Y-Combinator

Avi Hayoun, Ben Eyal and Lior Zur-Lotan

December 8, 2016

Contents

1 Introduction

1.1 Terms recap

- Higher-Order function - a function that either takes or returns a function or both.
- Combinator - A High-Order function that only uses function application and other combinators (no need for a global scope).
- Fixed-Point - an input x to a function f such that $f(x) = x$. Some functions have a single fixed-point, some functions have more than one and some have non at all.

Remark. *Note that fixed points are not just primitive values like 5 or 'c', they can be anything. Numbers, strings, sets, classes, and even functions can be the fixed point of some function*

1.2 Y-Combinator notes

- Y-Combinator is a Fixed-Point Combinator - a combinator y for which $f(y(f)) = y(f)$.
- Y-Combinator was introduced in Lambda Calculus. It helps solves (among other things) the lack of global definitions in Lambda Calculus in certain situations.
- The Y-Combinator can be defined like so:

  ```
  (define Y-comb (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))
  ```

  Where f is the function being fixed. This definition only works when using lazy evaluation, though (we'll see another one later).
- From a programming point of view applying the Y-Combinator to a function can be seen as mean to implement recursion (under some conditions it may be the best way).

2 Deriving the Y-Combinator

In a way, a recursive function is strange. Every "regular" (i.e. non-recursive) function can be viewed as a macro. When it's invoked it'll be expanded instead of applied. Attempting to do so with a recursive function will result in an infinite loop (you'll never stop expanding). To write a recursive functions, we'll store a reference to it in the environment (i.e. we'll name it). We'll derive the Y-Combinator by trying to implement a recursive function without using an environment.
Think of the following code:

```
(define fact
  (lambda (num)
    (if (zero? num)
        1
        (* num (fact (- num 1))))))
```

2.1 Almost factorial

This implementation of factorial requires us to create some environment in which the symbol **fact** is bound to our lambda. Lets try and remove our dependency on the environment. Lets take the recursive call as a parameter:

```
(define almost-fact
  (lambda (f)
    (lambda (num)
      (if (<= num 0)
          1
          (* num (f (- num 1)))))))
```

If we somehow could find a way to send the right function to **almost-fact** we'd be done.
Lets say we have a partial function $\mathbf{fact_i}$ that computes factorial up to i:

- (fact$_3$ 0) = 1
- (fact$_3$ 1) = 1
- (fact$_3$ 2) = 2
- (fact$_3$ 3) = 6
- (fact$_3$ 4) is undefined

What's interesting about $\mathbf{fact_3}$ is that if we send it as the parameter to **almost-fact** we will get $\mathbf{fact_4}$. Formally, **(almost-fact** $\mathbf{fact_i}$**)** $=$ $\mathbf{fact_{i+1}}$:

- (define fact$_1$ (almost-fact fact$_0$))
- (define fact$_2$ (almost-fact fact$_1$))

Or, better yet:

- (define fact$_1$ (almost-fact fact$_0$))
- (define fact$_2$ (almost-fact (almost-fact fact$_0$)))
- (define fact$_3$ (almost-fact (almost-fact (almost-fact fact$_0$))))
- ...
- (define fact (almost-fact (almost-fact (almost-fact ...))))

But how do we get fact_0? Well, the same way we created fact_1, except we don't care what we give **almost-fact**. Remember, fact_0 computes factorial up to 0. For any other value it's undefined. If we look at the body of **almost-fact**, we see that when given 0 as the value for n, it never applies f. 0 Is the termination condition of our recursion. As far as we care we can define fact_0 as (almost-fact 'bob). That 'bob is never going to be used anyways.

Question: If we would use this definition of fact, what would $(fact\ 5)$ be expanded to?
Incorrect Answer:

```
(almost-fact (almost-fact ... (almost-fact 5)...))
```

This is incorrect for two reasons. First, there is never an end to the tail of **almost-fact** calls so there can't be an argument at the (non-exiting) end. Second, **fact** is the result of the repeated application of **almost-fact**, the argument 5 is sent to the **result** (i.e. value) of that infinite application, like so:
Correct Answer:

```
((almost-fact (almost-fact ...)) 5)
```

2.2 Better almost factorial

You might've noticed that

```
(almost-fact fact) = fact
```

So **fact** is a fixed-point of **almost-fact**.
You might expect that running **(almost-fact almost-fact)** would produce **fact** (since **almost-fact** has the same termination condition and loop structure), but notice the type violations. **almost-fact** expects a function that itself expects a number, not another function. We could, however change **almost-fact** a bit to make it accept a function that expects a function:

```
(define better-almost-fact
  (lambda (f)
    (lambda (num)
      (if (<= num 0)
          1
          (* num ((f f) (- num 1)))))))
```

And now we can call **((better-almost-fact better-almost-fact) 5)** and get 5! (120). Note however, that **fact** isn't the fixed point of **better-almost-fact**. The idea is that when we apply **better-almost-fact** on itself it applies the core of **almost-fact** on itself. Just like we did with fact_0, fact_1, fact_2, ... all the way up to fact (infinity).

