CHAPTER 1

LAMBDA CALCULUS

1.1 Introduction to Lambda Calculus

In this section, we will study the syntax and the sematics of Lambda Calculus, along with some of its properties. Lambda Calculus (denoted λ-calculus) is a mathematical theory of functions, including:

- a formal syntax for defining functions;
- a formal semantics of how functions can be evaluated; and
- a formal theory of function equivalence.

We now define the syntax of λ-calculus. We denote by the set Λ the set of λ-terms. It is the smallest set satisfying the following properties:

- If x is a variable, then $x \in \Lambda$ (there are infinitely many variables);
- If $M \in \Lambda$, then $(\lambda x M) \in \Lambda$ (called *abstraction*);
- If $M, N \in \Lambda$, then $(MN) \in \Lambda$ (called *application*).

This is an inductive definition for Λ. An abstraction represents a function, e.g. $(\lambda x x)$ can be thought of as the Haskell function `\x -> x`. We can define λ-calculus using BNF as well:

$$\begin{aligned} M &::= x \\ &\mid (\lambda x M) \\ &\mid (MM) \end{aligned}$$

Further, we can use inference rules to define it:

$$\frac{\text{if } x \text{ is a variable}^*}{x \in \Lambda} \qquad \frac{M \in \Lambda \text{ and } x \text{ is a variable}^*}{(\lambda x M) \in \Lambda} \qquad \frac{M, N \in \Lambda}{(MN) \in \Lambda}.$$

We typically avoid using brackets for λ-terms (or terms). Moreover, an abstraction $(\lambda x M)$ is denoted $\lambda x.M$. If we have multiple abstractions, we write

$$\lambda x_1 \dots x_k.M \equiv \lambda \vec{x}.M \equiv (\lambda x_1 (\dots (\lambda x_k M) \dots))$$

So, abstraction is right-associative. Now, if we have multiple applications, we write

$$M N_1 \dots N_k \equiv M \vec{N} \equiv (((M N_1) \dots) N_k)$$

So, application is left-associative. The equivalence symbol $\equiv$ denotes syntactic equality.

Free and Bound Variables

The symbol λ binds variables, i.e. it gives rise to local variables. As such, different terms (e.g. $\lambda x.x$ and $\lambda y.y$) are equivalent. Moreover, this gives rise to free variables (not bound by λ) and bound variables (those bound by λ).

We will now define the set of bound variables for a term. It is a function $BV\colon \Lambda \to \mathcal{P}(Var)$, where Var is the set of variables. It is defined inductively:

$$\begin{aligned} BV\ x &= \varnothing \\ BV\ \lambda x.M &= (BV\ M) \cup \{x\} \\ BV\ MN &= (BV\ M) \cup (BV\ N) \end{aligned}$$

We illustrate this with an example. Consider the term $(\lambda y.(\lambda x.zx))y$. Then, we compute its bound variables as follows:

$$\begin{aligned} BV\ (\lambda y.(\lambda x.zx))y &= (BV\ \lambda y.(\lambda x.zx)) \cup (BV\ y) \\ &= ((BV\ \lambda x.zx) \cup \{y\}) \cup \varnothing \\ &= (((BV\ zx) \cup \{x\}) \cup \{y\}) \\ &= ((BV\ z) \cup (BV\ x)) \cup \{x, y\} \\ &= \varnothing \cup \varnothing \cup \{x, y\} \\ &= \{x, y\}. \end{aligned}$$

Next, we define free variables. It too is a function $FV\colon \Lambda \to \mathcal{P}(Var)$, given by:

$$\begin{aligned} FV\ x &= \{x\} \\ FV\ \lambda x.M &= (FV\ M) \setminus \{x\} \\ FV\ MN &= (FV\ M) \cup (FV\ N) \end{aligned}$$

