1 Introduction to Lambda Calculus

The lambda calculus is a very small but very expressive programming language. It is based on the idea of defining the behavior of functions by textually substituting arguments for input variables. It is also Turing complete: any function that can be computed with a Turing machine can be computed with a lambda calculus program.

The lambda calculus is due to Alonzo Church (see "The Calculi of Lambda Conversion", Princeton U. Press, 1941). Its ideas are incorporated in modern functional programming languages like OCaml and Haskell, and also used crucially in many branches of logic, particularly constructive logic, as well as in theorem provers and computer-checked proofs.

2 Syntax

Lambda calculus expressions are called terms. The syntax for terms $t$ is:

$$ t ::= x \mid (t t') \mid \lambda x. t $$

Here, $x$ is for variables, $(t t')$ is for applications of $t$ as a function to $t'$ as an argument, and $\lambda x. t$ is a lambda-abstraction, an anonymous function which takes input $x$ and returns output $t$.

2.1 Examples

- Assuming we have defined $\text{mult}$ somehow (we’ll see how later) to multiply numbers encoded as lambda terms, then the following term defines the squaring function:

  $$ \lambda x. ((\text{mult} x) x) $$

  Note that applications of a function like $\text{mult}$ to two arguments must be written in left-nested form: we apply $\text{mult}$ to the first argument $x$, and then apply the application $(\text{mult} x)$ to the second argument $x$. Because it is a bit cumbersome to write all these parentheses, by convention parentheses associate to the left, so we can write $(\text{mult} x x)$ instead of $((\text{mult} x) x)$.

- The following term takes a function $f$ and argument $x$ as inputs, and returns $(f (f x))$:

  $$ \lambda f. \lambda x. (f (f x)) $$

- The following term can be thought of as defining the composition of functions $f$ and $g$:

  $$ \lambda f. \lambda g. \lambda x. (f (g x)) $$

  Let us call that term $\text{compose}$. Then the following term behaves just the way the composition of functions $f$ and $g$ should:

  $$ (\text{compose} f g) $$

  This term, if given now any argument $a$, will return:

  $$ (f (g a)) $$

  The application $(\text{compose} f g)$ can itself be used as a function, that is just waiting for the argument $a$ in order to return $(f (g a))$. 

3 Operational Semantics

The starting point for studying the several different operational semantics for lambda-calculus is a non-deterministic reduction semantics called full $\beta$-reduction (“beta-reduction”). This semantics is defined by the following rules:

\[
(\lambda x. t) t' \rightarrow [t'/x]t \quad (\beta) \\
t \rightarrow t' \quad \text{cong-lam} \\
t_1 \rightarrow t'_1 \quad \text{cong-app1} \\
(t_1 t_2) \rightarrow (t'_1 t'_2) \quad \text{cong-app2} \\
t_2 \rightarrow t'_2
\]

The first rule is called the $\beta$-reduction rule, and passing from $(\lambda x. t) t'$ to $[t'/x]t$ is called $\beta$-reduction. Here, I am using the standard lambda-calculus notation $[t'/x]t$ for capture-avoiding substitution of $t'$ for $x$ in $t$. Substitution is capture-avoiding in the sense that we do not allow scoping of variables to change, and we rename variables introduced by a lambda-abstractions as necessary to ensure that. For example, in the following reduction, we have renamed the lambda-bound $x$ to $z$, in order to avoid having that lambda capture the underlined $x$:

\[
(\lambda y. (\lambda x. (x y))) x \rightarrow (\lambda z. (z x))
\]

As a bit of further terminology, any term of the form $((\lambda x. t) t')$ is called a $\beta$-redex (“reducible expression”), and $[t'/x]t$ is called its contractum.

