Basic Proof Theory

Second Edition

A.S. Troelstra
University of Amsterdam

H. Schwichtenberg
University of Munich

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK http://www.cup.cam.ac.uk
40 West 20th Street, New York, NY 10011-4211, USA http://www.cup.org
10 Stamford Road, Oakleigh, Melbourne 3166, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain

© Cambridge University Press 1996, 2000

This book is in copyright. Subject to statutory exception
and to the provisions of relevant collective licensing agreements,
no reproduction of any part may take place without
the written permission of Cambridge University Press.

First published 1996
Second edition 2000

Typeset by the author in Computer Modern 10/13pt, in $\LaTeX\,2_\varepsilon$ [EPC]

A catalogue record of this book is available from the British Library

Library of Congress Cataloguing in Publication data

ISBN 0 521 77911 1 paperback

Transferred to digital printing 2003

Contents

Chapter 1

Introduction

Proof theory may be roughly divided into two parts: structural proof theory and interpretational proof theory. Structural proof theory is based on a combinatorial analysis of the structure of formal proofs; the central methods are cut elimination and normalization.

In interpretational proof theory, the tools are (often semantically motivated) syntactical translations of one formal theory into another. We shall encounter examples of such translations in this book, such as the Gödel–Gentzen embedding of classical logic into minimal logic (2.3), and the modal embedding of intuitionistic logic into the modal logic **S4** (9.2). Other well-known examples from the literature are the formalized version of Kleene's realizability for intuitionistic arithmetic and Gödel's Dialectica interpretation (see, for example, Troelstra [1973]).

The present text is concerned with the more basic parts of structural proof theory. In the first part of this text (chapters 2–7) we study several formalizations of standard logics. "Standard logics", in this text, means minimal, intuitionistic and classical first-order predicate logic. Chapter 8 describes the connection between cartesian closed categories and minimal conjunction–implication logic; this serves as an example of the applications of proof theory in category theory. Chapter 9 illustrates the extension to other logics (namely the modal logic **S4** and linear logic) of the techniques introduced before in the study of standard logics. The final two chapters deal with first-order arithmetic and second-order logic respectively.

The first section of this chapter contains notational conventions and definitions, to be consulted only when needed, so a quick scan of the contents will suffice to begin with. The second section presents a concise introduction to simple type theory, with rigid typing; the parallel between (extensions of) simple type theory and systems of natural deduction, under the catch-phrase "formulas-as-types", is an important theme in the sequel. Then follows a brief informal introduction to the three principal types of formalism we shall encounter later on, the N-, H- and G-systems, or Natural deduction, Hilbert systems, and Gentzen systems respectively. Formal definitions of these systems will be given in chapters 2 and 3.

1.1 Preliminaries

The material in this section consists primarily of definitions and notational conventions, and may be skipped until needed.

Some very general abbreviations are "iff" for "if and only if", "IH" for "induction hypothesis", "w.l.o.g." for "without loss of generality". To indicate *literal identity* of two expressions, we use $\equiv$. (In dealing with expressions with bound variables, this is taken to be literal identity modulo renaming of bound variables; see 1.1.2 below.)

The symbol $\boxtimes$ is used to mark the end of proofs, definitions, stipulations of notational conventions.

$\mathbb{N}$ is used for the natural numbers, zero included. Set-theoretic notations such as $\in$, $\subset$ are standard.

1.1.1. *The language of first-order predicate logic*

The standard language considered contains $\vee, \wedge, \rightarrow, \perp, \forall, \exists$ as primitive logical operators ($\perp$ being the degenerate case of a zero-place logical operator, i.e. a logical constant), countably infinite supplies of individual variables, n-place relation symbols for all $n \in \mathbb{N}$, symbols for n-ary functions for all $n \in \mathbb{N}$. 0-place relation symbols are also called proposition letters or proposition variables; 0-argument function symbols are also called (individual) constants. The language will *not*, unless stated otherwise, contain $=$ as a primitive.

Atomic formulas are formulas of the form $Rt_1 \dots t_n$, R a relation symbol, $t_1, \dots, t_n$ individiual terms, $\perp$ is not regarded as atomic. For formulas which are either atomic or $\perp$ we use the term *prime* formula

We use certain categories of letters, possibly with sub- or superscripts or primed, as metavariables for certain syntactical categories (locally different conventions may be introduced):

- x, y, z, u, v, w for individual variables;
- f, g, h for arbitrary function symbols;
- c, d for individual constants;
- t, s, r for arbitrary terms;
- P, Q for atomic formulas;
- R for relation symbols of the language;
- A, B, C, D, E, F for arbitrary formulas in the language.

