer, not the full stack).
- `void longjmp(jmp_buf env, int val);` When `longjmp` is invoked with the same `jmp_buf env` variable it returns the value passed as second argument of `longjmp`.

There are 10 kinds of people in the world:

- people thinking that this is awful (and probably are asking themselves why only two cases if there are 10 kinds of people)
- people thinking that it can be amazing!
Stack Reification

But *setjmp* and *longjmp* are not safe!

- Can *setjmp* inside a procedure, then return
- Subsequent *longjmp* returns to a defunct (overwritten) stack!

These primitives promise more than they can deliver!
Can we *safely* reify control stacks without worrying about whether they’ll expire?

- Yes, because that’s what Unix does internally to switch processes.
- Yes, and we can do it at the language level rather than the OS level.

Key idea: use a *persistent* representation of the control stack.
First-Class Continuations

- Functional equivalent of GOTO
- Can be used to implement exceptions
- Can be used to build co-routines or threads
- Available in Scheme, Ruby, and SML/NJ but not Standard ML or OCaml
- Also useful as a programming abstraction for web services
Persistent and Ephemeral Data structures

Data structures in conventional imperative languages are *ephemeral* (mutable).

- Insertion into a linked list mutates the list. The old version is lost.
- Pushing onto a stack modifies the stack pointer and writes on the underlying memory. Popping writes the stack pointer.

It is difficult to avoid ephemeral data structures in these languages.
Persistent and Ephemeral Data structures

Data structures in functional languages are *persistent*.

- Inserting an element into a list yields a new list. The old version is still available.
- Stacks can be implemented so that pushing yields a new stack leaving the old stack still available.

ML supports *both* persistent and ephemeral data structures.

- Reference cells (as in HW4) are the fundamental ephemeral structure.
Ephemeral Stack Representations

Conventional runtime systems use an ephemeral (mutable) representation of the stack.

- There is only one stack active at a time.
- Push and pop destructively update the stack.

These representations prevent efficient reification of the stack.
Persistent Stack Representations

But we can use a persistent representation instead!

- For example can represent a stack as a linked list of frames.
- Persistent push and pop operations admit multiple copies of a stack.
- Rely on garbage collection to collect unused copies.

By using this, we can implement first-class continuations safely.
Recall: 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:

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

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

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

Introduce \texttt{cont} \(E\) to designate continuations.

- Values are reified control stacks.
- Two operations: \texttt{letcc} and \texttt{throw}
Informal Overview: **letcc** $x$. $e$

Seize the *current continuation*: **letcc** $x$. $e$.

- Reify the current control stack (*current continuation*)
  
  $k = \text{cont } E$

- Bind $x$ to $k$.

- Evaluate $e$.

Grab the current control point (*continuation*) for use elsewhere.
Informal Overview: \texttt{throw e_2 e_1}

Pass control to a \textit{reified} continuation: \texttt{throw e_2 e_1}.

- Evaluate \( e_1 \) to a value \( v_1 \).
- Evaluate \( e_2 \) to a continuation (stack) \( k = \text{cont } E' \).
- Pass \( v_1 \) to \( k \).

“Jump” to a given continuation, passing a value.
Informal Overview

Crucial intuition: the *current continuation* is the current control stack at the point of evaluation.

- Evaluation builds up the stack incrementally.
- The stack “unwinds” to an expression.

Remember: continuations *only* arise as reified control stacks!
Example: Simple Arithmetic Expressions

- $1 + \text{letcc}.x (2 + (\text{throw} \ x \ 3)) \mapsto_c^* 4$
  Upward use of continuations similar to exceptions where the addition of $2 + \Box$ is bypassed and discarded when we throw to $x$.

- $1 + \text{letcc}.x \ 2 \mapsto_c^* 3$
  Captured continuation need not be used, normal control flow remains in effect.

- $1 + \text{letcc}.x \ (\text{if} \ (\text{throw} \ x \ 2) \ \text{then} \ 3 \ \text{else} \ 4) \mapsto_c^* 3$
  A throw expression can occur anywhere; its type does not need to be tied to the type of the surrounding expression. This is because a throw expression never returns normally it always passes control to its continuation argument.
Example: Early Return (MinML)

Problem: multiply the integers in a list, stopping early on zero.

Solution: bind an “escape” point for the return. (In this example, for “letcc \( x \). \( e \)” we write “letcc \( x \) in \( e \)” and for “throw \( e_2 \) \( e_1 \)” we write “throw \( e_1 \) to \( e_2 \)”.)

```
fun mult_list (l: int list):int = 
  letcc ret in
  let fun mult
      nil = 1
      | mult (0::_) = throw 0 to ret
      | mult (n::l) = n * mult l
  in mult l end)
```

(binds the variable ret to the continuation of the entire letcc expression)
Another version:

```ml
fun mult_list l = 
  let fun mult
    nil ret = 1
    | mult (0::_) ret = throw 0 to ret
    | mult (n::l) ret = n * mult l ret
  in letcc ret in (mult l) ret end
```
Example: Early Return with Explicit Continuations

From last lecture you learned how to write functions using CPS.

```
fun cps_mult_list l k = 
  let fun cps_mult
      nil k0 k = k 1
    | fun cps_mult (0::_) k0 k = k0 0
    | fun cps_mult (n::l) k0 k =
      cps_mult k0 l (fn p => k (n * p))
  in cps_mult l k k end
```
Example (“time travel”)

Caml doesn’t have first-class continuations.

