CHAPTER 1

LAMBDA CALCULUS

1.1 Introduction to Lambda Calculus

In this section, we will study the syntax and the semantics 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 first define the syntax of λ-calculus. We denote by the set Λ the set of all λ-terms. It is the *smallest* set satisfying the following properties:

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

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

We can define λ-calculus using BNF as well:

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

Note that in the last line, although we have the application (MM), the two M values need not be the same. 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. Moreover, an abstraction $(\lambda x M)$ is denoted $\lambda x.M$. If we have multiple abstractions, we write

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

The symbol $\equiv$ denotes syntactic equivalence. Note that abstraction is *right-associative*. Now, if we have multiple applications, we write

$$MN_1 \dots N_k \equiv M\vec{N} \equiv (((MN_1) \dots) N_k)$$

Note that application is *left-associative*.

Free and Bound Variables

The symbol λ is a *variable binder*, i.e. it gives rise to local variables. As such, different terms are equivalent, e.g. $\lambda x.x$ and $\lambda y.y$. 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 formally. It is a function $BV\colon \Lambda \to \mathcal{P}(\mathit{Var})$, where Var is the set of variables, and $\mathcal{P}(\mathit{Var})$ is the set of all subsets of variables. It is defined recursively:

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

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 $FV\colon \Lambda \to \mathcal{P}(\mathit{Var})$, 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:

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

We now define *subterms* of a term. It is a function $\mathit{Sub}\colon \Lambda \to \mathcal{P}(\Lambda)$ defined by:

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

We illustrate this with an example:

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

Structural Induction

We will now look at proofs involving structural induction for λ-calculus. We start wth an easy example.

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 (λxM) 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, $FV\ M \subseteq Sub\ M$.*

Proof. We prove this by structural induction on M.

- In the base case, we have $M = x$, for some variable x. Then,

$$FV\ M = \{x\} \subseteq \{x\} = Sub\ M.$$

- Now, assume that $M = N_1 N_2$, for terms N_1 and N_2. By the inductive hypothesis, we know that $FV\ N_1 \subseteq Sub\ N_1$ and $FV\ N_2 \subseteq Sub\ N_2$. Hence,

$$\begin{aligned} FV\ M &= (FV\ N_1) \cup (FV\ N_2) \\ &\subseteq (Sub\ N_1) \cup (Sub\ N_2) \\ &\subseteq (Sub\ N_1) \cup (Sub\ N_2) \cup \{M\} = Sub\ M. \end{aligned}$$

- Finally, assume that $M = \lambda x.N$, for some term N. By the inductive hypothesis, $FV\ N \subseteq Sub\ N$. Then,

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

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 have replaced the hole [] with the given term.

In general, 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. □

Note that we only considered 2 of the cases from the definition of a context- a variable x has no holes, while the context $(C[]C[])$ has two.

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

$$(\lambda x.x + 1)2 \to_\beta (x + 1)[x := 2] \equiv 2 + 1.$$

We could further reduce this to 3 by extending mathematical operations, like in 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 $\lambda y.yz$. So, 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, and z is a new variable;
6. $(M_1M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$.

In this definition, we are renaming bound variables if they are also free (at step 5). This is to avoid *variable capture*.

We illustrate the definition with an example. So, 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 apply step 4 and get

$$\lambda z.zy$$

In programming language terms, this is similar to renaming a local variable so that it does not override a global variable.

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 convention* or *variable convention*. To do so, we need 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.

We use the symbol $\equiv_\alpha$ to denote α equivalence.

We illustrate α-equvalence with an example. We claim that $\lambda x.xy \equiv_\alpha \lambda z.zy$. To see this, let $C[] = []y$, and then

$$C[\lambda x.x] \equiv \lambda x.xy \qquad \text{and} \qquad C[\lambda z.x[x := z]] \equiv C[\lambda z.z] \equiv \lambda z.zy.$$

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 $M = M_1M_2$, 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,

$$\begin{aligned} M[x := N][y := L] &\equiv (M_1M_2)[x := N][y := L] \\ &\equiv (M_1[x := N][y := L])(M_2[x := N][y := L]) \\ &\equiv (M_1[y := L][x := N[y := L]]) \\ &\quad (M_2[y := L][x := N[y := L]]) \\ &\equiv (M_1M_2)[y := L][x := N[y := L]] \\ &\equiv M[y := L][x := N[y := L]]. \end{aligned}$$

  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* $\to_\beta$ 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. So, the β-reduction allows for zero or more β-step reductions. 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. By the inductive hypothesis, we know 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 $L \twoheadrightarrow_\beta N$. By the inductive hypothesis, we find that $C[M] \twoheadrightarrow_\beta C[L]$ 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. □

This property is called *computability*.

1.3 Theory of Equality

In this section, we will define the *theory of equality* between λ-terms. β-reduction is part of the equality, e.g. we say

$$(\lambda x.x+1)1 = 2.$$

We assume that mathematical computations are part of the λ-calculus. Unlike β-reduction, we also have

$$(\lambda x.x+1)2 = (\lambda x.x+2)1$$

We denote the λ formal theory by

$$\lambda \vdash M = N,$$

which means that in λ-calculus, we can derive $M = N$. This is a collection of axioms and inference rules. The theory of equality must satisfy the following:

- an application term must be equal to the result obtained by applying the function part of the term to the argument.
- equality must be an equivalence relation.
- equal terms should be equal in any context.

We will now go through the rules of λ-theory. The first axiom is β-reduction:

$$\frac{}{(\lambda x.M)N = M[x := N]}$$

This is the β-rule. Now, we have *reflexivity* axiom, and the *symmetry* and the *transitivity* inference rules to make equality an equivalence relation.

$$\frac{}{M = M} \qquad \frac{M = N}{N = M} \qquad \frac{M = N \quad N = L}{M = L}.$$

The remaining rules cover extend equality to applications and abstractions.

$$\frac{M = N}{MZ = NZ} \qquad \frac{M = N}{ZM = ZN} \qquad \frac{M = N}{\lambda x.M = \lambda x.N}$$

These are the ξ-rules, and ensure that equal terms are equal in every context. Note that the first two rules are required since function application is not commutative (i.e. $MZ \neq ZM$).

If $M = N$ can be derived from the axioms and the inference rules of λ-theory, then we write $\lambda \vdash M = N$. The *true statements* derived from applying axioms and inference rules of λ, are called *theorems*. In this case, $M = N$ is a *theorem*, and we say that M and N are *convertible*.

