Course Notes Equational Programming:
Lambda Calculus

Femke van Raamsdonk

November 6, 2017
Contents

1 Introduction 5

2 Terms and Reduction 7
  2.1 Lambda terms .......................... 7
  2.2 Beta reduction .......................... 14
  2.3 Exercises .............................. 16

3 Strategies and Confluence 21
  3.1 Call by value .......................... 22
  3.2 Call by need .......................... 22
  3.3 Lazy evaluation ....................... 24
  3.4 Termination ........................... 26
  3.5 Exercises .............................. 27

4 Data Types 31
  4.1 Definability ........................... 32
  4.2 Booleans ............................... 33
  4.3 Pairs ................................. 34
  4.4 Church numerals ........................ 35
  4.5 Lists ................................. 38
  4.6 Exercises .............................. 38

5 Recursion 41
  5.1 Fixed points .......................... 41
  5.2 Recursive functions ................. 43
  5.3 Expressivity .......................... 43
  5.4 Exercises .............................. 44

6 Typed Lambda Calculus 45
  6.1 Types .................................. 45
  6.2 Raw terms .............................. 46
  6.3 Subject reduction .................... 49
  6.4 Termination ........................... 50
  6.5 Decidability ........................... 50
CONTENTS

6.6 Polymorphism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
6.7 Exercises . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Chapter 1

Introduction
Chapter 2

Terms and Reduction

2.1 Lambda terms

The λ-calculus is an at first sight easy language in which functions and the application of a function to its argument can be expressed. The expressions of this language are the λ-terms which are built using abstraction and application. Abstraction is used to represent a function, and application to represent the application of a function to its argument.

Informally, a function is usually specified by giving its domain and codomain, and an equation describing its behaviour. Consider for example the following function $f$ mapping a natural number to its square:

$$f : \text{nat} \to \text{nat}$$
$$f(x) = \text{square}(x)$$

Here \text{nat} is the natural numbers 0, 1, 2, \ldots. An alternative notation is as follows:

$$f : \text{nat} \to \text{nat}$$
$$f : x \mapsto \text{square}(x)$$

If we want to say something about a function we usually use its name. For example, the application of the function $f$ from above to argument 5 is written as follows:

$$f(5)$$

and the summation of $f$ from 0 to 5 as follows:

$$\sum_0^5 f(x)dx.$$  

In the λ-calculus notation, we can use nameless functions. The function $f$ from the example is written as:

$$\lambda x. \text{square} x.$$
More generally, a term of the form
\[ \lambda x. M, \]
which is called an abstraction or a \( \lambda \)-abstraction, intuitively represents a function that maps input \( x \) to \( M \).

In the examples we will sometimes use predefined function such as \texttt{plus} : \texttt{nat} \( \times \) \texttt{nat} \( \rightarrow \) \texttt{nat} which adds two natural numbers to each other, and \texttt{mul} : \texttt{nat} \( \times \) \texttt{nat} \( \rightarrow \) \texttt{nat} which multiplies two natural numbers.

The application of a function \( F \) to its argument \( M \) is in the \( \lambda \)-calculus written as an \textit{application-term} \( F M \), so the juxtaposition of \( F \) and \( M \). For example, the application of the function \( \lambda x. \texttt{square} x \) to the argument \( 5 \) is written as follows:

\[ (\lambda x. \texttt{square} x) 5. \]

A term of the form
\[ F M, \]

is called an application or application-term.

\textbf{Lambda terms} We proceed by giving the definition of the set of \( \lambda \)-terms, built from variables, constants, abstraction, and application.

\textbf{Definition 2.1.1.} The set of \( \lambda \)-terms, denoted by \( \Lambda \), is inductively defined by the following clauses:

1. a variable \( x \) is a \( \lambda \)-term,
2. a constant \( c \) is a \( \lambda \)-term,
3. if \( M \) is a \( \lambda \)-term,
   then \( (\lambda x. M) \) is a \( \lambda \)-term (an abstraction),
4. if \( M \) and \( N \) are \( \lambda \)-terms,
   then \( (M N) \) is a \( \lambda \)-term (an application).

When we allow constants as building block of our \( \lambda \)-terms, as in the definition above, we sometimes speak of an enriched \( \lambda \)-calculus. If constants are not allowed, one sometimes speaks of the \textit{pure \( \lambda \)}-calculus.

We use the following notational conventions:

- Variables are written as \( x, y, z, \ldots \).
- Constants are written as \( c, d, e, \ldots \) or using more suggestive notation.
- Atoms are the union of the variables and the constants; atoms are written as \( a, b, \ldots \).
- Terms are written as \( M, N, P, F, \ldots \).
Parentheses. In the official notation a λ-term may contain many parentheses, which often does not contribute to readability. Therefore we adopt to the convention to omit some parentheses, but of course only if this does not introduce ambiguity. The (standard) conventions we assume are as follows:

- We omit outermost parentheses, and write $MN$ instead of $(MN)$.
- Application is assumed to be associative to the left. That is, we may write $(MNP)$ instead of $((MN)P)$. Note that the parentheses in $M(NP)$ are necessary.
- We write $(\lambda x. \lambda y. M)$ instead of $(\lambda x. (\lambda y. M))$.
- We sometimes combine consecutive λs and write for example $\lambda xy. M$ instead of $\lambda x. \lambda y. M$, or more generally $\lambda x_1 \ldots x_n. M$ instead of $\lambda x_1. \ldots. \lambda x_n. M$.
- The λ-abstraction extends to the right as far as possible. We write for example $\lambda x. xx$ instead of $\lambda x. (xx)$, or more generally $(\lambda x. MN)$ instead of $(\lambda x. (MN))$.

Using the conventions, we for example may write

$$(\lambda x y z. y (x y z)) \lambda w w v w$$

instead of

$$(\lambda x. (\lambda y. (\lambda z. (y ((x y) z))))) (\lambda w. (\lambda v. (v w))))$$

and

$$(\lambda x y. x x y) (\lambda z u. u (u z)) \lambda w w$$

instead of

$$(\lambda x y. ((x x) y)) (\lambda z u. (u (u z))) (\lambda w w)$$

Of course it is always allowed to add parentheses if this contributes to clarity, even if strictly speaking this is not necessary.

Terms as trees. Every λ-terms corresponds in a natural way with a tree. The variables and constants acts as leaves. The λ-abstraction is a unary node, and application a binary (ordered) node. That is, we have the following:

1. A variable $x$ is depicted as follows:

   $$x$$

2. A constant $c$ is depicted similarly to a variable:
3. An abstraction \( \lambda x. M \) is depicted as follows:

\[
\begin{array}{c}
\lambda x \\
| \\
M
\end{array}
\]

4. An application \( MN \) is depicted as follows:

\[
\begin{array}{c}
M \\
\oplus \\
\oplus \\
\oplus \\
\oplus \\
N
\end{array}
\]

It can be very helpful to draw terms as trees.

The term \( xyz \) is the following tree:

\[
\begin{array}{c}
\oplus \\
\oplus \\
\oplus \\
x \\
\oplus \\
y \\
\oplus \\
z
\end{array}
\]

The term \( x (yz) \) is the following tree:

\[
\begin{array}{c}
\oplus \\
\oplus \\
\oplus \\
x \\
\oplus \\
y \\
\oplus \\
z
\end{array}
\]

The term \( (\lambda x. x)y \) is the following tree:

\[
\begin{array}{c}
\oplus \\
\oplus \\
\oplus \\
\oplus \\
\oplus \\
\lambda x \\
| \\
x \\
y \\
| \\
x
\end{array}
\]
2.1. LAMBDA TERMS

**Subterms.** A subterm is a part of a term that corresponds to a subtree of the syntax tree. A term is also a subterm of itself. For example, $M$ is a subterm of $\lambda x. M$, but $\lambda x$ is not a subterm of $\lambda x. M$.

**Currying.** We have seen that a function mapping $x$ to $M$ (which may depend on $x$) can be written as the $\lambda$-term $\lambda x. M$. This is a function of one argument. A natural question is whether we can also represent function of two or more arguments. It turns out that this is the case.

Consider for example the predefined function $\text{plus} : \text{nat} \times \text{nat} \to \text{nat}$ that adds to natural numbers. We want to use it to write the function mapping $x$ and $y$ to $\text{plus} \ x \ y$ as a $\lambda$-term. Note that we cannot write $\lambda (x, y). \text{plus} \ x \ y$: this is not permitted by the $\lambda$-calculus syntax. It is, however, allowed, to first abstract over $x$ and next over $y$. This yields the following $\lambda$-term:

$$\lambda x. \lambda y. \text{plus} \ x \ y$$

This term represents in fact the function that takes as input a natural number for $x$ and maps it to a function that takes as input a natural number for $y$ and maps this to $x$ plus $y$. So this is a function which intuitively has type $\text{nat} \to (\text{nat} \to \text{nat})$. This is in set theory isomorphic to (but not the same as) $\text{nat} \times \text{nat} \to \text{nat}$.

The principle of transforming a function of multiple arguments to functions with a single argument is called *currying* after the logician Haskell B. Curry; it was already earlier described by Schönfinkel.

**Bound and free variables.** A crucial concept in the $\lambda$-calculus is that of a bound variable. An occurrence of a variable $x$ in a $\lambda$-term is bound by the first $\lambda x$ above it in the term tree. Then it is said to be in the scope of that $\lambda x$. For example in the term $\lambda x. \underline{x}$ the underlined occurrence of the variable $x$ is bound. Be careful if the same variable name is used more than once. For example, the underlined occurrence of $x$ in $\underline{\lambda x. \lambda y. \underline{x} y} x$ is bound by the first (underlined) $\lambda x$.

Bound variables play the role of formal parameters in programming languages. They occur in logic for example in the universal quantification, and in mathematics for example in the operation for integral. The underlined occurrences of $x$ are bound in $\forall x. P(x) \to P(\underline{x})$ and in $\int \underline{x}^2 dx$.

A variable occurrence that is not bound is said to be free. The set of free variables of a term $M$, notation $\text{FV}(M)$, is defined inductively as follows:

1. $\text{FV}(c) = \emptyset$
2. $\text{FV}(x) = \{x\}$
3. $\text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N)$
4. $\text{FV}(\lambda x. M) = \text{FV}(M) \setminus \{x\}$
A term is said to be *closed* if its set of free variables is empty.

A few examples:

- The underlined variable \( x \) is bound in \( \lambda x. x \).
- Both underlined occurrences of the variable \( x \) are bound in \( \lambda x. \text{plus} \, x \, y \).
- The underlined variable \( x \) is bound in \( \lambda x. \text{plus} \, x \, y \); the underlined variable \( y \) is free in \( \lambda x. \text{plus} \, x \, y \).
- The underlined variables \( x \) is bound in \( (\lambda x. x) \, y \); the underlined variable \( x \) is free in \( (\lambda x. x) \, y \).

**Substitution.** The computation rule of the \( \lambda \)-calculus is the \( \beta \)-reduction rule which expresses how the application of an abstraction to some argument is evaluated: \( (\lambda x. M) \, N \rightarrow_{\beta} M[x := N] \). The expression on the left-hand side of the arrow intuitively means: the term \( M \) in which all free occurrences of \( x \) are replaced by the term \( N \). We will study \( \beta \)-reduction below. Now we turn to the definition of substitution.

Intuitively, the substitution of a term \( N \) for all free occurrences of a variable \( x \) in a term \( M \) is as follows. We think of the tree corresponding to \( M \). The leaves are either constants or variables. Every leaf with label \( x \) is replaced by the tree corresponding to the term \( N \). The other leaves do not change. For example:

- \((\text{plus} \, x \, y)[x := 1] = \text{plus} \, 1 \, y\),
- \((\text{plus} \, x \, x)[x := 1] = \text{plus} \, 1 \, 1\),
- \((\lambda y. \text{plus} \, x \, y)[x := 1] = \lambda y. \text{plus} \, 1 \, y\),
- \(((\lambda x. \text{plus} \, x \, x) \, y)[y := 1] = (\lambda x. \text{plus} \, x \, x) \, 1\).

An important subtle point is that we should avoid that, when substituting \( N \) for \( x \) in \( M \), a free variable \( y \) in \( N \) gets bound by a \( \lambda y \) higher up in \( M \). This phenomenon is called capturing of free variables. For example \((\lambda y. x)[x := y]\) should *not* be \( \lambda y. y \).

