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 ($\to$E) could be denoted by something like $@(M, N) : \psi$, or perhaps simply written as $MN : \psi$. This gives an "annotated" rule ($\to$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 ($\to$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).

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

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

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

Figure 3.1: 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 $\sigma \in \Phi_{\to}$, 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 $M \in \Lambda_{\sigma\to\tau}$ and $N \in \Lambda_\sigma$ then $(MN) \in \Lambda_\tau$;
- If $M \in \Lambda_\tau$ and $x^\sigma \in \Upsilon_\sigma$ then $(\lambda x^\sigma M) \in \Lambda_{\sigma\to\tau}$.

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 $\lambda_\to$ 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.

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

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:

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

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:

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

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 $\lambda_{\to}$ 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

$$\boldsymbol{m}_M = (\delta_M, n_M),$$

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 $\boldsymbol{m}_M = (0, 0)$. The proof is by induction on lexicographically ordered pairs $\boldsymbol{m}_M$.

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 $\boldsymbol{m}_M > \boldsymbol{m}_{M'}$, 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 $a_0 \to a_1 \to a_2 \to \cdots$

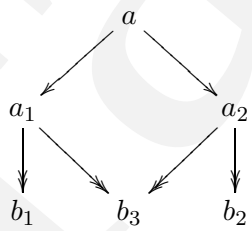
Clearly, CR implies WCR. The converse is not true, see Exercise 3.16. But the two properties coincide for strongly normalizing relations.

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.



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. □

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

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

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

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

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

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.

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

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 $\lambda\mathbf{I}$-term M, such that $M \to_\beta N$, for some typable term N.

3.4. Show an example of a typable closed $\lambda\mathbf{I}$-term M, and a type τ such that $\not\vdash M : \tau$, 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 $\rightarrow$.

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 $(\lambda_\rightarrow)$ à la Curry implies strong normalization for $(\lambda_\rightarrow)$ 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 $M = \mathbf{22}\cdots\mathbf{2}xy$ 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 $d(M_1) \leq 2^d$.

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 $\lambda_\rightarrow$-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.