1 Fixed Points

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

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

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 *fixed point* of the map T defined by

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

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

Note that applying T is like "unwinding" the definition of fact once. If we think of f as an approximation to fact, then Tf is a better approximation in the sense that if f agrees with fact on inputs $\overline{0}, \overline{1}, \ldots, \overline{n}$, then Tf agrees with fact on inputs $\overline{0}, \overline{1}, \ldots, \overline{n}, \overline{n+1}$. Thus we can start with any function f whatsoever, and no matter what f is, $T^n f$ will agree with fact on $\overline{0}, \overline{1}, \ldots, \overline{n-1}$.

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 fixed point of T?

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 T. Thus we might try applying T to itself. The only problem is that T takes an extra argument f, so this would only make sense if in the body of T we applied f to something. Well, we want to apply T to T, so let's apply f to f in the body and see what we get. Call this new version T'.

$$T' \quad \triangleq \quad \lambda f.\, \lambda n.\, \mathsf{if}\,(\mathsf{leq}\, n\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, n\, ((ff)\, (\mathsf{sub}\, n\, \overline{1})))$$

Now if we apply T' to itself, we get

$$T'T' \quad \to \quad \lambda n.\, \mathsf{if}\,(\mathsf{leq}\, n\, \overline{1})\, \overline{1}\, (\mathsf{mul}\, n\, ((T'T')\, (\mathsf{sub}\, n\, \overline{1}))).$$

It is a fixed point of T! Moreover, we can even see that it works as a definition of fact:

$$\begin{array}{rcl}
(T'T')\,\overline{4} & \to & (\lambda n.\,\mathsf{if}\,(\mathsf{leq}\,n\,\overline{1})\,\overline{1}\,(\mathsf{mul}\,n\,((T'T')\,(\mathsf{sub}\,n\,\overline{1}))))\,\overline{4} \\
& \to & \mathsf{if}\,(\mathsf{leq}\,\overline{4}\,\overline{1})\,\overline{1}\,(\mathsf{mul}\,\overline{4}\,((T'T')\,(\mathsf{sub}\,\overline{4}\,\overline{1}))) \\
& \to & \mathsf{mul}\,\overline{4}\,((T'T')\,(\mathsf{sub}\,\overline{4}\,\overline{1})) \\
& \to & \mathsf{mul}\,\overline{4}\,((T'T')\,\overline{3}) \\
& \vdots & \\
& \to & \mathsf{mul}\,\overline{4}\,(\mathsf{mul}\,\overline{3}\,(\mathsf{mul}\,\overline{2}\,((T'T')\,\overline{1})) \\
& \to & \mathsf{mul}\,\overline{4}\,(\mathsf{mul}\,\overline{3}\,(\mathsf{mul}\,\overline{2}\,\overline{1}) \\
& \to & \overline{4!}.
\end{array}$$

3 The Y Combinator

What just happened? We had an operator T describing the factorial function recursively, and wanted a fixed point of T. We constructed a new term

$$T' \;\triangleq\; \lambda f.\,T\,(ff),$$

then we applied T' to itself:

$$T'T' \;=\; (\lambda f.\,T\,(ff))\,(\lambda f.\,T\,(ff)).$$

This is a fixed point of T, since in one step

$$(\lambda f.\,T\,(ff))\,(\lambda f.\,T\,(ff)) \;\to\; T\,((\lambda f.\,T\,(ff))\,(\lambda f.\,T\,(ff))). \tag{2}$$

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

$$Y \;\triangleq\; \lambda t.\,(\lambda f.\,t(ff))\,(\lambda f.\,t(ff)),$$

then for any T, we have that YT is a fixed point of T; that is, $YT = T(YT)$.

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

Curiously, although *every* λ-term is a fixed point 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 Fixed-Point Combinators

4.1 A CBV Fixed-Point Combinator

The fixed-point combinator Y works perfectly well with call-by-name (CBN) evaluation, but with call-by-value (CBV), it produces divergent functions. The problem is that the self-application $T'T'$ can diverge. Note that the reduction sequence beginning with (2) would not terminate under CBV. When the Y combinator is used with the CBV reduction strategy, it tries to fully unroll the recursive function definition before applying the function, leading to divergence.

The CBV divergence problem can be fixed by wrapping the self-application ff in another lambda abstraction: $\lambda z.\,ffz$. This term yields the same result as ff when applied to any argument, but it is a value, 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 fixed-point combinator is:

$$Y_{\mathrm{CBV}} \;\triangleq\; \lambda t.\,(\lambda f.\,t(\lambda z.\,ffz))\,(\lambda f.\,t(\lambda z.\,ffz))$$

4.2 Kleene's Fixed-Point Combinator

Since YF is a fixed point of F, we have a solution to the equation $YF = F(YF)$. This construction works for any F. Therefore the equation $Y = \lambda f\,.\,f(Yf)$ constitutes another recursive function definition. Directly applying the same self-application trick of §2 to this function definition, we obtain another fixed-point combinator:

$$\Theta \triangleq (\lambda yf\,.\,f(yyf))\,(\lambda yf\,.\,f(yyf)).$$

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