The Why of Y

or

The Meaning of Recursion

or

The meaning of life

Eli Barzilay
eli@cs.cornell.edu

(Based on text by Matthias Felleisen, now part of “The Little Schemer”)

November, 1999

Tiny language?

We know[1] that it is possible to express:

- numbers,
- conditionals & booleans,
- cons cells & lists.

How far can we go?

[1] at least in theory, if anyone is interested, mail me and I'll post the code

Lambda Calculus

Consider **Lambda-Calulus** (the "spiritual" origin of Scheme). This is a formal language that have only the following expressions available:

Variables: $x, y, z, \ldots$;

Lambda expressions: $\lambda x.E$;

Application of expressions: $E_1 E_2$.

For example, the definition of `kons`[2]:

```
(define (kons x y) (lambda (s) (s x y)))
```

is written as:

$$\lambda x.\lambda y.\lambda s.sxy$$

[2] the `cons`-like function that uses only `lambda` expressions.

Problems with the Lambda Calculus

Some things are disturbing about the Lambda Calculus:

- The syntax is very terse — very hard to read bigger expressions,
- Only uses Lambda expressions with a single argument,
- Nothing else besides these expressions!

Our version of the Lambda Calculus

So we will fix these by the following:

- Use a Scheme-like syntax for our Lambda expressions;
- Use the fact that we can transform a program to a Curried-form using simple transformations, therefore use normal argument lists;
- Assume that we have predefined implementation of all basic things we need such as numbers, booleans, etc.

Guessing game . . .

We still have a very important feature that we have in Scheme and don't have in the Lambda Calculus.

? ? ?

Try to guess what.

No names!

What we're missing is the ability to define variables!

It looks like it is impossible to program anything **real** if we don't use symbols. Specifically, what can we do if there is no way to use free variables and bind such variables globally.

Actually we know how to convert a program so it does not uses global definitions or local `let`-bindings, for example this:

```
(define s (lambda (x) (* x 2)))
(define f
  (lambda (x)
    (let ((x1 (+ x 1)) (x2 (- x 1)))
      (s (* x1 x2)))))
(f 8)
```

gets converted to this:

```
((lambda (s f) (f 8))
 (lambda (x) (* x 2))
 (lambda (x)
   ((lambda (x1 x2) (s (* x1 x2)))
    (+ x 1) (- x 1))))
```

No recursion!

This conversion does not make everything work: it does not allow recursion — we need `letrec` for this.

This is related to the way that we introduced the Scheme `define` and `letrec` special forms back in the days of the **substitution model**:

- we waved our hands a lot at these forms hoping that you won't catch the problem,
- we gave you rules for the substitution model but following these rules strictly does not allow recursive definitions.

(You can go back to the RSM rules and get convinced by this — evaluating a recursive definition is impossible by those rules since you'd have to evaluate the symbol that you try to define.)

Even with the environment model, we used `set!` for translating `letrec`, and we used side-effects for defining variables.

No hope!

This is all trying to cover up the fact that recursion as we know it now, can only be used by a function that mentions its own name.

Is it impossible to use recursion without any form of side-effects like name bindings?

First example

Lets start now with with an example: a simple recursive function definition:

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

When we look at the **value** of `fact`, we see that by itself, it doesn't make any sense because `fact` is a free variable within the body:

```
(lambda (n)
  (if (zero? n)
    1
    (* n (fact (- n 1)))))
```

So, how is it possible to write the factorial function? (or maybe it isn't possible?)

Mathematicl example

This is similar to mathematical definitions — you **do not** have any real way to name object, **names are just short hand** for what they stand for.

For example, when you define:

$$\begin{aligned} A_0 &\equiv 1 \\ \forall n > 0 : A_n &\equiv n \cdot A_{n-1} \end{aligned}$$

you actually mean that you have this infinite list of definitions:

$$\begin{aligned} A_0 &\equiv 1 \\ A_1 &\equiv 1 \cdot A_0 = 1 \cdot 1 = 1 \\ A_2 &\equiv 2 \cdot A_1 = 2 \cdot 1 = 2 \\ A_3 &\equiv 3 \cdot A_2 = 3 \cdot 2 = 6 \\ &\vdots \end{aligned}$$

Induction & Recursion

Induction is the tool that we use to show that the above definition is actually defined for all of the natural numbers.

But when you do such an inductive proof, you are actually using name of the function as if it is already defined — **in the proof itself**.

What if defining the function is exactly what you try to do?

This is the same problem that we ran into with Lambda-Calculus.

Convenient syntax

To work on this we will continue using names — only use them as **shortcuts** for other forms, **disallowing recursive definitions**. We will use the "`:=`" meta-syntax to emphasize this, so when we write this:

```
F := (lambda (x) x)
G := (lambda (y) (F y))
(G F)
```

it is just shortcut for what we **actually mean** and were too lazy to write:

```
((lambda (y) ((lambda (x) x) y)) (lambda (x) x))
```

but this:

```
F := (lambda (x) (F x))
```

is meaningless since it stands for an infinite expression, and that is impossible.