Note that if $M \equiv N$, then $M = N$, it is not necessarily true the other way around. For instance, $(\lambda x.x)y = y$, but they are not syntactically equivalent.

We will now show that equal terms are equal in any context.

Theorem 1.3.1. *Let $C[]$ be a context and M, N be terms. If $\lambda \vdash M = N$, then $\lambda \vdash C[M] = C[N]$.*

Proof. We show this by structural induction:

- If $C[] = x$, then
$$x[M] \equiv x \equiv x[N].$$
- If $C[] = []$, then
$$C[M] \equiv M = N \equiv C[N].$$
- Now, let $C[] = \lambda x.C'[]$, where $C'[]$ is a context satisfying $C'[M] = C'[N]$. Hence,
$$C[M] \equiv \lambda x.C'[M] = \lambda x.C'[N] \equiv C[N],$$
applying the ξ-rule
$$\frac{C'[M] = C'[N]}{\lambda x.C'[M] = \lambda x.C'[N]}$$
- Finally, let $C[] = C_1[]C_2[]$, where $C_1[]$ and $C_2[]$ are contexts satisfying $C_1[M] = C_1[N]$ and $C_2[M] = C_2[N]$. Then,
$$C[M] \equiv C_1[M]C_2[M] = C_1[N]C_2[M] = C_1[N]C_2[N] \equiv C[N].$$
To do so, we apply the following ξ-rules:
$$\frac{M = N}{C_1[M]C_2[M] = C_1[N]C_2[M]} \qquad \frac{M = N}{C_1[N]C_2[M] = C_1[N]C_2[N]}$$
and
$$\frac{C_1[M]C_2[M] = C_1[N]C_2[M] \qquad C_1[N]C_2[M] = C_1[N]C_2[N]}{C_1[M]C_2[M] = C_1[N]C_2[N]}.$$

□

This property is called *referential transparency.* This is an advantage of functional languages that states that the value of the expression is the only thing that matters; not the way it is computed. This property does not hold if expressions have side effects (e.g. changing state or performing input/output).

We now establish the relationship between equality and substitution.

Theorem 1.3.2. *Let M, M', N, N' be terms.*

- *If $M = M'$ then $M[x := N] = M'[x := N]$.*
- *If $N = N'$, then $M[x := N] = M[x := N']$.*
- *If $M = M'$ and $N = N'$, then $M[x := N] = M'[x := N']$.*

Proof.

- By definition, we have
$$M[x := N] = (\lambda x.M)N.$$
Since $M = M'$, we have $\lambda x.M = \lambda x.M'$ by the ξ-rule. Moreover, this implies that $(\lambda x.M)N = (\lambda x.M')N$ by the ξ-rule again. Hence,
$$M[x := N] = (\lambda x.M)N = (\lambda x.M')N = M'[x := N].$$
So, by transitivity, it follows that $M[x := N] = M'[x := N]$.

- We show this by structural induction on substitution.
  - First, let $M = x$. Then,
  $$x[x := N] \equiv N = N' \equiv x[x := N'].$$
  - Now, let $M = y$, where y is distinct from x. Then,
  $$y[x := N] \equiv y \equiv y[x := N'].$$
  - Next, let $M = (\lambda x.M')$, where $M'[x := N] = M'[x := N']$. Then, we know that
  $$\begin{aligned} M[x := N] &\equiv (\lambda y.M')[x := N] \\ &\equiv \lambda y.(M'[x := N]) \\ &= \lambda y.(M'[x := N']) \\ &\equiv (\lambda y.M')[x := N'] \equiv M[x := N']. \end{aligned}$$
  - Finally, let $M = M_1M_2$, where $M_1[x := N] = M_1[x := N']$ and $M_2[x := N] = M_2[x := N']$. Then, we know that
  $$\begin{aligned} M[x := N] &\equiv (M_1M_2)[x := N] \\ &\equiv (M_1[x := N])(M_2[x := N]) \\ &= (M_1[x := N'])(M_2[x := N]) \\ &= (M_1[x := N'])(M_2[x := N']) \\ &\equiv (M_1M_2)[x := N'] \equiv M[x := N']. \end{aligned}$$

  So, the result follows from induction.
- Since $M[x := N] = M'[x := N]$ and $M'[x := N] = M'[x := N']$, this follows from transitivity.

□

We will now consider fixed points in λ-calculus.

Theorem 1.3.3 (Fixed Point Theorem)**.** *Let $F \in \Lambda$ be a term. Then, there exists a term $X \in \Lambda$ such that $FX = X$.*

Proof. Let $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then,
$$\begin{aligned} X &\equiv WW \\ &\equiv (\lambda x.F(xx))W \\ &= F(WW) && (\beta) \\ &\equiv FX. \end{aligned}$$

□

We say that X is a *fixed point* of X. The identity function $I \equiv \lambda x.x$ fixes every term, i.e. $IM \equiv (\lambda x.x)M = M$.

We will now use the algorithm given in the theorem to find a fixed point for the term $F = \lambda xy.xy$. We define

$$W \equiv \lambda x.F(xx) \equiv \lambda x.((\lambda xy.xy)(xx)) = \lambda x.\lambda y.(xx)y \equiv \lambda xy.(xx)y$$

So, the fixed point is

$$X \equiv WW \equiv (\lambda xy.(xx)y)(\lambda xy.(xx)y)$$

We can directly verify if X is a fixed point of F:

$$\begin{aligned} FX &\equiv (\lambda xy.xy)((\lambda xy.(xx)y)(\lambda xy.(xx)y)) \\ &= \lambda y.((\lambda xy.(xx)y)(\lambda xy.(xx)y))y \\ &= (\lambda xy.(xx)y)(\lambda xy.(xx)y) \\ &\equiv X \end{aligned}$$

In the fixed point theorem, we have self-application, i.e. subterms of the form xx. This is not typically seen in programming languages, but has huge consequences in λ-calculus.

