6 Lambda-definable functions

In this chapter we briefly discuss another algebraic approach to computability based on the lambda calculus. This calculus, which was introduced by Alonzo Church and Haskell B Curry in the 1930s, has deep connections with logic and proofs, and plays an important role in the development of Philosophy, Logic, and Computer Science. The lambda-calculus allows for the definition of higher-order operations and is at the core of any functional programming language. In this chapter we study the amazing fact that, despite its simplicity, the lambda calculus is Turing complete, that is, able to define every computable function.

Comprehensive and very readable expositions of the Lambda-calculus can be found in the books

H Barendregt, The Lambda Calculus - Its Syntax and Semantics, H Barendregt, North Holland, 1984

R Hindley and J P Seldin, Introduction to Combinators and lambda-calculus London Mathematical Society, Student Texts 1, Cambridge University Press, 1986

6.1 The lambda calculus

6.1.1 Lambda terms

Lambda-terms are defined inductively by the following clauses:

(i) Every variable is a lambda-term (we will use lower case letters, possibly with indices, to denote variables).

(ii) If $x$ is a variable and $M$ is a lambda-term then $(\lambda x M)$ is a lambda-term, called abstraction.

(iii) If $M$ and $N$ are lambda-terms then $(MN)$ is a lambda term, called application.

Equivalently, lambda-terms $M$ can be described by the grammar

$$M ::= x \mid (\lambda x M) \mid (MM)$$

Notation. We will frequently omit brackets if this doesn’t cause ambiguities. For example, we may write $\lambda x M$ and $MN$ instead of $(\lambda x M)$ and $(MN)$ respectively. We also write $M_1 M_2 \ldots M_k$ for $(\ldots(M_1 M_2)\ldots M_k)$ (association to the left). By a term we will in this chapter always mean a lambda-term.

Rationale: Intuitively, an abstraction $\lambda x M$ denotes a procedure mapping any value $a$ to $M[X \mapsto a]$, where $M[X \mapsto a]$ denotes the value of $M$ in the environment assigning to the variable $x$ the value $a$.

An application $MN$ is to be understood as calling the procedure $M$ at the argument $N$.

As we will see, the power of the lambda-calculus rests mainly on the fact that a term can at the same term denote a procedure and an argument of a procedure. A term can even be even applied to itself, which we call self-application.
A **free occurrence** of a variable $x$ in a term $M$ is an occurrence of $x$ in $M$ that is not
within the scope of an abstraction $\lambda x$. The set $\text{FV}(M)$ of **free variables** of $M$ is the
set of variables that have a free occurrence in $M$.
Equivalently, $\text{FV}(M)$ can be defined by *structural recursion* on $M$:

(i) $\text{FV}(x) := \{x\}$.
(ii) $\text{FV}(\lambda x M) := \text{FV}(M) \setminus \{x\}$.
(iii) $\text{FV}(M N) := \text{FV}(M) \cup \text{FV}(N)$.

Intuitively, a variable $x$ is free in a term $M$ if the meaning of $M$ depends on the meaning
of $x$.
A term $M$ is called **closed** if $M$ doesn’t contain free variables, i.e. $\text{FV}(M) = \emptyset$.
Closed terms are also called **combinators**.

### 6.1.2 Examples of combinators

\[
\begin{align*}
1 & := \lambda x x \\
K & := \lambda x \lambda y x \\
S & := \lambda x \lambda y \lambda z ((xz)(yz)) \\
\omega & := \lambda x (xx) \\
\Omega & := (\lambda x (xx)) (\lambda x (xx)) \quad (= \omega)
\end{align*}
\]

### 6.1.3 Bound renaming

Intuitively, the combinator $I$ ($= \lambda x x$) denotes the procedure mapping any argument
to itself, that is $I$ denotes the identity function. Of course the particular choice of the variable $x$ is completely immaterial: if we replace $x$ by any other variable, say $y$, we obtain $\lambda y y$, which, obviously, denotes the identity operation as well. Therefore it seems reasonable to identify the terms $\lambda x x$ and $\lambda y y$ and regard them as representatives of the same combinator $I$. Similarly, $\lambda x \lambda y x$ should be identified with, say $\lambda y \lambda x y$, or $\lambda z \lambda x z$ (but $\lambda x \lambda y x$ and $\lambda y \lambda x x$ should of course not be identified).

In general a change of bound variables that does not alter the meaning of the term is called **bound renaming**. We do not give a mathematically precise (but clumsy) definition of bound renaming, but rather trust in the readers intuitive understanding of this notion. Two terms $M, M'$ that can be obtained from each other by a sequence of bound renamings are also called $\alpha$-**equivalent** or $\alpha$-**variants**, written $M =_\alpha M'$.

