CIS 624: Structure of Programming Languages

Lecture 13 — Evaluation Contexts, Continuations, Efficient Lambda Interpreters

Boyana Norris
2017

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:

\[
\begin{align*}
\beta\text{-reduction:} & \quad (\lambda x.~e)~v \rightarrow e'[v/x] \quad \text{cool!} \\
& \quad \frac{e \rightarrow e'}{e_1~e_2 \rightarrow e'_1~e'_2} \quad zzz... \\
& \quad \frac{e_2 \rightarrow e'_2}{v~e_2 \rightarrow v~e'_2} \quad zzz...
\end{align*}
\]

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.

Toward Evaluation Contexts (cont.)

\( \lambda \)-calculus with extensions has many “boring” inductive rules:

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

And some “interesting” do-work rules:

\[
\begin{align*}
(\lambda x.~e)~v \rightarrow e[v/x] & \quad (v_1, v_2).1 \rightarrow v_1 & \quad (v_1, v_2).2 \rightarrow v_2 \\
\text{match } A(e) \text{ with } A\lambda.~e_1 \mid B\lambda.~e_2 & \rightarrow \text{match } e' \text{ with } A\lambda.~e_1 \mid B\lambda.~e_2[v/x] \\
\text{match } B(e) \text{ with } B\lambda.~e_1 \mid B\lambda.~e_2 & \rightarrow \text{match } B(e'[v/x])
\end{align*}
\]

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

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

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

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

Semantics: Use two judgments

- \( e \rightarrow e' \) with 1 rule: \( e \mathbin{\downarrow} e' \)
- \( e \mathbin{\rightarrow} e' \) with all the "interesting work":

\[
\begin{align*}
& (\lambda x. \ e) \mathbin{\downarrow} \lambda v.E[v/x] & (v_1, v_2).1 \mathbin{\downarrow} v_1 & (v_1, v_2).2 \mathbin{\downarrow} v_2 \\
& \text{match } A(v) \text{ with } Ax. \ e_1 | Bx. \ e_2 \mathbin{\downarrow} \lambda v.E[e_1[v/x]]
\end{align*}
\]

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 \( e_a \mathbin{\rightarrow} e'_a \)

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

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\(^1\)

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 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.

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

CPS Terminology

- 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.

- 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
define (print_int x; print_newline( ));
```

And a function that uses it:

```ocaml
define plusk a b k = k (a + b);; plusk 20 22
```

Example: Factorial

```ocaml
(* First, the non-CPS version: *)
define factorial n =
  if n = 0 then 1 else n * factorial (n - 1)

factorial 4
```

```ocaml
let rec factorial n =
  if n = 0 then 1
  else factorial (n - 1)

let rec factorial n =  
  if n = 0 then 1 else n * factorial (n - 1)

factorial 4
```

Example: Exceptions

```ocaml
exception Zero

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

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
  (fun r -> multkp x r k) kexcp

val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

```ocaml
# let rec list_multk list k =
  if list = []
  then k 1
  else list_multk (List.tl list) (fun m -> k (m * n))

val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

```ocaml
# list_multk [7;4;0];;
- : int = 0
```

```ocaml
# list_multk [7;4;0] (fun r -> multkp x r k) kexcp
- : int = 24
```

```ocaml
# list_mult [3;4;2];
- : int = 24
```

```ocaml
# list_mult [3;4;2];
- : int = 24
```

```ocaml
# list_mult [3;4;2] 0;
- : int = 0
```

```ocaml
# list_mult [3;4;2] 0;
- : int = 0
```

```ocaml
# list_multk 0 kexcp;
```

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

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

```ocaml
let rec list_multk list k =
  if list = []
  then k 1
  else list_multk (List.tl list) (fun m -> k (m * n))

val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```
Examples (exceptions-like)

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

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

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

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):

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

\[
\begin{align*}
\text{CPS}_V(v) & = \lambda k. \text{CPS}_V(v) \\
\text{CPS}_E(v_1 + v_2) & = \lambda k. \text{CPS}_E(v_1) \lambda x.1. \text{CPS}_E(v_2) \lambda x.2. k (x_1 + x_2) \\
\text{CPS}_E(e_1 e_2) & = \lambda k. \text{CPS}_E(e_1) \lambda f. \text{CPS}_E(e_2) \lambda x. f x k \\
\text{CPS}_V(c) & = c \\
\text{CPS}_V(x) & = x \\
\text{CPS}_V(\lambda x. e) & = \lambda x. \lambda k. \text{CPS}_E(e) k
\end{align*}
\]

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:
  \[
  \begin{align*}
e & ::= v \mid v v \mid v \ v v \mid v (v + v) \\
v & ::= x \mid \lambda x. e \mid c
\end{align*}
\]
- 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 letcc and throw can become $O(1)$ operations encodable in the source language

\[
\begin{align*}
\text{CPS}_E(\text{letcc} \ k. e) & = \lambda k. \text{CPS}_E(e) k \\
\text{CPS}_E(\text{throw} e_1 e_2) & = \lambda k. \text{CPS}_E(e_1) \lambda x.1. \text{CPS}_E(e_2) \lambda x.2. x_1 x_2 \\
& \text{or just } x_1
\end{align*}
\]

- $\text{letcc}$ gets passed the current continuation just as it needs
- $\text{throw}$ 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

Evaluation Contexts, Continuations (continued)

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/

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.