Now, we will see an application of the fixed point theorem-defining recursive functions. The normal definition of the factorial function is the following:

```
fac(n) = if n == 0 then 1 else n*fac(n-1)
```

Since λ-terms do not have the concept of recursion, we will have to implement this algorithm using the fixed point theorem. To do so, define the term

$$F = \lambda f.\lambda n.\mathit{if}(\mathit{eq}(n)(0))(1)(\mathit{mul}(n)(f(n-1))).$$

In this term, assume that the λ-calculus has the terms *if*, *eq* and *mul* with their expected meaning. Then, let *fac* be a fixed point of F. We can evaluate factorial of 1 using this term:

$$\begin{aligned} \mathit{fac}(1) &= F(\mathit{fac})(1) \\ &\equiv (\lambda f.\lambda n.\mathit{if}(\mathit{eq}\ n\ 0)\ 1\ (\mathit{mul}\ n\ (f(n-1))))(\mathit{fac})(1) \\ &= \mathit{if}(\mathit{eq}\ 1\ 0)1(\mathit{mul}\ 1\ (\mathit{fac}(0))) \\ &= \mathit{mul}\ 1\ \mathit{fac}(0) \\ &= \mathit{mul}\ 1\ F(\mathit{fac})(0) \\ &\equiv \mathit{mul}\ 1\ (\lambda f.\lambda n.\mathit{if}(\mathit{eq}\ n\ 0)\ 1\ (\mathit{mul}\ n(f(n-1))))(\mathit{fac})(0) \\ &= \mathit{mul}\ 1\ \mathit{if}(\mathit{eq}\ 0\ 0)1(\mathit{mul}\ 1(\mathit{fac}(-1))) \\ &= \mathit{mul}\ 1\ 1 = 1. \end{aligned}$$

Clearly, this formula can be used to evaluate higher factorials as well.

Extensionality

The concept of equality we have defined is the *convertibility* relationship. It says that two terms are equal if they encode the same algorithm. However, there are some terms we would naturally consider to be equal, but we cannot prove that they are equal in λ-theory. For instance, $\lambda x.Mx = M$ cannot be shown in λ. In the extensionality relationship, two terms are considered equal if they give equal results to equal arguments.

There are two ways of extending λ-theory so that $\lambda x.Mx$:

- We can extend the λ-theory to derive $\lambda x.Mx = M$. We can directly add it as a new axiom, called the η-axiom- it states that $\lambda x.Mx = M$ if x is not free in M. This is the $\lambda\eta$ theory.
- Another way of doing this is using the *ext* rule:

  $$\frac{Mx = Nx}{M = N}$$

  if x is not free in M and N. This is the $\lambda + ext$ theory.

We will show that the two theories are equivalent.

Theorem 1.3.4. *$\lambda\eta$ and $\lambda + ext$ are equivalent.*

Proof. Since $\lambda\eta$ and $\lambda + ext$ both extend λ, it suffices to show that their extensions are equivalent.

We first show that $\lambda\eta \vdash Mx = Nx \implies M = N$ if x is not free in M and N. So, let M, N be terms and x a variable not free in M and N such that $Mx = Nx$. This implies that $\lambda x.Mx = \lambda x.Nx$ by (ξ). Hence, by (η), we find that

$$M = \lambda x.Mx = \lambda x.Nx = N.$$

So, $\lambda\eta \vdash \lambda + ext$.

Now, we show that $\lambda + ext \vdash \lambda x.Mx = M$ if x is not free in M. So, let M be a term and let x be a variable not free in M. By (β), we know that $(\lambda x.Mx)x = Mx$. Hence, by *ext*, it follows that $\lambda x.Mx = M$. So, $\lambda + ext \vdash \lambda\eta$. ☐

Consistency

For a theory to be useful, not every equation can hold- there must be theorems (satisfied equations) and non-theorems (equations not satisfied).

Definition 1.3.5. An *equation* is a formula of the form $M = N$, for terms M and N. It is *closed* if it has no free variables.

Definition 1.3.6. Let $\mathcal{T}$ be a theory with equations as formulae. We say that $\mathcal{T}$ is *consistent*, denoted by $\text{Con}(\mathcal{T})$, if it does not prove every closed equation. If $\mathcal{T}$ is a set of equations, then $\lambda + \mathcal{T}$ is formed by adding the equations of $\mathcal{T}$ as axioms to λ, denoted bt $\text{Con}(\lambda + \mathcal{T})$.

The theories λ and $\lambda\eta$ are consistent. It is quite easily possible to lose consistency, e.g. by just adding a single equation. For instance, define

$$\begin{aligned} S &\equiv \lambda xyz.xz(yz) \\ K &\equiv \lambda xy.x \\ I &\equiv \lambda x.x \end{aligned}$$

If we add the equation $S = K$ to λ or $\lambda\eta$, we get an inconsistent theory. To show this, let D be an arbitrary term. We note that

$$\begin{aligned} SMNO &\equiv MO(NO) \\ KMN &\equiv M \\ IM &\equiv M \end{aligned}$$

for all terms M, N, O. So,

$$S = K \implies SABC = KABC \implies AC(BC) = AC$$

for all terms A, B, C. Now, if $A = C = I$, then

$$AC(BC) = AC \implies BI = I.$$

Finally, if $B = KD$, then

$$BI = I \implies KDI = I \implies D = I.$$

Hence, we have shown that any arbitrary term D is equal to the identity term I. So, the theory is inconsistent.

Definition 1.3.7. We say that the terms M and N are *incompatible*, denoted $M\#N$, if $\neg\,\mathrm{Con}(M = N)$, i.e. $\lambda + M = N$ is not consistent.

Note that this is different to $M = N$ not being derivable- this means that adding $M = N$ makes the theory inconsistent. From the example we considered above, S and K are incompatible.

We will now show that the terms xx and xy are incompatible. So, assume that $xx = xy$. In that case,

$$\begin{aligned} \lambda xy.xx &= \lambda xy.xy \\ (\lambda xy.xx)MN &= (\lambda xy.xx)MN \\ MM &= MN \end{aligned}$$

for any terms M and N. Now, if $M = I$, then we find that $N = I$, so this theory is also inconsistent.

Next, we show that $x(yz)$ and $(xy)z$ are inconsistent. To see this, assume $x(yz) = (xy)z$. Then,

$$\begin{aligned} \lambda xyz.x(yz) &= \lambda xyz.(xy)z \\ (\lambda xyz.x(yz))MNO &= (\lambda xyz.(xy)z)MNO \\ M(NO) &= (MN)O \end{aligned}$$

for any terms M, N and O. Now, let $M = \lambda xy.y$. Then,

$$P = M(NO)P = (MN)OP = OP.$$

So, if $P = N = I$ and O arbitrary, we find that $O = I$. So, this theory is inconsistent.

Normal Forms

We will now look at normal forms.

Definition 1.3.8. Let M be a term.