The substitution of \( N \) for \( x \) in \( M \), notation \( M[x := N] \), is defined by induction on the definition of terms as follows:

- \( x[x := N] = N \),
- \( a[x := N] = a \) where \( a \neq x \) is a variable or a constant
- \( (P \, Q)[x := N] = (P[x := N]) \, (Q[x := N]) \)
- \( (\lambda x. P)[x := N] = \lambda x. P \)
2.1. LAMBDA TERMS

- \((\lambda y. P)[x := N] = \lambda y. (P[x := N])\) if \(x \neq y\) and \(y \not\in \text{FV}(N)\)
- \((\lambda y. P)[x := N] = \lambda z. (P[y := z][x := N])\) if \(x \neq y\) and \(z \not\in \text{FV}(N) \cup \text{FV}(P)\),
  \(y \in \text{FV}(N)\),

A question is how to determine \(z\) in the last clause. Sometimes the variables are assumed to be linearly ordered; then \(z\) can be taken to be the first variable satisfying the conditions as given in the clause.

Note that the phenomenon of substitution in the presence of bound variables also occurs in logic and in mathematics. For example, in \(\forall y. R(x, y)\) we should not simply replace \(x\) by a term \(f(y)\).

A few examples of substitutions:

- \((\lambda y. xy)[x := y] = \lambda z. yz\),
- \((\lambda y. xy)[x := y] = \lambda u. yu\),
- \((x (\lambda y. xy))[x := y] = y (\lambda z. yz)\),
- \((\lambda y. xy)[x := z] = \lambda y. zy\).

The following substitutions are not correct:

- \((\lambda y. xy)[x := y] = \lambda y. yy\),
- \((x (\lambda y. xy))[x := y] = y (\lambda y. yy)\).

Finally we remark that the expression \(M[x := N]\) is not a \(\lambda\)-term, because the substitution operator \([x := \_]\) is not part of the syntax of \(\lambda\)-terms. Substitution is hence defined on a meta-level.

Alpha conversion. Two terms \(M\) and \(N\) are said to be \(\alpha\)-equivalent if they are identical up to the renaming of bound variables. For example, \(\lambda x. x\) is \(\alpha\)-equivalent to \(\lambda y. y\). The renaming of bound variables is called \(\alpha\)-conversion. The intuition is that the name of a bound variable does not matter.

We will usually identify terms that are \(\alpha\)-equivalent. That means that we work with \(\alpha\)-equivalence classes of \(\lambda\)-terms. Also, we will assume that bound variables are renamed such that they are different from the free variables.

This is also usual in logic and mathematics: the formula \(\forall x. P(x) \rightarrow P(x)\) is identified with the formula \(\forall y. P y \rightarrow P y\), and the expression \(\int x^2 dx\) is identified with the expression \(\int y^2 dy\).

The name of a free variable is of course important, and should not be changed.

A few examples:

- The term \(\lambda x. x\) is \(\alpha\)-equivalent to the term \(\lambda y. y\).
- The term \(\lambda x. \text{plus} \times x\) is \(\alpha\)-equivalent to the term \(\lambda y. \text{plus} y y\). The term \(\lambda x. \text{plus} \times x\) is not \(\alpha\)-equivalent to the term \(\lambda y. \text{plus} y x\).
• The term \( \lambda x. \text{plus} \, x \, y \) is \( \alpha \)-equivalent to the term \( \lambda z. \text{plus} \, z \, y \). The term \( \lambda x. \text{plus} \, x \, y \) is not \( \alpha \)-equivalent to the term \( \lambda y. \text{plus} \, y \, y \).

• The term \( x \, (\lambda x. \, x) \) is \( \alpha \)-equivalent to the term \( x \, (\lambda y. \, y) \). The term \( x \, (\lambda x. \, x) \) is not \( \alpha \)-equivalent to the term \( y \, (\lambda y. \, y) \).

• The term \( (\lambda x. \, x) \, y \, z \) is \( \alpha \)-equivalent to the term \( (\lambda y. \, y) \, y \, z \).

### 2.2 Beta reduction

The \( \beta \)-reduction rule describes the dynamics of the \( \lambda \)-calculus. It expresses how the application of a function to an argument is evaluated.

The \( \beta \)-reduction rule is as follows:

\[
(\lambda x. M) \, N \rightarrow_{\beta} M[x := N].
\]

The left-hand side of the \( \beta \)-reduction rule is the application of an abstraction to an arbitrary term. The right-hand side of the \( \beta \)-reduction rule stands for the term \( M \) in which all free occurrences of \( x \) are replaced by the term \( N \); here substitution as defined above is used.

Consider for example the abstraction \( \lambda x. \, x \). This is the identity function intuitively mapping a term to itself. According to the \( \beta \)-reduction rule, the application of the identity function to argument 3 rewrites to 3:

\[
(\lambda x. \, x) \, 3 \rightarrow_{\beta} 3.
\]

Another example of an application of the \( \beta \)-reduction rule:

\[
(\lambda x. \, \text{plus} \, x \, 3) \, 1 \rightarrow_{\beta} \text{plus} \, 1 \, 3.
\]

Now we proceed by giving two different but equivalent definitions of the \( \beta \)-reduction relation induced by the \( \beta \)-reduction rule.

The first definition is as follows.

**Definition 2.2.1.** The \( \beta \)-reduction relation, notation \( \rightarrow_{\beta} \) or sometimes \( \rightarrow \), is induced by the axiom for \( \beta \)-reduction, also called the \( \beta \)-reduction rule:

\[
(\lambda x. M) \, N \rightarrow_{\beta} M[x := N]
\]

and in addition three rules expressing that the \( \beta \)-reduction rule can be applied anywhere in a term:

\[
\frac{M \rightarrow_{\beta} M'}{\lambda x. M \rightarrow_{\beta} \lambda x. M'}
\]

\[
\frac{M \rightarrow_{\beta} M'}{M \, N \rightarrow_{\beta} M' \, N}
\]

\[
\frac{N \rightarrow_{\beta} N'}{M \, N \rightarrow_{\beta} M \, N'}
\]
2.2. BETA REDUCTION

A second definition for the same $\beta$-reduction relation is as follows. First, we define a $\beta$-redex to be a term of the form $(\lambda x. M) N$. The term redex stands for reducible expression and is also used for systems with other rewrite rules. Second, an elementary $\beta$-reduction step is obtained by contracting a $\beta$-redex: $(\lambda x. M) N \rightarrow_\beta M[x := N]$. Third, we define a context to be a term with one hole, written as $\square$, in it. If $C[]$ is a context, then $C[P]$ is obtained by replacing the hole in $C[]$ by the term $P$. Example: if $C[] = \lambda x. \square$ and $P = x$, then $C[P] = \lambda x. x$. Then the second (equivalent) definition of $\beta$-reduction is as follows.

**Definition 2.2.2.** The $\beta$-reduction relation is obtained by performing an elementary $\beta$-reduction step in a context:

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

We will be interested in sequences of $\beta$-reduction steps. Such a sequence, consisting of zero, one, or more $\beta$-reduction steps is called a $\beta$-reduction sequence, or $\beta$-rewrite sequence, or shortly reduction sequence, reduction, or rewrite sequence.

The many-step $\beta$-reduction relation is obtained by taking the reflexive-transitive closure of the (one-step) $\beta$-reduction relation $\rightarrow_\beta$. It is denoted by $\rightarrow_\beta^\ast$.

A few examples of reduction sequences:

- $(\lambda x. x) 3 \rightarrow_\beta 3$.
  The identity function $\lambda x. x$ does not do anything with the argument $3$; it is returned exactly once without changing it.

- $(\lambda x. f x x) 3 \rightarrow_\beta (f 3) 3$.
  The argument $3$ is duplicated.

- $(\lambda x. 5) 3 \rightarrow_\beta 5$.
  The argument $3$ is erased.

- An illustration of currying:
  $$((\lambda x. \lambda y. f x y) 3) 5 \rightarrow_\beta (\lambda y. f 3 y) 5$$
  $$\rightarrow_\beta (f 3) 5$$

- An argument can be a function itself:
  $$(\lambda y. y) (\lambda x. x) 3 \rightarrow_\beta (\lambda x. x) 3$$
  $$\rightarrow_\beta 3.$$

Note:
$(\lambda x.x) (\lambda x.x) 3$ is $((\lambda x.x) (\lambda x.x)) 3$ because application is left-associative.
The term $\lambda f . f \; (f \; x)$ is sometimes called *twice*:

\[
((\lambda f . \lambda x . f \; (f \; x)) \; \text{square}) \; 3 \quad \rightarrow_{\beta} \quad (\lambda x . \text{square} \; (\text{square} \; x)) \; 3 \\
\quad \rightarrow_{\beta} \quad \text{square} \; (\text{square} \; 3)
\]

The term $(\lambda x . x \; x) \; ((\lambda y . y) \; z)$ can be reduced to $(\lambda x . x \; x) \; z$ but also to $(\lambda y . y) \; z \; (\lambda y . y) \; z$. So $\beta$-reduction is non-deterministic.

The term $(\lambda x . x \; x) \; (\lambda x . x \; x)$ reduces to itself:

\[
(\lambda x . x \; x) \; (\lambda x . x \; x) \rightarrow_{\beta} (\lambda x . x \; x) \; (\lambda x . x \; x)
\]

So $\beta$-reduction is non-terminating.

A $\lambda$-term that does not contain a $\beta$-redex is said to be a *normal form* or *$\beta$-normal form*. Examples of $\beta$-normal forms are $x$, $\lambda x . x$, $x \; (\lambda y . y)$. The term $(\lambda x . x) \; 3$ is not a normal form (also: not in normal form) because it can perform a $\beta$-reduction step. Intuitively, a $\beta$-reduction sequence models a computation, and a $\beta$-normal form is a result of a computation.

**Famous terms.** A closed $\lambda$-term is also called a *combinator*. Four famous combinators are:

- $I = \lambda x . x$
- $K = \lambda x . \lambda y . x$
- $S = \lambda x . \lambda y . \lambda z . x \; z \; (y \; z)$
- $\Omega = (\lambda x . x \; x) \; (\lambda x . x \; x)$

### 2.3 Exercises

1. Write as a $\lambda$-term the function that takes one input and maps it to the constant $0$ as $\lambda$-term.

2. Write as a $\lambda$-term the function that takes two inputs and maps them to the constant $0$.

3. Write the following $\lambda$-terms in the notation with all parentheses and all $\lambda$s:

   - (a) $\lambda xy . (x \; (y \; z)) \; \lambda x . y \; x \; x$
   - (b) $\lambda xy . y \; y \; y \; y \; x \; x$
4. Write the following \(\lambda\)-terms in the notation with as few parentheses and \(\lambda\)s as possible:
   (a) \(((\lambda x. \lambda y. (((x y) z) u) (v w)))) (\lambda x. (y (x x))),
   (b) \(((\lambda x. x) (((\lambda x. \lambda y. ((x y) y)) (\lambda x. (\lambda y. (x (x y)))))\).

5. Draw the term trees of the following \(\lambda\)-terms, and indicate which variables are bound, and by which \(\lambda\) they are bound:
   (a) \((\lambda xyz. y (x y z)) \lambda uv. v w,\)
   (b) \((\lambda xy. x x y) (\lambda zu. u (u z)) \lambda w. w.\)

6. Underline in the following \(\lambda\)-terms the bound variables, and also indicate by which \(\lambda\) they are bound.
   (a) \((\lambda x. \lambda y. \lambda z. y (x y z)) \lambda uv. v w,\)
   (b) \((\lambda xy. x x y) (\lambda zu. u (u z)) \lambda w. w.\)

7. Indicate for all pairs of \(\lambda\)-terms below whether or not they are \(\alpha\)-convertible:
   (a) \(x\) and \(y\),
   (b) \(\lambda x. x x\) and \(\lambda y. y y\),
   (c) \(\lambda u. u u\) and \(\lambda u'. u' u'\),
   (d) \((\lambda xyz. y (x y z)) \lambda uv. v w\) and \((\lambda x' y z'. y (x' y z')) \lambda uv'. v w',\)
   (e) \((\lambda x. x) x\) and \((\lambda y. y) y,\)
   (f) \(\lambda x. x y\) and \(\lambda y. z y y,\)
   (g) \(\lambda x. \text{mul} x y\) and \(\lambda y. \text{mul} y y,\)
   (h) \(\lambda x y z. (y z)\) and \(\lambda z x y. z (x y),\)
   (i) \(\text{mul} x y\) and \(\text{mul} y x.\)

8. Calculate the following substitutions:
   (a) \(\text{mul} x x[x := 5]\),
   (b) \(\lambda x. \text{mul} x y[x := 5]\),
   (c) \(\lambda x. \text{mul} x y[y := 5]\),
   (d) \(\lambda x. \text{mul} x y[y := \text{plus} x x]\),
   (e) \(\lambda z. \text{plus} x 3[z := \text{mul} x 5]\),
   (f) \(\text{mul} x x[y := 5]\),
   (g) \(\lambda y. x[x := \text{plus} y 5]\),
   (h) \(\lambda x. f x[f := \lambda y. \text{plus} x y]\),
   (i) \(f 3[f := \lambda x. \text{mul} x 7]\),
(j) \((f (f x))[f := \lambda x. \text{mul } x \, 7].\)

9. Indicate whether the following \(\lambda\)-terms are in \(\beta\)- and/or \(\delta\)-normal form:

(a) \(5 (\lambda x. \text{plus } x \, x) \, 3\),
(b) \(\text{plus } x \, 5\),
(c) \(\lambda x. (\lambda y. \, y) \, 3\),
(d) \((\lambda x. x) \, (\lambda y. \, y) \, x\),
(e) \(\lambda x. x \, ((\lambda y. \, y) \, x)\),
(f) \(\text{plus } 3 \, 5\).