We illustrate this with an example:

$$\begin{aligned} FV\ (\lambda y.(\lambda x.zx))y &= (FV\ \lambda y.(\lambda x.zx)) \cup (FV\ y) \\ &= ((FV\ \lambda x.xz) \setminus \{y\}) \cup \{y\} \\ &= ((FV\ xz \setminus \{x\}) \setminus \{y\}) \cup \{y\} \\ &= (((FV\ x) \cup (FV\ z) \setminus \{x, y\})) \cup \{y\} \\ &= (\{x, z\} \setminus \{x, y\}) \cup \{y\} \\ &= \{y, z\}. \end{aligned}$$

Now, we define subterms of a term. It is a function $Sub\colon \Lambda \to \mathcal{P}(\Lambda)$ defined by:

$$\begin{aligned} Sub\ x &= \{x\} \\ Sub\ \lambda x.M &= (Sub\ M) \cup \{\lambda x.M\} \\ Sub\ MN &= (Sub\ M) \cup (Sub\ N) \cup \{MN\} \end{aligned}$$

We illustrate this with an example:

$$\begin{aligned}
\mathit{Sub}\ (\lambda y.(\lambda x.zx))y &= (\mathit{Sub}\ \lambda y.(\lambda x.zx)) \cup (\mathit{Sub}\ y) \cup \{(\lambda y.(\lambda x.zx))y\} \\
&= (\mathit{Sub}\ \lambda x.zx \cup \{\lambda y.(\lambda x.zx)\}) \cup \{y, (\lambda y.(\lambda x.zx))y\} \\
&= ((\mathit{Sub}\ zx) \cup \{\lambda x.zx\}) \cup \{\lambda y.(\lambda x.zx), y, (\lambda y.(\lambda x.zx))y\} \\
&= ((\mathit{Sub}\ z) \cup (\mathit{Sub}\ x) \cup \{zx\}) \cup \\
&\quad \{\lambda x.zx, \lambda y.(\lambda x.zx), y, (\lambda y.(\lambda x.zx))y\} \\
&= \{z\} \cup \{x\} \cup \{zx, \lambda x.zx, \lambda y.(\lambda x.zx), y, (\lambda y.(\lambda x.zx))y\} \\
&= \{z, x, zx, \lambda x.zx, \lambda y.(\lambda x.zx), y, (\lambda y.(\lambda x.zx))y\}.
\end{aligned}$$

Structural Induction

We will now look at proofs involving structural induction for λ-calculus.

Proposition 1.1.1. *A term in Λ has balanced parentheses.*

Proof. We show that using structural induction on a term:

- In the base case, the term is x, for some variable x. Since there are no parentheses here, the result follows trivially.
- Now, assume that the terms M and N have balanced parentheses. Then, the term (MN) must also have balanced parentheses.
- Finally, assume that the term M has balanced parentheses. Then, the term $(\lambda x M)$ must also have balanced parentheses.

So, the result follows by structural induction. □

Note that we have 2 inductive cases here- this is because there are 3 production rules. We now look at a more complicated example.

Proposition 1.1.2. *Let M be a term. Then, $\mathit{FV}\ M \subseteq \mathit{Sub}\ M$.*

Proof. We prove this by structural induction on M.

- In the base case, we have $M = x$, for some variable x. Then,
$$\mathit{FV}\ M = \{x\} \subseteq \{x\} = \mathit{Sub}\ M.$$
- Now, assume that $M = N_1 N_2$, for terms N_1 and N_2 that satisfy $\mathit{FV}\ N_1 \subseteq \mathit{Sub}\ N_1$ and $\mathit{FV}\ N_2 \subseteq \mathit{Sub}\ N_2$. In that case,
$$\begin{aligned}
\mathit{FV}\ M &= (\mathit{FV}\ N_1) \cup (\mathit{FV}\ N_2) \\
&\subseteq (\mathit{Sub}\ N_1) \cup (\mathit{Sub}\ N_2) \\
&\subseteq (\mathit{Sub}\ N_1) \cup (\mathit{Sub}\ N_2) \cup \{M\} = \mathit{Sub}\ M.
\end{aligned}$$
- Finally, assume that $M = \lambda x.N$, for some term N satisfying $\mathit{FV}\ N \subseteq \mathit{Sub}\ N$. Then,
$$\begin{aligned}
\mathit{FV}\ M &= (\mathit{FV}\ N) \setminus \{x\} \\
&\subseteq \mathit{FV}\ N \\
&\subseteq \mathit{Sub}\ N \\
&\subseteq (\mathit{Sub}\ N) \cup \{M\} = \mathit{Sub}\ M.
\end{aligned}$$

