Lecture 6: Programming and Compiling the λ-calculus

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 λ-Calculus Revisited

- Recall our syntax for the λ-calculus, where e are λ-calculus syntactic terms:
  - λ-*terms* are written $\lambda x.e$ or `(lambda (x) e)`, depending on your choice of surface syntax.
  - λ-*applications*, or function calls, are written $(e_1\ e_2)$
  - *Identifiers*, typically written as $x, y, z, \ldots$
- This syntax can be described in Racket code as:

```
;;; 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) #:transparent)
```

- We gave a big-step semantics for the λ-calculus last class called the *call-by-value semantics*:

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

- We also gave a Racket implementation for these semantics. First, we implement our evaluator, which directly corresponds to the big-step semantics above::

```
;;; eval : expr → expr
;;; evaluates a closed expression to a closed expression
(define (eval e)
  (match e
    [(ident x) (error x)]
    [(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))])]))
```

- Then, we implement substitution, which determines our scoping rules:

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

2 More Examples of Running the λ-calculus

- Let's evaluate the λ-term $((\lambda x.x)\ (\lambda y.y))$ by hand and draw its derivation tree.
- First, look at the program and parse it. It is a function application: the function being called is $(\lambda x.x)$, and it is being called with an argument $(\lambda y.y)$. Both of these functions are the identity function. So, we expect this program to evaluate to $(\lambda y.y)$.
- We can test that this is the case in several ways. First, we can test it using Racket, which has lambdas:

```
> (define mystery-fun ((lambda (x) x) (lambda (y) y)))
> (mystery-fun 10)
10
> (mystery-fun "hello")
"hello"
```

- That sure looks like it's behaving like the identity function, so we can be pretty confident that that's what it is here.
- We can also test it using the λ-calculus evaluator:

```
> (eval (app (lam "x" (ident "x")) (lam "y" (ident "y"))))
(lam "y" (ident "y"))
```

- If you want, you can use the provided parser to make this a little easier:

```
> (eval (parse-sexpr '((lambda x x) (lambda y y))))
(lam "y" (ident "y"))
```

- Now we can draw the derivation tree, which describes in detail the process of evaluating this λ-term:

$$\dfrac{\dfrac{}{\lambda x.x \Downarrow \lambda x.x} \qquad \dfrac{}{(\lambda y.y) \Downarrow (\lambda y.y)} \qquad \dfrac{}{x[x \mapsto (\lambda y.y)] \Downarrow (\lambda y.y)}\ (\text{E-}\textsc{Lam})}{((\lambda x.x)\ (\lambda y.y)) \Downarrow (\lambda y.y)}\ \text{E-}\textsc{App}$$

- *Note*: we left the substitution $x[x \mapsto (\lambda y.y)]$ in the above derivation tree. When drawing your derivation trees, you can either leave the substitution like this, or you can simplify it, like in this example tree:

$$\dfrac{\dfrac{}{\lambda x.x \Downarrow \lambda x.x} \qquad \dfrac{}{(\lambda y.y) \Downarrow (\lambda y.y)} \qquad \dfrac{}{(\lambda y.y) \Downarrow (\lambda y.y)}\ (\text{E-}\textsc{Lam})}{((\lambda x.x)\ (\lambda y.y)) \Downarrow (\lambda y.y)}\ \text{E-}\textsc{App}$$

3 Another Example of Running a λ-term

- Let's evaluate another example λ-term and draw its derivation tree: $((\lambda x.x)\ (\lambda y.((\lambda z.z)\ (\lambda w.w))))$
- This one's a bit harder to parse than the first example! let's break it down. The outer-most term is an application. We can label the two parts of that application:

$$(\underbrace{(\lambda x.x)}_{e_1}\ \underbrace{(\lambda y.((\lambda z.z)\ (\lambda w.w)))}_{e_2})$$

