The Why of Y

Richard P. Gabriel
Lucid, Inc. and Stanford University

Did you ever wonder how Y works and how anyone could ever have thought of it? Do you feel like a Lisp weakling when some heavy-duty Scheme hacker kicks sand in your face by admiring Y in public? In this note I'll try to explain to you not only how it works, but how someone could have invented it. I'll use Scheme notation because it is easier to understand when functions passed as arguments are being applied. At the end, I'll show you Common Lisp equivalents of some of the Scheme code.

The point of Y is to provide a mechanism to write self-referential programs without a special built-in means of accomplishing it. In Scheme there are several mechanisms for writing such programs, including global function definitions and `letrec`. Here is one way to write the factorial function in Scheme:

```
(define fact
  (lambda (n)
    (if (< n 2)
        1
        (* n (fact (- n 1))))))
```

This works because there is a global variable, `fact`, that has its value set to the value of the lambda expression. When the variable `fact` in the body of the function is evaluated to determine which function to invoke, the value is found in the global variable. In some sense using a global variable as a function name is unpleasant because it relies on a global and hence a vulnerable resource—the global variable space.

The Scheme self-reference form `letrec` is usually implemented using a side effect. It is easier to reason about programming languages and programs that have no side effects. Therefore it is of theoretical interest to establish the ability to write recursive functions without the use of side effects.

The following is a program that uses `letrec`:

```
(letrec ((f (lambda (n)
              (if (< n 2)
                  1
                  (* n (f (- n 1)))))))
  (f 10))
```

This program computes 10!. The reference to f inside the lambda expression is to the binding of f established by the `letrec`.

One could implement `letrec` using `let` and `set!`.

```
(letrec ((f (lambda ...))) ...)
```

This is equivalent to the following:

```
(let ((f <undefined>))
  (set! f (lambda ...)) ...)
```

All references to f in the body of the lambda expression will refer to the value of the lambda expression.

Y is a function that takes a function that could be viewed as describing a recursive or self-referential function, and returns another function that implements that recursive function. Here is how Y is used to compute 10!.

```
(let ((f (y (lambda (h)
              (lambda (n)
                (if (< n 2)
                    1
                    (* n (h (- n 1))))))))))
  (f 10))
```

Notice that the function passed to Y as an argument is one that takes a function as an argument and returns a function that looks like the factorial function we want to define. That is, the function passed to Y is `(lambda (h) ...)`. The body of this function looks like the factorial function, except that where we would expect a recursive call to the factorial function, h is called instead. Y arranges for an appropriate value to be supplied as the value of h.

People call Y the applicative-order *fixed point operator* for functionals. Let's look at what this means in our factorial example. Suppose $\mathcal{F}$ is the true mathematical factorial function, possibly in Plato's heaven. Let F denote the following function:

```
F = (lambda (h)
      (lambda (n)
        (if (< n 2)
            1
            (* n (h (- n 1))))))
```

Then ((F $\mathcal{F}$) n)= ($\mathcal{F}$ n). That is, $\mathcal{F}$ is a fixed point of F: F maps (in some sense) $\mathcal{F}$ onto $\mathcal{F}$. Y satisfies the following property: ((F (Y F)) x) = ((Y F) x). This is a very important property of Y. The other important property is that the least defined fixed point for functionals is unique, and therefore (Y F) and $\mathcal{F}$ are in some sense the same.

Applicative-order Y is not the same as classical Y, which is a combinator. In some texts, what we call Y is called Z.

My plan is to first derive Y, which is a good way to understand it, and then to prove it's correct in the case of factorial. You can then imagine how the general proof would go. I will also show that ((F (Y F)) x) = ((Y F) x). The uniqueness property is a little too hard to do here. All of my proofs will be informal, leaving out details. Finally I will examine some extensions to Y in both Scheme and Common Lisp.

