Chapter 1

Type-free λ-calculus

The λ-*calculus* is a model of computation. It was introduced a few years before another such model, *Turing machines*. With the latter, computation is expressed by reading from and writing to a tape, and performing actions depending on the contents of the tape. Turing machines resemble programs in imperative programming languages, like Java or C.

In contrast, in λ-calculus one is concerned with functions, and these may both take other functions as arguments, and return functions as results. In programming terms, λ-calculus is an extremely simple higher-order, functional programming language.

In this chapter we only cover the *type-free* or *untyped* λ-calculus. Later we introduce several variants where λ-terms are categorized into various *types*.

1.1. A gentle introduction

Computation in λ-calculus is expressed with λ-*terms*. These are similar to the nameless function notation $n \mapsto n^2$ used in mathematics. However, a mathematician employs the latter notation to denote functions as mathematical objects (defined as sets of pairs). In contrast, λ-terms are formal expressions (strings) which, intuitively, express functions and applications of functions in a pure form. Thus, a λ-term is one of the following:

- a *variable*;
- an *abstraction* $\lambda x M$, where x is a variable and M is a λ-term;
- an *application* MN (of M to N), where M and N are λ-terms.

In an abstraction $\lambda x M$, the variable x represents the function argument (or *formal parameter*), and it may occur in the *body* M of the function, but it does not have to. In an application MN there is no restriction on the shape of the *operator* M or the *argument* N; both can be arbitrary λ-terms.

For instance, the λ-term $\mathbf{I} = \lambda x\, x$ intuitively denotes a function that maps any argument to itself, i.e. the identity function. As another example, $\mathbf{K} = \lambda x \lambda y\, x$ represents a function mapping any argument x to the constant function that always returns x. Finally, $\mathbf{IK}$ expresses application of the function $\mathbf{I}$ to the argument $\mathbf{K}$.

In mathematics we usually write the application of a function, say f, to an argument, say 4, with the argument in parentheses: $f(4)$. In the λ-calculus we would rather write this as $f4$. The use of parentheses cannot be entirely eliminated though. For instance, the notation λxxy would be ambiguous, and we should instead write either $(\lambda xx)y$ if we mean an application of $\mathbf{I}$ to y, or $\lambda x(xy)$ to denote an abstraction on x with the body xy. In the latter case, it is customary to use *dot notation*, i.e. to write $\lambda x.xy$ instead. Similarly we may need parentheses to disambiguate applications; for instance, $\mathbf{I(KK)}$ expresses application of $\mathbf{I}$ to $\mathbf{KK}$

If λxM denotes a function, and N denotes an argument, the "value" of the application $(\lambda xM)N$ can be calculated by substituting N for x in M. The result of such a substitution is denoted by $M[x := N]$, and we formalize the calculation by the β-reduction rule: $(\lambda xM)N \to_\beta M[x := N]$. For instance,

$$(\mathbf{IK})z = ((\lambda xx)\mathbf{K})z \to_\beta x[x := \mathbf{K}]z = \mathbf{K}z = (\lambda y \lambda xy)z \to_\beta \lambda x z.$$

This process of calculating the value of an expression is similar to common practice in mathematics; if $f(n) = n^2$, then $f(4) = 4^2$, and we get from the application $f(4)$ to the result 4^2 by substituting 4 for n in the body of the definition of f. A programming language analogue is the *call-by-name* parameter passing mechanism, where the formal parameter of a procedure is replaced throughout by the actual parameter expression.

The variable x in a λ-abstraction λxM is *bound* (or *local*) within M in much the same way a formal parameter of a procedure is considered local within that procedure. In contrast, a variable y without a corresponding abstraction is called *free* (or *global*) and is similar to a global variable in most programming languages. Thus, x is bound and y is free in $\lambda x.xy$.

Some confusion may arise when we use the same name for bound and free variables. For example, in $x(\lambda x.xy)$, there are obviously two different x's: the free (global) x, and the bound (local) x, which is "shadowing" the free one in the body. If we instead consider the λ-term $x(\lambda z.zy)$, there is no confusion. As another example of confusion, $(\lambda x.xy)[y := x]$ should replace y in $(\lambda x.xy)$ by a free variable x, but $\lambda x.xx$ is not the desired result. In the latter term we have lost the distinction between the formal parameter x and the free variable x (the free variable has been *captured* by the lambda). If we use a bound variable z, the confusion disappears: $(\lambda z.zy)[y := x] = \lambda z.zx$.

A local variable of a procedure can always be renamed without affecting the meaning of the program. Similarly, in λ-calculus we do not care about the names of bound variables; the λ-terms $\lambda x x$ and $\lambda y y$ both denote the identity function. Because of this, it is usually assumed that *terms that differ only in their bound variables are identified.* This gives the freedom to choose bound variables so as to avoid any confusion, e.g. variable capture.

1.2. Pre-terms and λ-terms

We now define the notion of a *pre-term* and introduce λ-terms as equivalence classes of pre-terms. The section is rather dull, but necessary to make our formalism precise. However, to understand most of the book, the informal understanding of λ-terms of the preceding section suffices.

1.2.1. DEFINITION. Let Υ denote a countably infinite set of symbols, henceforth called *variables* (also *object variables* or *λ-variables* when other kinds of variables may cause ambiguity). We define the notion of a *pre-term* by induction as follows:

- Every variable is a pre-term.
- If M, N are pre-terms, then (MN) is a pre-term.
- If x is a variable and M is a pre-term, then $(\lambda x M)$ is a pre-term.

The set of all pre-terms is denoted by Λ^-.

REMARK. The definition can be summarized by the following *grammar*:

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

In the remainder of the book, we will often use this short style of definition.

Pre-terms, as defined above, are fully parenthesized. As the pre-term $(\lambda f((\lambda u(f(uu)))(\lambda v(f(vv)))))$ demonstrates, the heavy use of parentheses is rather cumbersome. We therefore introduce some notational conventions, which are used informally whenever no ambiguity or confusion can arise.

1.2.2. CONVENTION.

(i) The outermost parentheses in a term are omitted.

(ii) Application associates to the left: $((PQ)R)$ is abbreviated (PQR).

(iii) Abstraction associates to the right: $(\lambda x(\lambda y P))$ is abbreviated $(\lambda x \lambda y P)$.

(iv) A sequence of abstractions $(\lambda x_1(\lambda x_2 \ldots (\lambda x_n P)\ldots))$ can be written as $(\lambda x_1 x_2 \ldots x_n.P)$, in which case the outermost parentheses in P (if any) are usually omitted.[1]

[1] The dot represents a left parenthesis whose scope extends as far to the right as possible.

EXAMPLE.

- $(\lambda v(vv))$ may be abbreviated $\lambda v(vv)$ by (i).
- $(((\lambda xx)(\lambda yy))(\lambda zz))$ may be abbreviated $(\lambda xx)(\lambda yy)(\lambda zz)$ by (i), (ii).
- $(\lambda x(\lambda y(xy)))$ is written $\lambda x\lambda y(xy)$ by (i), (iii) or as $\lambda xy.xy$ by (i), (iv).
- $(\lambda f((\lambda u(f(uu)))(\lambda v(f(vv)))))$ is written $\lambda f.(\lambda u.f(uu))(\lambda v.f(vv))$.

1.2.3. DEFINITION. Define the set $\mathrm{FV}(M)$ of *free variables of M* as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda xP) & = & \mathrm{FV}(P) - \{x\}; \\ \mathrm{FV}(PQ) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

EXAMPLE. Let x, y, z be distinct variables; then $\mathrm{FV}((\lambda xx)(\lambda y.xyz)) = \{x, z\}$ There are two *occurrences* of x: one under λx and one under λy. An occurrence of x in M is called *bound* if it is in a part of M with shape λxL, and *free* otherwise. Then $x \in \mathrm{FV}(M)$ iff there is a free occurrence of x in M.

We now define *substitution* of pre-terms. It will only be defined when no variable is captured as a result of the substitution.

1.2.4. DEFINITION. The *substitution of N for x in M*, written $M[x := N]$, is defined iff no free occurrence of x in M is in a part of M with form λyL, where $y \in \mathrm{FV}(N)$. In such cases $M[x := N]$ is given by:[2]

$$\begin{array}{rcl} x[x := N] & = & N; \\ y[x := N] & = & y, \text{ if } x \neq y; \\ (PQ)[x := N] & = & P[x := N]\, Q[x := N]; \\ (\lambda xP)[x := N] & = & \lambda xP; \\ (\lambda yP)[x := N] & = & \lambda yP[x := N], \text{ if } x \neq y. \end{array}$$

REMARK. In the last clause, $y \notin \mathrm{FV}(N)$ or $x \notin \mathrm{FV}(P)$.

1.2.5. LEMMA.

(i) *If $x \notin \mathrm{FV}(M)$ then $M[x := N]$ is defined, and $M[x := N] = M$.*

(ii) *If $M[x := N]$ is defined then $y \in \mathrm{FV}(M[x := N])$ iff either $y \in \mathrm{FV}(M)$ and $x \neq y$ or both $y \in \mathrm{FV}(N)$ and $x \in \mathrm{FV}(M)$.*

(iii) *The substitution $M[x := x]$ is defined and $M[x := x] = M$.*

(iv) *If $M[x := y]$ is defined, then $M[x := y]$ is of the same length as M.*

[2] In our meta-notation, substitution binds stronger than anything else, so in the third clause the rightmost substitution applies to Q, not to $P[x := N]\, Q$.

PROOF. Induction on M. As an example we show (i) in some detail. It is clear that $M[x := N]$ is defined. To show that $M[x := N] = M$ consider the following cases. If M is a variable y, then we must have $y \neq x$, and $y[x := N] = y$. If $M = PQ$ then $x \notin \mathrm{FV}(P)$ and $x \notin \mathrm{FV}(Q)$, so by the induction hypothesis $P[x := N] = P$ and $Q[x := N] = Q$. Then also $(PQ)[x := N] = P[x := N]\,Q[x := N] = PQ$. Finally, if M is an abstraction, we may have either $M = \lambda xP$ or $M = \lambda yP$, where $x \neq y$. In the former case, $(\lambda xP)[x := N] = \lambda xP$. In the latter case, we have $x \notin \mathrm{FV}(P)$, so by the induction hypothesis $(\lambda yP)[x := N] = \lambda yP[x := N] = \lambda yP$. □

1.2.6. LEMMA. *Assume that $M[x := N]$ is defined, and both $N[y := L]$ and $M[x := N][y := L]$ are defined, where $x \neq y$. If $x \notin \mathrm{FV}(L)$ or $y \notin \mathrm{FV}(M)$ then $M[y := L]$ is defined, $M[y := L][x := N[y := L]]$ is defined, and*

$$M[x := N][y := L] = M[y := L][x := N[y := L]]. \qquad (*)$$

PROOF. Induction on M. The main case is when $M = \lambda zQ$, for $z \notin \{x, y\}$. By the assumptions

(i) $x \notin \mathrm{FV}(L)$ or $y \notin \mathrm{FV}(Q)$;

(ii) $z \notin \mathrm{FV}(N)$ or $x \notin \mathrm{FV}(Q)$;

(iii) $z \notin \mathrm{FV}(L)$ or $y \notin \mathrm{FV}(Q[x := N])$.

For the "defined" part, it remains, by the induction hypothesis, to show:

- $z \notin \mathrm{FV}(L)$ or $y \notin \mathrm{FV}(Q)$;
- $z \notin \mathrm{FV}(N[y := L])$ or $x \notin \mathrm{FV}(Q[y := L])$.

For the first property, if $z \in \mathrm{FV}(L)$, then $y \notin \mathrm{FV}(Q[x := N])$, so $y \notin \mathrm{FV}(Q)$. For the second property, assume $x \in \mathrm{FV}(Q[y := L])$. From (i) we have $x \in \mathrm{FV}(Q)$, thus $z \notin \mathrm{FV}(N)$ by (ii). Therefore $z \in \mathrm{FV}(N[y := L])$ could only happen when $y \in \mathrm{FV}(N)$ and $z \in \mathrm{FV}(L)$. Together with $x \in \mathrm{FV}(Q)$ this contradicts (iii). Now $(*)$ follows from the induction hypothesis. □

A special case of the lemma is $M[x := y][y := L] = M[x := L]$, if the substitutions are defined and $y \notin \mathrm{FV}(M)$.

1.2.7. LEMMA. *If $M[x{:=}y]$ is defined and $y \notin \mathrm{FV}(M)$ then $M[x{:=}y][y{:=}x]$ is defined, and $M[x{:=}y][y{:=}x] = M$.*

PROOF. By induction with respect to M one shows that the substitution is defined. The equation follows from Lemmas 1.2.5(iii) and 1.2.6. □

The next definition formalizes the idea of identifying expressions that "differ only in their bound variables."

1.2.8. DEFINITION. The relation $=_\alpha$ (*α-conversion*) is the least (i.e. smallest) transitive and reflexive relation on Λ^- satisfying the following.

- If $y \notin \mathrm{FV}(M)$ and $M[x := y]$ is defined then $\lambda x M =_\alpha \lambda y. M[x := y]$.
- If $M =_\alpha N$ then $\lambda x M =_\alpha \lambda x N$, for all variables x.
- If $M =_\alpha N$ then $MZ =_\alpha NZ$.
- If $M =_\alpha N$ then $ZM =_\alpha ZN$.

EXAMPLE. Let x, y be different variables. Then $\lambda xy.xy =_\alpha \lambda yx.yx$, but $\lambda x.xy \neq_\alpha \lambda y.yx$.

By Lemma 1.2.7 the relation $=_\alpha$ is symmetric, so we easily obtain:

1.2.9. LEMMA. *The relation of α-conversion is an equivalence relation.*

Strictly speaking, the omitted proof of Lemma 1.2.9 should go by induction with respect to the definition of $=_\alpha$. We prove the next lemma in more detail, to demonstrate this approach.

1.2.10. LEMMA. *If $M =_\alpha N$ then* $\mathrm{FV}(M) = \mathrm{FV}(N)$.

PROOF. Induction on the definition of $M =_\alpha N$. If $M =_\alpha N$ follows from transitivity, i.e. $M =_\alpha L$ and $L =_\alpha N$, for some L, then by the induction hypothesis $\mathrm{FV}(M) = \mathrm{FV}(L)$ and $\mathrm{FV}(L) = \mathrm{FV}(N)$. If $M = N$ (i.e. $M =_\alpha N$ by reflexivity) then $\mathrm{FV}(N) = \mathrm{FV}(M)$. If $M = \lambda x P$ and $N = \lambda y.P[x := y]$, where $y \notin \mathrm{FV}(P)$ and $P[x := y]$ is defined, then by Lemma 1.2.5 we have $\mathrm{FV}(M) = \mathrm{FV}(P) - \{x\} = \mathrm{FV}(P[x := y]) - \{y\} = \mathrm{FV}(N)$. If $M = \lambda x P$ and $N = \lambda x Q$, where $P =_\alpha Q$, then by the induction hypothesis $\mathrm{FV}(P) = \mathrm{FV}(Q)$, so $\mathrm{FV}(M) = \mathrm{FV}(N)$. If $M = PZ$ and $N = QZ$, or $M = ZP$ and $N = ZQ$, where $P =_\alpha Q$, then we use the induction hypothesis. □

1.2.11. LEMMA. *If $M =_\alpha M'$ and $N =_\alpha N'$ then $M[x := N] =_\alpha M'[x := N']$, provided both sides are defined.*

PROOF. By induction on the definition of $M =_\alpha M'$. If $M = M'$ then proceed by induction on M. The only other interesting case is when we have $M = \lambda z P$ and $M' = \lambda y.P[z := y]$, where $y \notin \mathrm{FV}(P)$, and $P[z := y]$ is defined. If $x = z$, then $x \notin \mathrm{FV}(M) = \mathrm{FV}(M')$ by Lemma 1.2.10. Hence $M[x := N] = M =_\alpha M' = M'[x := N']$ by Lemma 1.2.5. The case $x = y$ is similar. So assume $x \notin \{y, z\}$. Since $M[x := N]$ is defined, $x \notin \mathrm{FV}(P)$ or $z \notin \mathrm{FV}(N)$. In the former case $M[x := N] = \lambda z P$ and $x \notin \mathrm{FV}(P[z := y])$, so $M'[x := N'] = \lambda y.P[z := y] =_\alpha \lambda z.P$.

