Currying

From Wikipedia, the free encyclopedia

In mathematics and computer science, **currying** is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument. It was introduced by Gottlob Frege,[1] developed by Moses Schönfinkel,[2][3][4] and further developed by Haskell Curry.[5][6]

Uncurrying is the dual transformation to currying, and can be seen as a form of defunctionalization. It takes a function $f(x)$ that returns another function $g(y)$ as a result, and yields a new function $f'(x, y)$ that takes a number of additional parameters and applies them to the function returned by function f. The process can be iterated.

Contents

Motivation

There are analytical techniques that can only be applied to functions with a single argument. Practical functions frequently take more arguments than this. Frege showed that it was sufficient to provide solutions for the single argument case, as it was possible to transform a function with multiple arguments into a chain of single-argument functions instead. This transformation is the process now known as currying.[7]

Currying is similar to the process of calculating a function of multiple variables for some given values on paper.

For example, given the function $f(x, y) = y/x$:

To evaluate $f(2, 3)$, first replace x with 2

Since the result is a function of y, this new function $g(y)$ can be defined as $g(y) = f(2, y) = y/2$

Next, replace the y argument with 3, producing $g(3) = f(2, 3) = 3/2$

On paper, using classical notation, this is usually done all in one step. However, each argument can be replaced sequentially as well. Each replacement results in a function taking exactly one argument. This produces a chain of functions as in lambda calculus, and multi-argument functions are usually represented in curried form.

Some programming languages almost always use curried functions to achieve multiple arguments; notable examples are ML and Haskell, where in both cases all functions have exactly one argument.

If we let *f* be a function

$$f(x, y) = \frac{y}{x}$$

then the function *h* where

$$h(x) = y \mapsto f(x, y)$$

is a curried version of f. Here, $y \mapsto z$ is a function that maps an argument *y* to result *z*. In particular,

$$g(y) = h(2) = y \mapsto f(2, y)$$

is the curried equivalent of the example above.

Note that currying, while similar, is not the same operation as partial function application.

Definition

Given a function f of type $f\colon (X \times Y) \to Z$, **currying** it makes a function $\operatorname{curry}(f)\colon X \to (Y \to Z)$. That is, $\operatorname{curry}(f)$ takes an argument of type X and returns a function of type $Y \to Z$. **Uncurrying** is the reverse transformation, and is most easily understood in terms of its right adjoint, apply.

The → operator is often considered right-associative, so the curried function type $X \to (Y \to Z)$ is often written as $X \to Y \to Z$. Conversely, function application is considered to be left-associative, so that $f\ \langle x, y \rangle$ is equivalent to $\operatorname{curry}(f)\ x\ y$.

Curried functions may be used in any language that supports closures; however, uncurried functions are generally preferred for efficiency reasons, since the overhead of partial application and closure creation can then be avoided for most function calls.

Mathematical view

In theoretical computer science, currying provides a way to study functions with multiple arguments in very simple theoretical models such as the lambda calculus in which functions only take a single argument.

In a set-theoretic paradigm, currying is the natural correspondence between the set $A^{B\times C}$ of functions from $B \times C$ to A, and the set $\left(A^C\right)^B$ of functions from B to the set of functions from C to A. In category theory, currying can be found in the universal property of an exponential object, which gives rise to the following adjunction in cartesian closed categories: There is a natural isomorphism between the morphisms from a binary product $f\colon (X \times Y) \to Z$ and the morphisms to an exponential object $g\colon X \to Z^Y$. In other words, currying is the statement that product and Hom are adjoint functors; that is, there is a natural transformation:

$$\hom(A \times B, C) \cong \hom(A, C^B).$$

This is the key property of being a Cartesian closed category, and more generally, a closed monoidal category.[8] The latter, though more rarely discussed, is interesting, as it is the suitable setting for quantum computation,[9] whereas the former is sufficient for classical logic. The difference is that the Cartesian product can be interpreted simply as a pair of items (or a list), whereas the tensor product, used to define a monoidal category, is suitable for describing entangled quantum states.[10]

Under the Curry–Howard correspondence, the existence of currying and uncurrying is equivalent to the logical theorem $(A \wedge B) \to C \Leftrightarrow A \to (B \to C)$, as tuples (product type) corresponds to conjunction in logic, and function type corresponds to implication.

Curry is a continuous function in the Scott topology.[11]

Naming

The name "currying", coined by Christopher Strachey in 1967, is a reference to logician Haskell Curry. The alternative name "Schönfinkelisation" has been proposed as a reference to Moses Schönfinkel.[12] In the mathematical context, the principle can be traced back to work in 1893 by Frege.

