ldf.1 Fixpoints

lam:ldf:fp:sec

Suppose we wanted to define the factorial function by recursion as a term Fac with the following property:

$$\text{Fac} \equiv \lambda n.\, \text{IsZero}\, n\, \overline{1}(\text{Mult}\, n(\text{Fac}(\text{Pred}\, n)))$$

That is, the factorial of n is 1 if $n = 0$, and n times the factorial of $n - 1$ otherwise. Of course, we cannot define the term Fac this way since Fac itself occurs in the right-hand side. Such recursive definitions involving self-reference are not part of the lambda calculus. Defining a term, e.g., by

$$\text{Mult} \equiv \lambda ab.\, a(\text{Add}\, a)0$$

only involves previously defined terms in the right-hand side, such as Add. We can always remove Add by replacing it with its defining term. This would give the term Mult as a pure lambda term; if Add itself involved defined terms (as, e.g., Add′ does), we could continue this process and finally arrive at a pure lambda term.

However this is not true in the case of recursive definitions like the one of Fac above. If we replace the occurrence of Fac on the right-hand side with the definition of Fac itself, we get:

$$\begin{aligned}\text{Fac} \equiv \lambda n.\, &\text{IsZero}\, n\, \overline{1} \\ &(\text{Mult}\, n((\lambda n.\, \text{IsZero}\, n\, \overline{1}\, (\text{Mult}\, n\, (\text{Fac}(\text{Pred}\, n))))(\text{Pred}\, n)))\end{aligned}$$

and we still haven't gotten rid of Fac on the right-hand side. Clearly, if we repeat this process, the definition keeps growing longer and the process never results in a pure lambda term. Thus this way of defining factorial (or more generally recursive functions) is not feasible.

The recursive definition does tell us something, though: If f were a term representing the factorial function, then the term

$$\text{Fac}' \equiv \lambda g.\, \lambda n.\, \text{IsZero}\, n\, \overline{1}\, (\text{Mult}\, n\, (g(\text{Pred} n)))$$

applied to the term f, i.e., $\text{Fac}'\, f$, also represents the factorial function. That is, if we regard Fac′ as a function accepting a function and returning a function, the value of $\text{Fac}'\, f$ is just f, provided f is the factorial. A function f with the property that $\text{Fac}'\, f \overset{\beta}{=} f$ is called a *fixpoint* of Fac′. So, the factorial is a fixpoint of Fac′.

There are terms in the lambda calculus that compute the fixpoints of a given term, and these terms can then be used to turn a term like Fac′ into the definition of the factorial.

lam:ldf:fp:defn:Turing-Y

Definition ldf.1. The *Y-combinator* is the term:

$$Y \equiv (\lambda ux.\, x(uux))(\lambda ux.\, x(uux))$$

Theorem ldf.2. *Y has the property that $Yg \twoheadrightarrow g(Yg)$ for any term g. Thus, Yg is always a fixpoint of g.*

Proof. Let's abbreviate $(\lambda ux.\, x(uux))$ by U, so that $Y \equiv UU$. Then

$$\begin{aligned} Yg &\equiv (\lambda ux.\, x(uux))U\, g \\ &\twoheadrightarrow (\lambda x.\, x(UUx))g \\ &\twoheadrightarrow g(UUg) \equiv g(Yg) \end{aligned}$$

Since $g(Yg)$ and Yg both reduce to $g(Yg)$, $g(Yg) \overset{\beta}{=} Yg$, so Yg is a fixpoint of g. □

Of course, since Yg is a redex, the reduction can continue indefinitely:

$$\begin{aligned} Yg &\twoheadrightarrow g(Yg) \\ &\twoheadrightarrow g(g(Yg)) \\ &\twoheadrightarrow g(g(g(Yg))) \qquad\qquad \ldots \end{aligned}$$

So we can think of Yg as g applied to itself infinitely many times. If we apply g to it one additional time, we—so to speak—aren't doing anything extra; g applied to g applied infinitely many times to Yg is still g applied to Yg infinitely many times.

Note that the above sequence of β-reduction steps starting with Yg is infinite. So if we apply Yg to some term, i.e., consider $(Yg)N$, that term will also reduce to infinitely many different terms, namely $(g(Yg))N$, $(g(g(Yg)))N$, ... It is nevertheless possible that some *other* sequence of reduction steps does terminate in a normal form.