To derive Y, I will start with an example recursive function, factorial. In the derivation I will make use of three techniques. The first is to pass an additional argument to avoid using any self-reference primitives from Scheme. The second is to convert multiple-parameter functions to nested single-parameter functions in order to separate manipulation of the self-reference parameters from manipulation of ordinary parameters. The third is to introduce functions through abstraction.

All code examples will use the variables n and m to refer to integers, the variable x to refer to an unknown but undistinguished argument, and the variables f, g, h, q, and r to refer to functions.

The basic form of the factorial function is the following:

```
(lambda (n)
  (if (< n 2)
      1
      (* n (h (- n 1)))))
```

The variable h should refer to the function we wish to invoke when a recursive call is made, which is the factorial function itself. Since we have no way to have h refer directly to the correct function, let's pass it in as an argument:

```
(lambda (h n)
  (if (< n 2)
      1
      (* n (h h (- n 1)))))
```

In the recursive call to h, the first argument will also be h because we want to pass on the correct function to use in the recursive situation to later invocations of the function.

Therefore, to compute 10! we would write the following:

```
(let ((g (lambda (h n)
           (if (< n 2)
               1
               (* n (h h (- n 1)))))))
  (g g 10))
```

During the evaluation of the body of g, the value of h is the same as the value of g that the `let` established; that is, during execution of g, h refers to the executing function. When the function call `(h h (- n 1))` happens, this same value is passed along as an argument to h: h passes itself to itself.

What we want to do, though, is to pull apart the management of the self-reference to the function from the management of other arguments. In this case we want to separate the management of h from the management of n. The usual way to handle this is with a technique called *currying*. Before we curry this example, let's look at another example of currying. Here is a program that also computes 10!, but in a slightly more clever way.

```
(letrec ((f (lambda (n m)
              (if (< n 2)
                  m
                  (f (- n 1) (* m n))))))
  (f 10 1))
```

Here the trick is to use an accumulator, m, to compute the result. This function is iterative in Scheme, but that's not important. Let's curry the definition of f:

```
(letrec ((f (lambda (n)
              (lambda (m)
                (if (< n 2)
                    m
                    ((f (- n 1)) (* m n)))))))
  ((f 10) 1))
```

The idea of currying is that every function has one argument, and passing multiple arguments is accomplished with nested function application: the first application returns a function that will take the second argument and complete the computation of the value. In the above piece of code, the recursive call `((f (- n 1)) (* m n))` has two steps: the proper function to apply is computed, and then it is applied to the right argument.

We can use this idea to curry the other factorial program:

```
(let ((g (lambda (h)
           (lambda (n)
             (if (< n 2)
                 1
                 (* n ((h h) (- n 1)))))))))
  ((g g) 10))
```

In this piece of code, the recursive call also has two steps, and the first is to compute the proper function to apply. But that proper function is computed by applying a function to itself.

Applying a function to itself is how we get the basic functionality of a self-reference. The self-application (g g) in the last line of the program calls g with g itself as an argument. This returns a closure in which the variable h is bound to the outside g. This closure will take a number and do the basic factorial computation. If that computation needs to perform a recursive call, it invokes the closed-over h with the closed-over h as an argument, but all these h's are bound to the function g defined by the `let`.

We can summarize this trick. Suppose we have a self-referential function that uses `letrec` as in the following code skeleton:

```
(letrec ((f (lambda (x) ... f ...)))
  ... f ...)
```

Then this can be turned into a self-referential function that uses `let` as follows:

```
(let ((f (lambda (r)
           (lambda (x) ... (r r) ...))))
  ... (f f) ...))
```

where r is a fresh identifier.