NOTATION. For the countable set of *proposition variables* we write $\mathcal{PV}$. We introduce abbreviations:

$$\begin{aligned} A \leftrightarrow B &:= (A \to B) \land (B \to A), \\ \neg A &:= A \to \bot, \\ \top &:= \bot \to \bot. \end{aligned}$$

In this text, $\top$ ("truth") is sometimes added as a primitive. If Γ is a finite sequence $A_1, \ldots, A_n$ of formulas, $\bigwedge\Gamma$ is the iterated conjunction $(\ldots(A_1 \land A_2) \land \ldots A_n)$, and $\bigvee\Gamma$ the iterated disjunction $(\ldots(A_1 \lor A_2) \lor \ldots A_n)$. If Γ is empty, we identify $\bigvee\Gamma$ with $\bot$, and $\bigwedge\Gamma$ with $\top$. ⊠

NOTATION. (*Saving on parentheses*) In writing formulas we save on parentheses by assuming that $\forall, \exists, \neg$ bind more strongly than $\lor, \land$, and that in turn $\lor, \land$ bind more strongly than $\to, \leftrightarrow$. Outermost parentheses are also usually dropped. Thus $A \land \neg B \to C$ is read as $((A \land (\neg B)) \to C)$. In the case of iterated implications we sometimes use the short notation

$$A_1 \to A_2 \to \ldots A_{n-1} \to A_n \quad \text{for} \quad A_1 \to (A_2 \to \ldots (A_{n-1} \to A_n) \ldots).$$

We also save on parentheses by writing e.g. $Rxyz$, $Rt_0t_1t_2$ instead of $R(x, y, z)$, $R(t_0, t_1, t_2)$, where R is some predicate letter. Similarly for a unary function symbol with a (typographically) simple argument, so fx for $f(x)$, etc. In this case no confusion will arise. But readability requires that we write in full $R(fx, gy, hz)$, instead of $Rfxgyhz$. ⊠

1.1.2. *Substitution, free and bound variables*

Expressions $\mathcal{E}, \mathcal{E}'$ which differ only in the names of bound variables will be regarded by us as identical. This is sometimes expressed by saying that $\mathcal{E}$ and $\mathcal{E}'$ are α-equivalent. In other words, we are only interested in certain equivalence classes of (the concrete representations of) expressions, expressions "modulo renaming of bound variables". There are methods of finding unique representatives for such equivalence classes, for example the namefree terms of de Bruijn [1972]. See also Barendregt [1984, Appendix C].

For the human reader such representations are less convenient, so we shall stick to the use of bound variables. But it should be realized that the issues of handling bound variables, renaming procedures and substitution are essential and non-trivial when it comes to implementing algorithms.

In the definition of "substitution of expression $\mathcal{E}'$ for variable x in expression $\mathcal{E}$", either one requires that *no* variable free in $\mathcal{E}'$ becomes bound by a variable-binding operator in $\mathcal{E}$, when the free occurrences of x are replaced by $\mathcal{E}'$ (also expressed by saying that there must be no "clashes of variables"), "*$\mathcal{E}'$ is free for x in $\mathcal{E}$*", or the substitution operation is taken to involve a systematic renaming operation for the bound variables, avoiding clashes. Having stated that we are only interested in expressions modulo renaming bound variables, we can without loss of generality assume that substitution is always possible.

Also, it is never a real restriction to assume that distinct quantifier occurrences are followed by distinct variables, and that the sets of bound and free variables of a formula are disjoint.

NOTATION. "FV" is used for the (set of) free variables of an expression; so $\mathrm{FV}(t)$ is the set of variables free in the term t, $\mathrm{FV}(A)$ the set of variables free in formula A etc.

$\mathcal{E}[x/t]$ denotes the result of substituting the term t for the variable x in the expression $\mathcal{E}$. Similarly, $\mathcal{E}[\vec{x}/\vec{t}]$ is the result of *simultaneously* substituting the terms $\vec{t} = t_1, \ldots, t_n$ for the variables $\vec{x} = x_1, \ldots, x_n$ respectively.

For substitutions of predicates for predicate variables (predicate symbols) we use essentially the same notational conventions. If in a formula A, containing an n-ary relation variable X^n, X^n is to be replaced by a formula B, seen as an n-ary predicate of n of its variables $\vec{x} \equiv x_1, \ldots, x_n$, we write $A[X^n/\lambda\vec{x}.B]$ for the formula which is obtained from A by replacing every occurrence $X^n\vec{t}$ by $B[\vec{x}/\vec{t}]$ (neither individual variables nor relation variables of $\forall\vec{x}B$ are allowed to become bound when substituting).

Note that B may contain other free variables besides $\vec{x}$, and that the "$\lambda\vec{x}$" is needed to indicate which terms are substituted for which variables.

Locally we shall adopt the following convention. In an argument, once a formula has been introduced as $A(x)$, i.e., A with a designated free variable x, we write $A(t)$ for $A[x/t]$, and similarly with more variables. ⊠

1.1.3. *Subformulas*

DEFINITION. (*Gentzen subformula*) Unless stated otherwise, the notion of *subformula* we use will be that of a subformula in the sense of Gentzen. (Gentzen) subformulas of A are defined by

(i) A is a subformula of A;

(ii) if $B \circ C$ is a subformula of A then so are B, C, for $\circ = \vee, \wedge, \rightarrow$;

(iii) if $\forall xB$ or $\exists xB$ is a subformula of A, then so is $B[x/t]$, for all t free for x in B.

