

The Lambda Calculus

First published Wed Dec 12, 2012; substantive revision Tue Jul 25, 2023

The λ -calculus is, at heart, a simple notation for functions and application. The main ideas are *applying* a function to an argument and forming functions by *abstraction*. The syntax of basic λ -calculus is quite sparse, making it an elegant, focused notation for representing functions. Functions and arguments are on a par with one another. The result is a non-extensional theory of functions as rules of computation, contrasting with an extensional theory of functions as sets of ordered pairs. Despite its sparse syntax, the expressiveness and flexibility of the λ -calculus make it a cornucopia of logic and mathematics. This entry develops some of the central highlights of the field and prepares the reader for further study of the subject and its applications in philosophy, linguistics, computer science, and logic.

- [1. Introduction](#)
 - [1.1 Multi-argument operations](#)
 - [1.2 Non-Extensionality](#)
- [2. Syntax](#)
 - [2.1 Variables, bound and free](#)
 - [2.2 Combinators](#)
- [3. Brief history of \$\lambda\$ -calculus](#)
- [4. Reduction](#)
 - [4.1 Other notions of reduction](#)
 - [4.2 Reduction strategies](#)
- [5. \$\lambda\$ -theories](#)
 - [5.1 The basic theory \$\lambda\$](#)
 - [5.2 Extending the basic theory \$\lambda\$](#)
- [6. Consistency of the \$\lambda\$ -calculus](#)
- [7. Semantics of \$\lambda\$ -calculus](#)
 - [7.1 \$\lambda\$ -Models](#)
 - [7.2 Model Constructions](#)
- [8. Extensions and Variations](#)
 - [8.1 Combinatory logic](#)
 - [8.2 Adding types](#)
- [9. Applications](#)
 - [9.1 Logie à la \$\lambda\$](#)
 - [9.2 Computing](#)
 - [9.3 Relations](#)
- [Bibliography](#)
- [Academic Tools](#)
- [Other Internet Resources](#)
- [Related Entries](#)

1. Introduction

The λ -calculus is an elegant notation for working with *applications of functions to arguments*. To take a mathematical example, suppose we are given a simple polynomial such as $x^2 - 2 \cdot x + 5$. What is the value of this expression when $x = 2$? We compute this by ‘plugging in’ 2 for x in the expression: we get $2^2 - 2 \cdot 2 + 5$, which we can further reduce to get the answer 5. To use the λ -calculus to represent the situation, we start with the λ -term

$$\lambda x[x^2 - 2 \cdot x + 5].$$

The λ operators allows us to *abstract* over x . One can intuitively read ‘ $\lambda x[x^2 - 2 \cdot x + 5]$ ’ as an expression that is waiting for a value a for the variable x . When given such a value a (such as the number 2), the value of the expression is $a^2 - 2 \cdot a + 5$. The ‘ λ ’ on its own has no significance; it merely *binds* the variable x , guarding it, as it were, from outside interference. The terminology in λ -calculus is that we want to *apply* this expression to an *argument*, and get a value. We write ‘ Ma ’ to denote the application of the function M to the argument a . Continuing with the example, we get:

$$\begin{aligned} (\lambda x[x^2 - 2 \cdot x + 5])2 &\triangleright 2^2 - 2 \cdot 2 + 5 && \langle \text{Substitute 2 for } x \rangle \\ &= 4 - 4 + 5 && \langle \text{Arithmetic} \rangle \\ &= 5 && \langle \text{Arithmetic} \rangle \end{aligned}$$

The first step of this calculation, plugging in ‘2’ for occurrences of x in the expression ‘ $x^2 - 2 \cdot x + 5$ ’, is the passage from an **abstraction term** to another term by the operation of substitution. The remaining equalities are justified by computing with natural numbers.

This example suggests the central principle of the λ -calculus, called **β -reduction**, which is also sometimes called **β -conversion**:

$$(\beta) \quad (\lambda x[M])N \triangleright M[x := N]$$

The understanding is that we can *reduce* or *contract* (\triangleright) an application $(\lambda xM)N$ of an abstraction term (the left-hand side, λxM) to something (the right-hand side, N) by simply plugging in N for the occurrences of x inside M (that’s what the notation ‘ $M[x := N]$ ’ expresses). **β -reduction**, or **β -conversion**, is the heart of the λ -calculus. When one actually applies β -reduction to reduce a term, there is an important proviso that has to be observed. But this will be described in Section 2.1, when we discuss bound and free variables.

1.1 Multi-argument operations

What about functions of multiple arguments? Can the λ -calculus represent operations such as computing the length of the hypotenuse of a right triangle:

$$\text{Hypotenuse of a right triangle with legs of length } x \text{ and } y \Rightarrow \sqrt{x^2 + y^2}.$$

The length-of-hypotenuse operation maps two positive real numbers x and y to another positive real number. One can represent such multiple-arity operations using the apparatus of the λ -calculus by viewing the operation as taking one input at a time. Thus, the operation can be seen as taking one input, x , a positive real number, and producing as its value not a *number*, but an *operation*: namely, the operation that takes a positive real number y as input and produces as output the positive real number $\sqrt{x^2 + y^2}$. One could summarize the discussion by saying that the operation, **HYPOTENUSE-LENGTH**, that computes the length of the hypotenuse of a right triangle given the lengths a and b of its legs, is:

$$\mathbf{HYPOTENUSE-LENGTH} := \lambda a[\lambda b[\sqrt{a^2 + b^2}]]$$

By the principle of β -reduction, we have, for example, that **HYPOTENUSE-LENGTH** 3, the application of **HYPOTENUSE-LENGTH** to 3, is $\lambda b[\sqrt{3^2 + b^2}]$, which is a function of that is ‘waiting’ for another argument. The λ -term **HYPOTENUSE-LENGTH** 3 can be viewed as a function that computes the length of the hypotenuse of a right triangle one of whose legs has length 3. We find, finally, that **(HYPOTENUSE-LENGTH 3)4**—the application of **HYPOTENUSE-LENGTH** to 3 and then to 4—is 5, as expected.

Another way to understand the reduction of many-place functions to one-place functions is to imagine a machine M that initially starts out by loading the first a of multiple arguments a, b, \dots into memory. If one then suspends the machine after it has loaded the first argument into memory, one can view the result as another machine M_a that is awaiting one fewer input; the first argument is now fixed.

1.2 Non-Extensionality

An important philosophical issue concerning the λ -calculus is the question of its underlying concept of functions. In set theory, a function is standardly understood as a set of argument-value pairs. More specifically, a function is understood as a set f of ordered pairs satisfying the property that $(x, y) \in f$ and $(x, z) \in f$ implies $y = z$. If f is a function and $(x, y) \in f$, this means that the function f assigns the value y to the argument x . This is the concept of **functions-as-sets**. Consequently, the notion of equality of functions-as-sets is equality *qua* sets, which, under the standard principle of extensionality, entails that two functions are equal precisely when they contain the same ordered pairs. In other words, two functions are identical if and only if they assign the same values to the same arguments. In this sense, functions-as-sets are *extensional* objects.

In contrast, the notion of a function at work in λ -calculus is one where functions are understood as *rules*: a function is given by a rule for how to determine its values from its arguments. More specifically, we can view a λ -term $\lambda x[M]$ as a description of an operation that, given x , produces M ; the body M of the abstraction term is, essentially, a *rule* for what to do with x . This is the conception of **functions-as-rules**. Intuitively, given rules M and N , we cannot in general decide whether $\lambda x[M]$ is equal to $\lambda x[N]$. The two terms might ‘behave’ the same (have the same value given the same arguments), but it may not be clear what resources are needed for showing the equality of the terms. In this sense, functions-as-rules are *non-extensional* objects.

To distinguish the extensional concept of functions-as-sets from the non-extensional concept of functions-as-rules, the latter is often referred to as an ‘*intensional*’ function concept, in part because of the ostensibly intensional concept of a rule involved. This terminology is particularly predominant in the community of mathematical logicians and philosophers of mathematics working on the foundations of mathematics. But from the perspective of the philosophy of language, the terminology can be somewhat misleading, since in this context, the extensional-intensional distinction has a slightly different meaning.

In the standard possible-worlds framework of philosophical semantics, we would distinguish between an extensional and an intensional function concept as follows. Let us say that two functions are *extensionally equivalent at a world* if and only if they assign the same values to the same arguments at that world. And let us say that two functions are *intensionally equivalent* if and only if they assign the same values to the same arguments at *every* possible-world. To illustrate, consider the functions **HIGHEST-MOUNTAIN-ON-EARTH** and **HIGHEST-MOUNTAIN-IN-THE-HIMALAYAS**, where **HIGHEST-MOUNTAIN-ON-EARTH** assigns the highest mountain on earth as the value to every argument and **HIGHEST-MOUNTAIN-IN-THE-HIMALAYAS** assigns the highest mountain in the Himalayas as the value to every argument. The two functions are extensionally equivalent (at the actual world), but not intensionally so. At the actual world, the two functions assign the same value to every argument, namely Mt. Everest. Now consider a world where Mt. Everest is not the highest mountain on earth, but say, Mt. Rushmore is. Suppose further that this is so, just because Mt. Rushmore is 30.000 feet/9.100 m higher than it is at the actual world, while Mt. Everest, with its roughly 29.000 feet/8.800 m, is still the highest mountain in the Himalayas. At that world, **HIGHEST-MOUNTAIN-ON-EARTH** now assigns Mt. Rushmore as the value to every argument, while **HIGHEST-MOUNTAIN-IN-THE-HIMALAYAS** still assigns Mt. Everest to every object. In other words, **HIGHEST-MOUNTAIN-ON-EARTH** and **HIGHEST-MOUNTAIN-IN-THE-HIMALAYAS** are extensionally equivalent (at the actual world) but not intensionally equivalent.

A function concept may now be called *extensional* if and only if it requires functions that are extensionally equivalent at the actual world to be identical. And a function concept may be classified as *intensional* if and only if it requires intensionally equivalent functions to be identical. Note that these classifications are conceptually different from the distinctions commonly used in the foundations of mathematics. On the terminology used in the foundations of mathematics, functions-as-sets are classified as extensional since they use the axiom of extensionality as their criterion of identity, and functions-as-rules are classified as intensional because they rely on the ostensibly intensional concept of a rule. In the present possible-worlds terminology, function concepts are classified as extensional or intensional based of their behavior at possible-worlds.

An issue from which conceptual confusion might arise is that the two terminologies potentially pass different verdicts on the function concept at work in the λ -calculus. To see this, consider the following two functions:

$$\text{ADD-ONE} := \lambda x[x + 1]$$

$$\text{ADD-TWO-SUBTRACT-ONE} := \lambda x[[x + 2] - 1]$$

These two functions are clearly extensionally equivalent: they assign the same value to the same input at the actual world. Moreover, given standard assumptions in possible worlds semantics, the two functions are also *intensionally* equivalent. If we assume that mathematical facts, like facts about addition and subtraction, are necessary in the sense that they are the same at every possible world, then we get that the two functions give the same value to the arguments at *every* possible world. So, an intensional function concept would require the two functions to be identical. In the λ -calculus, however, it's not clear at all that we should identify the two functions. Formally speaking, without the help of some other principle, we cannot show that the two λ -terms denote the same function. Moreover, informally speaking, on the conception of *functions-as-rules*, it's not even clear that we *should* identify them: the two terms involve genuinely different rules, and so we might be tempted to say that they denote different functions.

