To this point, we have seen how to encode a variety of constructs in λ-calculus, including numbers, conditionals, and pairs. We can also essentially write simple for loops using Church numerals by defining the body of the loop as a function, applying a Church numeral to it, and then running that on an initial value. We have also seen how to encode an infinite loop as Ω. What we have not seen is how to write (interesting) unbounded loops or recursive functions.

1 Building a Fixed Point

In functional languages like OCaml, we could write a recursive factorial function over natural numbers as follows.

$$\mathsf{letrec\ fact\ n}\ =\ \mathsf{if\ (n == 0)\ then\ 1\ else}\ n * (\mathsf{fact}\ (n-1))$$

But how do we write this in λ-calculus where all functions are anonymous? We know how to encode conditionals and the basic arithmetic operations of checking if a number is zero, multiplication, and predecessor, but we must somehow construct a λ-term fact that satisfies the equation

$$\mathsf{fact}\ =\ \lambda n.\,\mathsf{if}\ (\mathsf{iszero}\ n)\ \mathsf{then}\ \overline{1}\ \mathsf{else}\ (\mathsf{mult}\ n\ (\mathsf{fact}\ (\mathsf{pred}\ n))).$$

As a first step, we could try giving the function a name, let's say f, and define a function F that, if we gave it a correct implementation of fact would give us what we needed:

$$F\ \triangleq\ \lambda f.\,\lambda n.\,\mathsf{if}\ (\mathsf{iszero}\ n)\ \mathsf{then}\ \overline{1}\ \mathsf{else}\ (\mathsf{mult}\ n\ (f\ (\mathsf{pred}\ n))).$$

When we were defining the denotational semantics of while loops, we built a transformation function like this and then found a fixed point of it by building better and better approximations. It would be nice to do something similar here, but we cannot build a λ-term by taking the limit of mathematical functions. Instead we need do something a little different.

By construction, we need to give F something that behaves like fact. If we could just give it fact, we would be done. We don't have fact yet, but we do have something very similar: F. So would we get a correct implementation if we just passed F to itself?

If we do so blindly, things do not work. The body of F applies its argument f directly to a number, but if we give it F it expects an extra argument (the function) before a number. However, the function it needs is one that behaves like fact, which is exactly what we passed in as f! Therefore, we can just tweak the definition of F so that, instead of applying its function argument directly to a number, it applies the argument to itself and a number. That produces a new F' defined as

$$F'\ \triangleq\ \lambda f.\,\lambda n.\,\mathsf{if}\ (\mathsf{iszero}\ n)\ \mathsf{then}\ \overline{1}\ \mathsf{else}\ (\mathsf{mult}\ n\ ((f\ f)\ (\mathsf{pred}\ n))).$$

We can now correctly define fact by

$$\mathsf{fact}\ \triangleq\ F'\ F'.$$

Notice, that fact, as defined this way, is, in fact, a fixed point of F. That is, $F\ \mathsf{fact} =_\beta \mathsf{fact}$ (under full β-reduction, at least).

2 The Y Combinator

What just happened? We had an operator F defining the factorial function recursively, and we wanted a fixed point of F. We were able to build one simply by replacing all references to f (the first argument) with the self-application $(f\ f)$. That is,

$$F' = \lambda f.\, F\ (f\ f).$$

Then we applied the result to itself to get

$$\mathsf{fact} = F'\ F' = (\lambda f.\, F\ (f\ f))\ (\lambda f.\, F\ (f\ f)).$$

This is a fixed point of F because in one step, we get

$$(\lambda f.\, F\ (f\ f))\ (\lambda f.\, F\ (f\ f)) \ \longrightarrow\ F\ (\lambda f.\, F\ (f\ f))\ (\lambda f.\, F\ (f\ f)).$$

Notably, this construction didn't depend on the particulars of F at all. We could have done it with any function! Indeed, we can define a general version

$$Y \ \triangleq\ \lambda g.\, (\lambda f.\, g\ (f\ f))\ (\lambda f.\, g\ (f\ f))$$

that takes any function g and produces a fixed point such that $Y\ g = g\ (Y\ g)$. For instance, we could define the factorial function simply as $\mathsf{fact} = Y\ F$. This Y is the famous *fixed point combinator*, a closed λ-term that constructs solutions to recursive equations in a uniform way.