10. Underline in the following \(\lambda\)-terms the redexes, and perform reductions to normal form is possible (write down every step):

(a) \(I \, z\)
(b) \((\lambda x. y) \, z\)
(c) \((\lambda x. \lambda y. \, y \, x) \, u \, v\)
(d) \((\lambda x. x \, x) \, (\lambda x. x \, x)\)
(e) \((\lambda x. x \, x \, x) \, (\lambda x. x \, x \, x)\)
(f) \((\lambda y. \text{mul } y \, 3) \, y\)
(g) \(((\lambda x y. \text{mul } x \, y) \, 3) \, 7\)
(h) \((\lambda x. x) \, (\lambda y. \text{mul } 3 \, y) \, 7\)
(i) \((\lambda f x. f) \, (\lambda y. \text{plus } y \, x) \, 3\)
(j) \((\lambda f. f \, 3) \, (\lambda x. \text{mul } x \, 7)\)
(k) \((\lambda f x. f \, (f \, (x))) \, (\lambda x. \text{mul } x \, 7) \, 3\).

11. (a) Draw the term-tree of the following \(\lambda\)-term:
\[M = (\lambda x. \lambda y. y \, y) \, (\lambda u. u) \, (\lambda u. u) \, ((\lambda u. u) \, (\lambda z. z))\]

(b) Give two different \(\alpha\)-equivalent variants of the term \(M\) from (11a).
(c) Reduce the term \((\lambda x. \lambda y. x \, y) \, (\lambda z. y)\) to \(\beta\)-normal form (write down every step).

12. (a) Draw the term-tree of the following \(\lambda\)-term:
\[\lambda x. x \, y \, x \, x \, (\lambda y. x \, (y \, u))\]

(b) Give two different \(\alpha\)-equivalent variants of the term from (12a).
(c) Reduce the \(\lambda\)-term \((\lambda x. x \, y) \, (\lambda z. y \, z)\) to \(\beta\)-normal form.

13. The \(\lambda\)-term \(M\) is defined as follows:
\[M = \lambda x. ((\lambda y. \lambda z. ((x \, y) \, ((\lambda u. \lambda v. z \, u) \, (v \, z)))) \, [x \, z])\]
2.3. EXERCISES

(a) Draw the term-tree of \( M \).
(b) Give two different \( \alpha \)-equivalent variants of \( M \).
(c) Reduce the \( \lambda \)-term \( M \) to \( \beta \)-normal form (write down every step).

14. The term \( N \) is defined as follows:

\[
N = \lambda z. ((\lambda x. \lambda y. x (y v)) (\lambda u. \lambda v. y u z))
\]

(a) Draw the term-tree for \( N \).
(b) Give two different \( \alpha \)-equivalent variants of \( N \).
(c) Reduce the term \( N \) to \( \beta \)-normal form (write down every step).

15. The term \( Q \) is defined as follows:

\[
Q = (\lambda x. \lambda y. x (\lambda u. u v y)) (\lambda z. \lambda v. z y)
\]

(a) Draw the term-tree for \( Q \).
(b) Give two different \( \alpha \)-equivalent variants of the term \( Q \).
(c) Reduce \( Q \) to \( \beta \)-normal form (write down every step).

16. 

(a) Give \( \lambda \)-terms \( M \) and \( N \) such that \( M \) and \( N \) are \( \beta \)-normal forms but \( M[x := N] \) is not a \( \beta \)-normal form.
(b) Give \( \lambda \)-terms \( M \) and \( N \) such that \( M \) and \( N \) both have a \( \beta \)-normal form, but \( M[x := N] \) does not have a \( \beta \)-normal form.
(c) Give \( \lambda \)-terms \( M \) and \( N \) such that \( M \) does not have a \( \beta \)-normal form, but \( M[x := N] \) has a \( \beta \)-normal form.
(d) Give terms \( M \) and \( N \) such that \( N \) does not have a \( \beta \)-normal form, but \( M[x := N] \) has a \( \beta \)-normal form.
Chapter 3

Strategies and Confluence

A term may be the starting point of different reduction sequences. As an example we consider the following term: \((\lambda x. \lambda y. x) \Omega\). This term admits an infinite reduction:

\[(\lambda x. \lambda y. x) \Omega \rightarrow^\beta (\lambda x. \lambda y. x) \Omega \rightarrow^\beta (\lambda x. \lambda y. x) \Omega \rightarrow^\beta \ldots\]

and also an efficient reduction to \(\beta\)-normal form:

\[(\lambda x. \lambda y. x) \Omega \rightarrow^\beta (\lambda y. \Omega) \rightarrow^\beta 1\]

and also a somewhat less efficient reduction to \(\beta\)-normal form:

\[(\lambda x. \lambda y. x) \Omega \rightarrow^\beta (\lambda x. \lambda y. x) \Omega \rightarrow^\beta (\lambda y. l) \Omega \rightarrow^\beta 1.\]

A strategy or reduction strategy tells us how to reduce a term, usually with the aim of finding a reduction sequence with a certain property. An example of a property we may be interested in is ‘ending in normal form’. A strategy can be considered as a function that takes as input a term not in normal form, and gives back as output one (or more, in case of a many-step strategy) redex(es) in that term. We usually describe strategies in an informal way.

This chapter is mainly concerned with the following reduction strategies:

- An efficient reduction strategy that yields reduction sequences in which no duplication of redexes occurs.
- A normalizing strategy that yields a reduction sequence to normal form whenever the initial term has a normal form.
- A strategy used in the implementation of some functional programming languages.

Often the goal of a reductions strategy is to find a reduction sequences that ends in normal form. Once the normal form has been found, we know that it is ‘the right one’ because of confluence. A consequence of confluence is that a \(\lambda\)-term has at most one normal form. That is: a term has either no normal form (an example of such a term is \(\Omega\)), or it has one normal form.
3.1 Call by value

The leftmost-innermost reduction strategy tells us to contract in a term that is not in normal form the leftmost-innermost redex. That is the redex that is the leftmost redex of all redexes that do not contain a redex as subterm. Examples of reduction sequences following the leftmost-innermost strategy:

- \((\lambda x. f x) ((\lambda x. x) 3) \to_\beta (\lambda x. f x) 3 \to_\beta f 3\)
- \((\lambda x. f x x) ((\lambda x. x) 3) \to_\beta (\lambda x. f x x) 3 \to_\beta f 3 3\)
- \((\lambda x. y) \Omega \to_\beta (\lambda x. y) \Omega \to_\beta (\lambda x. y) \Omega \to_\beta \ldots\)
- \((\lambda x. f y) ((\lambda x. x) 3) \to_\beta (\lambda x. f y) 3 \to_\beta f y\)

In a leftmost-innermost reduction sequence, the arguments of a function are passed to the function body only if they are completely evaluated. This explains the terminology call by value which is also used to indicate the leftmost-innermost reduction strategy.

Advantages of the leftmost-innermost strategy:

- Only terms in normal form may be copied. Hence no redex is contracted more than once. See the second example above.
- It is relatively easy to implement the leftmost-innermost strategy.

Disadvantages of the leftmost-innermost strategy:

- The strategy is not normalizing: It is possible that the strategy yields an infinite reduction sequence whereas the initial term has a normal form. See the third example above.
- A leftmost-innermost reduction sequence may contain steps that intuitively do not contribute to the normal form. See the fourth example above.

The functional languages ML and Hope are implemented according to the call-by-value evaluation mechanism. Such functional languages are also called eager.

3.2 Call by need

The leftmost-outermost reduction strategy prescribes to contract in a term that is not in normal form the leftmost-outermost redex. That is, from the outermost redexes (that are not contained by another redex) the leftmost-one. Examples of reduction sequences according to the leftmost-outermost reduction strategy:

- \((\lambda x. f x) ((\lambda x. x) 3) \to_\beta f ((\lambda x. x) 3) \to_\beta f 3\)
- \((\lambda x. f x x) ((\lambda x. x) 3) \to_\beta f ((\lambda x. x) 3) ((\lambda x. x) 3) \to_\beta f 3 ((\lambda x. x) 3) \to_\beta f 3 3\)
3.2. CALL BY NEED

- \((\lambda x. y) \Omega \rightarrow_\beta y\)
- \((\lambda x. f y) (\lambda x. 3) \rightarrow_\beta f y\)

In a leftmost-outermost reduction sequence the arguments are passed to the function body without evaluating them to normal form. Advantages of the leftmost-outermost reduction strategy:

- The leftmost-outermost reduction strategy is normalizing. That is: if a term has a normal form, then the reduction sequence according to the leftmost-outermost reduction strategy ends in that normal form.

- The leftmost-outermost redex is needed. That is, leftmost-outermost reductions steps contribute to the normal form; we cannot avoid doing them if we wish to find a normal form.

Because the \(\beta\)-normal form of a term is unique, the leftmost-outermost reduction strategy has the following property: if a term \(M\) has a normal form \(P\), then the leftmost-outermost reduction sequences starting in \(M\) ends after finitely many steps in \(P\). If a term \(M\) has no normal form, then the leftmost-outermost reduction sequences starting in \(M\) is infinite.

A disadvantage of the leftmost-outermost reduction strategy is that redexes may be copied. Hence a leftmost-outermost reduction sequence to normal form is not necessarily a shortest reduction to normal form. This happens for example in the following reduction sequence, where the abstraction term of the redex that is contracted is underlined. In addition we use the familiar \(I = \lambda x. x\):

\[
(\lambda x. x x) (11) \rightarrow_\beta (11) (11) \\
\rightarrow_\beta 1(11) \\
\rightarrow_\beta 1 \\
\rightarrow_\beta 1
\]

This reduction sequence consists of four \(\beta\)-reduction steps, whereas it is possible to reach the normal form in only three steps by postponing the copying step:

\[
(\lambda x. x x) (11) \rightarrow_\beta (\lambda x. x x) 1 \\
\rightarrow_\beta 1 \\
\rightarrow_\beta 1
\]

Disadvantages of the leftmost-outermost reduction strategy:

- Redexes may be copied; as a consequence the leftmost-outermost reduction strategy does not necessarily yield a shortest reduction to normal form.

- The leftmost-outermost reduction strategy is relatively difficult to implement.

Lambda calculus with the leftmost-outermost reduction strategy is also called \textit{call by need} lambda calculus.
CHAPTER 3. STRATEGIES AND CONFLUENCE

3.3 Lazy evaluation

Lazy evaluation is a refinement of the leftmost-outermost reduction strategy, where we reduce terms if possible to weak head normal form. Moreover, copying of redexes is avoided by using sharing.

**Weak head normal forms.** In the implementation of for example Miranda and Haskell a \( \lambda \)-term is not evaluated to normal form, but (roughly) to *weak head normal form*. A weak head normal form, abbreviation WHNF, is a \( \lambda \)-term of one of the following two shapes:

1. \( \lambda x. P \) with \( P \) some \( \lambda \)-term,
2. \( x \, P_1 \ldots \, P_n \) with \( n \geq 0 \) and \( P_1, \ldots, P_n \) arbitrary \( \lambda \)-terms.

We can give a definition of the set of (exactly) all normal forms similarly: A normal form is a \( \lambda \)-term of one of the following two shapes:

1. \( \lambda x. M \) with \( M \) a normal form,
2. \( x \, M_1 \ldots \, M_n \) with \( n \geq 0 \) and \( M_1, \ldots, M_n \) normal forms.

