Lecture 8: Alternative Implementations of the λ-calculus

Steven Holtzen
`s.holtzen@northeastern.edu`

CS4400/5400 Fall 2024

1 Environment Passing Semantics

- Similar to the `let`-lang, now we want to have *efficient implementations of the* λ*-calculus*.
- Recall the environment-passing semantics for the `let` language, which had the shape $\langle e, \rho \rangle \Downarrow v$ where ρ is an environment (let's assume this language has only let-bindings, identifiers, and Booleans for simplicity):

$$\frac{\langle e_1, \rho \rangle \Downarrow v_1 \qquad \langle e_2, \rho[x \mapsto v_1] \rangle \Downarrow v_2}{\langle \texttt{(let x } e_1 \ e_2\texttt{)}, \rho \rangle \Downarrow v_2} \text{(E-Let)} \qquad \frac{x \in \rho \qquad \rho[x] = v}{\langle x, \rho \rangle \Downarrow v} \text{(E-Id)}$$

$$\frac{}{\texttt{true} \Downarrow \texttt{true}} \text{(E-True)} \qquad \frac{}{\texttt{false} \Downarrow \texttt{false}} \text{(E-True)}$$

- This environment-passing semantics had a big benefit for the `let`-language: it significantly sped it up, since it did not require repeatedly substituting into subterms
- Let's develop a similar environment-passing approach to running λ-calculus terms

2 A First Attempt and Environment-Passing for the λ-calculus

- Here is a **first attempt**. Let's consider a λ-calculus with numbers. Here is the syntax in Racket:

  ```
  ;;; type lexpr =
  ;;;   | ident of string
  ;;;   | lam of string × lexpr
  ;;;   | app of expr × lexpr
  (struct ident (s) #:transparent)
  (struct lam (id body) #:transparent)
  (struct app (e1 e2) #:transparent)
  (struct num (n) #:transparent)
  ```

  The first thing we might try to do is copy the structure above, and evaluate λ-calculus. Here are the inference rules, which look quite similar to the `let`-language with environment-passing:

$$\frac{x \in \rho \qquad \rho[x] = v}{\langle x, \rho \rangle \Downarrow v}\ (\text{E-IDENT}) \qquad \frac{}{\langle \lambda x.e, \rho \rangle \Downarrow \lambda x.e}\ (\text{E-LAM}) \qquad \frac{}{\langle n, \rho \rangle \Downarrow n}\ \text{E-NUM}$$

$$\frac{\langle e_1, \rho \rangle \Downarrow \lambda x.e_b \qquad \langle e_2, \rho \rangle \Downarrow v_2 \qquad \langle e_b, \rho[x \mapsto v_2] \rangle \Downarrow v'}{\langle (e_1\ e_2), \rho \rangle \Downarrow v'}\ (\text{E-APP})$$

- Let's implement these rules in Racket:

  ```
   ;;; type value =
  ;;;   | vlam of string × lexpr × environment
  ;;;   | vnum of number
  (struct vlam (id body) #:transparent)
  (struct vnum (n) #:transparent)

  ;;; eval–lam–env : lexpr → env → value
  (define (eval-lam-env e env)
    (match e
      [(ident x)
       (if (hash-has-key? env x)
           (hash-ref env x)
           (error "unbound identifier"))]
      [(lam id body)
       (vlam id body)]
      [(num n) (vnum n)]
      [(app e1 e2)
       (match (eval-lam-env e1 env)
         [(vlam id body)
          (let× [(arg-v (eval-lam-env e2 env))
                 (extended-env (hash-set env id arg-v))]
            (eval-lam-env body extended-env))])]))
  ```

3 Why the First Attempt is Wrong

- Our goal is for our environment-passing semantics to behave exactly the same as our substitution semantics for the λ-calculus
- At first it might appear that this is the case. For example, this derivation tree looks fine:

$$\dfrac{\dfrac{}{\langle \lambda x.x, \{\}\rangle \Downarrow \lambda x.x}\ \text{E-LAM} \qquad \dfrac{}{\langle 5, \{\}\rangle \Downarrow 5}\ \text{E-NUM} \qquad \dfrac{}{\langle x, \{x \mapsto 5\}\rangle \Downarrow 5}\ \text{(E-IDENT)}}{\langle ((\lambda x.x)\ 5), \{\}\rangle \Downarrow 5}\ \text{E-APP}$$

- However, there is a subtle issue with scope that this approach to environment-passing is not accounting for.
- **Pause**: Can you come up with a program that evaluates differently in the substitution and environment-passing semantics for the λ-calculus?
- The issue arises with **capture**. Consider the following simple Racket program:

```
> (define weird-fun (let ([x 20]) (lambda (y) x)))
> (weird-fun 30)
20
```