A function concept that allows for intensionally equivalent functions to be distinct is called *hyperintensional*. The point is that in possible-worlds terminology, the function concept at work in the λ -calculus may be regarded not as intentional but *hyperintensional*—in contrast to what the terminology common in the foundations of mathematics says. Note that it's unclear how an intensional semantic framework, like the possible-worlds framework, could even in principle account for a non-intensional function concept. On the semantics of the λ -calculus, see section 7. The point here was simply to clarify any conceptual confusions that might arise from different terminologies at play in philosophical discourse.

The hyperintensionality of the λ -calculus is particularly important when it comes to its applications as a theory of not only functions, but more generally *n-ary relations*. On this, see section 9.3. It is effectively the hyperintensionality of the λ -calculus that makes it an attractive tool in this context. It should be noted, however, that the λ -calculus can be made extensional (as well as intensional) by postulating additional laws concerning the equality of λ -terms. On this, see section 5.

2. Syntax

The official syntax of the λ -calculus is quite simple; it is contained in the next definition.

Definition For the alphabet of the language of the λ -calculus we take the left and right parentheses, left and right square brackets, the symbol ' λ ', and an infinite set of variables. The class of λ -terms is defined inductively as follows:

1. Every variable is a λ -term.
2. If M and N are λ -terms, then so is (MN) .
3. If M is a λ -term and x is a variable, then $(\lambda x[M])$ is a λ -term.

By 'term' we always mean ' λ -term'. Terms formed according to rule (2) are called *application terms*. Terms formed according to rule (3) are called *abstraction terms*.

As is common when dealing with formal languages that have grouping symbols (the left and right parenthesis, in our case), some parentheses will be omitted when it is safe to do so (that is, when they can be reintroduced in only one sensible way). Juxtaposing more than two λ -terms is, strictly speaking, illegal. To avoid the tedium of always writing all needed parentheses, we adopt the following convention:

Convention (association to the left): When more than two terms $M_1M_2M_3 \dots M_n$ are juxtaposed we can recover the missing parentheses by *associating to the left*: reading from left to right, group M_1 and M_2 together, yielding $(M_1M_2)M_3 \dots M_n$; then group (M_1M_2) with M_3 : $((M_1M_2)M_3) \dots M_n$, and so forth.

The convention thus gives a unique reading to any sequence of λ -terms whose length is greater than 2.

2.1 Variables, bound and free

The function of λ in an abstraction term $(\lambda x[M])$ is that it *binds* the variable appearing immediately after it in the term M . Thus λ is analogous to the universal and existential quantifiers \forall and \exists of first-order logic. One can define, analogously, the notions of free and bound variable in the expected way, as follows.

Definition The syntactic functions **FV** and **BV** (for ‘free variable’ and ‘bound variable’, respectively) are defined on the set of λ -terms by structural induction thus:

For every variable x , term M , and term N :

Free	Bound
(1) $\mathbf{FV}(x) = \{x\}$	$\mathbf{BV}(x) = \emptyset$
(2) $\mathbf{FV}(MN) = \mathbf{FV}(M) \cup \mathbf{FV}(N)$	$\mathbf{BV}(MN) = \mathbf{BV}(M) \cup \mathbf{BV}(N)$
(3) $\mathbf{FV}(\lambda x[M]) = \mathbf{FV}(M) - \{x\}$	$\mathbf{BV}(\lambda x[M]) = \mathbf{BV}(M) \cup \{x\}$

If $\mathbf{FV}(M) = \emptyset$ then M is called a *combinator*.

Clause (3) in the two definitions supports the intention that λ binds variables (ensures that they are not free). Note the difference between **BV** and **FV** for variables.

As is typical in other subjects where the concepts appear, such as first-order logic, one needs to be careful about the issue; a casual attitude about substitution can lead to syntactic difficulties.^[1] We can defend a casual attitude by adopting the convention that we are interested not in terms themselves, but in a certain equivalence class of terms. We now define substitution, and then lay down a convention that allows us to avoid such difficulties.

Definition (substitution) We write ‘ $M[x := N]$ ’ to denote the substitution of N for the free occurrences of x in M . A precise definition^[2] by recursion on the set of λ -terms is as follows: for all terms A , B , and M , and for all variables x and y , we define

1. $x[x := M] \equiv M$
2. $y[x := M] \equiv y$ (y distinct from x)
3. $(AB)[x := M] \equiv A[x := M]B[x := M]$
4. $(\lambda x[A])[x := M] \equiv \lambda x[A]$
5. $(\lambda y[A])[x := M] \equiv \lambda y[A[x := M]]$ (y distinct from x)

Clause (1) of the definition simply says that if we are to substitute M for x and we are dealing simply with x , then the result is just M . Clause (2) says that nothing happens when we are dealing (only) with a variable different from x but we are to substitute something for x . Clause (3) tells us that substitution unconditionally distributes over applications. Clauses (4) and (5) concern abstraction terms and parallel clauses (1) and (2) (or rather, clauses (2) and (1), in opposite order): If the bound variable z of the abstraction term $\lambda z[A]$ is identical to the variable x for which we are to do a substitution, then we do not perform any substitution (that is, substitution “stops”). This coheres with the intention that $M[x := N]$ is supposed to denote the substitution of N for the *free* occurrences of x in M . If M is an abstraction term $\lambda x[A]$ whose bound variable is x , then x does not occur freely in M , so there is nothing to do. This explains clause 4. Clause (5), finally, says that if the bound variable of an abstraction term differs from x , then at least x has the “chance” to occur freely in the abstraction term, and substitution continues into the body of the abstraction term.

Definition (change of bound variables, α -convertibility). The term N is obtained from the term M by a *change of bound variables* if, roughly, any abstraction term $\lambda x[A]$ inside M has been replaced by $\lambda y[A[x := y]]$.

Let us say that terms M and N are α -convertible if there is a sequence of changes of bound variables starting from M and ending at N .

Axiom. β -conversion (stated with a no-capture proviso):

$$(\lambda x[M])N \triangleright M[x := N],$$

provided no variable that occurs free in N becomes bound after its substitution into M .

Roughly, we need to adhere to the principle that free variables ought to remain free; when an occurrence of a variable is threatened to become bound by a substitution, simply perform enough α -conversions to sidestep

the problem. If we keep this in mind, we can work with λ -calculus without worrying about these nettlesome syntactic difficulties. So, for example, we can't apply the function $\lambda x[\lambda y[x(y - 5)]]$ to the argument $2y$ because upon substitution of " $2y$ " for " x ", the " y " in " $2y$ " would be captured by the variable-binding operator " λy ". Such a substitution would yield a function different from the one intended. However, we can first transform $\lambda x[\lambda y[x(y - 5)]]$ to $\lambda x[\lambda z[x(z - 5)]]$ by α -conversion, and then apply this latter function to the argument $2y$. So whereas the following is *not* a valid use of β -conversion:

$$(\lambda x[\lambda y[x(y - 5)]]2y) \triangleright \lambda y[2y(y - 5)]$$

we can validly use β -conversion to conclude:

$$(\lambda x[\lambda z[x(z - 5)]]2y) \triangleright \lambda z[2y(z - 5)]$$

This example helps one to see why the proviso to β -conversion is so important. The proviso is really no different from the one used in the statement of an axiom of the predicate calculus, namely: $\forall x\phi \rightarrow \phi_x^\tau$, provided no variable that is free in the term τ before the substitution becomes bound after the substitution.

The syntax of λ -calculus is quite flexible. One can form all sorts of terms, even self-applications such as xx . Such terms appear at first blush to be suspicious; one might suspect that using such terms could lead to inconsistency, and in any case one might find oneself reaching for a tool with which to forbid such terms. If one were to view functions and sets of ordered pairs of a certain kind, then the x in xx would be a function (set of ordered pairs) that contains as an element a pair (x, y) whose first element would be x itself. But no set can contain itself in this way, lest the axiom of foundation (or regularity) be violated. Thus, from a set theoretical perspective such terms are clearly dubious. Below one can find a brief sketch of one such tool, type theory. But in fact such terms do not lead to inconsistency and serve a useful purpose in the context of λ -calculus. Moreover, forbidding such terms, as in type theory, does not come for free (e.g., some of the expressiveness of untyped λ -calculus is lost).

2.2 Combinators

As defined earlier, a *combinator* is a λ -term with no free variables. One can intuitively understand combinators as 'completely specified' operations, since they have no free variables. There are a handful of combinators that have proven useful in the history of λ -calculus; the next table highlights some of these special combinators. Many more could be given (and obviously there are infinitely many combinators), but the following have concise definitions and have proved their utility. Below is a table of some standard λ -terms and combinators.

Name Definition & Comments

S	$\lambda x[\lambda y[\lambda z[xz(yz)]]]$ Keep in mind that ' $xz(yz)$ ' is to be understood as the application $(xz)(yz)$ of xz to yz . S can thus be understood as a substitute-and-apply operator: z 'intervenes' between x and y : instead of applying x to y , we apply xz to yz .
K	$\lambda x[\lambda y[x]]$ The value of K M is the constant function whose value for any argument is simply M .
I	$\lambda x[x]$ The identity function.
B	$\lambda x[\lambda y[\lambda z[x(yz)]]]$ Recall that ' xyz ' is to be understood as $(xy)z$, so this combinator is not a trivial identity function.
C	$\lambda x[\lambda y[\lambda z[xzy]]]$ Swaps an argument.
T	$\lambda x[\lambda y[x]]$ Truth value true. Identical to K . We shall see later how these representations of truth values plays a role in the blending of logic and λ -calculus.
F	$\lambda x[\lambda y[y]]$ Truth value false.

- ω $\lambda x[xx]$
Self-application combinator
- Ω $\omega\omega$
Self-application of the self-application combinator. Reduces to itself.
- Y $\lambda f[(\lambda x[f(xx)])(\lambda x[f(xx)])]$
Curry's paradoxical combinator. For every λ -term X , we have:

$$\begin{aligned} YX &\triangleright (\lambda x[X(xx)])(\lambda x[X(xx)]) \\ &\triangleright X((\lambda x[X(xx)])(\lambda x[X(xx)])) \end{aligned}$$

The first step in the reduction shows that YX reduces to the application term $(\lambda x[X(xx)])(\lambda x[X(xx)])$, which is recurring in the third step. Thus, Y has the curious property that YX and $X(YX)$ reduce to a common term.

- Θ $(\lambda x[\lambda f[f(xx f)]])(\lambda x[\lambda f[f(xx f)]])$
Turing's fixed-point combinator. For every λ -term X , ΘX reduces to $X(\Theta X)$, which one can confirm by hand. (Curry's paradoxical combinator Y does not have this property.)

Below is a table of notational conventions employed in this entry.

Notation	Reading & Comments
----------	--------------------

MN	The application of the function M to the argument N .
------	---

Usually, parentheses are used to separate the function from the argument, like so: ' $M(N)$ '. However, in λ -calculus and kindred subjects the parentheses are used as grouping symbols. Thus, it is safe to write the function and the argument adjacent to one other.

PQR	The application of the function PQ —which is itself the application of the function P to the argument Q —to R .
-------	---

If we do not use parentheses to separate function and argument, how are we to disambiguate expressions that involve three or more terms, such as ' PQR '? Recall our convention that we are to understand such officially illegal expressions by working from left to right, always putting parentheses around adjacent terms. Thus, ' PQR ' is to be understood as $(PQ)R$. ' $PQRS$ ' is $((PQ)R)S$. The expression ' $(PQ)R$ ' is disambiguated; by our convention, it is identical to PQR . The expression ' $P(QR)$ ' is also explicitly disambiguated; it is distinct from PQR because it is the application of P to the argument QR (which is itself the application of the function Q to the argument R).