Note that the definition above is our first definition of the notion of weak head normal form.

A \( \beta \)-normal form is defined earlier as a term without a \( \beta \)-redex. We can prove that if a term does not have a redex, then it is in one of the shapes as given above, and vice versa, if it is in one of those shapes, then it does not have a redex. This proof is not very difficult. We restrict attention here to the pure untyped \( \lambda \)-calculus without constants.

Every normal form is a WHNF, but not every WHNF is a normal form. Examples of WHNFs (with \( I = \lambda x. x \)):

- \( \lambda x. x \). This WHNF is also a normal form.
- \( \lambda x. I \, x \). This WHNF has a normal form, but is not a normal form.
- \( \lambda x. \Omega \). This WHNF does not have a normal form
- \( x \). This WHNF is also a normal form.
- \( x \, (I \, x) \). This WHNF has a normal form, but is not a normal form.
- \( x \, \Omega \). This WHNF does not have a normal form.

The intuition of a WHNF is that it is a term where the first part of the result can be read off. Indeed, no matter how the term \( \lambda x. P \) is further reduced, it will always remain of the form \( \lambda x. P' \), for some \( P' \) a reduct of \( P \) (so with \( P \to \beta P' \)). Similarly, no matter how the term \( x \, P_1 \ldots \, P_n \) is further reduced, it will always remain of the form \( x \, P'_1 \ldots \, P'_n \), for \( P'_1, \ldots, P'_n \) with \( P_i \to \beta P'_i \) for all \( i \in \{1, \ldots, n\} \).
3.3. **LAZY EVALUATION**

**Lazy reduction sequences.** A λ-term that is not in WHNF, is of the shape

\[(\lambda x. N) P_1 \ldots P_n\]

The redex \((\lambda x. N) P_1\) in this term is sometimes said to be the *lazy redex*. A lazy redex is a leftmost-outermost redex, but a leftmost-outermost redex is not necessarily a lazy redex. The term \(x \Omega\) is in WHNF and hence does not have a lazy redex, but it has a leftmost-outermost redex: \(\Omega\).

A lazy reduction sequence is a reduction sequence in which in every step the lazy redex is contracted. If the initial term has a WHNF, then the lazy reduction sequence ends in a WHNF. If the initial term does not have a WHNF, then the lazy reduction sequence is infinite.

**Sharing.** We have seen that in a leftmost-outermost reduction sequence redexes may be copied more than once. As a consequence, a leftmost-outermost reduction sequence to normal form may be longer than strictly necessary. A way to partially remedy this phenomenon is by *sharing* subterms. Sharing cannot be expressed in the pure lambda calculus. We sketch here the rough idea. We use an extra operator to construct terms:

\[
\text{let } x = P \text{ in } M
\]

Here we mean that there is exactly one copy of the term \(P\), no matter whether the term \(M\) contains zero, one, or more occurrences of the variable \(x\). We can imagine the occurrences of \(x\) in \(M\) as pointers that point to the address of \(P\). The construction with let now can be used as follows to improve the efficiency of a leftmost-outermost reduction sequence:

\[
(\lambda x. x x) (11) \rightarrow_\beta \text{let } x = 11 \text{ in } x x
\]

\[
\rightarrow_\beta \text{let } x = 1 \text{ in } x x
\]

\[
\rightarrow \text{let } x = 1 \text{ in } 1 x
\]

\[
\rightarrow_\beta \text{let } x = 1 \text{ in } x
\]

\[
\rightarrow \text{let } x = 1 \text{ in } 1
\]

This reduction sequences is more efficient than the leftmost-outermost reduction sequence above, that is, in case we consider only \(\beta\)-reduction steps. However, note that in addition to \(\beta\) we need a form of ‘sharing management’. The let-construction is manipulated according to the intuition for lazy reduction: do the least possible and do it last moment. The variable \(x\) in the term \(x x\) is only replaced by \(1\) if that replacement is necessary to make the leftmost-outermost redex ‘visible’.

**Lazy evaluation.** The ingredients of the lazy evaluation method are: the leftmost-outermost reduction strategy, weak head normal forms, and sharing. A leftmost-outermost reduction sequence to WHNF is also called a lazy reduction sequence. Lazy evaluation can be summarized by the following slogan:

*lazy evaluation is call by need reduction to WHNF plus sharing.*
The idea is hence: do the least possible, do it last moment, and at most once.

An advantage of lazy evaluation is that it opens the possibility to working with terms that can grow without limit. Think for example of infinite lists. In Haskell one can work with potentially infinite data structures thanks to the lazy evaluation mechanism. A well-known example of a program using infinite lists is the sieve of Eratoshenes to compute all prime numbers: roughly we start with the infinite list of all natural numbers. Then we filter out first the two-folds, then the three-folds, then the five-folds, and so on. Using lazy evaluation the program first yields the first prime number, then the second, and so on. In the limit to infinity the program yields all prime numbers.

3.4 Termination

A term $P$ is said to be terminating or strongly normalizing (SN) if all reduction sequences starting in $P$ are finite. That is: $P$ is not the start of an infinite reduction. Of course the $\beta$-normal forms are terminating. A few examples of other terminating $\lambda$-terms:

- $(\lambda x. y)(((\lambda x. x)\ z),$
- $(\lambda x. x)(\lambda y. y).

A term that is not strongly normalizing has a an infinite reduction sequence. Example of terms that are not terminating (or not strongly normalizing):

- $\Omega = (\lambda x. x x)(\lambda x. x x),$
- $Y\ z.$

A term that admits a reduction sequence to normal form, we also say that the term has a normal form, is said to be weakly normalizing (WN). A strongly normalizing term is weakly normalizing. There are weakly normalizing terms that are not strongly normalizing. Examples of such terms:

- $(\lambda x. y)\Omega$
  because this terms has a normal form:
  $(\lambda x. y)\Omega \rightarrow_{\beta} y$
  and also is the starting point of an infinite reduction:
  $(\lambda x. y)\Omega \rightarrow_{\beta} (\lambda x. y)\Omega \rightarrow_{\beta} \ldots.$

- $(\lambda x. y)(Y\ z)$
  because this term has a normal form:
  $(\lambda x. y)(Y\ z) \rightarrow_{\beta} y$
  and also is the starting point of an infinite reduction:
  $(\lambda x. y)(Y\ z) \rightarrow_{\beta} (\lambda x. y)(z((\lambda x. z(x x))((\lambda x. z(x x))))) \rightarrow_{\beta}$
  $(\lambda x. y)(z(z((\lambda x. z(x x))((\lambda x. z(x x))))) \rightarrow_{\beta} \ldots.$
3.5 Exercises

1. We consider the following terms, with \( Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \) the fixed point combinator that we will also encounter later.

\[
M_1 = (\lambda f. f 3) (\lambda x. (\lambda y. y) x) \\
M_2 = \omega_3 \omega_3 \text{ with } \omega_3 = \lambda x. x x \\
N_1 = Y u \text{ with } u \text{ a variable} \\
N_2 = (\lambda x. (\lambda y z. z y) x) u (\lambda x. x)
\]

Answer for these terms the following questions:

(a) Give a reduction sequence using the leftmost-innermost reduction strategy.

(b) Give a reduction sequence using the leftmost-outermost reduction strategy.

(c) Does the term have a normal form? If not, why not? If yes, what is its normal form?

(d) Does the term have a WHNF? If no, why not? If yes, what is the WHNF?

2. (a) Give an example of a reduction strategy that is not normalizing, and show with an example why the strategy is not normalizing.

(b) Which redex in the term \((\lambda u. u (\lambda y. z)) (\lambda x. x ((\lambda v. v) (\lambda v. v)))\) does not have to be contracted in order to reach the normal form? Give a reduction sequence to normal form that does not contract this redex.

(c) Reduce in the term \((\lambda x. (\lambda y z. z y) x) (\lambda x. x x) \Omega\) repeatedly the lazy redex until the WHNF is reached.

We use the following notation: \( I = \lambda x. x \) en \( \Omega = (\lambda x. x x) (\lambda x. x x) \).

3. (a) The rightmost-outermost strategy prescribes to contract in every term not in normal form the rightmost-outermost redex (that is, the rightmost one among the outermost redexes).

Reduce the following term using the rightmost-outermost strategy.

\[(\lambda u. \lambda v. w) (\lambda y. y) ((\lambda x. x x) (\lambda x. x x))\]

Is the rightmost-outermost strategy normalizing? Why (not)?

(b) Which redex in the following term does not have to be contracted in order to reach the normal form?

\[(\lambda x. \lambda y z. z y) (\lambda z. x) (\lambda u. v ((\lambda w. w) (\lambda w. w)))\]

Also, give a reduction sequence to normal form (write down every step explicitly) in which that redex is not contracted.
(c) Explain using an example a drawback of the leftmost-outermost reduction strategy.

4. (a) Which reduction strategy is not normalizing? Illustrate this with an example.
   (b) Give a reduction sequence to normal form with an ‘unnecessary step’ of the following term:
   $$(\lambda u. u (\lambda y. z)) (\lambda x. x ((\lambda v. v) (\lambda v. v))).$$
   Indicate the useless step.
   (c) Reduce in the term
   $$\Pi x \Omega$$ repeatedly the lazy redex until a WHNF is reached.
   We use the notation
   $$\Pi = \lambda x. x$$ and $$\Omega = (\lambda x. x x) (\lambda x. x x).$$

5. We use the notation $$\Pi = \lambda x. x$$ and $$\Omega = (\lambda x. x x) (\lambda x. x x).$$
   (a) Reduce the following term in a minimum number of steps to normal form
   $$N = (\lambda x. (\lambda u. u (\lambda z. z x y)) (\lambda x. x \Omega)) (\lambda y. z).$$
   (Write down every step.)
   (b) Reduce in the term $$(\lambda x. f (x \Omega \Pi)) (\lambda u. \lambda v. v w)$$ repeatedly the lazy redex until a WHNF is reached.

6. Consider the following term:
   $$M = (\lambda x. ((\lambda u. u) x) (\lambda z. x \Omega)) (\lambda y. z)$$
   with $$\Omega = (\lambda x. x x) (\lambda x. x x).$$
   (a) Depict $M$ as tree.
   (The abbreviation $\Omega$ should not be used in the picture.)
   (b) Give two different $\alpha$-equivalent variants of $M$.
   (c) Reduce $M$ using the leftmost-innermost, and using the leftmost-outermost strategy.

7. Consider the following term $M$:
   $$M = (\lambda f. f \Pi \Omega) (\lambda x y. x x)$$
   with $$\Omega = (\lambda x. x x) (\lambda x. x x)$$ and $$\Pi = \lambda x. x.$$
   (a) Depict $M$ as a tree.
   (Then the abbreviations $\Omega$ and $\Pi$ should not be used.)
   (b) Give two different $\alpha$-equivalent variants of $M$. 
(c) Reduce $M$ using the leftmost-innermost, and using the leftmost-outermost strategy.

8. Give an example of terms $M$, $N$, $P$ such that $(M[x := P])[y := Q] \neq (M[y := P])[x := Q]$.

9. Indicate for the following terms whether they are strongly normalizing (that is: terminating), weakly normalizing, or neither of the two:

(a) $(\lambda x. x x)(\lambda y. (f y))$
(b) $(\lambda x. x \Omega)(\lambda y. z)$
(c) $(\lambda x. x \Omega)(\lambda y. y)$
(d) $(\lambda x. x x x)(\lambda x. x x)$
Chapter 4

Data Types

In the \(\lambda\)-calculus a function that maps a variable \(x\) to \(M\) is written as the abstraction \(\lambda x. M\), and the application of a function \(F\) to an argument \(P\) is written as \(FP\). The \(\beta\)-reduction rule \((\lambda x. M) N \rightarrow_\beta M[x := N]\) expresses how the application of a function to an argument is evaluated. In fact it expresses the interaction between abstraction and application. In this chapter we investigate the expressivity of the \(\lambda\)-calculus.

The elements abstraction, application, and \(\beta\)-reduction (using substitution) form the essence of the \(\lambda\)-calculus. Although the \(\lambda\)-calculus may seem simple at first sight, it is sufficiently expressive to encode all algorithms. This is expressed by Church’s Thesis:

Everything that is intuitively computable
is definable in the \(\lambda\)-calculus.

