

1 The λ -Calculus

The λ -calculus ($\lambda =$ “lambda”, the Greek letter λ)¹ was introduced by Alonzo Church (1903–1995) and Stephen Cole Kleene (1909–1994) in the 1930s to study the interaction of *functional abstraction* and *functional application*. The λ -calculus provides a succinct and unambiguous notation for the *intensional* representation of functions, as well as a general mechanism based on substitution for evaluating them.

The λ -calculus forms the theoretical foundation of all modern functional programming languages, including Lisp, Scheme, Haskell, OCaml, and Standard ML. One cannot understand the semantics of these languages without a thorough understanding of the λ -calculus.

It is common to use λ -notation in conjunction with other operators and values in some domain (e.g. $\lambda x. x+2$), but the *pure* λ -calculus has only λ -terms and only the operators of functional abstraction and functional application, nothing else. In the pure λ -calculus, λ -terms act as functions that take other λ -terms as input and produce λ -terms as output. Nevertheless, it is possible to code common data structures such as Booleans, integers, lists, and trees as λ -terms. The λ -calculus is computationally powerful enough to represent and compute any computable function over these data structures. It is thus equivalent to Turing machines in computational power.

1.1 Syntax

The following is the syntax of the *pure untyped* λ -calculus. Here *pure* means there are no constructs other than λ -terms, and *untyped* means that there are no restrictions on how λ -terms can be combined to form other λ -terms; every well-formed λ -term is considered meaningful.

A λ -term is defined inductively as follows. Let Var be a countably infinite set of variables x, y, \dots

- Any variable $x \in Var$ is a λ -term.
- If e is a λ -term, then so is $\lambda x. e$ (functional abstraction).
- If e_1 and e_2 are λ -terms, then so is $e_1 e_2$ (functional application).

We often write $e_1(e_2)$ or $(e_1 e_2)$ for $e_1 e_2$. Intuitively, this term represents the result of applying of e_1 as a function to e_2 as its input. The term $\lambda x. e$ represents a function with input parameter x and body e .

1.2 Examples

A term representing the identity function is $id = \lambda x. x$. The term $\lambda x. \lambda a. a$ represents a function that ignores its argument and return the identity function. This is the same as $\lambda x. id$.

The term $\lambda f. f a$ represents a function that takes another function f as an argument and applies it to a . Thus we can define functions that can take other functions as arguments and return functions as results; that is, functions are *first-class values*. The term $\lambda v. \lambda f. f v$ represents a function that takes an argument

¹Why λ ? To distinguish the bound variables from the unbound (free) variables, Church placed a caret on top of the bound variables, thus $\lambda x. x + yx^2$ was represented as $\hat{x}. x + yx^2$. Apparently, the printers could not handle the caret, so it moved to the front and became a λ .

v and returns a function $\lambda f. f v$ that calls its argument—some function f —on v . A function that takes a pair of functions f and g and returns their composition $g \circ f$ is represented by $\lambda f. \lambda g. \lambda x. g (f x)$. We could *define* the composition operator this way.

In the pure λ -calculus, every λ -term represents a function, since any λ -term can appear on the left-hand side of an application operator.

1.3 BNF Notation

Backus–Naur form (BNF) is a kind of grammar used to specify the syntax of programming languages. It is named for John Backus (1924–2007), the inventor of Fortran, and Peter Naur (1928–), the inventor of Algol 60.

We can express the syntax of the pure untyped λ -calculus very concisely using BNF notation:

$$e ::= x \mid e_1 e_2 \mid \lambda x. e$$

Here the e is a *metavariable* representing a *syntactic class* (in this case λ -terms) in the language. It is not a variable at the level of the programming language. We use subscripts to differentiate metavariables of the same syntactic class. In this definition, e_0 , e_1 and e all represent λ -terms.

The pure untyped λ -calculus has only two syntactic classes, variables and λ -terms, but we shall soon see other more complicated BNF definitions.

