Lecture 5: The λ-calculus

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 First-Class Functions

- In this lecture we will add an important new feature to our increasingly-rich mini-languages: *first-class functions* (meaning, functions can be values).
- We've already seen an example of using functions as values in Racket; we declared these using the `lambda` keyword. For example:

  ```
  > (lambda (x) (+ x 1))
  #<procedure>
  > ((lambda (x) (+ x 1)) 10)
  11
  ```

- A lambda term `(lambda (x) e)` has two syntactic components: an **argument**, which is an identifier (in the above example, `x` is the argument); and a **body** (in the above example, the body is `(+ x 1)`).
- A lambda term is a function. In Racket we can call a lambda using the syntax $(\texttt{e}_1\ \texttt{e}_2)$ where $\texttt{e}_2$ is the **argument** to the function $\texttt{e}_1$. The syntax $(\texttt{e}_1\ \texttt{e}_2)$ is called an **application**.
- Let's give a semantics to these two syntactic forms. Lambdas are values, so they evaluate to themselves:

$$\frac{}{\texttt{(lambda (x) e)} \Downarrow \texttt{(lambda (x) e)}}\ (\text{E-Lam})$$

- A function application is a bit more involved. We will define it again in terms of substitution, similar to `let`. To evaluate an application $(\texttt{e}_1\ \texttt{e}_2)$ we:
  1. Evaluate $\texttt{e}_1$ to a lambda term $(\texttt{lambda (x)}\ \texttt{e}_{\texttt{body}})$
  2. Evaluate $\texttt{e}_2$ to some value v
  3. Return the result of running $\texttt{e}_{\texttt{body}}$ with v substituted for x

  In inference rules:

$$\frac{e_1 \Downarrow (\texttt{lambda (x)}\ \texttt{e}_{\texttt{body}}) \qquad e_{\text{arg}} \Downarrow v_{\text{arg}} \qquad e_{\text{body}}[x \mapsto v_{\text{arg}}] \Downarrow v}{(e_1\ e_{\text{arg}}) \Downarrow v}\ (\text{E-App})$$

2 Let and Lambda

- Lambdas are a remarkably expressive tool for describing programming language features. For instance, we can describe `let` using `lambda`.
- This isn't too hard to see from a small example. Consider the following two equivalent programs:

  ```
  > (let ([my-var 10]) my-var)
  10
  > ((lambda (my-var) my-var) 10)
  10
  ```

- Observe that `let` is a *special kind of lambda term*: in general, we can always translate a let-binding `(let ([id e1]) e2)` into a special lambda application `((lambda (id) e2) e1)`.
- This is a small example of how language features can be represented using functions: we will see many more
- It is often the case that one programming language feature can be represented by directly translating it into a subset of the language that does not use that feature; we call this process **desugaring**.
- The above example illustrates how we can desugar `let` into `lambda`.
- Why do we also have `let` if we can simply express it using `lambda`? One reason is that the `let` form helps make the program's intent clearer: it is easy to identify the argument and the body, which helps understand programs.

3 Substitution and Semantics for Lambda Terms

- Now let's discuss how substitution is implemented for `lambda` terms. Similar to the `let` language, our goal is to implement a lexical scoping strategy where an identifier always refers to its inner-most binding in the abstract syntax tree.

- Racket's implementation of `lambda` obeys lexical scope; let's explore some example racket programs to get a feel for how lexical scope works with `lambda`-terms.

```
> (define prog1 (let ([x 20])
    (lambda (x) (+ x 1))))
> (prog1 1)
2
```

- Notice how in `prog1`, the inner-most `x` is the argument to the lambda, so it does not refer to the constant `20`.

- Let's keep exploring more examples of how scope and substitution works in lambda terms. What happens if we refer to a variable defined outside of a lambda term inside of that term?

```
> (define prog2
  (let ([v 15])
    (lambda (x) (+ x v))))
> (prog2 20)
35
```

- OK, so we have observed another scoping rule: lambda terms can refer to variables outside of their body, as long as those variables aren't shadowed by some argument to a lambda.

