Lecture 9: Let-rec and Recursion

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 Some Common Inference Rules and Derivation Tree Mistakes

- Recall the big-step semantics for the call-by-value λ-calculus, extended with numbers (denoted syntactically by n):

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

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

- **Mistake #1: Scope** Be careful when performing substitution. The following substitution is *wrong*:

$$(\lambda x.x)[x \mapsto 10] \stackrel{?}{=} \lambda x.10$$

  How can we tell? There's several ways:

  - First, this violates our lexical scoping rules. Identifiers always refer to their nearest binding.
  - Second, this doesn't match our specification of the substitution function. We can see this in several ways. One way is to go look at the definition we gave in Section 5 of Lecture 5 of the course notes. Another way is to load up the λ-calculus interpreter into DrRacket and try calling `subst`:

```
> (subst (lam "x" (ident "x")) "x" (lam "y" (ident "y")))
(lam "x" (ident "x"))
```

- **Mistake #2: Simplifying under lambdas** This was a very common mistake on the homework, and several people asked about it on Piazza. The following is not true:

$$(\lambda x.((\lambda x.x)\ 10)) \Downarrow^{?} \lambda x.10$$

  Why is this the case? We discussed this in some detail in Section 3 of Lecture 6, I encourage you to go there and see the reasoning.

- Helpful hints:

  - Remember that the notation $e \Downarrow v$ corresponds to calling the λ-calculus interpreter `(eval e)`, and $e_1[x \mapsto e_2]$ corresponds to calling the substitution function `(subst e1 x e2)` that we provided you in class. If you are ever wondering what these two do, go run the code.
  - Derivation trees describe what your interpreter does. If your interpreter doesn't do it, then your derivation tree shouldn't do it.

2 Recursion and Scope

- In class we've seen plenty of examples of recursive functions: for instance, all of your interpreters have been recursive functions.[1]
- In Racket we can easily implement recursive functions. For example here is the factorial function:

```
(define (fact n)
  (if (equal? n 0)
      1
      (* n (fact (- n 1)))))
```

- The reason this definition works is that the name of the function (`fact`) is in scope in the body of the function's definition. This lets the function call itself.
- Our goal is to be able to define recursive functions like `fact` in the λ-calculus. However, the λ-calculus doesn't give us a way to define names like Racket, so this seems super tricky!
- Continuing in Racket, we might start like this attempt to write a factorial function without using the `define` keyword:

```
(define almost-factorial (lambda (n)
  (if (equal? n 0)
      1
      (* n (self ( - n 1))))))
```

- Clearly this is not a valid Racket program, and indeed it will raise an error if you try to call it using the Racket REPL: it will say that this magical "self" term is not defined.

[1] This progression presenting the Y-combinator was heavily inspired by this blogpost `https://mvanier.livejournal.com/2897.html` which was heavily inspired by Eli Barzilay.

3 Unfolding Recursion

- What should we put in this `almost-factorial` function for `self`, if we can't put a self-referencing call to `almost-factorial` there?
- There's a simple approach that obviously won't work in general, but seems like it might help us: we can define *another copy of almost-factorial*, and call that one:

```
(define factorial (lambda (n)
  (if (equal? n 0)
      1
      (* n (self ( - n 1)))))))

(define almost-factorial (lambda (n)
  (if (equal? n 0)
      1
      (* n (factorial ( - n 1))))))
```

- Of course, this doesn't really solve our problem: now we're left with another function called `factorial`, that again has to refer to itself! But, seem to have made a bit of progress: at least the `almost-factorial` function is now well-defined.
- In general, any time I have a recursive function, I can split it up into (1) a non-recursive function that does one step of the recursion, and (2) a recursive function that does the rest of the work. This process is called **a one step unfolding of recursion**; in the above example, we generate a 1-step unfolding of `almost-factorial`.
- Here is the algorithm for unfolding a recursive function `f`, which is quite simple:
  - Generate a new function definition `f-fresh` by copy-and-pasting the definition of `f`.
  - Find all calls to `self` in `f` and replace them with calls to `f-fresh`.
- If you know that a recursive function has a particular maximum number of recursive calls, then you can unroll it to that number of recursive calls and it will work!
- Now you can imagine that, *if we can do unlimited unfoldings*, we would eventually be able to define an arbitrarily recursive function. This idea of unlimited unfoldings will be critical.

4 Letrec

- Racket has the following syntax for `let` that lets us define a recursive factorial function:

```
> (letrec [(fact (lambda (n) (if (equal? n 0)
                                  1
                                  (* n (fact (- n 1))))))]
    (fact 4))
24
```

- This `letrec` construct behaves very differently from our usual version of `let`! First, let's look at its scoping rules. We see that the identifier `fact` is in-scope in its binding (i.e., the lambda we are defining can refer to `fact`)! This is very different from `let`: the following program will fail with an unbound identifier error:

```
> (let [(fact (lambda (n) (if (equal? n 0)
       1
       (* n (fact (- n 1))))))]
    (fact 4))
```

- As usual in this class, let's try to implement a mini-language that has support for `letrec` to understand how it is implemented. Our goal is for it to behave like Racket's `letrec`.
- As usual, we want to implement `letrec` with substitution. But *what do we substitute?* We need to make sure we don't end up with an unbound identifier error after substituting.

