

-NoValue-

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:

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

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, p.ex., 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, p.ex., 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:

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

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

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

applied to the term f , O.s., $\text{Fac}' f$, 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 $\text{Fac}' f \stackrel{\beta}{=} f$ 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 **Definição ldf.1.** The *Y-combinator* is the term:

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

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

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

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

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

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

$$\begin{aligned} Yg &\rightarrowtail g(Yg) \\ &\rightarrowtail g(g(Yg)) \\ &\rightarrowtail g(g(g(Yg))) \\ &\dots \end{aligned}$$

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, O.s., consider $(Yg)N$, that term will also reduce to infinitely many different terms, namely $(g(Yg))N$, $(g(g(Yg)))N$, $\dots$. 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 $Y \text{Fac}'$ (O.s., a fixpoint of Fac'). Then:

$$\begin{aligned} \text{Fac } \bar{3} &\rightarrowtail Y \text{Fac}' \bar{3} \\ &\rightarrowtail \text{Fac}'(Y \text{Fac}') \bar{3} \\ &\equiv (\lambda x. \lambda n. \text{IsZero } n \bar{1} (\text{Mult } n (x(\text{Pred } n)))) \text{Fac } \bar{3} \\ &\rightarrowtail \text{IsZero } \bar{3} \bar{1} (\text{Mult } \bar{3} (\text{Fac}(\text{Pred } \bar{3}))) \\ &\rightarrowtail \text{Mult } \bar{3} (\text{Fac } \bar{2}). \end{aligned}$$

Similarly,

$$\begin{aligned} \text{Fac } \bar{2} &\rightarrowtail \text{Mult } \bar{2} (\text{Fac } \bar{1}) \\ \text{Fac } \bar{1} &\rightarrowtail \text{Mult } \bar{1} (\text{Fac } \bar{0}) \end{aligned}$$

but

$$\begin{aligned}
\text{Fac } \bar{0} &\rightarrow \text{Fac}'(Y \text{Fac}') \bar{0} \\
&\equiv (\lambda x. \lambda n. \text{IsZero } n \bar{1} (\text{Mult } n (x(\text{Pred } n)))) \text{Fac } \bar{0} \\
&\rightarrow \text{IsZero } \bar{0} \bar{1} (\text{Mult } \bar{0} (\text{Fac}(\text{Pred } \bar{0}))). \\
&\rightarrow \bar{1}.
\end{aligned}$$

So together

$$\text{Fac } \bar{3} \rightarrow \text{Mult } \bar{3} (\text{Mult } \bar{2} (\text{Mult } \bar{1} \bar{1})).$$

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

$$g x_1 \dots x_n \stackrel{\beta}{=} N$$

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

$$G x_1 \dots x_n \stackrel{\beta}{=} N[G/g].$$

For by the fixpoint theorem,

$$\begin{aligned}
G &\equiv (Y \lambda g. \lambda x_1 \dots x_n. N) \rightarrow \lambda g. \lambda x_1 \dots x_n. N(Y \lambda g. \lambda x_1 \dots x_n. N) \\
&\equiv (\lambda g. \lambda x_1 \dots x_n. N) G
\end{aligned}$$

and consequently

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

The Y combinator of [Definição 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 $Yg \stackrel{\beta}{=} g(Yg)$ but not $Yg \stackrel{\beta}{\rightarrow} g(Yg)$. Let V be the term $\lambda x. g(xx)$, so that $Y_C \equiv \lambda g. VV$. Then

$$\begin{aligned}
VV &\equiv (\lambda x. g(xx))V \rightarrow g(VV) \text{ and thus} \\
Y_C g &\equiv (\lambda g. VV)g \rightarrow VV \rightarrow g(VV), \text{ but also} \\
g(Y_C g) &\equiv g((\lambda g. VV)g) \rightarrow g(VV).
\end{aligned}$$

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

Créditos das Imagens

Referências Bibliográficas