$(\lambda x[M])$	The λ term that <i>binds</i> the variable x in the <i>body</i> term M .
------------------	---

The official vocabulary of the λ -calculus consists of the symbol ' λ ', left '(' and right ')' parentheses, and a set of variables (assumed to be distinct from the three symbols ' λ ', '(', and ')') lest we have syntactic chaos).

Alternative notation. It is not necessary to include two kinds of grouping symbols (parentheses and square brackets) in the syntax. Parentheses or square brackets alone would obviously suffice. The two kinds of brackets are employed in this entry for the sake of readability. Given the two kinds of grouping symbols, we could economize further and omit the parentheses from abstraction terms, so that ' $(\lambda x[M])$ ' would be written as ' $\lambda x[M]$ '.

Some authors write ' $\lambda x. M$ ' or ' $\lambda x \cdot M$ ', with a full stop or a centered dot separating the bound variable from the body of the abstraction term. As with the square brackets, these devices are intended to assist reading λ -terms; they are usually not part of the official syntax. (One sees this device used in earlier works of logic, such as *Principia Mathematica*, where

the function of the symbol \cdot in expressions such as $\forall x.\phi$ is to get us to read the whole of the formula ϕ as under the scope of the $\forall x$.)

Some authors write abstraction terms without any device separating the bound variable from the body: such terms are crisply written as, e.g., λxx , λyx . The practice is not without its merits: it is about as concise as one can ask for, and permits an even simpler official syntax of the λ -calculus. But this practice is not flawless. In λxyz , is the bound variable x or is it xy ? Usually the names of variables are single letters, and theoretically this is clearly sufficient. But it seems unduly restrictive to forbid the practice of giving longer names to variables; indeed, such constructions arise naturally in computer programming languages.

For the sake of uniformity, we will adopt the square bracket notation in this entry. (Incidentally, this notation is used in (Turing, 1937).)

$M[x := A]$ The λ -term that is obtained by substituting the λ -term A for all free occurrences of x inside M .

A bewildering array of notations to represent substitution can be found in the literature on λ -calculus and kindred subjects:

$$M[x/A], M[A/x], M_x^A, M_A^x, [x/A]M, \dots$$

Which notation to use for substitution seems to be a personal matter. In this entry we use a linear notation, eschewing superscripts and subscripts. The practice of representing substitution with $:=$ comes from computer science, where $:=$ is read in some programming languages as assigning a value to a variable.

As with the square brackets employed to write abstraction terms, the square brackets employed to write substitution are not officially part of the syntax of the λ -calculus. M and A are terms, x is a variable; $M[x := A]$ is another term.

$M \equiv N$ The λ -terms M and N are identical: understood as sequences of symbols, M and N have the same length and corresponding symbols of the sequences are identical.

The syntactic identity relation \equiv is not part of the official syntax of λ -calculus; this relation between λ -terms belongs to the metatheory of λ -calculus. It is clearly a rather strict notion of equality between λ -terms. Thus, it is not the case (if x and y are distinct variables) that $\lambda x[x] \equiv \lambda y[y]$, even though these two terms clearly ‘behave’ in the same way in the sense that both are expressions of the identity operation $x \Rightarrow x$. Later we will develop formal theories of equality of λ -terms with the aim of capturing this intuitive equality of $\lambda x[x]$ and $\lambda y[y]$.

3. Brief history of λ -calculus

λ -calculus arose from the study of functions as rules. Already the essential ingredients of the subject can be found in Frege’s pioneering work (Frege, 1893). Frege observed, as we did above, that in the study of functions it is sufficient to focus on unary functions (i.e., functions that take exactly one argument). (The procedure of viewing a multiple-arity operation as a sequence of abstractions that yield an equivalent unary operation is called *currying* the operation. Perhaps it would be more historically accurate to call the operation *fregeing*, but there are often miscarriages of justice in the appellation of mathematical ideas.) In the 1920s, the mathematician Moses Schönfinkel took the subject further with his study of so-called *combinators*. As was common in the early days of the subject, Schönfinkel was interested in the kinds of transformations that one sees in formal logic, and his combinators were intended to be a contribution to the foundations of formal logic. By analogy with the reduction that one sees in classical propositional logic with the Sheffer stroke, Schönfinkel established the astonishing result that the all functions (in the sense of all transformations) could be given in terms of the combinators **K** and **S**; later we will see the definition of these combinators.

Theorem For every term M made up of **K** and **S** and the variable x , there exists a term F (built only from **K** and **S**) such that we can derive $Fx = M$.

(The proof that these two suffice to represent all functions is beyond the scope of this entry. For further discussion, see the entry on [combinatory logic](#).) One can prove the theorem constructively: there is an algorithm that, given M , produces the required F . Church called this F ‘ $\lambda x[M]$ ’ (Church, 1932).^[3] From this perspective, the β -rule can be justified: if ‘ $\lambda x[M]$ ’ is to be a function F satisfying $Fx = M$, then $\lambda x[M]$ x should transform to M . This is just a special case of the more general principle that for all N , $(\lambda x[M])N$ should transform to $M[x := N]$.

Although today we have more clearly delimited systems of abstraction and rewriting, in its early days λ -calculus and combinatory logic (à la Schönfinkel) were bound up with investigations of foundations of mathematics. In the hands of Curry, Church, Kleene, and Rosser (some of the pioneers in the subject) the focus was on defining mathematical objects and carrying out logical reasoning inside the these new systems. It turned out that these early attempts at so-called illative λ -calculus and combinatory logic were inconsistent. Curry isolated and polished the inconsistency; the result is now known as Curry’s paradox. See the entry on [Curry’s paradox](#) and appendix B of (Barendregt, 1985).

The λ -calculus earns a special place in the history of logic because it was the source of the first undecidable problem. The problem is: given λ -terms M and N , determine whether $M = N$. (A theory of equational reasoning about λ -terms has not yet been defined; the definition will come later.) This problem was shown to be undecidable.

Another early problem in the λ -calculus was whether it is consistent at all. In this context, inconsistency means that all terms are equal: one can reduce any λ -term M to any other λ -term N . That this is not the case is an early result of λ -calculus. Initially one had results showing that certain terms were not interconvertible (e.g., **K** and **S**); later, a much more powerful result, the so-called Church-Rosser theorem, helped shed more light on β -conversion and could be used to give quick proofs of the non-inter-convertibility of whole classes of λ -terms. See below for more detailed discussion of consistency.

The λ -calculus was a somewhat obscure formalism until the 1960s, when, at last, a ‘mathematical’ semantics was found. Its relation to programming languages was also clarified. Till then the only models of λ -calculus were ‘syntactic’, that is, were generated in the style of Henkin and consisted of equivalence classes of λ -terms (for suitable notions of equivalence). Applications in the semantics of natural language, thanks to developments by Montague and other linguists, helped to ‘spread the word’ about the subject. Since then the λ -calculus enjoys a respectable place in mathematical logic, computer science, linguistics (see, e.g., Heim and Kratzer 1998), and kindred fields.

4. Reduction

Various notions of reduction for λ -terms are available, but the principal one is β -reduction, which we have already seen earlier. Earlier we used the notation ‘ \triangleright ’; we can be more precise. In this section we discuss β -reduction and some extensions.

Definition (one-step β -reduction $\triangleright_{\beta,1}$) For λ -terms A and B , we say that A β -reduces in one step to B , written $A \triangleright_{\beta,1} B$, just in case there exists an (occurrence of a) subterm C of A , a variable x , and λ -terms M and N such that $C \equiv (\lambda x[M])N$ and B is A except that the occurrence of C in A is replaced by $M[x := N]$.

Here are some examples of β -reduction:

1. The variable x does not β -reduce to anything. (It does not have the right shape: it is simply a variable, not an application term whose left-hand side is an abstraction term.)
2. $(\lambda x[x])a \triangleright_{\beta,1} a$.

3. If x and y are distinct variables, then $(\lambda x[y])a \triangleright_{\beta,1} y$.

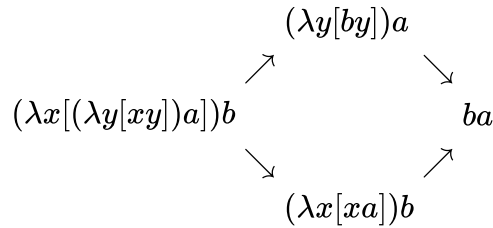
4. The λ -term $(\lambda x[(\lambda y[xy])a])b$ β -reduces in one step to two different λ -terms:

$$(\lambda x[(\lambda y[xy])a])b \triangleright_{\beta,1} (\lambda y[by])a$$

and

$$(\lambda x[(\lambda y[xy])a])b \triangleright_{\beta,1} (\lambda x[xa])b$$

Moreover, one can check that these two terms β -reduce in one step to a common term: ba . We thus have:



As with any binary relation, one can ask many questions about the relation $\triangleright_{\beta,1}$ holding between λ -terms, and one can define various derived notions in terms of $\triangleright_{\beta,1}$.

Definition A β -reduction sequence from a λ -term A to a λ -term B is a finite sequence s_1, \dots, s_n of λ -terms starting with A , ending with B , and whose adjacent terms (s_k, s_{k+1}) satisfy the property that $s_k \triangleright_{\beta,1} s_{k+1}$.

More generally, any sequence s —finite or infinite—starting with a λ -term A is said to be a β -reduction sequence commencing with A provided that the adjacent terms (s_k, s_{k+1}) of s satisfy the property that $s_k \triangleright_{\beta,1} s_{k+1}$.

1. Continuing with β -reduction Example 1, there are no β -reduction sequences at all commencing with the variable x .
2. Continuing with β -reduction Example 2, the two-term sequence

$$(\lambda x[x])a, a$$

is a β -reduction sequence from $(\lambda x[x])a$ to a . If a is a variable, then this β -reduction sequence cannot be prolonged, and there are no other β -reduction sequences commencing with $(\lambda x[x])a$; thus, the set of β -reduction sequences commencing with $(\lambda x[x])a$ is finite and contains no infinite sequences.

3. The combinator $\mathbf{\Omega}$ has the curious property that $\mathbf{\Omega} \triangleright_{\beta,1} \mathbf{\Omega}$. Every term of every β -reduction sequence commencing with $\mathbf{\Omega}$ (finite or infinite) is equal to $\mathbf{\Omega}$.
4. Consider the term $\mathbf{Ka}\mathbf{\Omega}$. There are infinitely many reduction sequences commencing with this term:
 - $\mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} a$
 - $\mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} \mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} a$
 - $\mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} \mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} \mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} a$
 - $\mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} \mathbf{Ka}\mathbf{\Omega} \triangleright_{\beta,1} \mathbf{Ka}\mathbf{\Omega} \dots$

If a is a variable, one can see that all finite reduction sequences commencing with $\mathbf{Ka}\mathbf{\Omega}$ end at a , and there is exactly one infinite reduction sequence.

Definition A β -*redex* of a λ -term M is (an occurrence of) a subterm of M of the form $(\lambda x[P])Q$. ('redex' comes from 'reducible expression.') A β -redex is simply a candidate for an application of β -reduction. Doing so, one *contracts* the β -redex. A term is said to be in β -*normal form* if it has no β -redexes.

(Can a term have multiple β -normal forms? The answer is literally 'yes', but substantially the answer is 'no': If a M and M' are β -normal forms of some term, then M is α -convertible to M' . Thus, β -normal forms are unique up to changes of bound variables.)

So far we have focused only on one step of β -reduction. One can combine multiple β -reduction steps into one by taking the transitive closure of the relation $\triangleright_{\beta,1}$.