Church’s Thesis, formulated by Church in the years 1930, immediately indicates the importance of the \(\lambda\)-calculus for computer science. If every computable function, or every algorithm, can be represented in the \(\lambda\)-calculus, then it is possible to see a program as a \(\lambda\)-term. Indeed in the years 1960 various people have recognized the \(\lambda\)-calculus as an elementary programming language. The programming languages that are or turn out to be based on the \(\lambda\)-calculus are called functional programming languages. The terminology functional comes from the fact that a program is a function, built from more elementary functions. Examples are Haskell, ML, and Miranda.

Concerning Church’s Thesis: it is not possible to formally prove it, because it is about an intuitive idea of what is computable. It is, however, possible to refute Church’s Thesis: give a function that is intuitively computable, and prove that it is not definable in the \(\lambda\)-calculus. This has not happened so far, and it is not expected to happen: a large majority of mathematicians and computer scientists believe in Church’s Thesis.

Also, there is not a precise definition of what a programming language is. It is however clear that at least elementary data types, such as natural num-
bers, booleans, paring, should be expressible. In this chapter we will see that elementary data types are definable in the pure $\lambda$-calculus.

## 4.1 Definability

What does it mean to be definable in the $\lambda$-calculus?

To investigate this question we first consider the natural numbers. A representation of the natural numbers as $\lambda$-terms should at least have the property that the representation of $n$ is different from the one of $m$ if $n$ and $m$ are different. Further, we wish to represent a natural number by a normal form, that is, by a term that does not admit further computation. Finally, we wish the representation of a natural number to be a closed term, such that there is no free variable that can be captured by putting it in a larger context.

Hence a representation of the natural numbers in the $\lambda$-calculus is an infinite sequence of different closed normal forms. This is still a rather elementary set of requirements. Usually, the representation of natural numbers will not only be used to represent natural numbers, but also to compute with them. In order to do so, our infinite sequence of closed normal forms should also be systematic in some sense.

Assume that we know how to represent the natural numbers in the $\lambda$-calculus. Let us write $[n]$ for the $\lambda$-term representing the natural number $n$. We say that a function $f : \texttt{nat} \to \texttt{nat}$ is definable in the $\lambda$-calculus if there exists a $\lambda$-term $[f]$ such that:

$[f][n] \equiv_\beta [f(n)]$

for every natural number $n$. That is: the encoding of the function $f$ applied to the encoding of the natural number $n$ is $\beta$-equivalent to the encoding of $f(n)$.

Note that the ‘equality’ used is $\beta$-equivalence, also called $\beta$-convertibility. This is the reflexive-symmetric-transitive closure of $\beta$-reduction. Terms $M$ and $N$ are $\beta$-convertible if there is a finite sequence of $\beta$-steps, either from left to right or from right to left, from $M$ to $N$. Intuitively, $M$ and $N$ represent the same object (compare for instance $3^2$ and $9$ and $16 - 7$ all representing the same object). We have for example the following:

$$(\lambda xy. x)(\lambda x. x)(\lambda xy. x) \equiv_\beta (\lambda xy. y)(\lambda xy. x)(\lambda x. x)$$

because

$$(\lambda xy. x)(\lambda x. x)(\lambda xy. x) \rightarrow_\beta (\lambda y. (\lambda x. x))(\lambda xy. x)$$

$\rightarrow_\beta \lambda x. x$

$\leftrightarrow_\beta (\lambda y. y)(\lambda x. x)$

$\leftrightarrow_\beta (\lambda xy. y)(\lambda xy. x)(\lambda x. x)$$
4.2. **BOOLEANS**

It is desirable to be able to represent the booleans *true* and *false* and the conditional in the $\lambda$-calculus. For representing the booleans as $\lambda$-terms we require, analoguously to what we discussed above about representing the natural numbers, that the representations of *true* and *false* should be two different closed normal forms. The best known representation of the booleans as $\lambda$-terms *true* and *false* is as follows:

- $\text{true} = \lambda xy. x$
- $\text{false} = \lambda xy. y$

So *true* can be seen as a function of two arguments that selects the first one (and throws away the second one), and *false* as a function that selects the second of its two arguments (and throws away the first).

We also need a conditional (if–then–else) operator. That is, we look for a $\lambda$-term with the following behaviour:

\[
\text{ite true } PQ \rightarrow_{\beta} P, \quad \text{ite false } PQ \rightarrow_{\beta} Q
\]

for arbitrary $\lambda$-terms $P$ and $Q$. From the definition of *true* and *false* it follows that for arbitrary $\lambda$-terms $P$ and $Q$ we have the following reductions:

\[
\text{true } PQ = (\lambda xy. x) PQ \rightarrow_{\beta} P
\]

and

\[
\text{false } PQ = (\lambda xy. y) PQ \rightarrow_{\beta} Q.
\]

We define the conditional *ite* as a function that, if applied to three arguments $B$, $P$, and $Q$ $\beta$-reduces to the term $BPQ$. That means that we should define *ite* as follows:

\[
\text{ite} = \lambda bxy. bx y.
\]

Then we have the following reductions:

\[
\text{ite true } PQ \rightarrow_{\beta} P, \quad \text{ite false } PQ \rightarrow_{\beta} Q.
\]

Here we imagine $B$ as a term representing a boolean, or that reduces to or is convertible to a term representing a boolean. This is however only to set our thoughts: nothing in the definition excludes $B$ having a totally different form. See also the exercises for the rewrite sequences.

It is also possible to represent operations on booleans in the $\lambda$-calculus. To start with we try to represent negation in the $\lambda$-calculus. That is, we look for a term not with the following behaviour:

\[
\text{not true } \rightarrow_{\beta} \text{false}, \quad \text{not false } \rightarrow_{\beta} \text{true}
\]
Because \texttt{true} and \texttt{false} are defined as functions of two arguments, \texttt{true} selecting (only) the first one, and \texttt{false} selecting (only) the second one, we have:

\[
\text{true} \, \text{false} \, \text{true} = (\lambda xy. x) \, \text{false} \, \text{true} \\
\rightarrow_{\beta} \text{false}
\]

and

\[
\text{false} \, \text{false} \, \text{true} = (\lambda xy. y) \, \text{false} \, \text{true} \\
\rightarrow_{\beta} \text{true}
\]

Hence negation can be represented in the $\lambda$-calculus as a function that, if applied to argument $B$, reduces to $B \, \text{false} \, \text{true}$. Following this line of thought, that means that negation can be represented in the $\lambda$-calculus by a term $\text{not}$ defined as follows:

\[
\text{not} = \lambda x. x \, \text{false} \, \text{true}.
\]

Besides negation we can also represent the binary operations conjunction and disjunction on booleans, by terms $\text{and}$ and $\text{or}$ defined as follows:

\[
\text{and} = \lambda xy. x \, y \, \text{false} \\
\text{or} = \lambda xy. x \, \text{true} \, y
\]

We omit here the proof that these terms indeed exhibit the desired behaviour; in the exercises more details can be found.

### 4.3 Pairs

Also an operator for pairing can be defined in the $\lambda$-calculus. Its definition is as follows:

\[
\pi = \lambda rmz. z \, l \, r.
\]

Then we have for arbitrary terms $P$ and $Q$ the following:

\[
\pi \, P \, Q = (\lambda rmz. z \, l \, r) \, P \, Q \\
\rightarrow_{\beta} \lambda z. z \, P \, Q.
\]

It is important that we can, from a pair of terms $P$ and $Q$, so from the $\lambda$-term $\lambda z. z \, P \, Q$, get back the separate terms $P$ or $Q$. This is done by means of a projection function. The two projection functions are represented by $\lambda$-terms $\pi_1$ and $\pi_2$ that are defined as follows:

\[
\pi_1 = \lambda u. u \, (\lambda r. l) \\
\pi_2 = \lambda u. u \, (\lambda r. r)
\]

Then we have the following reductions:

\[
\pi_1 \, (\pi \, P \, Q) \rightarrow_{\beta} P \\
\pi_2 \, (\pi \, P \, Q) \rightarrow_{\beta} Q
\]
4.4. CHURCH NUMERALS

We do not have
\[ \pi \left( \pi_1 P \right) \left( \pi_2 P \right) =_\beta P. \]

Barendregt has shown that it is not possible to define \( \lambda \)-terms \( \pi', \pi_1', \) and \( \pi_2' \) that together have the following properties:

\[ \pi_1' \left( \pi' P_1 P_2 \right) =_\beta P_1 \]
\[ \pi_2' \left( \pi' P_1 P_2 \right) =_\beta P_2 \]
\[ \pi' \left( \pi_1' P \right) \left( \pi_2' P \right) =_\beta P \]

Put differently: surjective paring is not definable in the \( \lambda \)-calculus.

4.4 Church numerals

There are different ways to represent the natural numbers in the pure \( \lambda \)-calculus. We use here the representation of natural numbers as Church numerals.

In the definition of the Church numerals we use function composition. If \( F \) and \( P \) are \( \lambda \)-terms, we write the application of \( F \) to \( F \) applied to \( P \) as \( F^2(P) \) instead of as \( F(FP) \), and the iteration of three applications of \( F \) to \( P \) as \( F^3(P) \) instead of as \( F(F(FP)) \), etcetera. In general, we write \( F^n(P) \) for iterating \( n \) times the application of \( F \) to \( P \). This is defined as follows:

\[
\begin{align*}
F^0(P) &= P, \\
F^{n+1}(P) &= F(F^n(P)).
\end{align*}
\]

For every natural number \( n \) the Church numeral \( c_n \) is defined as follows:

\[ c_n = \lambda s. \lambda z. \ s^n(z). \]

The names of the variables, although technically speaking irrelevant, are on purpose chosen to be the first letters of ‘successor’ and ‘zero’. Of course renaming the bound variables does not change anything because we work modulo \( \alpha \)-conversion. The first five Church numerals look as follows:

\[
\begin{align*}
c_0 &= \lambda s. \lambda z. z, \\
c_1 &= \lambda s. \lambda z. s z, \\
c_2 &= \lambda s. \lambda z. s (s z), \\
c_3 &= \lambda s. \lambda z. s (s (s z)), \\
c_4 &= \lambda s. \lambda z. s (s (s (s z))).
\end{align*}
\]

We can also encode the arithmetic operations. For example, there are \( \lambda \)-terms \( \text{Plus} \) and \( \text{Mul} \) representing addition and multiplication of natural numbers. That means that for all natural numbers \( n \) and \( m \) we have \( \beta \)-reductions

\[
\begin{align*}
\text{Plus} \ c_m \ c_n & \to_\beta c_{m+n} \\
\text{Mul} \ c_m \ c_n & \to_\beta c_{m \times n}
\end{align*}
\]
Successor. We start by considering a more simple case: the representation of the successor function adding one to a natural number. We define:

$$\text{Suc} = \lambda xsz. s(xsz).$$

Then we have for example the reduction

$$\text{Suc}\,c_2 = (\lambda xsz. s(xsz)) (\lambda sz. s(sz)) \rightarrow_{\beta} \lambda s'z'. s'(s'(z')) \rightarrow_{\eta\beta} \lambda s'z'. s'(s'(z')) = \lambda sz. s(sz) = \lambda sz. s^3(z) = c_3.$$

More generally, we have $\text{Suc}\,c_n \rightarrow_{\beta} c_{n+1}$:

$$\text{Suc}\,c_n = (\lambda xsz. (xs z)) (\lambda sz. s^n(z)) \rightarrow_{\beta} \lambda sz. s((\lambda s'z'. s^n(z')) s z) \rightarrow_{\eta\beta} \lambda sz. s(s^n(z)) = \lambda sz. s^{n+1}(z) = c_{n+1}.$$

That means that $\text{Suc}$ is a representation of the successor.

The idea behind this definition of $\text{Suc}$ can be explained as follows. We look for a way to transform, using $\beta$-reduction, the Church numeral $c_n = \lambda sz. s^n(z)$ to the Church numeral $c_{n+1} = \lambda sz. s^{n+1}(z)$. To start with, for arbitrary variables $s$ and $z$ we have the following:

$$c_n s z = (\lambda s'z'. s^n(z')) s z \rightarrow_{\beta} s^n(z).$$

Hence we also have

$$s(c_n s z) \rightarrow_{\beta} s(s^n(z)) = s^{n+1}(z).$$

This means that there is a $\lambda$-term having $c_n$ as subterm, and that reduces to $c_{n+1}$. This $\lambda$-term is $\lambda sz. s(c_n s z)$, and the $\beta$-reduction is as follows:

$$\lambda sz. s(c_n s z) \rightarrow_{\beta} \lambda sz. s^{n+1}(z) = c_{n+1}.$$

The last step is now to define a term that, once applied to $c_n$, reduces to the term $\lambda sz. s(c_n s z)$. We take $\lambda x. \lambda sz. s(xs z)$. This term, above already indicated by $\text{Suc}$, indeed has the behavior $\text{Suc}\,c_n \rightarrow_{\beta} c_{n+1}$ for every natural number $n$.

