CS704: Lecture 4
Reduction, Transitions, Equality, and Encodings in λ-Calculus

Thomas Reps
[Based on notes taken by Rich Joiner on January 27, 2010]

Abstract

This lecture explores further topics relating to λ-calculus, including the α, β, and η reduction rules, an analysis of equality, congruence, and inequality of λ-terms, and an introduction to the use of encodings to compensate for the lack of data in pure λ-calculus.

1 Review

1.1 Substitution

The substitution of N for all free occurrences of variable x in M is denoted by $M[N/x]$, and often read as "M with N in for x". The action taken to apply the substitution depends on the form of M, as defined below:

$$\begin{array}{rcl}
x[N/x] & \stackrel{\text{def}}{=} & N \\
y[N/x] & \stackrel{\text{def}}{=} & y \text{ iff } y \not\equiv x \\
(PQ)[N/x] & \stackrel{\text{def}}{=} & (P[N/x]Q[N/x]) \\
(\lambda x.P)[N/x] & \stackrel{\text{def}}{=} & \lambda x.P \\
(\lambda y.P)[N/x] & \stackrel{\text{def}}{=} & \lambda z.((P[z/y])[N/x]), \text{ where } y \not\equiv x \text{ and } z \text{ is fresh}
\end{array}$$

In the last case, the inner recursive call on the substitution operator introduces a fresh variable z to replace all free occurrences of y in P. That way, we can change $(\lambda y. \ldots)$ to $(\lambda z. \ldots)$ so that the outer recursive call on the substitution operator avoids any possibility of capturing free occurrences of y in N.

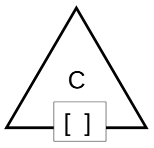
1.2 Reduction Rules

λ-calculus employs two reduction rules, α- and β-reduction, which are defined as follows:
α-reduction: $\lambda x.M \rightarrow_\alpha \lambda y.M[y/x]$
β-reduction: $(\lambda x.M)N \rightarrow_\beta M[N/x]$.
α-reduction represents the renaming of a formal parameter. β-reduction represents the application of a function to an actual parameter.

2 From Reduction Rules to a Transition System

Recall that in Lecture 1 we used a finite number of *rules* to define an infinite *transition system.* For instance, a single string-rewriting rule "$PP \rightarrow P$" defined an infinite number of transitions of the form "$\gamma PP\delta \Rightarrow \gamma P\delta$," where γ and δ are strings that form the *context* for the application of the rewriting rule.

We have the identical situation in λ-calculus: we have two rules, α-reduction and β-reduction; these define an infinite transition system on λ-calculus terms, which is formalized using the notion of a *context.* In this case, a context can be thought of as a tree with a hole in it as shown below:



The formal definition of a context is as follows:

Definition 2.1 (Context)

$$\begin{array}{rcl} exp & ::= & var \\ & | & (exp\ exp) \\ & | & (\lambda var.exp) \\ \\ context & ::= & [\ ] \\ & | & (context\ exp) \\ & | & (exp\ context) \\ & | & (\lambda var.context) \end{array}$$

A generic context will be denoted by $C[\ ]$, where "$[\ ]$" denotes the hole. When we want to emphasize that a term is partitioned into a context $C[\ ]$ and a sub-term T, where T fills the hole of $C[\ ]$, we use $C[T]$. $\square$

The transition system of λ-calculus is defined as follows:

Definition 2.2 (λ-Calculus Transitions) *For all $U_1 \overset{\text{def}}{=} C[T_1]$ and $U_2 \overset{\text{def}}{=} C[T_2]$, there is a transition*

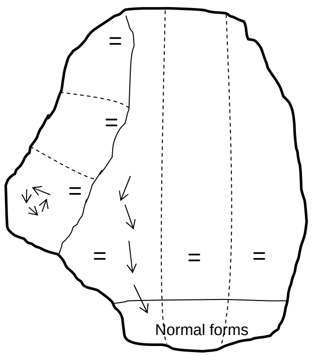
$$C[T_1] \longrightarrow C[T_2] \text{ if } T_1 \longrightarrow_\alpha T_2 \text{ or } T_1 \longrightarrow_\beta T_2.$$

For emphasis, we sometimes denote a transition by $C[T_1] \longrightarrow_{\alpha,\beta} C[T_2]$, or by $U_1 \longrightarrow_{\alpha,\beta} U_2$ if it is not necessary to emphasize the context. $\square$