- OK, now that we've broken it up, things are a bit easier to see. The term labeled e_1 is the identity function, and we are calling it with the argument labeled e_2. The term e_2 is a lambda-term, so it is a *value*. So, we can conclude that this term runs to $(\lambda y.((\lambda z.z)\ (\lambda w.w)))$
- We can see this using our evaluator:

```
> (eval (parse-sexpr '((lambda x x) (lambda y ((lambda z z) (lambda w w))))))
(lam "y" (app (lam "z" (ident "z")) (lam "w" (ident "w"))))
```

- *Notice*: even though the term e_2 above has a function call inside it, we **do not** evaluate that function call. Why not?
  - The first reason is by definition. Our semantics say that we do not evaluate this term. We said *lambdas are values*. Therefore, we do not run them until they are called. See for yourself why the semantics we gave (both big-step and our evaluator in Racket) do not evaluate this function call.
  - The second reason is by intuition. In most programming languages you've used (Python, C, Java, etc.), the bodies of functions are not evaluated until they are called. This would result in some weird behavior if it were the case! So, we defined our λ-calculus to behave similarly by not evaluating the bodies of functions until they are called.

4 Compilers and Compiler Correctness

- The λ-calculus is a very minimal language, much like assembly language. It consists solely of functions and function calls! You wouldn't want to program in this language exclusively, just like you wouldn't want to program in assembly language.
- We would probably much prefer to program in a higher-level language with more expressive features that is then *compiled to* a low-level representation. You've probably used compiled languages before, like C, that compile to assembly languages like `x86`.
- We call the language that is being compiled the **source language**; in the above example, the host language is C. We call the language being compiled to the **target language**; in the above example, this is assembly language.
- A **compiler** is a semantics-preserving translation from a source language into a target language.
- We can visualize the semantics-preservation requirement on the following *compiler correctness square*:

$$\begin{tikzcd}
S \arrow[r, "\text{Compile}"] \arrow[d, "\text{Run}_S"] & T \arrow[d, "\text{Run}_T"] \\
V_s \arrow[r, "\text{Compile}_V"] & V_t
\end{tikzcd}$$

  This square visually describes the following situation. There are ways to run a source program S to a target value V_t. We can either can either run it under its native semantics Run_s to get an S-value V_s, and then compile the resulting value into a target value. Or, we can first compile the program into a target program T, and then run it under the target semantics Run_T. The **compiler correctness requirement** states that these two ways of running the same program produce the same target value.
- Compilers are the essence of programming languages, and are some of the most important pieces of software we've made. We will spend a lot of time in this course on studying compilers.

5 Compiling to the λ-calculus

- Let's see our first example of a compiler: compiling a tiny language of Booleans into the λ-calculus.
- The syntax of our language of Booleans will be the following:

```
;;; type boolexpr =
;;;   | ebool of bool
;;;   | and of boolexpr × boolexpr
(struct ebool (v) #:transparent)
(struct eand (e1 e2) #:transparent)
```

- We can quickly write an evaluation function for this language:

```
;;; interp-boolexpr : boolexpr → bool
(define (interp-boolexpr e)
  (match e
    [(ebool v) v]
    [(eand e1 e2) (and (interp-boolexpr e1) (interp-boolexpr e2))]
    ))
```

- We will label terms in our λ-calculus as `lexpr`:

```
;;; 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)
```

- Then, our goal is to write a function `compileprog : boolexpr -> lexpr` that compiles programs written in `boolexpr` into programs into equivalent lambda-calculus programs. We will also need a function `compilevalue : bool -> lexpr` that compiles Booleans into lambda-calculus values (i.e., lambda terms).
- Ultimately, we want our compiler to satisfy that, for any boolexpr program `e`, it is the case that `(compilevalue (interp-boolexpr e)) = (eval-lam (compileprog e))`. Note that we will need to be careful about how we define quality here; we will return to this point later.