There are different options for representing the successor as a $\lambda$-term. Another possibility is as follows. To start with, we have for arbitrary variables $s$ and $z$ the following:

$$c_n s(z) = (\lambda s'z'. s^n(z')) s(z) \rightarrow_{\beta} s^n(s(z)).$$
4.4. CHURCH NUMERALS

The question is whether $s^n(sz) = s^{n+1}(z)$. Indeed we can, more generally, show that for every natural number $n$ and for all $\lambda$-terms $F$ and $P$ we have the following:

$$F^n(FP) = F^{n+1}(P).$$

This can be proved by induction on $n$. Now we conclude

$$c_n s(sz) \rightarrow_{\beta} s^{n+1}(z).$$

Then also

$$\lambda sz. c_n s(sz) \rightarrow_{\beta} \lambda sz. s^{n+1}(z) = c_{n+1}.$$

The last step is to present a $\lambda$-term that, once applied to $c_n$, reduces to the term $\lambda sz. c_n s(sz)$. We take $\lambda x. \lambda sz. x(sz)$. We denote this term by $\text{Suc}'$. The following reduction shows that the term $\text{Suc}'$ is indeed a good alternative definition of successor:

$$\text{Suc}' c_n = (\lambda x. \lambda sz. x(sz)) (\lambda sz'. s^n(sz')) \rightarrow_{\beta} \lambda sz. (\lambda sz'. s^n(sz')) s(sz) \rightarrow_{\beta} \lambda sz. s^n(sz) = \lambda sz. s^{n+1}(z) = c_{n+1}.$$

In the one but last step we use $F^n(FP) = F^{n+1}(P)$.

**More arithmetical operations.** The successor is a very elementary function on natural numbers. Also the more complex binary functions addition, multiplication, and exponentiation can be defined in the $\lambda$-calculus. These function are represented by the $\lambda$-terms $\text{Plus}$, $\text{Mul}$, en $\text{Exp}$ which are defined as follows:

$$\text{Plus} = \lambda xy. \lambda sz. x(syz)$$

$$\text{Mul} = \lambda xy. \lambda sz. x(syz)$$

$$\text{Exp} = \lambda xy. x(yx)$$

We can prove that these terms indeed have the desired behaviour, that is, for arbitrary natural numbers $n$ and $m$ we have the following reductions:

$$\text{Plus} c_n c_m \rightarrow_{\beta} c_{n+m}$$

$$\text{Mul} c_n c_m \rightarrow_{\beta} c_{n \times m}$$

$$\text{Exp} c_n c_m \rightarrow_{\beta} c_{n^m}$$

In the last case we need $m > 0$. We omit the proofs here.

More difficult is the definition of prececssor in the $\lambda$-calculus. First the following auxiliary definition is given:

$$\text{prefn} := \lambda fp. \pi \text{false} (\text{ite}(\pi_1 p)(\pi_2 p)(f(\pi_2 p)))$$
Then the $\lambda$-term $\text{pred}$ is defined as follows:

$$\text{pred} := \lambda x. \lambda s z. \pi_2 (x (\text{prefn} s) (\pi \text{true} z))$$

We can show that $\text{pred}$ has the desired behaviour:

$$\text{pred} c_0 \rightarrow_\beta c_0$$
$$\text{pred} c_{n+1} \rightarrow_\beta c_n$$

### 4.5 Lists

Pairs can be used to define lists. A list is seen as either the empty list, often denoted by $\text{nil}$, or an element (the head) followed by a list (the tail). The operation that adds in the front an element to a list is often called $\text{cons}$. We represent the empty list by the $\lambda$-term $\text{nil}$, and the operator $\text{cons}$ by the $\lambda$-term $\text{cons}$, defined as follows:

$$\text{nil} = \lambda x y. y$$
$$\text{cons} = \lambda h t. \lambda z. z h t$$

Some examples of lists consisting of (predefined) natural numbers:

$$\text{cons} 1 \text{ nil}$$
$$\text{cons} 1 (\text{cons} 2 \text{ nil})$$
$$\text{cons} 1 (\text{cons} 2 (\text{cons} 3 \text{ nil}))$$

The two main operations on lists are $\text{head}$, returning the first element from a non-empty list, and $\text{tail}$, returning what that list that remains if the first element is removed from a non-empty list. These operations are represented in the $\lambda$-calculus by the terms $\text{head}$ and $\text{tail}$, defined as follows:

$$\text{head} = \lambda l. l (\lambda h t. h)$$
$$\text{tail} = \lambda l. l (\lambda h t. t)$$

These terms indeed have the desired behaviour, that is, we have:

$$\text{head} (\text{cons} H T) \rightarrow_\beta H$$
$$\text{tail} (\text{cons} H T) \rightarrow_\beta T$$

Here $H$ and $T$ are arbitrary $\lambda$-terms. To set our thoughts it may be helpful to think of $H$ as the head and of $H$ of the tail of the list $\text{cons} H T$.

### 4.6 Exercises

Give all reduction steps explicitly.

1. Show that for every natural number $n$ we have the following:

$$F^n(FP) = F^{n+1}(P).$$
2. Prove that \( \text{Suc} \ c_1 \rightarrow_{\beta} c_2 \), with \( \text{Suc} \) defined as \( \lambda xsz. s \ (x \ s \ z) \).

3. Prove that \( \text{Suc'} \ c_1 \rightarrow_{\beta} c_2 \).

4. Show the following:
   (a) \( \text{Plus} \ c_1 \ c_2 \rightarrow_{\beta} c_3 \),
   (b) \( \text{Mul} \ c_1 \ c_2 \rightarrow_{\beta} c_2 \).

5. Show that \( \text{ite} \) is indeed a good representation of the conditional, that is, show that for arbitrary \( P \) and \( Q \) we have the following:
   (a) \( \text{ite} \ \text{true} \ P \ Q \rightarrow_{\beta} \ P \),
   (b) \( \text{ite} \ \text{false} \ P \ Q \rightarrow_{\beta} \ Q \).

6. Show the following:
   (a) \( \text{and} \ \text{true} \ \text{false} \rightarrow_{\beta} \text{false} \),
   (b) \( \text{or} \ \text{true} \ \text{false} \rightarrow_{\beta} \text{true} \).

7. Show that for arbitrary \( P \) and \( Q \) we have the following:
   (a) \( \pi_1 \ (\pi \ P \ Q) \rightarrow_{\beta} P \),
   (b) \( \pi_2 \ (\pi \ P \ Q) \rightarrow_{\beta} Q \).

8. Define
   \[
   \tau = \lambda x_1 x_2 x_3 z. z \ x_1 \ x_2 \ x_3.
   \]
   This \( \lambda \)-term can be used to build triples:
   \[
   \tau \ P_1 \ P_2 \ P_3 \rightarrow_{\beta} \lambda z. z \ P_1 \ P_2 \ P_3.
   \]
   Define \( \lambda \)-terms \( \tau_1 \), \( \tau_2 \) en \( \tau_3 \) representing the first, second, and third projectors. Show that they have the right behaviour, that is:
   \[
   \begin{align*}
   \tau_1 \ (\tau \ P_1 \ P_2 \ P_3) & \rightarrow_{\beta} P_1, \\
   \tau_2 \ (\tau \ P_1 \ P_2 \ P_3) & \rightarrow_{\beta} P_2, \\
   \tau_3 \ (\tau \ P_1 \ P_2 \ P_3) & \rightarrow_{\beta} P_3.
   \end{align*}
   \]

9. a. Give a specification for the operation exclusive or, notation \( \text{xor} \), that takes two booleans as input and yields \( \text{true} \) if exactly one of the inputs is \( \text{true} \)
   b. Give a \( \lambda \)-term for \( \text{xor} \).
   c. Show for one of the four possible inputs that \( \text{xor} \) satisfies the specification.

10. a. Give the definition of the pairing operator in the \( \lambda \)-calculus.
b. Give the specification of the two projection functions.

c. Give the definition of the projection function for first, and show that it satisfies its specification.

11. a. Give a λ-term for not and show that it has the desired behaviour both for input \( \text{true} = \lambda xy. x \) and for input \( \text{false} = \lambda xy. y \).

b. Give a specification of the function empty taking as input a list and returning \( \text{true} \) exactly if the input-list is empty.

c. Use nil = \( \lambda z. \text{true} \) as representation for the empty list, and the usual definition cons = \( \lambda ht.z h t \).

Give a λ-term for empty from 11b.

12. a. Give the Church numerals for 0, 1, en 2.

Give the general definition of the Church numeral \( c_n \) for \( n \) a natural number.

b. Give a λ-term iszero taking as input a Church numeral and giving back as output \( \text{true} \) if the input was the Church numeral for 0 and \( \text{false} \) if the input was a Church numeral for a natural number \( \neq 0 \).

c. Give the λ-term ite for conditional and show that it satisfies its specification.
Chapter 5

Recursion

Consider the representation of lists from the previous chapter:

\[ \text{nil} = \lambda xy.y \]
\[ \text{cons} = \lambda h t. \lambda z. z h t \]

and the representation of the operations for \textit{head} and \textit{tail} defined on lists:

\[ \text{head} = \lambda l. l (\lambda h t. h) \]
\[ \text{tail} = \lambda l. l (\lambda h t. t) \]

We now discuss how to represent the well-known function \textit{map}, that takes as input a list and a function defined on the elements of the list, and gives as output the list obtained by applying the function element-wise to the list. A definition of \textit{map} in the \(\lambda\)-calculus should satisfy the following:

\[ \text{map } F \text{ nil} = \beta \text{ nil} \]
\[ \text{map } F \text{ (cons } H T) = \beta \text{ cons } (F H) \text{ (map } F \text{ T}) \]

After some calculations we find the following ‘temporary definition’:

\[ \text{map} = \lambda fl. l (\lambda ht z. \text{cons } (f h) \text{ (map } f t)) \text{ nil} \]

This is not a real definition because \textit{map} is ‘defined’ in terms of itself; it is a recursive definition (a kind of characterizing property), an important phenomenon in mathematics and computer science. We will discuss below how recursively defined function can be represented in the \(\lambda\)-calculus using fixed point combinators.

5.1 Fixed points

A \textit{fixed point} of a function \(f\) is an element \(x\) from the domain of \(f\) for which the following holds:

\[ f(x) = x. \]
For example, 5 is a fixed point of the constant function \( x \mapsto 5 \), and 0 is a fixed point of the function \( x \mapsto x^2 \). In general a function may have zero, one or more fixed points. The function \( x \mapsto x^2 \) has besides 0 also 1 as fixed point, and the identity function \( x \mapsto x \) has as fixed points all elements in its domain.

A \( \lambda \)-term, any \( \lambda \)-term, can be seen as a function on domain the set of \( \lambda \)-terms. A fixed point of a \( \lambda \)-term \( F \) is hence a \( \lambda \)-term \( M \) for which we have the following:

\[
FM = \beta M.
\]

The combinator \( I = \lambda x.x \) has infinitely many fixed points: all \( \lambda \)-terms.

An important result in the \( \lambda \)-calculus is that every term has a fixed point. The result is even stronger: there is a method to, given a \( \lambda \)-term \( F \), construct a fixed point for \( F \). This is done using a so-called fixed point combinator.

The fixed point combinator due to Curry is the term \( Y \) defined as follows:

\[
Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))
\]

Take an arbitrary \( \lambda \)-term \( F \). Then we have:

\[
YF = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) F
\]

\[
\rightarrow_\beta (\lambda x. F (x x)) (\lambda x. F (x x))
\]

\[
\rightarrow_\beta F ((\lambda x. F (x x)) (\lambda x. F (x x)))
\]

\[
\leftrightarrow_\beta F ((\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) F)
\]

\[
= F (Y F)
\]

Hence:

\[
F (Y F) = \beta Y F
\]

That is, \( YF \) is a fixed point van \( F \). Note however that we do not have \( F (Y F) \rightarrow_\beta Y F \); here we only have conversion, no reduction.

Note further that \( YF \) admits infinite reduction sequences:

\[
F ((\lambda x. F (x x)) (\lambda x. F (x x))) \rightarrow_\beta F (F ((\lambda x. F (x x)) (\lambda x. F (x x))))
\]

The term \( Y \) is not the only fixed point combinator in the \( \lambda \)-calculus.