  In this example, the outer-scope variable `x` is **captured** by `weird-fun`: even though the local variable `x` has gone out of scope, its value is held inside of the local λ term
- Let's see how our example semantics fail to handle this scenario with capturing. Here is a small λ-term that should cause issues:

$$(((\lambda x.(\lambda y.x))\ 20)\ 30)$$

  We can write this as a lambda term and try to run it using our evaluator:

```
> (eval-lam-env (app (app (lam "x" (lam "y" (ident "x"))) (num 20)) (num 30)) (
   hash))
. . unbound identifier
```

- Why did this happen? We can try to a derivation tree to see:

$$\dfrac{\dfrac{\dfrac{}{\langle (\lambda x.(\lambda y.x)), \{\}\rangle \Downarrow (\lambda x.(\lambda y.x))} \qquad \dfrac{}{20 \Downarrow 20} \qquad \dfrac{}{\langle (\lambda y.x), \{x \mapsto 20\}\rangle \Downarrow \lambda y.x}}{\langle ((\lambda x.(\lambda y.x))\ 20), \{\}\rangle \Downarrow \lambda y.x}\ \text{E-APP} \qquad \dfrac{}{30 \Downarrow 30} \qquad \dfrac{}{\langle x, \{y \mapsto 30\}\rangle \Downarrow^{?} ???}}{\langle (((\lambda x.(\lambda y.x))\ 20)\ 30), \{\}\rangle \Downarrow^{?} 20}$$

- Look at the above tree carefully. The problem is that these inference rules get stuck! The lambda term $((\lambda x.(\lambda y.x))\ 20)$ evaluates to $\lambda y.x$. Then, we try to call $\lambda y.x$ with the argument 30. This results in an unbound identifier error because nothing is ever put into the environment for x!
- Step through your interpreter to see this happening.

4 Closures

- The above exercise is an important illustration of how careful we need to be about scope!
- In order to get scope right with environments, we need to deal carefully with *captured variables*. To do this, a λ-term needs to keep track of the local variables that are in scope.
- There are many possible solutions to this problem of scope. The one we will explore here is to *update our rule for evaluating lambda terms* to make them hold onto the local environment when they are evaluated. We will call the local environment associated with a lambda a **closure**.
- As always, whenever we design a language, we write down its semantics in 3 ways: once in English, once in inference rules, and once in Racket code. We do this because each one has its strengths and weaknesses. Inference rules are concise and precise and let us draw derivation trees, but tend to hide detail; English is difficult to be precise, but is good at conveying intuition; Racket code is runnable and can be tested, but tends to contain a lot of extraneous information.
- This time let's start by seeing them as inference rules, and then explain the rules in English. Here are our new rules:

$$\frac{x \in \rho \qquad \rho[x] = v}{\langle x, \rho \rangle \Downarrow v}\ (\text{E-Ident}) \qquad \frac{}{\langle \lambda x.e, \rho \rangle \Downarrow (\lambda x.e, \rho)}\ (\text{E-Lam}) \qquad \frac{}{\langle n, \rho \rangle \Downarrow n}\ \text{E-Num}$$

$$\frac{\langle e_1, \rho \rangle \Downarrow (\lambda x.e_b, \rho_C) \qquad \langle e_2, \rho \rangle \Downarrow v_2 \qquad \langle e_b, \rho_C[x \mapsto v_2] \rangle \Downarrow v'}{\langle (e_1\ e_2), \rho \rangle \Downarrow v'}\ (\text{E-App})$$

- These rules are *extremely tricky*. We need to walk through them very slowly. One of the keys to understanding these rules is to consider the alternatives and why they *don't* work.
- Let's look at each rule and try to explain it in English:
  - The base case rules for identifiers and numbers are unchanged.
  - The E-Lam rule evaluates to a *pair* $(\lambda x.e, \rho)$. This pair holds the current environment ρ as well as the lambda itself. This means we will need to extend our set of values: now, lambdas evaluate to a special new kind of value that holds both a lambda and an environment!
  - The E-App rule first (1) evaluates e_1 to a pair $(\lambda x.e_b, \rho_C)$ where ρ_C is the closure and e_b is the body of the lambda. Then, it evaluates the argument e_2 to a value v_2. Then, it evaluates λ_b *in the closure extended by x mapped to v_2*.
- **Note**: you will almost certainly encounter closures because they are how first-class functions are implemented in practice in many languages. For example, in JavaScript, closures are ubiquitous in code with callbacks. Closures can be a source of memory inefficiency and resource leaks, since it's possible that a resource is "held onto" by a local function. So, closures really do affect the way that you program, and you should be aware of them and how they work as a practicing programmer.