If we replace the third clause by:

(iii)′ if $\forall xB$ or $\exists xB$ is a subformula of A then so is B,

we obtain the notion of *literal* subformula. ⊠

DEFINITION. The notions of *positive, negative, strictly positive* subformula are defined in a similar style:

(i) A is a positive and a stricly positive subformula of itself;

(ii) if $B \wedge C$ or $B \vee C$ is a positive [negative, strictly positive] subformula of A, then so are B, C;

(iii) if $\forall x B$ or $\exists x B$ is a positive [negative, strictly positive] subformula of A, then so is $B[x/t]$ for any t free for x in B;

(iv) if $B \to C$ is a positive [negative] subformula of A, then B is a negative [positive] subformula of A, and C is a positive [negative] subformula of A;

(v) if $B \to C$ is a strictly positive subformula of A then so is C.

A strictly positive subformula of A is also called a *strictly positive part (s.p.p.)* of A. Note that the set of subformulas of A is the union of the positive and the negative subformulas of A.

Literal positive, negative, strictly positive subformulas may be defined in the obvious way by restricting the clause for quantifiers. ⊠

EXAMPLE. $(P \to Q) \to R \vee \forall x R'(x)$ has as s.p.p.'s the whole formula, $R \vee \forall x R'(x)$, R, $\forall x R'(x)$, $R'(t)$. The positive subformulas are the s.p.p.'s and in addition P; the negative subformulas are $P \to Q$, Q.

1.1.4. *Contexts and formula occurrences*

Formula occurrences (f.o.'s) will play an even more important role than the formulas themselves. An f.o. is nothing but a formula with a position in another structure (prooftree, sequent, a larger formula etc.). If no confusion is to be feared, we shall permit ourselves a certain "abus de langage" and talk about formulas when really f.o.'s are meant.

The notion of a (sub)formula occurrence in a formula or sequent is intuitively obvious, but for formal proofs of metamathematical properties it is sometimes necessary to use a rigorous formal definition. This may be given via the notion of a *context.* Roughly speaking, a context is nothing but a formula with an occurrence of a special propositional variable, a "placeholder". Alternatively, a context is sometimes described as a formula with a hole in it.

DEFINITION. We define *positive* ($\mathcal{P}$) and *negative (formula-)contexts* ($\mathcal{N}$) simultaneously by an inductive definition given by the three clauses (i)–(iii) below. The symbol "$*$" in clause (i) functions as a special proposition letter (not in the language of predicate logic), a *placeholder* so to speak.

(i) $* \in \mathcal{P}$;

and if $B^+ \in \mathcal{P}$, $B^- \in \mathcal{N}$, and A is any formula, then

(ii) $A{\wedge}B^+$, $B^+{\wedge}A$, $A{\vee}B^+$, $B^+{\vee}A$, $A{\to}B^+$, $B^-{\to}A$, $\forall xB^+$, $\exists xB^+ \in \mathcal{P}$;

(iii) $A{\wedge}B^-$, $B^-{\wedge}A$, $A{\vee}B^-$, $B^-{\vee}A$, $A{\to}B^-$, $B^+{\to}A$, $\forall xB^-$, $\exists xB^- \in \mathcal{N}$.

The set of *formula contexts* is the union of $\mathcal{P}$ and $\mathcal{N}$. Note that a context contains always only a single occurrence of $*$. We may think of a context as a formula in the language extended by $*$, in which $*$ occurs only once. In a positive [negative] context, $*$ is a positive [negative] subformula. Below we give a formal definition of (sub)formula occurrence *via* the notion of context.

For arbitrary contexts we sometimes write $F[*]$, $G[*]$, ... Then $F[A]$, $G[A]$, ... are the formulas obtained by replacing $*$ by A (literally, without renaming variables).

The notion of context may be generalized to a context with several placeholders $*_1, \dots, *_n$, which are treated as extra proposition variables, each of which may occur only once in the context.

The *strictly positive* contexts $\mathcal{SP}$ are defined by

(iv) $* \in \mathcal{SP}$; and if $B \in \mathcal{SP}$, then

(v) $A \wedge B$, $B \wedge A$, $A \vee B$, $B \vee A$, $A \to B$, $\forall xB$, $\exists xB \in \mathcal{SP}$.

An alternative style of presentation of this definition is

$$\begin{aligned}
&\mathcal{P} = * \mid A \wedge \mathcal{P} \mid \mathcal{P} \wedge A \mid A \vee \mathcal{P} \mid \mathcal{P} \vee A \mid A \to \mathcal{P} \mid \mathcal{N} \to A \mid \forall x\mathcal{P} \mid \exists x\mathcal{P},\\
&\mathcal{N} = A \wedge \mathcal{N} \mid \mathcal{N} \wedge A \mid A \vee \mathcal{N} \mid \mathcal{N} \vee A \mid A \to \mathcal{N} \mid \mathcal{P} \to A \mid \forall x\mathcal{N} \mid \exists x\mathcal{N},\\
&\mathcal{SP} = * \mid A \wedge \mathcal{SP} \mid \mathcal{SP} \wedge A \mid A \vee \mathcal{SP} \mid \mathcal{SP} \vee A \mid A \to \mathcal{SP} \mid \forall x\mathcal{SP} \mid \exists x\mathcal{SP}.
\end{aligned}$$

A *formula occurrence* (*f.o.* for short) in a formula B is a literal subformula A together with a context indicating the place where A occurs (so B may be obtained by replacing $*$ in the context by A). In the obvious way we can now define *positive, strictly positive* and *negative occurrence.* ⊠

1.1.5. *Finite multisets*

Finite *multisets*, i.e. "sets with multiplicity", or to put it otherwise, finite sequences modulo the ordering, will play an important role in this text.