1.4 Other Domains

The λ -calculus can be used in conjunction with other domains of primitive values and operations on them. Some popular choices are the *natural numbers* $\mathbb{N} = \{0, 1, 2, \dots\}$ and *integers* $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$ along with the basic arithmetic operations $+$, \cdot and tests $=$, \leq , $<$; and the two-element Boolean algebra $\mathbb{2} = \{\text{false}, \text{true}\}$ along with the basic Boolean operations \wedge (and), \vee (or), \neg (not), and \Rightarrow (implication).²

The λ -calculus gives a convenient notation for describing functions built from these objects. We can incorporate them in the language by assuming there is a constant for each primitive value and each distinguished operation, and extending the definition of *term* accordingly. This allows us to write expressions like $\lambda x. x^2$ for the squaring function on integers.

In mathematics, it is common to define a function f by describing its value $f(x)$ on a typical input x . For example, one might specify the squaring function on integers by writing $f(x) = x^2$, or anonymously by $x \mapsto x^2$. Using λ -notation, we would write $f = \lambda x. x^2$ or $\lambda x. x^2$, respectively.

1.5 Abstract Syntax and Parsing Conventions

The BNF definition above actually defines the *abstract syntax* of the λ -calculus; that is, we consider a term to be already parsed into its *abstract syntax tree*. In the text, however, we are constrained to use sequences of symbols to represent terms, so we need some conventions to be sure that they are read unambiguously.

²The German mathematician Leopold Kronecker (1823–1891) was fond of saying, “God created the natural numbers; all else is the work of Man.” Actually, there is not much evidence that God created \mathbb{N} . But for $\mathbb{2}$, there is no question:

And the earth was without form, and void... And God said, Let there be light... And God divided the light from the darkness... —Genesis 1:2–4

We use parentheses to show explicitly how to parse expressions, but we also assign a precedence to the operators in order to save parentheses. Conventionally, function application binds tighter than λ -abstraction; thus $\lambda x. x \lambda y. y$ should be read as $\lambda x. (x \lambda y. y)$, not $(\lambda x. x) (\lambda y. y)$. If you want the latter, you must use explicit parentheses.

Another way to view this is that the body of a λ -abstraction $\lambda x. \dots$ extends as far to the right as it can—it is *greedy*. Thus the body is delimited on the right only by a right parenthesis whose matching left parenthesis is to the left of the λx , or by the end of the entire expression.

Another convention is that application is *left-associative*, which means that $e_1 e_2 e_3$ should be read as $(e_1 e_2) e_3$. If you want $e_1 (e_2 e_3)$, you must use parentheses.

It never hurts to include parentheses if you are not sure.

1.6 Terms and Types

Typically, programming languages have two different kinds of expressions: *terms* and *types*. We have not talked about types yet, but we will soon. A *term* is an expression representing a value; a *type* is an expression representing a class of similar values.

The value represented by a term is determined at runtime by evaluating the term; its value at compile time may not be known. Indeed, it may not even have a value if the evaluation does not terminate. On the other hand, types can be determined at compile time and are used by the compiler to rule out ill-formed terms. When we say a given term has a given type (for example, $\lambda x. x^2$ has type $\mathbb{Z} \rightarrow \mathbb{Z}$), we are saying that the value of the term after evaluation at runtime, if it exists, will be a member of the class of similar values represented by the type.

In the pure untyped λ -calculus, there are no types, and all terms are meaningful.

