# CSE-505: Programming Languages

## Lecture 14 — Efficient Lambda Interpreters

Zach Tatlock
2015

---

## Where are we

Done:

- Formal definition of evaluation contexts and first-class continuations
- Continuation-passing style as a programming idiom
- The CPS transform

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 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:

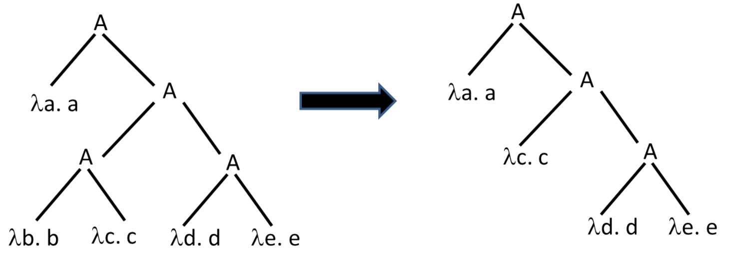
1. Plain-old small-step with substitution
2. Evaluation contexts, re-decomposing at each step
3. Incremental decomposition, made efficient by representing evaluation contexts (i.e., continuations) as a linked list with "shallow end" of the stack at the beginning of the list
4. Replacing substitution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

The last interpreter is trivial to port to assembly or C

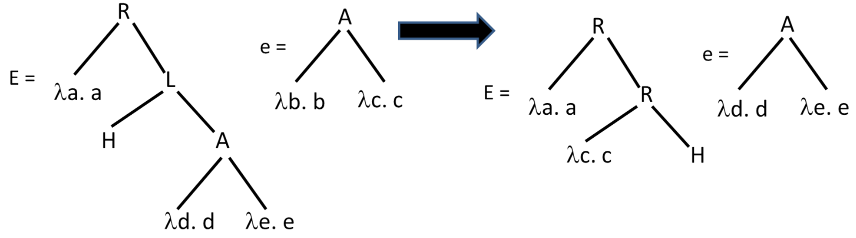
---

## Example

Small-step (first interpreter):

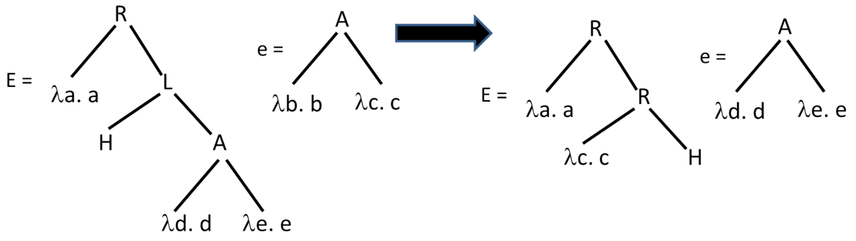


Decomposition (second interpreter):



# 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
  - Left as a homework exercise as a way to understand the code

Making evaluation contexts explicit data structures was key