6 Church Booleans

- The λ-calculus clearly does not have Booleans! It only has functions and function calls. So, we will need to choose a way to represent the Boolean values `true` and `false` in the λ-calculus.
- We only have 1 kind of value in the λ-calculus: functions. So, we need to choose particular functions to represent `true` and `false`. Let's pick two, somewhat arbitrarily:

$$\texttt{true} = \lambda a.\lambda b.a$$
$$\texttt{false} = \lambda a.\lambda b.b$$

- This choice of encoding of Boolean values is named after Alonzo Church, the inventor of the λ-calculus. They are called the **Church-Booleans**. [1]
- These functions seem mysterious. Let's look at their behavior *as lambda terms*. The `true` function *ignores its second argument*, and the `false` function *ignores its first argument*.
- Let's assume these are our encodings of `true` and `false`. How can we compile logical operations like `and`, `or`, etc.? It's quite tricky! Our target language (the λ-calculus) does not have any logical primitives like if-then-else, etc.
- Let's solve this by finding a λ-term that *behaves like if-then-else*. Here is a candidate:

$$\texttt{ite} = (\lambda g.(\lambda t.(\lambda e.((g\ t)\ e))))$$

- Wow, what does this `ite` program do? Let's break it down slowly (read this carefully). First, it takes 3 arguments, g, t, and e. The first argument g is the guard; it is a Church Boolean. If g is equal to the Church-Boolean `true`, then we should return t (which can be thought of as the then-branch); otherwise, if g is equal to the Church-Boolean `false`, then the else-branch e should be returned.
- How does `ite` do this? It's surprisingly simple. Recall the Church Booleans are themselves *functions*: they take 2 arguments. `true` forgets its first argument, and `false` forgets its second argument. Look all the way inside the `ite` lambda to the inner-most body. In this body we see the function application $((g\ t)\ e)$. What does this do? If g is equal to `true`, it ignores the second argument e and returns t; if it is equal to `false`, it ignores t and returns e; this is what we expect if-then-else to do.
- Try evaluating the following λ-term! What does it evaluate to?

$$(((\texttt{ite}\ \texttt{true})\ \texttt{false})\ \texttt{true})$$

- This `ite` function lets us define other logical operations like `and`. Concretely, `a and b` is equivalent to `if a then b else false`, which we can write as follows:

$$\texttt{and} = (\lambda g.(\lambda t.((g\ t)\ \texttt{false})))$$

- *Pause*: We could have defined the Church Booleans in the other way: we could have defined `true` to forget its first argument and `false` to forget its second argument. How would this change our implementation of `ite`?

[1] Fun fact: Alonzo Church was Alan Turing's PhD. advisor. Together, they invented the two most widely-used foundations for computing: the Turing machine and the λ-calculus.

7 Compiling to Church Booleans

- We are ready to make our first compiler that translates Boolean expressions into λ-calculus terms!
- The code is surprisingly simple:

```
(define lamfalse (lam "a" (lam "b" (ident "b"))))
(define lamtrue (lam "a" (lam "b" (ident "a"))))

;;; church : boolexpr → lexpr
(define (church e)
  (match e
    [(ebool #t) lamtrue]
    [(ebool #f) lamfalse]
    [(eand e1 e2)
     (let ([and-term (lam "a" (lam "b" (app (app (ident "a") (ident "b")) lamfalse
    ))))])
       (app (app and-term (church e1)) (church e2)))]))
```

- We can test that this program satisfies the compiler correctness criteria as follows:

```
(check-equal? (eval-lam (church (eand (ebool #t) (ebool #f)))) lamfalse)
(check-equal? (eval-lam (church (eand (ebool #t) (ebool #t)))) lamtrue)
(check-equal? (eval-lam (church (eand (eand (ebool #t) (ebool #t)) (ebool #t))))
   lamtrue)
(check-equal? (eval-lam (church (eand (eand (ebool #t) (ebool #f)) (ebool #t))))
   lamfse)
```

- Take some time on your own to look at the results of Church-encoding some small functions.