**Exercise.** Which of the following pairs of terms are $\alpha$-equivalent?

\[
\begin{align*}
\lambda y \lambda z z & \quad \lambda x \lambda y x \\
\lambda f \lambda g \lambda x ((fx)(gx)) & \quad \lambda x \lambda y \lambda z ((xz)(yz)) \\
\lambda x (xx) & \quad \lambda x (yy) \\
(\lambda x (xx))(\lambda x (xx)) & \quad (\lambda x (xx))(\lambda u (uu))
\end{align*}
\]
6.1.4 Substitution

The operative semantics of terms is based on substitution, that is, the operation of replacing a free variable by a term. However, due to the presence of bound variables, some care is necessary:

\[ M[N/x] := \text{the result of replacing every free occurrence of } x \text{ in } M \text{ by } N, \text{ possibly renaming bound variables in } M \text{ in order to avoid variable clashes} \]

By a ‘variable clash’ we mean the situation that a free variable \( y \) in \( N \) becomes bound by the substitution, that is, is placed into the scope of an abstraction \( \lambda y \) in \( M \). In that case the abstraction \( \lambda y \) and all variables in \( M \) bound by it have to be renamed.

**Example.** If in the term \( \lambda y(xy) \) we replace the free variable \( x \) by the term \( y \) we obtain \( \lambda y(yy) \). But this is not the correct substitution. To make the substitution correct we must rename the term \( \lambda y(xy) \) to, say, \( \lambda z(xz) \) and then do the replacement:

\[ \lambda y(xy)[y/x] = \lambda z(yz) \]

6.1.5 \( \beta \)-reduction

According to the intuitive understanding of a lambda abstraction \( \lambda x M \) as procedure the result of applying this procedure to a term \( M \), that is, \( \lambda x M N \) should have the same meaning as \( M[N/x] \). This motivates the following definition of \( \beta \)-conversion

\[(\lambda x M) N \rightarrow_\beta M[N/x]\]

A term of the form \( (\lambda x M) N \) is called a redex (reducible expression), and the term \( M[N/x] \) its \( \beta \)-contractum.

**Examples.**

\[
\begin{align*}
(\lambda x \lambda y x)(\lambda x x) & \rightarrow_\beta \lambda y \lambda x x \\
(\lambda x x)(\lambda x x) & \rightarrow_\beta \lambda x x \\
(\lambda x x)(\lambda x (xx)) & \rightarrow_\beta (\lambda x (xx))(\lambda x (xx))
\end{align*}
\]

We say that a term \( L \) \( \beta \)-reduces to a term \( R \), written

\[ L \rightarrow_\beta R \]

if \( R \) is obtained from \( L \) by replacing an occurrence of a redex by its \( \beta \)-contractum. We write \( \rightarrow^*_\beta \) for the reflexive transitive closure of \( \rightarrow_\beta \), that is, \( M \rightarrow^*_\beta N \) means that \( N \) is obtained from \( M \) by zero or more \( \beta \)-reductions.

**Example.**

\[
\begin{align*}
(\lambda x \lambda y \lambda z((xz)(yz)))(\lambda u u)(\lambda v w) & \rightarrow_\beta (\lambda y \lambda z(((\lambda y u)z)(yz)))(\lambda v w) \\
& \rightarrow_\beta (\lambda y \lambda z(z(yz)))(\lambda v w) \\
& \rightarrow_\beta \lambda z(z((\lambda v w)z)) \\
& \rightarrow_\beta \lambda z(zw)
\end{align*}
\]

Hence \( (\lambda x \lambda y \lambda z((xz)(yz)))(\lambda u u)(\lambda v w) \rightarrow^*_\beta \lambda z(zw) \).
6.1.6 Normal forms

A term is in **normal form** if it doesn’t contain a redex.

**Examples.** The combinators $I$, $K$, $S$, $\omega$ are in normal form, but $\Omega$ is not.

A term $N$ is **a normal form of** $M$ if $M \rightarrow^* N$ and $N$ is in $\beta$-normal form.

A term is **weakly normalizing** if it has a normal form.

A term $M$ is **strongly normalizing** if every reduction sequence $M \rightarrow_{\beta} \ldots$ eventually ends with a term in normal form.

