A λ-abstract \( \lambda x. M \) represents a function of one argument, which is quite a limitation when we want to define function accepting multiple arguments. One way to do this would be by extending the λ-calculus to allow the formation of pairs, triples, etc., in which case, say, a three-place function \( \lambda x. M \) would expect its argument to be a triple. However, it is more convenient to do this by Currying.

Let’s consider an example. We’ll pretend for a moment that we have a + operation in the λ-calculus. The addition function is 2-place, i.e., it takes two arguments. But a λ-abstract only gives us functions of one argument: the syntax does not allow expressions like \( \lambda (x, y). (x + y) \). However, we can consider the one-place function \( f_x(y) \) given by \( \lambda y. (x + y) \), which adds \( x \) to its single argument \( y \). Actually, this is not a single function, but a family of different functions “add \( x \),” one for each number \( x \). Now we can define another one-place function \( g \) as \( \lambda x. f_x \). Applied to argument \( x \), \( g(x) \) returns the function \( f_x \)—so its values are other functions. Now if we apply \( g \) to \( x \), and then the result to \( y \) we get: \((g(x))y = f_x(y) = x + y \). In this way, the one-place function \( g \) can do the same job as the two-place addition function. “Currying” simply refers to this trick for turning two-place functions into one place functions (whose values are one-place functions).

Here is an example properly in the syntax of the λ-calculus. How do we represent the function \( f(x, y) = x \)? If we want to define a function that accepts two arguments and returns the first, we can write \( \lambda x. \lambda y. x \), which literally is a function that accepts an argument \( x \) and returns the function \( \lambda y. x \). The function \( \lambda y. x \) accepts another argument \( y \), but drops it, and always returns \( x \).

Let’s see what happens when we apply \( \lambda x. \lambda y. x \) to two arguments:

\[
(\lambda x. \lambda y. x) MN \rightarrow^\beta (\lambda y. M) N
\]

\[
\rightarrow^\beta M
\]

In general, to write a function with parameters \( x_1, \ldots, x_n \) defined by some term \( N \), we can write \( \lambda x_1. \lambda x_2. \ldots \lambda x_n. N \). If we apply \( n \) arguments to it we get:

\[
(\lambda x_1. \lambda x_2. \ldots \lambda x_n. N) M_1 \ldots M_n \rightarrow^\beta
\]

\[
\rightarrow^\beta ((\lambda x_2. \ldots \lambda x_n. N)[M_1/x_1]) M_2 \ldots M_n
\]

\[
\equiv (\lambda x_2. \ldots \lambda x_n. N[M_1/x_1]) M_2 \ldots M_n
\]

\[
\vdots
\]

\[
\rightarrow^\beta P[M_1/x_1] \ldots [M_n/x_n]
\]

The last line literally means substituting \( M_i \) for \( x_i \) in the body of the function definition, which is exactly what we want when applying multiple arguments to a function.
Photo Credits

Bibliography