3 Notions of Equality

3.1 Equality and Congruence in λ-Calculus

We want a notion of equality between λ-terms. For instance, one idea is that two terms that reduce to the same normal form are equal. However, it also seems desirable to be able to say that terms that do not have a normal form are equal (e.g, if one reduces to another they represent equal values). The situation we would like to have is suggested by the following diagram:



We can formalize this notion by allowing $M \rightarrow N$ to induce the equality $M = N$, and then forming equivalence classes in the usual way, via rules of reflexivity, symmetry, and transitivity:

Definition 3.1 (Equality and Congruence) *The notion of* equality on λ-terms*, is defined as follows:*

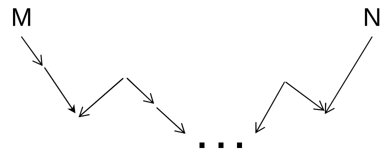
$$\begin{aligned}&\textit{if } M \longrightarrow_{\alpha,\beta} N \textit{ then } M = N\\ &M = M\\ &\textit{if } M = N \textit{ then } N = M\\ &\textit{if } M = N \textit{ and } N = P \textit{ then } M = P\end{aligned}$$

Moreover, two λ-terms M and N are said to be congruent*, denoted by $M \simeq N$, iff $M = N$ via a sequence of 0 or more transitions using only α-reduction steps.* □

Translated into English, what equality means is the following:

> Two λ-terms M and N are equal if there is a "zig-zag" path of reductions and expansions that connects them.

Pictorially, $M = N$ means that there is a path like the one shown below (where an arrow indicates the direction of a β-reduction):



We use α-congruence to avoid having to make distinctions among terms that differ only in the names chosen for formal parameters. The relation $\simeq$ defines equivalence classes of α-congruent λ-terms. The terms in such an equivalence class are said to be *identical up to α-congruence*.[1] Later, we will show that an answer (normal form) computed in λ-calculus (if the starting term has a normal form) is unique up to α-congruence. That is, strictly speaking λ-calculus does not have the unique normal-form property, but it does have the property up to α-congruence.

3.2 The η-Reduction Rule

The λ-calculus that we have been working with is called the $\alpha\beta$-λ-calculus. A related (still pure) λ-calculus has a third reduction rule, called the η-reduction rule. The augmented λ-calculus forms a whole new system, called the $\alpha\beta\eta$-λ-calculus, which is slightly different from the $\alpha\beta$-λ-calculus. (It shares many of the same properties, such as the Church-Rosser property that will be considered later; however, in general, it is necessary to provide separate proofs of a given property in each system.)

The η-reduction rule is defined as follows:

$$\lambda x.Mx \longrightarrow_\eta M, \text{ provided that } x \text{ is not free in } M.$$

The addition of this rule naturally has an effect on the transition system defined:

$$C[T_1] \longrightarrow C[T_2] \text{ iff } T_1 \longrightarrow_\alpha T_2 \text{ or } T_1 \longrightarrow_\beta T_2 \text{ or } T_1 \longrightarrow_\eta T_2.$$

[1]Note that when the De Bruijn representation for λ-terms is used, all of the terms in a given α-congruence class are, in fact, identical.

with equality defined by the analog of Defn. 3.1.

The motivation behind the η-reduction rule is that one can "observe" how a λ-term behaves by "standing it next to" another term and seeing what results. In particular, the application of each side of the η-rule to any given λ-term Q always produces the same result, namely MQ, because

$$(\lambda x.Mx)Q \longrightarrow_\eta MQ.$$

Because $(\lambda x.Mx)$ and M always produce equal results (under our "stand next to an arbitrary term Q" test), we would like them to be considered equal. By adding the reduction rule $\lambda x.Mx \longrightarrow_\eta M$ (provided that x is not free in M), the corresponding notion of equality in the $\alpha\beta\eta$-λ-calculus has the desired property: $\lambda x.Mx = M$.

3.3 Intensional and Extensional Equality

Compared to the equivalence classes in the $\alpha\beta$-λ-calculus, the equivalence classes in the $\alpha\beta\eta$-λ-calculus are coarser. That is, the $\alpha\beta\eta$-λ-calculus allows some λ-terms to be considered equal that are not considered to be equal in the $\alpha\beta$-λ-calculus.