1.7 Multi-Argument Functions and Currying

We would like to allow multiple arguments to a function, as for example in $(\lambda(x, y). x + y) (5, 2)$. However we can consider this an abbreviation for $(\lambda x. \lambda y. x + y) 5 2$. That is, instead of the function taking two arguments and adding them, the function takes only the first argument and returns a function that takes the second argument and then adds the two arguments. The notation $\lambda x_1 \dots x_n. e$ is considered an abbreviation for $\lambda x_1. \lambda x_2. \lambda x_3. \dots \lambda x_n. e$. Thus we consider the multi-argument version of the λ -calculus as just syntactic sugar. The “desugaring” transformation

$$\begin{aligned} \lambda x_1 \dots x_n. e &\Rightarrow \lambda x_1. \lambda x_2. \lambda x_n. e \\ e_0 (e_1, \dots, e_n) &\Rightarrow e_0 e_1 e_2 \dots e_n \end{aligned}$$

for this particular form of sugar is called *currying* after Haskell B. Curry (1900–1982).

2 Evaluation

Now we arrive at a central question: How does one *evaluate* a λ -calculus term? This is analogous to running a program in a functional language.

The traditional evaluation mechanism of the λ -calculus is based on the notion of *substitution*. The main computational rule is called β -*reduction*. This rule applies whenever there is a subterm of the form $(\lambda x. e_1) e_2$

representing the application of a function $\lambda x. e_1$ to an argument e_2 . The β -reduction rule substitutes e_2 for the variable x in the body of e_1 , then recursively evaluates the resulting expression.

We must be very careful about the formal definitions, however, because trouble can arise if we just substitute terms for variables blindly.

3 Scope, Bound and Free Occurrences, Closed Terms

The *scope* of the abstraction operator λx shown in the term $\lambda x. e$ is its *body* e . An occurrence of a variable y in a term is said to be *bound* in that term if it occurs in the scope of an abstraction operator λy ; otherwise, it is *free*. A bound occurrence of y is *bound to* the abstraction operator λy with the smallest scope in which it occurs. Note that a variable can have bound and free occurrences in the same term, and can have bound occurrences that are bound to different abstraction operators.

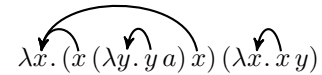


Figure 1: Scope and bindings

For example, in the term shown in Fig. 1, all three occurrences of x are bound. The first two are bound to the first λx , and the last is bound to the second λx . The first occurrence of y is bound, the a is free, and the last y is free, since it is not in the scope of any λy .

This scoping discipline is called *lexical* or *static scoping*. It is called so because the variable's scope is defined by the text of the program, and it is possible to determine its scope before the program runs by inspecting the program text.

3.1 Free Variables

Formally, the set of *free variables* of a term, denoted $FV(e)$, is defined inductively as follows:

$$FV(x) \triangleq \{x\} \qquad FV(e_0 e_1) \triangleq FV(e_0) \cup FV(e_1) \qquad FV(\lambda x. e) \triangleq FV(e) - \{x\}.$$

This definition is inductive on the structure of e . The basis is the leftmost equation, and the other two are the inductive cases. In each of the two inductive cases, the right-hand side defines the value of $FV(e)$ in terms of proper subterms of e , which are smaller. Since all terms have finite size, this means that the definition eventually reaches the base case of variables. This is an example of *structural induction*. We will see many more definitions by structural induction in this course.

A term is *closed* if it contains no free variables; thus all occurrences of any variable x occur in the scope of a binding operator λx . A term is *open* if it is not closed.

4 Substitution

4.1 Variable Capture

Intuitively, to perform β -reduction on the term $(\lambda x. e_1) e_2$, we substitute the argument e_2 for all free occurrences of the formal parameter x in the body e_1 , then evaluate the resulting expression (which may involve further such steps).

However, we cannot just substitute e_2 blindly for x in e_1 because of the problem of *variable capture*. This would occur if e_2 contained a free occurrence of a variable y , and there were a free occurrence of x in the

scope of a λy in e_1 . In that case, the free occurrence of y in e_2 would be “captured” by that λy and would end up bound to it after the substitution, which would incorrectly alter the semantics.

For example, consider the substitution of x for y in $\lambda x. x y$. Raw substitution would yield $\lambda x. x x$. The variable x has been *captured* by the binding operator λx .