3.1 Nondeterminism

The reduction relation defined by the rules above is non-deterministic, in the sense that there are terms $t$, $t_1$, and $t_2$, with $t_1$ and $t_2$ distinct, such that $t \rightarrow t_1$ and $t \rightarrow t_2$. Here is an example, where I have underlined the $\beta$-redexes being reduced in each case:

\[
((\lambda x. x) ((\lambda y. y) z)) \rightarrow ((\lambda y. y) z) \\
((\lambda x. x) ((\lambda y. y) z)) \rightarrow ((\lambda x. x) z)
\]

Now it happens that even though full $\beta$-reduction is non-deterministic, it is still confluent: whenever we have $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$, then there exists a $\hat{t}$ such that $t_1 \rightarrow^* \hat{t}$ and $t_2 \rightarrow^* \hat{t}$. That means that however differently we reduce $t$ (to get $t_1$ and $t_2$), we can always get back to a common term $\hat{t}$. We will prove that lambda-calculus is confluent, a few lectures from now.

3.2 Defining reduction order with contexts

Since any given lambda-term can contain many different $\beta$-redexes (giving rise to different reductions of the term, as explained in the previous section), we may define different operational semantics by specifying different orders for reduction of the $\beta$-redexes in a term. One technical device for doing this is using contexts. A context is a term containing a single occurrence of a special variable denoted $\ast$, and called the hole of the context. Often people use $C$ as a meta-variable for contexts. If $C$ is a context, then $C[t]$ is the term obtained by inserting the term $t$ into the context’s hole. More formally, $C[t]$ is obtained by grafting the term $t$ in for $\ast$. Grafting is simply a form of substitution which allows variables in $t$ to be captured by lambda-abstractions in $C$. For example, if $C$ is $\lambda x. \ast$, then $C[x]$ is actually $\lambda x. x$. In contrast, using the capture-avoiding substitution mentioned above, we would have

\[
[x/\ast]\lambda x. \ast = \lambda y. x
\]

To define reduction using a particular order, we use a set of contexts to specify where reductions may take place. For example, for full $\beta$-reduction, the contexts are all possible ones:
\[ C ::= * \mid (C \ t) \mid (t \ C) \mid \lambda x.C \]

Wherever the * is, a reduction is allowed. This is made precise by using the following rule to define (completely) the operational semantics (for whatever set of contexts \( C \) is specified):

\[ C[(\lambda x.t) \ t'] \rightarrow C[[t'/x]t] \]

This rule decomposes a reduced term into context \( C \) and redex. Several other operational semantics can now be defined by specifying different sets of contexts:

- **Left-to-right call-by-value.**

  \[ C ::= * \mid (C \ t) \mid (v \ C) \]

  \[ v ::= \lambda x.t \]

  Here, we also need a restricted form of the context+redex rule, called \( \beta_v \) (for “\( \beta \) value”):

  \[ C[(\lambda x.t) \ v] \rightarrow C[[v/x]t] \]

  This semantics does not allow reduction inside a lambda-abstraction, and it evaluates applications \((t \ t')\) by first evaluating \(t\), and then \(t'\). So the argument is always a value (at least for terms without free variables) when a \( \beta \)-reduction is done with it. This is a deterministic strategy.

- **Right-to-left call-by-value.**

  \[ C ::= * \mid (t \ C) \mid (C \ v) \]

  \[ v ::= \lambda x.t \]

  We again use the \( \beta_v \) rule. This is just like the previous operational semantics, except now we evaluate applications \((t \ t')\) by first evaluating \(t'\) and then \(t\).

- **Normal order (leftmost-outermost).**

  \[ C ::= D \mid \lambda x.C \]

  \[ D ::= * \mid (D \ t) \mid (n \ C) \]

  \[ n ::= x \mid n \ N \]

  \[ N ::= \lambda x.N \mid n \]

  We always reduce the redex that is leftmost and outermost first. This is a deterministic strategy.

- **Call-by-name.**

  \[ C ::= * \mid (C \ t) \]

  We do not reduce inside lambda-abstractions. We do not require an argument to be evaluated before doing a \( \beta \)-reduction with it.