Design Doc - Recursion

Contents

A cute sounding but essentially meaningless quote

> *In order to understand unbounded recursion, one must understand unbounded recursion.*

Recursion is nameless

The implementation of recursion in LETREC is whack at worst and whack at best.

Here's factorial, for example :

```
((fix (lambda (self)
                (lambda (n)
                  (if (zero? n)
                      1
                      (* n (self (- n 1)))))))) 5)
; reduces to 120
```

For starters, LETREC allows recursive definitions without names. Secondly, what on earth is this `fix` form? Am I admitting to the fact that the way I've implemented recursion requires `fix`ing? Also, what is this `self`? How can a procedure take *itself* as it's parameter? That makes no sense!

Recursion as an equation

In order to justify my case of a very whacky implementation of recursion (the honest justification is laziness but using abstract mathematics makes it look otherwise) I will start with the classic example of a while loop.

Let us say we have a programming language like follows :

```
bool-expr ::= true | false | bool and bool | bool or bool | not bool
int-expr ::= integer | int-expr + int-expr | int-expr * int-expr | int-expr - int-expr
expr ::= bool-expr | int-expr
assignment ::= variable <- expr
command ::= skip | assignment | if bool-expr then command else command
                 | while bool-expr do command | command ; command
```

Basically integers, booleans, if-else, assignments and while loops and a `skip` command that skips over to the next operation. (think empty semicolon `;` in langauge like C++)

Observe that `while` loops would, then, satisfy the following equation :

```
while B do C = if B then { C ; while B do C }
               else skip
```

Parametrising over `B` (`B` ranges over `bool-expr`) and `C` (ranges over `command`) we get that the `while` form $W_{B,C}$ is of the form

$$W_{B,C} = \phi(W_{B,C})$$

where ϕ takes any form `f` to the `if B then { C ; f } else skip` form.

More formally, `while` is completely defined by the above equation and we say that $W_{B,C}$ is the *fixpoint* of $\phi_{B,C}$. Such equations are called *fixpoint equations*.

We have an entire field of mathematics dedicated to the study of fixpoints - domain theory. It should be no surprise that since recursion is *defined* mathematically

using fixpoint equations; domain theory is a favourite subject of computer scientists.

Recursion is defined recursively

Let us go to the definition of factorial in a classical programming language like Python :

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n * fact(n - 1)
```

Observe that in this definition is implicit the fact that for `fact(n)` to be well-defined we need a proof of termination of `fact(n - 1)`. It should also be evident that while not explicitly parametrised over some form of `self`, this definition still implicitly adds `fact` to the environment in which it is defined.

Any implementation of a programming langauge is inevitably written in *some* langauge (the metalanguage) and practically every metalanguage will have recursion. This allows the implementor to exploit the recursion already present in the metalanguage to define the operational semantics of recursion in the object language.

Recursive functions as fixpoints?

If any recursive definition can be written as a fixpoint equation and doing so allows us to talk about recursion without circular definitions - the natural question to ask is what about recursive functions? The ubiquitous form of recursion must also be described by this.

In order to see the sort of equation that defines recursive functions, let us make a few things more explicit in our older factorial function :

```
def fact(self, n):
  if n == 0 :
    return 1
  else :
    return n * self(self, n - 1)
```

Observe that if `self` is only capable of computing the factorial of $n-1$ that too is good enough - we can compute the factorial of n through that. So what we have established is that given a function that can compute the factorial of $n-1$ it is possible to construct a function that can compute the factorial of n.

This is starting to sound suspiciously similar to induction!

We need a higher-order function - one that takes an approximation of factorial that can compute upto $n!$ and returns a new function that can compute $(n+1)!$.

This leads to a certain strategy - recursive functions must be fixpoints of *higher-order functions*.

Let us try to actually write down such a function :

