1 Fixpoints

With an encoding for if, we have some control over the flow of a program. We can also write simple `for` loops using the Church numerals $\overline{n}$. However, we do not yet have the ability to write an unbounded while loop or a recursive function.

In OCaml, we can write the factorial function recursively as

$$\mathsf{let\ rec\ fact\ n = if\ n \leq 1\ then\ 1\ else\ n * fact\ (n - 1)}$$

But how can we write this in the λ-calculus, in which all functions are anonymous? We must somehow construct a λ-term fact that satisfies the equation

$$\mathsf{fact} \quad = \quad \lambda n.\, \mathsf{if}\,(\mathsf{leq}\, n\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, n\, (\mathsf{fact}\, (\mathsf{sub}\, n\, \overline{1}))). \tag{1}$$

Equivalently, we must construct a *fixpoint* of the map F defined by

$$F \quad \triangleq \quad \lambda f.\, \lambda n.\, \mathsf{if}\,(\mathsf{leq}\, n\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, n\, (f\, (\mathsf{sub}\, n\, \overline{1})));$$

that is, a λ-term fact such that $F\,\mathsf{fact} = \mathsf{fact}$. Any fixpoint of F will do; different fixpoints may disagree on non-integers, but one can show inductively that any fixpoint of F is a solution of (1) and will yield $\overline{n!}$ on input $\overline{n}$.

Note that applying F is like "unwinding" the definition of fact once. If we think of f as an approximation to fact, then $F\,f$ is a better approximation in the sense that if f agrees with fact on inputs $\overline{0}, \overline{1}, \ldots, \overline{n}$, then $F\,f$ agrees with fact on inputs $\overline{0}, \overline{1}, \ldots, \overline{n}, \overline{n+1}$. In fact, we can start with any function f whatsoever and $F^n\,f$ will agree with fact on $\overline{0}, \overline{1}, \ldots, \overline{n}$ for all $n > 0$.

But this is not good enough to construct fact from scratch. All we will ever get this way are better and better finite approximations, but we will never achieve fact itself. So how can we ever hope to construct a fixpoint of F?

2 Recursion via Self-Application

The key observation is that, although we do not have fact itself, we do have something very similar, namely the function F. Thus we might try applying F to itself. The only problem is that F takes an extra argument f, so this would only make sense if in the body of F we applied f to something. Well, we want to apply F to F, so let's apply f to f in the body and see what we get. Call this new version F'.

$$F' \quad \triangleq \quad \lambda f.\, \lambda n.\, \mathsf{if}\,(\mathsf{leq}\, n\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, n\, ((f\, f)\, (\mathsf{sub}\, n\, \overline{1})))$$

Now if we apply F' to itself, we get

$$F'\, F' \quad \xrightarrow{1} \quad \lambda n.\, \mathsf{if}\,(\mathsf{leq}\, n\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, n\, ((F'\, F')\, (\mathsf{sub}\, n\, \overline{1}))).$$

It is a fixpoint of F! Moreover, we can even see that it works as a definition of fact:

$$\begin{array}{rcl}
(F'\, F')\, \overline{4} & \to & (\lambda n.\, \mathsf{if}\,(\mathsf{leq}\, n\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, n\, ((F'\, F')\, (\mathsf{sub}\, n\, \overline{1}))))\, \overline{4} \\
& \to & \mathsf{if}\,(\mathsf{leq}\, \overline{4}\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, \overline{4}\, ((F'\, F')\, (\mathsf{sub}\, \overline{4}\, \overline{1}))) \\
& \to & \mathsf{mul}\, \overline{4}\, ((F'\, F')\, (\mathsf{sub}\, \overline{4}\, \overline{1})) \\
& \to & \mathsf{mul}\, \overline{4}\, ((F'\, F')\, \overline{3}) \\
& \vdots &
\end{array}$$

3 The Y Combinator

What just happened? We wanted a fixpoint of the operator F. We constructed a new term

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

then we applied F' to itself:

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

This is a fixpoint of F, since in one step

$$(\lambda f.\, F\,(f\,f))\,(\lambda f.\, F\,(f\,f)) \quad\to\quad F\,((\lambda f.\, F\,(f\,f))\,(\lambda f.\, F\,(f\,f))).$$

Moreover, this construction does not depend on the nature of F. Thus if we define

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

then for any f, we have that $Y\,f$ is a fixpoint of f; that is, $Y\,f = f\,(Y\,f)$.

This Y is the infamous *fixpoint combinator*, a closed λ-term that constructs solutions to recursive equations in a uniform way.

Curiously, although *every* λ-term is a fixpoint of the identity map $\lambda x.\,x$, the Y combinator produces a particularly unfortunate one, namely the divergent λ-term Ω introduced in Lecture 2.

4 Other Fixpoint Combinators

4.1 A CBV Fixpoint Combinator

The Y combinator shown works perfectly in CBN evaluation, but in CBV evaluation, it produces divergent functions. The problem is in passing in a self-application term $(f\,f)$ that can diverge. If that term diverges, the left-hand side of the equation will diverge when applied to an argument, even though the right-hand side would not. When the Y combinator is used in a CBV setting, it tries to fully unroll the recursive function definition before any use of the function, leading to divergence.

The CBV divergence problem can be fixed by wrapping the self-application term in another lambda abstraction: $\lambda z.\,(x\,x)\,z$. This term yields the same result as $(x\,x)$ when applied to any argument, but is a value, and therefore will only be evaluated when it is applied. The effect of wrapping the term is to delay evaluation as long as possible, simulating what would have happened in CBN evaluation.

The CBV fixpoint combinator is:

$$Y_{\text{CBV}} \triangleq \lambda f.\,(\lambda x.\, f\,(\lambda z.\,(x\,x)\,z))\,(\lambda x.\, f\,(\lambda z.\,(x\,x)\,z))$$

4.2 Turing's Fixpoint Combinator

Since $Y\,f$ is a fixed point of f, we have a solution to $Yf = f\,(Yf)$ that works for any f. Therefore $Y = \lambda f.\, f\,(Yf)$, which is a recursive function definition. Directly applying the same self-application trick of Section 2 to this function definition, we obtain a fixed-point combinator discovered by Alan Turing (1912–1954):

$$\Theta \triangleq (\lambda y.\,\lambda f.\, f\,(y\,y\,f))\,(\lambda y.\,\lambda f.\, f\,(y\,y\,f)).$$

In fact, there are infinitely many fixed-point combinators!