CS152: Programming Languages

Lecture 14 —
Evaluation Contexts
First-Class Continuations
Lambda Interpreters
Continuation-Passing Style

Dan Grossman
Spring 2011

Gimme A Break (from types)

We have more to do with type systems:

- Parametric Polymorphism
- Recursive Types
- 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 powerful *control operators*
- How lambda-calculus can be efficiently implemented
- Cool programming idioms related to these concepts

Toward Evaluation Contexts

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

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \quad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \quad \frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \quad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')}$$

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \quad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')} \quad \frac{e \to e'}{e.1 \to e'.1} \quad \frac{e \to e'}{e.2 \to e'.2}$$

$$\frac{e \to e'}{\textbf{match } e \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \to \textbf{match } e' \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2}$$

And some "interesting do-work rules":

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

$$\frac{}{\textbf{match A}(v) \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \to e_1[v/x]}$$

$$\frac{}{\textbf{match B}(v) \textbf{ with A}y.\ e_1 \mid \textbf{B}x.\ e_2 \to e_2[v/x]}$$

Evaluation Contexts

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

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\ & \mid & \mathsf{A}(E) \mid \mathsf{B}(E) \mid (\textbf{match } E \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2) \end{array}$$

Define "filling the hole" $E[e]$ in the obvious way (stapling)

- A metafunction of type EvalContext$\to$Exp$\to$Exp

Semantics: Use two judgments

- $e \to e'$ with 1 rule: $\dfrac{e \xrightarrow{p} e'}{E[e] \to E[e']}$
- $e \xrightarrow{p} e'$ with all the "interesting work":

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

$$\frac{}{\textbf{match A}(v) \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \xrightarrow{p} e_1[v/x]}$$

$$\frac{}{\textbf{match B}(v) \textbf{ with A}y.\ e_1 \mid \textbf{B}x.\ e_2 \xrightarrow{p} e_2[v/x]}$$

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

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

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:

$$\begin{array}{rcl} e & ::= & \ldots \mid \textbf{letcc } x.\ e \mid \textbf{throw } e\ e \mid \textbf{cont } E \\ v & ::= & \ldots \mid \textbf{cont } E \\ E & ::= & \ldots \mid \textbf{throw } E\ e \mid \textbf{throw } v\ E \end{array}$$

$$\frac{}{E[\textbf{letcc } x.\ e] \to E[(\lambda x.\ e)(\textbf{cont } E)]} \qquad \frac{}{E[\textbf{throw } (\textbf{cont } E')\ v] \to E'[v]}$$

- New operational rules for $\to$ not $\xrightarrow{\text{p}}$ 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)

$$1 + (\textbf{letcc } k.\ 2 + 3) \to^* 6$$

$$1 + (\textbf{letcc } k.\ 2 + (\textbf{throw } k\ 3)) \to^* 4$$

$$1 + (\textbf{letcc } k.\ (\textbf{throw } k\ (2 + 3))) \to^* 6$$

$$1 + (\textbf{letcc } k.\ (\textbf{throw } k\ (\textbf{throw } k\ (\textbf{throw } k\ 2)))) \to^* 3$$

Example ("time travel")

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

```
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)
        else !y
```

