Y-Combinator

Avi Hayoun, Shahaf Shperberg and Lior Zur-Lotan

November 27, 2017

Contents

1 Introduction ........................................... 1
   1.1 Terms recap ........................................ 1
   1.2 Y-Combinator notes ............................... 1

2 Deriving the Y-Combinator ............................ 2
   2.1 Almost factorial ................................. 2
   2.2 Better almost factorial ......................... 3

3 Fixed Points and Y .................................... 5

4 Using a Y-Combinator .................................. 5

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 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. In this case, we will discuss functions that are 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.

2 Deriving the Y-Combinator

A Y-Combinator allows us to implement recursive functions without using an environment.

Consider the factorial function:

```scheme
(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. Let's try and remove our dependency on the environment. Let's take the recursive call as a parameter:

```scheme
(lambda (f)
 (lambda (num)
   (if (<= num 0)
       1
       (* num (f (- num 1))))))
```
Denote this function as \textbf{almost-fact}. If we somehow could find a way to send the right function to \textbf{almost-fact} we’d be done.

Lets say we have a partial function \textbf{fact}_i that computes factorial up to \textit{i}:

\begin{itemize}
  \item \textbf{fact}_3(0) = 1
  \item \textbf{fact}_3(1) = 1
  \item \textbf{fact}_3(2) = 2
  \item \textbf{fact}_3(3) = 6
  \item \textbf{fact}_3(4) is undefined
\end{itemize}

What’s interesting about \textbf{fact}_3 is that if we send it as the parameter to \textbf{almost-fact} we will get \textbf{fact}_4. Formally, \((\text{almost-fact } \textbf{fact}_i) = \textbf{fact}_{i+1}:

\begin{itemize}
  \item (define \textbf{fact}_1 (almost-fact \textbf{fact}_0))
  \item (define \textbf{fact}_2 (almost-fact \textbf{fact}_1))
\end{itemize}

Or, better yet:

\begin{itemize}
  \item (define \textbf{fact}_1 (almost-fact \textbf{fact}_0))
  \item (define \textbf{fact}_2 (almost-fact (almost-fact \textbf{fact}_0)))
  \item (define \textbf{fact}_3 (almost-fact (almost-fact (almost-fact \textbf{fact}_0))))
  \item \ldots
  \item (define \textbf{fact} (almost-fact (almost-fact (almost-fact \ldots))))
\end{itemize}

But how do we get \textbf{fact}_0? Well, the same way we created \textbf{fact}_1, except we don’t care what we give \textbf{almost-fact}. Remember, \textbf{fact}_0 computes factorial up to 0. For any other value it’s undefined. If we look at the body of \textbf{almost-fact}, we see that when given 0 as the value for \textit{n}, it never applies \textit{f}. 0 is the termination condition of our recursion. As far as we care we can define \textbf{fact}_0 as \((\text{almost-fact 'bob})\). That ‘bob is never going to be used anyways.

\textbf{Question:} If we would use this definition of fact, what would \((\text{fact 5})\) be expanded to?

\textbf{Incorrect Answer:}

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

\textbf{Correct Answer:}

\[(\texttt{almost-fact \ (almost-fact \ ...)) \ 5)}\]

\section*{2.2 Better almost factorial}

You might have noticed that

\[(\texttt{almost-fact \ fact}) = \texttt{fact}\]

So \texttt{fact} is a fixed-point of \texttt{almost-fact}.

You might expect that running \[(\texttt{almost-fact \ almost-fact})\] would produce \texttt{fact} (since \texttt{almost-fact} has the same termination condition and loop structure), but notice the type violations. \texttt{almost-fact} expects a function that itself expects a number, not another function. We could, however change \texttt{almost-fact} a bit to make it accept a function that expects a function:

\begin{verbatim}
1 (lambda (f)
2   (lambda (num)
3     (if (<= num 0)
4        1
5        (* num ((f f) (- num 1))))))
\end{verbatim}

We’ll call this function \texttt{better-almost-fact}. And now we can call \[(\texttt{better-almost-fact \ better-almost-fact} \ 5)\] and get 5! (120). Note however, that \texttt{fact} isn’t the fixed point of \texttt{better-almost-fact}. The idea is that when we pass \texttt{better-almost-fact} to itself as an argument, it applies the core of \texttt{almost-fact} on itself. Just like we did with \texttt{fact_0, fact_1, fact_2, …} all the way up to \texttt{fact} (infinity).

What we managed to do is to create a looping mechanism that simulates an infinite self-application of \texttt{almost-fact}.

Now inspect a simple Y-combinator implementation:

\begin{verbatim}
1 (lambda (almost-func)
2  ((lambda (f) (almost-func (lambda (x) ((f f) x))))))
\end{verbatim}
3 (\lambda (f) (\text{almost-func} (\lambda (x) ((f f) x))))

But that looks a lot like (\text{better-almost-fact} \ \text{better-almost-fact}):

\begin{verbatim}
((\lambda (f)
  (\lambda (num)
    (if (\leq num 0)
      1
      (* num ((f f) (- num 1))))))

(\lambda (f)
  (\lambda (num)
    (if (\leq num 0)
      1
      (* num ((f f) (- num 1))))))
\end{verbatim}

Remark. In the expression (\lambda (x) ((f f) x)) in the Y-combinator definition, wrapping (f f) with a \lambda is required because Scheme is not a lazily-evaluated language.

So that means that a Y-combinator provides a way to generate looping mechanisms for an arbitrary function \text{almost-func}. This is very similar (but not identical) to while loops in other languages you know:

\begin{verbatim}
var x;
while (true) {
    //the operations of almost-func
    // which make use of x
}
\end{verbatim}

3 Fixed Points and Y

Since (Y f) returns a fixed point of f, the following expressions are all equivalent:

- (Y f)
- (f (Y f))
- (f (f (Y f)))
- (Y f \circ f)

Note that the above expressions are not necessarily equivalent to the following:
\[
\begin{align*}
\bullet & \; f \\
\bullet & \; (Y \ (ff)) \\
\bullet & \; ((Y \ Y) \ f) \\
\bullet & \; Y \circ f \\
\bullet & \; (Y \ (Y \ f)) \\
\bullet & \; (Y \circ Y \ f)
\end{align*}
\]

4 Using a Y-Combinator

We can use the above Y-combinator to define many other recursive functions. For instance we can define the Fibonacci function:

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

We need to define an almost-fib function, similar to almost-fact:

```scheme
(lambda (f)
  (lambda (n)
    (cond
      ((zero? n) 1)
      ((= 1 n) 2)
      (else (+ (f (- n 1)) (f (- n 2)))))))
```

Now we can generate fib by applying Y to almost-fib: \(f\equiv (Y \text{ almost-fib})\).