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")

December 2002

Tiny language?

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

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

using **only** functions.

The question is: **How far can we go?**

[1] At least in theory, if anyone is interested, tell me.

Lambda Calculus

Consider **Lambda-Calulus** (the "spiritual" origin of functional programming). This is a formal language that has nothing but functions. Formally, it is made of these expressions:

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

Lambda expressions: $\lambda x.E$;

Application of expressions: $E_1 E_2$.

For example, we can define a pair type using the following definitions:

$$
\begin{aligned}
\texttt{pair} &\equiv \lambda x.\lambda y.\lambda s.sxy \\
\texttt{fst} &\equiv \lambda p.(p\ \lambda x.\lambda y.x) \\
\texttt{snd} &\equiv \lambda p.(p\ \lambda x.\lambda y.y)
\end{aligned}
$$

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;
- This is the simple untyped version of the lambda-calculus;
- Nothing else besides these expressions!
  (So we can make pairs — of what??)

Our version of the Lambda Calculus

So we fix these by the following:

- Use an ML-like syntax for our Lambda expressions, for example, use curried argument lists for simplicity;
- Assume that we have some “built-in” values and functions, like numbers, booleans, and functions that use them.

For example, for the above pair we define:

```
pair  ≡  fun x y → fun s -> s x y
 fst  ≡  fun p → p (fun x y -> x)
 snd  ≡  fun p → p (fun x y -> y)
```

Notes:

- I will actually use OCaml syntax, should be close enough;
- Some of the expressions I will use aren’t properly typed, I will return to this in the end.

Guessing game...

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

? ? ?

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 can't define variables and functions.

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

```
val dbl = fun x -> x*2 ;;
val foo = fun x ->
            let x1 = x+1 and x2 = x-1 in
              dbl (x1*x2) ;;
foo 8 ;;
```

gets converted to this:

```
((fun dbl foo -> foo 8)
 (fun x -> x*2)
 (fun x -> ((fun x1 x2 -> dbl (x1*x2)) (x+1) (x-1))))
```

No recursion!

This conversion does not make everything work: it does not allow recursion — we need some form of `fun` and `let rec` for this.

This is related to the way that the substitution model was defined:

- there was a lot of hand-waving around how they actually work,
- you had substitution rules but recursion was always fishy.

(In your version, the fishy part was probably the question of where you stop substituting the defined function.)

So the big question is:

Can we do 'real' computations without recursive definitions?

(You know that without it you don't have any way of looping.)

First example

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

```
val fact =
  fun n -> if n=0 then 1
                  else 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, which is:

```
fun n -> if n=0 then 1
                else n * fact(n-1)
```

So, how is it possible to write the factorial function?

(or maybe it isn’t possible after all?)

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 and Recursion

Induction is the major 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 the 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 instead of '`fun`' or `val`, so when we write this:

```
F := fun x -> x
G := fun y -> F y
(G F)
```

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

```
((fun y -> (fun x -> x) y) (fun x -> x))
```

but remember that this:

```
F := fun x -> (F x)
```

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

[Note that this syntax can be used in ML by not using any recursive definitions, and these programs can always be converted to a single **closed** expression.]

Start working on a solution

So lets begin to try writing the above factorial function:

```
fact :=
  fun n -> if n=0 then 1
                  else n * fact(n-1)
```

We will work by incrementally changing this, marking modifications like <u>this</u>.

Begin by noting that we cannot use a recursive call, so we can write anything we want instead of the internal "`fact`", just to make it a valid expression. For example, use `666`[2]:

```
fact :=
  fun n -> if n=0 then 1
                  else n * 666(n-1)
```

[2] Note that `666` **will** be a function in pure Lambda Calculus, but that application will still be meaningless. Also note that it is less fashionable than **42** in a CS crowd, I don't care.

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 :=
  fun n -> if n=0 then 1
                  else n * 666(n-1)
```

Making it work for more inputs

Now that we have a factorial that works for **0**, we can use it to write "`fact1`" which is works for `0` and `1`:

```
fact1 :=
  fun n -> if n=0 then 1
                  else n * fact0(n-1)
```

Again, remeber that this is actually shorthand for:

```
fact1 :=
  fun n -> if n=0
    then 1
    else n * ((fun n -> if n=0 then 1
                                else n * 666(n-1))
              (n-1))
```

And even more...

