Fixed Points

CIS352 — Fall 2022

Kris Micinski

Last lecture: encoding Scheme in the lambda calculus

```
e ::= (letrec ([f (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

Last lecture: encoding Scheme in the lambda calculus

But didn't do letrec

```
e ::= (letrec ([x (lambda (x …) e)]))
    | (let ([x e] …) e)
    | (lambda (x …) e)
    | (e e …)
    | x
    | (if e e e)
    | (prim e e) | (prim e)
    | d
d ::= ℕ | #t | #f | ‘()
x ::= <vars>
prim ::= + | - | * | not | cons | …
```

`letrec` lets us define recursive loops

```
(letrec ([f (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (sub1 x)))))])
  (f 20))
```

`letrec` lets us define recursive loops

```
(letrec ([f (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (sub1 x)))))])
  (f 20))
```

Unlike **let**, letrec allows referring to f **within** its definition

Unlike **let**, letrec allows referring to f **within** its definition

```
(define (fib-using-letrec x)
  (letrec ([fib (lambda (x)
                  ;; Your answer:
                  'todo)])
    (fib x)))
```

Today, we will discuss a magic term, **Y**, that allows us to write…

```
(letrec ([f (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (sub1 x)))))])
  (f 20))
```

```
(let ([f
       (Y (lambda (f)
            (lambda (x)
              (if (= x 0)
                  1
                  (* x (f (- x 1))))))))])
  (f 20))
```

This magic term, named Y, allows us to construct recursive functions.

```
(define Y (λ (g) ((λ (f) (g (λ (x) ((f f) x))))
                  (λ (f) (g (λ (x) ((f f) x)))))))
```

First, the U combinator

```
(define U (lambda (x) (x x)))
```

The U combinator lets us do something very crucial: pass a copy of a function to itself.

Let's say I didn't have letrec, what could I do…?

First observation: pass f to **itself**

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) x)))))])
  ((f f) 20))
```

`mk-f` is pronounced "make f"

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let’s see why this works!

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let’s see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

This initial call “makes the next copy”

```
(let ([f (lambda (mk-f)
           (lambda (x) ;; x = 20
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x)))))))])
  ((f f) 20))
```

Let’s see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

2: Second, apply that (lambda (x) …) to 20, take false branch

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let's see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

2: Next, apply that (lambda (x) …) to 20, take false branch

3: Next, compute (mk-f mk-f), which **gives us another copy of (lambda (x) …)**

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((f f) 20))
```

Let's see why this works!

1: First, apply f to itself. First lambda goes away, returns (lambda (x) …) with mk-f bound to mk-f

2: Next, apply that (lambda (x) …) to 20, take false branch

3: Next, compute (mk-f mk-f), which **gives us another copy of (lambda (x) …)**

4: Apply that same function again (until base case)!

The U combinator recipe for recursion…

```
(letrec ([f (lambda (x) e-body)])
  letrec-body)
```

Systematically translate any letrec by:

- Wrapping `(lambda (x) e-body)` in `(lambda (f) …)`
- Changing occurrences of `f` (in e-body) to `(f f)`
- Apply U combinator / apply function to itself
- Changing `letrec` to `let`

Think carefully why this works..!

The U combinator recipe for recursion…

```
(letrec ([f (lambda (x) e-body)])
  letrec-body)
```

Systematically translate any letrec by:

- Wrapping `(lambda (x) e-body)` in `(lambda (f) …)`
- Changing occurrences of `f` (in e-body) to `(f f)`
- Apply U combinator / apply function to itself
- Changing `letrec` to `let`

```
(let ([f (U (lambda (f)
              ;; replace f w/ (f f)
              (lambda (x) e-body))])
  letrec-body)
```

Let's do an example…

```
(define (length-using-letrec lst)
  (letrec ([len (lambda (x)
                  (if (null? x)
                      0
                      (add1 (len (rest x)))))])
    (len lst)))
```

Your job…

```
(define (length-using-u lst)
  (let ([len (U (lambda (f)
                  (lambda (x)
                    'todo)))])
    (len lst)))
```

Now another example…