The following fixed point combinator is due to Turing:

\[
T = (\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y))
\]

We write \( T = tt \) with \( t = \lambda x. \lambda y. y (x x y) \). Take an arbitrary \( \lambda \)-term \( F \). We have the following:

\[
TF = (\lambda x. \lambda y. y (x x y)) (\lambda x. \lambda y. y (x x y)) F
\]

\[
\rightarrow_\beta (\lambda y. y (tt y)) F
\]

\[
\rightarrow_\beta F (tt F)
\]

\[
= F (TF)
\]

Hence:

\[
TF \rightarrow_\beta F (TF)
\]

Hence \( TF \) is a fixed point of \( F \).
5.2 Recursive functions

A fixed point combinator can be used to represent recursively defined functions in the $\lambda$-calculus. In the following we will mainly use the fixed point combinator $Y$ for this purpose. The idea is as follows. Suppose that a function $G$ is defined in terms of itself, that is, we have

$$G = \beta \ldots G \ldots$$

In the right-hand side we replace (all occurrences of) $G$ by $g$, and we write the term obtained in this way as $E$. Then we can replace the equation above by

$$G = \beta (\lambda g.E) G$$

That is: we are looking for a term $G$ that is a fixed point of the term $\lambda g.E$. Using a fixed point combinator we can find such a term: $Y (\lambda g.E)$ is a fixed point of $\lambda g.E$. Hence we can defined $G$ as

$$G = Y (\lambda g.E)$$

As an example we consider the function $\text{sum}$ that takes as input a list of natural numbers, and gives back as output the sum of the elements of the list. We use the predefined natural numbers, and the predefined addition $\text{plus}$ on those. Further, we use the operations $\text{head}$, $\text{tail}$ and $\text{empty}$ in lists, and the conditional $\text{ite}$. A temporary definition of $\text{sum}$ is as follows:

$$\text{sum} = \lambda l. \text{ite} (\text{empty} l) (0) (\text{plus} (\text{head} l) (\text{sum} (\text{tail} l)))$$

Hence the empty list is mapped to 0, and in case the input is a non-empty list, the first element is added to the result of applying $\sim$ to the tail. The term $\text{empty}$ can be defined, see the exercises. Using the fixed point operator $Y$ the definition of $\text{sum}$ can now be given as follows:

$$\text{sum} = Y (\lambda s. \lambda l. \text{ite} (\text{empty} l) (0) (\text{plus} (\text{head} l) (s (\text{tail} l))))$$

The temporary definition of $\text{map}$ is given as follows:

$$\text{map} = \lambda f l. l (\lambda h t z. \text{cons} (f h) (\text{map} f t)) \text{nil}$$

Using $Y$, we define $\text{map}$ as follows:

$$\text{map} = Y (\lambda m. \lambda f l. l (\lambda h t z. \text{cons} (f h) (m f t)) \text{nil})$$

5.3 Expressivity

The $\lambda$-calculus is a model of computation. A classical result is the equivalence of definability in the $\lambda$-calculus, Turing-computability, $\mu$-recursive.
5.4 Exercises

1. Show that for arbitrary terms \( H \) and \( T \) we have
   (a) \( \text{head} (\text{cons } H T) \rightarrow_\beta H \),
   (b) \( \text{tail} (\text{cons } H T) \rightarrow_\beta T \).

2. Reduce a bit the term \( \text{Y} (\lambda x. x) \).

3. Define \( \text{empty} = \lambda l.l (\lambda xyz. \text{false}) \text{true} \). Show that for arbitrary terms \( H \) and \( T \) we have
   (a) \( \text{empty} \text{ nil} \rightarrow_\beta \text{ true} \),
   (b) \( \text{empty} (\text{cons } H T) \rightarrow_\beta \text{ false} \).

4. Let \( \text{nil}' = \lambda z. \text{true} \) be an alternative representation of the empty list. Give the corresponding alternative definition \( \text{empty}' \) for \( \text{empty} \) and show
   (a) \( \text{empty}' \text{ nil}' \rightarrow_\beta \text{ true} \),
   (b) \( \text{empty}' (\text{cons } H T) \rightarrow_\beta \text{ false} \).

5. Use a fixed point combinator to define the function for factorial. You may use the predefined natural numbers, and representations of multiplication and subtraction.

6. Give a definition in the \( \lambda \)-calculus of a function that takes as input a finite list of natural numbers, and gives back as output the sum of those numbers. If the input is the empty list the output is 0.
   You may use a fixed point combinator, a representation for empty, addition, head, and tail.

7. Give a definition in the \( \lambda \)-calculus of the function \( \text{map} \) satisfying the following specification:

\[
\begin{align*}
\text{map } F \text{ nil} & \quad =_\beta \text{ nil} \\
\text{map } F (\text{cons } H T) & \quad =_\beta \text{ cons } (F H) (\text{map } F T)
\end{align*}
\]

You may use a fixed point combinator, and representations of \( \text{head}, \text{tail}, \) and \( \text{empty} \).

8. Give a definition in the \( \lambda \)-calculus of the function \( \text{quad} \) that takes as input a finite list of natural numbers, and gives back as output the list consisting of the squares of those numbers. The empty list is mapped to the empty list.
   You may use representations for \( \text{empty}, \text{head}, \text{tail}, \) and \( \text{square} \) (for squaring a number), and a fixed point combinator.
Chapter 6

Typed Lambda Calculus

Types are used in logic and in programming for various reasons. In 1901 Russel presented the paradox \( \{ X \mid X \notin X \} \). Early type systems are due to Russel and Whitehead, and also to Church who introduces a typed \( \lambda \)-calculus. In this chapter we consider an introduction to typed \( \lambda \)-calculus.

Types are used in programming languages to detect errors in an early stage, often at compile-time. Examples of elementary types are the type of natural numbers and the type of booleans. An example of an error detected via the typing is if one tries to add a boolean to a natural number. Also the functional programming languages Haskell and ML are typed. Typing can be static (for example for ML, Haskell, Java) or dynamic (for example for Lisp). Many typed programming languages are polymorphically typed, meaning that an expression may have more than one type. Examples are ML, Haskell, and Java. Some typed programming languages are monomorphically typed, for example Pascal. Another very expressive typing paradigm is that of dependent types. Intuitively, the output type may depend on the input type.

6.1 Types

We start by defining the set of types. We assume a set \( B \) of base types. Examples of base types are the type of natural numbers, written as \( \text{nat} \), and the type of booleans, written as \( \text{bool} \). We assume that we have more, at the moment unspecified, base types available.

Later on in this chapter we will assume besides a set of base types also a set (disjoint from the one of base types) of type variables. They are intuitively used to express "typing schemes".

Types are (here) built from base types and a binary type constructor \( \rightarrow \); a type of the form \( (A \rightarrow B) \) is said to be a function type. Intuitively, this is the type of a function with domain \( A \) and range \( B \). More concretely, \( \text{nat} \rightarrow \text{bool} \) is a function type, intuitively the type of a term that expects an input of type \( \text{nat} \) and returns an output of type \( \text{bool} \).
Formally, the set of types is inductively defined by the following two clauses:

1. A base type in $B$ is a type.
2. If $A$ and $B$ are types, then $(A \to B)$ is a type (a function type).

We write types as $A, B, C, \ldots$. We adopt the following (standard) conventions:

- We write $(A \to B \to C)$ instead of $(A \to (B \to C))$. That is, the type constructor $\to$ is associative to the right.
- Outermost parentheses are omitted.

Using these conventions we write $(A \to B) \to C$ instead of $((A \to B) \to C)$, and $A \to B \to C$ instead of $(A \to (B \to C))$.

### 6.2 Raw terms

We consider here terms that are the same as the one for the untyped $\lambda$-calculus: so a term is either a variable $x$, or an abstraction $\lambda x. M$, or an application $FP$. Note further that we work modulo $\alpha$-conversion, so with $\alpha$-equivalence classes of terms. The new element is that we distinguish terms that are typable from terms that are not. We work with typable terms only. An example of a term that will turn out not to be typable is $xx$. So a next question is: how is typability of terms defined?

Before we explain this, first a remark. An alternative approach is to consider terms that differ from the untyped ones by the annotation of the abstracted variable by a type. In that approach, we have for example the term $\lambda x: \text{nat}.x$.

Typability is defined using a formal system called a typing system. In order to defined our typing system, we first need more notions and notations.

**Declarations, environments, and judgements.** A declaration for a variable $x$ is an expression of the form

$$x : A$$

with $A$ a type.

An environment is a set (or alternatively a list) of type declarations satisfying the restriction that for every variable $x$ there is at most one declaration $x : A$. So for example $\{x : \text{nat}, x : \text{bool}\}$ is not an environment.

Since we use sets for environments, the order of declarations does not matter. Environments are written as $\Gamma, \Delta, \ldots$. If we want to stress that an environment $\Gamma$ contains a declarations $x : A$, we write instead of $\Gamma$ also $\Gamma', x : A$ with $\Gamma' = \Gamma \setminus \{x : A\}$.

A judgement is an expression of the form

$$\Gamma \vdash M : A$$

We read this as: in environment $\Gamma$ the term $M$ has type $A$. 

The typing system. We use here a typing systems \textit{à la Curry}, with terms exactly as the untyped terms as also mentioned above. An alternative is a typing system \textit{à la Church}, where the abstracted variable in a term is labelled with a type. The approach \textit{à la Curry} is closed to ML and Haskell that also use implicit types.

The typing system is a formal system defined by three derivation rules. It is used to infer judgements of the form $\Gamma \vdash M : A$. The three derivation rules are as follows:

1. The \textit{variable rule}.
   $$\Gamma, x : A \vdash x : A$$

2. The \textit{abstraction rule}.
   $$\Gamma, x : A \vdash M : B \\ \Gamma \vdash (\lambda x. M) : A \rightarrow B$$

3. The \textit{application rule}.
   $$\Gamma \vdash F : A \rightarrow B \quad \Gamma \vdash M : A \\ \Gamma \vdash (F M) : B$$

We say that a term $M$ is \textit{typable} (with type $A$) if there exists an environment $\Gamma$ and a type $A$ such that the judgment $\Gamma \vdash M : A$ is derivable. Instead of $\emptyset \vdash M : A$ we also write $\vdash M : A$.

Examples.

1. Let $A$ be an arbitrary type. We show that the combinator $I = \lambda x. x$ is typable with type $A \rightarrow A$:
   $$x : A \vdash x : A \\ \vdash \lambda x. x : A$$

2. Let $A$ and $B$ be arbitrary types. We show that the combinator $K = \lambda xy. x$ is typable with type $A \rightarrow B \rightarrow A$:
   $$x : A, y : B \vdash x : A \\ x : A \vdash \lambda y. x : B \rightarrow A \\ \vdash \lambda xy. x : A \rightarrow B \rightarrow A$$

3. Let $A$, $B$, and $C$ be arbitrary types. We show that the combinator $S = \lambda xyz. (xz)(yz)$ is typable with type $(A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$. In the derivation we use three abbreviations:
   $$\Gamma_0 = \{ x : A \rightarrow B \rightarrow C, y : A \rightarrow B, z : A \}$$
   $$\Gamma_1 = \{ x : A \rightarrow B \rightarrow C, y : A \rightarrow B \}$$
   $$\Gamma_2 = \{ x : A \rightarrow B \rightarrow C \}$$

   The typing derivation is as follows:
CHAPTER 6. TYPED LAMBDA CALCULUS

\[
\Gamma_0 \vdash x : A \rightarrow B \rightarrow C \\
\Gamma_0 \vdash z : A \\
\Gamma_0 \vdash y : A \rightarrow B \\
\Gamma_0 \vdash z : A
\]

\[
\Gamma_0 \vdash x z : B \rightarrow C \\
\Gamma_0 \vdash y z : B
\]

\[
\Gamma_0 \vdash x (y z) : C \\
\Gamma_1 \vdash \lambda z. x z (y z) : A \rightarrow C \\
\Gamma_2 \vdash \lambda y z. x z (y z) : (A \rightarrow B) \rightarrow (A \rightarrow C)
\]

\[
\vdash \lambda x y z. x z (y z) : (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)
\]

4. Type derivation:

\[
\Gamma_1 \vdash x : B \rightarrow (A \rightarrow B) \rightarrow C \\
\Gamma_1 \vdash y : B \\
\Gamma_2 \vdash y : B
\]

\[
\Gamma_1 \vdash (x y) : (A \rightarrow B) \rightarrow C \\
\Gamma_1 \vdash \lambda y. (x y) \lambda z. y : B \rightarrow C
\]

\[
\vdash \lambda x. \lambda y. (x y) \lambda z. y : (B \rightarrow (A \rightarrow B) \rightarrow C) \rightarrow B \rightarrow C
\]