We can continue in this way and write "`fact2`" that will work for $0 \leq n \leq 3$:

```
fact2 :=
  fun n -> if n=0 then 1
                  else n * fact1(n-1)
```

or, in its real full g(l)ory:

```
fact2 :=
  fun n ->
    if n=0 then 1
           else n * (fun n -> if n=0
                       then 1
                       else n * ((fun n -> if n=0 then 1
                                                  else n * 666(n-1))
                                 (n-1))
                    (n-1))
```

Can we get to the holy grail...?

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 := fun n -> if n=0 then 1 n * 666(n-1)
fact1 := fun n -> if n=0 then 1 n * fact0(n-1)
fact2 := fun n -> if n=0 then 1 n * fact1(n-1)
fact3 := fun n -> if n=0 then 1 n * fact2(n-1)
  ⋮
```

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

— back to the original problem. . .

A little hope

Here is 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 with a function — pull the value that is being used as the internal call as an argument to a function.

Rule 1:

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

$$\Updownarrow$$

```
((fun x -> (... x ...))  y)
```

— making sure that no variables get captured.

Use Rule 1

`fact1` now becomes:

```
fact1 :=
  ((fun fact ->
     (fun n -> if n=0 then 1
                      else n * fact(n-1)))
   fact0)
```

Which is actually:

```
fact1 :=
  ((fun fact ->
     (fun n -> if n=0 then 1
                      else n * fact(n-1)))
   ((fun fact ->
      (fun n -> if n=0 then 1
                       else 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 `(fun n` $\rightarrow$ `...)` expression and will apply that on `666`, `fact0`, or whatever.

Rule 2:

$$\texttt{f(x)} \implies \texttt{((fun g -> g(x)) f)}$$

— actually, an instance of Rule 1, used for a function.

Use Rule 2

Use this to create a function that gets a `makeFact` argument and uses it to create the result, start from `fact0`:

```
fact0 :=
  ((fun makeFact -> makeFact(666))
   (fun fact ->
     (fun n -> if n=0 then 1
                      else n * fact(n-1))))
```

Now `fact1` can be written easily:

```
fact1 :=
  ((fun makeFact -> makeFact(makeFact(666)))
   (fun fact ->
     (fun n -> if n=0 then 1
                      else n * fact(n-1))))
```

And the principle should clear.

Back to the "real" factorial

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

```
fact :=
  ((fun makeFact ->
      makeFact(makeFact(...makeFact(666)...)))
   (fun fact ->
     (fun n -> if n=0 then 1
                      else 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 `(fun makeFact -> ...)` expression — 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, `makeFact` does what we want:

```
makeFact(f) ⟶
  f(f(f(...f(x)...)))
```

⋆ Main Idea ⋆

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

Let ‘makeFact’ do the work

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

```
fact :=
  ((fun makeFact -> makeFact(makeFact))
   (fun makeFact ->
     (fun n -> if n=0 then 1
                      else n * makeFact(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 using simple substitution rules will demonstrate this (that would be a good workout).

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

```
fact :=
  ((fun makeFact -> makeFact(makeFact))
   (fun makeFact ->
     (fun n -> if n=0 then 1
                      else n * makeFact(makeFact)(n-1))))
```

Voila! The ZF-1!

Now we have something which **is** `fact` (convince yourself by fixing the previous workout by replacing `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 `makeFact(makeFact)` call outside the expression (Rule 1) so we get the `fact`'s original body in one piece:

```
  fact :=
    ((fun makeFact -> makeFact(makeFact))
     (fun makeFact ->
       (fun fact ->
         fun n -> if n=0 then 1
                          else n * fact(n-1))
       (makeFact(makeFact))))
```

We can do even better...