```
(define (fib-using-letrec n)
  (letrec ([fib
            (lambda (x)
              (cond [(= x 0) 1]
                    [(= x 1) 1]
                    [else (+ (fib (- x 1))
                             (fib (- x 2)))]))])
    (fib n)))
```

Translate **this** one to use U

```
(define (fib-using-U n)
  (letrec ([fib (U 'todo)])
    (fib n)))
```

```
(let ([f (lambda (mk-f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x ((mk-f mk-f) (sub1 x))))))])
  ((U f) 20))
```

One pesky thing: need to rewrite function so that calls to mk-f need to first "get another copy" by doing (mk-f mk-f)

By contrast, the **Y** combinator will allow us to write **this**

```
(let ([f (lambda (f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x (f (sub1 x))))))])
  ((Y f) 20))
```

```
(let ([f (Y (lambda (f)
              ;; no change to e-body
              (lambda (x) e-body))])
  letrec-body)
```

Let’s ask ourselves: what does f need to **be** when Y plugs it in…?

$$(Y\, f) = f\ (Y\, f)$$

Deriving Y

```
(Y f) = (f (Y f))
```

```
Y = (λ (f) (f (Y f)))
```

1. Treat as definition

```
mY = (λ (mY)
       (λ (f)
         (f ((mY mY) f))))
```

2. Lift to mY,
use self-application

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

3. Eta-expand

U-combinator: `(U U)` **is Omega**

```
Y  =  (U (λ (y) (λ (f)
          (f (λ (x) (((y y) f) x))))))
```

```
mY = (λ (mY)
       (λ (f)
         (f (λ (x) (((mY mY) f) x)))))
```

$$(Y\,f) = f\,(Y\,f)$$

By contrast, the **Y** combinator will allow us to write **this**

```
(let ([f (lambda (f)
           (lambda (x)
             (if (= x 0)
                 1
                 (* x (f (sub1 x))))))])
  ((Y f) 20))
```

Closing words of advice:

- Understand how to write recursive functions w/ U / Y
- Do not need to remember precisely why Y works
  - But do need to remember how to use it!
- If you want to understand: just think carefully about what U / Y are doing (with examples)

Continuations

CIS352 — Fall 2022

Kris Micinski

Often speak of evaluating programs in a sequence of steps:

```
(+ (* 2 1) 3) -> (+ 2 3) -> 5
```

E.g., textual reduction. We defined textual reduction for IfArith and for lambda calculus (beta, …)

Textual Reduction Review

Key idea: at each step, we just decided which expression to reduce (using reduction strategy)

```
((lambda (x) ((lambda (y) x) z))
 (lambda (z) (lambda (…) …))
```

In a real implementation, this would be slow (would have to traverse term at each step)

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
(+ (* 2 1) 3)    stack = □ (empty stack)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we "finish" the current expression, we "fill in" the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
-> 3                stack = (+ 2 □)
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
-> 3                stack = (+ 2 □)
-> (+ 2 3)          stack = □
```

Another way to conceptualize this would be to think of an **explicit stack**

The rule here is: once we “finish” the current expression, we “fill in” the stack

```
   (+ (* 2 1) 3)    stack = □ (empty stack)
-> (* 2 1)          stack = (+ □ 3)
-> 2                stack = (+ □ 3)
-> 3                stack = (+ 2 □)
-> (+ 2 3)          stack = □
-> 5                stack = □ (done!)
```

These stacks have another appeal: the fact that they make only local changes makes them fast (compared to identifying redex each time).

Instead, we will observe that this style offers an additional flexibility: we can always conceptualize the return point as a function!

We call this function the "continuation," since it lets us "continue" the computation.

```
   (+ (* 2 1) 3) ;; (lambda (rtn) rtn)
-> (* 2 1)       ;; (lambda (x) (+ x 3))
-> 2             ;; (lambda (x) (+ x 3))
-> 3             ;; (lambda (x) (+ 2 x))
-> (+ 2 3)       ;; (lambda (x) x)
-> 5             ;; (lambda (x) x)
```