NOTATION. If Δ is a multiset, we use $|\Delta|$ for the number of its elements. For the multiset union of Γ and Δ we write $\Gamma \cup \Delta$ or in certain situations simply Γ, Δ or even $\Gamma\Delta$ (namely when writing sequents, which will be introduced later). The notation Γ, A or ΓA then designates a multiset which is the union of Γ and the singleton multiset containing only A.

If "c" is some unary operator and $\Gamma \equiv A_1, \dots, A_n$ is a finite multiset of formulas, we write $c\Gamma$ for the multiset $cA_1, \dots, cA_n$.

Finite sets may be regarded as special cases of finite multisets: a multiset where each element occurs with multiplicity one represents a finite set. For the set underlying a multiset Γ, we write $\mathrm{Set}(\Gamma)$; this multiset contains the formulas of Γ with multiplicity one. ⊠

NOTATION. We shall use the notations $\bigwedge \Gamma$, $\bigvee \Gamma$ also in case Γ is a multiset. $\bigwedge \Gamma$, $\bigvee \Gamma$ are then the conjunction, respectively disjunction of Γ' for some sequence Γ' corresponding to Γ. $\bigwedge \Gamma$, $\bigvee \Gamma$ are then well-defined modulo logical equivalence, as long as in our logic $\wedge, \vee$ obey the laws of symmetry and associativity. ⊠

DEFINITION. The notions of (*positive, negative*) *formula occurrence* may be defined for sequents, i.e., expressions of the form $\Gamma \Rightarrow \Delta$, with Γ, Δ finite multisets, as (positive, negative) formula occurrences in the corresponding formulas $\bigwedge \Gamma \to \bigvee \Delta$. ⊠

1.1.6. *Deducibility and deduction from hypotheses, conservativity*

NOTATION. In our formalisms, we derive either formulas or sequents (as introduced in the preceding definition). For sequents derived in a formalism **S** we write

$$\mathbf{S} \vdash \Gamma \Rightarrow \Delta \quad \text{or} \quad \vdash_{\mathbf{S}} \Gamma \Rightarrow \Delta,$$

and for formulas derived in **S**

$$\mathbf{S} \vdash A \quad \text{or} \quad \vdash_{\mathbf{S}} A.$$

If we want to indicate that a deduction $\mathcal{D}$ derives $\Gamma \Rightarrow \Delta$, we can write $\mathcal{D} \vdash_{\mathbf{S}} \Gamma \Rightarrow \Delta$ (or $\mathcal{D} \vdash \Gamma \Rightarrow \Delta$ if **S** is evident).

For formalisms based on sequents, $\mathbf{S} \vdash A$ will coincide with $\mathbf{S} \vdash \Rightarrow A$ (sequent $\Gamma \Rightarrow A$ with Γ empty).

If a formula A is derivable from a finite multiset Γ of hypotheses or assumptions, we write

$$\Gamma \vdash_{\mathbf{S}} A.$$

In systems with sequents this is equivalent to $\mathbf{S} \vdash \Gamma \Rightarrow A$. (N.B. In the literature $\Gamma \vdash A$ is sometimes given a slightly different definition for which the deduction theorem does not hold; cf. remark in 9.1.2. Moreover, some authors use $\vdash$ instead of our sequent-arrow $\Rightarrow$.)

A *theory* is a set of sentences (closed formulas); with each formalism is associated a theory of deducible sentences. Since for the theories associated with the formalisms in this book, it is always true that the set of deducible *formulas* and the set of pairs $\{(\Gamma, A) \mid \Gamma \vdash A\}$ are uniquely determined by the theory, we shall also speak of formulas belonging to a theory, and use the expression "A is deducible from Γ in a theory".

In particular, we write

$$\Gamma \vdash_{\mathrm{m}} A, \qquad \Gamma \vdash_{\mathrm{i}} A, \qquad \Gamma \vdash_{\mathrm{c}} A$$

for deducibility in our standard logical theories **M**, **I**, **C** respectively (cf. the next subsection). ⊠

DEFINITION. A system **S** is *conservative* over a system $\mathbf{S}' \subset \mathbf{S}$, if for formulas A in the language of $\mathbf{S}'$ we have that if $\mathbf{S} \vdash A$, then $\mathbf{S}' \vdash A$. For systems with sequents, conservativity similarly means: if $\vdash \Gamma \Rightarrow A$ in **S**, with $\Gamma \Rightarrow A$ in the language of $\mathbf{S}'$, then $\vdash \Gamma \Rightarrow A$ in $\mathbf{S}'$. Similarly for theories. ⊠

1.1.7. *Names for theories and systems*

Where we are only interested in the logics as theories, i.e. as *sets of theorems*, we use **M**, **I** and **C** for minimal, intuitionistic and classical predicate calculus respectively; **Mp**, **Ip** and **Cp** are the corresponding propositional systems. If we are interested only in formulas constructed from a set of operators $\mathcal{A}$ say, we write $\mathcal{A}$-S or $\mathcal{A}$**S** for the system **S** restricted to formulas with operators from $\mathcal{A}$. Thus $\to\wedge$-**M** is **M** restricted to formulas in $\wedge, \to$ only.