[Note: This syntax can be used in Scheme by just using `define`s for the "`:=`" assignments, if you are careful to to have any function that references itself. It can also get converted easily to a `define`-less program as shown earlier.]

Start working on a solution

So lets begin to try writing the above factorial function:

```
fact :=
  (lambda (n)
    (if (zero? n)
      1
      (* n (fact (- n 1)))))
```

We will work by incrementally changing this, marking modifications *like this*.

However, we cannot use a recursive call, so we can write anything we want instead of the "`fact`", just to make it a valid expression. For example, use 666[3]:

```
fact :=
  (lambda (n)
    (if (zero? n)
      1
      (* n (666 (- n 1)))))
```

[3] Note that 666 **will** be a function in pure Lambda Calculus, but such application is still meaningless.

When does it work?

This function will not work in the general case, but there is **one** case where it **will** work: when `n`=0 (since then we do not reach that bogus application).

We can note this by renaming this function as "`fact0`":

```
fact0 :=
  (lambda (n)
    (if (zero? n)
      1
      (* n (666 (- n 1)))))
```

Making it work for more inputs

Now that we have that, we can use it to write "`fact1`" which is the same as our desired factorial for arguments of `0` and `1`:

```
fact1 :=
  (lambda (n)
    (if (zero? n)
      1
      (* n (fact0 (- n 1)))))
```

Remeber that this is actually shorthand for:

```
fact1 :=
  (lambda (n)
    (if (zero? n)
      1
      (* n ((lambda (n)
              (if (zero? n)
                1
                (* n (666 (- n 1)))))
            (- n 1)))))
```

And even more . . .

We can continue in this way and write "`fact2`" that will work for `n`≤3:

```
fact2 :=
  (lambda (n)
    (if (zero? n)
      1
      (* n (fact1 (- n 1)))))
```

or, in full form:

```
fact2 :=
  (lambda (n)
    (if (zero? n)
      1
      (* n ((lambda (n)
               (if (zero? n)
                 1
                 (* n ((lambda (n)
                          (if (zero? n)
                            1
                            (* n (666 (- n 1)))))
                        (- n 1)))))
             (- n 1)))))
```

Is there an end to this . . . ?

If we continue this way we **will** get the true `fact` function,

But: the problem is that to handle any possible integer argument, it will still have to be an infinite one!

Here is what it is supposed to look like:

```
fact0 := (lambda (n) (if (zero? n) 1 (* n (666 (- n 1)))))
fact1 := (lambda (n) (if (zero? n) 1 (* n (fact0 (- n 1)))))
fact2 := (lambda (n) (if (zero? n) 1 (* n (fact1 (- n 1)))))
fact3 := (lambda (n) (if (zero? n) 1 (* n (fact2 (- n 1)))))
...
```

And our `fact` is actually $\mathtt{fact}_\infty$, with an infinite size.

— back to the original problem . . .

A little hope

We can see a faint beam of light:
This bigger and bigger definition uses multiple instances of the same original `fact` code, so we can try to abstract this away — pull the value that is being used as the internal call as an argument to a function.

Rule#1:

```
(... y ...)
```

$$\Updownarrow$$

```
((lambda (x) (... x ...))  y)
```

— making sure that no variables get captured.

Use Rule#1

`fact1` now becomes:

```
fact1 :=
  ((lambda (fact)
     (lambda (n)
       (if (zero? n)
         1
         (* n (fact (- n 1))))))
   fact0)
```

Which is actually:

```
fact1 :=
  ((lambda (fact)
     (lambda (n)
       (if (zero? n)
         1
         (* n (fact (- n 1))))))
   ((lambda (fact)
      (lambda (n)
        (if (zero? n)
          1
          (* n (fact (- n 1))))))
    666))
```

Make it look even better

This way we do have something that looks better, but we still repeat ourselves.

To solve this problem, we'll use a function that will take the `(lambda (n) ...)` expression and will apply that on `666`, `fact0`, or whatever.

Rule#2:

```
(f x)  ⟹  ((lambda (g) (g x)) f)
```

— actually, an instance of Rule#1.

Use Rule#2

Use this to create a function that gets `make-fact`:

```
fact0 :=
  ((lambda (make-fact)
     (make-fact 666))
   (lambda (fact)
     (lambda (n)
       (if (zero? n)
         1
         (* n (fact (- n 1)))))))
```

Now `fact1` can be written easily:

```
fact1 :=
  ((lambda (make-fact)
     (make-fact (make-fact 666)))
   (lambda (fact)
     (lambda (n)
       (if (zero? n)
         1
         (* n (fact (- n 1)))))))
```

And the principle is clear.