We use:

\[
\Gamma_0 = \{x : B \rightarrow (A \rightarrow B) \rightarrow C\} \\
\Gamma_1 = \{x : B \rightarrow (A \rightarrow B) \rightarrow C, y : B\} \\
\Gamma_2 = \{x : B \rightarrow (A \rightarrow B) \rightarrow C, y : B, z : A\}
\]

5. Type derivation:

\[
\Gamma_1 \vdash x : A \rightarrow A \rightarrow B \\
\Gamma_1 \vdash y : A \\
\Gamma_1 \vdash y : A
\]

\[
\Gamma_1 \vdash (x y) : A \rightarrow B \\
\Gamma_1 \vdash \lambda y. (x y) : A \rightarrow B
\]

\[
\vdash \lambda x. \lambda y. (x y) : (A \rightarrow A \rightarrow B) \rightarrow A \rightarrow B
\]

We use:

\[
\Gamma_0 = \{x : A \rightarrow A \rightarrow B\} \\
\Gamma_1 = \{x : A \rightarrow A \rightarrow B, y : A\}
\]

6. Type derivation:

\[
\Gamma_2 \vdash y : A \\
\Gamma_1 \vdash \lambda z. y : B \rightarrow A
\]

\[
\Gamma_0 \vdash x : (A \rightarrow B \rightarrow A) \rightarrow C \\
\Gamma_0 \vdash \lambda y. \lambda z. y : A \rightarrow B \rightarrow A
\]

\[
\Gamma_0 \vdash (x (\lambda y. \lambda z. y)) : C \\
\Gamma_0 \vdash \lambda x. (x (\lambda y. \lambda z. y)) : ((A \rightarrow B \rightarrow A) \rightarrow C) \rightarrow C
\]
6.3. **SUBJECT REDUCTION**

We use:

\[
\begin{align*}
\Gamma_0 &= \{ x : (A \to B \to A) \to C \} \\
\Gamma_1 &= \{ x : (A \to B \to A) \to C, y : A \} \\
\Gamma_2 &= \{ x : (A \to B \to A) \to C, y : A, z : B \}
\end{align*}
\]

7. Type derivation:

\[
\begin{align*}
\Gamma_2 \vdash y : C & \quad \Gamma_3 \vdash z : B \\
\Gamma_2 \vdash x : (A \to B) \to C \to D & \quad \Gamma_2 \vdash \lambda u. z : A \to B \\
\Gamma_2 \vdash x (\lambda u. z) : C \to D & \\
\Gamma_1 \vdash \lambda x. x (\lambda u. z) y : B \to D & \\
\Gamma_0 \vdash \lambda y. \lambda z. x (\lambda u. z) y : C \to B \to D \\
\vdash \lambda x. \lambda y. \lambda z. x (\lambda u. z) y : ((A \to B) \to C \to D) \to C \to B \to D
\end{align*}
\]

We use:

\[
\begin{align*}
\Gamma_0 &= \{ x : (A \to B) \to C \to D \} \\
\Gamma_1 &= \{ x : (A \to B) \to C \to D, y : C \} \\
\Gamma_2 &= \{ x : (A \to B) \to C \to D, y : C, z : B \} \\
\Gamma_3 &= \{ x : (A \to B) \to C \to D, y : C, z : B, u : A \}
\end{align*}
\]

**Untypable terms.** In the (simply) typed \(\lambda\)-calculus we work with typable terms only. However, not all raw terms are typable. An example of a term that is not is \(\lambda x. xx\). The reason for this term not being typable is roughly that within one derivation we need for the variable \(x\) both a type \(x : A \to B\) and a type \(x : A\). Hence also the term \(\Omega\) is not typable. The fixed point combinators \(Y\) and \(T\) are not typable either.

6.3 **Subject reduction**

**\(\alpha\)-conversie.** Exactly as in the untyped \(\lambda\)-calculus also in the typed \(\lambda\)-calculus terms are considered modulo \(\alpha\)-conversion. So terms that only differ in the naming of bound variables, such as for example \(\lambda x. x\) and \(\lambda y. y\) are identified. One could wonder whether maybe the typing system would treat two \(\alpha\)-equivalent terms (so two representatives of the same equivalence class) differently. This is fortunately not the case: if a term \(M\) has type \(A\) in an environment \(\Gamma\), then every term \(M'\) that is \(\alpha\)-equivalent to \(M\) also has type \(A\) in environment \(\Gamma\). More formally:

\[
\begin{align*}
\Gamma \vdash M : A \\
M \equiv_\alpha M' \\
\end{align*}
\]

\[
\Rightarrow \Gamma \vdash M' : A
\]
The \(\beta\)-reduction relation for typed \(\lambda\)-calculus is defined as for the untyped \(\lambda\)-calculus. An important result, called subject reduction states that the type of a term is invariant under \(\beta\)-reduction. More formally:

\[
\Gamma \vdash M : A \\
M \to_\beta M' \}
\Rightarrow \Gamma \vdash M' : A
\]

Note that we have \(\{ y : A \} \vdash y : A\) and \((\lambda x. y) \Omega \to_\beta y\), but not \(\{ y : A \} \vdash (\lambda x. y) \Omega : A\) because \(\Omega\) is not typable. That is: types are not preserved under \(\beta\)-expansion (which is the inverse of reduction).

### 6.4 Termination

The untyped \(\lambda\)-calculus has terms without normal form: for example \(\Omega\) is such a term. Also, there are terms having a normal form but also admitting an infinite reduction sequence. An example of such a term is \((\lambda x. y) \Omega\). In the simply typed \(\lambda\)-calculus, we do not have such terms. We hebben het volgende belangrijke resultaat:

All simply typed \(\lambda\)-terms are terminating.

So if \(\Gamma \vdash M : A\), for certain \(\Gamma\) and certain \(A\), then all reduction sequences starting in \(M\) are finite. There are many nice proofs of this property.

The implication the other way around (if a term is terminating then it is simply typable) is not true: \(\lambda x. x x\) is an example of a term that is terminating but not simply typable. This should not be surprising, given that termination is undecidable, but typability is decidable.

Just like in the untyped \(\lambda\)-calculus (and by the same proof) the simply typed \(\lambda\)-calculus is confluent. As a consequence, every term has at most one normal form. Together with termination, this yields that every simply typed \(\lambda\)-term has exactly one normal form.

Hence the simply typed \(\lambda\)-calculus is not Turing complete.

### 6.5 Decidability

It is decidable whether a term is typable. That is, there is an algorithm that takes as input a closed term \(M\) and gives back as output a type \(A\) such that \(\vdash M : A\) in case \(M\) is typable, and that returns failure if \(M\) is not typable.

This can be used to decide typability of a term that is not closed. Let \(P\) be a term with free variables \(x_1, \ldots, x_n\). Consider the term \(M = \lambda x_1 \ldots x_n. P\). Then \(M\) is closed. We feed it as input to the algorithm mentioned above. In case this yields failure, then \(M\) is not typable. As a consequence, \(P\) is not typable. In case a type for \(M\) is found, then this type has the form \(A_1 \to \ldots \to A_n \to B\). We then conclude that the term \(P\) has type \(B\) in the environment \(\{ x_1 : A_1, \ldots, x_n : A_n \}\), that is, \(\{ x_1 : A_1, \ldots, x_n : A_n \} \vdash P : B\).
6.6 Polymorphism

The identity function $\lambda x. x$ is an example of a function that is typable with different types. We have for example: $\vdash \lambda x. x : \text{nat} \rightarrow \text{nat}$, but also $\vdash \lambda x. x : \text{bool} \rightarrow \text{bool}$, and $\vdash \lambda x. x : (\text{nat} \rightarrow \text{bool}) \rightarrow (\text{nat} \rightarrow \text{bool})$. This illustrates a form of polymorphism: one term can have more than one different types. However, all its types have the same structure.

There are different ways to express polymorphism in a typing system. We consider here a tiny bit of polymorphism à la ML. We then consider as basic types both constants, think of nat and bool, and also type variables, written as $A_i$. As before, function types are built using $\rightarrow$.

We can substitute a type for a type variable. This is defined as follows:

$$a_i[a_1 := A_1 \ldots a_n := A_n] = A_i$$
$$b[a_1 := A_1 \ldots a_n := A_n] = b$$
$$k[a_1 := A_1 \ldots a_n := A_n] = k$$
$$(B \rightarrow B')[a_i := A_i] = (B[a_i := A_i]) \rightarrow (B'[a_i := A_i])$$

Here $k$ stands for a base type, and $b$ is a type variable different from all $a_i$.

The type variables are used to express the notion of most general type, also called principle type. The most general type for the identity is $a \rightarrow a$ with $a$ a type variable. All other types for the identity can be obtained by substitution for $a$ a certain type. In general, a type $A$ is said to be more general than a type $B$, notation $A \preceq B$, if $A[a_1 := A_1 \ldots a_n := A_n] = B$ for some substitution $[a_1 := A_1 \ldots a_n := A_n]$. We have for example the following:

$$a \rightarrow b \preceq b \rightarrow a$$
$$a \rightarrow a \preceq \text{nat} \rightarrow \text{nat}$$
$$a \rightarrow \text{nat} \preceq (b \rightarrow b) \rightarrow \text{nat}$$

The following is not true: $a \rightarrow a \preceq \text{nat} \rightarrow \text{bool}$.

Every typable term has a most general type. More precisely: if $\Gamma \vdash M : A$, then there exists a type $B$ such that $\Gamma \vdash M : B$ and such that for all $B'$ with $\Gamma \vdash M : B'$ we have $B \preceq B'$. The proof is omitted here. The kernel of the proof is the algorithm called the Hindley-Milner algorithm which takes as input a $\lambda$-term and gives back as output either the most general (or principle) type for the input-term, or failure in case the input-term is not typable. For an elementary $\lambda$-term it is often quite easy to see intuitively what is its most general type.

6.7 Exercises

1. Are the following terms typable? If yes, give the types of the free variables. If no, indicate why not.

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

(b) $\lambda x. f x x$,
(c) \((\lambda x. y) (z z)\),
(d) \((\lambda x. \lambda y. f (x y)) z z\).

2. Give for each of the following types an closed inhabitant in \(\beta\)-normal form, that is, a closed \(\beta\)-normal form of that type:

(a) \(\text{nat} \rightarrow \text{nat}\)
  Answer: \(\lambda x. x\).
(b) \(\text{nat} \rightarrow \text{bool} \rightarrow \text{nat}\),
(c) \(\text{nat} \rightarrow \text{bool} \rightarrow \text{bool}\),
(d) \(\text{nat} \rightarrow (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat}\).

3. Give type derivations for the following judgements:

(a) \(\vdash \text{I}: (\text{nat} \rightarrow \text{nat}) \rightarrow (\text{nat} \rightarrow \text{nat})\),
(b) \(\vdash \text{K}: \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}\),
(c) \(\vdash \text{KI}: B \rightarrow (C \rightarrow C)\),
(d) \(\{ f : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} \} \vdash \lambda xy. f x y : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}\),
(e) \(\vdash \text{S}: (\text{nat} \rightarrow \text{nat} \rightarrow \text{bool}) \rightarrow (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \rightarrow \text{bool}\),
(f) \(\vdash \text{SK}: (A \rightarrow B) \rightarrow A \rightarrow A\),
(g) \(\vdash \text{SKK}: A \rightarrow A\),
(h) \(\{ g : \text{bool} \rightarrow \text{nat}, f : \text{nat} \rightarrow \text{bool} \} \vdash \lambda x. g (f x) : \text{nat} \rightarrow \text{nat}\).

4. Show that the following terms are typable (give a derivation):

(a) \(\lambda x y z. x y z\),
(b) \(\lambda x y z. x (y z)\),
(c) \(\lambda f x. f x x\),
(d) \(\lambda x. \lambda y. x y\),
(e) \(\lambda x. \lambda y. \lambda z. y (\lambda u. x)\),
(f) \(\lambda x. \lambda y. x (x y)\),
(g) \(\lambda x. \lambda y. \lambda z. ((\lambda u. y) x)\),
(h) \(\lambda x. \lambda y. x (\lambda u. y)\),
(i) \(\lambda x. \lambda y. (\lambda u. y) x\).
(j) \(\lambda f. f (\lambda x. x)\)

5. How can the Church numerals be types?
Bibliography