**Examples.** The term $(\text{KI})\omega$ has the normal form $\lambda$ since

$$(\text{KI})\omega \rightarrow_{\beta} \lambda y \lambda \omega \rightarrow_{\beta} \lambda$$

but it is not strongly normalizing, since

$$(\text{KI})\omega \rightarrow_{\beta} (\text{KI})\omega \rightarrow_{\beta} \ldots$$

The term $(\lambda x \lambda y \lambda z ((x z) (y z)) (\lambda u \omega) (\lambda \omega \omega))$ from a previous example is strongly normalizing: It is easy to see that every reduction sequence starting with this term eventually ends in the term $\lambda z(z \omega)$.

6.1.7 Uniqueness of normal forms

An important theorem of the lambda calculus is that $\beta$-reduction is **confluent**, or has the **Church-Rosser property**. This means: If $M \rightarrow^*_\beta M_1$ and $M \rightarrow^*_\beta M_2$, then there is a term $M_3$ such that $M_1 \rightarrow^*_\beta M_3$ and $M_2 \rightarrow^*_\beta M_3$.

\[
\begin{array}{c}
M \\
\beta \\
M_1 \\
\beta \\
M_3 \\
\beta \\
M_2 \\
\end{array}
\]

From the Church-Rosser property it follows that every term has at most one normal form, and that the relation

\[ M =^\beta N : \iff M \rightarrow^*_\beta L \text{ and } N \rightarrow^*_\beta L \text{ for some term } L \]

is an equivalence relation.

**Exercise.** Prove this.
6.2 Representing the partial recursive functions by lambda-terms

6.2.1 Representing numbers by lambda-terms

In order to represent partial recursive functions by lambda-terms we first have to represent natural numbers by lambda-terms. The representation we choose is due to Church. It not just codes natural numbers, but can be viewed as revealing the essence of natural numbers, as the following tries to show:

What kind of objects are natural numbers? A possible answer to this question is: the number \( n \) means to iterate an operation \( n \) times. In other words: the number \( n \) transforms any given operation into its \( n \)-th iterate. For example, the number 5 transforms any operation \( f \) into its 5-th iterate \( \lambda x (f(f(f(f(f(x))))) \), in other words, the number 5 is the operation represented by the lambda-term \( \lambda f \lambda x (f(f(f(f(f(x)))))) \).

This leads us to a representation of the natural numbers 0, 1, 2, \ldots by so-called Church-numeral \( c_0, c_1, c_2, \ldots \), defined by

\[
c_n := \lambda f \lambda x (f^{[n]} x) := \lambda f \lambda x \left( f \ldots f(x) \right)
\]

So, we have \( c_0 = \lambda f \lambda x . x, c_1 = \lambda f \lambda x . f(x), c_2 = \lambda f \lambda x . (f(f(x))), \) e.t.c.

Note that all Church-numerals are in normal form.

6.2.2 Representing partial functions by lambda-terms

Once numbers are represented by lambda-terms, it is obvious how to use lambda-terms to represent partial functions \( f : \mathbb{N}^k \to \mathbb{N} \):

To every combinator \( M \) and every number \( k \) we assign a partial function

\[
M^{(k)} : \mathbb{N}^k \to \mathbb{N}
\]

by defining for \( a_1, \ldots, a_k \in \mathbb{N} \):

\[
M^{(k)}(a_1, \ldots, a_k) := \begin{cases} 
  b & \text{if } M c_{a_1} \ldots c_{a_k} \to^* \beta c_0 \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

A partial function \( f : \mathbb{N}^k \to \mathbb{N} \) is called lambda-definable if there is a combinator \( M \) such that \( M^{(k)} = f \). In this case we say that \( M \) represents \( f \).

Note that \( M^{(k)}(a_1, \ldots, a_k) \) is well-defined because of the uniqueness of normal forms.

**Example.** The combinator \( A := \lambda u \lambda x \lambda f \lambda x ((uf)(nf)) \) represents addition. This can be seen as follows:

\[
A c_n c_m \to^* \beta \lambda f \lambda x ((c_m f)(c_m f x)) \\
\to^* \beta \lambda f \lambda x (c_m f(f^{[n]} x)) \\
\to^* \beta \lambda f \lambda x (f^{(m)}(f^{(n)} x)) = c_{n+m}
\]

Therefore addition is lambda-definable.
6.2.3 Fixed points

The key to the Turing completeness of the lambda-calculus lies in the existence of a so-called fixed point combinator, that is a combinators $M$ such that for all lambda-terms $F$ we have

$$MF =_\beta F(MF)$$

An example of such a combinator is

$$Y := \lambda f((\lambda y(f(yy)))(\lambda y(f(yy))))$$

which was called by Curry the paradoxical combinator.