Better?

In that last expression, we had that internal factorial seed function that can be easily isolated, getting a more uniform expression, name it `iFact`:

```
iFact := fun fact ->
           fun n -> if n=0 then 1
                                else n * fact(n-1)
fact := ((fun makeFact -> makeFact(makeFact))
         (fun makeFact -> iFact(makeFact(makeFact))))
```

and just to make things look even more uniform, we can reduce the `fact` expression once to get:

```
iFact := fun fact ->
           fun n -> if n=0 then 1
                                else n * fact(n-1)
fact := ((fun makeFact -> iFact(makeFact(makeFact)))
         (fun makeFact -> iFact(makeFact(makeFact))))
```

Improving names

So the mechanism that implements recursion isn't really related to fact, we can just as well replace its input with a generic 'x':

```
iFact := fun fact ->
           fun n -> if n=0 then 1
                            else n * fact(n-1)
fact := ((fun x -> iFact(x(x)))
         (fun x -> iFact(x(x))))
```

And since it can be used with any function, it would be useful to pull out a function that does that on any input function:

```
iFact := fun fact ->
           fun n -> if n=0 then 1
                            else n * fact(n-1)
makeRec := fun f ->
             ((fun x -> f(x(x))) (fun x -> f(x(x))))
fact := makeRec(iFact)
```

We now have one last unfinished business to take care of.

One last problem

This should all work well on paper, if you're not careful enough. The last problem is that if you will type this in — you'll never see the prompt again, or you might see a stack overflow.

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.

If ML tries to evaluate the `makeRec(iFact)` expression, it gets into this loop that begins with:

```
((fun x -> iFact(x(x))) (fun x -> iFact(x(x))))
```

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

```
((fun x -> x(x)) (fun x -> x(x)))
```

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

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

This expression is the key for creating a loop — we use it to create the recursion. The original '`makeRec(iFact)`' expression evaluates as follows:

```
(fun x -> f(x(x)))(fun x -> f(x(x)))
f((fun x -> f(x(x)))(fun x -> f(x(x))))
f(f((fun x -> f(x(x)))(fun x -> f(x(x)))))
f(f(f((fun x -> f(x(x)))(fun 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. If we would have used a lazy language, we would be done.

The standard way to delay evaluation is . . . using a function[3]. Use this rule:

Rule 3:

$$\texttt{f} \implies \texttt{(fun z -> (f z))}$$

— as long as f is a one-arg function.

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

Use Rule 3

Using this modification we wrap the ‘`(x x)`’ and get the final working version:

```
iFact := fun fact ->
           fun n -> if n=0 then 1
                           else n * fact(n-1)
makeRec := fun f ->
             ((fun x -> f(fun z -> x(x)(z)))
              (fun x -> f(fun z -> x(x)(z))))
fact := makeRec(iFact)
```

More examples

Using this we can define any recursive function, for example:

```
iFib := fun fib ->
           fun n -> if n<=0 then n
                                else fib(n-1) + fib(n-2)
fib := makeRec(iFib)
```

and

```
length :=
  makeRec(fun length l ->
            if l=[] then 0 else length(List.tl(l)))
```

The “Y” Combinator

Our “`makeRec`” 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 := fun f -> ((fun x -> f(x x)) (fun x -> f(x x)))
```

In any case, its main property is that

$$Y(f) = f(Y(f))$$

And this all comes from the loop generated by:

```
((fun x -> x x) (fun 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 (fun f -> (fun x -> f)))
```

which is actually the same as:

```
fun f x = f
```

in ML.

Didn’t have enough?

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

1. This final version does not type check — why, and how can this be bypassed?

2. Our `makeRecursive` 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, how?

Finally... The Meaning of Life!