If you're used to programming in Java/C++, you can think of a continuation as a "callback we invoke to return from a function."

```
   (+ (* 2 1) 3) ;; (lambda (x) x)
-> (* 2 1)       ;; (lambda (x) (+ x 3))
-> 2             ;; (lambda (x) (+ x 3))
-> 3             ;; (lambda (x) (+ 2 x))
-> (+ 2 3)       ;; (lambda (x) x)
-> 5             ;; (lambda (x) x)
```

The call/cc form allows us to **bind** this continuation to a **function**

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

When control reaches call/cc, the program binds the **current continuation** to k

In this case, the current continuation is…

```
(+ 4 (call/cc (lambda (k) (k 3))))
;; (lambda (x) (+ 4 x))
```

How could we write the continuation at the **underlined point**?

```
(let* ([x (+ (* 2 3) 4)]
       [y (add1 x)])
  y)

(lambda (z)
 (let* ([x (+ z 4)] [y (add1 x])) y))
```

How could we write the continuation at the **underlined point**?

```
(let* ([x (+ (* 2 3) 4)]
       [y (add1 x)])
  y)
```

```
(lambda (result)
  (let* ([x (+ result 4)]
         [y (add1 x)])
  y)
```

DANGER

Continuations are normal functions in most ways. One crucial difference: when you invoke a continuation, it **abandons** the current stack and **reinstates** the continuation!

Again: invoking a continuation is **different** than invoking a *normal* (non-continuation) function.

Students **frequently** find this confusing!

When execution reaches **this point**, k is bound as the continuation

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

Then, when we **invoke** the continuation, we **abandon** the *current* continuation and **reinstate** the *saved* continuation

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

Then, when we **invoke** the continuation, we **abandon** the *current* continuation and **reinstate** the *saved* continuation

```
(+ 4 (call/cc (lambda (k) (k 3))))
```

But in this example, the saved continuation is equivalent to the current continuation, so we observe no difference!

The program never returns from call `(k 3)` because ***undelimited continuations*** run until the program exits.

`call/cc` gives us undelimited (a.k.a. full) continuations.

```
(+ 1 (call/cc (lambda (k) (k 3) (print 0))))
;; =>  4      (print 0) is never reached
```

The program never returns from call `(k 3)` because ***undelimited continuations*** run until the program exits.

`call/cc` gives us undelimited (a.k.a. full) continuations.

```
(+ 1 (call/cc (lambda (k) (k 3) (print 0))))
;; =>  4      (print 0) is never reached
```

Pause the video and type this one into Dr. Racket!

Do you understand why `(print 0)` is never reached?

```scheme
(+ 1 (call/cc (lambda (k) (k 2))))
;; => 3
```

This `call/cc`'s behavior is *roughly* the same as the application:

```scheme
((lambda (k) (k 2))
 (lambda (n) (exit (print (+ 1 n)))))
;; => 3
```

Where the high-lit continuation `(lambda (n) …)` takes a return value for the `(call/cc …)` expression and finishes the program.

When execution reaches **this point**, k is bound as the continuation

```
(+ 4 (call/cc (lambda (k) (+ 5 (k 3)))))

  k = <continuation> (lambda (x) (+ 4 x))
```

When control **reaches** this point, the current continuation is…

```
(lambda (x) (+ 4 (+ 5 x)))
```

```
(+ 4 (call/cc (lambda (k) (+ 5 (k 3)))))
```

```
(+ 4 (call/cc (lambda (k) (+ 5 (k 3)))))
```

And, **by invoking k**, then we abandon it to *reinstate* k

```
(lambda (x) (+ 4 x))
```

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

Try an example. What do each of these 3 examples return?
(Hint: Racket evaluates argument expressions left to right.)

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 3)))))))
```

3

```
(call/cc (lambda (k0)
           (+ 1 (call/cc (lambda (k1)
                           (+ 1 (k0 (k1 3))))))))
```

4

```
(call/cc (lambda (k0)
           (+ 1
              (call/cc (lambda (k1)
                         (+ 1 (k1 3))))
              (k0 1))))
```

1

Lecture Summary

- Continuations allow us to capture the stack in a first-class way
- call/cc (call-with-current-continuation)
  - Let's us bind special **continuation** functions
- When invoked, continuations **reset the stack**
- As we will soon see, this enables building non-local control constructs (loops, exceptions, etc…)