- We say that M is a *β-normal form*, denoted β-nf. or nf, if M has no subterms of the form $(\lambda x.R)S$.
- We say that M *has* a *β-normal form* if there exists an $N = M$ such that N is β-nf.
- We say that M is a $\beta\eta$-normal form if M is a β-nf. with no subterms of the form $\lambda x.Rx$ where x is free in R.
- We say that M *has* a $\beta\eta$-normal form if there exists an $N = M$ such that N is $\beta\eta$-nf.

The basic idea of λ-calculus as a programming language is that computation consists of applying the β-rule from left to right and converting $(\lambda x.M)N$ to $M[x := N]$ until we reach a β-nf, which is the result. We show this with some examples:

- The term $\lambda x.x$ is a normal form;
- The term $(\lambda xy.x)(\lambda x.x)$ has a normal form- $\lambda yx.x$;
- The term $(\lambda x.xx)(\lambda x.xx)$ does not have a normal form since it β-reduces to itself.

As we just saw, it might be that a term does not have a normal form.

We will now look at some properties for normal forms.

Proposition 1.3.9. *Let M be a term.*

- *M has a β-nf if and only if M has a $\beta\eta$-nf;*
- *If M and N are distinct β-nfs, then $M = N$ is not a theorem of λ (and similarly for $\lambda\eta$ with $\beta\eta$-nfs);*
- *If M and N are distinct $\beta\eta$-nfs, then $M \# N$.*

The final equation states that any two distinct $\beta\eta$-nfs cannot be set equal without making the theory inconsistent. Hence, $\beta\eta$ is the weakest consistent form of equality. Note that this does not holds for just β-nfs: y and $\lambda x.yx$ are two distinct β-nfs, but do not lead to inconsistency (since they share the $\beta\eta$-nf).

$\beta\eta$ normal forms having the equal-or-inconsistent property is the property of *completeness*, which is given below.

Proposition 1.3.10. *Let M and N be λ-terms. Then, either $\lambda\eta \vdash M = N$ or $\lambda\eta + (M = N)$ is inconsistent.*

1.4 Reduction and Normal Forms

We defined one-step β-reduction by the rule

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

The transitive and reflexive closure of one-step β-reduction gives the actual β-reduction, denoted $\twoheadrightarrow_\beta$.

Intuitively, the aim of reduction is to simplify an expression as much as possible. We saw in the previous chapter that normal forms (nfs) are the simplest form of terms; they cannot be β-reduced any further. We also saw that not every term has a nf.

Moreover, because β-reduction can be applied anywhere within a term, we might have a choice when applying reduction. For instance, consider the definition of the factorial function we saw above:

$$\begin{aligned} F &= \lambda f.\lambda n.\texttt{if}(\texttt{eq}(n)(0))(1)(n \cdot f(n-1)) \\ \texttt{fac} &= \texttt{fix}(F). \end{aligned}$$

Then, if we wanted to compute $\texttt{fac}(0)$, the following is a possible sequence of reduction:

$$\texttt{fac}(0) = F(\texttt{fac})(0) = F(F(\texttt{fac}))(0) = \dots$$

Clearly, if we continued expanding like this, we will not get to the normal form. So, not every reduction sequence leads to its nf. However, we have seen that there *is* a reduction sequence that does lead to the nf. So, we want to answer whether, and how, we can reach the nf if it exists. Note that there are some terms where each reduction sequence leads to the nf, e.g. $(\lambda xy.x+y)(1+2)(3+4)$.

In this section, we aim to answer the following 2 questions:

- if a term has a nf, is it unique? and
- does every term have a reduction sequence that leads to its nf?

The Church-Rosser theorem helps us answer the first question.

Theorem 1.4.1 (Church-Rosser). *If $L \twoheadrightarrow_\beta M$ and $L \twoheadrightarrow_\beta N$, then there exists a term Z such that $M \twoheadrightarrow_\beta Z$ and $N \twoheadrightarrow_\beta Z$.*

This theorem states that if L can be reduced to two different values, then these two values are in the middle of the computation, and we can reduce both of these values to some term Z. We can rewrite this result in terms of β-equality.

Corollary 1.4.2. *If $M =_\beta N$[1], then there exists a term Z such that $M \twoheadrightarrow Z$ and $N \twoheadrightarrow Z$.*

Now, we can answer the questions.

Corollary 1.4.3. *Let M be a term.*

1. *If M has N as β-nf, then $M \twoheadrightarrow_\beta N$.*
2. *If M has β-nfs N_1 and N_2, then $N_1 \equiv N_2$.*

[1]We use the equality symbol $=_\beta$ to denote equality arising from β-reduction $\twoheadrightarrow_\beta$.

Proof.

1. We know that $M =_\beta N$. So, there exists a term Z such that $M \twoheadrightarrow_\beta Z$ and $N \twoheadrightarrow_\beta Z$. Since N is a β-nf, we find that $N \equiv Z$. Hence, $M \twoheadrightarrow_\beta N$.

2. We know that $M =_\beta N_1$ and $M =_\beta N_2$. This implies that $N_1 =_\beta N_2$. Hence, there exists a term Z such that $N_1 \twoheadrightarrow_\beta Z$ and $N_2 \twoheadrightarrow_\beta Z$. Since N_1 and N_2 are nfs, we find that $N_1 \equiv Z$ and $N_2 \equiv Z$. So, $N_1 \equiv N_2$.

□

The first result tells us that there is a reduction sequence that reduces M to its normal form. We are yet to find how this happens; we will come back to that question. The second result tells us that the β-nf is unique.

We now prove the Church-Rosser Theorem. To do so, we will define some new concepts, starting with the diamond relation.

Definition 1.4.4. Let $\rhd$ be a binary relation on λ-terms. We say that the relation $\rhd$ satisfies the *diamond property*, denoted $\rhd \vDash \diamond$, if for terms M, M_1, M_2,

$$M \rhd M_1, M \rhd M_2 \implies \exists M_3.(M_1 \rhd M_3, M_2 \rhd M_3).$$

We will later set $\rhd$ to be the reduction relation, which will allow us to conclude the Church-Rosser. Note that we can apply this to the transitive closure of the relation as well.

Proposition 1.4.5. *Let $\rhd$ be a binary relation on λ-terms such that $\rhd \vDash \diamond$. Then, the transitive closure $\rhd^* \vDash \diamond$ as well.*

We will now connect the diamond to reduction types (e.g. $\rightarrow_\beta$, $\twoheadrightarrow_\beta$, etc.).

Definition 1.4.6. Let R be a definition of reduction. We say that R is *Church-Rosser* if $\twoheadrightarrow_R \vDash \diamond$.