```
def fact_combinator(approx, n):
  def base_case(input):
    return 1
  def inductive_case(input):
    if input == n :
      return n * approx(n - 1)
    return approx(input)
  if n == 0:
    return base_case
  return inductive_case
```

If given the nth approximation to factorial, this function will return the $(n+1)$th approximation.

What about our *first ever approximation*? Since this procedure is never invoked we can put it as anything : even a function that doesn't even terminate! Indeed, the "zeroth" approximation is chosen as the "empty function" - the one that doesn't return for any input whatsoever :

```
def empty(n):
  return empty(n)
```

Observe that using the first ever approximation is a very general trick that will work for *any fixpoint equation* rather than just the one for factorials!

Finding fixpoints - the mathematics

Technical Jargon

The only real theorem we need is that for any given bounded partial order, any monotone sequence converges to a certain limit that can be calculated using the join or the meet operation as appropriate.

I'm going to be intentionally informal here and that probably sounded like Greek.

A partial order is essentially a generalisation of the familiar $\leq$ operator for abstract mathematical objects rather than just numbers. An important restriction is that for any two a and b it is not necessary that $a \leq b$ or $b \leq a$.

A good example is the $|$ "divides" operation on natural numbers. Every number is divisible by itself. If a is divisible by b and b is divisible by c then a is divisible

by c. Note that for any two a and b it is not necessary that a divides b or that b divides a - it might be the case that neither is true.

Another good example from computer science is the topological ordering of a graph.

What's a *bounded* partial order? Essentially, whatever objects we are considering have a "maximum" and a "minimum" in the sense that $\exists a.\forall b.a \leq b$ and $\exists a.\forall b.b \leq a$.

Now, what's a *monotone sequence*? Well any sequence a_n (finite or infinite) such that either $a_i \leq a_{i+1}$ or $a_{i+1} \leq a_i$.

What we're saying is that *any* such sequence always *converges*. Of course, I have not actually defined convergence. One way of thinking about it is that there is one *unique* element that to which the elements in our sequence appear to get closer to. For reasons of brevity I will not go too much into the discussion about convergence and instead state the result and we'll accept it as an axiom. Also, since we will only be talking a particular kind of partial order - the partial order of subsets of a set, I will only talk about that.

Observe that given two subsets of some set U, say A and B, the relation $\subseteq$ is a partial order!

Also, the empty set $\emptyset$ is the subset of any subset of U and the subset U is the superset of any subset of U. It is not *just* a partial order - it is a *bounded partial order*!

Now we will state the theorems in their entire glory.

Assume that F_n is a monotone sequence of subsets of some universe U. Specifically, $F_n \leq F_{n+1}$.

Then we define the *limit* of this sequence as the countable union of it's elements :

$$\lim_{n\to\infty} F_n = \bigcup_{i=1}^{\infty} F_i$$

This will suffice for our purposes.

A monotone sequence of approximations

Remember our old sequence of approximations of the factorial function?

This sequence starts with the empty function and we have a combinator $\mathbb{F}$ that can take the nth approximation and return the $(n+1)$th approximation $(\mathbb{F}(F_n) = F_{n+1})$.

One way to model these approximations is as *partial functions*. Conventional functions have an output defined for *any* input but partial functions have an

output defined for *some* inputs. Our approximation F_n can really only produce outputs for inputs upto n.

Sparing some details, this means that F_n is a function defined for a subset of the natural numbers $(0 \dots n)$ and the base case is defined for the empty set. Also, since functions themselves are sets (of input-output pairs) we can ask whether two functions' domains are subsets of each other or not. This means that our approximations form the good-old partial ordering we defined in the last section.

The function with the empty set as the domain acts as the lower bound and any function with the entire universe (in our case $\mathbb{N}$) is the upper bound. So this partial order is also bounded! It is also a subset partial order so our old form applies too!

I will not prove it but the solution to our recursive equation, our fixpoint to the combinator defined earlier is *exactly* the limit of this sequence. But this limit is guranteed to exist since the sequence is guranteed to converge!