Now lets write fact without using the environment. We claim that **(better-almost-fact better-almost-fact)** = **fact**, so:

```
(define envless-fact
  ;; Applying better-almost-fact on better-almost-fact
  ((lambda (f)
     (lambda (num)
       (if (<= num 0)
           1
           (* num ((f f) (- num 1))))))
   (lambda (f)
     (lambda (num)
       (if (<= num 0)
           1
           (* num ((f f) (- num 1))))))))
```

This is a definition of factorial without the need for an environment.
Now, lets try and generalize this infinite self application (we might want to create a function other than fact someday). First thing we need to do is gather up all the parts that are specific to factorial, so we can encapsulate them (and replace them with whatever function we want to fix-point).
The original code for almost-fact was:

```
(define almost-fact
  (lambda (f)
    (lambda (num)
      (if (<= num 0)
          1
          (* num (f (- num 1)))))))
```

Notice that there's only a single **f** in line 6. The conversion to **(f f)** was not unique to factorial, so it should be extracted. We need to get that **(f f)** out of there. Since we know that **((lambda (x) (func x)) arg)** is the same as **(func arg)** we can write:

```
1 (define better-almost-fact
2   (lambda (f)
3     (lambda (num)
4       (if (<= num 0)
5           1
6           (* num ((lambda (x) ((f f) x)) (- num 1)))))))
```

(- num 1) at line 6 stands for **arg** from the relation above.

What's so great about this? Look:

```
1 (define better-almost-fact
2   (lambda (f)
3     ((lambda (arg-func)
4        (lambda (num)
5          (if (<= num 0)
6              1
7              (* num (arg-func (- num 1))))))
8      (lambda (arg) ((f f) arg)))))
```

Note that line 3 is an application. This encapsulation should affect what **better-almost-fact** looks like from the "outside".
The body of the inner lambda is now the same as the body for **almost-fact**, which is purely **fact**-specific (also, the lambda wrapper allows this to work under eager computation, instead of lazy).
Now lets update **envless-fact**:

```
1 (define envless-fact
2   ((lambda (f)
3      ((lambda (arg-func)
4         (lambda (num)
5           (if (<= num 0)
6               1
7               (* num (arg-func (- num 1))))))
```

```
      (lambda (x) ((f f) x))))
   (lambda (f)
     ((lambda (arg-func)
        (lambda (num)
          (if (<= num 0)
              1
              (* num (arg-func (- num 1))))))
      (lambda (x) ((f f) x))))))
```

Now the body of the **(lambda (f) . . .)** (both of them) is purely factorial-specific. Lets call that part **fact-loop** and update the **envless-fact**:

```
(define fact-loop
  (lambda (arg-func)
    (lambda (num)
      (if (<= num 0)
          1
          (* num (arg-func (- num 1)))))))

(define envless-fact
  ((lambda (f) (fact-loop (lambda (x) ((f f) x))))
   (lambda (f) (fact-loop (lambda (x) ((f f) x))))))
```

We're just one step from the Y-Combinator. We completely move the core of factorial out, but we just need to abstract **fact-loop**:

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

3 Using the Y-Combinator

If we allow ourselves to define **Y** in the environment we can define **fact** (still envless) like so:

```
(define fact
  (Y
  ((lambda (arg-func)
     (lambda (num)
       (if (<= num 0)
           1
           (* num (arg-func (- num 1)))))))))
```

All we did was send the loop definitions to **Y** and it gave us back the fixed point (which is **fact**). If we don't allow ourselves to define even **Y** in the environment, we just write down it's expansion:

```
(define fact-final
  ((lambda (X)
     ((lambda (f) (X (lambda (x) ((f f) x))))
      (lambda (f) (X (lambda (x) ((f f) x))))))
   (lambda (arg-func)
     (lambda (num)
       (if (<= num 0)
           1
           (* num (arg-func (- num 1))))))))
```

We can use **Y** to define many other recursive functions. For instance we can create a Fibonacci function:

```
(define fib
  (lambda (n)
    (cond
     ((zero? n) 1)
     ((= 1 n) 2)
     (else (+ (fib (- n 1)) (fib (- n 2)))))))
```

To use the Y-Combinator we need to extract the core of the loop:

```
(define fib-Y-loop
  (lambda (func)
    (lambda (n)
      (cond
       ((zero? n) 1)
       ((= 1 n) 2)
       (else (+ (func (- n 1)) (func (- n 2))))))))
```

Now we can define **envless-fib**:

```
(define envless-fib (Y fib-Y-loop))
```