Definition For λ -terms A and B , one says that A β -*reduces* to B , written $A \triangleright_{\beta} B$, if either $A \equiv B$ or there exists a finite β -reduction sequence from A to B .

Definition A term M *has a β -normal form* if there exists a term N such that N is in β -normal form and $M \triangleright_{\beta} N$.

Reducibility as defined is a one-way relation: it is generally not true that if $A \triangleright_{\beta} B$, then $B \triangleright_{\beta} A$. However, depending on one's purposes, one may wish to treat A and B as equivalent if either A reduces to B or B reduces to A . Doing so amounts to considering the reflexive, symmetric, and transitive closure of the relation $\triangleright_{\beta,1}$.

Definition For λ -terms A and B , we say that $A =_{\beta} B$ if either $A \equiv B$ or there exists a sequence s_1, \dots, s_n starting with A , ending with B , and whose adjacent terms (s_k, s_{k+1}) are such that either $s_k \triangleright_{\beta,1} s_{k+1}$ or $s_{k+1} \triangleright_{\beta,1} s_k$.

4.1 Other notions of reduction

We have thus far developed the theory of β -reduction. This is by no means the only notion of reduction available in the λ -calculus. In addition to β -reduction, a standard relation between λ -terms is that of η -*reduction*:

Definition (one-step η -reduction) For λ -terms A and B , we say that A $\beta\eta$ -reduces in one step to B , written $A \triangleright_{\beta\eta,1} B$, just in case there exists an (occurrence of a) subterm C of A , a variable x , and λ -terms M and N such that either

$$C \equiv (\lambda x[M])N \text{ and } B \text{ is } A \text{ except that the occurrence of } C \text{ in } A \text{ is replaced by } M[x := N]$$

or

$$C \equiv (\lambda x[Mx]) \text{ and } B \text{ is } A \text{ except that the occurrence of } C \text{ in } A \text{ is replaced by } M.$$

The first clause in the definition of $\triangleright_{\beta\eta,1}$ ensures that the relation extends the relation of one-step β -reduction. As we did for the relation of one-step β -reduction, we can replay the development for η -reduction. Thus, one has the notion of an η -redex, and from $\triangleright_{\eta,1}$ one can define the relation \triangleright_{η} between λ -terms as the reflexive and transitive closure of $\triangleright_{\eta,1}$, which captures zero-or-more-steps of η -reduction. Then one defines $=_{\eta}$ as the symmetric and transitive closure of \triangleright_{η} .

If $A \triangleright_{\eta,1} B$, then the length of B is strictly smaller than that of A . Thus, there can be no infinite η -reductions. This is not the case of β -reduction, as we saw above in [beta-reduction sequence examples 3](#) and [4](#).

One can combine notions of reduction. One useful combination is to blend β - and η -reduction.

Definition (one-step $\beta\eta$ -reduction) $\lambda x[Mx] \triangleright_{\beta\eta,1} M$ and $(\lambda x[M]N) \triangleright_{\beta\eta,1} M[x := N]$. A λ -term A $\beta\eta$ -reduces in one step to a λ -term B just in case either A β -reduces to B in one step or A η -reduces to B in one step.

Again, one can replay the basic concepts of reduction, as we did for β -reduction, for this new notion of reduction $\beta\eta$.

4.2 Reduction strategies

Recall that a term is said to be in β -normal form if it has no β -redexes, that is, subterms of the shape $(\lambda x[M])N$. A term has a β -normal form if it can be reduced to a term in β -normal form. It should be intuitively clear that if a term has a β -normal form, then we can find one by exhaustively contracting all β -redexes of the term, then exhaustively contracting all β -redexes of all resulting terms, and so forth. To say that a term has a β -normal form amounts to saying that this blind search for one will eventually terminate.

Blind search for β -normal forms is not satisfactory. In addition to be aesthetically unpleasant, it can be quite inefficient: there may not be any need to exhaustively contract all β -redexes. What is wanted is a *strategy*—preferably, a computable one—for finding a β -normal form. The problem is to effectively decide, if there are multiple β -redexes of a term, which ought to be reduced.

Definition A β -reduction strategy is a function whose domain is the set of all λ -terms and whose value on a term M not in β -normal form is a redex subterm of M , and whose value on all terms M in β -normal form is simply M .

In other words, a β -reduction strategy selects, whenever a term has multiple β -redexes, which one should be contracted. (If a term is in β -normal form, then nothing is to be done, which is why we require in the definition of β -reduction strategy that it does not change any term in β -normal form.) One can represent a strategy S as a relation \triangleright_S on λ -terms, with the understanding that $M \triangleright_S N$ provided that N is obtained from M in one step by adhering to the strategy S . When viewed as relations, strategies constitute a subrelation of $\triangleright_{\beta,1}$.

A β -reduction strategy may or may not have the property that adhering to the strategy will ensure that we (eventually) reach a β -normal form, if one exists.

Definition A β -reduction strategy S is *normalizing* if for all λ -terms M , if M has a β -normal form N , then the sequence $M, S(M), S(S(M)), \dots$ terminates at N .

Some β -reduction strategies are normalizing, but others are not.

- The *rightmost strategy*, whereby we always choose to reduce the rightmost β -redex (if there are any β -redexes) is not normalizing. Consider, for example, the term $\mathbf{KI}\Omega$. This term has two β -redexes: itself, and Ω (which, recall, is the term $\omega\omega \equiv (\lambda x[xx])(\lambda x[xx])$). By working with left-hand β -redexes, we can β -reduce $\mathbf{KI}\Omega$ to \mathbf{I} in two steps. If we insist on working with the rightmost β -redex Ω we reduce $\mathbf{KI}(\Omega)$ to $\mathbf{KI}(\Omega)$, then $\mathbf{KI}(\Omega)$,
- The *leftmost strategy*, whereby we always choose to reduce the leftmost β -redex (if there are any β -redexes) is normalizing. The proof of this fact is beyond the scope of this entry; see (Barendregt, 1985, section 13.2) for details.

Once we have defined a reduction strategy, it is natural to ask whether one can improve it. If a term has a β -normal form, then a strategy will discover a normal form; but might there be a shorter β -reduction sequence that reaches the same normal form (or a term that is α -convertible to that normal form)? This is the question of *optimality*. Defining optimal strategies and showing that they are optimal is generally considerably more difficult than simply defining a strategy. For more discussion, see (Barendregt, 1984 chapter 10).

For the sake of concreteness, we have discussed only β -reduction strategies. But in the definitions above the notion of reduction β is but one possibility. For any notion R of reduction we have the associated theory of R -reduction strategies, and one can replay the problems of normalizability, optimality, etc., for R .

5. λ -theories

We discussed earlier how the λ -calculus is a non-extensional theory of functions. If, in the non-extensional spirit, we understand λ -terms as descriptions, how should we treat equality of λ -terms? Various approaches are available. In this section, let us treat the equality relation $=$ as a primitive, undefined relation holding between two λ -terms, and try to *axiomatize* the properties that equality should have. The task is to identify axioms and formulate suitable **rules of inference** concerning the equality of λ -terms.

Some obvious properties of equality, having nothing to do with λ -calculus, are as follows:

$$\text{(Reflexivity)} \quad \frac{}{X = X}$$

$$\text{(Symmetry)} \quad \frac{X = Y}{Y = X}$$

$$\text{(Transitivity)} \quad \frac{X = Y \quad Y = Z}{X = Z}$$

As is standard in proof theory, the way to read these rules of inference is that above the horizontal rule $\frac{}{}$ are the **premises** of the rule (which are equations) and the equation below the horizontal rule is the **conclusion** of the rule of inference. In the case of the reflexivity rule, nothing is written above the horizontal rule. We understand such a case as saying that, for all terms X , we may infer the equation $X = X$ from no premises.

5.1 The basic theory λ

The three rules of inference listed in the previous section governing equality have nothing to do with the λ -calculus. The following lists rules of inference that relate the undefined notion of equality and the two term-building operations of the λ -calculus, application and abstraction.

$$\frac{M = N}{AM = AN} \quad \frac{M = N}{MA = NA}$$

$$\text{(\xi)} \quad \frac{M = N}{\lambda x[M] = \lambda x[N]}$$

Together, these rules of inference say that $=$ is a **congruence relation** on the set of λ -terms: it ‘preserves’ both the application and abstraction term-building operations

The final rule of inference, β -conversion, is the most important:

$$\text{(\beta)} \quad \frac{}{(\lambda x[M])A = M[x := A]}$$

As before with the reflexivity rule, the rule β has no premises: for any variable x and any terms M and A , one can infer the equation $(\lambda x[M])A = M[x := A]$ at any point in a formal derivation in the theory λ .

5.2 Extending the basic theory λ

A number of extensions to λ are available. Consider, for example, the rule (η) , which expresses the principle of η -reduction as a rule of inference:

$$\text{(\eta)} \quad \frac{}{\lambda x[Mx] = M} \text{ provided } x \notin \mathbf{FV}(M)$$

Rule η tells us that a certain kind of abstraction is otiose: it is safe to identify M with the function that, given an argument x , applies M to x . Through this rule we can also see that all terms are effectively functions. One can intuitively justify this rule using the principle of β -reduction.

$$\text{(Ext)} \quad \frac{Mx = Nx}{M = N} \text{ provided } x \notin \mathbf{FV}(M) \cup \mathbf{FV}(N)$$

One can view rule **Ext** as a kind of generalization principle. If we have derived that $Mx = Nx$, but x figures in neither M nor N , then we have effectively shown that M and N are alike. Compare this principle to the principle of universal generalization in first-order logic: if we have derived $\phi(x)$ from a set Γ of hypotheses in which x is not free, then we can conclude that Γ derives $\forall x\phi$.

Another productive principle in the λ -calculus permits us to identify terms that ‘act’ the same:

$$\text{(\omega)} \quad \frac{\text{For all terms } x, Mx = Nx}{M = N}$$

The rule ω has infinitely many hypotheses: on the assumption that $Mx = Nx$, no matter what x may be, then we can conclude that $M = N$. The ω rule is an analogue in the λ -calculus of the rule of inference under the same name in formal number theory, according to which one can conclude the universal formula $\forall x\phi$ provided one has proofs for $\phi(x := \mathbf{0})$, $\phi(x := \mathbf{1})$, \dots . Note that unlike the rule **Ext**, the condition that x not occur freely in M or N does not arise.

6. Consistency of the λ -calculus

Is the λ -calculus consistent? The question might not be well-posed. The λ -calculus is not a logic for reasoning about propositions; there is no apparent notion of contradiction (\perp) or a method of forming absurd propositions (e.g., $p \wedge \neg p$). Thus ‘inconsistency’ of the λ -calculus cannot mean that \perp , or some formula tantamount to \perp , is derivable. A suitable notion of ‘consistent’ is, however, available. Intuitively, a logic is inconsistent if it permits us to derive too much. The theory λ is a theory of equations. We can thus take inconsistency of λ to mean: *all equations are derivable*. Such a property, if it were true of λ , would clearly show that λ is of little use as a formal theory.

Early formulations of the idea of λ -calculus by A. Church were indeed inconsistent; see (Barendregt, 1985, appendix 2) or (Rosser, 1985) for a discussion. To take a concrete problem: how do we know that the equation $\mathbf{K} = \mathbf{I}$ is not a theorem of λ ? The two terms are obviously intuitively distinct. \mathbf{K} is a function of two arguments, whereas \mathbf{I} is a function of one argument. If we could show that $\mathbf{K} = \mathbf{I}$, then we could show that $\mathbf{K}\mathbf{K} = \mathbf{I}\mathbf{K}$, whence $\mathbf{K}\mathbf{K} = \mathbf{K}$ would be a theorem of λ , along with many other equations that strike us as intuitively unacceptable. But when we’re investigating a formal theory such as λ , intuitive unacceptability by no means implies underderivability. What is missing is a deeper understanding of β -reduction.