5 A Correct Implementation of Environment-Passing for the λ-calculus

Now we are ready to give a correct implementation in Racket using the above correct inference rules:

```
#lang racket
(require rackunit)

;;; type lexpr =
;;;   | ident of string
;;;   | lam of string × lexpr
;;;   | app of expr × lexpr
(struct ident (s) #:transparent)
(struct lam (id body) #:transparent)
(struct app (e1 e2) #:transparent)
(struct num (n) #:transparent)
(struct add (e1 e2) #:transparent)

;;; type value =
;;;   | vlam of string × lexpr × environment
;;;   | vnum of number
(struct vlam (id body closure) #:transparent)
(struct vnum (n) #:transparent)

;;; get-num : value → number
;;; extracts a number from a value or errors
(define (get-num v)
  (match v
    [(vnum n) n]
    [_ (error "not a number")]))

;;; eval-lam-env : lexpr → value
(define (eval-lam-env e env)
  (match e
    [(ident x)
     (if (hash-has-key? env x)
         (hash-ref env x)
         (error "unbound identifier"))]
    [(lam id body)
     (vlam id body env)]
    [(num n) (vnum n)]
    [(add e1 e2)
     (vnum (+ (get-num (eval-lam-env e1 env))
              (get-num (eval-lam-env e2 env))))]
    [(app e1 e2)
     (match (eval-lam-env e1 env)
       [(vlam id body closure)
        (let× [(arg-v (eval-lam-env e2 env))
               (extended-env (hash-set closure id arg-v))]
          (eval-lam-env body extended-env))])]))
```

- Let's see if this new interpreter works on our challenging example from earlier:

```
> (eval-lam-env (app (app (lam "x" (lam "y" (ident "x"))) (num 20)) (num 30)) (
    hash))
(vnum 20)
```

6 Alternatives to Call-by Value: Call-by-Name

- So far in class we've been exploring the call-by-value evaluation scheme for the λ-calculus. In this scheme, all arguments are evaluated *before* functions are called.
- This is a class about language design and implementation, so it's interesting and educational to explore the consequences of alternative language design decisions and how they affect our languages.
- There is an alternative evaluation scheme called **call-by-name** where arguments are not evaluated
- The following are the substitution-based big-step semantics rules for call-by-name:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ (\textsc{E-Lam}) \qquad \frac{e_1 \Downarrow \lambda x.e_{\text{body}} \qquad e_{\text{body}}[x \mapsto e_2] \Downarrow v}{(e_1\ e_{\text{arg}}) \Downarrow v}\ (\textsc{E-App})$$

- Call-by-name semantics delays evaluating arguments; for this reason, they are sometimes called *lazy semantics*. Some programming languages (like Haskell) use these lazy semantics (though, they do some optimizations to avoid recomputing arguments unnecessarily).
- Here is a small implementation of call-by-name for λ-calculus in Racket, which doesn't require changing our substitution semantics but does require changing our evaluator:

```
;;; type expr =
;;;   | ident of string
;;;   | lam of string × expr
;;;   | app of expr × expr
(struct ident (s) #:transparent)
(struct lam (id body) #:transparent)
(struct app (e1 e2))

;;; subst : expr → string → expr → expr
;;; performs the substitution e1[x |→ e2] with lexical scope
(define (subst e1 id e2)
  (match e1
    [(ident x)
     (if (equal? x id) e2 (ident x))]
    [(lam x body)
     (if (equal? x id)
         (lam x body)  ; shadowing case; do nothing
         (lam x (subst body id e2))  ; non-shadowing case
         )]
    [(app f arg)
     (app (subst f id e2) (subst arg id e2))]))

;;; eval : expr → expr
;;; evaluates an expression to an expression in call-by-name semantics
(define (eval e)
  (match e
    [(ident x) (error "unbound ident")]
    [(lam id x) (lam id x)]
    [(app e1 e2)
     (match (eval e1)
       [(lam id body)
        (eval (subst body id e2))])]))
```

7 Consequences of Call-By-Name

- A logical question you might ask now is: Are there any programs that behave differently for call-by-value vs. call-by-name? (This question will be important next class when we discuss recursion combinators)
- These semantics may seem so similar that they must be identical, but they aren't! How can we see that?
- Recall last lecture that we discussed *terminating behavior* as a way to differentiate the behavior of two programs. So, *what is a program that terminates for the call-by-value semantics and doesn't for the call-by-name semantics*?
- Here is a simple program that has different terminating behavior for these two different semantics, calling the identity function with Ω (recall, $\Omega = ((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$ is the term that never terminates):

$$((\lambda x.x)\ \Omega)$$

- See for yourself why this program behaves differently for call-by-value vs. call-by-name
- Fun-fact: the set of programs that terminates for call-by-name is a strict superset of the set of programs that terminates in call-by-value (meaning, any program that terminates in call-by-value must also terminate in call-by-name)! Think for yourself about why this might be the case.