- Now for an interesting case: what happens if we return a lambda term that refers to some in-scope variable? For instance:

```
> (define make-adder
    (lambda (to-add)
      (lambda (arg) (+ to-add arg))))
> (define add-10 (make-adder 10))
> (add-10 20)
30
```

- What's going on in this example? To understand it, let's break it down into the individual applications. First, we call `(make-adder 10)`. What is this term? It is the result of substituting in `10` for `to-add` in the `make-adder` body, i.e.:

```
  (lambda (arg) (+ 10 arg))
```

- Notice how this local variable `to-add` *escaped the scope in which it was initially defined*: it was "captured" by the body of the lambda term when it was substituted in. This is why we don't get an "unbound identifier" error when we run this program even though `to-add` has gone out of scope when we invoke `add-10`.

- **Note**: If you are ever wonder what a particular scoping rule is, plug the program into Racket and see what it does! Come up with small examples that illustrate specific edge-cases you are wondering about.

4 Syntax and Semantics of the λ-calculus

- Now let's gain a deeper understanding of `lambda` terms by implementing them ourselves.
- As usual, to study a new feature, we make a very tiny language to study it in isolation. This language will only have three syntactic terms: lambda terms, lambda application, and identifiers. This tiny language is called the **λ-calculus**:

Listing 1: Syntax of λ-calculus

```
;;; 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))
```

- We will use a convenient surface syntax for λ-calculus: the syntax $\lambda x.e$ denotes a lambda term with argument x and body e, and the syntax $(e_1\ e_2)$ denotes application.
- Now we can give the big-step semantics for the λ-calculus:

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}\ (\text{E-LAM}) \qquad \frac{e_1 \Downarrow \lambda x.e_{\text{body}} \qquad e_{\text{arg}} \Downarrow v_{\text{arg}} \qquad e_{\text{body}}[x \mapsto v_{\text{arg}}] \Downarrow v}{(e_1\ e_{\text{arg}}) \Downarrow v}\ (\text{E-APP})$$

- This semantics is called the **call-by-value semantics** because the argument to each lambda term is run before it is substituted.
- Note that, like the `let`-language, there is no inference rule for an unbound identifier (meaning, this would be a runtime error).
- We call the set of unbound identifiers for a lambda term its **free variables**. For instance, the following lambda term has a free variable y:

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

- λ-terms that have no free variables are called **closed terms**. Conversely, a λ-term with a free variable is called an **open term**.
- Here is a small parser for the lambda calculus:

```
;;; parse-sexpr: sexpr → expr
;;; parsers an s-expression into a lambda expression
;;;   expr ::= (lamda id <expr>) | id | (<expr> <expr>)
(define (parse-sexpr s)
  (match s
    [(list l id e)
     (lam (symbol→string id) (parse-sexpr e))]
    [(list e1 e2)
     (app (parse-sexpr e1) (parse-sexpr e2))]
    [id (ident (symbol→string id))]))
```

5 Substitution for λ-calculus

- Substitution for the λ-calculus is quite similar to the `let`-language
- We will make an assumption here to simplify the situation: we will assume that we are only evaluating closed terms.[1] If we only evaluate closed terms, then we define substitution as follows:

$$x[y \mapsto e] = \begin{cases} x & \text{if } x \neq y \\ e & \text{if } x = y. \end{cases} \tag{1}$$

$$(e_1\ e_2)[x \mapsto e_3] = (e_1[x \mapsto e_3]\ e_1[x \mapsto e_3]) \tag{2}$$

$$(\lambda\ x.e)[y \mapsto e] = \begin{cases} \lambda x.e & \text{if } x = y \\ \lambda x.e[y \mapsto e] & \text{otherwise.} \end{cases} \tag{3}$$

- Now we can give a Racket implementation of this substitution:

```
;;; 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))]))
```

[1]There are technical reasons for why we make this assumption based on *capture avoidance*. We won't discuss them here since they are not relevant to us yet; if you are curious, see Chapter 5 of *Types and Programming Languages* for a very detailed discussion.