An early result that gave such an understanding is known as the Church-Rosser theorem:

Theorem (Church-Rosser) If $P \triangleright_{\beta} Q$ and $P \triangleright_{\beta} R$, then there exists a term S such that both $Q \triangleright_{\beta} S$ and $R \triangleright_{\beta} S$.

(The proof of this theorem is quite non-trivial and is well-beyond the scope of this entry.) The result is a deep fact about β -reduction. It says that no matter how we diverge from P by β -reductions, we can always converge again to a common term.

The Church-Rosser theorem gives us, among other things, that the plain λ -calculus—that is, the theory λ of equations between λ -terms—is consistent, in the sense that not all equations are derivable.

As an illustration, we can use the Church-Rosser theorem to solve the earlier problem of showing that the two terms \mathbf{K} and \mathbf{I} are not identified by λ . The two terms are in β -normal form, so from them there are no β -reduction sequences at all. If $\mathbf{K} = \mathbf{I}$ were a theorem of λ , then there would be a term M from which there is a β -reduction path to both \mathbf{I} and \mathbf{K} . The Church-Rosser theorem then implies the two paths diverging from M can be merged. But this is impossible, since \mathbf{K} and \mathbf{I} are distinct β -normal forms.

The Church-Rosser theorem implies the existence of β -reduction sequences commencing from \mathbf{K} and from \mathbf{I} that end at a common term. But there are no β -reduction sequences at all commencing from \mathbf{I} , because it is in

β -normal form, and likewise for \mathbf{K} .

Theorem λ is consistent, in the sense that not every equation is a theorem.

To prove the theorem, it is sufficient to produce one underivable equation. We have already worked through an example: we used the Church-Rosser theorem to show that the equation $\mathbf{K} = \mathbf{I}$ is not a theorem of λ . Of course, there's nothing special about these two terms. A significant generalization of this result is available: *if M and N in β -normal form but M is distinct from N , then the equation $M = N$ is not a theorem of λ .* (This simple condition for underivability does not generally hold if we add additional rules of inference to λ .)

The theories $\lambda\eta$ and $\lambda\omega$ are likewise consistent. One can prove these consistency results along the lines of the consistency proof for λ by extending the Church-Rosser theorem to the wider senses of derivability of these theories.

7. Semantics of λ -calculus

As we've said at the outset, the λ -calculus is, at heart, about functions and their applications. But it is surprisingly difficult to cash this idea out in semantic terms. A natural approach would be to try to associate with every λ -term M a function f_M over some domain D and to interpret application terms (MN) using function application as $f_M(f_N)$. But this idea quickly runs into difficulties. To begin with, it's easy to see that, in this context, we can't use the standard set-theoretic concept of functions-as-sets (see section 1.2 of this entry). According to this concept, remember, a function f is a set of argument-value pairs, where every argument gets assigned a unique value. The problem arises in the context of *self-applications*. Remember from section 2.1 that the untyped λ -calculus allows λ -terms such as (xx) , which intuitively apply x to itself. On the semantic picture we're exploring, we can obtain the associated function $f_{(xx)}$ for the term (xx) by taking the function f_x for x and applying it to itself:

$$f_{(xx)} = f_x(f_x)$$

But following functions-as-sets, this would mean that the set f_x needs to contain an argument-value pair that has f_x as its first component and $f_{(xx)}$ as the second:

$$f_x = \{\dots, (f_x, f_{(xx)}), \dots\}$$

But this would make f_x a non-well-founded object: defining f_x would involve f_x itself. In fact, sets like this are excluded in standard axiomatic set theory by the axiom of foundation (also known as the axiom of regularity). —This is further *semantic* evidence that the concept of a function underlying the λ -calculus can't be the extensional functions-as-sets concept.

But the problem runs even deeper than that. Even when we use a non-extensional notion of a function, such as the functions-as-rules conception (see again section 1.2), we run into difficulties. In the untyped λ -calculus, everything can both be function and an argument to functions. Correspondingly, we should want our domain D to include, in some sense, the function space D^D , which contains all and only the functions with both arguments and values from D . To see this:

- Every element of D can be a function that applies to elements of D , and what's returned can then be again be an argument for elements of D . So, every element of D intuitively corresponds to a member of D^D .
- If, in turn, we take a member of D^D , i.e., a function with arguments and values from D , this is precisely the kind of thing we want to include in our domain D . So, intuitively, we want every member of D^D to correspond to a member of D .

In short, we want there to be a *one-to-one correspondence* between our domain and its own function space, i.e., we want them to satisfy the 'equation' $X \cong X^X$. But this is impossible since it contradicts Cantor's theorem.

Given these difficulties, the question arises whether it's possible to give a set-theoretic model for the λ -calculus in the first place? It turns out that it is. D. Scott was the first to describe such a model in an unpublished manuscript from 1969. This model, D_∞ , solves the aforementioned problems with Cantor's theorem by suitably restricting the function space D^D , by only letting *some* members of D^D correspond to members of D . Covering Scott's construction goes beyond the scope of this entry, since it involves advanced tools from algebra and topology; see (Meyer 1982), (Barendregt, 1985, chapter 18.2), or (Hindley and Seldin, 2008, chapter 16) for details. Instead, we'll discuss the more general question: *What is a model for the λ -calculus?* That is, leaving aside for a moment the question whether sets are functions, rules, or something altogether different, we ask what kind of mathematical structure a model for the λ -calculus is in the first place.

7.1 λ -Models

It turns out that there are multiple, essentially equivalent, ways of defining the notion of a model for the λ -calculus; see (Barendregt, 1985, chapter 5) or (Hindley and Seldin, 2008, chapter 15). In the following, we'll discuss what we consider the most palatable notion for philosophers familiar with the standard semantics for first-order logic (see, e.g., the entry on [Classical Logic](#)), the so-called *syntactical λ -models*. These models first appear in the work of (Hindley and Longo, 1980), (Koymans, 1982), and (Meyer 1982). They derive their name from the fact that their clauses closely correspond to the syntactic rules of the calculus λ . This is somewhat unsatisfactory and motivates 'syntax-free' definitions (see below). At the same time, the syntactical λ -models provide a fairly transparent and accessible route into the world of λ -models. In addition, despite their conceptual shortcomings, syntactical models have proven a technically useful tool in the semantical study of the λ -calculus.

In order to avoid the set-theoretic problems mentioned above, most definitions of λ -models use so-called *applicative structures*. The idea is to treat the denotations of λ -terms not as set-theoretic functions, but as unanalyzed, first-order 'function-objects', instead. Correspondingly, then, we treat function application as an unanalyzed binary operation on these function-objects:

Definition An *applicative structure* is a pair (D, \cdot) , where D is some set and \cdot a binary operation on D . To avoid trivial models, we usually assume that D has at least two elements.

Applicative structures are, in a sense, first-order models of function spaces that satisfy the problematic equation $X \cong X^X$. λ -models, in turn, are defined over them.

For the definition of our λ -models, we work with *valuations*—a concept familiar from first-order semantics. Valuations assign denotations to the variables and are used primarily in the semantic clauses for the λ -operator. Additionally, they can be used to express general claims over the domain, in a way that is familiar from the semantics for the first-order quantifiers $\exists x$ and $\forall x$.

Definition A *valuation* in an applicative structure (D, \cdot) is a function ρ that assigns an element $\rho(x) \in D$ to every variable x .

As a useful piece of notation, for ρ a valuation in some applicative structure (D, \cdot) , x a variable, and $d \in D$ an object, we define the valuation $\rho[x \mapsto d]$ by saying that:

$$\rho[x \mapsto d](y) = \begin{cases} d & \text{if } y = x \\ \rho(y) & \text{otherwise} \end{cases}$$

That is, $\rho[x \mapsto d]$ is the result of changing the value of x to be d , while leaving all other other values under ρ unchanged.

Definition A *syntactical λ -model* is a triple $\mathfrak{M} = (D, \cdot, \llbracket _ \rrbracket)$, where (D, \cdot) is an applicative structure and $\llbracket _ \rrbracket$ is a function that assigns to every λ -term M and valuation ρ a denotation $\llbracket M \rrbracket_\rho \in D$ subject to the following constraints:

1. $\llbracket x \rrbracket_\rho = \rho(x)$

2. $\llbracket MN \rrbracket_\rho = \llbracket M \rrbracket_\rho \cdot \llbracket N \rrbracket_\rho$
3. $\llbracket \lambda x M \rrbracket_\rho \cdot d = \llbracket M \rrbracket_{\rho[x \mapsto d]}$, for all $d \in D$
4. $\llbracket \lambda x M \rrbracket_\rho = \llbracket \lambda x N \rrbracket_\rho$, whenever for all $d \in D$, we have $\llbracket M \rrbracket_{\rho[x \mapsto d]} = \llbracket N \rrbracket_{\rho[x \mapsto d]}$
5. $\llbracket M \rrbracket_\rho = \llbracket M \rrbracket_\sigma$, whenever $\rho(x) = \sigma(x)$ for all $x \in \mathbf{FV}(M)$

Intuitively, in a model \mathfrak{M} , $\llbracket M \rrbracket_\rho$ is the function-object denoted by the λ -term M under the valuation ρ .

It is now straight-forward to define what it means for a λ -model \mathfrak{M} to satisfy an equation $M = N$, symbolically $\mathfrak{M} \models M = N$:

Definition (satisfaction).

$$\mathfrak{M} \models M = N \text{ iff for all } \rho, \text{ we have } \llbracket M \rrbracket_\rho = \llbracket N \rrbracket_\rho$$

In words: an equation $M = N$ holds in a model \mathfrak{M} just in case the λ -terms M and N have the same denotation under every valuation in the underlying applicative structure.

Note that clauses 3. and 4. from the definition of a syntactical λ -model directly mirror the λ -rules β and ξ , respectively (see section 5.1 above). This is the ‘syntactic’ nature of our models. While this might be semantically unsatisfactory (see below), it makes it relatively straight-forward to prove a soundness theorem for the semantics provided by the syntactical λ -models; see (Barendregt, 1985, Theorem 5.3.4) and (Hindley and Seldin, 2008, Theorem 15.12):

Theorem For all terms M, N , if $M = N$ is derivable in λ , then for all syntactical λ -models \mathfrak{M} , we have that $\mathfrak{M} \models M = N$.

This theorem provides a first ‘sanity-check’ for the semantics. But note that, so far, we haven’t shown that there *exist* any syntactical λ -models at all.

This worry is addressed by constructing so-called ‘term models’, which are not unlike the well-known Henkin constructions from first-order semantics. In order to define these models, we first need the notion of a λ -equivalence class for a given λ -term M . This class contains precisely the terms that λ proves identical to M :

$$[M]_\lambda = \{N : \lambda \text{ proves } M = N\}$$

We then define the *term model* for λ , \mathfrak{T} , by setting:

- $D = \{[M]_\lambda : M \text{ is a } \lambda\text{-term}\}$
- $[M]_\lambda \cdot [N]_\lambda = [MN]_\lambda$
- $\llbracket M \rrbracket_\rho = [M[x_1 := N_1] \dots [x_n := N_n]]_\lambda$, where $\mathbf{FV}(M) = \{x_1, \dots, x_n\}$ and $\rho(x_1) = N_1, \dots, \rho(x_n) = N_n$

It is easily seen that this indeed defines a syntactical λ -model. In fact, it is easily checked that in the term model for λ , we have that:

$$\mathfrak{T} \models M = N \text{ iff } \lambda \text{ derives } M = N.$$

This paves a way for a very simple completeness proof for λ with respect to the class of syntactical λ -models; see (Meyer, 1982, 98–99) for one of the few explicit mentions of this kind of result in the literature:

Theorem For all terms M, N , if for all syntactical λ -models \mathfrak{M} , we have that $\mathfrak{M} \models M = N$, then $M = N$ is derivable in λ .

