Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Lambda calculus

Lecture 7 Tuesday, February 19, 2019

The lambda calculus (or λ-calculus) was introduced by Alonzo Church and Stephen Cole Kleene in the 1930s to describe functions in an unambiguous and compact manner. Many real languages are based on the lambda calculus, such as Lisp, Scheme, Haskell, and ML. A key characteristic of these languages is that functions are values, just like integers and booleans are values: functions can be used as arguments to functions, and can be returned from functions.

The name "lambda calculus" comes from the use of the Greek letter lambda (λ) in function definitions. (The letter lambda has no significance.) "Calculus" means a method of calculating by the symbolic manipulation of expressions.

Intuitively, a function is a rule for determining a value from an argument. Some examples of functions in mathematics are

$$\begin{aligned} f(x) &= x^3 \\ g(y) &= y^3 - 2y^2 + 5y - 6. \end{aligned}$$

1 Syntax

The pure λ-calculus contains just function definitions (called *abstractions*), variables, and function *application* (i.e., applying a function to an argument). If we add additional data types and operations (such as integers and addition), we have an *applied* λ-calculus. In the following text, we will sometimes assume that we have integers and addition in order to give more intuitive examples.

The syntax of the pure λ-calculus is defined as follows.

$$\begin{array}{rll} e ::= & x & \text{variable} \\ \mid & \lambda x.\, e & \text{abstraction} \\ \mid & e_1\ e_2 & \text{application} \end{array}$$

An abstraction $\lambda x.\, e$ is a function: variable x is the *argument*, and expression e is the *body* of the function. Note that the function $\lambda x.\, e$ doesn't have a name. Assuming we have integers and arithmetic operations, the expression $\lambda y.\, y \times y$ is a function that takes an argument y and returns square of y.

An application $e_1\ e_2$ requires that e_1 is (or evaluates to) a function, and then applies the function to the expression e_2. For example, $(\lambda y.\, y \times y)\ 5$ is, intuitively, equal to 25, the result of applying the squaring function $\lambda y.\, y \times y$ to 5.

Here are some examples of lambda calculus expressions.

$\lambda x.\, x$	a lambda abstraction called the *identity function*
$\lambda x.\, (f\ (g\ x)))$	another abstraction
$(\lambda x.\, x)\ 42$	an application
$\lambda y.\, \lambda x.\, x$	an abstraction that ignores its argument and returns the identity function

Lambda expressions extend as far to the right as possible. For example $\lambda x.\, x\ \lambda y.\, y$ is the same as $\lambda x.\, (x\ (\lambda y.\, y))$, and is not the same as $(\lambda x.\, x)\ (\lambda y.\, y)$. Application is left associative. For example $e_1\ e_2\ e_3$ is the same as $(e_1\ e_2)\ e_3$. In general, use parentheses to make the parsing of a lambda expression clear if you are in doubt.

1.1 Variable binding and α-equivalence

An occurrence of a variable in an expression is either *bound* or *free*. An occurrence of a variable x in a term is bound if there is an enclosing $\lambda x.\, e$; otherwise, it is *free*. A *closed term* is one in which all identifiers are bound.

Consider the following term:

$$\lambda x.\, (x\ (\lambda y.\, y\ a)\ x)\ y$$

Both occurrences of x are bound, the first occurrence of y is bound, the a is free, and the last y is also free, since it is outside the scope of the λy.

If a program has some variables that are free, then you do not have a complete program as you do not know what to do with the free variables. Hence, a well formed program in lambda calculus is a closed term.

The symbol λ is a *binding operator*, as it binds a variable within some scope (i.e., some part of the expression): variable x is bound in e in the expression $\lambda x.\, e$.

The name of bound variables is not important. Consider the mathematical integrals $\int_0^7 x^2 dx$ and $\int_0^7 y^2 dy$. They describe the same integral, even though one uses variable x and the other uses variable y in their definition. The meaning of these integrals is the same: the bound variable is just a placeholder. In the same way, we can change the name of bound variables without changing the meaning of functions. Thus $\lambda x.\, x$ is the same function as $\lambda y.\, y$. Expressions e_1 and e_2 that differ only in the name of bound variables are called *α-equivalent* ("alpha equivalent"), sometimes written $e_1 =_\alpha e_2$.

1.2 Higher-order functions

In lambda calculus, functions are values: functions can take functions as arguments and return functions as results. In the pure lambda calculus, every value is a function, and every result is a function!