**Lemma.** $Y$ is a fixed point combinator.

**Proof.** Let $F$ be a lambda-term. Set $W := \lambda y(F(yy))$. Then

$$YF \to_\beta WW \to_\beta F(WW)$$

and

$$F(YF) \to_\beta F(WW)$$

Therefore $YF =_\beta F(YF)$.

**Remark.** Another fixed point combinator is due to Turing: Set $A := \lambda x\lambda y(y(xxy))$ and $\Theta := AA$. Then even

$$\Theta F = AAF \to_\beta F(AAM) = F(\Theta F)$$

6.2.4 Theorem (Kleene)

A partial function on the natural numbers is lambda-definable if and only if it is partial recursive.

**Proof** (sketch). In order to show that every partial recursive function is lambda-definable one has to show that all basic functions are lambda-definable and that the lambda-definable partial functions enjoy the same closure properties as the partial recursive functions.

**Basic function** $\rightarrow$ **representing term**

- $\text{zero} : \mathbb{N} \to \mathbb{N}$, $\text{zero}(x) := 0$ $\rightarrow \lambda u c_0$
- $\text{succ} : \mathbb{N} \to \mathbb{N}$, $\text{succ}(x) := x + 1$ $\rightarrow \lambda u \lambda f \lambda x(f(ufx))$
- $\text{proj}_i^n : \mathbb{N}^n \to \mathbb{N}$, $\text{proj}_i^n(x_1, \ldots, x_n) := x_i$ $\rightarrow \lambda u_1 \ldots \lambda u_n u_i$

**Closure under composition.** If $M$ represents a partial function function $g$ of $k$ arguments and $N_1, \ldots, N_k$ represent partial functions $h_1, \ldots, h_k$, each of $n$ arguments, then the term

$$\lambda u_1 \ldots \lambda u_k(M(N_1 u_1 \ldots u_n) \ldots (N_k u_1 \ldots u_n))$$

represents the composition of $g$ and $h_1, \ldots, h_k$.

**Closure under primitive recursion.** We only give the idea: Suppose that $f$ is defined by primitive recursion from $g$ and $h$, that is,
\[ f(x, 0) = g(x) \]
\[ f(x, y + 1) = h(x, y, f(x, y)) \]

(for simplicity we assumed that there is only one parameter \( x \)). The idea is to show that \( f \) is a fixed point of a suitable lambda-term. To this end we use functions \( \mathsf{if}(0, u, v) := u \), \( \mathsf{if}(y, u, v) := v \) for \( y > 0 \), and \( P(y) := y - 1 \) to rewrite the definition of \( f \) as

\[ f(x, y) = \mathsf{if}(y, g(x), h(x, P(y), f(x, P(y)))) \]

or, using lambda-notation

\[ f = \lambda x \lambda y \mathsf{if}(y, g(x), h(x, P(y), f(x, P(y)))) \]

which again can be rewritten as

\[ f = F \mathsf{f} \]

where

\[ F := \lambda z \lambda x \lambda y \mathsf{if}(y, g(x), h(x, P(y), z(x, P(y)))) \]

So, \( f \) is a fixed point of \( F \), and it easy to prove that \( F \) has only one fixed point. If we can show that \( F \) can be defined as a lambda-term, we are done, because then we can set \( f := YF \), which shows that \( f \) is lambda-definable.

In order to define \( F \) as a lambda-term it suffices to show that \( \mathsf{if} \) and \( P \) can be defined by lambda-terms. This can be done rather easily, but we omit the somewhat tedious definitions.

**Closure under minimization.** Similar to primitive recursion.

In order to prove that, conversely, every lambda-definable partial function is partial recursive, one has to code lambda-terms into numbers and show that \( \beta \)-reduction \( M \rightarrow_\beta N \) can be simulated by a primitive recursive functions. Using minimization one can then iterate \( \beta \)-reductions to simulate the any lambda-definable partial function by a partial recursive function.

### 6.3 Typed lambda-calculi

In most functional programming languages (e.g. ML, Haskell) every expression (that is, lambda-term) has a **type**. The most important consequence of typing is the fact that application of terms is restricted. For example, if \( M \) is of type \( \rho \rightarrow \sigma \) (the type of functions mapping objects of type \( \rho \) to objects of type \( \sigma \)) then for an application \( MN \) is allowed only if the term \( N \) has type \( \rho \).

Different typing disciplines give rise to different interesting classes of computable functions. This classes are closely connected with logical systems and can be used to calibrate the strenght of these systems. More information on this subject is given in the courses MAP121 Foundations and Logic and CS376 Programming with Abstract data Types.