So the factorial function, a function that computes the factorial of any given natural number is precisely defined by the following operation :

$$F = \lim_{n \to \infty} F_n = \bigcup_{i=1}^{\infty} F_i$$

Wow, all this just to *define* mathematically recursion without actually using recursion. This might look unnecessarily complex (it is) but it's still important since this gives a totally new perspective on what recursion is rather than just "stack go brr".

Finding fixpoints - the computer science

During the last section we dived into what some might call "heavy mathematics". It might be difficult to relate it back to CS.

Our first question is - does the fixpoint *always* exist? Does any arbitrary recursive definition produce a valid term? (spoiler alert : yes). Secondly, if the fixpoint does exist, can we write a computer program to actually *compute it*? (spoiler alert : yes).

Curry's Y Combinator

This section assumes some basic knowledge about the untyped lambda calculus.

We define the (in)famous Y combinator as follows :

$$Y = \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$$

The Y combinator is the most popular of a family *fixpoint combinators*. A combinator is a lambda term with no free variables. Observe that our definition indeed does not contain any free variables. The Y combinator has the property that $Yf = f(Yf)$ for any term f. This means that the Y combinator takes any lambda abstraction and returns it's fixpoint.

Of course, we have not considered other important questions - is the fixpoint unique? If the fixpoint is not unique, then which fixpoint is returned by the Y combinator? I'll leave all this for now.

Let us take an example : let's compute the fixpoint of the identity function. The term $\lambda y.y$ will represent the identity function.

$$\begin{aligned} Yid \\ &= Y\lambda y.y \\ &= \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))\lambda y.y \\ &= ((\lambda x.(\lambda y.y(xx)))(\lambda x.(\lambda y.y(xx)))) \\ &= ((\lambda x.(xx))(\lambda x.(xx))) \end{aligned} \tag{1}$$

Observe that the last term is the classic example of a non-terminating infinite loop.

This makes sense, we only expected that the Y combinator will produce a well-formed term for any recursive definition; we cannot gurantee that this well-formed term be terminating.

Actually implementing the recursion in our language

One might argue that actually computing the fixpoint for a recursive function this way seems to be a bad idea. In this argument, they would be correct. Actually computing fixpoints would be slow and inefficient. A better idea is to simply use the recursion present in the metalanguage.

I have taken the *call-by-value* flavour of the Y-combinator. Scheme has call-by-value semantics and the
object language has inherited these semantics.

The call-by-value Y-combinator is defined as follows :

$$Y = \lambda f.((\lambda x.f(\lambda y.(xx)y))(\lambda x.f(\lambda y.(xx)y)))$$

In the interpreter this is defined as a primitive by evaluating a simple parse tree in the empty environment :

```
(define y-combinator
    (value-of (parse-tree '(lambda (f)
                             ((lambda (x)
```

```
                                (f (lambda (y)
                                     ((x x) y))))
                              (lambda (x)
                                (f (lambda (y)
                                     ((x x) y)))))))
          (empty-env)))
```

This means that fixpoints of recursive functions is computed with the Y combinator.

Conclusion

I know very well that I'll have to implement proper fixpoint handling and proper recursion handling since algebraic datatypes require so. Nonetheless, this works well enough for now and I can add a new `defrec` form as syntactic sugar that reduces to the `fix` form.

I have also been very informal and non-rigourous with the mathematics. I will be fixing both of these very soon.

One might argue that putting in so much mathematics and extra garbage in this design doc is unnecessary - but this fundamentally misunderstands the philosophy. Since the design is motivated entirely by personal laziness, the design doc itself is essentially a glorified shitpost.

References

- For understanding about the untyped lambda calculus, one may see *Type Theory and Formal Proof* or *Lectures on the Curry-Howard Isomorphism*
- For understanding about general recursion and algebraic datatypes see Pierce's *Types and Programming Languages*
- For understanding the mathematics behind recursion see any book on domain theory or programming language semantics.