Back to the "real" `fact`

We can now continue by working on the "real" `fact`, which is still infinite at this stage:

```
fact :=
  ((lambda (make-fact)
     (make-fact
      (make-fact
       (... (make-fact 666) ...))))
   (lambda (fact)
     (lambda (n)
       (if (zero? n)
         1
         (* n (fact (- n 1)))))))
```

But the infiniteness problem is now localized: we look for a **finite** expression that will evaluate to

```
(f (f (f (... (f x) ...))))
```

Continue please . . .

Now, examine the `(lambda (make-fact) ...)` — all it does is get something and apply it to the result of applying it to the result of applying it . . . to 666.

So, if we succeed, `make-fact` does what we want:

```
(make-fact f) ⟶
   (f (f (f (...(f x) ...))))
```

We can now use this idea: pass `make-fact` itself to `make-fact`, and have it do the extra calls **itself** when needed.

Let `make-fact` do the work

Now the second closure gets `make-fact` itself, so the internal `fact` variable is renamed to make this clearer, and we initially apply it on 666:

```
fact :=
  ((lambda (make-fact)
     (make-fact make-fact))
   (lambda (make-fact)
     (lambda (n)
       (if (zero? n)
         1
         (* n ((make-fact 666)
               (- n 1)))))))
```

Still problems . . . and a solution!

That will make this function do the same as `fact1` — if we try to continue, we'll bump into 666, a reduction will demonstrate this (you're invited try this) . . .

Instead — use `make-fact` instead of 666 and we'll be able to do as many calls as needed:

```
fact :=
  ((lambda (make-fact)
     (make-fact make-fact))
   (lambda (make-fact)
     (lambda (n)
       (if (zero? n)
         1
         (* n ((make-fact make-fact)
                (- n 1)))))))
```

Voila! The ZF-1!

Now, we have something which **is** `fact` (convince yourself by fixing the reductions made with 666).

So we see that it is possible to write recursive functions using finite expressions!

— But we still have some problems to overcome . . .

What problems?

First, there is still the problem of having a solution which is quite different from the original factorial function.

To make things more clear, we use more abstractions.

First, abstract the second lambda expression, putting the `(make-fact make-fact)` call outside the expression (Rule#1) so we get the `fact`'s original body in one piece:

```
fact :=
  ((lambda (make-fact)
     (make-fact make-fact))
   (lambda (make-fact)
     ((lambda (fact)
        (lambda (n)
          (if (zero? n)
              1
              (* n (fact (- n 1))))))
      (make-fact make-fact))))
```

Still, we have more problems . . .

More problems??

The problem now is that if you will type this in — you'll never see the prompt again . . .

This is because we use **eager** (applicative-order) evaluation — the rule that says that you **first** evaluate the function and its arguments, **then** do the apply step.

What we just did is wrote an expression that looks like:

```
((lambda (x) (x x))
 (lambda (x) (f (x x))))
```

And this is a variation on the well-known and much loved expression:

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

. . . which evaluates to itself . . . forever . . .

((λx . x x) (λx . x x))

This expression is the key for creating a loop — we use it to create the recursion. The original expression evaluates as follows:

```
((lambda (x) (x x)) (lambda (x) (f (x x))))
((lambda (x) (f (x x))) (lambda (x) (f (x x))))
(f ((lambda (x) (f (x x))) (lambda (x) (f (x x)))))
(f (f ((lambda (x) (f (x x))) (lambda (x) (f (x x))))))
(f (f (f ((lambda (x) (f (x x))) (lambda (x) (f (x x)))))))
⋮
```

The problem is that we must **delay** the evaluation of the looping expression until we actually need more `f`'s to avoid an infinite loop.

The solution for this is to use `lambda` to delay evaluation[4]. Use this rule:

Rule#3:

$$\texttt{f} \implies \texttt{(lambda (z) (f z))}$$

— as long as f is a one-arg function.

[4]This is similar to the way we get streams.

Use Rule#3

So we will use this wrapper on the "`(x x)`":

```
((lambda (x) (x x))
 (lambda (x) (f (lambda (z) ((x x) z)))))
```

and when we evaluate this we get to something similar to this:

```
(f (lambda (z)
     (((lambda (x) (f (x x)))
       (lambda (x) (f (x x))))
      z)))
```

Except that we have `(lambda (z) ((x x) z))` instead of the original `(x x)` so this is really:

```
(f (lambda (z)
     (((lambda (x)
         (f (lambda (z) ((x x) z))))
       (lambda (x)
         (f (lambda (z) ((x x) z)))))
      z)))
```

Some more

. . . Now the loop is deferred inside the lambda expression — and only when we need to use it, we apply it and evaluate its body. The result will still be the same.