It remains to consider the case when $x \in \mathrm{FV}(P)$ and $z \notin \mathrm{FV}(N)$. Since $M'[x := N'] = (\lambda y.P[z := y])[x := N']$ is defined, we have $y \notin \mathrm{FV}(N')$, and thus also $y \notin \mathrm{FV}(P[x := N'])$. By Lemma 1.2.6 it then follows that $M'[x:=N'] = \lambda y.P[z:=y][x:=N'] = \lambda y.P[x:=N'][z:=y] =_\alpha \lambda z.P[x:=N']$. By the induction hypothesis $\lambda z.P[x:=N'] =_\alpha \lambda z.P[x:=N] = M[x:=N]$. □

1.2.12. LEMMA.

(i) *For all M, N and x, there exists an M' such that $M =_\alpha M'$ and the substitution $M'[x := N]$ is defined.*

(ii) *For all pre-terms M, N, P, and all variables x, y, there exist M', N' such that $M' =_\alpha M$, $N' =_\alpha N$, and the substitutions $M'[x := N']$ and $M'[x := N'][y := P]$ are defined.*

PROOF. Induction on M. The only interesting case in part (i) is $M = \lambda y P$ and $x \neq y$. Let z be a variable different from x, not free in N, and not occurring in P at all. Then $P[y := z]$ is defined. By the induction hypothesis applied to $P[y := z]$, there is a P' with $P' =_\alpha P[y := z]$ and such that the substitution $P'[x := N]$ is defined. Take $M' = \lambda z P'$. Then $M' =_\alpha M$ and $M'[x:=N] = \lambda z.P'[x:=N]$ is defined. The proof of part (ii) is similar. □

1.2.13. LEMMA.

(i) *If $MN =_\alpha R$ then $R = M'N'$, where $M =_\alpha M'$ and $N =_\alpha N'$.*

(ii) *If $\lambda x P =_\alpha R$, then $R = \lambda y Q$, for some Q, and there is a term P' with $P =_\alpha P'$ such that $P'[x := y] =_\alpha Q$.*

PROOF. Part (i) is by induction with respect to the definition of $MN =_\alpha R$. Part (ii) follows in a similar style. The main case in the proof is transitivity. Assume $\lambda x P =_\alpha R$ follows from $\lambda x P =_\alpha M$ and $M =_\alpha R$. By the induction hypothesis, we have $M = \lambda z N$ and $R = \lambda y Q$, and there are $P' =_\alpha P$ and $N' =_\alpha N$ such that $P'[x := z] =_\alpha N$ and $N'[z := y] =_\alpha Q$. By Lemma 1.2.12(ii) there is $P'' =_\alpha P$ with $P''[x := z]$ and $P''[x := z][z := y]$ defined. Then by Lemma 1.2.11, we have $P''[x := z] =_\alpha N =_\alpha N'$ and thus also $P''[x := z][z := y] =_\alpha Q$. And $P''[x := z][z := y] = P''[x := y]$ by Lemmas 1.2.5(iii) and 1.2.6. (Note that $z \notin \mathrm{FV}(P'')$ or $x = z$.) □

We are ready to define the true objects of interest: λ-terms.

1.2.14. DEFINITION. Define the set Λ of *λ-terms* as the quotient set of $=_\alpha$:

$$\Lambda \quad = \quad \{[M]_\alpha \mid M \in \Lambda^-\},$$

where[3] $[M]_\alpha = \{N \in \Lambda^- \mid M =_\alpha N\}$.

[3]For simplicity we write $[M]_\alpha$ instead of $[M]_{=_\alpha}$.

EXAMPLE. Thus, for every variable x, the string λxx is a pre-term, while $\mathbf{I} = [\lambda xx]_\alpha = \{\lambda xx, \lambda yy, \ldots\}$, where $x, y, \ldots$ are all the variables, is a λ-term.

WARNING. The notion of a pre-term and the explicit distinction between pre-terms and λ-terms are not standard in the literature. Rather, it is customary to call our pre-terms λ-terms and remark that "α-equivalent terms are identified" (cf. the preceding section).

We can now "lift" the notions of free variables and substitution.

1.2.15. DEFINITION. The free variables $\mathrm{FV}(M)$ of a λ-term M are defined as follows. Let $M = [M']_\alpha$. Then

$$\mathrm{FV}(M) = \mathrm{FV}(M'). \tag{$*$}$$

If $\mathrm{FV}(M) = \varnothing$ then we say that M is *closed* or that it is a *combinator*.

Lemma 1.2.10 ensures that any choice of M' yields the same result.

1.2.16. DEFINITION. For λ-terms M and N, we define $M[x := N]$ as follows. Let $M = [M']_\alpha$ and $N = [N']_\alpha$, where $M'[x := N']$ is defined. Then

$$M[x := N] = [M'[x := N']]_\alpha.$$

Here Lemma 1.2.11 ensures that any choice of M' and N' yields the same result, and Lemma 1.2.12 guarantees that suitable M' and N' can be chosen.

The term formation operations can themselves be lifted.

1.2.17. NOTATION. Let P and Q be λ-terms, and let x be a variable. Then PQ, λxP, and x denote the following unique λ-terms:

$$\begin{array}{lcll} PQ & = & [P'Q']_\alpha\,, & \text{where } [P']_\alpha = P \text{ and } [Q']_\alpha = Q; \\ \lambda xP & = & [\lambda xP']_\alpha\,, & \text{where } [P']_\alpha = P; \\ x & = & [x]_\alpha. & \end{array}$$

Using this notation, we can show that the equations defining free variables and substitution for pre-terms also hold for λ-terms. We omit the easy proofs.

1.2.18. LEMMA. *The following equations are valid:*

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda xP) & = & \mathrm{FV}(P) - \{x\}; \\ \mathrm{FV}(PQ) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

1.2.19. LEMMA. *The following equations on λ-terms are valid:*

$$\begin{array}{r} x[x := N] = N; \\ y[x := N] = y, \;\; \textit{if } x \neq y; \\ (PQ)[x := N] = P[x := N]\; Q[x := N]; \\ (\lambda yP)[x := N] = \lambda y.P[x := N], \end{array}$$

where $x \neq y$ and $y \notin \mathrm{FV}(N)$ in the last clause.

The next lemma "lifts" a few properties of pre-terms to the level of terms.

1.2.20. LEMMA. *Let P, Q, R, L be λ-terms. Then*

(i) $\lambda x P = \lambda y.P[x := y]$, *if* $y \notin \mathrm{FV}(P)$.

(ii) $P[x := Q][y := R] = P[y := R][x := Q[y := R]]$, *if* $y \neq x \notin \mathrm{FV}(R)$.

(iii) $P[x := y][y := Q] = P[x := Q]$, *if* $y \notin \mathrm{FV}(P)$.

(iv) *If* $PQ = RL$ *then* $P = R$ *and* $Q = L$.

(v) *If* $\lambda y P = \lambda z Q$, *then* $P[y := z] = Q$ *and* $Q[z := y] = P$.

PROOF. An easy consequence of Lemmas 1.2.6 and 1.2.12–1.2.13. □

From now on, expressions involving abstractions, applications, and variables are always understood as λ-terms, as defined in Notation 1.2.17. In particular, with the present machinery at hand, we can formulate definitions by induction on the structure of λ-terms, rather than first introducing the relevant notions for pre-terms and then lifting. The following definition is the first example of this; its correctness is established in Exercise 1.3.

1.2.21. DEFINITION. For $M, \vec{N} \in \Lambda$ and distinct variables $\vec{x} \in \Upsilon$, the *simultaneous substitution of $\vec{N}$ for $\vec{x}$ in M* is the term $M[\vec{x} := \vec{N}]$, such that

$$\begin{aligned}
x_i[\vec{x} := \vec{N}] &= N_i; \\
y\,[\vec{x} := \vec{N}] &= y, \quad \text{if } y \neq x_i, \text{ for all } i; \\
(PQ)[\vec{x} := \vec{N}] &= P[\vec{x} := \vec{N}]Q[\vec{x} := \vec{N}]; \\
(\lambda y P)[\vec{x} := \vec{N}] &= \lambda y.P[\vec{x} := \vec{N}],
\end{aligned}$$

where, in the last clause, $y \neq x_i$ and $y \notin \mathrm{FV}(N_i)$, for all i.

OTHER SYNTAX. In this book we define many different languages (logics and λ-calculi) with various binding operators (e.g. quantifiers). Expressions (terms, formulas, types etc.) that differ only in their bound variables are always identified as we just did it in the untyped λ-calculus. However, in order not to exhaust the reader, we generally present the syntax in a slightly informal manner, thus avoiding the explicit introduction of "pre-expressions."

In all such cases, however, we actually have to deal with equivalence classes of some α-conversion relation, and a precise definition of the syntax must take this into account. We believe that the reader is able in each case to reconstruct all missing details of such a definition.

1.3. Reduction

1.3.1. DEFINITION. A relation $\succ$ on Λ is *compatible* iff it satisfies the following conditions for all $M, N, Z \in \Lambda$.

(i) If $M \succ N$ then $\lambda x M \succ \lambda x N$, for all variables x.

(ii) If $M \succ N$ then $MZ \succ NZ$.

(iii) If $M \succ N$ then $ZM \succ ZM$.

1.3.2. DEFINITION. The least compatible relation $\to_\beta$ on Λ satisfying

$$(\lambda x.P)Q \to_\beta P[x := Q],$$

is called *β-reduction*. A term of form $(\lambda x P)Q$ is called a *β-redex,* and the term $P[x := Q]$ is said to arise by *contracting* the redex. A λ-term M is in *β-normal form* (notation $M \in \mathrm{NF}_\beta$) iff there is no N such that $M \to_\beta N$, i.e. M does not contain a β-redex.

1.3.3. DEFINITION.

(i) The relation $\twoheadrightarrow_\beta$ (*multi-step β-reduction*) is the transitive and reflexive closure of $\to_\beta$. The transitive closure of $\to_\beta$ is denoted by $\twoheadrightarrow_\beta^+$.

(ii) The relation $=_\beta$ (called *β-equality* or *β-conversion*) is the least equivalence relation containing $\to_\beta$.

(iii) A *β-reduction sequence* is a finite or infinite sequence

$$M_0 \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots$$

1.3.4. REMARK. The above notation applies in general: for any relation $\to_\circ$, the symbol $\twoheadrightarrow_\circ^+$ (respectively $\twoheadrightarrow_\circ$) denotes the transitive (respectively transitive and reflexive) closure of $\to_\circ$, and the symbol $=_\circ$ is used for the corresponding equivalence. We often omit β (or in general $\circ$) from such notation when no confusion arises, with one exception: the symbol $=$ without any qualification always denotes ordinary equality. That is, we write $A = B$ when A and B denote the same object.

WARNING. In the literature, and contrary to the use in this book, the symbol $=$ is often used for β-equality.

1.3.5. EXAMPLE.

(i) $(\lambda x.xx)(\lambda zz) \to_\beta (xx)[x := \lambda zz] = (\lambda zz)(\lambda yy)$.

(ii) $(\lambda zz)(\lambda yy) \to_\beta z[z := \lambda yy] = \lambda yy$.

(iii) $(\lambda x.xx)(\lambda zz) \twoheadrightarrow_\beta \lambda yy$.

(iv) $(\lambda xx)yz =_\beta y((\lambda xx)z)$.

1.3.6. EXAMPLE (Some common λ-terms).

(i) Let $\mathbf{I} = \lambda xx$, $\mathbf{K} = \lambda xy.x$, and $\mathbf{S} = \lambda xyz.xz(yz)$. Then $\mathbf{SKK} \twoheadrightarrow_\beta \mathbf{I}$.

(ii) Let $\omega = \lambda x.xx$ and $\mathbf{\Omega} = \omega\omega$. Then $\mathbf{\Omega} \rightarrow_\beta \mathbf{\Omega} \rightarrow_\beta \mathbf{\Omega} \rightarrow_\beta \cdots$.

(iii) Let $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. Then $\mathbf{Y} \rightarrow_\beta \mathbf{Y}' \rightarrow_\beta \mathbf{Y}'' \rightarrow_\beta \cdots$, where $\mathbf{Y}, \mathbf{Y}', \mathbf{Y}'', \ldots$ are all different.

1.3.7. REMARK. A term is in normal form iff it is an abstraction λxM, where M is in normal form, or it is $xM_1 \ldots M_n$, where $n \geq 0$ and all M_i are in normal form ("if" is obvious and "only if" is by induction on the length of M). Even more compact: a normal form is $\lambda y_1 \ldots y_m.xM_1 \ldots M_n$, where $m, n \geq 0$ and all M_i are normal forms.

The following little properties are constantly used.

1.3.8. LEMMA.

(i) *If* $N \rightarrow_\beta N'$ *then* $M[x := N] \twoheadrightarrow_\beta M[x := N']$.

(ii) *If* $M \rightarrow_\beta M'$ *then* $M[x := N] \rightarrow_\beta M'[x := N]$.

PROOF. By induction on M and $M \rightarrow_\beta M'$ using Lemma 1.2.20. □

In addition to β-reduction, other notions of reduction are considered in the λ-calculus. In particular, we have *η-reduction.*

1.3.9. DEFINITION. Let $\rightarrow_\eta$ denote the least compatible relation satisfying

$$\lambda x.Mx \rightarrow_\eta M, \quad \text{if } x \notin \mathrm{FV}(M).$$

The symbol $\rightarrow_{\beta\eta}$ denotes the union of $\rightarrow_\beta$ and $\rightarrow_\eta$.

REMARK. In general, when we have compatible relations $\rightarrow_R$ and $\rightarrow_Q$, the union is written $\rightarrow_{RQ}$. Similar notation is used for more than two relations.

The motivation for this notion of reduction (and the associated notion of equality) is, informally speaking, that two functions should be considered equal if they yield equal results whenever applied to equal arguments. Indeed:

1.3.10. PROPOSITION. *Let* $=_{ext}$ *be the least equivalence relation such that:*

- *If* $M =_\beta N$, *then* $M =_{ext} N$;
- *If* $Mx =_{ext} Nx$ *and* $x \notin FV(M) \cup FV(N)$, *then* $M =_{ext} N$;
- *If* $P =_{ext} Q$, *then* $PZ =_{ext} QZ$ *and* $ZP =_{ext} ZQ$.

Then $M =_{ext} N$ *iff* $M =_{\beta\eta} N$.

PROOF. The implication from left to right is by induction on the definition of $M =_{ext} N$, and from right to left is by induction with respect to the definition of $M =_{\beta\eta} N$. Note that the definition of $=_{ext}$ does not include the rule "If $P =_{ext} Q$ then $\lambda x P =_{ext} \lambda x Q$." This is no mistake: this property (known as *rule* ξ) follows from the others. $\square$

We do not take $\to_{\beta\eta}$ as our standard notion of reduction. We want to be able to distinguish between two algorithms, even if their input-output behaviour is the same. Nevertheless, many properties of $\beta\eta$-reduction are similar to those of β-reduction. For instance, we have the following.

1.3.11. LEMMA. *If there is an infinite $\beta\eta$-reduction sequence starting with a term M then there is an infinite β-reduction sequence from M.*

PROOF. First observe that in an infinite $\beta\eta$-reduction sequence there must be infinitely many β-reduction steps (cf. Exercise 1.6). These β-reduction steps can be "permuted forward", yielding an infinite β-reduction. Indeed, by induction with respect to $M \to_\eta N$, one can show that $M \to_\eta N \to_\beta L$ implies $M \to_\beta P \twoheadrightarrow_{\beta\eta} L$, for some P. $\square$

OTHER SYNTAX. In the numerous lambda-calculi occurring in the later chapters, many notions introduced here will be used in an analogous way. For instance, the notion of a compatible relation (Definition 1.3.1) generalizes naturally to other syntax. A compatible relation "respects" the syntactic constructions. Imagine that, in addition to application and abstraction, we have in our syntax an operation of "acclamation," written as $M!!$, i.e. whenever M is a term, $M!!$ is also a term. Then we should add the clause

- If $M \succ N$ then $M!! \succ N!!$.

to our definition of compatibility. Various additional reduction relations will also be considered later, and we usually define these by stating one or more reduction axioms, similar to the β-rule of Definition 1.3.2 and the η-rule of Definition 1.3.9. In such cases, we usually assume that the reduction relation is the least compatible relation satisfying the given reduction axiom.