We can define equality for a reduction R by the following infrence rules:

$$\frac{M \twoheadrightarrow_R N}{M =_R N} \qquad \frac{M =_R N}{N =_R M} \qquad \frac{M =_R N \quad N =_R L}{M =_R L}$$

We can prove Church-Rosser theorem for the relations that satisfy the Church-Rosser property.

Theorem 1.4.7. *Let R be Church-Rosser. If $M =_R N$, then there exists a term Z such that $M \twoheadrightarrow_R Z$ and $N \twoheadrightarrow_R Z$.*

Proof. We show this using structural induction.

- First, assume that $M =_R N$ since $M \twoheadrightarrow_R N$. Then, we can take $Z \equiv N$.
- Now, assume that $M =_R N$ since $N =_R M$. By the inductive hypothesis, we can find a term Z such that $N \twoheadrightarrow_R Z$ and $M \twoheadrightarrow_R Z$. So, the result holds in this case as well.

- Finally, assume that $M =_R N$ since $M =_R N$ and $N =_R L$. By the inductive hypothesis, we can find terms Z_1, Z_2 such that $M \twoheadrightarrow_R Z_1$, $N \twoheadrightarrow_R Z_1$, $N \twoheadrightarrow_R Z_2$ and $L \twoheadrightarrow_R L$. Since R is Church-Rosser, we know that there exists a term Z such that $Z_1 \twoheadrightarrow_R Z$ and $Z_2 \twoheadrightarrow_R Z$. Since $\twoheadrightarrow_R$ is transitive, we find that $M \twoheadrightarrow_R Z$ and $N \twoheadrightarrow_R Z$.

□

We now want to show that $\twoheadrightarrow_\beta$ is Church-Rosser. We would like to do this by showing that $\rightarrow_\beta$ is Church-Rosser, but that does not hold. So, we have to define another reduction relation that is more powerful than $\rightarrow_\beta$, but its transitive closure is still $\twoheadrightarrow_\beta$. This is the grand reduction.

Definition 1.4.8. The *grand reduction* $\twoheadrightarrow_1$ is defined by the following inference rules:

$$\frac{}{M \twoheadrightarrow_1 M} \qquad \frac{M \twoheadrightarrow_1 M'}{\lambda x.M \twoheadrightarrow_1 \lambda x.M'}$$

$$\frac{M \twoheadrightarrow_1 M' \quad N \twoheadrightarrow_1 N'}{MN \twoheadrightarrow_1 M'N'} \qquad \frac{M \twoheadrightarrow_1 M' \quad N \twoheadrightarrow_1 N'}{(\lambda x.M)N \twoheadrightarrow_1 M'[x := N']}$$

Proposition 1.4.9. *Let M and N be terms such that $M \rightarrow_\beta N$. Then, $M \twoheadrightarrow_1 N$.*

Proof. We prove using structural induction on $M \rightarrow_\beta N$.

- First, assume that $M \equiv (\lambda x.A)B$ and $N \equiv A[x := B]$. We have $A \twoheadrightarrow_1 A$ and $B \twoheadrightarrow_1 B$. Hence, $M \equiv (\lambda x.A)B \twoheadrightarrow_1 A[x := B] \equiv N$.
- Now, assume that $M \equiv \lambda x.M'$ and $N \equiv \lambda x.N'$, with $M' \rightarrow_\beta N'$. By inductive hypothesis, we have $M' \twoheadrightarrow_1 N'$. Hence,
$$M \equiv \lambda x.M' \twoheadrightarrow_1 \lambda x.N' \equiv N.$$
- Next, assume that $M \equiv M'Z$ and $N \equiv N'Z$, where $M' \rightarrow_\beta N'$. By inductive hypothesis, we have $M' \twoheadrightarrow_1 N'$. Since we also have $Z \twoheadrightarrow_1 Z$,
$$M \equiv M'Z \twoheadrightarrow_1 N'Z \equiv N.$$
- Finally, assume that $M \equiv ZM'$ and $N \equiv ZN'$, where $M' \rightarrow_\beta N'$. By inductive hypothesis, we have $M' \twoheadrightarrow_1 N'$. Since we also have $Z \twoheadrightarrow_1 Z$,
$$M \equiv ZM' \twoheadrightarrow_1 ZN' \equiv N.$$

So, the result follows from induction. □

Proposition 1.4.10. *Let M, N, N' be terms such that $N \twoheadrightarrow_1 N'$. Then, $M[x := N] \twoheadrightarrow_1 M[x := N']$.*

Proof. We prove using structural induction on M.

- If $M \equiv x$, then we have
$$M[x := N] \equiv N \twoheadrightarrow_1 N' \equiv M[x := N'].$$

- If $M \equiv y$, where x and y are distinct, then we have

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

  Since $y \twoheadrightarrow_1 y$, we find that $M[x := N] \twoheadrightarrow_1 M[x := N']$.

- Now, assume that $M \equiv \lambda y.M'$. The inductive hypothesis tells us that $M'[x := N] \twoheadrightarrow_1 M'[x := N']$. Hence,

  $$M[x := N] \equiv (\lambda y.M'[x := N]) \twoheadrightarrow_1 (\lambda y.M'[x := N']) \equiv M[x := N'].$$

- Finally, assume that $M \equiv KL$. The inductive hypothesis tells us that $K[x := N] \twoheadrightarrow_1 K[x := N']$ and $L[x := N] \twoheadrightarrow_1 L[x := N']$. So,

  $$M[x := N] \equiv K[x := N]L[x := N] \twoheadrightarrow_1 K[x := N']L[x := N'] \equiv M[x := N'].$$

So, the result follows from induction. □

Proposition 1.4.11. *Let M, M', N, N' be terms such that $M \twoheadrightarrow_1 M'$ and $N \twoheadrightarrow_1 N'$. Then, $M[x := N] \twoheadrightarrow_1 M'[x := N']$.*

Proof. We prove using structural induction on $M \twoheadrightarrow_1 M'$.