Let's concentrate on how to further separate the management of h in our factorial function from the management of n. Recall that the factorial program looks like this:

```
(let ((g (lambda (h)
           (lambda (n)
             (if (< n 2)
                 1
                 (* n ((h h) (- n 1)))))))))
  ((g g) 10))
```

Our plan of attack is to abstract the `if` expression over (h h) and n. This will accomplish

two things: the resulting function will be independent of its surrounding bindings, and the management of the control argument will be separated from the numeric argument. The following is the result of the abstraction:

```
(let ((g (lambda (h)
           (lambda (n)
             (let ((f (lambda (q n)
                        (if (< n 2)
                            1
                            (* n (q (- n 1)))))))
               (f (h h) n))))))
  ((g g) 10))
```

We can curry the definition of f, which will also change the call to it.

```
(let ((g (lambda (h)
           (lambda (n)
             (let ((f (lambda (q)
                        (lambda (n)
                          (if (< n 2)
                              1
                              (* n (q (- n 1))))))))
               ((f (h h)) n))))))
  ((g g) 10))
```

Notice that the definition of the function f does not need to be deeply embedded in the function g. Therefore, we can extract the main part of the function—the part that computes factorial—from the rest of the code.

```
(let ((f (lambda (q)
           (lambda (n)
             (if (< n 2)
                 1
                 (* n (q (- n 1))))))))
  (let ((g (lambda (h)
             (lambda (n)
               ((f (h h)) n)))))
    ((g g) 10)))
```

Notice two things: first, the form of f is once again the parameterized form of factorial; second, we can abstract this expression over f, which produces Y as follows:

```
(define Y (lambda (f)
            (let ((g (lambda (h)
                       (lambda (x)
                         ((f (h h)) x)))))
              (g g))))
```

This is one way to derive Y.

Recall that Y is the applicative-order fixed point operator for functionals, which can be stated as follows:

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

Let's use this property to prove that the following function computes factorial:

```
(Y (lambda (f)
     (lambda (n)
       (if (< n 2)
           1
           (* n (f (- n 1)))))))
```

We will prove it using induction. Let F be as follows:

```
F = (lambda (f)
      (lambda (n)
        (if (< n 2)
            1
            (* n (f (- n 1))))))
```

We will show that $\forall n,\ n > 0$, ((Y F) n) $= n!$. Using the key property, the first step is to notice the following:

```
((Y F) x) = ((F (Y F)) x)
```

Let G be (Y F). ((Y F) 1) = ((F (Y F)) 1) = ((F G) 1). Looking at the definition of F, we see the following:

```
((F G) 1) = (if (< n 2)
                1
                (* n (G (- n 1))))
```

where n is bound to 1. Since $1 < 2$, the value of the `if` expression is 1, which is 1!. That

is the base step.

Now the induction step. Assume that ((Y F) i) = $i!$ for $1 \leq i \leq m$. Consider ((Y F) $m+1$). As before we can reduce this to the following:

```
((Y F) m+1) = ((F G) m+1) = (if (< n 2) 1 (* n (G (- n 1))))
```

where n is bound to $m+1$. Since m is at least 1, $m+1$ is at least 2, so the value of this expression is the following:

```
(* n (G (- n 1)))
```

Since G is (Y F), this reduces to the following:

```
(* n ((Y F) (- n 1)))
```

which by the induction hypothesis is just $(m+1)m! = (m+1)!$. That completes this informal proof that (Y F) is the factorial function.

Now let's prove that Y is the applicative-order fixed point operator for functionals. Namely, let's prove that $\forall\ x$ the following holds:

```
((Y F) x) = ((F (Y F)) x)
```

Recall the definition of Y:

```
(define Y (lambda (f)
            (let ((g (lambda (h)
                       (lambda (x)
                         ((f (h h)) x)))))
              (g g))))
```

Looking at the definition of Y, we see the following:

```
(Y F) = (g g) = (lambda (x)
                  ((f (h h)) x))
```

where g is the inner function in the definition of Y, f is bound to F, and h is bound to g. If we supply an argument to the result of (Y F) the following is true:

```
((Y F) x) = ((f (h h)) x)
```

where h is bound to the inner function g in the definition of Y. By substituting g for h we could rewrite this equality as follows:

```
((Y F) x) = ((f (g g)) x)
```

where g is the inner function in the definition of Y. But remembering that f is bound to F and that (Y F) = (g g), this just reduces to the following:

```
((Y F) x) = ((F (Y F)) x)
```

This is what we wanted to show. Now, we played a little free and easy with talking about the domain over which x can range, but this is just an overview, not a formal proof. When this property is coupled with the uniqueness property, we can see that Y is exactly the function we need to implement self-reference.