On the other hand, where the notion of formal deduction is under investigation, we have to distinguish between the various formalisms characterizing the same theory. In choosing designations, we use some mnemonic conventions:

- We use "N", "**H**", "G" for "Natural Deduction", "Hilbert system" and "Gentzen system" respectively. "**GS**" (from "Gentzen–Schütte") is used as a designation for a group of calculi with one-sided sequents (always classical).
- We use "**c**" for "classical", "**i**" for "intuitionistic", "**m**" for "minimal", "s" for "S4", "**p**" for "propositional", "**e**" for "E-logic". If **p** is absent, the system includes quantifiers. The superscript "^{2}" is used for second-order systems.
- Variants may be designated by additions of extra boldface capitals, numbers, superscripts such as "*" etc. Thus, for example, **G1c** is close to the original sequent calculus LK of Gentzen (and **G1i** to Gentzen's LJ), **G2c** is a variant with weakening absorbed into the logical rules, **G3c** a system with weakening *and* contraction absorbed into the rules, **GK** (from Gentzen–Kleene) refers to Gentzen systems very close to the system G3 of Kleene, etc.

- In order to indicate several formal systems at once, without writing down the exhaustive list, we use the following type of abbreviation: S[**abc**] refers to **Sa**, **Sb**, **Sc**; S[**ab**][**cd**] refers to **Sac**, **Sbc**, **Sad**, **Sbd**, etc.; [**mic**] stands for "**m**, or **i** or **c**"; [**mi**] for "**m** or **i**"; [**123**] for "**1**, **2** or **3**", etc. In such contracted statements an obvious parallelism is maintained, e.g. "**G**[**123**]**c** satisfies $\mathcal{A}$ iff **G**[**123**]**i** satisfies $\mathcal{B}$" is read as: "**G1c** (respectively **G2c**, **G3c**) satisfies $\mathcal{A}$ iff **G1i** (respectively **G2i**, **G3i**) satisfies $\mathcal{B}$".

1.1.8. *Finite trees*

DEFINITION. (*Terminology for trees*) Trees are partially ordered sets $(X, \leq)$ with a lowest element and all sets $\{y : y \leq x\}$ for $x \in X$ linearly ordered. The elements of X are called the *nodes* of the tree; *branches* are maximal linearly ordered subsets of X (i.e. subsets which cannot be extended further).

Trees are supposed to grow upwards; the single node at the bottom is called the *root* or *bottom node* of the tree. If a *branch* of a tree is finite, it ends in a *leaf* or *top node* of the tree. If n, m are nodes of a tree with partial ordering $\prec$, and $n \prec m$, then m is a *successor* of n, n a *predecessor* of m. If $n \prec m$ and there are no nodes properly between n and m, then n is an *immediate* predecessor of m, and m an *immediate* successor of n.

A tree is said to be *k-branching* (*strictly k-branching*), if each node has at most k (exactly k) immediate successors.

We also consider labelled trees, with a function assigning objects (e.g. formulas) to the nodes. The terminology for trees is also applied to labelled trees. ⊠

1.1.9. DEFINITION. The *length* or *size* of a finite tree is the number of nodes in the tree. We write $\mathrm{s}(\mathcal{T})$ for the size of $\mathcal{T}$.

The *depth (of a tree)* or *height (of a tree)* $|\mathcal{T}|$ of a tree $\mathcal{T}$ is the maximum length of the branches in the tree, where the *length* of a branch is the number of nodes in the branch minus 1.

The *leafsize* $\mathrm{ls}(\mathcal{T})$ of a tree $\mathcal{T}$ is the number of top nodes of the tree. ⊠

For future use we note: Let $\mathcal{T}$ be a tree which is at most k-branching, i.e. each node has at most k ($k \geq 1$) immediate successors. Then

$$\mathrm{s}(\mathcal{T}) \leq k^{|\mathcal{T}|+1}, \quad \mathrm{ls}(\mathcal{T}) \leq \mathrm{s}(\mathcal{T}).$$

For strictly 2-branching trees $\mathrm{s}(\mathcal{T}) = 2\mathrm{ls}(\mathcal{T}) - 1$.

Formulas may also be regarded as (labelled) trees. The definitions of size and depth specialized to formulas yield the following definition.

DEFINITION. The *depth* $|A|$ of a formula A is the maximum length of a branch in its construction tree. In other words, we define recursively $|P| = 0$ for atomic P, $|\bot| = 0$, $|A \circ B| = \max(|A|, |B|) + 1$ for binary operators $\circ$, $|\circ A| = |A| + 1$ for unary operators $\circ$.