The proof is a simple proof by contraposition, which uses the term model \mathfrak{T} as a countermodel to any non-derivable identity in λ .

But there are reasons to be dissatisfied with the syntactical λ -models as a semantics for the λ -calculus. For one, by virtue of clauses 3. and 4. mirroring rules β and ξ , the soundness result is ‘baked into’ the semantics, as it were. This is unsatisfactory from a semantic perspective since it means that via the syntactical λ -models, we don’t really learn anything directly about what conditions an applicative structure needs to satisfy in order to adequately model the λ -calculus.

A related worry is that the clauses 3. and 4. are not *recursive* in nature. That is, they don’t allow us to compute the denotation of a complex λ -term from the denotations of its parts and information about the syntactic operation used to combine them. In our syntax (see section 2), there are two ways of constructing complex λ -terms: application terms of the form MN and abstraction terms of the form $(\lambda x[M])$. Clause 1. of our syntactical λ -models is a recursive clause for the syntactical application operation, but we don’t have a recursive clause for the syntactical operation of λ -abstraction. Clauses 3. and 4. are rather conditions on the denotation function $\llbracket _ \rrbracket$ than recursive clauses. This is unsatisfactory since it means that we’re not really given a compositional semantics for the λ -operator by the syntactical λ -models.

These worries are taken care of in the development of *syntax-free λ -models*. A comprehensive discussion of syntax-free models goes beyond the scope of this entry; but see (Barendregt, 1985, chapter 5.2) and (Hinley and Seldin, 2008, chapter 15B) for the details. Suffice it to say that the definition of syntax-free λ -models involves determining precisely under which conditions an applicative structure is suitable for interpreting the λ -calculus. The resulting λ -models, then, indeed provide (something much closer to) a recursive, compositional semantics, where the syntactical operation of λ -abstraction is interpreted by a corresponding semantic operation on applicative structures.

It is worth noting, however, that syntactical λ -models and the syntax-free λ -models are, in a certain sense, equivalent: every syntactical λ -model defines a syntax-free λ -model and vice versa; see (Barendregt, 1985, theorem 5.3.6) and (Hinley and Seldin, 2008, theorem 15.20) for the details. From a technical perspective, this result allows us to freely move between the different presentations of λ -models and to use, in a given context, the notion of a model that is most expedient. At the same time, there may be *philosophical* reasons to prefer one presentation over the other, such as the semantic worries about syntactical λ -models mentioned above.

Before moving to model constructions, let us briefly mention that there are various ways of approaching λ -models. One particularly interesting approach we’ve neglected so far is from the perspective of category theory and categorical logic. There are well-known model descriptions using so-called ‘Cartesian closed categories’; see (Koymans, 1982). Covering these model descriptions goes beyond the scope of the present entry since it requires a familiarity with a wide range of concepts from category theory; see the entry [Category Theory](#) for a sense of the machinery involved. For the details of these model descriptions, instead, (Barendregt, 1985, sections 5.4–6). In recent years, there has been a renewed interest in categorical approaches to the λ -calculus, which have mainly focused on typed versions of the λ -calculus (see sections [8.2](#) and [9.1.2](#) below) but also include the untyped λ -calculus discussed in this article. See, for example, (Hyland, 2017) for a recent discussion.

7.2 Model Constructions

The term model we’ve seen in section [7.1](#) is rather trivial: it directly reflects the syntactic structure of the λ -terms by modeling precisely syntactic equality modulo λ -provable equality. This makes the term model mathematically and philosophically rather uninteresting. The construction and study of more interesting concrete λ -models is one of the principal aims of the model theory for the λ -calculus.

We’ve already mentioned what’s perhaps the most important, but was definitely the first non-trivial model for the λ -calculus: Scott’s D_∞ . But there are also other interesting model constructions, such as Plotkin and Scott’s graph model P_ω , first described in (Plotkin 1972) and (Scott, 1974). These model constructions, however, usually rely on fairly involved mathematical methods, both for their definitions and for verifying that they are indeed λ -models. Consequently, covering these constructions goes beyond the scope of this entry; see (Hinley and Seldin, 2008, chapter 16F) for an overview of various model constructions and (Barendregt, 1985, chapter 18) for many of the formal details.

One of the advantages of having different models is that one sees different aspects of equality in the λ -calculus: each of the different models takes a different view on what λ -terms get identified. An interesting question in this context is: *What is the λ -theory of a given class of models?* In this context, we call a class \mathcal{C} of λ -models *complete* just in case every (consistent) λ -theory is satisfied by some model in \mathcal{C} . See (Salibra, 2003) for an overview of various completeness and incompleteness results for interesting classes of λ -models.

8. Extensions and Variations

8.1 Combinatory logic

A sister formalism of the λ -calculus, developed slightly earlier, deals with variable-free combinations. *Combinatory logic* is indeed even simpler than the λ -calculus, since it lacks a notion of variable binding.

The language of combinatory logic is built up from *combinators* and variables. There is some flexibility in precisely which combinators are chosen as basic, but some standard ones are **I**, **K**, **S**, **B** and **C**. (The names are not arbitrary.)

As with the λ -calculus, with combinatory logic one is interested in *reducibility* and *provability*. The principal reduction relations are:

Combinator	Reduction Axiom
I	$\mathbf{I}x = x$
K	$\mathbf{K}xy = x$
S	$\mathbf{S}xyz = xz(yz)$
B	$\mathbf{B}xyz = x(yz)$
C	$\mathbf{C}xyz = xzy$

There is a passage from λ -calculus to combinatory logic via translation. It turns out that although combinatory logic lacks a notion of abstraction, one can define such a notion and thereby simulate the λ -calculus in combinatory logic. Here is one translation; it is defined recursively.

Rule	Expression	Translation	Condition
1	x	x	(unconditional)
2	MN	$\mathbf{M}^*\mathbf{N}^*$	(unconditional)
3	$\lambda x[M]$	$\mathbf{K}M$	x does not occur freely in M
4	$\lambda x[x]$	I	(unconditional)
5	$\lambda x[Mx]$	M	x does not occur freely in M
6	$\lambda x[MN]$	$\mathbf{B}M(\lambda x[N])^*$	x does not occur freely in M
7	$\lambda x[MN]$	$\mathbf{C}(\lambda x[M])^*\mathbf{N}$	x does not occur freely in N
8	$\lambda x[MN]$	$\mathbf{S}M^*\mathbf{N}^*$	x occurs freely in both M and N

This translation works inside-out, rather than outside-in. To illustrate:

1. The translation of the term $\lambda y[y]$, a representative of the identity function, is mapped by this translation to the identity combinator **I** (because of Rule 4), as expected.
2. The λ -term $\lambda x[\lambda y[x]]$ that we have been calling ‘**K**’ is mapped by this translation to:

$$\begin{aligned} \lambda x[\lambda y[x]] &\equiv \lambda x[\mathbf{K}x] && \langle \text{Rule 1} \rangle \\ &\equiv \mathbf{K} && \langle \text{Rule 3} \rangle \end{aligned}$$

3. The λ -term $\lambda x[\lambda y[yx]]$ that switches its two arguments is mapped by this translation to:

$$\begin{aligned}
\lambda x[\lambda y[yx]] &\equiv \lambda x[\mathbf{C}(\lambda y[y])^*x] && \langle \text{Rule 8} \rangle \\
&\equiv \lambda x[\mathbf{CI}x] && \langle \lambda y[y] \equiv \mathbf{I}, \text{ by Rule 4} \rangle \\
&\equiv \mathbf{BCI}(\lambda x[x])^* && \langle \text{Rule 7} \rangle \\
&\equiv \mathbf{B(CI)I} && \langle (\lambda x[x])^* \equiv \mathbf{I}, \text{ by Rule 4} \rangle
\end{aligned}$$

We can confirm that the λ -term $\lambda x[\lambda y[yx]]$ and the translated combinatory logic term $\mathbf{B(CI)I}$ have analogous applicative behavior: for all λ -terms P and Q we have

$$(\lambda x[\lambda y[yx]])PQ \triangleright (\lambda y[yP]) \triangleright QP;$$

likewise, for all combinatory logic terms P and Q we have

$$\mathbf{B(CI)IP}Q \triangleright (\mathbf{CI})(\mathbf{IP})Q \triangleright \mathbf{IQ}(\mathbf{IP}) \triangleright Q(\mathbf{IP}) \triangleright QP$$

We can give but a glimpse of combinatory logic; for more on the subject, consult the entry on [combinatory logic](#). Many of the issues discussed here for λ -calculus have analogues in combinatory logic, and vice versa.

8.2 Adding types

In many contexts of reasoning and computing it is natural to distinguish between different kinds of objects. The way this distinction is introduced is by requiring that certain formulas, functions, or relations accept arguments or permit substitution only of some kinds of objects rather than others. We might require, for example, that addition $+$ take numbers as arguments. The effect of this restriction is to forbid, say, the addition of 5 and the identity function $\lambda x. x$.^[4] Regimenting objects into types is also the idea behind the passage from (unsorted, or one-sorted) first-order logic to *many-sorted* first-order logic. (See (Enderton, 2001) and (Manzano, 2005) for more about many-sorted first-order logic.) As it stands, the λ -calculus does not support this kind of discrimination; any term can be applied to any other term.

It is straightforward to extend the untyped λ -calculus so that it discriminates between different kinds of objects. This entry limits itself to the *type-free* λ -calculus. See the entries on [type theory](#) and [Church's type theory](#) for a detailed discussion of the extensions of λ -calculus that we get when we add types, and see (Barendregt, Dekkers, Statman, 2013) for a book length treatment of the subject.

From a model-theoretic perspective, it's interesting to add that (Scott, 1980) uses the semantic fact that categorical models for the *untyped* λ -calculus (see section 7.1) derive from the categorical models of the typed λ -calculus to argue for a conceptual priority of the typed over the untyped calculus.

9. Applications

9.1 Logic à la λ

Here are two senses in which λ -calculus is connected with logic.

9.1.1 Terms as logical constants

In the [table of combinators](#) above, we defined combinators \mathbf{T} and \mathbf{F} and said that they serve as representations in the λ -calculus of the truth values true and false, respectively. How do these terms function as truth values?

It turns out that when one is treating λ -calculus as a kind of programming language, one can write conditional statements “If P then A else B ” simply as PAB , where of course P , A , and B are understood as λ -terms. If $P \triangleright \mathbf{T}$, that is, P is ‘true’, then we have

$$\text{if-}P\text{-then-}A\text{-else-}B := PAB \triangleright \mathbf{T}AB \triangleright A,$$

(recall that, by definition, $\mathbf{T} \equiv \mathbf{K}$) and if $P \triangleright \mathbf{F}$, that is, P is ‘false’, then

$$\text{if-}P\text{-then-}A\text{-else-}B := PAB \triangleright \mathbf{F}AB \triangleright B,$$

(recall that, by definition, $\mathbf{F} \equiv \mathbf{KI}$) which is just what we expect from a notion of if-then-else. If P reduces neither to \mathbf{T} nor \mathbf{F} , then we cannot in general say what if- P -then- A -else- B is.