We can characterize the type of equality that the $\alpha\beta\eta$-λ-calculus is intended to capture via the notions of intensional equality and extensional equality. In logic, two objects are *extensionally equal* if they have the same external properties. For instance, in our case we are concerned with two representations of a function—two λ-terms—and the external properties are their "input/output" (argument/result) behavior (or, alternatively, the "graphs" of their functions are identical). Two objects are *intensionally equal* when their descriptions are the same; in the case of two representations of a function, "the same" means that they use the same rules/recipes for performing the computation. Extensionally equality can be illustrated for an applied λ-calculus using the "add 1" function.

$$\lambda x.x + 1$$
$$\lambda x.(x + 17) - 16$$

Because both of the expressions above produce the same computation graph, namely the set of pairs $\{(0, 1), (1, 2), (2, 3), \ldots\}$, the terms are extensionally equal.

Terms that are equal (=) in the $\alpha\beta\eta$-λ-calculus are not intensionally equal, because $\lambda x.Mx$ and M perform (slightly) different computations. However, $\lambda x.Mx$ and M are extensionally equal because the application of both terms produces the same result.

3.4 An Example Reduction in the $\alpha\beta\eta$-λ-Calculus

Example 3.2 The following example is due to Stoy [1, p. 69]:

$$\begin{aligned} \text{Let } S &\stackrel{\text{def}}{=} \lambda x.\lambda y.\lambda z.xz(yz) \\ K &\stackrel{\text{def}}{=} \lambda x.\lambda y.x \\ I &\stackrel{\text{def}}{=} \lambda x.x \\ \text{Show that } S(KI) &= I \end{aligned}$$

$$\begin{aligned} S(KI) = \quad & (\lambda x.\lambda y.\lambda z.xz(yz))((\lambda x.\lambda y.x)(\lambda x.x)) \\ \longrightarrow_\beta \quad & (\lambda x.\lambda y.\lambda z.xz(yz))(\lambda y.\lambda x.x) \\ \longrightarrow_{\alpha,\beta} \quad & \lambda y.\lambda z.((\lambda a.\lambda b.b)z)(yz) \\ \longrightarrow_\beta \quad & \lambda y.\lambda z.(\lambda b.b)(yz) \\ \longrightarrow_\beta \quad & \lambda y.\lambda z.yz \text{ (which has no more } \beta\text{-redexes)} \\ \longrightarrow_\eta \quad & \lambda y.y = I \end{aligned}$$

□

Remark. As an historical aside, its interesting to note the relationship of λ-calculus (developed by Church) and combinatory logic (developed by Schönfinkel and Curry). The combinators of combinatory logic, such as S, K, and I, can be encoded as λ-terms (as S, K, and I were above). In combinatory logic, there is a rewriting rule associated with each of S, K, and I:

$$\begin{aligned} SABC &\longrightarrow AC(BC) \\ KAB &\longrightarrow A \\ IA &\longrightarrow A \end{aligned}$$

Moreover, any closed λ-term can be translated into an equivalent combinatory-logic term expressed exclusively with the combinators S, K, and I. (There are also other translation schemes, which yield shorter results, that involve other combinators, such as $B \stackrel{\text{def}}{=} \lambda x.\lambda y.\lambda z.x(yz)$.) Such ideas were once thought to be useful as the basis for creating special-purpose computers to run functional programming languages. In the past, functional programming languages have had much appeal in a certain segment of the programming-languages community, although their usage in practice has always been quite limited. With the advent of computations that involve massive numbers of independent computing elements (e.g., cloud computing, data centers, and multi-core CPUs), perhaps the situation will change, in which case such ideas could conceivably be resurrected. □

3.5 Inequality

The proper treatment of inequality will come from the Church-Rosser Theorem. Here, we will just give a "conditional" treatment of inequality, using a rather minimal assumption—i.e., that there are least two unequal terms (say M and N). We assume nothing about the structure of M and N, just that they are unequal (i.e., there is no "zig-zag" path of reductions and expansions that connects them).

