Comp Sci 704
Fall 2024

Lecture 11:
The λ-Calculus

Ethan Cecchetti

To this point, we have worked with a small arithmetic language, Arith, and a small imperative language, Imp, to explore several different types of semantics. Imp, in particular, has the core of a standard imperative programming language, which makes it good for building intuition, but it is lacking in two important ways. First, much of programming languages research aims to investigate fundamental ideas underlying the theory of computation using the simplest possible structures. Imp is considerably more complicated than is fundamentally necessary for this purpose. Second, Imp is missing a critical feature of many languages: functions. The absence of functions makes certain types of reasoning much simpler, but it also how we can write programs.

To address these challenges, we turn to the λ-calculus. Originally introduced by Alonzo Church (1903–1995) and Stephen Cole Kleene[1] (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.

We will now spend some time investigating how it operates and what we can do with it.

1 Syntax of λ-calculus

Syntactically, λ-calculus is extremely simple. Using λ notation with with other operators and values in some domain (e.g. $\lambda x.\, x + 2$) is common, 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.

We now consider pure λ-calculus. The BNF grammar for pure λ-calculus is as follows.

$$e \quad ::= \quad x \mid e_1\; e_2 \mid \lambda x.\, e$$

That is, an expression is either a variable x, a function application, or a *λ-abstraction* (function) $\lambda x.\, e$.

1.1 Parsing Conventions

When reading the concrete syntax of λ-calculus, conventionally function application to bind more tightly (has higher precedence) than abstraction. That means, for example, we read $\lambda x.\, x\; \lambda y.\, y$ as $\lambda x.\, (x\; \lambda y.\, y)$, not $(\lambda x.\, x)\; (\lambda y.\, y)$. If you want the latter, you need explicit parentheses.

Another way to view this convention is that the body of a λ-abstraction $\lambda x. \ldots$ extends as far to the right as it can—it is *greedy*. The body is delimited only by the end of the term or by a right parenthesis whose matching left parenthesis is to the left of the λx.

Another convention is that function application is *left-associative*. That means that $e_1\; e_2\; e_3$ is conventionally interpreted as $(e_1\; e_2)\; e_3$. If you want $e_1\; (e_2\; e_3)$, you mush include parentheses. We will see in a moment why this convention is useful for expressing mutli-argument functions.

As a general rule, it never hurts to include parentheses if you are not sure.

[1]Kleene spent most of his career right here at UW–Madison. He joined the Department of Mathematics in 1935, left from 1941–1946, mostly in the navy, and later joined the Department of Numerical Analysis (renamed the Department of Computer Sciences in 1964). He also served as the Dean of the College of Letters & Sciences from 1969–1974 before retiring in 1979.

1.2 Multi-Argument Functions and Currying

We would like to allow functions of multiple arguments, as in $(\lambda(x, y).\, x + y)\ (5, 2)$, but we do not need a special primitive to do this. Instead, because λ-calculus functions return other λ-calculus terms, which are simply functions, we can write, $(\lambda x.\, \lambda y.\, x + y)\ 5\ 2$. That is, instead of a function taking two arguments and adding them together, we have a function of one argument that *returns another function of one argument*, and that second function will add its argument to the argument from the first (outer) function. We will sometimes use the notation $\lambda x_1 \ldots x_n.\, e$ as a shorthand for $\lambda x_1.\, \lambda x_2.\, \ldots \lambda x_n.\, e$.

This particular shorthand (or "syntactic sugar") is called *currying*, after Haskell Curry (1900–1982).

Notice that the left associativity of function application is very helpful here. If we think of $(\lambda xy.\, e_0)\ e_1\ e_2$ as a two-argument function taking e_1 and e_2 as arguments, that produces the same result as currying the function and interpreting $(\lambda x.\, \lambda y.\, e_0)\ e_1\ e_2$ using our standard convention.