The *size* or *length* $\mathrm{s}(A)$ of a formula A is the number of occurrences of logical symbols and atomic formulas (parentheses not counted) in A: $\mathrm{s}(P) = 1$ for P atomic, $\mathrm{s}(\bot) = 0$, $\mathrm{s}(A \circ B) = \mathrm{s}(A) + \mathrm{s}(B) + 1$ for binary operators $\circ$, $\mathrm{s}(\circ A) = \mathrm{s}(A) + 1$ for unary operators $\circ$. ⊠

For formulas we therefore have

$$\mathrm{s}(A) \leq 2^{|A|+1}.$$

1.2 Simple type theories

This section briefly describes typed combinatory logic and typed lambda calculus, and may be skipped by readers already familiar with simple type theories. For more detailed information on type theories, see Barendregt [1992], Hindley [1997]. Below, we consider only formalisms with *rigid typing*, i.e. systems where every term and all subterms of a term carry a fixed type. Hindley [1997] deals with systems of *type assignment*, where untyped terms are assigned types according to certain rules. The untyped terms may possess many different types, or no type at all. There are many parallels between rigidly typed systems and type-assignment systems, but in the theory of type assignment there is a host of new questions, sometimes very subtle, to study. But theories of type assignment fall outside the scope of this book.

1.2.1. DEFINITION. (*The set of simple types*) The set of *simple types* $\mathcal{T}_{\to}$ is constructed from a countable set of *type variables* $P_0, P_1, P_2, \ldots$ by means of a type-forming operation (*function-type constructor*) $\to$. In other words, simple types are generated by two clauses:

(i) type variables belong to $\mathcal{T}_{\to}$;

(ii) if $A, B \in \mathcal{T}_{\to}$, then $(A \to B) \in \mathcal{T}_{\to}$.

A type of the form $A \to B$ is called a *function type*. "Generated" means that nothing belongs to $\mathcal{T}_{\to}$ except on the basis of (i) and (ii). Since the types have the form of propositional formulas, we can use the same abbreviations in writing types as in writing formulas (cf. 1.1.1). ⊠

Intuitively, types denote special sets. We may think of the type variables as standing for arbitrary, unspecified sets, and given types A, B, the type $A \to B$ is a set of functions from A to B.

1.2.2. DEFINITION. (*Terms of the simply typed lambda calculus* $\lambda_{\to}$) All terms appear with a type; for terms of type A we use t^A, s^A, r^A, possibly with extra sub- or superscripts. The terms are generated by the following three clauses:

(i) For each $A \in \mathcal{T}_{\to}$ there is a countably infinite supply of variables of type A; for arbitrary variables of type A we use $u^A, v^A, w^A, x^A, y^A, z^A$ (possibly with extra sub- or superscripts);

(ii) if $t^{A\to B}$, s^A are terms of types $A \to B$, A, then $\mathrm{App}(t^{A\to B}, s^A)^B$ is a term of type B;

(iii) if t^B is a term of type B and x^A a variable of type A, then $(\lambda x^A.t^B)^{A\to B}$ is a term of type $A \to B$. ⊠

NOTATION. For $\mathrm{App}(t^{A\to B}, s^A)^B$ we usually write simply $(t^{A\to B}s^A)^B$.

There is a good deal of redundancy in the typing of terms; provided the types of x, t, s are known, the types of $(\lambda x.t)$, (ts) are known and need not be indicated by a superscript. In general, we shall omit type-indications whenever possible without creating confusion. When writing ts it is always assumed that this is a meaningful application, and hence that for suitable A, B the term t has type $A \to B$, s type A.

If the type of the whole term is omitted, we usually simplify (ts) by dropping the outer parentheses and writing simply ts. The abbreviation $t_1t_2\dots t_n$ is defined by recursion on n as $(t_1t_2\dots t_{n-1})t_n$, i.e. $t_1t_2\dots t_n$ is $(\dots((t_1t_2)t_3\dots)t_n)$.

For $\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.t)\dots))$ we write $\lambda x_1x_2\dots x_n.t$. Application binds more strongly than $\lambda x.$, so $\lambda x.tt'$ is $\lambda x.(tt')$, not $(\lambda x.t)t'$.

A frequently used alternative notation for x^A, t^B is $x{:}\,A, t{:}\,B$ respectively. The notations t^A and $t{:}\,A$ are used interchangeably and may occur mixed; readability determines the choice. ⊠

EXAMPLES. $\mathrm{k}_\lambda^{A,B} := \lambda x^A y^B.x^A, \quad \mathrm{s}_\lambda^{A,B,C} := \lambda x^{A\to(B\to C)}y^{A\to B}z^A.xz(yz)$.

1.2.3. DEFINITION. The set $\mathrm{FV}(t)$ of variables free in t is specified by:

$$\begin{aligned}
\mathrm{FV}(x^A) &:= x^A,\\
\mathrm{FV}(ts) &:= FV(t) \cup \mathrm{FV}(s),\\
\mathrm{FV}(\lambda x.t) &:= \mathrm{FV}(t) \setminus \{x\}.
\end{aligned}$$ ⊠

1.2.4. DEFINITION. (*Substitution*) The operation of substitution of a term s for a variable x in a term t (notation $t[x/s]$) may be defined by recursion on the complexity of t, as follows.

$$\begin{array}{ll} x[x/s] & := s, \\ y[x/s] & := y \text{ for } y \not\equiv x, \\ (t_1t_2)[x/s] & := t_1[x/s]t_2[x/s], \\ (\lambda x.t)[x/s] & := \lambda x.t, \\ (\lambda y.t)[x/s] & := \lambda y.t[x/s] \text{ for } y \not\equiv x; \text{ w.l.o.g. } y \notin \mathrm{FV}(s). \end{array}$$