SML/NJ (Standard ML of New Jersey) does have first-class continuations. This runs and binds 10 to z:

```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;
```

- **callcc**— *call-with-current-continuation*: (’a cont->’a) -> ’a

  callcc f takes the current continuation (stack k) as an object and applies the function f to it. If f invokes this continuation with argument x, it is as if (callcc f) had returned x as a result.

- **throw** k a: Invoke continuation k with argument a. Note that the stack k we capture can be returned past the point in which it was in effect, hence the “time travel” analogy.
Example (Factorial)

SML/NJ Factorial with callcc

fun factorial (n: int): int =
  let
    fun aux (n: int) (k: int cont): int =
      if n = 0 then throw k 1
      else aux (n-1) (comp_fun_cont (fn (res:int) => n * res) k)
  in
    callcc (fn k => aux n k)
  end
Where are we

Done:
- Formal definition of evaluation contexts and first-class continuations
- Continuation-passing style as a programming idiom
- Persistent stack representations

Now:
- Implement an efficient lambda-calculus interpreter using little more than malloc and a single while-loop
  - Explicit evaluation contexts (i.e., continuations) is essential
  - Key novelty is maintaining the current context incrementally
  - `letcc` and `throw` can be $O(1)$ operations (homework problem)
See the code

See lec14code.ml for four interpreters where each is:

- More efficient than the previous one and relies on less from the meta-language
- Close enough to the previous one that equivalence among them is tractable to prove

The interpreters:

1. Plain-old small-step with substitution
2. Evaluation contexts, re-decomposing at each step
3. Incremental decomposition, made efficient by representing evaluation contexts (i.e., continuations) as a linked list with “shallow end” of the stack at the beginning of the list
4. Replacing substitution with environments

The last interpreter is trivial to port to assembly or C
Example

Small-step (first interpreter):

\[
\begin{array}{c}
\begin{array}{c}
A \\
\lambda a. a
\end{array} \\
\begin{array}{c}
A \\
\lambda b. b \quad \lambda c. c \quad \lambda d. d \quad \lambda e. e
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\begin{array}{c}
A \\
\lambda a. a \\
\lambda c. c
\end{array} \\
\begin{array}{c}
A \\
\lambda d. d \quad \lambda e. e
\end{array}
\end{array}
\]

Decomposition (second interpreter):

\[
\begin{array}{c}
\begin{array}{c}
R \\
E = \lambda a. a
\end{array} \\
\begin{array}{c}
L \\
\lambda b. b \quad \lambda c. c \quad \lambda d. d \quad \lambda e. e
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\begin{array}{c}
R \\
E = \lambda a. a \\
\lambda c. c
\end{array} \\
\begin{array}{c}
R \\
\lambda d. d \quad \lambda e. e
\end{array}
\end{array}
\]

Decomposition (second interpreter):

$E = \lambda a. a$

$e = \lambda b. b \quad \lambda c. c$

$E = \lambda a. a$

$e = \lambda d. d \quad \lambda e. e$

Decomposition rewritten with linked list (hole implicit at front):

$c = L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []$

$e = A(\lambda b. b, \lambda c. c)$

$c = R(\lambda c. c) :: R(\lambda a. a) :: []$

$e = A(\lambda d. d, \lambda e. e)$
Example

Decomposition rewritten with linked list (hole implicit at *front*):

\[
c = L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: [] \\
e = A(\lambda b. b, \lambda c. c)
\]

\[
c = R(\lambda c. c) :: R(\lambda a. a) :: [] \\
e = A(\lambda d. d, \lambda e. e)
\]

Some loop iterations of third interpreter:

\[
e = A(\lambda b. b, \lambda c. c) \\
c = L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\]

\[
e = \lambda b. b \\
c = L(\lambda c. c) :: L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\]

\[
e = \lambda c. c \\
c = R(\lambda b. b) :: L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\]

\[
e = \lambda c. c \\
c = L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\]

\[
e = A(\lambda d. d, \lambda e. e) \\
c = R(\lambda c. c) :: R(\lambda a. a) :: []
\]

Fourth interpreter: replace substitution with environment/closures
**The end result**

The last interpreter needs just:

- A loop
- Lists for contexts and environments
- Tag tests

Moreover:

- Function calls execute in $O(1)$ time
- Variable look-ups don’t, but that’s fixable
- Other operations, including pairs, conditionals, letcc, and throw also all work in $O(1)$ time
  - Need new kinds of contexts and values

Making evaluation contexts explicit data structures was key