So, the result follows from structural induction. □

Contexts

A context is a term containing a *hole*, which is represented by $[]$. This hole can be filled by a term, to make the context a term. It is defined as follows:

$$\begin{aligned} C[] &::= x \\ &\mid [] \\ &\mid (\lambda x C[]) \\ &\mid (C[] C[]) \end{aligned}$$

The variable has no hole; the empty context is $[]$. The context $C[]C'[]$ has two holes. We will now illustrate how to fill a context with a term. So, consider the context $C[] = ((\lambda x.[]x)M)$. Then, $C[\lambda y.y] = ((\lambda x.(\lambda y.y)x)M)$.

We can fill contexts with a hole given one term.

Proposition 1.1.3. *Let $C[]$ be a context with one hole and M a term. Then, $C[M]$ is a term.*

Proof. We show this using structural induction on the context $C[]$.

- First, let $C[] = []$. Since M is a term, we find that $C[M] = M$ is a term.
- Now, let $C[] = \lambda x.C'[]$, where $C'[]$ is a context where $C'[M]$ is a term. Then, $C[M] = \lambda x.C'[M]$ must be a term.

So, the result follows from structural induction. □

In general, we can fill a context with n holes given n terms- this follows from the result above and mathematical induction.

1.2 Reduction

In this section, we will consider how we can evaluate λ-terms, using the β-rule in a cumbersome manner, and later using β-rule with α-equivalence to make it compact and efficient.

β-reduction

The key definition for expressing computation in λ-calculus is the β rule. It states:

$$(\lambda x.M)N \to_\beta M[x := N]$$

The notation $M[x := N]$ is substitution- every occurrence of x in M is replaced by N. For example,

$$(\lambda x.x + 1)2 \to_\beta 2 + 1.$$

We could further reduce this to 3 by extending mathematical operations, like the language Expr. The β rule expresses how we would like to pass parameters into a function.

Substitution is the main aspect of β-reduction. However, naive substitution is not necessarily what we want in general. For instance, if we have the term $(\lambda x.\lambda y.yx)y$, then we can β-reduce it to $\lambda y.yy$. However, this is not what we would like- the first y is meant to be bound by the local y, but the second one is not- it is the value we have just substituted. This would not have happened if we have the term $(\lambda x.\lambda y.yx)z$, in which case we get $\lambda y.yz$. In particular, this happens in the term $(\lambda x.\lambda y.yx)y$ because y is both a free and a bound variable in the term.

Hence, we need to define substitution in a more careful manner. This is done as follows:

1. $x[x := N] \equiv N$;
2. $y[x := N] \equiv y$ if x and y are distinct;
3. $(\lambda x.M)[x := N] \equiv \lambda x.M$;
4. $(\lambda y.M)[x := N] \equiv \lambda y.M[x := N]$ if x is not a free variable in M or y is not a free variable in N;
5. $(\lambda y.M)[x := N] \equiv (\lambda z.M[y := z])[x := N]$ if x is a free variable in M and y a free variable in N;
6. $(M_1M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$.

Now, consider the term $(\lambda x.\lambda y.yx)y$. This β-reduces to

$$(\lambda y.yx)[x := y].$$

The variable x is free in $\lambda y.yx$ and the variable y is free in y. Hence, we apply rule 5 to get

$$(\lambda y.yx)[x := y] \equiv (\lambda z.yx[y := z])[x := y].$$

We have $yx[y := z] \equiv zx$ by rules 6 and 1, so

$$(\lambda z.yx[y := z])[x := y] \equiv (\lambda z.zx)[x := y].$$

Now, although x is free in $\lambda z.zx$, z is not free in y, so we finally get

$$\lambda z.zy$$

In programming language terms, this is similar to renaming a variable in a local scope so that it does not mask a variable in an outer scope.