2 Evaluating λ-calculus

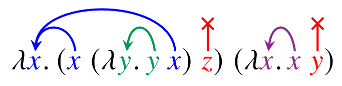
The traditional evaluation mechanism of the λ-calculus is based on the notion of substitution. The main computational rule is called β-reduction. This rule applies whenever there is a subterm of the form $(\lambda x.\, e_1)\ e_2$ representing the application of a function $\lambda x.\, e_1$ to an argument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

2.1 Scope, Bound and Free Variables

An *occurrence* of a variable x in a λ-term is a leaf of the abstract syntax tree containing a variable; that is, we do not include those appearing in the binding operator λx. The *scope* of an abstraction operator λx in the term $\lambda x.\, e$ is its *body* e. Each occurrence of a variable is either *bound*, if it occurs within the scope of an abstraction operator binding that variable, or it is *free* otherwise. If a variable y occurs within the scope of more than one abstraction operator binding y, the variable is bound to the operator with the *smallest scope*.

Note that a variable can have both bound and free occurrences in the same term, and can have bound occurrences that are bound to different abstraction operators. For example, in the term below, there are three occurrences of x, two of y, and one of a.



All three occurrences of x are bound. The first two (blue) are bound to the first λx, and the last (purple) is bound to the second λx. The first occurrence of y (green) is bound, while the z (red) is free, and the last y (also red) is also free, since it is not in the scope of any λy.

This scoping discipline is called *lexical* or *static* scoping because the variable's scope is defined by the text of the program. It is possible to determine its scope before the program runs by inspecting the program text. We will see other kinds of scoping later in the course.

Free Variables. It is useful to be able to refer to the set of free variables in a term. We therefore define the function $\mathrm{FV}(e)$ recursively on e as follows.

$$\mathrm{FV}(x) \triangleq \{x\} \qquad \mathrm{FV}(e_1\ e_2) \triangleq \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \qquad \mathrm{FV}(\lambda x.\, e) \triangleq \mathrm{FV}(e) - \{x\}$$

An expression e is said to be *closed* if it contains no free variables ($\mathrm{FV}(e) = \varnothing$), and *open* otherwise.

2.2 β-reduction

To evaluate a λ-calculus term, we perform β-reduction. Intuitively, to reduce the term $(\lambda x.\, e_1)\ e_2$, we substitute the argument e_2 for every free occurrence of the formal parameter x in the body e_1, and then evaluate the resulting expression. Formally, we write

$$(\lambda x.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2].$$

An instance of the left-hand side is called a *redex* and the corresponding instance on the right-hand side is called a *contractum* (though this latter term is rarely used). In pure λ-calculus, a β-reduction may be performed at any time on any subterm that is the redex of the β-rule. The rule is applied by replacing the redex by its corresponding contractum. For example,

$$\lambda x.\, \underbrace{(\lambda y.\, y)\ x}_{\text{redex}} \longrightarrow \lambda x.\, x$$

Here the subterm $(\lambda y.\, y)\ x$, which is a redex of the β-rule, is replaced by its contractum $x = y[y \mapsto x]$.

2.3 Safe Substitution

Notably, when performing the substitution in β-reduction, we cannot substitute e_2 blindly into e_1. If we did, it could result in a problem called *variable capture*. Variable capture occurs when a free variable incorrectly becomes bound. That is, if some variable y is free in the argument e_2, then it should remain free in the copy of e_2 inserted for any formal parameter x. However, if there is a free occurrence of x in e_1 *inside the body of a binder* λy, the free occurrence of y in e_2 may be captured, which would incorrectly alter the semantics.

For example, consider the term $\lambda x.\, x\ y$. If we were to substitute x for y blindly, it would result in the term $\lambda x.\, x\ x$, which has a very different meaning!

To prevent variable capture, we introduce a notion called *safe substitution* that renames variables to prevent variable capture. In the example above, when substituting x for y into $\lambda x.\, x\ y$, we could rename x to z in the λ-term, resulting in the term $\lambda z.\, z\ x$, which properly avoids capturing x.