- If $M \equiv M'$, then we have

  $$M[x := N] \twoheadrightarrow_1 M[x := N'] \equiv M'[x := N].$$

- Now, assume that $M \equiv \lambda y.L$ and $M' \equiv \lambda y.L'$. By inductive hypothesis, we know that $L[x := N] \twoheadrightarrow_1 L'[x := N']$. Hence,

  $$\begin{aligned} M[x := N] &\equiv \lambda y.L[x := N] \\ &\twoheadrightarrow_1 \lambda y.L'[x := N'] \\ &\equiv M'[x := N']. \end{aligned}$$

- Next, assume that $M \equiv KL$ and $M' \equiv K'L'$. By inductive hypothesis, we know that $K[x := N] \twoheadrightarrow_1 K'[x := N']$ and $L[x := N] \twoheadrightarrow_1 L'[x := N']$. Hence,

  $$\begin{aligned} M[x := N] &\equiv K[x := N]L[x := N] \\ &\twoheadrightarrow_1 K'[x := N']L'[x := N'] \\ &\equiv M'[x := N']. \end{aligned}$$

- Finally, assume that $M \equiv (\lambda y.K)L$ and $M' \equiv K'[y := L']$. By inductive hypothesis, we know that $K[x := N] \twoheadrightarrow_1 K'[x := N']$ and $L[x := N] \twoheadrightarrow_1 L'[x := N']$. Hence,

  $$\begin{aligned} M[x := N] &\equiv (\lambda y.K[x := N])L[x := N] \\ &\twoheadrightarrow_1 (\lambda y.K'[x := N'])L'[x := N'] \\ &\equiv M'[x := N']. \end{aligned}$$

So, the result follows from induction. □

Proposition 1.4.12. *Let M and N be terms with $\lambda x.M \twoheadrightarrow_1 N$. Then, there exists a term M' such that $N \equiv \lambda x.M'$ with $M \twoheadrightarrow_1 M'$.*

Proof. We show this by considering the cases of $\twoheadrightarrow_1$:

- First, assume that $N \equiv \lambda x.M$. The result holds since $M \twoheadrightarrow_1 M$.
- Now, assume that $N \equiv \lambda x.M'$, where $M \twoheadrightarrow_1 M'$. Here as well, the result holds.

□

Proposition 1.4.13. *Let M, N, L be terms such that $MN \twoheadrightarrow_1 L$. Then, either*

1. *$L \equiv M'N'$ with $M \twoheadrightarrow_1 M'$ and $N \twoheadrightarrow_1 N'$ or*
2. *$M \equiv \lambda x.P$ and $L \equiv P'[x := N']$, with $P \twoheadrightarrow_1 P'$ and $N \twoheadrightarrow_1 N'$.*

Proof. We show this by considering the cases of $\twoheadrightarrow_1$:

- First, assume that $L \equiv MN$. Since $M \twoheadrightarrow_1 M$ and $N \twoheadrightarrow_1 N$, case (1) holds.
- Next, assume that $L \equiv M'N'$, where $M \twoheadrightarrow_1 M'$ and $N \twoheadrightarrow_1 N'$. Clearly, case (1) holds.
- Finally, assume that $M \equiv \lambda x.P$, and $L \equiv P'[x := N']$, with $P \twoheadrightarrow_1 P'$ and $N \twoheadrightarrow_1 N'$. Clearly, case (2) holds in this case.

□

Proposition 1.4.14. *The relation $\twoheadrightarrow_1$ has the diamond property, i.e. for terms M, M_1, M_2,*

$$M \twoheadrightarrow_1 M_1, M \twoheadrightarrow_1 M_2 \implies \exists M_3.(M_1 \twoheadrightarrow_1 M_3, M_2 \twoheadrightarrow_1 M_3).$$

Proof. We show this by induction on $M \twoheadrightarrow_1 M_1$.

- If $M \equiv M_1$, then let $M_3 \equiv M_2$. We know that $M_1 \equiv M \twoheadrightarrow_1 M_2$ and $M_2 \twoheadrightarrow_1 M_2$, so the result holds.
-
-

□

Theorem 1.4.15. *The transitive closure of the relation $\twoheadrightarrow_1$ is $\twoheadrightarrow_\beta$.*

Now, we have shown that β-reduction satisfies the Church-Rosser, and hence the Church-Rosser Theorem holds.

We now consider how to get the normal form.

Theorem 1.4.16 (Normalisation Theorem)**.** *Let M be a term. If it has a normal form, then iterated reductions of the leftmost redex leads to the normal form.*

The leftmost reduction strategy is called *normalising*. It can be used to find the normal form for a term, or to show that a term has no normal form. There are two forms of normalisation:

- Weak normalisation, where there exists a path that leads to the normal form, e.g. the factorial function.
- Strong normalisation, where every path leads to the normal form, e.g. $(\lambda xy.x + y)(1 + 2)(3 + 4)$.

In λ-calculus, there are 2 reduction strategies, both of which are leftmost- call-by-value and call-by-need. A term of the form $\lambda x.M$ is considered a *value*- it is the final result of a computation. Call-by-value uses leftmost innermost strategy (and is seen in Python), and call-by-need (or lazy evaluation, in Haskell) uses leftmost outermost strategy:

- By leftmost, we mean that in an application MN, M is reduced first, and then N.
- By innermost, we mean that in the term $(\lambda x.M)N$, N is reduced to a value before $M[x := N]$ is computed.
- By outermost, we mean that in the term $(\lambda x.M)N$, $M[x := N]$ is reduced to a value before N is computed.

In leftmost outermost strategy, we might have to compute N multiple times. To avoid this, we can compute it first, and then use the same value for other occurrences of N. If the expressions do not have side-effects, then this produces the same result as a pure leftmost outermost strategy, where N is evaluated every time it appears.

1.5 Computability

In this section, we consider computability. Computability requires a definition of computation. There are many notions of computation, including λ-calculus, automata, register machines. They all capture the same notion of computability. The Church-Turing Thesis states that collectively, these definitions are the 'true' notion of computation.

To formalise computability, we will consider ways of defining functions on positive integers. We can use λ-terms to represent natural numbers and booleans, and fixed-point operators to define recursive functions. We will show that recursive functions can be converted into λ-terms.

We start by defining primitive recursive functions. They can be defined using a set type of recursive functions. The primitive recursive functions are functions $\mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ that can be defined using the following *initial functions*:

- the constant zero function $Z(n) = 0$;
- the successor function $S^+(n) = n + 1$; and
- the selector function $U_{i,p}(n_0, \dots, n_p) = n_i$ for $0 \leq i \leq p$ and $p \in \mathbb{N}$;
- composition $F(\vec{n}) = H(G_1(\vec{n}), \dots, G_m(\vec{n}))$, where $H, G_1, \dots, G_m$ are primitive recursive functions.