5 Unfolding letrec

- Let's try to do one step of unfolding `letrec`
- Here is our strategy:
  - Replace `letrec` with `let`, because we know how `let` is supposed to work using substitution.
  - Anywhere we have a recursive call inside the assignment of the `letrec`, replace that with a `letrec`.
- For example, we can unfold the factorial `letrec` as:

```
(letrec [(fact (lambda (n) (if (equal? n 0)
                                1
                                (* n (fact (- n 1))))))]
    (fact 4))

-- unfolding 1 step -->

(let [(fact (lambda (n) (if (equal? n 0)
                              1
                              (* n ((letrec [(fact (lambda (n)
                                                      (if (equal? n 0)
                                                          1
                                                          (* n (fact (- n 1)))))
    )]
                                        fact)
                                      (- n 1))))))]
    (fact 4))
```

- (matching up parenthesis on the above example is pretty tricky! It might help to copy/paste it into Racket to see)
- Notice: we have a few copy/pasted copies of the body of the factorial function. *But*, we got rid of one level of `letrec`!
- Notice: when unfolding, we replaced the self-call to `fact` with another instance of `letrec`:

```
(letrec [(fact (lambda (n)
                  (if (equal? n 0)
                      1
                      (* n (fact (- n 1))))))]
  fact)
```

- This `letrec` simply defines the `fact` function and then returns it.

6 Implementing letrec

- Now hopefully you can see a way to implement `letrec`. The idea is to *unfold one step* whenever you encounter a let-rec during execution. The key is to unfold *only when needed*: this keeps us from infinitely unfolding forever.

- We can define the AST for our tiny `letrec` language:

```
(struct eident (s) #:transparent)
(struct eletrec (f fun body) #:transparent)
(struct elet (id assgn body) #:transparent)
(struct elam (id body) #:transparent)
(struct eapp (e1 e2) #:transparent)
```

- In the above syntax, we require that the binding `fun` of a `letrec` is a `lambda`

- Then, we can define an evaluator. The evaluator is the same as all the other interpreters except for the rule for `letrec`:

```
(define (interp e)
  (match e
    [(eident x) (error x)]
    [(elam id x) (elam id x)]
    ...
    [(eletrec f fun body)
    ; run (elet f fun[f |→ (eletrec f lamarg lambody f)] body)
    (interp (elet f (subst fun f (eletrec f fun (eident f))) body))]
    ...))
```

- As always, we can also describe this using inference rules as well:

$$\frac{\texttt{(elet f fun[x} \mapsto \texttt{(eletrec f fun (eident f))] body)} \Downarrow v}{\texttt{(eletrec f fun body)} \Downarrow v} \text{(E-LETREC)}$$

- We've provided an implementation for this `letrec` language with everything you need to implement the factorial function on the course webpage. See for yourself that it works!

7 Recursion in the λ-calculus

- Note: This is a pretty advanced section. It won't be required for or built on further in the course, and we may not have time to get to it in lecture.
- We've discussed in class a few times about how the λ-calculus is a Turing-complete language and can represent all Racket programs.
- So, how is it possible to represent programs involving `letrec` in the λ-calculus?
- Put differently: how can we write the factorial function in Racket without needing to use `define` or `letrec` (i.e., only using `lambda`, `if`, and numeric operations)?
- We saw that the key insight to implementing `letrec` is to unfold the recursive function one level whenever it is called. So, we need to come up with a special λ-term that *unfolds one level of recursion whenever it is called*.
- First-things-first: we need to make this magical λ-term available to our factorial function. The only way to do that is to make the function take in an extra argument called `self`:

```
> (define almost-factorial (lambda (self)
                             (lambda (n)
                              (if (equal? n 0)
                                  1
                                  (* n (self (- n 1)))))))
```

- Now, where does this magical `self` parameter come from? It should be a function that, when called, generates a once-unfolded version of `almost-factorial`.
- Coming up with this special function took a long time; it was discovered by Haskell Curry. It is called the **Y-Combinator**, and it looks like this:

```
(define Y
  (lambda (f)
    ((lambda (x) (f (lambda (y) ((x x) y))))
     (lambda (x) (f (lambda (y) ((x x) y)))))))
```

- It looks super weird. Before we unpack it, let's see how it's used:

```
> (define factorial (Y almost-factorial))
(factorial 4)
24
```

- Wow! Let's take stock of what's happening: we've implemented the recursive factorial function *without using letrec or explicit recursion!* But how?

8 Unfolding Y

- First, let's see what happens when we call `Y` with the argument `almost-factorial`.

```
(Y almost-factorial)
=  ((lambda (x) (f (lambda (y) ((x x) y))))
    (lambda (x) (f (lambda (y) ((x x) y)))))[f |→ almost-factorial]
=  ((lambda (x) (almost-factorial (lambda (y) ((x x) y))))
    (lambda (x) (almost-factorial (lambda (y) ((x x) y)))))
```

  Notice, this is a function call, so we keep evaluating. To keep our

```
= (almost-factorial (lambda (y) ((x x) y)))
      [x |→ (lambda (x) (almost-factorial (lambda (y) ((x x) y))))]
= (almost-factorial (lambda (y)
      (((lambda (x) (almost-factorial (lambda (y) ((x x) y))))
        (lambda (x) (almost-factorial (lambda (y) ((x x) y))))) y)))
= (almost-factorial (lambda (y) (Y almost-factorial) y))
```

- Bingo! Remember, the first argument to `almost-factorial` is `self`. So, the function call `(Y almost-factorial)` evaluates to `almost-factorial` with `self` replaced by `(lambda (y) (Y almost-factorial) y)`; this is exactly 1-level deep of recursive unfolding.