A similar definition may be given for simultaneous substitution $t[\vec{x}/\vec{s}]$. ⊠

LEMMA. *(Substitution lemma) If $x \not\equiv y$, $x \notin \mathrm{FV}(t_2)$, then*

$$t[x/t_1][y/t_2] \equiv t[y/t_2][x/t_1[y/t_2]].$$

PROOF. By induction on the depth of t. ⊠

1.2.5. DEFINITION. (*Conversion, reduction, normal form*) Let T be a set of terms, and let conv be a binary relation on T, written in infix notation: t conv s. If $t\,\mathrm{conv}\,s$, we say that t *converts to* s; t is called a *redex* or *convertible* term, and s the *conversum* of t. The replacement of a redex by its conversum is called a *conversion*. We write $t \succ_1 s$ (t *reduces in one step to* s) if s is obtained from t by replacement of (an occurrence of) a redex t' of t by a conversum t'' of t', i.e. by a single conversion. The relation $\succ$ ("*properly reduces to*") is the transitive closure of $\succ_1$ and $\succeq$ ("*reduces to*") is the reflexive and transitive closure of $\succ_1$. The relation $\succeq$ is said to be the notion of reduction *generated* by cont. $\prec_1, \prec, \preceq$ are the relations converse to $\succ_1, \succ, \succeq$ respectively.

With the notion of reduction generated by conv we associate a relation on T called *conversion equality*: $t =_{\mathrm{conv}} s$ (t *is equal by conversion to* s) if there is a sequence $t_0, \ldots, t_n$ with $t_0 \equiv t$, $t_n \equiv s$, and $t_i \preceq t_{i+1}$ or $t_i \succeq t_{i+1}$ for each i, $0 \leq i < n$. The subscript "conv" is usually omitted when clear from the context.

A term t is *in normal form*, or t is *normal*, if t does not contain a redex. t *has a normal form* if there is a normal s such that $t \succeq s$.

A *reduction sequence* is a (finite or infinite) sequence of pairs (t_0, δ_0), (t_1, δ_1), (t_2, δ_2), ... with δ_i an (occurrence of a) redex in t_i and $t_i \succ t_{i+1}$ by conversion of δ_i, for all i. This may be written as

$$t_0 \overset{\delta_0}{\succ_1} t_1 \overset{\delta_1}{\succ_1} t_2 \overset{\delta_2}{\succ_1} \ldots .$$

We often omit the δ_i, simply writing $t_0 \succ_1 t_1 \succ_1 t_2 \ldots$.

Finite reduction sequences are partially ordered under the initial part relation ("sequence σ is an initial part of sequence τ"); the collection of finite reduction sequences starting from a term t forms a tree, the *reduction tree* of t. The branches of this tree may be identified with the collection of all infinite and all terminating finite reduction sequences.

A term is *strongly normalizing* (is SN) if its reduction tree is finite. ⊠

REMARKS. (i) As to the terminology, in the literature on lambda calculus and combinatory logic, writers use mostly "contraction", "contracts", "contractum", instead of "conversion", "converts", "conversum". In the lambda calculus literature "conversion" is used for a more general notion: there t converts to s if t and s can be shown to be equal by reduction steps (going in both directions). On the other hand, there is a tradition, deriving from Prawitz [1965], of using "conversion" instead of "contraction" for the corresponding notion applied to natural deductions.

Moreover, "contraction" is also widely used in the literature on Gentzen systems (to be discussed later) for a specific deduction rule, whereas the notion of "conversion" of the lambda calculus literature is hardly used here. Therefore after prolonged hesitation we have chosen the terminology adopted here.

(ii) Usually it is more convenient to think of the reduction tree of a term t as a tree with its nodes labelled with terms; t is put at the root, and if s is the label of the node ν, there is, for each pair (s', δ) such that $s \overset{\delta}{\succ}_1 s'$, an immediate successor ν' to ν, with label s'.

Instead of the notion defined above, we may also consider a less refined notion of reduction sequence by disregarding the redexes; that is to say, we identify sequences

$$t_0 \overset{\delta_0}{\succ}_1 t_1 \overset{\delta_1}{\succ}_1 t_2 \overset{\delta_2}{\succ}_1 \dots \quad \text{and} \quad t'_0 \overset{\epsilon_0}{\succ}_1 t'_1 \overset{\epsilon_1}{\succ}_1 t'_2 \overset{\epsilon_2}{\succ}_1 \dots$$

if $t_i = t'_i$ for all i. The notion of reduction tree is then changed accordingly. The arguments in this book using reduction sequences hold with both notions of reduction sequence.

NOTATION. We shall distinguish different conversion relations by subscripts; so we have, for example, cont_β, $\mathrm{cont}_{\beta\eta}$ (to be defined below). Similarly for the associated relations of one-step reduction: $\succ_{\beta,1}$, $\succ_\beta$, $\succeq_\beta$, etc. We write $=_\beta$ instead of $=_{\mathrm{cont}_\beta}$ etc. ⊠

1.2.6. EXAMPLES. For us, the most important reduction is the one induced by β-conversion:

$$(\lambda x^A.t^B)s^A \ \ \mathrm{cont}_\beta \ \ t^B[x^A/s^A].$$

η-conversion is given by

$$\lambda x^A.tx \ \ \mathrm{cont}_\eta \ \ t \qquad (x \notin \mathrm{FV}(t)).$$

$\beta\eta$-conversion $\mathrm{cont}_{\beta\eta}$ is $\mathrm{cont}_\beta \cup \mathrm{cont}_\eta$.