The encoding we've just sketched of some of the familiar truth values and logical connectives of classical truth-table logic does not show that λ -calculus and classical logic are intimately related. The encoding shows little more than embeddability of the rules of computation of classical truth-table logic in λ -calculus. Logics other than classical truth-table logic can likewise be represented in the λ -calculus, if one has sufficient computable ingredients for the logic in question (e.g., if the logical consequence relation is computable, or if a derivability relation is computable, etc.). For more on computing with λ -calculus, see section 9.2 below. A more intrinsic relationship between logic and λ -calculus is discussed in the next section.

9.1.2 Typed λ -calculus and the Curry-Howard-de Bruijn correspondence

The correspondence to be described here between logic and the λ -calculus is seen with the help of an apparatus known as *types*. This section sketches the beginnings of the development of the subject known as *type theory*. We are interested in developing type theory only so far as to make the so-called Curry-Howard-de Bruijn correspondence visible. A more detailed treatment can be found in the entry on [type theory](#); see also (Hindley, 1997) and (Barendregt, Dekkers, Statman, 2013).

Type theory enriches the untyped λ -calculus by requiring that terms be given *types*. In the untyped λ -calculus, the application MN is a legal term regardless of what M and N are. Such freedom permits one to form such suspicious terms as xx , and thence terms such as the paradoxical combinator \mathbf{Y} . One might wish to exclude terms like xx on the grounds that x is serving both as a function (on the left-hand side of the application) and as an argument (on the right-hand side of the application). Type theory gives us the resources for making this intuitive argument more precise.

Assigning types to terms The language of type theory begins with an (infinite) set of *type variables* (which is assumed to be disjoint from the set of variables of the λ -calculus and from the symbol ' λ ' itself). The set of types is made up of type variables and the operation $\sigma \rightarrow \tau$. Variables in type theory now come with a *type annotation* (unlike the unadorned term variables of untyped λ -calculus). Typed variables are rendered ' $x : \sigma$ '; the intuitive reading is 'the variable x has the type σ '. The intuitive reading of the judgment ' $t : \sigma \rightarrow \tau$ ' is that the term t is a function that transforms arguments of type σ into arguments of type τ . Given an assignment of types to term variables, one has the typing rules:

$$(M : \sigma \rightarrow \tau)(N : \sigma) : \tau$$

and

$$(\lambda x : \sigma[M : \tau]) : \sigma \rightarrow \tau$$

The above two rules define the assignment of types to applications and to abstraction terms. The set of terms of type theory is the set of terms built up according to these formation rules.

The above definition of the set of terms of type theory is sufficient to rule out terms such as xx . Of course, ' xx ' is not a typed term at all for the simple reason that no types have been assigned to it. What is meant is that there is no type σ that could be assigned to x such that ' xx ' could be annotated in a legal way to make a typed term. We cannot assign to x a type variable, because then the type of the left-hand x would fail to be a function type (i.e., a type of the shape ' $\sigma \rightarrow \tau$ '). Moreover, we cannot assign to x a function type $\sigma \rightarrow \tau$, because then then σ would be equal to $\sigma \rightarrow \tau$, which is impossible.

As a leading example, consider the types that are assigned to the combinators \mathbf{I} , \mathbf{K} , and \mathbf{S} :

Combinator	Type ^[5]
\mathbf{I}	$a \rightarrow a$
\mathbf{K}	$a \rightarrow (b \rightarrow a)$

S $(a \rightarrow (b \rightarrow c)) \rightarrow ((a \rightarrow b) \rightarrow (a \rightarrow c))$

(See Hindley (1997) *Table of principal types* for a more extensive listing.) If we read ‘ \rightarrow ’ as implication and type variables as propositional variables, then we recognize three familiar tautologies in the right-hand column of the table. The language used is meager: there are only propositional variables and implication; there are no other connectives.

The table suggests an interesting correspondence between the typed λ -calculus and formal logic. Could it really be that the types assigned to formulas, when understood as logical formulas, are valid? Yes, though ‘validity’ needs to be understood not as classical validity:

Theorem If τ is the type of some λ -term, then τ is intuitionistically valid.

The converse of this theorem holds as well:

Theorem If ϕ is an intuitionistically valid logical formula whose only connective is implication (\rightarrow), then ϕ is the type of some λ -term.

The correspondence can be seen when one identifies intuitionistic validity with derivability in a certain natural deduction formalism. For a proof of these two theorems, see (Hindley, 1997, chapter 6).

The correspondence expressed by the previous two theorems between intuitionistic validity and typability is known as the *Curry-Howard-de Bruijn* correspondence, after three logicians who noticed it independently. The correspondence, as stated, is between only propositional intuitionistic logic, restricted to the fragment containing only the implication connective \rightarrow . One can extend the correspondence to other connectives and to quantifiers, too, but the most crisp correspondence is at the level of the implication-only fragment. For details, see (Howard, 1980).

9.2 Computing

One can represent natural numbers in a simple way, as follows:

Definition (ordered tuples, natural numbers) The ordered tuple $\langle a_0, \dots, a_n \rangle$ of λ -terms is defined as $\lambda x[x a_0 \dots a_n]$. One then defines the λ -term $\ulcorner n \urcorner$ corresponding to the natural number n as: $\ulcorner 0 \urcorner = \mathbf{I}$ and, for every k , $\ulcorner k + 1 \urcorner = \langle \mathbf{F}, \ulcorner k \urcorner \rangle$.

1. The λ -term corresponding to the number 1, on this representation, is:

$$\begin{aligned} \ulcorner 1 \urcorner &\equiv \langle \mathbf{F}, \ulcorner 0 \urcorner \rangle \\ &\equiv \langle \mathbf{F}, \mathbf{I} \rangle \\ &\equiv \lambda x[x \mathbf{F} \mathbf{I}] \end{aligned}$$

2. The λ -term corresponding to the number 2, on this representation, is:

$$\begin{aligned} \ulcorner 2 \urcorner &\equiv \langle \mathbf{F}, \ulcorner 1 \urcorner \rangle \\ &\equiv \lambda x[x \mathbf{F} \lambda x[x \mathbf{F} \mathbf{I}]] \end{aligned}$$

3. Similarly, $\ulcorner 3 \urcorner$ is $\lambda x[x \mathbf{F} \lambda x[x \mathbf{F} \lambda x[x \mathbf{F} \mathbf{I}]]]$.

Various representations of natural numbers are available; this representation is but one. ^[6]

Using the ingredients provided by the λ -calculus, one can represent all recursive functions. This shows that the model is exactly as expressive as other models of computing, such as Turing machines and register machines. For a more detailed discussion of the relation between these different models of computing, see the section comparing the Turing and Church approaches in the entry on the [Church-Turing Thesis](#).

Theorem For every recursive function f of arity n , there exists a λ -term f^* such that

for all natural numbers a_1, \dots, a_n : $f(a_1, \dots, a_n) = y$ iff $\lambda \vdash f^* \langle \bar{a}_1, \dots, \bar{a}_n \rangle = \bar{y}$

For a proof, see [the appendix](#).

Since the class of recursive functions is an adequate representation of the class of all computable (number-theoretic) functions, thanks to the work above we find that all computable (number-theoretic) functions can be faithfully represented in the λ -calculus.

9.3 Relations

The motivation for the λ -calculus given at the beginning of the entry was based on reading λ -expressions as descriptions of functions. Thus, we have understood ' $\lambda x[M]$ ' to be a (or the) function that, given x , gives M (which generally, though not necessarily, involves x). But it is not necessary to read λ -terms as functions. One could understand λ -terms as denoting relations, and read an abstraction term ' $\lambda x[M]$ ' as the unary relation (or property) R that holds of an argument x just in case M does (see Carnap 1947, p. 3). On the relational reading, we can understand an application term MN as a form of predication. One can make sense of these terms using the principle of β -conversion:

$$(\lambda x[M])a = M[x := A],$$

which says that the abstraction relation $\lambda x[M]$, predicated of A , is the relation obtained by plugging in A for all free occurrences of x inside M .

As a concrete example of this kind of approach to λ -calculus, consider an extension of first-order logic where one can form new atomic formulas using λ -terms, in the following way:

Syntax: For any formula ϕ and any finite sequence x_1, \dots, x_n of variables, the expression ' $\lambda x_1 \dots x_n[\phi]$ ' is a predicate symbol of arity n . Extend the notion of free and bound variables (using the functions **FV** and **BV**) in such a way that

$$\mathbf{FV}(\lambda x_1 \dots x_n[\phi]) = \mathbf{FV}(\phi) - \{x_1, \dots, x_n\}$$

and

$$\mathbf{BV}(\lambda x_1 \dots x_n[\phi]) = \mathbf{BV}(\phi) \cup \{x_1, \dots, x_n\}$$

Deduction Assume as axioms the universal closures of all equivalences

$$\lambda x_1 \dots x_n[\phi](t_1, \dots, t_n) \leftrightarrow \phi[x_1, \dots, x_n := t_1, \dots, t_n]$$

where $\phi[x_1, \dots, x_n := t_1, \dots, t_n]$ denotes the simultaneous substitution of the terms t_k for the variables x_k ($1 \leq k \leq n$).

Semantics For a first-order structure A and an assignment s of elements of A to variables, define

$$\begin{aligned} A \models \lambda x_1 \dots x_n[\phi](t_1, \dots, t_n)[s] &\text{ iff} \\ A \models \phi[x_1, \dots, x_n := t_1, \dots, t_n][s] \end{aligned}$$

According to this approach, one can use a λ to treat essentially any formula, even complex ones, as if they were atomic. We see the principle of β -reduction in the deductive and semantic parts. That this approach adheres to the relational reading of λ terms can be seen clearly in the semantics: according to the standard Tarski-style semantics for first-order logic, the interpretation of a formula (possibly with free variables) denotes a set of tuples of elements of the structure, as we vary the variable assignment that assigns elements of the structure to the variables.

One can 'internalize' this functional approach. This is done in the case of various *property theories*, formal theories for reasoning about properties as metaphysical objects (Bealer 1982, Zalta 1983, Menzel 1986, 1993,

and Turner 1987). This kind of theory is employed in certain metaphysical investigations where properties are metaphysical entities to be investigated. In these theories, metaphysical relations are (or are among) the objects of interest; just as we add term-building symbols $+$ and \times in formal theories of arithmetic to build numbers, λ is used in property theory to build relations. This approach contrasts with the approach above. There, λ was added to the grammar of first-order logic by making it a recipe for building atomic formulas; it was a new formula-building operator, like \vee or \rightarrow or the other connectives. In the case of property theories, the λ plays a role more like $+$ and \times in formal theories of arithmetic: it is used to construct relations (which, in this setting, are to be understood as a kind of metaphysical object). Unlike $+$ and \times , though, the λ binds variables.

To give an illustration of how λ is used in this setting, let us inspect the grammar of a typical application (McMichael and Zalta, 1980). One typically has a *predication operator* (or, more precisely, a family of predication operators) $p_k (k \geq 0)$. In a language where we have terms MARY and JOHN and a binary relation loves, we can formally express:

- John loves Mary: loves(JOHN, MARY)
- The property that John loves Mary: $\lambda[\text{loves}(\text{JOHN}, \text{MARY})]$ (note that the λ is binding no variables; we might call this ‘vacuous binding’. Such properties can be understood as propositions.)
- The property of an object x that John loves it: $\lambda x[\text{loves}(\text{JOHN}, x)]$.
- The property that Mary is loved by something: $\lambda[\exists x(\text{loves}(x, \text{MARY}))]$ (another instance of vacuous binding, viz., proposition)
- The predication of the property of x that John loves x to Mary: $p_1(\lambda x[\text{loves}(\text{JOHN}, x)], \text{MARY})$.
- The (0-ary) predication of the property that John loves Mary: $p_0(\lambda x[\text{loves}(\text{JOHN}, \text{MARY})])$.
- The property of objects x and y that x loves y : $\lambda xy[\text{loves}(x, y)]$.
- The property of an objects x that x loves itself: $\lambda x[\text{loves}(x, x)]$.
- The predication of the property of objects x and y that x loves y to John and Mary (in that order): $p_2(\lambda xy[\text{loves}(x, y)], \text{JOHN}, \text{MARY})$.