Now, we say that a function F is primitive recursive if it is of the form

$$\begin{aligned} F(0, \vec{n}) &= H(\vec{n}) \\ F(k+1, \vec{n}) &= G(F(k, \vec{n}), k, \vec{n}), \end{aligned}$$

where H and G are primitive recursive. In other words, $F(0, \vec{n})$ is the base case, and only depends on the 'other' data $\vec{n}$, and the recursive case $F(k+1, \vec{n})$ depends on the previous result $F(k, \vec{n})$ along with the number k and the data $\vec{n}$.

We now give examples of functions in primitive recursion:

- the identity function on $\mathbb{N}$ is given by $id(n) = U_{0,0}(n)$.
- the addition function is given by

  $$\begin{aligned} \text{add}(0, n) &= id(n) \\ \text{add}(k+1, n) &= G(\text{add}(k, n), k, n), \end{aligned}$$

  where $G(x, y, z) = S^+(U_{0,2}(x, y, z))$.
- the multiplication function is given by

  $$\begin{aligned} \text{mult}(0, n) &= 0 \\ \text{mult}(k+1, n) &= G(\text{mult}(k, n), k, n), \end{aligned}$$

  where $G(x, y, z) = \text{add}(U_{0,2}(x, y, z), U_{2,2}(x, y, z))$.

- the factorial function is given by

$$\begin{aligned}\text{fact}(0) &= S^+(0)\\ \text{fact}(k+1) &= G(\text{fact}(k), k),\end{aligned}$$

  where $G(x, y) = \text{mult}(U_{0,1}(x, y), S^+(U_{1,1}(x, y)))$.

- the sum of the first n numbers is given by

$$\begin{aligned}\text{sum}(0) &= 0\\ \text{sum}(k+1) &= G(\text{sum}(k), k),\end{aligned}$$

  where $G(x, y) = \text{sum}(U_{0,1}(x, y), S^+(U_{1,1}(x, y)))$.

It turns out that primitive recursion is not the right definition for computability. To see this, consider the following function, called the Ackermann's function:

$$\begin{aligned}A(0, n) &= n + 1\\ A(m + 1, 0) &= A(m, 1)\\ A(m + 1, n + 1) &= A(m, A(m + 1, n)).\end{aligned}$$

Because each m or n is decreasing at each step, we know that the computation $A(m, n)$ terminates for every natural number m and n. However, the values of this function grow very fast, e.g. $A(4, 2)$ has about 20 000 digits! It turns out that Ackermann's function grows faster than any primitive recursive function, so it cannot be primitive recursive. However, we were able to give a recursive definition for it above, so primitive recursion is not the definition of computability we want.

To define the bigger class of recursive functions, we need one more notion. If $P(n)$ is a predicate on $\mathbb{N}$, i.e. a function $\mathbb{N} \to$ *bool*, then we denote by $\min n[P(n)]$ the smallest number n such that $P(n)$ holds; if there is no such number, then the value is undefined. We say that a function F is *recursive* if it can be defined using primitive recursion and minimalisation:

$$F(\vec{n}) = \min m[H(\vec{n}, m) = 0],$$

where for any $\vec{n}$, there is a natural number m such that $H(\vec{n}, m) = 0$.

Intuitively, primitive recursion allows us to write bounded loops, i.e. for loops. In particular, the number of iterations is bounded by a value in the program, i.e. the initial value of the argument. Minimalisation corresponds to a while loop- we continue looping until the predicate becomes false. This is an intuitive argument as to why recursive functions are a good definition for computable functions. Although not clear, it is possible to write Ackermann's function using primitive recursion and minimalisation.

Now, we will show that λ-calculus can be used to write recursive functions. We can define the initial functions as follows:

$$\begin{aligned}U_{i,p} &\equiv \lambda x_0 \ldots x_p.x_i\\ S^+ &\equiv \lambda x.\texttt{pair}(F)(x)\\ Z &\equiv \lambda x.x\end{aligned}$$

For composition $F(\vec{n}) = H(G_1(\vec{n}, \dots, G_m(\vec{n})))$, we define

$$F \equiv \lambda\vec{x}.H(G_1\vec{x}) \dots (G_m\vec{x})$$

Next, let F be a primitive recursive function

$$\begin{aligned} F(0, \vec{n}) &= H(\vec{n}) \\ F(k+1, \vec{n}) &= G(F(k, \vec{n}), k, \vec{n}), \end{aligned}$$

where H and G are defined by λ-terms H and G. Then, we can define F to be the fixed point of the function f:

$$f \equiv \lambda f x \vec{y}.(\texttt{if}(\texttt{iszero}x)(H\vec{y})(G(f(Px)\vec{y})(Px)(\vec{y}))),$$

where P is the predecessor function.

Finally, we want to define minimalisation. To do so, let P be the predicate. We define a function that iterates from a given number n until Pn is true, and returns n. So, it can be the fixed point of the function

$$h_P \equiv \lambda hz.\texttt{if}(Pz)(z)(h(S^+z)).$$

So, $\min P \equiv \texttt{fix}(H_p)(Z)$, where Z is zero. If F is given by

$$F(\vec{n}) = \min m[H(\vec{n}, m) = 0],$$

where H is defined by a λ-term H, then we can define F by

$$F \equiv \lambda\vec{x}.\texttt{min}(\lambda y.\texttt{iszero}H\vec{x}y).$$

Hence, every recursion function can be defined in λ-calculus. This means that λ-calculus is sufficient for general recursion and can be used to extend primitive recursion instead of minimalisation.

It is also true that every function $\mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ that we can defined in λ-calculus can be defined in λ-calculus. To do so, we can encode λ-terms as numbers and define reduction as a function.

1.6 Simply-typed λ-calculus

In this section, we introduce type checking for λ-calculus. We have presented λ-calculus as a foundation for functional programming, and shown how to represent data and define interesting functions. Many standard programming languages have static or compile-time type-checking. The types we add to λ-calculus give rise to *simply-typed λ-calculus*, where simply refers to the fact that this is the simplest of its kind and we can build on it in many ways, e.g. by adding polymorphism.

Initially, we will not give concrete types such as `int` or `bool`. We will instead start with some type A, using which we define functions. Hence, types are defined by the following BNF rule:

$$\begin{aligned} T &::= A \\ &\mid \quad T \to T \end{aligned}$$

So, some types are A, $A \to A$ and $(A \to A) \to (A \to A)$. By convention, the arrow operation $\to$ is right-associative, meaning that the final type can be written as $(A \to A) \to A \to A$.