α-equivalence

We will now give another, simpler, definition for substitution. This works by assuming that bound variables have been renamed so that they are different from any free variables. This ensures that variables cannot be captured. This is called Barendregt or variable convention. To define substitution, we define α-equivalence.

Definition 1.2.1. Let M and M' be terms. We say that M' is produced by M by a *change of bound variables* if $M \equiv C[\lambda x.N]$ and $M' \equiv C[\lambda y.N[x := y]]$, where y does not occur in N, and $C[]$ is a context with one hole.

Note that filling a hole in a context is different to substitution- we do not care about variable capture when doing so, unlike in substitution. For instance, if $C[] \equiv \lambda x.x[]$, then $C[x] \equiv \lambda x.xx$, even though substitution would have resulted in a variable change.

Definition 1.2.2. Let M and N be terms. We say that M is *α-equivalent* to N if N is produced from M by a series of changes of bound variable.

For instance, $\lambda x.xy \equiv_\alpha \lambda z.zy$ since we can let $C[] = []y$, and then $C[\lambda x.x] \equiv \lambda x.xy$ and $C[\lambda z.x[x := z]] \equiv \lambda z.zy$. However, $\lambda x.xy$ is not α-equivalent to $\lambda x.xx$. We consider α-equivalent terms to be the same as each other.

From now, we will assume that the terms obey variable convention, i.e. all bound variables are different from each other and from all free variables. If this is not the case, we can achieve this using α-equivalence. With this convention, the definition of substitution simplifies to the following:

1. $x[x := N] \equiv N$;
2. $y[x := N] \equiv y$ if y and x are distinct;
3. $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$;
4. $(M_1M_2)[x := N] \equiv M_1[x := N]M_2[x := N]$.

We now consider how substitution interacts with two variables.

Lemma 1.2.3 (Substitution Lemma)**.** *Let M and N be terms, and x, y distinct variables with x not free in L. Then,*

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Proof. We prove this by structural induction on M:

- Let $M = z$, where z is a variable distinct to both x and y. In that case,

$$\begin{aligned} M[x := N][y := L] &\equiv z[x := N][y := L] \\ &\equiv z[y := L] \equiv z, \end{aligned}$$

  and

$$\begin{aligned} M[y := L][x := N[y := L]] &\equiv z[y := L][x := N[y := L]] \\ &\equiv z[x := N[y := L]] \equiv z. \end{aligned}$$

  So, the result holds in this case.

- Now, let $M = x$. Then,

$$\begin{aligned} M[x := N][y := L] &\equiv x[x := N][y := L] \\ &\equiv N[y := L], \end{aligned}$$

  and, since x is not free in L,

$$\begin{aligned} M[y := L][x := N[y := L]] &\equiv x[y := L][x := N[y := L]] \\ &\equiv x[x := N[y := L]] \equiv N[y := L]. \end{aligned}$$

  So, the result holds in this case.

- Next, let $M = y$. Then,

$$M[x := N][y := L] \equiv y[x := N][y := L] \equiv y[y := L] \equiv L,$$

  and, since x is not free in L,

$$M[y := L][x := N[y := L]] \equiv L[x := N[y := L]] \equiv L.$$

  So, the result holds in this case.