Note that this wouldn't be a problem if we would use **lazy** (normal-order) evaluation which is the general case with Lambda Calculus.

We can also note that nothing changes if we modify the first part of:

```
((lambda (x) (x x))
 (lambda (x) (f (lambda (z) ((x x) z)))))
```

to be identical with the second part for the sake of symmetry — the evaluation still looks the same except for skipping the first step now:

```
((lambda (x) (f (lambda (z) ((x x) z))))
 (lambda (x) (f (lambda (z) ((x x) z)))))
```

Back to the factorial

Applying the delay fix to the above factorial definition gives:

```
fact :=
  ((lambda (make-fact)
      (make-fact make-fact))
   (lambda (make-fact)
      ((lambda (fact)
          (lambda (n)
             (if (zero? n)
                 1
                 (* n (fact (- n 1))))))
       (lambda (z)
         ((make-fact make-fact) z)))))
```

Yet more

And the symmetric version is:

```
fact :=
  ((lambda (make-fact)
     ((lambda (fact)
        (lambda (n)
          (if (zero? n)
              1
              (* n (fact (- n 1))))))
      (lambda (z)
        ((make-fact make-fact) z))))
   (lambda (make-fact)
     ((lambda (fact)
        (lambda (n)
          (if (zero? n)
              1
              (* n (fact (- n 1))))))
      (lambda (z)
        ((make-fact make-fact) z)))))
```

This does look more complex — but it is actually simpler since the two parts are the same.

Make it simple again

Now take the "`(lambda (fact) ...)`" thing, and call it `fooct`, we get:

```
fooct :=
  (lambda (fact)
    (lambda (n)
      (if (zero? n)
        1
        (* n (fact (- n 1))))))

fact :=
  ((lambda (make-fact)
     (fooct
      (lambda (z)
        ((make-fact make-fact) z))))
   (lambda (make-fact)
     (fooct
      (lambda (z)
        ((make-fact make-fact) z)))))
```

Even better

Using Rule#1, make `fooct` a variable, call the result function `make-real-fact` since `fact` is now the result of applying it:

```
fooct :=
  (lambda (fact)
    (lambda (n)
      (if (zero? n)
        1
        (* n (fact (- n 1))))))

make-real-fact :=
  (lambda (fooct)
    ((lambda (make-fact)
        (fooct
         (lambda (z)
           ((make-fact make-fact) z))))
      (lambda (make-fact)
        (fooct
         (lambda (z)
           ((make-fact make-fact) z))))))
```

And now: `fact`

And now the **real** `fact` can be defined with:

```
fact := (make-real-fact fooct)
```

This works — and the whole thing is not recursive! (Actually it is, but it is not **defined** recursively).

The last thing to note is that `make-fact` is totally independent of `fact`, so we can redefine it as a general function:

```
make-recursive :=
  (lambda (f)
    ((lambda (x)
       (f (lambda (z) ((x x) z))))
     (lambda (x)
       (f (lambda (z) ((x x) z))))))

fact := (make-recursive fooct)
```

We now have general recursion!

This also work for other recursive functions:

```
foob :=
  (lambda (fib)
    (lambda (n)
      (if (<= n 1)
        n
        (+ (fib (- n 1))
           (fib (- n 2))))))
fib := (make-recursive foob)
```

And another example, without an explicit ‘`loongth`’ definition:

```
length :=
  (make-recursive
    (lambda (length)
      (lambda (l)
        (if (null? l)
          0
          (+ (length (cdr l)) 1)))))
```

The “Y” Combinator

Our “`make-recursive`” function is usually called the **fixpoint operator** or the **Y combinator**.

It looks really simple when using the lazy version (remember: our version is the eager one):

```
Y := (lambda (f)
      ((lambda (x) (f (x x)))
       (lambda (x) (f (x x)))))
```

And this all comes from the loop generated by:

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

The core of looping

`((λx . x x) (λx . x x))` is also the idea behind many deep mathematical facts. (as you will discover in future courses.)

As an example, follow the next rule:

> `I will say the next sentence twice`:
> “`I will say the next sentence twice`”.

(Note the usage of colon for the first and quotes for the second — what is the equivalent of that in Lambda-Calculus?)

Final note: Here is a function that returns **itself** (not its code):

```
(Y (lambda (f) (lambda (x) f)))
```

which is the same as:

```
(define (f) f)
```

in Scheme.

Didn't have enough?

Final exercises for whoever is interested (and still alive):

1. Write an expression that returns its own **S-expression** when evaluated.
   A similar problem with C and strings:

   ```
   char *s="char *s=%c%s%c;%cmain()printf(s,34,s,34,10,10);%c";
   main()printf(s,34,s,34,10,10);
   ```

2. Our make-recursive function can only work on one-argument functions. Why? Write a version for two arguments.

3. How can mutual recursion be implemented using this idea?