We reason with these λ -terms using a β -conversion principle such as:

$$p_n(\lambda x_1, \dots, x_n[A], t_1, \dots, t_n) \leftrightarrow A[x_1, \dots, x_n := t_1, \dots, t_n]$$

Formally, the predication operator p_k is a $(k + 1)$ -ary predicate symbol. The first argument is intended to be a λ -term of k arguments, and the rest of the arguments are intended to be the arguments of the body of the λ -term. The β -principle above says that the predication of an n -ary λ -term L to n terms holds precisely when the body of L holds of those terms.

It turns out that in these theories, we may or may not be able to be fully committed to the principle of β -conversion. Indeed, in some property theories, the full principle of β -conversion leads to paradox, because one can replay a Russell-style argument when the full principle of β -conversion is in place. In such settings, one restricts the formation of λ -formulas by requiring that the body of a λ -term not contain further λ -terms or quantifiers. For further discussion, see (Orilia, 2000).

One of the reasons why property theories formulated in the λ -calculus are of a particular philosophical importance is the *hyperintensional* nature of the calculus (see section 1.2). A property concept may be called ‘*hyperintensional*’ if and only if it does not identify necessarily coextensional properties, i.e., properties that are instantiated by exactly the same objects at every possible world. The properties and relations described by the theories of Bealer, Zalta, Menzel, and Turner have exactly this characteristic. In other words, the theories are hyperintensional property theories. Recent years have seen a significant rise of interest in hyperintensional concepts of properties in metaphysics (Nolan 2014), and correspondingly property theories formulated in the λ -calculus will likely experience a rise of interest as well.

In the context of the foundations of mathematics, Zalta and Oppenheimer (2011) argue for the conceptual priority of the relational interpretation of λ -terms over the functional one.

Bibliography

- Baader, Franz and Tobias Nipkow, 1999, *Term Rewriting and All That*, Cambridge: Cambridge University Press.
- Barendregt, Henk, 1985, *The Lambda Calculus: Its Syntax and Semantics* (Studies in Logic and the Foundations of Mathematics 103), 2nd edition, Amsterdam: North-Holland.
- Barendregt, Henk, 1993, “Lambda calculi with types”, in S. Abramsky, D. Gabbay, T. Maibaum, and H. Barendregt (eds.), *Handbook of Logic in Computer Science* (Volume 2), New York: Oxford University Press, pp. 117–309.
- Barendregt, Henk, Wil Dekkers, and Richard Statman., 2013, *Lambda Calculus With Types*, Cambridge: Cambridge University Press.
- Bealer, George, 1982, *Quality and Concept*, Oxford: Clarendon Press.
- van Benthem, Johan, 1998, *A Manual of Intensional Logic*, Stanford: CSLI Publications.
- Carnap, Rudolf, 1947, *Meaning and Necessity*, Chicago: University of Chicago Press.
- Church, Alonzo, 1932, “A set of postulates for the foundation of logic”, *Annals of Mathematics* (2nd Series), 33(2): 346–366.
- Cutland, Nigel J., 1980, *Computability*, Cambridge: Cambridge University Press.
- Doets, Kees and Jan van Eijk, 2004, *The Haskell Road to Logic, Maths and Programming*, London: College Publications.
- Enderton, Herbert B., 2001, *A Mathematical Introduction to Logic*, 2nd edition, San Diego: Harcourt/Academic Press.
- Frege, Gottlob, 1893, *Grundgesetze der Arithmetik*, Jena: Verlag Hermann Pohle, Band I; partial translation as *The Basic Laws of Arithmetic*, M. Furth (trans.), Berkeley: University of California Press, 1964.
- Kleene, Stephen C., 1981, “Origins of recursive function theory”, *Annals of the History of Computing*, 3(1): 52–67.
- Heim, Irene and Angelika Kratzer, 1998, *Semantics in Generative Grammar*, Malden, MA: Blackwell.
- Hindley, J. Roger, 1997, *Basic Simple Type Theory* (Cambridge Tracts in Theoretical Computer Science 42), New York: Cambridge University Press.
- Hindley, J. Roger and G. Longo, 1980, “Lambda-calculus Models and Extensionality.” *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 26: 289–310.
- Hindley, J. Roger and Jonathan P. Seldin, 2008, *Lambda-Calculus and Combinators: An Introduction*, 2nd edition, Cambridge: Cambridge University Press.
- Howard, William A., 1980, “The formula-as-types notion of construction”, in J. Hindley and J. Seldin (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, London: Academic Press, pp. 479–490.
- Hyland, J. Martin E., 2017, “Classical Lambda Calculus in Modern Dress”, *Mathematical Structures in Computer Science*, 27(5): 762–781.
- Koymans, C.P.J., 1982, “Models of the Lambda Calculus”, *Information and Control*, 52: 306–332.
- Manzano, Maria, 2005, *Extensions of First-order Logic* (Cambridge Tracts in Theoretical Computer Science 19), Cambridge: Cambridge University Press.
- McCarthy, John, 1960, “Recursive functions of symbolic expressions and their computation by machine (Part I)”, *Communications of the ACM*, 3(4): 184–195.
- McMichael, Alan and Edward N. Zalta, 1980, “An alternative theory of nonexistent objects”, *Journal of Philosophical Logic*, 9: 297–313.
- Menzel, Christopher, 1986, “A complete, type-free second order logic of properties, relations, and propositions”, Technical Report #CSLI-86-40, Stanford: CSLI Publications.
- Menzel, Christopher, 1993, “The proper treatment of predication in fine-grained intensional logic”, *Philosophical Perspectives* 7: 61–86.
- Meyer, Albert R., 1982, “What is a model of the lambda calculus?”, In *Information and Control*, 52(1): 87–122.
- Nederpelt, Rob, with Herman Geuvers and Roel de Vrieger (eds.), 1994, *Selected Papers on Automath* (Studies in Logic and the Foundations of Mathematics 133), Amsterdam: North-Holland.
- Nolan, Daniel, 2014, “Hyperintensional metaphysics”, *Philosophical Studies*, 171(1): 149–160.
- Orilia, Francesco, 2000, “Property theory and the revision theory of definitions”, *Journal of Symbolic Logic*, 65(1): 212–246.
- Partee, Barbara H., with Alice ter Meulen and Robert E. Wall, 1990, *Mathematical Methods in Linguistics*, Berlin: Springer.


- Plotkin, G.D., 1972, *A Set-Theoretical Definition of Application*, School of Artificial Intelligence, Memo MIP-R-95, University of Edinburgh.
- Revesz, George E., 1988, *Lambda-Calculus, Combinators, and Functional Programming*, Cambridge: Cambridge University Press; reprinted 2008.
- Rosser, J. Barkley, 1984, “Highlights of the History of the Lambda-Calculus”, *Annals of the History of Computing*, 6(4): 337–349.
- Salibra, Antonio, 2003, “Lambda calculus: models and theories”, in *Proceedings of the Third AMAST Workshop on Algebraic Methods in Language Processing (AMiLP-2003)*, No. 21, University of Twente, pp. 39–54.
- Schönfinkel, Moses, 1924, “On the building blocks of mathematical logic”, in J. van Heijenoort (ed.), *From Frege to Gödel: A Source Book in Mathematical Logic*, Cambridge, MA: Harvard University Press, 1967, pp. 355–366.
- Scott, Dana, 1974, “The LAMBDA language”, *Journal of Symbolic Logic*, 39: 425–427.
- , 1980, “Lambda Calculus: Some Models, Some Philosophy”, in J. Barwise, H.J. Keisler, and K. Kunen (eds.), *The Kleene Symposium*, Amsterdam: North-Holland, pp. 223–265.
- Troelstra, Anne and Helmut Schwichtenberg, 2000, *Basic Proof Theory* (Cambridge Tracts in Theoretical Computer Science 43), 2nd edition, Cambridge: Cambridge University Press.
- Turing, Alan M., 1937, “Computability and λ -definability”, *Journal of Symbolic Logic*, 2(4): 153–163.
- Turner, Richard, 1987, “A theory of properties”, *Journal of Symbolic Logic*, 52(2): 455–472.
- Zalta, Edward N., 1983, *Abstract Objects: An Introduction to Axiomatic Metaphysics*, Dordrecht: D. Reidel.
- Zalta, Edward N. and Paul Oppenheimer, 2011, “Relations versus functions at the foundations of logic: type-theoretic considerations”, *Journal of Logic and Computation* 21: 351–374.
- Zerpa, L., 2021, “The Teaching and Learning of the Untyped Lambda Calculus Through Web-Based e-Learning Tools”, in K. Arai *Intelligent Computing* (Lecture Notes in Networks and Systems: Volume 285), Cham: Springer, pp. 419–436.

Academic Tools

 [How to cite this entry.](#)

 [Preview the PDF version of this entry](#) at the [Friends of the SEP Society](#).

 [Look up topics and thinkers related to this entry](#) at the Internet Philosophy Ontology Project (InPhO).

 [Enhanced bibliography for this entry](#) at [PhilPapers](#), with links to its database.

Other Internet Resources

- [The Lambda Calculator](#), a tool for working with λ -terms with an eye toward their use in formal semantics of natural language.
- [Lambda Evaluator](#), for visualizing reductions. Standard combinators and Church numerals are predefined.
- [Lambda calculus reduction workbench](#), for visualizing reduction strategies.
- [“ \$\lambda\$ -Calculus: Then and Now,”](#) useful handout on the milestones in, contributors to, and bibliography on the λ -calculus, presented at the several Turing Centennial conferences. There also exists a [video recording](#) of the lecture given on the occasion of Princeton University’s celebration of the Turing Centennial in 2012.

In addition, see the very helpful discussion in (Zerpa 2021) on the use of e-learning tools for teaching and learning the untyped λ -calculus.

Related Entries

[computer science, philosophy of](#) | [Curry’s paradox](#) | [logic: combinatory](#) | [logic: intensional](#) | [logic: intuitionistic](#) | [logic: second-order and higher-order](#) | [properties](#) | [semantics: Montague](#) | [type theory](#) | [type theory: Church’s type theory](#)

Acknowledgments

The first author wishes to acknowledge the contributions of Henk Barendregt, Elizabeth Coppock, Reinhard Kahle, Martin Sørensen, and Ed Zalta in helping to craft this entry. He also thanks Nic McPhee for introducing him to the λ -calculus.

The second author would like to acknowledge the useful comments and suggestions of Fabrizio Cariani, Cameron Moy, Peter Percival, and Ed Zalta.

[Copyright © 2023](#) by

[Jesse Alama](#)

[Johannes Korbmacher](#) <j.korbmacher@uu.nl>

[Open access to the SEP is made possible by a world-wide funding initiative.](#)

[Please Read How You Can Help Support the Growth and Development of the Encyclopedia](#)

The Stanford Encyclopedia of Philosophy is [copyright © 2023](#) by [The Metaphysics Research Lab](#),
Department of Philosophy, Stanford University

Library of Congress Catalog Data: ISSN 1095-5054