1.4. The Church-Rosser theorem

Since a λ-term M may contain several β-redexes, there may be several N with $M \to_\beta N$. For instance, $\mathbf{K}(\mathbf{II}) \to_\beta \lambda x.\mathbf{II}$ and $\mathbf{K}(\mathbf{II}) \to_\beta \mathbf{KI}$. However, as shown below, there must be some term to which all such N reduce in one or more steps. In fact, even if we make several reduction steps, we can still converge to a common term (possibly using several steps):

$$
\begin{tikzcd}
M_1 \arrow[r, two heads] \arrow[d, two heads] & M_2 \arrow[d, two heads, dotted] \\
M_3 \arrow[r, two heads, dotted] & M_4
\end{tikzcd}
$$

This property is known as *confluence* or the *Church-Rosser property.* If the above diagram was correct in a stronger version with $\to$ in place of $\twoheadrightarrow$, then we could prove the theorem by a *diagram chase:*

$$
\begin{tikzcd}
M_1 \arrow[r] \arrow[d] & \cdot \arrow[r] \arrow[d, dotted] & \cdot \arrow[r] \arrow[d, dotted] & M_2 \arrow[d, dotted] \\
\cdot \arrow[r, dotted] \arrow[d] & \cdot \arrow[r, dotted] \arrow[d, dotted] & \cdot \arrow[r, dotted] \arrow[d, dotted] & \cdot \arrow[d, dotted] \\
M_3 \arrow[r, dotted] & \cdot \arrow[r, dotted] & \cdot \arrow[r, dotted] & M_4
\end{tikzcd}
$$

Unfortunately, our prerequisite fails. For instance, in the diagram

$$
\begin{tikzcd}
M_1 = \omega(\mathbf{II}) \arrow[rr] \arrow[d] & & \omega\mathbf{I} = M_2 \arrow[d, dotted] \\
M_3 = (\mathbf{II})(\mathbf{II}) \arrow[r, dotted] & \mathbf{I}(\mathbf{II}) \arrow[r, dotted] & \mathbf{II} = M_4
\end{tikzcd}
$$

two reductions are needed to get from M_3 to M_4. The problem is that the redex contracted in the reduction from M_1 to M_2 is duplicated in the reduction to M_3. We can solve the problem by working with *parallel reduction,* i.e. an extension of $\to_\beta$ allowing such duplications to be contracted in one step.

1.4.1. DEFINITION. Let $\Rightarrow_\beta$ be the least relation on Λ such that:

- $x \Rightarrow_\beta x$ for all variables x.
- If $P \Rightarrow_\beta Q$ then $\lambda x.P \Rightarrow_\beta \lambda x.Q$.
- If $P_1 \Rightarrow_\beta Q_1$ and $P_2 \Rightarrow_\beta Q_2$ then $P_1P_2 \Rightarrow_\beta Q_1Q_2$.
- If $P_1 \Rightarrow_\beta Q_1$ and $P_2 \Rightarrow_\beta Q_2$ then $(\lambda x.P_1)P_2 \Rightarrow_\beta Q_1[x := Q_2]$.

1.4.2. LEMMA.

(i) *If $M \to_\beta N$ then $M \Rightarrow_\beta N$.*

(ii) *If $M \Rightarrow_\beta N$ then $M \twoheadrightarrow_\beta N$.*

(iii) *If $M \Rightarrow_\beta M'$ and $N \Rightarrow_\beta N'$ then $M[x:=N] \Rightarrow_\beta M'[x:=N']$.*

PROOF. (i) is by induction on the definition of $M \to_\beta N$ (note that $P \Rightarrow_\beta P$ for all P), and (ii), (iii) are by induction on the definition of $M \Rightarrow_\beta M'$. $\square$

1.4.3. DEFINITION. Let M^* (*the complete development of* M) be defined by:

$$
\begin{array}{lcl}
x^* & = & x; \\
(\lambda x M)^* & = & \lambda x M^*; \\
(MN)^* & = & M^*N^*, \text{ if } M \text{ is not an abstraction;} \\
((\lambda x M)N)^* & = & M^*[x := N^*].
\end{array}
$$

Note that $M \Rightarrow_\beta N$ if N arises by reducing *some* of the redexes present in M, and that M^* arises by reducing *all* redexes present in M.

1.4.4. LEMMA. *If $M \Rightarrow_\beta N$ then $N \Rightarrow_\beta M^*$. In particular, if $M_1 \Rightarrow_\beta M_2$ and $M_1 \Rightarrow_\beta M_3$ then $M_2 \Rightarrow_\beta M_1^*$ and $M_3 \Rightarrow_\beta M_1^*$.*

PROOF. By induction on the definition of $M \Rightarrow_\beta N$, using Lemma 1.4.2. □

1.4.5. THEOREM (Church and Rosser). *If $M_1 \twoheadrightarrow_\beta M_2$ and $M_1 \twoheadrightarrow_\beta M_3$, then there is $M_4 \in \Lambda$ with $M_2 \twoheadrightarrow_\beta M_4$ and $M_3 \twoheadrightarrow_\beta M_4$.*

PROOF. If $M_1 \to_\beta \cdots \to_\beta M_2$ and $M_1 \to_\beta \cdots \to_\beta M_3$, the same holds with $\Rightarrow_\beta$ in place of $\to_\beta$. By Lemma 1.4.4 and a diagram chase, $M_2 \Rightarrow_\beta \cdots \Rightarrow_\beta M_4$ and $M_3 \Rightarrow_\beta \cdots \Rightarrow_\beta M_4$ for some M_4. Then $M_2 \twoheadrightarrow_\beta M_4$ and $M_3 \twoheadrightarrow_\beta M_4$. □

1.4.6. COROLLARY.

(i) *If $M =_\beta N$, then $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$ for some L.*

(ii) *If $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$ for β-normal forms N_1, N_2, then $N_1 = N_2$.*

(iii) *If there are β-normal forms L_1 and L_2 such that $M \twoheadrightarrow_\beta L_1$, $N \twoheadrightarrow_\beta L_2$, and $L_1 \neq L_2$, then $M \neq_\beta N$.*

PROOF. Left to the reader. □

REMARK. One can consider λ-calculus as an equational theory, i.e. a formal theory with formulas of the form $M =_\beta N$. Part (i) establishes *consistency* of this theory, in the following sense: there exists a formula that cannot be proved, e.g. $\lambda x x =_\beta \lambda xy.x$ (cf. Exercise 2.5).

Part (ii) in the corollary is similar to the fact that when we calculate the value of an arithmetical expression, e.g. $(4+2) \cdot (3+7) \cdot 11$, the end result is independent of the order in which we do the calculations.

1.5. Leftmost reductions are normalizing

1.5.1. DEFINITION. A term M is *normalizing* (notation $M \in \mathrm{WN}_\beta$) iff there is a reduction sequence from M ending in a normal form N. We then say that M *has* the normal form N. A term M is *strongly normalizing* ($M \in \mathrm{SN}_\beta$ or just $M \in \mathrm{SN}$) if all reduction sequences starting with M are finite. We write $M \in \infty_\beta$ if $M \notin \mathrm{SN}_\beta$. Similar notation applies to other notions of reduction.

Any strongly normalizing term is also normalizing, but the converse is not true, as $\mathbf{K I \Omega}$ shows. But Theorem 1.5.8 states that a normal form, if it exists, can always be found by repeatedly reducing the leftmost redex, i.e. the redex whose λ is the furthest to the left. The following notation and definition are convenient for proving Theorem 1.5.8.

VECTOR NOTATION. Let $n \geq 0$. If $\vec{P} = P_1, \ldots, P_n$, then we write $M\vec{P}$ for $MP_1 \ldots P_n$. In particular, if $n = 0$, i.e. $\vec{P}$ is empty, then $M\vec{P}$ is just M. Similarly, if $\vec{z} = z_1, \ldots, z_n$, then we write $\lambda\vec{z}.M$ for $\lambda z_1 \ldots z_n.M$. Again, $\lambda\vec{z}.M$ is just M, if $n = 0$, i.e. $\vec{z}$ is empty.

REMARK. Any term has exactly one of the following two forms: $\lambda\vec{z}.x\vec{R}$ or $\lambda\vec{z}.(\lambda xP)Q\vec{R}$, in which case $(\lambda xP)Q$ is called *head* redex (in the former case, there is no head redex). Any redex that is not a head redex is called *internal*. A head redex is always the leftmost redex, but the leftmost redex in a term is not necessarily a head redex—it may be internal.

1.5.2. DEFINITION. For a term M not in normal form, we write

- $M \xrightarrow{l}_\beta N$ if N arises from M by contracting the leftmost redex.
- $M \xrightarrow{h}_\beta N$ if N arises from M by contracting a head redex.
- $M \xrightarrow{i}_\beta N$ if N arises from M by contracting an internal redex.

1.5.3. LEMMA.

(i) *If* $M \xrightarrow{h}_\beta N$ *then* $\lambda xM \xrightarrow{h}_\beta \lambda xN$.

(ii) *If* $M \xrightarrow{h}_\beta N$ *and* M *is not an abstraction, then* $ML \xrightarrow{h}_\beta NL$.

(iii) *If* $M \xrightarrow{h}_\beta N$ *then* $M[x := L] \xrightarrow{h}_\beta N[x := L]$.

PROOF. Easy. □

The following technical notions are central in the proof of Theorem 1.5.8.

1.5.4. DEFINITION. We write $\vec{P} \Rightarrow_\beta \vec{Q}$ if $\vec{P} = P_1, \ldots, P_n$, $\vec{Q} = Q_1, \ldots, Q_n$, $n \geq 0$, and $P_j \Rightarrow_\beta Q_j$ for all $1 \leq j \leq n$. *Parallel internal reduction* $\overset{i}{\Rightarrow}_\beta$ is the least relation on Λ satisfying the following rules.

- If $\vec{P} \Rightarrow_\beta \vec{Q}$ then $\lambda\vec{x}.y\vec{P} \overset{i}{\Rightarrow}_\beta \lambda\vec{x}.y\vec{Q}$.
- If $\vec{P} \Rightarrow_\beta \vec{Q}$, $S \Rightarrow_\beta T$ and $R \Rightarrow_\beta U$, then $\lambda\vec{x}.(\lambda yS)R\vec{P} \overset{i}{\Rightarrow}_\beta \lambda\vec{x}.(\lambda yT)U\vec{Q}$.

REMARK. If $M \xrightarrow{i}_\beta N$, then $M \overset{i}{\Rightarrow}_\beta N$. Conversely, if $M \overset{i}{\Rightarrow}_\beta N$, then $M \overset{i}{\twoheadrightarrow}_\beta N$. Also, if $M \overset{i}{\Rightarrow}_\beta N$, then $M \Rightarrow_\beta N$.

1.5.5. DEFINITION. We write $M \Rrightarrow_\beta N$ if there are $M_0, M_1, \ldots, M_n$ with

$$M = M_0 \xrightarrow{h}_\beta M_1 \xrightarrow{h}_\beta \cdots \xrightarrow{h}_\beta M_n \overset{i}{\Rightarrow}_\beta N$$

and $M_i \Rightarrow_\beta N$ for all $i \in \{0, \ldots, n\}$, where $n \geq 0$.

1.5.6. LEMMA.

(i) *If* $M \Rrightarrow_\beta M'$ *then* $\lambda x M \Rrightarrow_\beta \lambda x M'$.

(ii) *If* $M \Rrightarrow_\beta M'$ *and* $N \Rightarrow_\beta N'$, *then* $MN \Rrightarrow_\beta M'N'$.

(iii) *If* $M \Rrightarrow_\beta M'$ *and* $N \Rrightarrow_\beta N'$, *then* $M[x := N] \Rrightarrow_\beta M'[x := N']$.

PROOF. Part (i) is easy. For (ii), let

$$M = M_0 \xrightarrow{h}_\beta M_1 \xrightarrow{h}_\beta \cdots \xrightarrow{h}_\beta M_n \overset{i}{\Rrightarrow}_\beta M'$$

where $M_i \Rightarrow_\beta M'$ for all $i \in \{0, \ldots, n\}$. Assume at least one M_i is an abstraction, and let k be the smallest number such that M_k is an abstraction. Then $M_k N \overset{i}{\Rrightarrow}_\beta M'N'$. By Lemma 1.5.3:

$$MN = M_0 N \xrightarrow{h}_\beta M_1 N \xrightarrow{h}_\beta \cdots \xrightarrow{h}_\beta M_k N \overset{i}{\Rrightarrow}_\beta M'N', \qquad (*)$$

where $M_i N \Rightarrow_\beta M'N'$. If there is no abstraction among M_i, $0 \le i \le n$, then $(*)$ still holds with $k = n$.

For (iii) first assume $M \overset{i}{\Rrightarrow}_\beta M'$. We have either $M = \lambda\vec{z}.(\lambda y.P)Q\vec{R}$ or $M = \lambda\vec{z}.y\vec{R}$. In the former case, $M' = \lambda\vec{z}.(\lambda y.P')Q'\vec{R}'$, where $P \Rightarrow_\beta P'$, $Q \Rightarrow_\beta Q'$, and $\vec{R} \Rightarrow_\beta \vec{R}'$. By Lemma 1.4.2, $M[x := N] \overset{i}{\Rrightarrow}_\beta M'[x := N']$. In the latter case, $M' = \lambda\vec{z}.y\vec{R}'$. We consider two possibilities. If $x \neq y$, we proceed as in the case just considered. If $x = y$, let $\vec{S}$ and $\vec{S}'$ arise from $\vec{R}$ (respectively $\vec{R}'$) by substituting N (respectively N') for x. By (i), (ii) and Lemma 1.4.2 we then have $M[x := N] = \lambda\vec{z}.N\vec{S} \Rrightarrow_\beta \lambda\vec{z}.N'\vec{S}' = M'[x := N']$.

Now consider the general case. By the above and Lemma 1.5.3, we have

$$M[x := N] = M_0[x := N] \xrightarrow{h}_\beta \cdots \xrightarrow{h}_\beta M_n[x := N] \Rrightarrow_\beta M'[x := N'].$$

Also, $M_i[x := N] \Rightarrow_\beta M'[x := N']$ holds by Lemma 1.4.2. □

1.5.7. LEMMA.

(i) *If* $M \Rightarrow_\beta N$ *then* $M \overset{h}{\twoheadrightarrow}_\beta L \overset{i}{\Rrightarrow}_\beta N$ *for some* L.

(ii) *If* $M \overset{i}{\Rrightarrow}_\beta N \xrightarrow{h}_\beta L$, *then* $M \overset{h\,+}{\twoheadrightarrow}_\beta O \overset{i}{\Rrightarrow}_\beta L$ *for some* O.

PROOF. For (i), show that $M \Rightarrow_\beta N$ implies $M \Rrightarrow_\beta N$. For (ii), note that $M = \lambda\vec{z}.(\lambda x.P)Q\vec{R}$, $N = \lambda\vec{z}.(\lambda x.P')Q'\vec{R}'$, where $P \Rightarrow_\beta P'$, $Q \Rightarrow_\beta Q'$, and $\vec{R} \Rightarrow_\beta \vec{R}'$. Then $L = \lambda\vec{z}.P'[x := Q']\vec{R}'$. Define $O = \lambda\vec{z}.P[x := Q]\vec{R}$, and we then have $M \xrightarrow{h}_\beta O \Rightarrow_\beta L$. Now use (i). □

1.5.8. THEOREM. *If* M *has normal form* N, *then* $M \overset{l}{\twoheadrightarrow}_\beta N$.

PROOF. Induction on the length of N. We have $M \Rightarrow_\beta \cdots \Rightarrow_\beta N$. By Lemma 1.5.7, the reduction from M to N consists of head reductions and parallel internal reductions, and the head reductions can be brought to the beginning. Thus $M \overset{h}{\twoheadrightarrow}_\beta L \overset{i}{\twoheadrightarrow}_\beta N$, where $L = \lambda\vec{z}.y\vec{P}$ and $N = \lambda\vec{z}.y\vec{P}'$, and where P_i has normal form P_i'. By the induction hypothesis, leftmost reduction of each P_i yields P_i'. Then $M \overset{l}{\twoheadrightarrow}_\beta N$. □

1.5.9. DEFINITION. A *reduction strategy* F is a map from λ-terms to λ-terms such that $F(M) = M$ when M is in normal form, and $M \to_\beta F(M)$ otherwise. Such an F is *normalizing* if for all $M \in \mathrm{WN}_\beta$, there is an i such that $F^i(M)$ is in normal form.