SML/NJ does: 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
```

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:

- Implement an efficent 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 4)
- Can achieve the same effect as a source-to-source transformation
  - The continuation-passing-style (CPS) transformation
  - Programming in explicit CPS-style is also a powerful idiom

See the code

See `lec14_interp.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

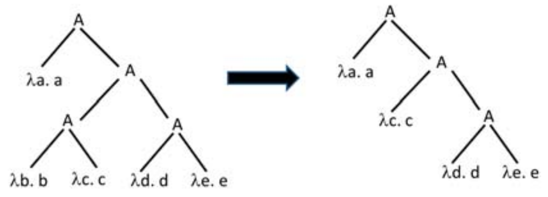
The interpreters:

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

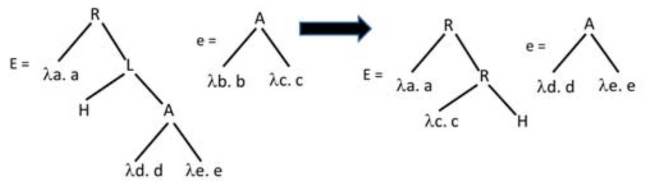
The last interpreter is trivial to port to assembly or C

Example

Small-step (first interpreter):

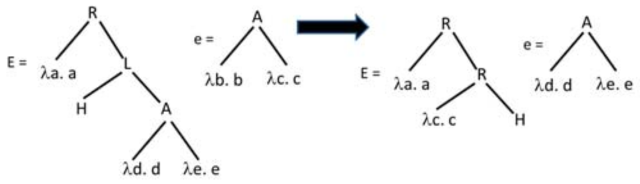


Decomposition (second interpreter):

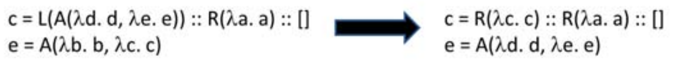


Example

Decomposition (second interpreter):



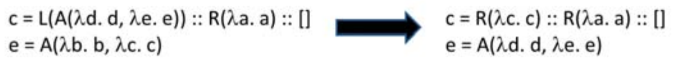
Decomposition rewritten with linked list (hole implicit at *front*):


c = L(A(λd. d, λe. e)) :: R(λa. a) :: []
e = A(λb. b, λc. c)

c = R(λc. c) :: R(λa. a) :: []
e = A(λd. d, λe. e)

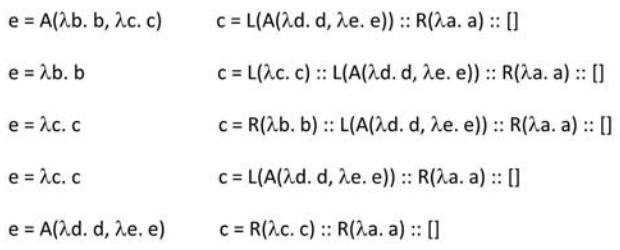
Example

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


c = L(A(λd. d, λe. e)) :: R(λa. a) :: []
e = A(λb. b, λc. c)

c = R(λc. c) :: R(λa. a) :: []
e = A(λd. d, λe. e)

Some loop iterations of third interpreter:



e = A(λb. b, λc. c)	c = L(A(λd. d, λe. e)) :: R(λa. a) :: []
e = λb. b	c = L(λc. c) :: L(A(λd. d, λe. e)) :: R(λa. a) :: []
e = λc. c	c = R(λb. b) :: L(A(λd. d, λe. e)) :: R(λa. a) :: []
e = λc. c	c = L(A(λd. d, λe. e)) :: R(λa. a) :: []
e = A(λd. d, λe. e)	c = R(λc. c) :: R(λa. 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
  - (e.g., de Bruijn indices and arrays for environments)
- Other operations, including pairs, conditionals, letcc, and throw also all work in $O(1)$ time
  - Need new kinds of contexts and values
  - Last problem of homework 4

Making evaluation contexts explicit data structures was key

Toward the CPS transform

Rather than a *fancy abstract machine* where...

- The metalanguage doesn’t need recursion (a call-stack)
- letcc/throw are easy-to-encode $O(1)$ operations

... 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

The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\ e \mid e\ e \mid c \mid e + e \\ v & ::= & x \mid \lambda x.\ e \mid c \end{array}$$

$$\begin{array}{rcl} \mathbf{CPS_E}(v) & = & \lambda k.\ k\ \mathbf{CPS_V}(v) \\ \mathbf{CPS_E}(e_1 + e_2) & = & \lambda k.\ \mathbf{CPS_E}(e_1)\ \lambda x_1.\ \mathbf{CPS_E}(e_2)\ \lambda x_2.\ k\ (x_1{+}x_2) \\ \mathbf{CPS_E}(e_1\ e_2) & = & \lambda k.\ \mathbf{CPS_E}(e1)\ \lambda f.\ \mathbf{CPS_E}(e_2)\ \lambda x.\ f\ x\ k \end{array}$$

$$\begin{array}{rcl} \mathbf{CPS_V}(c) & = & c \\ \mathbf{CPS_V}(x) & = & x \\ \mathbf{CPS_V}(\lambda x.\ e) & = & \lambda x.\ \lambda k.\ \mathbf{CPS_E}(e)\ k \end{array}$$

To run the whole program e, do $\mathbf{CPS_E}(e)\ \lambda x.\ x$

Result of the CPS transformation

- Correctness: e is equivalent to $\mathbf{CPS_E}(e)\ \lambda x.\ x$
- If whole program has type τ_P and e has type τ, then $\mathbf{CPS_E}(e)$ has type $(\tau \rightarrow \tau_P) \rightarrow \tau_P$
- Fixes evaluation order: $\mathbf{CPS_E}(e)$ 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 $\mathbf{CPS_E}(e)$ stays in this sublanguage:

$$\begin{array}{rcl} e & ::= & v \mid v\ v \mid v\ v\ v \mid v\ (v + v) \\ v & ::= & x \mid \lambda x.\ e \mid c \end{array}$$

- 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

$$\begin{array}{rcl} \mathbf{CPS_E}(\mathbf{letcc}\ k.\ e) & = & \lambda k.\ \mathbf{CPS_E}(e)\ k \\ \mathbf{CPS_E}(\mathbf{throw}\ e_1\ e_2) & = & \lambda k.\ \mathbf{CPS_E}(e_1)\ \lambda x_1.\ \mathbf{CPS_E}(e_2)\ \underbrace{\lambda x_2.\ x_1\ x_2}_{\text{or just } x_1} \end{array}$$

- **letcc** gets passed the current continuation just as it needs
- **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 `lec14_tailcalls.ml` for this and related idioms

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