3 The Z Combinator

The Y combinator is extremely useful, but it does something curious in a very simple case: the identity function $\mathsf{id} = \lambda x.\, x$. Although *every* λ-term is a fixed point of id, Y produces a particularly unfortunate one:

$$Y\ \mathsf{id}\ \longrightarrow\ (\lambda f.\, \mathsf{id}\ (f\ f))\ (\lambda f.\, \mathsf{id}\ (f\ f))\ =_\beta\ \Omega.$$

This curiosity is actually indicative of a bigger shortcoming of Y. It works great with CBN semantics, but with CBV it immediately goes into an infinite loop. To see why, we can just evaluate it for a few steps with CBV semantics when applied to any function G.

$$\begin{array}{rcl}
Y\ G & = & (\lambda g.\, (\lambda f.\, g\ (f\ f))\ (\lambda f.\, g\ (f\ f)))\ G \\
& \longrightarrow & (\lambda f.\, G\ (f\ f))\ (\lambda f.\, G\ (f\ f)) \\
& \longrightarrow & G\ ((\lambda f.\, G\ (f\ f))\ (\lambda f.\, G\ (f\ f))) \\
& \longrightarrow & G\ (G\ ((\lambda f.\, G\ (f\ f))\ (\lambda f.\, G\ (f\ f)))) \\
& \longrightarrow & \cdots
\end{array}$$

To fix this problem, we would like to delay computation of the argument after the first step. In previous lectures, we delayed computation with thunks, but those require dummy arguments to restart computation. Instead here, we simply want to wrap the argument in a λ-abstraction that behaves the same way, but delayed. That is, we would like the second step to produce

$$G\ (\lambda z.\, (\lambda f.\, G\ \cdots)\ (\lambda f.\, G\ \cdots)\ z)$$

with something appropriate for the "$\cdots$" to cause this to happen every time. This change avoids an infinite loop, as the argument to G is already a value, but when it is applied to anything, it will behave just as it should.

This type of wrapping, where a λ-term e is wrapped in an abstraction $\lambda x.\, e\; x$ that simply applies e to its argument (assume here $x \notin \mathrm{FV}(e)$), is known as *η-expansion*.[1] In general, an η-expanded term produces the same result as the original when applied to a value, but it is already a value itself, meaning it will delay evaluation until the argument is available. This approach simulates what CBN would do.

Modifying Y to accomplish this goal requires η-expanding the self-application $f\; f$, changing it to $\lambda z.\, f\; f\; z$. The result is a CBV-friendly fixed point combinator, often called the Z combinator.

$$Z \triangleq \lambda g.\, (\lambda f.\, g\; (\lambda z.\, f\; f\; z))\; (\lambda f.\, g\; (\lambda z.\, f\; f\; z))$$

4 Turing's Θ Combinator

For any fixed point combinator *Fix*, it is the case that $\mathit{Fix}\; F \equiv F\; (\mathit{Fix}\; F)$ for any function F and a suitable equivalence $\equiv$. That is the point of a fixed point combinator. For the Y and Z combinators above, this notion of equivalence was a behavioral one. That is $Y\; F$ *behaves* as a fixed point of F Syntactically, however, there is some λ-term e such that $Y\; F \longrightarrow^{+} e$ and $e \longrightarrow^{+} F\; e$ (or $Z\; F \longrightarrow^{+} e$ and $e \longrightarrow^{+} F\; (\lambda z.\, e\; z)$, respectively), but e itself does not syntactically contain Y (or Z, respectively).

We might want to change this and make our notion of equivalence that *Fix F* should reduce, syntactically, to $F\; (\mathit{Fix}\; F)$. To accomplish this coal, we can let this equivalence guide our construction of the combinator. Specifically, we could try to build an *Fix* that satisfies the recursive equation

$$\mathit{Fix} = \lambda f.\, f\; (\mathit{Fix}\; f).$$

We can use the same trick we used to build Y on this equation. That is, we make *Fix* an argument, replace each use of it with a self-application, and then apply the resulting λ-term to itself. The result, which satisfies the original equation, is the fixed point combinator Θ:

$$\Theta \triangleq (\lambda x f.\, f\; (x\; x\; f))\; (\lambda x f.\, f\; (x\; x\; f)).$$

This Θ combinator was first discovered by Alan Turing (1912–1954) and published as part of his PhD dissertation, and is often called Turing's combinator, or Turing's Θ. Indeed, Turing's Θ has exactly the property we wanted: $\Theta\; F$ reduces *syntactically* to $F\; (\Theta\; F)$. In fact, in a single step,

$$\begin{aligned}
\Theta &= (\lambda x f.\, f\; (x\; x\; f))\; (\lambda x f.\, f\; (x\; x\; f)) \\
&\longrightarrow \lambda f.\, f\; ((\lambda x f.\, f\; (x\; x\; f))\; (\lambda x f.\, f\; (x\; x\; f))\; f) \\
&= \lambda f.\, f\; (\Theta\; f).
\end{aligned}$$

As a result, for any F, $\Theta\; F \longrightarrow^{2} F\; (\Theta\; F)$.

Note that, when using CBV semantics, Θ has the same nontermination problem as Y. Fixing it requires an η-expansion similar to the one we used to create Z. We leave the precise details as an exercise.

These are the only fixed point combinators we will talk about here, but there are infinitely many more!

[1]There is a corresponding η-contraction by removing such expansions, and η-equivalence when one term can be converted to another through a combination of η-expansions and η-contractions.