To prevent this, we can rename the bound variable x to z to obtain $\lambda z. z y$ before doing the substitution. This transformation does not change the semantics. Now substituting x for y yields $\lambda z. z x$; the variable has not been captured.

4.2 Safe Substitution

This idea leads to the following formal definition of *safe substitution*. The definition is by structural induction. We write $e_1 \{e_2/x\}$ to denote the result of substituting e_2 for all free occurrences of x in e_1 according to the following rules.³

$$\begin{array}{lcl}
 x \{e/x\} & \triangleq & e \\
 y \{e/x\} & \triangleq & y \qquad \text{where } y \neq x \\
 (e_1 e_2) \{e/x\} & \triangleq & (e_1 \{e/x\}) (e_2 \{e/x\}) \\
 (\lambda x. e_0) \{e/x\} & \triangleq & \lambda x. e_0 \\
 (\lambda y. e_0) \{e/x\} & \triangleq & \lambda y. (e_0 \{e/x\}) \qquad \text{where } y \neq x \text{ and } y \notin FV(e) \\
 (\lambda y. e_0) \{e/x\} & \triangleq & \lambda z. (e_0 \{z/y\}) \{e/x\} \qquad \text{where } y \neq x, z \neq x, z \notin FV(e_0), \text{ and } z \notin FV(e).
 \end{array}$$

Note that the rules are applied inductively. That is, the result of a substitution in a compound term is defined in terms of substitutions on its subterms.

The last of the six rules applies when $y \in FV(e)$. In this case, we rename the bound variable y to z to avoid capture of the free occurrence of y . One might well ask: but what if y occurs free in the scope of a λz in e_0 ? Wouldn't the z then be captured? The answer is that it will be taken care of in the same way, but inductively on a smaller term.

Despite the importance of substitution, it was not until the mid-1950's that a completely satisfactory definition of substitution was given by Haskell Curry. Previous mathematicians, from Newton to Hilbert to Church, worked with incomplete or incorrect definitions. It is the last of the rules above that is the hardest to get right, because it is easy to forget one of the three restrictions on the choice of z or to falsely convince oneself that they are not needed.

Rewriting $(\lambda x. e_1) e_2$ to $e_1 \{e_2/x\}$ is the basic computational step of the λ -calculus and is called *β -reduction*. In the pure λ -calculus, we can start with a λ -term and perform β -reductions on subterms in any order.

4.3 Safe Substitution in Mathematics

The problem of variable capture arises in many other mathematical contexts. It can arise anywhere there is a notion of variable binding and substitution.

For example, in the integral calculus, the integral operator is a binder. In the following naive attempt to

³There is no standard notation for substitution. Pierce [Pie02] writes $[x \mapsto e_2]e_1$. Other notations for the same idea are encountered frequently, including $e_1[x \mapsto e_2]$, $e_1[x \leftarrow e_2]$, $e_1[x/e_2]$, and $e_1[x := e_2]$. Because we will be using brackets for other purposes, we will use the notation $e_1 \{e_2/x\}$.

evaluate a definite integral, a variable is incorrectly captured:

$$\int_0^x (1 + \int_0^1 x dx) dy = (y + \int_0^1 yx dx) \Big|_{y=0}^{y=x} = (x + \int_0^1 x^2 dx) - 0 = x + \frac{1}{3}x^3 \Big|_{x=0}^{x=1} = x + \frac{1}{3}$$

This is incorrect. The substitution of x for y under the integral in the second step is erroneous, because x is the variable of integration and is bound by the integral operator, whereas y is free. To fix this, we need only change the variable of integration to z .

$$\int_0^x (1 + \int_0^1 z dz) dy = (y + \int_0^1 yz dz) \Big|_{y=0}^{y=x} = (x + \int_0^1 xz dz) - 0 = x + \frac{1}{2}xz^2 \Big|_{z=0}^{z=1} = \frac{3}{2}x$$

The λ -calculus formalizes this informal notion and provides a solution in the form of safe substitution.

References

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.