What does Y look like in Common Lisp? Like this:

```
(defun y (f)
  (let ((g #'(lambda (g)
               #'(lambda (x)
                   (funcall (funcall f (funcall g g)) x)))))
    (funcall g g)))
```

Y only works for functions of one argument. However, Common Lisp has a mechanism for handling a variable number of arguments, so this isn't a big problem.

```
(defun super-y (f)
  (let ((g #'(lambda (g)
               #'(lambda (&rest x)
                   (apply (funcall f (funcall g g)) x)))))
   (funcall g g)))
```

We can pass in functions of any number of arguments into super-y. Here's an example:

```
(defun fact (n)
  (funcall
    (super-y
      #'(lambda (f)
          #'(lambda (n m)
              (if (< n 2)
                  m
                  (funcall f (- n 1) (* m n))))))
    n 1))
```

Now that we have learned about Y, I want to extend it to handle two mutually recursive functions. Let's recall the definition of Y:

```
(define Y (lambda (f)
            (let ((g (lambda (h)
                       (lambda (x)
                         ((f (h h)) x)))))
              (g g))))
```

Notice that in the body of Y the call to f is built in. There is no real problem with this, but the extension requires that f be passed as an argument to the analog of g. Here is what Y would look like with this simple change:

```
(define Y (lambda (f)
            (let ((g (lambda (h r)
                       (lambda (x)
                         ((r (h h f)) x)))))
              (g g f))))
```

This function has the same effect as the previous one, but g has an extra parameter, which means that it is possible to use g to handle more than one function at a time, though Y would have to be a little different.

Y2 will be the function that is analogous to Y but which will handle a pair of mutually recursive functions. The descriptions of these mutually recursive functions will be lambda expressions with two parameters, one for each of the pair of functions. Here is a simple example of how one would use Y2:

```
((Y2 (lambda (f g) (lambda (n) (if (< n 1) 'even (g (- n 1)))))
     (lambda (f g) (lambda (n) (if (< n 1) 'odd  (f (- n 1))))))
 n)
```

In the bodies of these two functions, f refers to the first of the pair of functions, and g to the second. The above expression is intended to be equivalent to the following:

```
(letrec ((f (lambda (n) (if (< n 1) 'even (g (- n 1)))))
         (g (lambda (n) (if (< n 1) 'odd  (f (- n 1))))))
  (f n))
```

Y2 will be easier to understand if we rename some parameters in the definition of Y2, so let's do the renaming now:

```
(define y (lambda (f)
            (let ((h (lambda (h r)
                       (lambda (x)
                         ((r (h h f)) x)))))
              (h h f))))
```

Looking at the this definition of Y we can see what has to change. First, Y2 will need to have two parameters, which will be called f and g. Second, the call to r in h will need to pass two arguments, one to represent f and the other to represent g. Here is the final result:

```
(define y2 (lambda (f g)
             (let ((h (lambda (h r)
                        (lambda (n)
                          ((r (h h f) (h h g)) n)))))
               (h h f))))
```

Finally, let's look at a version of super-Y that handles any positive number of functions:

```
(defun super-yn (f &rest g)
  (let ((h #'(lambda (h r)
               #'(lambda (&rest x)
                   (apply
                     (apply
                       r
                       (funcall h h f)
                       (mapcar
                         #'(lambda (f)
                             (funcall h h f)) g))
                     x)))))
    (funcall h h f)))
```