CIS 624: Structure of Programming Languages

Lecture 14 — Evaluation Contexts, Continuations, Efficient Lambda Interpreters

Boyana Norris
2013
Evaluation Contexts, Continuations

Today – 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>$e + E$</td>
<td>$e + e$</td>
</tr>
<tr>
<td>$E * e$</td>
<td>$e * e$</td>
</tr>
<tr>
<td>$e * E$</td>
<td>$e * e$</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>if $E$ $s$ $s$</td>
<td>if $e$ $s$ $s$</td>
</tr>
<tr>
<td>while $E$ $s$</td>
<td>while $e$ $s$</td>
</tr>
</tbody>
</table>
Example: IMP (cont.)

Examples of **correct** evaluation contexts for the IMP grammar

□
3 + □
□ + 3
□; x := 4, where x is any variable
if □ s₁ s₂, where s₁ and s₂ 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
□ := 4 – 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.)

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

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

Contexts can have various types depending on the types of their holes of their result.
Evaluation Contexts: Characteristic Rule

Recall $e \xrightarrow{p} e'$, where $e; e'$ are well-formed fragments and $E[e] \rightarrow E[e']$, where $E$ 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\(^1\) – the semantics for exceptions \textbf{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 \textbf{prefix} of the control stack.

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

\(^1\)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.

- `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!
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.
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:

\[
\begin{align*}
  e & ::= \ldots | \lambda x. e | \text{letcc } x. e | \text{throw } e_1 e_2 | \text{cont } E \\
  v & ::= \ldots | \text{cont } E \\
  E & ::= \ldots | \text{throw } E_1 e_1 | \text{throw } v_1 E_1
\end{align*}
\]

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

- New operational rules for $\rightarrow$ not $\rightarrow_p$ because “the $E$ matters”
- $\lambda x. e$ 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)”
Informal Overview: \texttt{cont} \textit{E}

Introduce \texttt{cont} \textit{E} to designate continuations.

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

Seize the \textit{current continuation}: \textbf{letcc }x. \ e.

- Reify the current control stack (current continuation)
  \[ k = \textbf{cont } E \]
- 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 \textit{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 \textit{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 $2 + \square$ 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 \text{throw} expression can occur anywhere; its type does not need to be tied to the type of the surrounding expression. This is because a \text{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)
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
From last lecture you learned how to write functions using CPS.

```haskell
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:

```ml
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.
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):

```
A
  λa. a
  A
  λb. b  λc. c  λd. d  λe. e
```

Decomposition (second interpreter):

```
E = λa. a
  R
  L
  H
  A
  λd. d  λe. e

E = λb. b  λc. c
  A
  e

E = λa. a
  R
  H
  A
  λd. d  λe. e
```
**Example**

Decomposition (second interpreter):

\[
E = \quad \lambda a. a \\
\quad L \\
\quad H \\
\quad \lambda d. d \\
\quad \lambda e. e
\]

\[
e = \quad \lambda b. b \\
\quad \lambda c. c \\
\]

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 \textit{front}):

\[
\begin{align*}
    c &= L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: [] \\
    e &= A(\lambda b. b, \lambda c. c)
\end{align*}
\]

\[
\begin{align*}
    c &= R(\lambda c. c) :: R(\lambda a. a) :: [] \\
    e &= A(\lambda d. d, \lambda e. e)
\end{align*}
\]

Some loop iterations of third interpreter:

\[
\begin{align*}
    e &= A(\lambda b. b, \lambda c. c) \\
    c &= L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\end{align*}
\]

\[
\begin{align*}
    e &= \lambda b. b \\
    c &= L(\lambda c. c) :: L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\end{align*}
\]

\[
\begin{align*}
    e &= \lambda c. c \\
    c &= R(\lambda b. b) :: L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\end{align*}
\]

\[
\begin{align*}
    e &= \lambda c. c \\
    c &= L(A(\lambda d. d, \lambda e. e)) :: R(\lambda a. a) :: []
\end{align*}
\]

\[
\begin{align*}
    e &= A(\lambda d. d, \lambda e. e) \\
    c &= R(\lambda c. c) :: R(\lambda a. a) :: []
\end{align*}
\]

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