This idea allows us to inductively define safe substitution on λ-calculus terms, which we denote $e_0[x \mapsto e_1]$. The definition relies on the FV function defined above to compute the free variables of a term.

$$\begin{array}{rcll}
x[x \mapsto e] & \triangleq & e & \\
y[x \mapsto e] & \triangleq & y & \text{where } y \neq x \\
(e_1\ e_2)[x \mapsto e] & \triangleq & (e_1[x \mapsto e])\ (e_2[x \mapsto e]) & \\
(\lambda x.\, e_0)[x \mapsto e] & \triangleq & \lambda x.\, e_0 & \\
(\lambda y.\, e_0)[x \mapsto e] & \triangleq & \lambda y.\, (e_0[x \mapsto e]) & \text{where } y \neq x \text{ and } y \notin \mathrm{FV}(e) \\
(\lambda y.\, e_0)[x \mapsto e] & \triangleq & \lambda z.\, (e_0[y \mapsto z][x \mapsto e]) & \text{where } y \neq x,\, z \neq x, \\
 & & & z \notin \mathrm{FV}(e_0), \text{ and } z \notin \mathrm{FV}(e)
\end{array}$$

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 first five rules define how to substitute variables, function application, and λ abstractions when either the variable being substituted is bound or no variable capture will occur. The final rule applies when $y \in \mathrm{FV}(e)$, which is when a blind substitution would result in variable capture. In this case, we rename the bound variable from y to z for some z that is not free in either e_0 or e. One might ask: what if y appears free inside a λz inside e_0? The answer is that it will be taken care of the same way, but renaming the inner bound variable z in the smaller term.

Despite the importance of substitution, it was not until the mid-1950s that a completely satisfactory definition 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. 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.

2.3.1 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 evaluate a definite integral, a variable is incorrectly captured:

$$\int_0^x \left(1 + \int_0^1 x\,dx\right) dy \;=\; \left(y + \int_0^1 yx\,dx\right)\Bigg|_{y=0}^{y=x} \;=\; \left(x + \int_0^1 x^2\,dx\right) - 0 \;=\; x + \left(\frac{1}{3}x^3\right)\Bigg|_{x=0}^{x=1} \;=\; x + \frac{1}{3}$$

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

$$\int_0^x \left(1 + \int_0^1 x\,dx\right) dy \;=\; \left(y + \int_0^1 yx\,dx\right)\Bigg|_{y=0}^{y=x} \;=\; \left(x + \int_0^1 xz\,dz\right) - 0 \;=\; x + \left(\frac{1}{2}xz^2\right)\Bigg|_{z=0}^{z=1} \;=\; \frac{3}{2}x$$

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

2.4 Operational Semantics

To formalize the notion of β-reduction above, we can write it as a small-step operational semantics. This will require a single axiom for the basic substitution-based β-rule, as well as some inductive rules. Because reductions are allowed on any subterm, we must have inductive rules for each possible subterm. The result is the following set of operational semantic rules.

$$\frac{}{(\lambda x.\, e_1)\; e_2 \longrightarrow e_1[x \mapsto e_2]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\; e_2 \longrightarrow e_1\; e_2'} \qquad \frac{e \longrightarrow e'}{\lambda x.\, e \longrightarrow \lambda x.\, e'}$$

2.5 α-conversion

The solution to safe substitution relies on a deep fact about λ-calculus: the names of the variables being bound are irrelevant as long as everything is consistent. For instance $\lambda x.\, x\; y$ and $\lambda z.\, z\; y$ are semantically identical.

Renaming like this is know as *α-conversion* or *α-renaming*. In α-conversion, the new variable name must be chosen to avoid variable capture. If one term α-converts to another, the two terms are said to be *α-equivalent*, as the conversion works equally well in both directions. This defines an equivalence relation on the set of terms denoted $e_1 =_\alpha e_2$.