1.5.10. COROLLARY. *Define $F_l(M) = M$ for each normal form M, and $F_l(M) = N$, where $M \overset{l}{\to}_\beta N$, otherwise. Then F_l is normalizing.*

1.5.11. DEFINITION. A reduction sequence is called

- *quasi-leftmost* if it contains infinitely many leftmost reductions;
- *quasi-head* if it contains infinitely many head reductions.

1.5.12. COROLLARY. *Let M be normalizing. We then have the following.*

(i) *M has no infinite head-reduction sequence.*

(ii) *M has no quasi-head reduction sequence.*

(iii) *M has no quasi-leftmost reduction sequence.*

PROOF. Part (i) follows directly from the theorem. For (ii), suppose M has a quasi-head reduction sequence. By Lemma 1.5.7(ii), we can postpone the internal reductions in the quasi-head reduction indefinitely to get an infinite head reduction, contradicting Theorem 1.5.8.

Part (iii) is by induction on the length of the normal form of M. Suppose M has a quasi-leftmost reduction sequence. By (ii) we may assume that, from some point on, the quasi-leftmost reduction contains no head reductions. One of the reductions after this point must be leftmost, so the sequence is

$$M \twoheadrightarrow_\beta L_1 \overset{i}{\to}_\beta L_2 \overset{i}{\to}_\beta L_3 \overset{i}{\to}_\beta \cdots \qquad (*)$$

where $L_1 = \lambda\vec{z}.y\vec{P}$. Infinitely many of the leftmost steps in $(*)$ must occur within the same P_i and these steps are leftmost relative to P_i, contradicting the induction hypothesis. □

1.6. Perpetual reductions and the conservation theorem

Theorem 1.5.8 provides a way to obtain the normal form of a term, when it exists. There is also a way to *avoid* the normal form, i.e. to find an infinite reduction, when one exists.

1.6.1. DEFINITION. Define $F_\infty : \Lambda \to \Lambda$ as follows. If $M \in \mathrm{NF}_\beta$ then $F_\infty(M) = M$; otherwise

$$\begin{array}{lcll} F_\infty(\lambda\vec{z}.x\vec{P}Q\vec{R}) & = & \lambda\vec{z}.x\vec{P}F_\infty(Q)\vec{R}, & \text{if } \vec{P} \in \mathrm{NF}_\beta \text{ and } Q \notin \mathrm{NF}_\beta; \\ F_\infty(\lambda\vec{z}.(\lambda x P)Q\vec{R}) & = & \lambda\vec{z}.P[x := Q]\vec{R}, & \text{if } x \in \mathrm{FV}(P) \text{ or } Q \in \mathrm{NF}_\beta; \\ F_\infty(\lambda\vec{z}.(\lambda x P)Q\vec{R}) & = & \lambda\vec{z}.(\lambda x P)F_\infty(Q)\vec{R}, & \text{if } x \notin \mathrm{FV}(P) \text{ and } Q \notin \mathrm{NF}_\beta. \end{array}$$

It is easy to see that $M \to F_\infty(M)$ when $M \notin \mathrm{NF}_\beta$.

1.6.2. LEMMA. *Assume $Q \in \mathrm{SN}_\beta$ or $x \in \mathrm{FV}(P)$. If $P[x := Q]\vec{R} \in \mathrm{SN}_\beta$, then $(\lambda x.P)Q\vec{R} \in \mathrm{SN}_\beta$.*

PROOF. Let $P[x := Q]\vec{R} \in \mathrm{SN}_\beta$. Then $P, \vec{R} \in \mathrm{SN}_\beta$. If $x \notin \mathrm{FV}(P)$, then $Q \in \mathrm{SN}_\beta$ by assumption. If $x \in \mathrm{FV}(P)$, then Q is part of $P[x := Q]\vec{R}$, so $Q \in \mathrm{SN}_\beta$. If $(\lambda x.P)Q\vec{R} \in \infty_\beta$, then any infinite reduction must have form

$$(\lambda x.P)Q\vec{R} \twoheadrightarrow_\beta (\lambda x.P')Q'\vec{R}' \to_\beta P'[x := Q']\vec{R}' \to_\beta \cdots$$

Then $P[x := Q]\vec{R} \twoheadrightarrow_\beta P'[x := Q']\vec{R}' \to_\beta \cdots$, a contradiction. □

1.6.3. THEOREM. *If $M \in \infty_\beta$ then $F_\infty(M) \in \infty_\beta$.*

PROOF. By induction on M. If $M = \lambda\vec{z}.x\vec{P}$, apply the induction hypothesis as necessary. We consider the remaining cases in more detail.
CASE 1: $M = \lambda\vec{z}.(\lambda x.P)Q\vec{R}$ where $x \in \mathrm{FV}(P)$ or $Q \in \mathrm{NF}_\beta$. Then we have $F_\infty(M) = \lambda\vec{z}.P[x := Q]\vec{R}$, and thus $F_\infty(M) \in \infty_\beta$, by Lemma 1.6.2.
CASE 2: $M = \lambda\vec{z}.(\lambda x.P)Q\vec{R}$ where $x \notin \mathrm{FV}(P)$ and $Q \in \infty_\beta$. Then we have $F_\infty(M) = \lambda\vec{z}.(\lambda x.P)F_\infty(Q)\vec{R}$. By the induction hypothesis $F_\infty(Q) \in \infty_\beta$, so $F_\infty(M) \in \infty_\beta$.
CASE 3: $M = \lambda\vec{z}.(\lambda x.P)Q\vec{R}$ where $x \notin \mathrm{FV}(P)$ and $Q \in \mathrm{SN}_\beta - \mathrm{NF}_\beta$. Then we have $F_\infty(M) = \lambda\vec{z}.(\lambda x.P)F_\infty(Q)\vec{R} \twoheadrightarrow_\beta P\vec{R}$. From Lemma 1.6.2, we obtain $P\vec{R} \in \infty_\beta$, but then also $F_\infty(M) \in \infty_\beta$. □

1.6.4. DEFINITION. A reduction strategy F is *perpetual* iff for all $M \in \infty_\beta$,

$$M \to_\beta F(M) \to_\beta F(F(M)) \to_\beta \cdots$$

is an infinite reduction sequence from M.

1.6.5. COROLLARY. *F_∞ is perpetual.*

PROOF. Immediate from the preceding theorem. □

1.6.6. DEFINITION. The set of *λI-terms* is defined as follows.

- Every variable is a λI-term.
- An application MN is a λI-term iff both M and N are λI-terms.
- An abstraction $\lambda x M$ is a λI-term iff M is a λI-term and $x \in \mathrm{FV}(M)$.

The following is known as the *conservation theorem* (for λI-terms).

1.6.7. COROLLARY.

(i) *For all λI-terms M, if $M \in \mathrm{WN}_\beta$ then $M \in \mathrm{SN}_\beta$.*

(ii) *For all λI-terms M, if $M \in \infty_\beta$ and $M \to_\beta N$ then $N \in \infty_\beta$.*

PROOF. For part (i), assume $M \in \mathrm{WN}_\beta$. Then $M \overset{l}{\twoheadrightarrow}_\beta N$ for some normal form N. Now note that for all λI-terms L not in normal form, $L \overset{l}{\to}_\beta F_\infty(L)$. Thus $N = F_\infty^k(M)$ for some k, so $M \in \mathrm{SN}_\beta$, by Corollary 1.6.5.

For part (ii), suppose $M \to_\beta N$. If $M \in \infty_\beta$, then $M \notin \mathrm{WN}_\beta$, by (i). Hence $N \notin \mathrm{WN}_\beta$, in particular $N \in \infty_\beta$. □

1.7. Expressibility and undecidability

The untyped λ-calculus is so simple that it may be surprising how powerful it is. In this section we show that λ-calculus in fact can be seen as an alternative formulation of recursion theory.

We can use λ-terms to represent various constructions, e.g. truth values:

$$\begin{aligned} \mathrm{true} &= \lambda xy.x; \\ \mathrm{false} &= \lambda xy.y; \\ \textbf{if } P \textbf{ then } Q \textbf{ else } R &= PQR. \end{aligned}$$

It is easy to see that

$$\begin{array}{lcl} \textbf{if } \mathrm{true} \textbf{ then } \mathrm{P} \textbf{ else } \mathrm{Q} & \twoheadrightarrow_\beta & P; \\ \textbf{if } \mathrm{false} \textbf{ then } \mathrm{P} \textbf{ else } \mathrm{Q} & \twoheadrightarrow_\beta & Q. \end{array}$$

Another useful construction is the ordered pair

$$\begin{array}{rcl} \langle M, N\rangle & = & \lambda x.xMN; \\ \pi_1 & = & \lambda p.p(\lambda xy.x); \\ \pi_2 & = & \lambda p.p(\lambda xy.y). \end{array}$$

As expected we have

$$\pi_i\langle M_1, M_2\rangle \twoheadrightarrow_\beta M_i.$$

1.7.1. DEFINITION. We represent the natural numbers in the λ-calculus as *Church numerals*:

$$\mathbf{c}_n = \lambda f x. f^n(x),$$

where $f^n(x)$ abbreviates $f(f(\cdots(x)\cdots))$ with n occurrences of f. Sometimes we write $\mathbf{n}$ for $\mathbf{c}_n$, so that for instance

$$\begin{array}{rcl}\mathbf{0} & = & \lambda f x. x;\\ \mathbf{1} & = & \lambda f x. f x;\\ \mathbf{2} & = & \lambda f x. f(f x).\end{array}$$

1.7.2. DEFINITION. A partial function $f : \mathbb{N}^k \to \mathbb{N}$ is *λ-definable* iff there is an $F \in \Lambda$ such that:

- If $f(n_1, \ldots, n_k) = m$ then $F\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_k} =_\beta \mathbf{c}_m$.
- If $f(n_1, \ldots, n_k)$ is undefined then $F\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_k}$ has no normal form.

We say that the term F *defines* the function f.

REMARK. If F defines f, then in fact $F\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_k} \twoheadrightarrow_\beta \mathbf{c}_{f(n_1,\ldots,n_k)}$.

1.7.3. EXAMPLE. The following terms define a few commonly used functions.

- Successor: $\mathrm{succ} = \lambda n f x. f(n f x)$.
- Addition: $\mathrm{add} = \lambda m n f x. m f(n f x)$.
- Multiplication: $\mathrm{mult} = \lambda m n f x. m(n f)x$.
- Exponentiation: $\exp = \lambda m n f x. m n f x$.
- The constant zero function: $\mathrm{zero} = \lambda m.\mathbf{0}$.
- The i-th projection of k-arguments: $\Pi_i^k = \lambda m_1 \ldots m_k. m_i$.

We show that all partial recursive functions are λ-definable.

1.7.4. PROPOSITION. *The primitive recursive functions are λ-definable.*

PROOF. It follows from Example 1.7.3 that the initial functions are definable. It should also be obvious that the class of λ-definable total functions is closed under composition. It remains to show that λ-definability is closed under primitive recursion. Assume that f is given by

$$\begin{array}{rcl}f(0, n_1, \ldots, n_m) & = & g(n_1, \ldots, n_m);\\ f(n+1, n_1, \ldots, n_m) & = & h(f(n, n_1, \ldots, n_m), n, n_1, \ldots, n_m),\end{array}$$

where g and h are λ-definable by G and H. Define auxiliary terms

$$\begin{array}{rcl}\text{Init} & = & \langle 0, Gx_1 \dots x_m \rangle. \\ \text{Step} & = & \lambda p.\langle \text{succ}(\pi_1 p), H(\pi_2 p)(\pi_1 p)x_1 \dots x_m \rangle;\end{array}$$

The function f is then λ-definable by

$$F = \lambda x x_1 \dots x_m.\pi_2(x \,\text{Step}\,\text{Init}).$$

This expresses the following algorithm: Generate a sequence of pairs

$$(0, a_0), (1, a_1), \dots, (n, a_n),$$

where $a_0 = g(n_1, \dots, n_m)$ and $a_{i+1} = h(a_i, i, n_1, \dots, n_m)$, so at the end of the sequence, we have $a_n = f(n, n_1, \dots, n_m)$. □

1.7.5. THEOREM. *All partial recursive functions are λ-definable.*

PROOF. Let f be a partial recursive function. By Theorem A.3.8

$$f(n_1, \dots, n_m) = \ell(\mu y[g(y, n_1, \dots, n_m) = 0]),$$

where g and ℓ are primitive recursive. We show that f is λ-definable. For this, we first define a test for zero:

$$\text{zero?} = \lambda x.x(\lambda y.\text{false})\text{true}.$$

By Proposition 1.7.4, the functions g and ℓ are definable by some terms G and L, respectively. Let

$$W = \lambda y.\,\text{if zero?}(Gyx_1 \dots x_m)\ \textbf{then}\ \lambda w.Ly\ \textbf{else}\ \lambda w.w(\text{succ}\, y)w.$$

Note that $x_1, \dots, x_m$ are free in W. The following term defines f:

$$F = \lambda x_1.\dots\lambda x_m.W\mathbf{c}_0 W.$$

Indeed, take any $n_1, \dots, n_m$ and let $\vec{c} = \mathbf{c}_{n_1} \dots \mathbf{c}_{n_m}$. Then

$$F\vec{c} \twoheadrightarrow_\beta W'\mathbf{c}_0 W',$$

where $W' = W[\vec{x} := \vec{c}]$. Suppose that $g(n, n_1, \dots, n_m) = 0$, and n is the least number with this property. Then

$$W'\mathbf{c}_0 W' \twoheadrightarrow_\beta W'\mathbf{c}_1 W' \twoheadrightarrow_\beta \cdots \twoheadrightarrow_\beta W'\mathbf{c}_n W' \twoheadrightarrow_\beta L\mathbf{c}_n \twoheadrightarrow_\beta \mathbf{c}_{\ell(n)}.$$

It remains to see what happens when the minimum is not defined. Then we have the following infinite quasi-leftmost reduction sequence

$$W'\mathbf{c}_0 W' \twoheadrightarrow_\beta W'\mathbf{c}_1 W' \twoheadrightarrow_\beta W'\mathbf{c}_2 W' \twoheadrightarrow_\beta \cdots$$

so Corollary 1.5.12 implies that $F\vec{c}$ has no normal form. □

1.7.6. REMARK. A close inspection of the proof of Theorem 1.7.5 reveals that it shows more than stated in the theorem: For every partial recursive function $f : \mathbb{N}^m \to \mathbb{N}$, there is a defining term F such that every application $F\mathbf{c}_{n_1} \dots \mathbf{c}_{n_m}$ is *uniformly normalizing*, i.e. either strongly normalizing or without normal form. The details of this claim can be found in Exercise 1.21.

1.7.7. COROLLARY. *The following problems are undecidable:*

(i) *Given $M \in \Lambda$, does M have a normal form?*

(ii) *Given $M \in \Lambda$, is M strongly normalizing?*

PROOF. For (i), suppose we have an algorithm to decide whether any term has a normal form. Take any recursively enumerable set $A \subseteq \mathbb{N}$ that is not recursive, and let f be a partial recursive function with domain A. Clearly, f is λ-definable by some term F. Now, for a given $n \in \mathbb{N}$, we can effectively decide whether $n \in A$ by checking whether the term $F\mathbf{n}$ has a normal form. Part (ii) now follows from Remark 1.7.6. □

1.8. Notes

The λ-calculus and the related systems of *combinatory logic* were introduced around 1930 by Alonzo Church [69, 70] and Haskell B. Curry [98, 99, 101], respectively. From the beginning, the calculi were parts of systems intended to be a foundation for logic. Unfortunately, Church's students Kleene and Rosser [271] discovered in 1935 that the original systems were inconsistent, and Curry [103] simplified the result, which became known as *Curry's paradox*. Consequently, the subsystem dealing with λ-terms, reduction, and conversion, i.e. what we call λ-calculus, was studied independently.

The notions of λ-binding and α-convertible terms are intuitively very clear, but we have seen in Section 1.2 that various technical difficulties must be overcome in order to handle them properly. This issue becomes especially vital when one faces the problem of a practical implementation. A classical solution [59] is to use a nameless representation of variables (so called *de Bruijn indices*). For more on this and related subjects, see e.g. [392, 395, 401].

The Church-Rosser theorem, which can be seen as a consistency result for the λ-calculus, was proved in 1936 by Church and Rosser [76]. Many proofs appeared later. Barendregt [31] cites Tait and Martin-Löf for the technique using parallel reductions; our proof is from Takahashi [470]. Proofs of the Church-Rosser theorem and an extension of Theorem 1.5.8 for $\beta\eta$-reductions can also be found there.

