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

\[ \text{Fac} \equiv \lambda n. \text{IsZero } n \ 1 (\text{Mult } n (\text{Fac} (\text{Pred } n))) \]

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

\[ \text{Mult} \equiv \lambda ab. a(\text{Add} b)0 \]

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:

\[ \text{Fac} \equiv \lambda n. \text{IsZero } n \ 1 (\text{Mult } n ((\lambda n. \text{IsZero } n \ 1 (\text{Mult } n (\text{Fac} (\text{Pred } n)))) (\text{Pred } n))) \]

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

\[ \text{Fac}' \equiv \lambda g. \lambda n. \text{IsZero } n \ 1 (\text{Mult } n (g(\text{Pred} n))) \]

applied to the term \( f \), i.e., \( \text{Fac}' \ f \), also represents the factorial function. That is, if we regard \( \text{Fac}' \ f \) 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 \( \text{Fac}' \). So, the factorial is a fixpoint of \( \text{Fac}' \).

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 \( \text{Fac}' \) into the definition of the factorial.

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

\[ Y \equiv (\lambda ux. x(uxx))(\lambda ux. x(uxx)) \].

**fixpoints rev. 2c33e9e (2021-03-08) by OLP / CC–BY**
**Theorem ldf.2.** \( Y \) has the property that \( Yg \rightarrow g(Yg) \) 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

\[
Yg \equiv (\lambda ux.x(uux))Ug \\
\rightarrow (\lambda x.x(UUx))g \\
\rightarrow g(UUg) \equiv g(Yg).
\]

Since \( g(Yg) \) and \( Yg \) both reduce to \( g(Yg) \), \( g(Yg) \beta \equiv Yg \), so \( Yg \) is a fixpoint of \( g \). \( \qed \)

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

\[
Yg \rightarrow g(Yg) \\
\rightarrow g(g(Yg)) \\
\rightarrow g(g(g(Yg)))
\]

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—are doing nothing 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 \( \beta \)-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 \((g(Yg))N, (g(g(Yg)))N, \ldots \). 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 \( Y \text{ Fac'} \) (i.e., a fixpoint of Fac’). Then:

\[
\text{Fac 3} \rightarrow Y \text{ Fac'} 3 \\
\rightarrow \text{Fac'}(Y \text{ Fac'}) 3 \\
\equiv (\lambda x. \lambda n. \text{IsZero} n \text{ If} (\text{Mult} n (x(\text{Pred} n)))) \text{ Fac 3} \\
\rightarrow \text{IsZero 3 If} (\text{Mult} 3 (\text{Fac(Pred 3)})) \\
\rightarrow \text{Mult 3 (Fac 2)}.
\]

Similarly,

\[
\text{Fac 2} \rightarrow \text{Mult 2 (Fac 1)} \\
\text{Fac 1} \rightarrow \text{Mult 1 (Fac 0)}
\]
but

\[
\text{Fac} \overline{0} \rightarrow \text{Fac}'(\text{Y Fac}') \overline{0} \\
\equiv (\lambda x. \lambda n. \text{IsZero } n \overline{I} (\text{Mult } n (x(\text{Pred } n)))) \text{ Fac} \overline{0} \\
\rightarrow \text{IsZero} \overline{0} \overline{I} (\text{Mult } \overline{0} (\text{Fac}(\text{Pred } \overline{0}))). \\
\rightarrow \overline{I}.
\]

So together

\[
\text{Fac} \overline{3} \rightarrow \text{Mult } \overline{3} (\text{Mult } \overline{2} (\text{Mult } \overline{1} \overline{1})).
\]

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

\[
g x_1 \ldots x_n \overset{\beta}{=} N
\]

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

\[
G x_1 \ldots x_n \overset{\beta}{=} N[G/g].
\]

For by the fixpoint theorem,

\[
G \equiv (Y \lambda g. \lambda x_1 \ldots x_n. N) \rightarrow (\lambda g. \lambda x_1 \ldots x_n. N(Y \lambda g. \lambda x_1 \ldots x_n. N)) \\
\equiv (\lambda g. \lambda x_1 \ldots x_n. N) G
\]

and consequently

\[
G x_1 \ldots x_n \rightarrow (\lambda g. \lambda x_1 \ldots x_n. N) G x_1 \ldots x_n \\
\rightarrow (\lambda x_1 \ldots x_n. N[G/g]) x_1 \ldots x_n \\
\rightarrow N[G/g].
\]

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 \(Y g \overset{\beta}{=} g(Y g)\) but not \(Y g \overset{\beta}{
rightarrow} g(Y g)\). Let \(V\) be the term \(\lambda x. g(xx)\), so that \(Y_C \equiv \lambda g. VV\). Then

\[
VV \equiv (\lambda x. g(xx))V \rightarrow g(VV) \text{ and thus} \\
Y_C g \equiv (\lambda g. VV) g \rightarrow VV \rightarrow g(VV), \text{ but also} \\
g(Y_C g) \equiv g((\lambda g. VV) g) \rightarrow g(VV).
\]
In other words, \( Y_C g \) and \( g(Y_C g) \) reduce to a common term \( g(VV) \); so \( Y_C g \overset{\beta}{=} g(Y_C g) \). This is often enough for applications.

Photo Credits

Bibliography