6 Implementation of call-by-value substitution semantics for λ-calculus

- Now we are ready to implement the semantics for the λ-calculus using our above substitution function:

```
;;; eval : expr → expr
;;; evaluates a closed expression to a closed expression
(define (eval e)
  (match e
    [(ident x) (error "unbound ident")]
    [(num n) (num n)]
    [(lam id x) (lam id x)]
    [(app e1 e2)
     (match (eval e1)
       [(lam id body)
        (let× [(arg-v (eval e2))
               (subst-body (subst body id arg-v))]
          (eval subst-body))])]))
```

- We should test our implementation on some small examples to make sure it works:

```
> (eval (parse-sexpr '((lambda x x) (lambda y y))))
(lam "y" (ident "y"))
> (eval (parse-sexpr '((lambda x x) (lambda x x))))
(lam "x" (ident "x"))
; check for substitution under application
(check-equal? (subst app-lam "z" id-lam)
              (parse-sexpr '((lambda x x) (lambda y (lambda x x)))))
; check for substitution under application
(check-equal? (subst (parse-sexpr '(x x)) "x" (parse-sexpr '(lambda x (x x
    ))))
              (parse-sexpr '((lambda x (x x)) (lambda x (x x)))))
```

7 Running some λ-calculus programs

- Let's run a few λ-calculus programs by hand and see what they do.

$$\dfrac{\dfrac{}{\lambda x.x \Downarrow \lambda x.x} \qquad \dfrac{}{(\lambda y.y) \Downarrow (\lambda y.y)} \qquad \dfrac{}{x[x \mapsto (\lambda y.y)] \Downarrow (\lambda y.y)}\ (\text{E-L}{\small\text{AM}})}{((\lambda x.x)\ (\lambda y.y)) \Downarrow (\lambda y.y)}\ \text{E-A}{\small\text{PP}}$$

- Here is another example:

$$\dfrac{\dfrac{}{(\lambda x.\lambda y.y) \Downarrow (\lambda x.\lambda y.y)} \qquad \dfrac{}{(\lambda z.z) \Downarrow (\lambda z.z)} \qquad \dfrac{}{(\lambda y.y)[x \mapsto (\lambda z.z)] \Downarrow (\lambda y.y)}}{((\lambda x.\lambda y.y)\ (\lambda z.z)) \Downarrow (\lambda y.y)}\ \text{E-A}{\small\text{PP}}$$

  Pause: In English, what would you say this little program does? It "forgets its first argument".

- In-class exercise: let's extend our lambda calculus with numbers and addition; they behave in the familiar ways. Let's try running some programs involving those.

8 Programs that do not terminate: Ω

- It may seem at first that the λ-calculus is so simple and restricted that it cannot represent any useful programs.
- This is not the case: in fact, the λ-calculus is a *Turing-complete language*! This means that the λ-calculus is capable of representing all Racket programs, all C programs, etc. This is the so-called *Church-Turing thesis*.
- We will see more of this in upcoming lectures, but here is a taste. We know it's possible to write Racket programs that don't terminate. So, if the the λ-calculus is as expressive as Racket is, then it must also be possible to write λ-calculus programs that don't terminate. What is an example of such a program?
- You probably won't come up with it yourself, it's surprisingly tricky. Let's build it in stages. First, let's define a lambda-term ω that performs a *self-call*: it calls its argument on itself, like so:

$$\omega = \lambda x.(x\ x)$$

- Clearly a program consisting solely of ω terminates right away: lambdas are values. However, *what happens if we call ω with itself as an argument*?

$$\Omega = (\omega\ \omega)$$

- We can plug this program into our λ-calculus implementation and see that indeed it does not terminate:

```
> (define omega (parse-sexpr '((lambda x (x x)) (lambda x (x x)))))
> (eval omega)
... runs forever
```

- What's going on with this program? We can see more by attempting to run it by hand:

$$\frac{\dfrac{}{(\lambda x.(x\ x)) \Downarrow (\lambda x.(x\ x))} \qquad \dfrac{}{(\lambda x.(x\ x)) \Downarrow (\lambda x.(x\ x))} \qquad \dfrac{?}{(x\ x)[x \mapsto \lambda x.(x\ x)] \Downarrow ?}}{(\lambda x.(x\ x))\ (\lambda x.(x\ x)) \Downarrow ?}$$

- Uh oh! Look carefully at the term $(x\ x)[x \mapsto (\lambda x.(x\ x))]$: this substitution is itself equal to Ω! So, in order to evaluate Ω, we must evaluate Ω, so this tree will never terminate.