Church and Rosser [76] also proved the conservation theorem for λI-terms (which is sometimes called the *second Church-Rosser theorem*). Again, many different proofs have appeared. Our proof uses the effective perpetual strategy from [31], and the fact, also noted in [412], that the perpetual strategy always contracts the leftmost redex, when applied to a λI-term. More about perpetual strategies and their use in proving conservation theorems can be found in [406] and [361].

The λ-calculus turned out to be useful for formalizing the intuitive notion of effective computability. Kleene [267] showed that every partial recursive function was λ-definable and vice versa. This led Church to conjecture that λ-definability is an appropriate formalization of the intuitive notion of effective computability [72], which became known as *Church's thesis*.

The problems of deciding membership of WN_β and SN_β can be seen as variants of the halting problem. Church [71, 72] inferred from the former his celebrated theorem stating that first-order arithmetic is undecidable, as well as the undecidability of the *Entscheidungsproblem* (the "decision problem" for first-order logic), results that were "in the air" in this period [164].

Curry and his co-workers continued the work on *illative* combinatory logic [107, 108], i.e. logical systems including formulas as well as combinators and types. The calculus of combinators was then studied as an independent subject, and a wealth of results was obtained. For instance, the theorem about leftmost reductions is from [107]. Like many other classical results in λ-calculus it has been proved in many different ways ever since; our proof is taken from [470].

With the invention of computing machinery came also programming languages. Already in the 1960's λ-calculus was recognized as a useful tool in the design, implementation, and theory of programming languages [295]. In particular, type-free λ-calculus constitutes a model of higher-order untyped *functional* programming languages, e.g. Scheme [3] and Lisp [200], while typed calculi correspond to functional languages like Haskell [478] and ML [387].

The classic texts on type-free λ-calculus are [241] and [31]. First-hand historical information may be obtained from Curry and Feys' book [107], which contains a wealth of historical information, and from Rosser and Kleene's eyewitness statements [270, 420]. Other interesting papers are [28, 238].

1.9. Exercises

1.1. For a pre-term M, the directed labeled graph $G(M)$ is defined by induction.

- If $M = x$ then $G(M)$ has a single root node labeled x and no edges.
- If $M = PQ$ then $G(M)$ is obtained from the union of $G(P)$ and $G(Q)$ by adding a new initial (root) node labeled @. This node has two outcoming edges: to the root nodes of $G(P)$ and $G(Q)$.
- If $M = \lambda x P$ then $G(M)$ is obtained from $G(P)$ by
  - Adding a new root node labeled λx, and an edge from there to the root node of $G(P)$;
  - Adding edges to the new root from all final nodes labeled x.

For a given graph G, let $\mathit{erase}(G)$ be a graph obtained from G by deleting all labels from the variable nodes that are not final and renaming every label λx, for some variable x, to λ. Prove that the conditions $\mathit{erase}(G(M)) = \mathit{erase}(G(N))$ and $M =_\alpha N$ are equivalent for all $M, N \in \Lambda^-$.

1.2. Modify Definition 1.2.4 so that the operation $M[x := N]$ is defined for all M, N and x. Then prove that $M[x := N] =_\alpha M'[x := N']$ holds for all $M =_\alpha M'$ and $N =_\alpha N'$ (cf. Lemma 1.2.11).

1.3. Let $\vec{x} \in \Upsilon$ and $\vec{N} \in \Lambda$ be fixed. Show that Definition 1.2.21 determines a total function from Λ to Λ. *Hint:* Rewrite the definition as a relation $r \subseteq \Lambda \times \Lambda$ and show that for every $M \in \Lambda$ there is exactly one $L \in \Lambda$ such that $r(M, L)$. It may be beneficial to show uniqueness and existence separately.

1.4. Prove that if $(\lambda x P)Q = (\lambda y M)N$ then $P[x := Q] = M[y := N]$. In other words, the contraction of a given redex yields a unique result.

1.5. Show that M is in normal form if and only if M is either a variable or an abstraction $\lambda x M'$, where M' is normal, or $M = M'[x := yN]$, for some normal forms M' and N, and some x occurring free exactly once in M'.

1.6. Show that every term is strongly normalizing with respect to eta-reductions.

1.7. Which of the following are true?

(i) $\lambda x.Mx =_\beta M$, for any abstraction M with $x \notin \mathrm{FV}(M)$.

(ii) $\lambda x\mathbf{\Omega} =_\beta \mathbf{\Omega}$.

(iii) $(\lambda x.xx)(\lambda xy.y(xx)) =_\beta (\lambda x.x\mathbf{I}x)(\lambda zxy.y(xzx))$.

1.8. Prove the *weak Church-Rosser theorem:* For all $M_1, M_2, M_3 \in \Lambda$, if $M_1 \to_\beta M_2$ and $M_1 \to_\beta M_3$, then there is $M_4 \in \Lambda$ with $M_2 \twoheadrightarrow_\beta M_4$ and $M_3 \twoheadrightarrow_\beta M_4$. Do not use the Church-Rosser theorem.

1.9. Which of the following are true?

(i) $\mathbf{II}(\mathbf{II}) \Rightarrow_\beta \mathbf{II}$.

(ii) $\mathbf{II}(\mathbf{II}) \Rightarrow_\beta \mathbf{I}$.

(iii) $\mathbf{IIII} \Rightarrow_\beta \mathbf{III}$.

(iv) $\mathbf{IIII} \Rightarrow_\beta \mathbf{I}$.

1.10. Find terms M, N such that $M \Rightarrow_\beta N$ and $M \overset{i}{\twoheadrightarrow}_\beta N$, but not $M \overset{i}{\Rightarrow}_\beta N$.

1.11. Find terms M, N such that $M \overset{i}{\to}_\beta N$ and $M \overset{h}{\to}_\beta N$ both hold.

1.12. Let $M \to_\beta F_\infty(M) \to_\beta F_\infty(F_\infty(M)) \to_\beta \cdots \to_\beta F_\infty^n(M)$, where $F_\infty^n(M) \in \mathrm{NF}$. Show that there is no reduction from M with more than n reduction steps. *Hint:* Generalize Lemma 1.6.2 to show that

$$l_\beta((\lambda x.P)Q\vec{R}) = l_\beta(P[x := Q]\vec{R}) + \epsilon(P) \cdot l_\beta(Q) + 1,$$

where $l_\beta(M)$ denotes the length of the longest reduction sequence from M and $\epsilon(P)$ is 1 if $x \notin \mathrm{FV}(P)$ and 0 else. Show that $l_\beta(M) = 1 + l_\beta(F_\infty(M))$, if $M \notin \mathrm{NF}$.

1.13. Show that there is no total computable $l : \Lambda \to \mathbb{N}$ such that, for all $M \in \mathrm{SN}_\beta$,

$$l(M) \geq l_\beta(M),$$

where $l_\beta(M)$ is as in the Exercise 1.12. *Hint:* That would imply decidability of SN.

1.14. Consider the *fixed point combinator:*

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

Show that $F(\mathbf{Y}F) =_\beta \mathbf{Y}F$ holds for all F. (Thus in the untyped lambda-calculus every fixpoint equation $X = FX$ has a solution.)

1.15. Let M be any other fixed point combinator, i.e. assume that $F(MF) =_\beta MF$ holds for all F. Show that M has no normal form.

1.16. Define the predecessor function in the untyped lambda-calculus.

1.17. (B. Maurey, J.-L. Krivine.) Let $C = \lambda xy.(xF(\lambda z\mathbf{1}))(yF(\lambda z\mathbf{0}))$, where $F = \lambda fg.gf$. Show that C defines the function

$$c(m,n) = \begin{cases} 1, & \text{if } m \leq n; \\ 0, & \text{otherwise.} \end{cases}$$

How many steps are needed to reduce $C\mathbf{c}_m\mathbf{c}_n$ to normal form? Will the same hold if we define c using Proposition 1.7.4?

1.18. (From [74].) Find λI-terms P_1, P_2 (projections) such that $P_1\langle \mathbf{c}_m, \mathbf{c}_n\rangle =_\beta \mathbf{c}_m$ and $P_2\langle \mathbf{c}_m, \mathbf{c}_n\rangle =_\beta \mathbf{c}_n$, for all m, n.

1.19. Show that the following functions are λ-definable:

- For each n, the function $f(i, m_1, \ldots, m_n) = m_i$.
- Integer division, i.e. a function f such that $f(mn, m) = n$, for all $m, n \neq 0$.
- Integer square root, i.e. a function f such that $f(n^2) = n$, for all n.

1.20. Assume that $M \twoheadrightarrow_\beta z$. Show that if $M[z := \lambda x N] \twoheadrightarrow_\beta \lambda u Q$, where N is normal, then $\lambda u Q = \lambda x N$. Will this remain true if we replace $\twoheadrightarrow_\beta$ by $\twoheadrightarrow_{\beta\eta}$?

1.21. For $n \in \mathbb{N}$ put $d_n = \mathrm{succ}^n(\mathbf{c}_0)$. Assume the following:[4]

> Every primitive recursive function is definable by a term F such that all applications $Fd_{n_1} \ldots d_{n_m}(\lambda v.\,\mathrm{false})\mathrm{true}$ are strongly normalizing.

Prove the claim in Remark 1.7.6. *Hint:* Use Exercise 1.20.

1.22. Prove that β-equality is undecidable. *Hint:* See Corollary 1.7.7(i).

[4]The proof of this is beyond our reach at the moment. We return to it in Exercise 11.28

Chapter 3

Simply typed λ-calculus

It is always a central issue in logic to determine if a formula is *valid* with respect to a certain semantics. Or, more generally, if a set of assumptions entails a formula in all models. In the "semantic tradition" of classical logic, this question is often the primary subject, and the construction of sound and complete proof systems is mainly seen as a tool for determining validity. In this case, ***provability*** of formulas and judgements is the only proof-related question of interest. One asks whether a proof exists, but not necessarily which proof it is.

In proof theory the perspective is different. We want to study the structure of proofs, compare various proofs, identify some of them, and distinguish between others. This is particularly important for constructive logics, where a proof (construction), not semantics, is the ultimate criterion.

It is thus natural to ask for a convenient proof notation. We can for instance write $M : \varphi$ to denote that M is a proof of φ. In presence of additional assumptions Γ, we may enhance this notation to

$$\Gamma \vdash M : \varphi.$$

Now, if M and N are proofs of $\varphi \to \psi$ and φ, respectively, then the proof of ψ obtained using $(\to\mathrm{E})$ could be denoted by something like $@(M, N) : \psi$, or perhaps simply written as $MN : \psi$. This gives an "annotated" rule $(\to\mathrm{E})$

$$\frac{\Gamma \vdash M : \varphi \to \psi \quad \Gamma \vdash N : \varphi}{\Gamma \vdash MN : \psi}$$

When trying to design an annotated version of (Ax), one discovers that it is also convenient to use names for assumptions so that e.g.,

$$x : \varphi,\ y : \psi \vdash x : \varphi$$

represents the use of the first assumption. This idea also comes in handy when we want to annotate rule $(\to\mathrm{I})$. The result of discharging an assumption x

in a proof M can be then written for example as $\sharp x\, M$, or $\xi x\, M$, or... why don't we try lambda?

$$\frac{\Gamma, x : \varphi \vdash M : \psi}{\Gamma \vdash \lambda x\, M : \varphi \to \psi}$$

Yes, what we get is lambda-notation. To use the words of a famous writer in an entirely different context, the similarity is not intended and not accidental. It is unavoidable. Indeed, a proof of an implication represents a construction, and according to the BHK interpretation, a construction of an implication is a function.

However, not every lambda-term can be used as a proof notation. For instance, the self-application xx does not seem to represent any propositional proof, no matter what the assumption annotated by x is. So before we explore the analogy between proofs and terms (which will happen in Chapter 4) we must look for the appropriate subsystem of lambda-calculus.

As we said, the BHK interpretation identifies a construction of an implication with a function. In mathematics, a function f is usually defined on a certain domain A and ranges over a co-domain B. This is written as $f : A \to B$. Similarly, a construction of a formula $\varphi \to \psi$ can only be applied to designated arguments (constructions of the premise). Then the result is a construction of the conclusion, i.e. it is again of a specific *type*.

In lambda-calculus, one introduces types to describe the functional behaviour of terms. An application MN is only possible when M has a *function type* of the form $\sigma \to \tau$ and N has type σ. The result is of type τ. This is a type discipline quite like that in strictly typed programming languages.

The notion of type assignment expressing functionality of terms has been incorporated into combinatory logic and lambda-calculus almost from the very beginning, and a whole spectrum of typed calculi has been investigated since then. In this chapter we introduce the most basic formalization of the concept of type: system $\lambda_{\to}$.

3.1. Simply typed λ-calculus à la Curry

We begin with the *simply typed λ-calculus à la Curry*, where we deal with the same ordinary lambda-terms as in Chapter 1.

3.1.1. DEFINITION.

(i) An implicational propositional formula is called a *simple type*. The set of all simple types is denoted by $\Phi_{\to}$.

(ii) An *environment* is a finite set of pairs of the form $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$, where x_i are distinct λ-variables and τ_i are types. In other words, an environment is a finite partial function from variables to types.[1]

[1] Occasionally, we may talk about an "infinite environment," and this means that a certain finite subset is actually used.

Thus, if $(x : \tau) \in \Gamma$ then we may write $\Gamma(x) = \tau$. We also have:

$$\mathrm{dom}(\Gamma) = \{x \in \Upsilon \mid (x : \tau) \in \Gamma, \text{ for some } \tau\};$$
$$\mathrm{rg}(\Gamma) = \{\tau \in \Phi_{\to} \mid (x : \tau) \in \Gamma, \text{ for some } x\}.$$

It is quite common in the literature to consider a variant of simply typed lambda-calculus where all types are built from a single type variable (which is then called a type *constant*). The computational properties of such a lambda-calculus are similar to those of our $\lambda_{\to}$. But from the "logical" point of view the restriction to one type constant is not as interesting, cf. Exercise 4.10.

NOTATION. The abbreviation $\tau^n \to \sigma$ is used for $\tau \to \cdots \to \tau \to \sigma$, where τ occurs n times. An environment $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is often simply written as $x_1 : \tau_1, \ldots, x_n : \tau_n$. If $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \varnothing$ then we may also write Γ, Γ' for $\Gamma \cup \Gamma'$. In particular, $\Gamma, x : \tau$ stands for $\Gamma \cup \{x : \tau\}$, where it is assumed that $x \notin \mathrm{dom}(\Gamma)$. Similar conventions will be used in later chapters.

3.1.2. DEFINITION. A *judgement* is a triple, written $\Gamma \vdash M : \tau$, consisting of an environment, a lambda-term and a type. The rules in Figure 3.1 define the notion of a *derivable* judgement of system $\lambda_{\to}$. (One has to remember that in rules (Var) and (Abs) the variable x is not in the domain of Γ.) If $\Gamma \vdash M : \tau$ is derivable then we say that M *has type* τ *in* Γ, and we write $\Gamma \vdash_{\lambda_{\to}} M : \tau$ or just $\Gamma \vdash M : \tau$ (cf. Definition 2.2.1).

(Var)	$\Gamma, x : \tau \vdash x : \tau$
(Abs)	$\dfrac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x\, M) : \sigma \to \tau}$
(App)	$\dfrac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$

FIGURE 3.1: THE SIMPLY TYPED LAMBDA-CALCULUS $\lambda_{\to}$

3.1.3. EXAMPLE. Let σ, τ, ρ be arbitrary types. Then:

(i) $\vdash \mathbf{I} : \sigma \to \sigma$;

(ii) $\vdash \mathbf{K} : \sigma \to \tau \to \sigma$;

(iii) $\vdash \mathbf{S} : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$.

A type assignment of the form $M : \tau \to \sigma$ can of course be explained as "M is a function with the domain τ and co-domain σ". But we must understand that this understanding of a "domain" and "co-domain" is not set-theoretic. In Curry-style typed calculi, types are more adequately decribed as predicates or specifications (to be satisfied by terms) than as set-theoretical function spaces. The meaning of $f : A \to B$ in set theory is that arguments of f are exactly the elements of A, and that all values must belong to B. In contrast, $M : \tau \to \sigma$ only means that M applied to an argument of type τ must yield a result of type σ.

We conclude this section with a brief review of some of the basic properties of system $\lambda_{\to}$.

3.1.4. LEMMA.

(i) *If* $\Gamma \vdash M : \tau$ *then* $\mathrm{FV}(M) \subseteq \mathrm{dom}(\Gamma)$.