Example: IMP

Examples of correct evaluation contexts for the IMP grammar

- $3 + □$
- $□ + 3$
- $□; x := 4$, where $x$ is any variable
  - if $□ s_1 s_2$, where $s_1$ and $s_2$ are any well-formed statements

Examples of incorrect evaluation contexts for the IMP grammar

- $□ + □$ – a context can have only one hole
- $x + 3$ – a context must contain a hole
- $x ::= 4 □; □$ – the hole can only appear in the first statement in a seq
- $□ ::= □$ – the hole cannot appear as the first argument of $::=$
- if $2 □ x ::= 4$ – the hole is only allowed in the if condition

Example: IMP (cont.)

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
Example: IMP (cont.)

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

\begin{align*}
7 & = (\square)[7] \\
3 + x & = (3 + \square)[x] = (\square + x)[3] = (\square)[3 + x] \\
3 + 2 \times x + 7 & = (3 + \square + 7)[2 \times x] = (\square + 2 \times x + 7)[3] = \ldots
\end{align*}

Contexts can have various types depending on the types of their holes of their result.

Evaluation Contexts: Characteristic Rule

Recall \( e \xrightarrow{\cdot} e' \), 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.

\begin{itemize}
  \item Represents the control stack as an ordinary value.
  \item Saves control stack on the handler stack.
  \item Replaces the control stack with the saved stack.
\end{itemize}

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

\begin{itemize}
  \item Save a “finger” or “bookmark” on the stack. Pop back to the finger on restore.
  \item Similar to \texttt{setjmp} and \texttt{longjmp} in C.
\end{itemize}

\(^2\)Based on slides by David Walker, Princeton

Similar to What?

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

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

\begin{itemize}
  \item \texttt{int setjmp(jmp_buf env);} \hspace{1cm} \texttt{int val}
    \begin{itemize}
      \item Returns 0 after saving a limited environment (only the stack pointer, not the full stack).
      \item \texttt{void longjmp(jmp_buf env, int val);} \hspace{1cm} \texttt{int val}
        \begin{itemize}
          \item When longjmp is invoked with the same jmp_buf env variable it returns the value passed as second argument of longjmp.
        \end{itemize}
    \end{itemize}
\end{itemize}

There are 10 kinds of people in the world:

\begin{itemize}
  \item people thinking that this is awful (and probably are asking themselves why only two cases if there are 10 kinds of people)
  \item people thinking that it can be amazing!
\end{itemize}

Stack Reification

But \texttt{setjmp} and \texttt{longjmp} are not safe!

\begin{itemize}
  \item Can \texttt{setjmp} inside a procedure, then return
  \item Subsequent \texttt{longjmp} returns to a defunct (overwritten) stack!
\end{itemize}

These primitives promise more than they can deliver!
First-Class Continuations to the Rescue

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 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 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

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 λ-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:

- $e ::= \cdots | \text{letcc } x. e | \text{throw } e | \text{cont } E$
- $v ::= \cdots | \text{throw } v E$
- $E ::= \cdots | \text{cont } E$

Informal Overview: **cont $E$**

Introduce $\text{cont } E$ to designate continuations.

- Values are reified control stacks.
- Two operations: $\text{letcc}$ and $\text{throw}$

Informal Overview: **letcc $x. e$**

Seize the current continuation: $\text{letcc } x. e$.

- 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: **throw $e_2 e_1$**

Pass control to a reified continuation: $\text{throw } e_2 e_1$.

- 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.

Example: Simple Arithmetic Expressions

- $1 + \text{letcc } x (2 + (\text{throw } x 3)) \mapsto v_4^*$
  Upward use of continuations similar to exceptions where the addition of $2 + □$ is bypassed and discarded when we throw to $x$.

- $1 + \text{letcc } x 2 \mapsto v_3^*$
  Captured continuation need not be used, normal control flow remains in effect.

- $1 + \text{letcc } x (\text{if } (\text{throw } x 2) \text{ then } 3 \text{ else } 4) \mapsto v_3^*$
  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**.

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: 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 \( e_2 \) to \( e_1 \).")

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 end

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

Example: Early Return (cont.)

Another version:

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 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 = cps_mult k0 k (fn p => k (n * p)) in cps_mult l k k end

Example: Factorial

SML/NJ Factorial with callcc

fun factorial (n: int): int = let
  fun aux (n: int) (k: int cont): 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
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 \( z \):

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 \( \text{callcc} \ f \) 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.

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
- \texttt{letcc} and \texttt{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

Decomposition (second interpreter):

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

Some loop iterations of third interpreter:

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