CIS 624: Structure of Programming Languages

Lecture 13 — Evaluation Contexts, Continuations, Efficient Lambda Interpreters

Boyana Norris
2015

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

Evaluation Contexts

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 (cont.)

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

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

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: \(E[e] \rightarrow E[e']\)
- \(e \rightarrow e'\) with all the “interesting work”:

\[
\frac{(\lambda x. \cdot e) \triangleright v \cdot e[v/x]}{(v_1, v_2). \cdot 1 \triangleright v_1 \quad (v_1, v_2). \cdot 2 \triangleright v_2} \quad \text{match } A(v) \text{ with } A. \cdot e_1 \mid B. \cdot e_2 \cdot e_1[v/x]
\]

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.
Example

A simple reporting continuation:

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

And a function that uses it:

```
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
- : int = 24
```

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

```
# factorial 4 print_int;;
```

Example: Exceptions

```
# exception Zero;;
exception Zero
# let rec list_mult_aux list =
match list with
  | x :: xs -> if x = 0 then raise Zero else list_mult_aux xs;
val list_mult_aux : 'a list -> 'a list option = <fun>
# list_mult [7;4;0];;
- : int = 0
# 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

```
# 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_mult_aux list k kexcp =
match list with
  | [] -> k 1
  | x :: xs -> if x = 0 then kexcp 0 else list_mult_aux xs (fun r -> multkp x r k) kexcp
val list_mult_aux : 'a list -> (int -> 'a) -> 'a list option = <fun>
```
Exceptions, Part 2

```ocaml
# list_multk [3;4;2] report;;
product result: 2
product result: 8
product result: 24
- : unit = ()
# list_multk [7;4;0] report;;
0
- : unit = ()
```

Boyana Norris
CIS 624 2015, Lecture 13
25

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:

\[
E ::= \ldots |
\text{letcc } x. e |
\text{throw } e e |
\text{cont } E
\]

\[
v ::= \ldots |
\text{cont } E
\]

\[
E ::= \ldots |
\text{throw } e e |
\text{throw } v E
\]

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

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

Examples (exceptions-like)

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

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

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

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

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
### Encoding first-class continuations

A metafunction from expressions to expressions

Example source language (other features similar):

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

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

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

To run the whole program \( e \), do \( \text{CPS}_E(e) \lambda x. x \)

### The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

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

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

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

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:

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

- 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.

### 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/

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.

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>$x ::= E$</td>
<td>$x ::= e$</td>
</tr>
<tr>
<td>$E; s$</td>
<td>$s ::= s$</td>
</tr>
<tr>
<td>if $E$ $s$</td>
<td>if $e$ $s$</td>
</tr>
<tr>
<td>while $E$ $s$</td>
<td>while $e$ $s$</td>
</tr>
<tr>
<td>skip</td>
<td></td>
</tr>
</tbody>
</table>

Evaluation Contexts

- 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

Examples of correct evaluation contexts for the IMP grammar

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

Examples of incorrect evaluation contexts for the IMP grammar

\[
\begin{align*}
\Box & + \Box \quad \text{– a context can have only one hole} \\
\Box & + 3 \quad \text{– a context must contain a hole} \\
x & := 4; \Box \quad \text{– the hole can only appear in the first statement in a seq} \\
\Box & := 4 \quad \text{– the hole cannot appear as the first argument of :=} \\
\Box & 2 \Box x & := 4 \quad \text{– the hole is only allowed in the if condition}
\end{align*}
\]
Examples of decompositions of syntactic terms into a context and a redex (here we enclose evaluation contexts in parentheses for clarity):

\[
7 = (□)[7] \\
3 + x = (3 + □)[x] = (□ + x)[3] = (□)[3 + x] \\
3 + 2 \times x + 7 = (3 + □ + 7)[2 \times x] = (□ + 2 \times x + 7)[3] = ...
\]

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 \( E \) 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 \( E \).

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 \texttt{setjmp} and \texttt{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 \texttt{try-throw-catch} statement.

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

▶ \texttt{int setjmp(jmp_buf env)}; Returns 0 after saving a limited environment (only the stack pointer, not the full stack).
▶ \texttt{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!
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.

---

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 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:

\[ e ::= \ldots \mid \text{letcc } x . \ e \mid \text{throw } e \mid \text{cont } E \]

\[ v ::= \ldots \mid \text{throw } v \ E \]

\[ E ::= \ldots \mid \text{letcc } x . \ e \]

\[ v \ E \]

\[ E \rightarrow \ E[(\lambda x . \ e)(\text{cont } E)] \]

\[ E[\text{throw } (\text{cont } E') v] \rightarrow E'[v] \]

- New operational rules for \( \rightarrow \) 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)”

Informal Overview: \text{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: \text{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

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

Upward use of continuations similar to exceptions where the addition of \( 2 + □ \) is bypassed and discarded when we throw to \( x \).

\[ 1 + \text{letcc } x . 2 \rightarrow^* 3 \]

Captured continuation need not be used, normal control flow remains in effect.

\[ 1 + \text{letcc } x . (\text{if } (\text{throw } x 2) \text{ then } 3 \text{ else } 4) \rightarrow^* 3 \]

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 \texttt{letcc } \texttt{x } e \texttt{" we write \texttt{letcc x in } e \texttt{" and for \"throw e}\texttt{e } e \texttt{" we write \"throw e to } e \texttt{".} )

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)

Boyana Norris  
CIS 624 2015, Lecture 13  
61

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 

Boyana Norris  
CIS 624 2015, Lecture 13  
62

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 

Boyana Norris  
CIS 624 2015, Lecture 13  
65

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;  

boyana norris

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

Boyana Norris  
CIS 624 2015, Lecture 13  
65

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 \texttt{O}(1) operations (homework problem)

Boyana Norris  
CIS 624 2015, Lecture 13  
66
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):

\[
\begin{align*}
\text{e} &= \lambda a. a \\
\text{c} &= \lambda b. b \\
\text{f} &= \lambda c. c \\
\end{align*}
\]

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

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

Some loop iterations of third interpreter:

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

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