(ii) *If* $\Gamma,\, x{:}\tau \vdash M : \sigma$ *and* $y \notin \mathrm{dom}(\Gamma) \cup \{x\}$ *then* $\Gamma,\, y{:}\tau \vdash M[x := y] : \sigma$.

PROOF. Both parts are shown by induction with respect to the length of M. As an example we treat the case of abstraction in part (ii).

Suppose that $M = \lambda z M'$ and $\sigma = \sigma'' \to \sigma'$ and that we have derived $\Gamma,\, x{:}\tau \vdash M : \sigma$ from $\Gamma,\, x{:}\tau,\, z{:}\sigma'' \vdash M' : \sigma'$. If $z \neq y$ then from the induction hypothesis we know that $\Gamma,\, y{:}\tau,\, z{:}\sigma'' \vdash M'[x := y] : \sigma'$, whence $\Gamma,\, y{:}\tau \vdash \lambda z\, M'[x := y] : \sigma$. Now note that $\lambda z M'[x := y] = (\lambda z M')[x := y]$.

If $z = y$ and $\Gamma,\, x{:}\tau,\, y{:}\sigma'' \vdash M' : \sigma'$ then we can choose a variable $v \notin \mathrm{dom}(\Gamma) \cup \{x, y\}$ and obtain $\Gamma,\, x{:}\tau,\, v{:}\sigma'' \vdash M'[y := v] : \sigma'$ from the induction hypothesis. The next application of the induction hypothesis yields $\Gamma,\, y{:}\tau,\, v{:}\sigma'' \vdash M'[y := v][x := y] : \sigma'$, because the size of the term has not been increased. It follows that $\Gamma, y{:}\tau \vdash \lambda v.\, M'[y := v][x := y] : \sigma$. The reader will easily verify that $\lambda v. M'[y := v][x := y] = M[x := y]$. □

The following lemma is a direct consequence of the "syntax-oriented" character of the rules.

3.1.5. LEMMA (Generation lemma). *Suppose that* $\Gamma \vdash M : \tau$.

(i) *If* M *is a variable* x *then* $\Gamma(x) = \tau$.

(ii) *If* M *is an application* PQ *then* $\Gamma \vdash P : \sigma \to \tau$, *and* $\Gamma \vdash Q : \sigma$, *for some* σ.

(iii) *If* M *is an abstraction* $\lambda x N$ *and* $x \notin \mathrm{dom}(\Gamma)$ *then* $\tau = \tau_1 \to \tau_2$, *where* $\Gamma,\, x{:}\tau_1 \vdash N : \tau_2$.

PROOF. The last step in the derivation of $\Gamma \vdash M : \tau$ is determined by the shape of M. This is all that is needed to prove the first two parts. In part (iii) the last rule must be (Abs) applied to $\Gamma, y:\tau_1 \vdash N_1 : \tau_2$, where $\lambda x N = \lambda y N_1$, and it can happen that $y \neq x$. Thus, by Lemma 1.2.20(v), we have $\Gamma, y:\tau_1 \vdash N[x := y] : \tau_2$, and then from Lemma 3.1.4(ii) we obtain $\Gamma, x:\tau_1 \vdash N[x := y][y := x] : \tau_2$. But $N[x := y][y := x] = N$, by Lemmas 1.2.20(iii) and 1.2.5(iii). □

3.1.6. LEMMA.

(i) *If* $\Gamma \vdash M : \sigma$ *and* $\Gamma(x) = \Gamma'(x)$ *for all* $x \in \mathrm{FV}(M)$ *then* $\Gamma' \vdash M : \sigma$.

(ii) *If* $\Gamma, x : \tau \vdash M : \sigma$ *and* $\Gamma \vdash N : \tau$ *then* $\Gamma \vdash M[x := N] : \sigma$.

PROOF. We proceed by induction with respect to the size of M. In the proof of (i) we only consider the case when $M = \lambda y M'$ and $\sigma = \sigma_1 \to \sigma_2$. If y is chosen so that $y \notin \mathrm{FV}(\Gamma) \cup \mathrm{FV}(\Gamma')$ then by Lemma 3.1.5(iii) we have $\Gamma, y:\sigma_1 \vdash M' : \sigma_2$. The induction hypothesis yields $\Gamma', y:\sigma_1 \vdash M' : \sigma_2$, which in turn gives $\Gamma' \vdash M : \sigma$.

The proof of part (ii) is also routine. Note that we use part (i) in the case of abstraction. □

The following result establishes the correctness of the type assignment system. A well-typed expression remains well-typed after reductions. In particular, no run-time error can be caused by an ill-typed function application. ("Well-typed programs do not go wrong.").

3.1.7. THEOREM (Subject reduction). *If* $\Gamma \vdash M : \sigma$ *and* $M \twoheadrightarrow_\beta N$, *then* $\Gamma \vdash N : \sigma$.

PROOF. By induction with respect to the definition of $\twoheadrightarrow_\beta$. We consider the base case when M is a redex, $M = (\lambda x\, P)Q$, and $N = P[x := Q]$. Without loss of generality we can assume that $x \notin \mathrm{dom}(\Gamma)$, so by the generation lemma we have $\Gamma, x : \tau \vdash P : \sigma$ and $\Gamma \vdash Q : \tau$. From Lemma 3.1.6(ii) we obtain $\Gamma \vdash P[x := Q] : \sigma$. □

3.1.8. DEFINITION. The *substitution of type τ for type variable p in type σ*, written $\sigma[p := \tau]$, is defined by:

$$\begin{array}{rcl} p[p := \tau] & = & \tau; \\ q[p := \tau] & = & q, \text{ if } q \neq p; \\ (\sigma_1 \to \sigma_2)[p := \tau] & = & \sigma_1[p := \tau] \to \sigma_2[p := \tau]. \end{array}$$

The notation $\Gamma[p := \tau]$ stands for $\{(x : \sigma[p := \tau]) \mid (x : \sigma) \in \Gamma\}$. Similar notation applies for equations, sets of equations etc.

The following shows that the type variables range over all types; this is a limited form of *polymorphism* (cf. Chapter 11).

3.1.9. PROPOSITION. *If* $\Gamma \vdash M : \sigma$, *then* $\Gamma[p := \tau] \vdash M : \sigma[p := \tau]$.

PROOF. By induction on the derivation of $\Gamma \vdash M : \sigma$. □

3.2. Type reconstruction algorithm

A term $M \in \Lambda$ is *typable* if there are Γ and σ such that $\Gamma \vdash M : \sigma$. The set of typable terms is a proper subset of the set of all λ-terms. It is thus a fundamental problem to determine exactly which terms can be assigned types in system $\lambda_{\to}$ and how to find these types effectively. In fact, one can consider a number of decision problems arising from the analysis of the ternary predicate "$\Gamma \vdash M : \tau$". The following definition makes sense for every type assignment system deriving judgements of this form.

3.2.1. DEFINITION.

(i) The *type checking* problem is to decide whether $\Gamma \vdash M : \tau$ holds, for a given environment Γ, a term M, and a type τ.

(ii) The *typability* problem, also called *type reconstruction* problem, is to decide if a given term M is typable.

(iii) The *type inhabitation* problem, also called *type emptiness* problem, is to decide, for a given type τ, whether there exists a closed term M, such that $\vdash M : \tau$ holds. (Then we say that τ is *non-empty*, and has the *inhabitant* M.)

The type inhabitation problem will be discussed in Chapter 4. In this section we consider typability and type checking. At first sight it might seem that determining whether a given term has a given type in a given environment could be easier than determining whether it has any type at all. This impression however is generally wrong. For many type assignment systems, typability is easily reducible to type checking. Indeed, to determine if a term M is typable, where $\mathrm{FV}(M) = \{x_1, \ldots, x_n\}$, we may ask if

$$x_0 : p \vdash \mathbf{K}x_0(\lambda x_1 \ldots x_n . M) : p,$$

and this reduces typability to type checking. In fact, in the simply typed case, the two problems are equivalent (Exercise 3.11), although reducing the latter to the former is not as easy. But for some type assignment systems, the two problems are not equivalent: compare Proposition 13.4.3 and Theorem 13.4.4.

We now show how the typability problem can be reduced to unification[2] over the signature consisting only of the binary function symbol $\to$. Terms over this signature are identified with simple types. For every term M we define by induction

- a system of equations E_M;
- a type τ_M.

[2]See the Appendix for definitions related to terms and unification.

The idea is as follows: E_M has a solution iff M is typable, and τ_M is (informally) a pattern of a type for M. Type variables (unknowns) occurring in E_M are of two sorts: some of them, denoted p_x, correspond to types of free variables x of M, the other variables are auxiliary.

3.2.2. DEFINITION.

(i) If M is a variable x, then $E_M = \varnothing$ and $\tau_M = p_x$, where p_x is a fresh type variable.

(ii) Let M be an application PQ. First rename all auxiliary variables in E_Q and τ_Q so that auxiliary variables used by E_P and τ_P are distinct from those occurring in E_Q and τ_Q. Then define $\tau_M = p$, where p is a fresh type variable, and $E_M = E_P \cup E_Q \cup \{\tau_P = \tau_Q \to p\}$.

(iii) If M is an abstraction $\lambda x P$, then we define $E_M = E_P[p_x := p]$ and $\tau_M = p \to \tau_P[p_x := p]$, where p is a fresh variable.

In the definition above, it should be assumed that the renamings and the choice of "fresh" variables are made according to a certain systematic pattern, so that E_M is defined in a unique way for each M. An alternative is to think about M as a fixed alpha-representative (a pre-term) where the choice of bound and free variables is made so that no confusion is possible.

3.2.3. LEMMA.

(i) *If $\Gamma \vdash M : \rho$, then there exists a solution S of E_M, such that $\rho = S(\tau_M)$ and $S(p_x) = \Gamma(x)$, for all variables $x \in \mathrm{FV}(M)$.*

(ii) *Let S be a solution of E_M, and let Γ be such that $\Gamma(x) = S(p_x)$, for all $x \in \mathrm{FV}(M)$. Then $\Gamma \vdash M : S(\tau_M)$.*

PROOF. Induction with respect to the length of M. □

It follows that M is typable iff E_M has a solution. But E_M then has a principal solution, and this has the following consequence. (Here, $S(\Gamma)$ is the environment such that $S(\Gamma)(x) = S(\Gamma(x))$.)

3.2.4. DEFINITION. A pair (Γ, τ), consisting of an environment and a type, is a *principal pair* for M iff the following are equivalent for all Γ' and τ':

(i) $\Gamma' \vdash M : \tau'$;

(ii) $S(\Gamma) \subseteq \Gamma'$ and $S(\tau) = \tau'$, for some substitution S.

We then also say that τ is the *principal type* of M.

3.2.5. COROLLARY. *If a term M is typable, then there exists a principal pair for M. This principal pair is unique up to renaming of type variables.*

PROOF. Immediate from Lemma A.2.1. □

We conclude that a judgement $\Gamma \vdash M : \tau$ is derivable if and only if (Γ, τ) is a substitution instance of the principal pair. In this way, the principal pair provides a full characterization of all type assignments possible for M.

3.2.6. EXAMPLE.

- The principal type of $\mathbf{S}$ is $(p \to q \to r) \to (p \to q) \to p \to r$. The type $(p \to q \to p) \to (p \to q) \to p \to p$ can also be assigned to $\mathbf{S}$, but it is not principal.
- Type $\mathbf{int} = (p \to p) \to p \to p$ is the principal type of Church numerals $\mathbf{c}_n$, for $n \geq 2$. For $\mathbf{0}$ and $\mathbf{1}$ the principal types are respectively $p \to q \to q$ and $(p \to q) \to p \to q$. But every Church numeral can also be assigned the type $((p \to q) \to p \to q) \to (p \to q) \to p \to q$.

3.2.7. THEOREM. *Typability and type checking in the simply typed lambda-calculus are decidable in polynomial time.*

PROOF. The system of equations E_M can be constructed in logarithmic space (and thus also in polynomial time) from M. Thus, by Lemma 3.2.3, typability reduces in logarithmic space to unification, which is decidable in polynomial time (Theorem A.5.4).

To decide if $\Gamma \vdash M : \tau$ holds, consider a signature containing (in addition to the binary symbol $\to$) all free variables occurring in Γ and τ as constant symbols. It is now enough to extend E_M to include the equations $\tau_M = \tau$ and $p_x = \Gamma(x)$, for $x \in \mathrm{FV}(M)$. The extended system of equations has a solution if and only if $\Gamma \vdash M : \tau$. □

3.2.8. REMARK (Related problems). The typability problem is often written as "$? \vdash M : ?$", and the type inhabitation problem is abbreviated "$\vdash ? : \tau$". This notation can be used for other related problems, as one can choose to replace various parts of our ternary predicate by question marks, and choose the environment to be empty or not. A little combinatorics shows that we have a total of 12 problems. Out of these 12 problems, four are completely trivial, since the answer is always "yes":

$$? \vdash ? : ? \qquad \Gamma \vdash ? : ? \qquad \vdash ? : ? \qquad ? \vdash ? : \tau.$$

Thus we end up with eight non-trivial problems, as follows:

(i) $? \vdash M : ?$ (typability);

(ii) $\vdash M : ?$;

(iii) $\Gamma \vdash M : ?$;

(iv) $\Gamma \vdash M : \tau$ (type checking);

(v) $\vdash M : \tau$;

(vi) $? \vdash M : \tau$;

(vii) $\vdash ? : \tau$ (inhabitation);

(viii) $\Gamma \vdash ? : \tau$;

We have already noticed that problem (i) reduces to (iv). In fact, for the simply typed lambda-calculus, all problems (i)–(vi) are equivalent to unification, and thus also to each other, with respect to logarithmic-space reductions (Exercise 3.11). Thus, all these problems are PTIME-complete. We will see in Chapter 4 that (vii) and (viii) are complete for polynomial space.

3.3. Simply typed λ-calculus à la Church

Typed lambda-calculi usually occur in two variants, called *Curry-style* and *Church-style* systems. In the previous section we have seen an example of a Curry-style system. In such calculi, types are assigned (or not) to ordinary type-free lambda-terms, according to a set of rules. In this way, one term can be assigned more than one type.

The idea of a typed calculus à la Church is different. In the "orthodox" approach, all the type information is contained in a term, as follows.[3] For each $\sigma \in \Phi_{\rightarrow}$, let Υ_σ be a separate denumerable set of variables. Define the sets Λ_σ of simply typed terms of type σ so that $\Upsilon_\sigma \subseteq \Lambda_\sigma$ and:

- If $M \in \Lambda_{\sigma\rightarrow\tau}$ and $N \in \Lambda_\sigma$ then $(MN) \in \Lambda_\tau$;
- If $M \in \Lambda_\tau$ and $x^\sigma \in \Upsilon_\sigma$ then $(\lambda x^\sigma M) \in \Lambda_{\sigma\rightarrow\tau}$.

The set of simply typed terms is then taken as the union of all Λ_σ.

It is sometimes convenient to think of typed lambda-terms this way, but nowadays it is more customary to define Church-style calculi in a slightly different manner. Instead of assuming that the set of variables is partitioned into disjoint sets indexed by types one uses environments to declare types of free variables as in the system à la Curry. But types of bound variables remain part of the term syntax.

3.3.1. DEFINITION.

(i) *Raw terms* of Church-style $\lambda_{\rightarrow}$ are defined by the following rules:

- An object variable is a raw term;

[3]In case the reader cannot remember which approach is named after Church and which one after Curry, we recommend *Walukiewicz's test*: Observe that the name "Church" is longer than "Curry". And Church-style typed terms are longer too.

- If M, N are raw terms then the *application* (MN) is a raw term;
- If M is a raw term, x is a variable, and σ is a type then the *abstraction* $(\lambda x{:}\sigma M)$ is a raw term.

(ii) *Free variables* of a raw term M are defined as follows.

$$\begin{array}{lll} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(\lambda x{:}\sigma P) & = & \mathrm{FV}(P) - \{x\} \\ \mathrm{FV}(PQ) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q) \end{array}$$

If $\mathrm{FV}(M) = \varnothing$ then M is called *closed.*

(iii) The variable x is considered *bound* in the term $(\lambda x{:}\sigma\, P)$. We identify raw terms which differ only in their bound variables.[4]

3.3.2. CONVENTION. We adopt similar terminology, notation, and conventions for raw terms as for untyped λ-terms, see Chapter 1, *mutatis mutandis.* In particular, we omit parentheses if this does not create confusion, and we use dot notation, so that e.g. $\lambda x{:}\tau.\, yx$ stands for $(\lambda x{:}\tau\,(yx))$. In addition, to enhance readability, we sometimes write x^τ instead of $x:\tau$, like for instance in $\lambda x^{p\to q\to r}\lambda y^{p\to q}\lambda z^p.xz(yz)$.