For example, the following function takes a function f as an argument, and applies it to the value 42.

$$\lambda f.\, f\ 42$$

This function takes an argument v and returns a function that applies its own argument (a function) to v.

$$\lambda v.\, \lambda f.\, (f\ v)$$

2 Semantics

2.1 β-equivalence

Application $(\lambda x.\, e_1)\ e_2$ applies the function $\lambda x.\, e_1$ to e_2. In some ways, we would like to regard the expression $(\lambda x.\, e_1)\ e_2$ as equivalent to the expression e_1 where every (free) occurrence of x is replaced with e_2. For example, we would like to regard $(\lambda y.\, y \times y)\ 5$ as equivalent to 5×5.

We write $e_1\{e_2/x\}$ to mean expression e_1 with all free occurrences of x replaced with e_2. There are several different notations to express this substitution, including $[x \mapsto e_2]e_1$ (used by Pierce), $[e_2/x]e_1$ (used by Mitchell), and $e_1[e_2/x]$ (used by Winskel).

Using our notation, we would like expressions $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$ to be equivalent.

We call this equivalence, between $(\lambda x.\, e_1)\ e_2$ and $e_1\{e_2/x\}$, is called *β-equivalence*. Rewriting $(\lambda x.\, e_1)\ e_2$ into $e_1\{e_2/x\}$ is called a *β-reduction*. Given a lambda calculus expression, we may, in general, be able to perform β-reductions. This corresponds to executing a lambda calculus expression.

There may be more than one possible way to β-reduce an expression. Consider, for example, $(\lambda x.\, x + x)\ ((\lambda y.\, y)\ 5)$. We could use β-reduction to get either $((\lambda y.\, y)\ 5) + ((\lambda y.\, y)\ 5)$ or $(\lambda x.\, x + x)\ 5$. The order in which we perform β-reductions results in different semantics for the lambda calculus.

2.2 Evaluation strategies

There are many different evaluation strategies for the lambda calculus. The most permissive is full β-reduction, which allows any redex—i.e., any expression of the form $(\lambda x.\, e_1)\ e_2$—to step to $e_1\{e_2/x\}$ at any time. It is defined formally by the following small-step operational semantics rules.

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'} \qquad \frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

A term e is said to be in *normal form* when it cannot be reduced any further, that is, when there is no e' such that $e \longrightarrow e'$. It is convenient to say that term e *has* normal form e' if $e \longrightarrow^* e'$ with e' in normal form.

Not every term has a normal form under full β-reduction. Consider the expression $(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$, which we will refer to as Ω for brevity. Let's try evaluating Ω.

$$\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) \longrightarrow (\lambda x.\, x\ x)\ (\lambda x.\, x\ x) = \Omega$$

Evaluating Ω never reaches a term in normal form! It is an infinite loop!

When a term has a normal form, however, it never has more than one. This is not a given, because clearly the full β-reduction strategy is non-deterministic. Look at the term $(\lambda x.\, \lambda y.\, y)\ \Omega\ (\lambda z.\, .z)$, for example. It has two redexes in it, the one with abstraction λx, and the one inside Ω. But this nondeterminism is well behaved: when different sequences of reduction reach a normal form (they need not) those normal forms are equal.

Formally, full β-reduction is confluent in the following sense:

Theorem 1 (Confluence). *If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ then there exists e' such that $e_1 \longrightarrow^* e'$ and $e_2 \longrightarrow^* e'$.*

Confluence can be depicted graphically as follows (where $\twoheadrightarrow$ is used to represent $\longrightarrow^*$):

$$\begin{tikzcd} & e \arrow[dl, two heads] \arrow[dr, two heads] & \\ e_1 \arrow[dr, two heads] & & e_2 \arrow[dl, two heads] \\ & e' & \end{tikzcd}$$

Confluence is often also called the Church-Rosser property. It is not an easy result to prove. (It would make sense for it to be a proof by induction on the multi-step reduction $\longrightarrow^*$.Try it, and see where you get stuck.)

Corollary 1. *If $e \longrightarrow^* e_1$ and $e \longrightarrow^* e_2$ and both e_1 and e_2 are in normal form, then $e_1 = e_2$.*

Proof. An easy consequence of confluence. □

Other evaluation strategies are possible, which impose a deterministic order on the reductions. For example, *normal order evaluation* uses the full β-reduction rules, except imposes the order that the left-most redex—that is, the redex in which the leading λ appears left-most in the term—is always reduced first. Normal order evaluation guarantees that if a term has a normal form, applying reductions in normal order will eventually yield that normal form.

Normal order evaluation allows reducing redexes inside abstractions, which may strike you as odd if you rely on your programmer's intuition: a function definition does not simply reduce its body, unprompted. That's because most programming languages use reduction strategies that when put in lambda calculus terms do not perform reductions inside abstractions.

Two common evaluations strategies that occur in programming languages are call-by-value and call-by-name.