Take the factorial for instance. Define Fac as $Y\,\mathrm{Fac}'$ (i.e., a fixpoint of Fac'). Then:

$$\begin{aligned} \mathrm{Fac}\,\overline{3} &\twoheadrightarrow Y\,\mathrm{Fac}'\,\overline{3} \\ &\twoheadrightarrow \mathrm{Fac}'(Y\,\mathrm{Fac}')\,\overline{3} \\ &\equiv (\lambda x.\, \lambda n.\, \mathrm{IsZero}\, n\,\overline{1}\,(\mathrm{Mult}\, n\,(x(\mathrm{Pred}\, n))))\,\mathrm{Fac}\,\overline{3} \\ &\twoheadrightarrow \mathrm{IsZero}\,\overline{3}\,\overline{1}\,(\mathrm{Mult}\,\overline{3}\,(\mathrm{Fac}(\mathrm{Pred}\,\overline{3}))) \\ &\twoheadrightarrow \mathrm{Mult}\,\overline{3}\,(\mathrm{Fac}\,\overline{2}) \end{aligned}$$

Similarly,

$$\begin{aligned} \mathrm{Fac}\,\overline{2} &\twoheadrightarrow \mathrm{Mult}\,\overline{2}\,(\mathrm{Fac}\,\overline{1}) \\ \mathrm{Fac}\,\overline{1} &\twoheadrightarrow \mathrm{Mult}\,\overline{1}\,(\mathrm{Fac}\,\overline{0}) \end{aligned}$$

but

$$\begin{aligned}\mathrm{Fac}\,\overline{0} &\twoheadrightarrow \mathrm{Fac}'(Y\,\mathrm{Fac}')\,\overline{0}\\ &\equiv (\lambda x.\,\lambda n.\,\mathrm{IsZero}\,n\,\overline{1}\,(\mathrm{Mult}\,n\,(x(\mathrm{Pred}\,n))))\,\mathrm{Fac}\,\overline{0}\\ &\twoheadrightarrow \mathrm{IsZero}\,\overline{0}\,\overline{1}\,(\mathrm{Mult}\,\overline{0}\,(\mathrm{Fac}(\mathrm{Pred}\,\overline{0})))\\ &\twoheadrightarrow \overline{1}\end{aligned}$$

So together

$$\mathrm{Fac}\,\overline{3} \twoheadrightarrow \mathrm{Mult}\,\overline{3}\,(\mathrm{Mult}\,\overline{2}\,(\mathrm{Mult}\,\overline{1}\,\overline{1}))$$

What goes for Fac′ goes for any recursive definition. Suppose we have a recursive equation

$$g\,x_1\ldots x_n \stackrel{\beta}{=} N$$

where N may contain g and x_1, …, x_n. Then there is always a term $G \equiv (Y\lambda g.\,\lambda x_1\ldots x_n.\,N)$ such that

$$G\,x_1\ldots x_n \stackrel{\beta}{=} N[G/g]$$

for by the fixpoint theorem,

$$\begin{aligned}G \equiv (Y\lambda g.\,\lambda x_1\ldots x_n.\,N) &\twoheadrightarrow \lambda g.\,\lambda x_1\ldots x_n.\,N(Y\lambda g.\,\lambda x_1\ldots x_n.\,N)\\ &\equiv (\lambda g.\,\lambda x_1\ldots x_n.\,N)\,G\end{aligned}$$

and consequently

$$\begin{aligned}G\,x_1\ldots x_n &\twoheadrightarrow (\lambda g.\,\lambda x_1\ldots x_n.\,N)\,G\,x_1\ldots x_n\\ &\twoheadrightarrow (\lambda x_1\ldots x_n.\,N[G/g])\,x_1\ldots x_n\\ &\twoheadrightarrow N[G/g]\end{aligned}$$

The Y combinator of Definition ldf.1 is due to Alan Turing. Alonzo Church had proposed a different version which we'll call Y_C:

$$Y_C \equiv \lambda g.\,(\lambda x.\,g(xx))(\lambda x.\,g(xx))$$

Church's combinator is a bit weaker than Turing's in that $Yg \equiv g(Yg)$ but not $Yg \stackrel{\beta}{\twoheadrightarrow} g(Yg)$. Let V be the term $\lambda x.\,g(xx)$, so that $Y_C \equiv \lambda g.\,VV$. Consider

$$\begin{aligned}VV &\equiv (\lambda x.\,g(xx))V\\ &\twoheadrightarrow g(VV)\end{aligned}$$

and consequently

$$\begin{aligned} Y_C g &\equiv (\lambda g.\, VV)g \twoheadrightarrow VV \twoheadrightarrow g(VV) \\ g(Y_C g) &\equiv g((\lambda g.\, VV)g) \twoheadrightarrow g(VV) \end{aligned}$$

In other words, Yg and $g(Yg)$ reduce to a common term; so $Yg \overset{\beta}{=} g(Yg)$. This is often enough for applications.

Photo Credits

Bibliography