The following definition of substitution on raw terms takes into account our assumption of identifying alpha-equivalent expressions.

3.3.3. DEFINITION. The *substitution* of a raw term N for x in M, written $M[x := N]$, is defined as follows:

$$\begin{array}{lll} x[x := N] & = & N; \\ y[x := N] & = & y, \text{ if } x \neq y; \\ (PQ)[x := N] & = & P[x := N]Q[x := N]; \\ (\lambda y{:}\sigma.P)[x := N] & = & \lambda y{:}\sigma.P[x := N], \text{ where } x \neq y \text{ and } y \notin \mathrm{FV}(N). \end{array}$$

The notion of β-reduction for Church-style expressions is also similar to that for Curry-style terms. In the definition below, the notion of "compatible" applies to Church-style syntax (see the explanation following Definition 1.3.1).

3.3.4. DEFINITION. The relation $\to_\beta$ (*single step β-reduction*) is the least compatible relation on raw terms, such that

$$(\lambda x{:}\sigma P)Q \to_\beta P[x := Q]$$

The notation $\twoheadrightarrow_\beta$ and $=_\beta$ is used accordingly, cf. Remark 1.3.4.

3.3.5. DEFINITION. We say that M is a *term* of type τ in Γ, and we write $\Gamma \vdash M : \tau$, when $\Gamma \vdash M : \tau$ can be derived using the rules in Figure 3.2.

[4] Strictly speaking, we should proceed as in the case of λ-terms and define a notion of raw pre-terms, then define substitution and α-equivalence on these, and finally adopt the convention that by a term we always mean the α-equivalence class, see Section 1.2.

$$\text{(Var)} \qquad \Gamma, x : \tau \vdash x : \tau$$

$$\text{(Abs)} \qquad \frac{\Gamma,\, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x{:}\sigma\, M) : \sigma \to \tau}$$

$$\text{(App)} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

FIGURE 3.2: THE SIMPLY TYPED LAMBDA-CALCULUS À LA CHURCH

3.3.6. EXAMPLE. Let σ, τ, ρ be arbitrary simple types. Then:

(i) $\vdash \lambda x^{\sigma} x : \sigma \to \sigma$;

(ii) $\vdash \lambda x^{\sigma} \lambda y^{\tau}. x : \sigma \to \tau \to \sigma$;

(iii) $\vdash \lambda x^{\sigma \to \tau \to \rho} \lambda y^{\sigma \to \tau} \lambda z^{\sigma}. xz(yz) : (\sigma \to \tau \to \rho) \to (\sigma \to \tau) \to \sigma \to \rho$.

As in the Curry-style calculus, we have a subject reduction theorem:

3.3.7. THEOREM. *If* $\Gamma \vdash M : \sigma$ *and* $M \twoheadrightarrow_{\beta} N$ *then* $\Gamma \vdash N : \sigma$.

PROOF. Similar to the proof of Theorem 3.1.7. □

It is sometimes convenient to consider also η-reduction and η-equality on typed terms.

3.3.8. DEFINITION. The relation $\to_{\eta}$ is the smallest compatible relation on raw terms, such that

$$\lambda x{:}\sigma.\, Mx \to_{\eta} M,$$

whenever $x \notin \mathrm{FV}(M)$.

It is not difficult to see that $\to_{\eta}$ preserves types, i.e. that an analogue of Theorem 3.3.7 holds for eta-reductions.

3.4. Church versus Curry typing

As mentioned at the beginning of Section 3.3, the principle of typing à la Church (at least in the orthodox way) is that types of all variables and terms are "fixed". The full type information is "built into" an expression, and a given well-formed term is correctly typed by definition. There is no issue of typability because the type of a term is simply a part of it. This corresponds to the use of types in programming languages like e.g. Pascal. In such languages, it is the responsibility of the programmer to provide proper types of all identifiers, functions, etc. In contrast, Curry-style typing resembles ML or Haskell, where a compiler or interpreter does the type inference.

Because of the difference above, $\lambda_\rightarrow$ à la Curry and other similar systems are often called *type assignment* systems, in contrast to $\lambda_\rightarrow$ à la Church and similar systems which are called *typed* systems.

Our formulation of simply typed lambda-calculus à la Church is however halfway between the Curry style and the "orthodox" Church style. Types of bound variables are "embedded" in terms, but types of free variables are declared in the environment rather than being part of syntax. A raw term becomes a true "typed Church-style term" only within an environment which determines types of its free variables. Then, unlike in Curry style (cf. Proposition 3.1.9) a type of a term is unique.

3.4.1. PROPOSITION. *If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$ in the simply-typed λ-calculus à la Church then $\sigma = \tau$.*

PROOF. Induction with respect to M. □

3.4.2. CONVENTION. In what follows, we often refer to Church-style terms without explicitly mentioning the environment, but if not stated otherwise it is implicitly assumed that some environment is given, and types of all variables are known. By Proposition 3.4.1 we can thus assume that every term has a uniquely determined type. We then proceed as if we actually dealt with "orthodox" Church-style terms.

In order to improve readability we sometimes write types of terms as superscripts, like in $(M^{\sigma\rightarrow\tau}N^{\sigma})^{\tau}$. This notation is not part of the syntax and is only used informally to stress that e.g. M has type $\sigma \rightarrow \tau$ in a certain fixed environment.

Similar conventions are used in the later chapters, whenever Church-style systems are discussed. Also the word *term* always refers to a well-typed expression, cf. Definition 3.3.5.

Although the simply typed λ-calculi à la Curry and Church are different, one has the feeling that essentially the same thing is going on. To a large extent this intuition is correct. A Church-style term M can in fact be seen as a linear notation for a type derivation that assigns a type to a Curry-style term. This term is the "core" or "skeleton" of M, obtained by erasing the domains of abstractions.

3.4.3. DEFINITION. The *erasing* map $|\cdot|$ from Church-style to Curry-style terms is defined as follows:

$$\begin{array}{rcl}|x| & = & x;\\ |MN| & = & |M||N|;\\ |\lambda x{:}\sigma\, M| & = & \lambda x|M|.\end{array}$$

3.4.4. PROPOSITION.

(i) *If* $M \to_\beta N$ *then* $|M| \to_\beta |N|$.

(ii) *If* $\Gamma \vdash M : \sigma$ *à la Church then* $\Gamma \vdash |M| : \sigma$ *à la Curry.*

PROOF. For (i) prove by induction on M that

$$|M[x := N]| = |M|[x := |N|] \qquad (*)$$

Then proceed by induction on the definition of $M \to_\beta N$ using $(*)$.

Part (ii) follows by induction on the derivation of $\Gamma \vdash M : \sigma$. □

Conversely, one can "lift" every Curry derivation to a Church one.

3.4.5. PROPOSITION. *For all* $M, N \in \Lambda$:

(i) *If* $M \to_\beta N$ *and* $M = |M'|$ *then* $M' \to_\beta N'$, *for some* N' *such that* $|N'| = N$.

(ii) *If* $\Gamma \vdash M : \sigma$ *then there is a Church-style term* M' *with* $|M'| = M$ *and* $\Gamma \vdash M' : \sigma$.

PROOF. By induction on $M \to_\beta N$ and $\Gamma \vdash M : \sigma$, respectively. □

The two propositions above allow one to "translate" various properties of Curry-style typable lambda-terms to analogous properties of Church-style typed lambda-terms, or conversely. For instance, strong normalization for one variant of $\lambda_\to$ implies strong normalization for the other (Exercise 3.15). But one has to be cautious with such proof methods (Exercise 3.19).

3.5. Normalization

In this section we show that all simply typed terms have normal forms. Even more, all such terms M are *strongly* normalizing, i.e. there exists no infinite reduction $M = M_1 \to_\beta M_2 \to_\beta \cdots$ In other words, no matter how we evaluate a well-typed expression, we must eventually reach a normal form. In programming terms: a program in a language based on the simply typed lambda-calculus can only diverge as a result of an explicit use of a programming construct such as a loop, a recursive call, or a circular data type.

Strong normalization makes certain properties easier to prove. Newman's lemma below is a good example. Another example is deciding equality of typed terms by comparing their normal forms (Section 3.7). Yet another related aspect is normalization of proofs, which will be seen at work in the chapters to come.

The results of this section hold for both Church-style and Curry-style terms. Our proofs are for Church style, but the Curry style variant can be easily derived (Exercise 3.15). In what follows, we assume that types of all variables are fixed, so that we effectively work with the "orthodox" Church style (cf. Convention 3.4.2).

3.5.1. THEOREM. *Every term of Church-style $\lambda_{\rightarrow}$ has a normal form.*

PROOF. We show that a certain reduction strategy is normalizing. The idea is to always reduce a redex of a most complex type available and to begin from the right if there are several candidates. To make this idea precise, define the *degree* $\delta(\tau)$ of a type τ by:

$$\begin{array}{lcl}\delta(p) & = & 0;\\ \delta(\tau\to\sigma) & = & 1+\max(\delta(\tau),\delta(\sigma)).\end{array}$$

The *degree* $\delta(\Delta)$ of a redex $\Delta = (\lambda x^{\tau} P^{\rho})R$ is $\delta(\tau\to\rho)$. If a term M is not in normal form then we define

$$m_M = (\delta_M, n_M),$$

where $\delta_M = \max\{\delta(\Delta) \mid \Delta \text{ is a redex in } M\}$ and n_M is the number of redex occurrences in M of degree δ_M. For $M \in \mathrm{NF}_\beta$ put $m_M = (0,0)$. The proof is by induction on lexicographically ordered pairs m_M.

If $M \in \mathrm{NF}_\beta$ the assertion is trivially true. If $M \notin \mathrm{NF}_\beta$, let Δ be the rightmost redex in M of maximal degree δ (we determine the position of a subterm by the position of its first symbol, i.e. the rightmost redex means the redex which *begins* as much to the right as possible).

Let M' be obtained from M by reducing the redex Δ. The term M' may in general have more redexes than M. But we claim that the number of redexes of degree δ in M' is smaller than in M. Indeed, the redex Δ has disappeared, and the reduction of Δ may only create new redexes of degree less than δ. To see this, note that the number of redexes can increase by either copying existing redexes or by creating new ones. The latter happens when a non-abstraction A occurs in a context AB, and it is turned into an abstraction by the reduction of Δ. This is only possible when A is a variable or $A = \Delta$. It follows that one of the following cases must hold:

1. The redex Δ has form $(\lambda x^{\tau} \ldots x P^{\rho} \ldots)(\lambda y^{\rho} Q^{\mu})^{\tau}$, where $\tau = \rho\to\mu$, and it reduces to $\ldots(\lambda y^{\rho} Q^{\mu})P^{\rho}\ldots$ The new redex $(\lambda y^{\rho} Q^{\mu})P^{\rho}$ is of degree $\delta(\tau) < \delta$.

2. We have $\Delta = (\lambda x^\tau \lambda y^\rho . R^\mu)P^\tau$, occurring in the context $\Delta^{\rho\to\mu}Q^\rho$. The reduction of Δ to $\lambda y^\rho R_1^\mu$, for some R_1, creates a new redex $(\lambda y^\rho R_1^\mu)Q^\rho$ of degree $\delta(\rho \to \mu) < \delta(\tau \to \rho \to \mu) = \delta$.

3. The last case is when $\Delta = (\lambda x^\tau x)(\lambda y^\rho P^\mu)$, with $\tau = \rho \to \mu$, and it occurs in the context $\Delta^\tau Q^\rho$. The reduction creates the new redex $(\lambda y^\rho P^\mu)Q^\rho$ of degree $\delta(\tau) < \delta$.

The other way to add redexes is by copying. If $\Delta = (\lambda x{:}\tau . P^\rho)Q^\tau$, and the term P contains more than one free occurrence of x, then all redexes in Q are multiplied by the reduction. But we have chosen Δ to be the rightmost redex of degree δ, and thus all redexes in Q must be of smaller degrees, because they are to the right of Δ.

Thus, in all cases $m_M > m_{M'}$, so by the induction hypothesis M' has a normal form, and then M also has a normal form. □

Theorem 3.5.1 states that every typed term is normalizing. We now set out to show that every term is in fact *strongly* normalizing, i.e. that *every* reduction sequence from the term must eventually terminate. Our aim is to infer the strong property from the weak one, with the help of the conservation property of $\lambda\mathbf{I}$-terms (Corollary 1.6.7). This can be done by translating an arbitrary typed λ-term M into a $\lambda\mathbf{I}$-term $\iota(M)$, of the same type, such that $\iota(M) \in \mathrm{SN}$ implies $M \in \mathrm{SN}$.

3.5.2. DEFINITION. For every propositional variable p and every type σ, we choose a fixed λ-variable $k_{p,\sigma}$ of type $p \to \sigma \to p$. If M is a variable or an application then $\iota(M)$ is defined as:

$$\iota(x) = x, \qquad \iota(PQ) = \iota(P)\iota(Q).$$

Otherwise our term is an abstraction, say of the form

$$M = \lambda x_1^{\sigma_1} \dots x_r^{\sigma_r} . N^{\tau_1 \to \dots \to \tau_m \to p},$$

where $r > 0$ and N is not an abstraction. In this case we define

$$\iota(M) = \lambda x_1^{\sigma_1} \dots x_r^{\sigma_r} y_1^{\tau_1} \dots y_m^{\tau_m} . k_{p,\sigma_1}(\cdots(k_{p,\sigma_r}(\iota(N)y_1 \dots y_m)x_r)\cdots)x_1.$$

Finally, we define a term $t(M)$ by replacing every $k_{p,\sigma}$ occurring in $\iota(M)$ by the appropriate version of the $\mathbf{K}$ combinator, namely $\mathbf{K}_{p,\sigma} = \lambda x^p \lambda y^\sigma . x$.

Note that $t(M) \twoheadrightarrow_{\beta\eta} M$. Note also that $\iota(M)$ is a typed $\lambda\mathbf{I}$-term, and thus it is strongly normalizing. We now want to prove that also $t(M) \in \mathrm{SN}$, by showing that reductions involving $\mathbf{K}_{p,\sigma}$ are not essential.

3.5.3. LEMMA. *If $M \notin \mathrm{SN}_\beta$ then M has an infinite reduction sequence where no redex of the form $\mathbf{K}_{p,\sigma}A$ is reduced.*

PROOF. If a redex of the form $\mathbf{K}_{p,\sigma}A$ is reduced to $\lambda y^\sigma A$, where p is a type variable, then we say that the reduction step is *of type* 1. Clearly, an infinite reduction sequence may not consist exclusively of type 1 steps. It would thus suffice to show that steps of type 1 can always be "postponed" by permuting them with other reductions. Unfortunately, this property is not true: In the context $\mathbf{K}_{p,\sigma}AB$, a reduction of type 1 creates a redex which can be reduced in the next step. These two steps cannot be permuted.

To solve this problem we first postpone reduction steps where a redex of the form $(\lambda y^\sigma A^p)B$, with $y \notin \mathrm{FV}(A)$, is reduced to A. Let us say that such reductions are *of type* 2. We write $M \to_2 M'$ to indicate that the reduction is of type 2, and we write $M \to_0 M'$ otherwise. We prove that $M \to_2 M' \to_0 M''$ implies $M \to_0 M''' \twoheadrightarrow_2 M''$ for some M'''.

Let $\Delta = (\lambda y^\sigma A^p)B^\sigma$ be the redex reduced in the step from $M \to_2 M'$. Since A cannot be an abstraction, and $y \notin \mathrm{FV}(A)$, the redex Σ reduced in the next step is not a "new" one, i.e. it is obtained from a redex Σ' in M (possibly containing Δ). It is left to the reader to check that the two reduction steps can easily be permuted. (There is a double arrow in $M''' \twoheadrightarrow_2 M''$, because Δ can be duplicated or erased by reducing Σ.)