- Next, let $M = \lambda z.M'$, where z is distinct from x and y and

$$M'[x := N][y := L] \equiv M'[y := L][x := N[y := L]].$$

  Then,

$$\begin{aligned} M[x := N][y := L] &\equiv (\lambda z.M')[x := N][y := L] \\ &\equiv (\lambda z.M'[x := N])[y := L] \\ &\equiv (\lambda z.M'[x := N][y := L]) \\ &\equiv (\lambda z.M'[y := L][x := N[y := L]]) \\ &\equiv (\lambda z.M'[y := L])[x := N[y := L]] \\ &\equiv (\lambda z.M')[y := L][x := N[y := L]] \\ &\equiv M[y := L][x := N[y := L]]. \end{aligned}$$

  So, the result holds in this case.

- Finally, let $M = M_1 M_2$, where

$$\begin{aligned} M_1[x := N][y := L] &\equiv M_1[y := L][x := N[y := L]] \\ M_2[x := N][y := L] &\equiv M_2[y := L][x := N[y := L]]. \end{aligned}$$

  Then,

$$\begin{aligned} M[x := N][y := L] &\equiv (M_1 M_2)[x := N][y := L] \\ &\equiv (M_1[x := N][y := L])(M_2[x := N][y := L]) \\ &\equiv (M_1[y := L][x := N[y := L]]) \\ &\quad (M_2[y := L][x := N[y := L]]) \\ &\equiv (M_1 M_2)[y := L][x := N[y := L]] \\ &\equiv M[y := L][x := N[y := L]]. \end{aligned}$$

  So, the result holds in this case.

Hence, the result follows from induction. □

We can now complete the definition on evaluating λ-terms. First, we define one-step β-reduction $\to_\beta$, given by the following inference rules:

$$\frac{}{(\lambda x.M)N \to_\beta M[x := N]} \qquad \frac{M \to_\beta N}{MZ \to_\beta NZ}$$

$$\frac{M \to_\beta N}{ZM \to_\beta ZN} \qquad \frac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N}$$

We have seen the first definition before. The other three allow us to apply the β-rule in larger terms.

We can define the β-reduction, denoted by $\twoheadrightarrow_\beta$ that is the reflexive and transitive closure of one-step β-reduction. This is defined by the following inference rules:

$$\frac{}{M \twoheadrightarrow_\beta M} \qquad \frac{M \to_\beta N}{M \twoheadrightarrow_\beta N} \qquad \frac{M \twoheadrightarrow_\beta N \quad N \twoheadrightarrow_\beta L}{M \twoheadrightarrow_\beta L}.$$

It turns out that this definition also extends to contexts.

Proposition 1.2.4. *Let $C[]$ be a context with one hole. If $M \to_\beta N$, then $C[M] \to_\beta C[N]$.*

Proof. We prove this by structural induction on the context $C[]$:

- First, let $C[] = []$. In that case, if $M \to_\beta N$, then $C[M] = M \to_\beta N = C[N]$.
- Now, let $C[] = \lambda x.C'[]$, where $C'[]$ is another context such that $C'[M] \to_\beta C'[N]$. In that case, applying rule 4 of $\to_\beta$, we find that $C[M] = \lambda x.C'[M] \to \lambda x.C'[N] = C[N]$.

So, the result follows from induction. □

Proposition 1.2.5. *Let $C[]$ be a context with one hole. If $M \twoheadrightarrow_\beta N$, then $C[M] \twoheadrightarrow_\beta C[N]$.*

Proof. We prove this by structural induction on $\twoheadrightarrow_\beta$:

- If $M \equiv N$, then $C[M] \equiv C[N]$. Hence, $C[M] \twoheadrightarrow_\beta C[N]$.
- Instead, if $M \to_\beta N$, then by the result above, we have $C[M] \to_\beta C[N]$. Hence, $C[M] \twoheadrightarrow_\beta C[N]$.
- Otherwise, we have $M \twoheadrightarrow_\beta L$ and $L \twoheadrightarrow_\beta N$, with $C[M] \twoheadrightarrow_\beta C[L]$ and $C[L] \twoheadrightarrow_\beta C[N]$. Hence, we can apply the third inference rule of $\twoheadrightarrow_\beta$ to conclude that $C[M] \twoheadrightarrow_\beta C[N]$.

So, the result follows from structural induction. □