In λ-terms, we have bound and free variables. The bound variables have a fixed type that can be stated within the λ-term. This is similar to many programming languages, where the arguments are given with their types. But, for us to keep track of the types of free variables, we keep use of a *typing context.* A typing context Γ is a set of variables associated to types, and defined by the following BNF:

$$\begin{aligned} \Gamma &::= \varnothing \\ &\mid \quad \Gamma, x \colon T. \end{aligned}$$

In the second rule $\Gamma, x \colon T$, we have extended an existing typing context Γ with a new variable-type pair: $x \colon T$.

Now, let Γ be a typing context and M a term. If the term M has type T under the typing context Γ, then we denote this by $\Gamma \vdash M \colon T$. We introduce a variation on the syntax for bounded variables, i.e. $\lambda x \colon T.M$, which means that x has type T. This is a *Church-style definition.*

We now define typing rules using inference rules:

$$\frac{x \colon T \in \Gamma}{\Gamma \vdash x \colon T}\text{TVar} \qquad \frac{\Gamma, x \colon T \vdash M \colon U}{\Gamma \vdash \lambda x \colon T.M \colon T \to U}\text{TAbs}$$

$$\frac{\Gamma \vdash M \colon T \to U \quad \Gamma \vdash N \to T}{\Gamma \vdash MN \colon U}\text{TApp}$$

Note that TVar is an axiom, i.e. $x \colon T \in \Gamma$ is a side condition. This is an inductive definition.

We will now consider some typing derivations for λ-terms. We start with the term $\lambda x.x$:

$$\cfrac{\cfrac{x \colon A \in x \colon A}{x \colon A \vdash x \colon A}\text{TVar}}{\vdash \lambda x \colon A.x \colon A \to A}\text{TAbs}$$

In this case, the typing context is empty, and is denoted by nothing to the left of $\vdash$. We typically do not keep the TVar applications in the inference rules.

Next, we do the typing derivation for $\lambda xy.x$:

$$\dfrac{\dfrac{x\colon A, y\colon A \vdash x\colon A}{x\colon A \vdash \lambda y\colon A.x\colon A \to A}\text{TAbs}}{\vdash \lambda x\colon A.\lambda y\colon A.x\colon A \to A \to A}\text{TAbs}$$

We now add some concrete rules in all typing judgments. We start by adding some axioms:

$$\dfrac{}{\Gamma \vdash \texttt{true}\colon \texttt{bool}}\text{TTrue} \qquad \dfrac{}{\Gamma \vdash \texttt{false}\colon \texttt{bool}}\text{TFalse} \qquad \dfrac{v \text{ is an integer literal}}{\Gamma \vdash v\colon \texttt{int}}.$$

Next, we add some mathematical operations and conditions.

$$\dfrac{\Gamma \vdash M\colon \texttt{bool} \quad \Gamma \vdash N\colon \texttt{bool}}{\Gamma \vdash M \text{ and } N\colon \texttt{bool}}\text{TAnd} \qquad \dfrac{\Gamma \vdash M\colon \texttt{int} \quad \Gamma \vdash N\colon \texttt{int}}{\Gamma \vdash M + N\colon \texttt{int}}\text{TPlus}$$

$$\dfrac{\Gamma \vdash M\colon \texttt{int} \quad \Gamma \vdash N\colon \texttt{int}}{\Gamma \vdash M == N\colon \texttt{bool}}\text{TEq}$$

$$\dfrac{\Gamma \vdash L\colon \texttt{bool} \quad \Gamma \vdash M\colon T \quad \Gamma \vdash N\colon T}{\Gamma \vdash \texttt{if } L \texttt{ then } M \texttt{ else } N\colon T}\text{TCond}$$

We will now show that well-typed programs do not go wrong. The point of introducing a static (compile-time) type system is that type checking guarantees the absence of runtime type errors. To show this, we first need to define wrong programs. This is achieved by adding the special term *wrong* to the syntax of λ-terms, and modify the reduction rules, e.g. $1 + \texttt{true} \to_\beta$ *wrong*. This term models stuck computations that are not values.

To show that well-type programs don't go wrong, we require the Type Safety Theorem. This is given in two parts- type preservation and progress.

Theorem 1.6.1 (Type Preservation)**.** *If $\Gamma \vdash M\colon T$ and $M \to_\beta N$, then $\Gamma \vdash N\colon T$.*

Theorem 1.6.2 (Progress)**.** *If $\Gamma \vdash M\colon T$, then either M is a value or there exists M' such that $M \to_\beta M'$.*

We say that the term *wrong* is untypable, i.e. if $\Gamma \vdash M\colon T$, then we cannot have $M \twoheadrightarrow_\beta$ *wrong*.

Unfortunately, simply-typed λ-calculus does not support recursion. This is because any fix term isn't typable. For instance, consider the following

$$\text{fix} = \lambda f.(\lambda y.f(\lambda z.yyz))(\lambda y.f(\lambda z.yyz))$$

The term yyz is not typable since that implies y is of the form $A \to A \to A$ and A at the same time. Hence, we cannot define recursive functions.

So, in a typed functional programming, we have to add a built-in construct, by either adding the fix term to have type

$$\text{fix}\colon (T \to T) \to T$$

for every type T, or by supporting recursive functions, e.g.

$$\text{fac}(n\colon \texttt{int})\colon \texttt{int} = \texttt{if } n == 0 \texttt{ then } 1 \texttt{ else } n * \text{fac}(n-1).$$

Adding recursion this way maintains the property that well-typed programs do not contain runtime type errors.

The simply-typed λ-calculus has an important property of *strong normalisation.* This means that every typable term has a normal form, which can be reached by β-reduction. Because of the Church-Rosser Theorem, it doesn't matter which reduction strategy is used; we no longer have to worry about non-termination, so any strategy will eventually produce the unique normal form of a term. This is another way to see why fix is untypable- if we could type fix, then we would be able to define recursive functions that fail to terminate, which contradicts the strong normalisation property.

It is possible to implement an algorithm to check whether or not a given term is *typable* (type inference), and to check whether or not a given term *has a given type* (type checking). Typability and type checking are different problems- the typability problem is solved by a type inference algorithm and finds the most general type of a given term, if typable. Type inference is used in Haskell.

Now, we consider the expressivity of simply-typed λ-calculus. We know that λ-calculus is able to express all computations, but this required the construction of recursive functions. Since simply-typed λ-calculus does not allow recursion, it is not Turing complete. We can add recursion in a type-safe manner to give us a typed programming language that is Turing complete.