The above allows us to postpone reduction steps of type 2, i.e. to conclude that there is an infinite reduction sequence without such steps. Using a similar argument, we can also postpone steps of type 1. Indeed, if $(\lambda x^p \lambda y^\sigma . x)A^p$ is reduced to $\lambda y^\sigma A$ then again the next redex is either "inside" or "outside" the term A. □

3.5.4. LEMMA. *Terms of the form $t(M)$ are strongly β-normalizing.*

PROOF. Suppose that $t(M) = M_0 \to_0 M_1 \to_0 M_2 \to_0 \cdots$ is an infinite reduction sequence where no redex of the form $\mathbf{K}_{p,\sigma}A$ is reduced. In this reduction sequence, the combinators $\mathbf{K}_{p,\sigma}$ behave just like variables. It follows that $\iota(M) = M_0' \to_\beta M_1' \to_\beta M_2' \to_\beta \cdots$ where M_i are obtained from M_i' by substituting $\mathbf{K}_{p,\sigma}$ for $k_{p,\sigma}$. But $\iota(M)$ is strongly normalizing. □

3.5.5. THEOREM. *The simply typed lambda-calculus has the strong normalization property: Any term is strongly normalizing.*

PROOF. If M is not β-normalizing then $t(M)$ has an infinite $\beta\eta$-reduction sequence $t(M) \twoheadrightarrow_{\beta\eta} M \twoheadrightarrow_\beta \cdots$ By Lemma 1.3.11, it must also have an infinite β-reduction sequence, contradicting Lemma 3.5.4. □

3.6. Church-Rosser property

Proving the Church-Rosser property for Church-style typed terms is not as obvious as it may seem (Exercise 3.19). Fortunately, the typed lambda-calculus is strongly normalizing, and under this additional assumption, it is enough to show the *weak Church-Rosser property*.

3.6.1. DEFINITION. Let $\to$ be a binary relation in a set A. Recall from Chapter 1 that $\to$ has the *Church-Rosser property* (CR) iff for all $a, b, c \in A$ such that $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$ there exists $d \in A$ with $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$. We say that the relation $\to$ has the *weak Church-Rosser property* (WCR) when $a \to b$ and $a \to c$ imply $b \twoheadrightarrow d$ and $c \twoheadrightarrow d$, for some d.

We also say that a binary relation $\to$ is *strongly normalizing* (SN) iff there is no infinite sequence $a_0 \to a_1 \to a_2 \to \cdots$

Clearly, CR implies WCR. The converse is not true, see Exercise 3.16. But the two properties coincide for strongly normalizing relations.

3.6.2. PROPOSITION (Newman's lemma). *Let $\to$ be a binary relation satisfying* SN. *If $\to$ satisfies* WCR, *then $\to$ satisfies* CR.

PROOF. Let $\to$ be a relation on a set A satisfying SN and WCR. As usual, a *normal form* is an $a \in A$ such that $a \not\to b$, for all $b \in A$.

Since $\to$ satisfies SN, any $a \in A$ reduces to a normal form. Call an element a *ambiguous* if a reduces to two distinct normal forms. It is easy to see that $\to$ satisfies CR if there are no ambiguous elements of A.

But for any ambiguous a there is another ambiguous a' such that $a \to a'$. Indeed, suppose $a \twoheadrightarrow b_1$ and $a \twoheadrightarrow b_2$ and let b_1, b_2 be different normal forms. Both of these reductions must make at least one step since b_1 and b_2 are distinct, so the reductions have form $a \to a_1 \twoheadrightarrow b_1$ and $a \to a_2 \twoheadrightarrow b_2$. If $a_1 = a_2$ we can choose $a' = a_1 = a_2$. If $a_1 \neq a_2$ we know by WCR that $a_1 \twoheadrightarrow b_3$ and $a_2 \twoheadrightarrow b_3$, for some b_3. We can assume that b_3 is a normal form.

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
a_1 \arrow[d, two heads] \arrow[dr, two heads] & & a_2 \arrow[dl, two heads] \arrow[d, two heads] \\
b_1 & b_3 & b_2
\end{tikzcd}$$

Since b_1 and b_2 are distinct, b_3 is different from b_1 or b_2 so we can choose $a' = a_1$ or $a' = a_2$. Thus, a has an infinite reduction sequence, contradicting strong normalization. Hence, there are no ambiguous terms. $\square$

Newman's lemma is a very useful tool for proving the Church-Rosser property. We will use it many times, and here is its debut.

3.6.3. THEOREM. *The Church-style simply typed λ-calculus has the Church-Rosser property.*

PROOF. By Newman's lemma it suffices to check that the simply typed lambda-calculus has the WCR property. This is almost immediate if we observe that two different β-redexes in a lambda-term can only be disjoint, or one is a part of the other. That is, redexes never "overlap" and if we reduce one of them we can still reduce the other. A formal proof can be done by induction with respect to the definition of β-reduction. □

The subject reduction property together with the Church-Rosser property and strong normalization imply that reduction of any typed λ-term terminates in a normal form of the same type, where the normal form is independent of the particular order of reduction chosen.

3.7. Expressibility

As we saw in the preceding section, every simply typed λ-term has a normal form, and the normalization process always terminates. To verify whether two given terms of the same type are beta-equal or not, it thus suffices to reduce them to normal form and compare the results. However, this straightforward algorithm is of unexpectedly high complexity. It requires time and space proportional to the size of normal forms of the input terms. As demonstrated by Exercises 3.20–3.22, a normal form of a term of length n can (in the worst case) be of size

$$\left.2^{2^{\cdot^{\cdot^{2}}}}\right\}O(n)$$

This is a *non-elementary* function, i.e. it grows faster than any of the iterated exponentials defined by $\exp_0(n) = n$ and $\exp_{k+1}(n) = 2^{exp_k(n)}$. The following result, which we quote without proof (see [318, 456]), states that the difficulty caused by the size explosion cannot be avoided.

3.7.1. THEOREM (Statman). *The problem of deciding whether any two given Church-style terms M and N of the same type are beta-equal is of non-elementary complexity. That is, for each r, every decision procedure takes more than $\exp_r(n)$ steps on some inputs of size n.*

The strong normalization result gives a hint that the expressive power of $\lambda_\to$ should be weaker than that of the untyped lambda-calculus, i.e. that one should not be able to represent all recursive functions by simply typed λ-terms. On the other hand, Theorem 3.7.1 might lead one to expect that the class of definable total functions should still be quite rich.

As we shall now see, the latter expectation is not quite correct, but first the notion of a definable function must be made precise.

3.7.2. DEFINITION. Let $\mathbf{int} = (p \to p) \to (p \to p)$, where p is an arbitrary type variable. A function $f : \mathbb{N}^k \to \mathbb{N}$ is *$\lambda_\to$-definable* if there is an $F \in \Lambda$ with $\vdash F : \mathbf{int}^k \to \mathbf{int}$, such that

$$F\mathbf{c}_{n_1} \ldots \mathbf{c}_{n_k} =_\beta \mathbf{c}_{f(n_1,\ldots,n_k)}$$

for all $n_1, \ldots, n_k \in \mathbb{N}$.

3.7.3. DEFINITION. The class of *extended polynomials* is the smallest class of functions over $\mathbb{N}$ which is closed under compositions and contains the constant functions 0 and 1, projections, addition, multiplication, and the conditional function

$$cond(n_1, n_2, n_3) = \begin{cases} n_2, & \text{if } n_1 = 0; \\ n_3, & \text{otherwise.} \end{cases}$$

3.7.4. THEOREM (Schwichtenberg). *The $\lambda_{\to}$-definable functions are exactly the extended polynomials.*

PROOF. Exercises 3.24–3.26. □

3.8. Notes

Types are often seen as a method to avoid paradoxes occurring in the type-free world as a result of various forms of self-application. Undoubtedly, paradoxes gave the impulse for the creation of various type theories at the beginning of the 20th century. But, as pointed out in [165, 257], it is natural in mathematics to classify or stratify objects into categories or "types", and that occurred well before the paradoxes were discovered.

The history of formal type theory begins with Russell. The work of Chwistek, Ramsey, Hilbert, and others contributed to the development of the subject. An influential presentation of the simple theory of types was given in Church's paper [73] of 1940. For this purpose Church introduced the simply typed lambda-calculus, the core language of his type theory.

A typed version of combinatory logic was proposed by Curry a few years earlier, in the the 1934 paper [100], although Curry must have had the idea already in 1928, see [439, 440]. Curry's full "theory of functionality" turned out later to be contradictory [104] but it was readily corrected [105]. Soon types became a standard concept in the theory of combinators and in lambda-calculus.

Later types turned out to be useful in programming languages. Just like the type-free λ-calculus provides a foundation of untyped programming languages, various typed λ-calculi provide a foundation of programming languages with types. In particular, the design of languages like ML [387] motivated the research on type checking and typability problems. But, as noted in [237, pp. 103–104], the main ideas of a type reconstruction algorithm can be traced as far back as the 1920's.[5] The unification-based principal type inference algorithm, implicit in Newman's 1943 paper [362], was first described by Hindley [234] in 1969 (see [233, 237, 238] for historical notes). The PTIME-completeness of typability in simple types was shown in 1988 by Tyszkiewicz. Hindley's algorithm was later reinvented by Milner [342] for the language ML (but for ML it is no longer polynomial [259, 263]).

If we look at the typability problem from the point of view of the Curry-Howard isomorphism, then we may restate it by asking whether a given "proof skeleton" can actually be turned into a correct proof by inserting the missing formulas. Interestingly, such questions (like the "skeleton instantiation" problem of [500]) are indeed motivated by proof-theoretic research.

[5] The good old Polish school again...

Our normalization proof follows Turing's unpublished note from the 1930's [166]. This method was later rediscovered by several other authors, including Prawitz [403]. The first published normalization proof can be found in [107] (see [166] for discussion). The translation from "weak" to strong normalization is based on Klop's [272], here modified using ideas of Gandy [167]. A similar technique was earlier used by Nederpelt [359]. Variations of this approach have been invented (some independently from the original) by numerous authors, see e.g. [184, 202, 261, 264, 265, 312, 508, 507]. Paper [448] gives a survey of some of these results, see also [199].

A widely used normalization proof method, the *computability method*, is due to Tait. We will see it for the first time in Chapter 5. Another approach (first used by Lévyand van Daalen in the late 1970's) is based on inductive characterizations of strongly normalizable terms [65, 120, 255]. Other proofs are based on different forms of explicit induction, or semantical completeness [87, 167, 398, 406, 501]. Complexity bounds for the length of reductions are given in [433].

Various extensions of the simply typed lambda calculus can be proved strongly normalizable and we will do a number of such proofs in the chapters to follow. See also Chapter 7 for a related subject: cut-elimination.

Newman's lemma dates from 1942. A special case of it was known to Thue as early as in 1910 [458]. The proof given here, a variant of Huet's proof in [249], is taken from [31]. Theorem 3.7.4 is from [431] and remains true if type **int** is replaced by $\mathbf{int}_\sigma = (\sigma \to \sigma) \to \sigma \to \sigma$, for any fixed σ, see [305]. If one does not insist that σ is fixed, more functions become $\lambda_\to$-definable [153]. For instance the predecessor function is definable by a term of type $\mathbf{int}_\sigma \to \mathbf{int}_\tau$, for suitable σ and τ. But various natural functions, like e.g. subtraction, remain undefinable even this way. To obtain a more interesting class of definable functions, one has to extend $\lambda_\to$ by some form of iteration or recursion, cf. Chapter 10.

The books [237, 241] and the survey [32] are recommended for further reading on simply typed lambda-calculus and combinatory logic. Papers [165, 257] give the historical perspective. For applications in programming languages we suggest [349, 393]. In our short exposition we have entirely omitted semantical issues and extensions of simply typed λ-calculus, such as PCF. See [8, 211, 285, 349] for more on these subjects.

3.9. Exercises

3.1. Show that the following λ-terms are not typable in $\lambda_\to$ à la Curry.

$$\mathbf{K}\mathbf{I}\Omega, \qquad \mathbf{Y}, \qquad \lambda xy.y(x\mathbf{K})(x\mathbf{I}), \qquad \mathbf{2K}.$$

3.2. Does the Curry style $\lambda_\to$ have the *subject conversion* property: *If* $\Gamma \vdash M : \sigma$ *and* $M =_\beta N$, *then* $\Gamma \vdash N : \sigma$? What if we assume in addition that N is typable?

3.3. Show an example of an untypable λI-term M, such that $M \to_\beta N$, for some typable term N.

3.4. Show an example of a typable closed λI-term M, and a type τ such that $\nvdash M : \tau$, but $M \to_\beta N$, for some term N with $\vdash N : \tau$.

3.5. Assume an environment Γ consisting of type assumptions of the form $(x_p : p)$. Define terms t_τ such that $\Gamma \vdash t_\tau : \sigma$ holds if and only if $\tau = \sigma$.

3.6. How long (in the worst case) is the shortest type of a Curry-style term of length n?

3.7. Show that the general unification problem can be reduced to the case of a signature consisting exclusively of the binary function symbol $\to$.

3.8. Show that the unification problem reduces to solving a single equation, provided the signature contains a binary function symbol.

3.9. Show that the unification problem reduces in logarithmic space to the typability problem.

3.10. Show that problems (vii) and (viii) of Remark 3.2.8 reduce to each other in logarithmic space.

3.11. Prove that the unification problem and problems (i)–(vi) of Remark 3.2.8 reduce to each other in logarithmic space.

3.12. What is wrong with the following reduction of problem (vi) to problem (i): *To answer* $? \vdash M : \tau$ *ask the question* $? \vdash \lambda yz.y(zM)(zt_\tau) : ?$

3.13. Prove the following *converse principal type theorem*: If τ is a non-empty type, then there exists a closed term M such that τ is the principal type of M. (In fact, if N is a closed term of type τ, then we can require M to be beta-reducible to N.) *Hint:* Use the technique of Exercise 3.5.

3.14. Let $\Gamma \vdash M[x := N] : \sigma$ in Church-style, and let $x \in \mathrm{FV}(M)$. Show that $\Gamma, x : \tau \vdash M : \sigma$, for some τ with $\Gamma \vdash N : \tau$. Does it hold for Curry style?

3.15. Show that strong normalization for $(\lambda_\to)$ à la Curry implies strong normalization for $(\lambda_\to)$ à la Church, and conversely.

3.16. Show that the weak Church-Rosser property does not in general imply the Church-Rosser property.

3.17. Let M_1 and M_2 be Church-style normal forms of the same type, and let $|M_1| = |M_2|$. Show that $M_1 = M_2$.

3.18. Let M_1 and M_2 be Church-style terms of the same type, and assume that $|M_1| =_\beta |M_2|$. Show that $M_1 =_\beta M_2$. Does $|M_1| = |M_2|$ imply $M_1 = M_2$?

3.19. Can you derive Church-Rosser property for Church-style terms from the Church-Rosser property for untyped terms?

3.20. How long is the normal form of the term $M = \mathbf{2}\mathbf{2}\cdots\mathbf{2}xy$ with n occurrences of $\mathbf{2}$? How long (including types) is a Church-style term M' such that $|M'| = M$?

3.21. (Based on [153].) This exercise uses the notation introduced in the proof of Theorem 3.5.1. In addition we define the *depth* $d(M)$ of M as:

$$\begin{array}{lcl} d(x) & = & 0; \\ d(MN) & = & 1 + \max(d(M), d(N)); \\ d(\lambda x{:}\sigma\, M) & = & 1 + d(M). \end{array}$$

Let $\delta = \delta(M)$ and let $d(M) = d$. Show that M reduces in at most 2^d steps to some term M_1 such that $\delta(M_1) < \delta$ and $d(M_1) \le 2^d$.

3.22. Use Exercise 3.21 to prove that if $\delta(M) = \delta$ and $d(M) = d$ then the normal form of a term M is of depth at most $\exp_\delta(d)$, and can be obtained in at most $\exp_{\delta+1}(d)$ reduction steps.

3.23. How long (in the worst case) is the normal form of a Curry-style typable term of length n?

3.24. Show that all the extended polynomials are $\lambda_\to$-definable.

3.25. Let $\Gamma = \{f : p \to p,\ a : \mathbf{int},\ b : \mathbf{int},\ x_1 : p,\ \ldots,\ x_r : p\}$, and let $\Gamma \vdash M : p$. Prove that there exists a polynomial $P(m, n)$ and a number $i \in \{1, \ldots, r\}$ such that $M[a := \mathbf{c}_m, b := \mathbf{c}_n] =_\beta f^{P(m,n)}x_i$ holds for all $m, n \neq 0$.

3.26. Prove that all functions definable in simply typed lambda-calculus are extended polynomials (Theorem 3.7.4). *Hint:* Use Exercise 3.25.