Formally, we can define α-equivalence by

$$\lambda x.\, e \;=_\alpha\; \lambda y.\, e[x \mapsto y] \text{ if } y \notin \mathrm{FV}(e).$$

The requirement that $y \notin \mathrm{FV}(e)$ avoids capturing free instances of y by the newly-named binder λy. The y substituted for x cannot be captured by a binding operator λy already in e because safe substitution $e[x \mapsto y]$ would not let that happen—it would rename the bound variable accordingly.

Also note that, while we do formally define safe substitution in terms of α-conversion—but vice versa—the renaming operation in the last rule of safe substitution relies on the same intuition as α-equivalence for its validity: changing names of variables does not change semantics, as long as we avoid capture.

3 Values and Ω

In classic λ-calculus, a *value* is a term with no β-redexes. Such a term is said to be in *normal form*; no further β-reductions can be applied. Starting from some λ-term, we might perform β-reductions as long as possible, seeking to produce a value.

Do we always find a value eventually? Let us define an expression we will call Ω:

$$\Omega \triangleq (\lambda x.\, x\, x)\ (\lambda x.\, x\, x)$$

What happens when we try to evaluate it?

$$\Omega = (\lambda x.\, x\, x)\ (\lambda x.\, x\, x) \longrightarrow (x\, x)[x \mapsto \lambda x.\, x\, x] = (\lambda x.\, x\, x)\ (\lambda x.\, x\, x) = \Omega$$

We have just coded an infinite loop! Thus the term Ω has no value.

4 Confluence

A λ-term in general may have many redexes. A *reduction strategy* is a rule for determining which redex to reduce next. We can think of a reduction strategy as a mechanism for resolving the nondeterminism. In the classical λ-calculus, no reduction strategy is specified; any redex may be chosen to be reduced next, so the process is nondeterministic. One sequence of reductions may terminate, but another may not. It is even conceivable that different terminating reduction sequences result in different values. Luckily, it turns out that the latter cannot happen.

It turns out that the λ-calculus is *confluent* under α- and β-reductions. Confluence says that if e reduces by some sequence of reductions to e_1, and if e also reduces by some other sequence of reductions to e_2, then there exists an e_3 such that both e_1 and e_2 reduce to e_3, as illustrated in the diagram below.

$$\begin{tikzcd}
 & e \arrow[dl] \arrow[dr] & \\
e_1 \arrow[dr] & & e_2 \arrow[dl] \\
 & e_3 &
\end{tikzcd}$$

It follows that normal forms are unique up to α-equivalence. That is, if $e \longrightarrow^* v_1$ and $e \longrightarrow^* v_2$, where v_1 and v_2 are both values, then $v_1 =_\alpha v_2$. Moreover, regardless of the order of previous reductions, it is always possible to get to the unique normal form if it exists.

Confluence is sometimes referred to as the *Church–Rosser property*, named for Alonzo Church and J. Barkley Rosser[2] (1907–1989), who first proved it for λ-calculus.

Note that it is still possible for a reduction sequence not to terminate, even if the term has a normal form. For example, $(\lambda xy.\, y)\ \Omega$ has a nonterminating reduction sequence

$$(\lambda xy.\, y)\ \Omega \longrightarrow (\lambda xy.\, y)\ \Omega \longrightarrow \cdots$$

by applying β-reduction to Ω repeatedly. However, there is also a terminating sequence, namely

$$(\lambda xy.\, y)\ \Omega \longrightarrow \lambda y.\, y$$

by applying β-reduction to the whole term. Confluence guarantees that, even if we get stuck in a loop, if a normal form exists, it is always possible to get unstuck and reach that normal form.

[2]Rosser was also a long-time UW–Madison faculty member as director of the Mathematical Research Center from 1963–1978!