Call-by-value (or CBV) evaluation strategy is more restrictive: it only allows an application to reduce after its argument has been reduced to a value and does not allow evaluation under a λ. That is, given an application $(\lambda x.\, e_1)\; e_2$, CBV semantics makes sure that e_2 is a value before calling the function.

So, what is a value? In the pure lambda calculus, any abstraction is a value. Remember, an abstraction $\lambda x.\, e$ is a function; in the pure lambda calculus, the only values are functions. In an *applied lambda calculus* with integers and arithmetic operations, values also include integers. Intuitively, a value is an expression that can not be reduced/executed/simplified any further.

We can give small step operational semantics for call-by-value execution of the lambda calculus. Here, v can be instantiated with any value (e.g., a function).

$$\frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \qquad \frac{e \longrightarrow e'}{v\; e \longrightarrow v\; e'} \qquad \beta\text{-REDUCTION}\;\frac{}{(\lambda x.\, e)\; v \longrightarrow e\{v/x\}}$$

We can see from these rules that, given an application $e_1\; e_2$, we first evaluate e_1 until it is a value, then we evaluate e_2 until it is a value, and then we apply the function to the value—a β-reduction.

Let's consider some examples. (These examples use an applied lambda calculus that also includes reduction rules for arithmetic expressions.)

$$\begin{aligned}
(\lambda x.\, \lambda y.\, y\; x)\; (5+2)\; \lambda x.\, x+1 &\longrightarrow (\lambda x.\, \lambda y.\, y\; x)\; 7\; \lambda x.\, x+1\\
&\longrightarrow (\lambda y.\, y\; 7)\; \lambda x.\, x+1\\
&\longrightarrow (\lambda x.\, x+1)\; 7\\
&\longrightarrow 7+1\\
&\longrightarrow 8
\end{aligned}$$

$$\begin{aligned}
(\lambda f.\, f\; 7)\; ((\lambda x.\, x\; x)\; \lambda y.\, y) &\longrightarrow (\lambda f.\, f\; 7)\; ((\lambda y.\, y)\; (\lambda y.\, y))\\
&\longrightarrow (\lambda f.\, f\; 7)\; (\lambda y.\, y)\\
&\longrightarrow (\lambda y.\, y)\; 7\\
&\longrightarrow 7
\end{aligned}$$

Call-by-name (or CBN) semantics are more permissive that CBV, but less permissive than full β-reduction. CBN semantics applies the function as soon as possible. The small step operational semantics are a little simpler, as they do not need to ensure that the expression to which a function is applied is a value.

$$\frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \qquad \beta\text{-REDUCTION}\;\frac{}{(\lambda x.\, e_1)\; e_2 \longrightarrow e_1\{e_2/x\}}$$

Let's consider the same examples we used for CBV.

$$\begin{aligned}
(\lambda x.\, \lambda y.\, y\; x)\; (5+2)\; \lambda x.\, x+1 &\longrightarrow (\lambda y.\, y\; (5+2))\; \lambda x.\, x+1\\
&\longrightarrow (\lambda x.\, x+1)\; (5+2)\\
&\longrightarrow (5+2)+1\\
&\longrightarrow 7+1\\
&\longrightarrow 8
\end{aligned}$$

$$\begin{aligned}
(\lambda f.\, f\; 7)\; ((\lambda x.\, x\; x)\; \lambda y.\, y) &\longrightarrow ((\lambda x.\, x\; x)\; \lambda y.\, y)\; 7\\
&\longrightarrow ((\lambda y.\, y)\; (\lambda y.\, y))\; 7\\
&\longrightarrow (\lambda y.\, y)\; 7\\
&\longrightarrow 7
\end{aligned}$$

Note that the answers are the same, but the order of evaluation is different. (Later we will see languages where the order of evaluation is important, and may result in different answers.)

One way in which CBV and CBN differ is when arguments to functions have no normal forms. For instance, consider the following term:

$$(\lambda x.(\lambda y.y))\ \Omega$$

If we use CBV semantics to evaluate the term, we must reduce Ω to a value before we can apply the function. But Ω never evaluates to a value, so we can never apply the function. Under CBV semantics, this term does not have a normal form.

If we use CBN semantics, then we can apply the function immediately, without needing to reduce the actual argument to a value. We have

$$(\lambda x.(\lambda y.y))\ \Omega \longrightarrow_{\text{CBN}} \lambda y.y$$

CBV and CBN are common evaluation orders; many programming languages use CBV semantics. So-called "lazy" languages, such as Haskell, typically use Call-by-need semantics, a more efficient semantics similar to CBN in that it does not evaluate actual arguments unless necessary. However, Call-by-need semantics ensures that arguments are evaluated at most once.