Under this assumption, we can proceed to show that specific terms are not equal, using an argument by contradiction:

Example 3.3 Let $T \stackrel{\text{def}}{=} \lambda x.\lambda y.x$, $F \stackrel{\text{def}}{=} \lambda x.\lambda y.y$, and M and N be two non-equal λ-terms. Assume for the sake of argument that $T = F$.

$$\begin{aligned} TM &= FM \\ TMN &= FMN \\ (\lambda x.\lambda y.x)MN &= (\lambda x.\lambda y.y)MN \end{aligned}$$

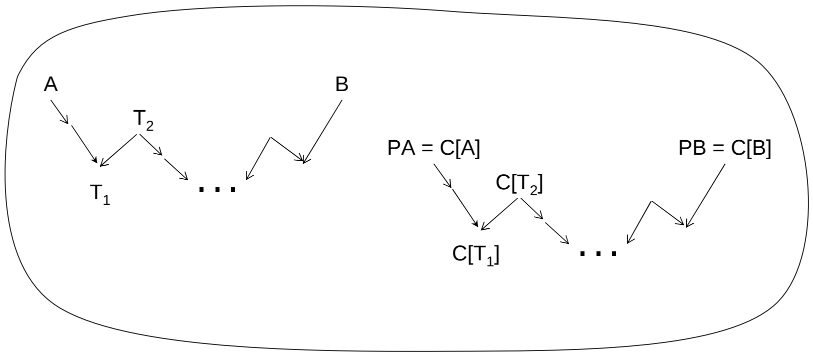
However, the left-hand side β-reduces in two steps to M, and the right-hand side β-reduces in two steps to N, which means $M = N$ (contradicting our assumption that $M \neq N$). □

3.6 An Alternative Way to Define Equality of λ-Terms

The following system defines an alternative way to define equality of λ-terms.

$$\begin{aligned}
\lambda x.M &= \lambda y.M[y/x] \\
(\lambda x.M)N &= M[N/x] \\
M &= M \\
&\text{if } M = N \text{ then } PM = PN \\
&\text{if } M = N \text{ then } MP = NP \\
&\text{if } M = N \text{ then } \lambda x.M = \lambda x.N \\
&\text{if } M = N \text{ then } N = M \\
&\text{if } M = N \text{ and } N = P \text{ then } M = P
\end{aligned}$$

Given the rules above, we know that if $A = B$, then for all P, $PA = PB$. Another way of thinking about this is that if $A = B$, there is a zig-zag path between them. From that path, we can construct a zig-zag path between PA and PB, by wrapping all terms in the context $C[\,] \overset{\text{def}}{=} (P[\,])$



On the other hand, if $PA = PB$, it is not necessarily valid that $A = B$! The example below illustrates why:

$$(\lambda x.M)T = M = (\lambda x.M)F$$

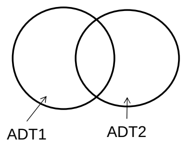
However, if we drop $(\lambda x.M)$ from both sides, we are left with $T = F$ (contradicting our previous demonstration that $T \neq F$).

4 Encodings

We can overcome the lack of explicit "data" in the λ-calculus by using *encodings*. That is, we will identify a subset of the λ-terms as representing a given kind of data, and design combinators that operate on those terms to mimic desired operations on data of that kind. In other words, we use encodings to define abstract data types (ADTs).

For a data value V, the combinator that represents V is denoted as $\underline{V}$. We have already seen the combinators that we will use at the encodings of the Booleans T ($\underline{T} \overset{\text{def}}{=} \lambda x.\lambda y.x$) and F ($\underline{F} \overset{\text{def}}{=} \lambda x.\lambda y.y$).

As it turns out, in one of the systems for encoding the natural numbers, $\underline{0}$ (which encodes 0) is represented with the same term as $\underline{F}$. This shows that the following Venn Diagram of the encodings of two ADTs is possible.



The if-then-else combinator is defined to be $\lambda b.\lambda m.\lambda n.bmn$. The if-then-else combinator is intended to build a term of the form TMN or FMN, so that the T and F act a selectors of the result (i.e., M and N, respectively).

The discussion above illustrates a general requirement of encodings: an application of an encoded operation on the λ-terms that encode the intended data must produce the intended result. (However, the result of an application of an encoded operation on a λ-term outside the set of terms that represent the intended data is not restricted.)

Remark. Under $\alpha\beta\eta$-λ-calculus, if-then-else is equal to the identity function. □

References

[1] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The M.I.T. Press, 1977.