Contrast with partial function application

Currying and partial function application are often conflated.[13] One of the significant differences between the two is that a call to a partially applied function returns the result right away, not another function down the currying chain; this distinction can be illustrated clearly for functions whose arity is greater than two.[14]

Given a function of type $f\colon (X \times Y \times Z) \to N$, currying produces $\operatorname{curry}(f)\colon X \to (Y \to (Z \to N))$. That is, while an evaluation of the first function might be represented as $f(1, 2, 3)$, evaluation of the curried function would be represented as $f_{\mathbf{curried}}(1)(2)(3)$,

applying each argument in turn to a single-argument function returned by the previous invocation. Note that after calling $f_{\mathrm{curried}}(1)$, we are left with a function that takes a single argument and returns another function, not a function that takes two arguments.

In contrast, **partial function application** refers to the process of fixing a number of arguments to a function, producing another function of smaller arity. Given the definition of f above, we might fix (or 'bind') the first argument, producing a function of type $\mathrm{partial}(f)\colon (Y \times Z) \to N$. Evaluation of this function might be represented as $f_{\mathrm{partial}}(2, 3)$. Note that the result of partial function application in this case is a function that takes two arguments.

Intuitively, partial function application says "if you fix the first arguments of the function, you get a function of the remaining arguments". For example, if function *div* stands for the division operation *x*/*y*, then *div* with the parameter *x* fixed at 1 (i.e., *div* 1) is another function: the same as the function *inv* that returns the multiplicative inverse of its argument, defined by *inv*(*y*) = 1/*y*.

The practical motivation for partial application is that very often the functions obtained by supplying some but not all of the arguments to a function are useful; for example, many languages have a function or operator similar to `plus_one`. Partial application makes it easy to define these functions, for example by creating a function that represents the addition operator with 1 bound as its first argument.

See also

- Lazy evaluation
- Closure (computer science)
- s_{mn} theorem
- Closed monoidal category

Notes

1. Quine, introduction to *Bausteine der mathematischen Logik", pp. 305–316. Translated by Stefan Bauer-Mengelberg as "On the building blocks of mathematical logic" in Jean van Heijenoort, 1967.* A Source Book in Mathematical Logic, 1879–1931. *Harvard Univ. Press: 355–66.*
2. Strachey, Christopher (2000). "Fundamental Concepts in Programming Languages". *Higher-Order and Symbolic Computation* **13**: 11–49. doi:10.1023/A:1010000313106. "There is a device originated by Schönfinkel, for reducing operators with several operands to the successive application of single operand operators." (Reprinted lecture notes from 1967.)
3. Reynolds, John C. (1998). "Definitional Interpreters for Higher-Order Programming Languages". *Higher-Order and Symbolic Computation* **11** (4): 374. doi:10.1023/A:1010027404223. "In the last line we have used a trick called Currying (after the logician H. Curry) to solve the problem of introducing a binary operation into a language where all functions must accept a single argument. (The referee comments that although "Currying" is tastier, "Schönfinkeling" might be more accurate.)"
4. Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. p. 144.
5. Henk Barendregt, Erik Barendsen, "Introduction to Lambda Calculus (ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf)", March 2000, page 8.
6. Curry, Haskell; Feys, Robert (1958). *Combinatory logic* **I** (2 ed.). Amsterdam, Netherlands: North-Holland Publishing Company.
7. Graham Hutton. "Frequently Asked Questions for comp.lang.functional: Currying". *nott.ac.uk.*
8. Currying (https://ncatlab.org/nlab/show/currying) in *nLab*

9. Samson Abramsky and Bob Coecke, "A Categorical Semantics for Quantum Protocols (http://arxiv.org/abs/quantph/0402130/)."
10. John c. Baez and Mike Stay, "Physics, Topology, Logic and Computation: A Rosetta Stone (http://math.ucr.edu/home/baez/rosetta/rose3.pdf)", (2009) ArXiv 0903.0340 (http://arxiv.org/abs/0903.0340/) in *New Structures for Physics*, ed. Bob Coecke, *Lecture Notes in Physics* vol. **813**, Springer, Berlin, 2011, pp. 95-174.
11. Barendregt, H.P. (1984). *The Lambda Calculus*. North-Holland. ISBN 0-444-87508-5. *(See theorems 1.2.13, 1.2.14)*
12. I. Heim and A. Kratzer (1998). *Semantics in Generative Grammar*. Blackwell.
13. "The Uncarved Blog: Partial Function Application is not Currying". *uncarved.com*.
14. "Functional Programming in 5 Minutes". *Slides*.

References

- Schönfinkel, Moses (1924). "Über die Bausteine der mathematischen Logik". *Math. Ann.* **92** (3–4): 305–316. doi:10.1007/BF01448013.
- Heim, Irene; Kratzer, Angelika (1998). "Semantics in a Generative Grammar". Malden: Blackwall Publishers

External links

- Currying Schonfinkelling (http://c2.com/cgi/wiki?CurryingSchonfinkelling) at the Portland Pattern Repository
- Currying != Generalized Partial Application! (http://lambda-the-ultimate.org/node/2266) - post at Lambda-the-Ultimate.org

Look up ***currying*** in Wiktionary, the free dictionary.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Currying&oldid=700360183"

Categories: Higher-order functions | Functional programming | Lambda calculus

---