It is to be noted that in defining $\succ_{\beta,1}, \succ_{\beta\eta,1}$ conversion of redexes occurring within the scope of a λ-abstraction operator is permitted. However, no free variables may become bound when executing the substitution in a β-conversion. An example of a reduction sequence is the following:

$$\begin{array}{ll}
(\lambda xyz.xz(yz))(\lambda uv.u)(\lambda u'v'.u') & \succ_{\beta,1} \\
(\lambda yz.(\lambda uv.u)z(yz))(\lambda u'v'.u') & \succ_{\beta,1} \\
(\lambda yz.(\lambda v.z)(yz))(\lambda u'v'.u') & \succ_{\beta,1} \\
(\lambda yz.z)(\lambda u'v'.u') & \succ_{\beta,1} \\
\lambda z.z. &
\end{array}$$

Relative to $\mathrm{cont}_{\beta\eta}$ conversion of different redexes may yield the same result: $(\lambda x.yx)z \succ_1 yz$ either by converting the β-redex $(\lambda x.yx)z$ or by converting the η-redex $\lambda x.yx$. So here the crude and the more refined notion of reduction sequence, mentioned above, differ.

DEFINITION. A relation R is said to be *confluent*, or to have the *Church–Rosser property (CR)*, if, whenever $t_0 R t_1$ and $t_0 R t_2$, then there is a t_3 such that $t_1 R\ t_3$ and $t_2 R t_3$. A relation R is said to be *weakly confluent*, or to have the *weak Church–Rosser property (WCR)*, if, whenever $t_0 R t_1, t_0 R t_2$ then there is a t_3 such that $t_1 R^* t_3$, $t_2 R^* t_3$, where R^* is the reflexive and transitive closure of R. ⊠

1.2.7. THEOREM. *For a confluent reduction relation $\succeq$ the normal forms of terms are unique. Furthermore, if $\succeq$ is a confluent reduction relation we have: $t = t'$ iff there is a term t'' such that $t \succeq t''$ and $t' \succeq t''$.*

PROOF. The first claim is obvious. The second claim is proved as follows. If $t = t'$ (for the equality induced by $\succeq$), then by definition there is a chain $t \equiv t_0, t_1, \ldots, t_n \equiv t'$, such that for all $i < n$ $t_i \succeq t_{i+1}$ or $t_{i+1} \succeq t_i$. The existence of the required t'' is now established by induction on n. Consider the step from n to $n+1$. By induction hypothesis there is an s such that $t_0 \succeq s$, $t_n \succeq s$. If $t_{n+1} \succeq t_n$, take $t'' = s$; if $t_n \succeq t_{n+1}$, use the confluence to find a t'' such that $s \succeq t''$ and $t_{n+1} \succeq t''$. ⊠

1.2.8. THEOREM. *(Newman's lemma) Let $\succeq$ be the transitive and reflexive closure of $\succ_1$, and let $\succ_1$ be weakly confluent. Then the normal form w.r.t. $\succ_1$ of a strongly normalizing t is unique. Moreover, if all terms are strongly normalizing w.r.t. $\succ_1$, then the relation $\succeq$ is confluent.*

PROOF. Assume WCR, and let us write $s \in \mathrm{UN}$ to indicate that s has a unique normal form. If a term is strongly normalizing, then so are all terms occurring in its reduction tree. In order to show that a strongly normalizing t has a unique normal form (and hence satisfies CR), we argue by contradiction.

We show that if $t \in \mathrm{SN}$, $t \notin \mathrm{UN}$, then we can find a $t_1 \prec t$ with $t_1 \notin \mathrm{UN}$. Repeating this construction leads to an infinite sequence $t \succ t_1 \succ t_2 \succ \ldots$ contradicting the strong normalizability of t.

So let $t \in \mathrm{SN}$, $t \notin \mathrm{UN}$. Then there are two reduction sequences $t \succ_1 t'_1 \succ t'_2 \succ \ldots \succ_1 t'$ and $t \succ_1 t''_1 \succ_1 t''_2 \succ \ldots \succ_1 t''$ with t', t'' distinct normal terms. Then either $t'_1 = t''_1$, or $t'_1 \neq t''_1$. In the first case we can take $t_1 := t'_1 = t''_1$. In the second case, by WCR we can find a t^* such that $t^* \prec t'_1, t''_1$; $t \in \mathrm{SN}$, hence $t^* \succ t'''$ for some normal t'''. Since $t' \neq t'''$ or $t'' \neq t'''$, either $t'_1 \notin \mathrm{UN}$ or $t''_1 \notin \mathrm{UN}$; so take $t_1 := t'_1$ if $t' \neq t'''$, $t_1 := t''_1$ otherwise. The final statement of the theorem follows immediately. ⊠

1.2.9. Definition. The *simple typed lambda calculus* $\boldsymbol{\lambda}_{\rightarrow}$ is the calculus of β-reduction and β-equality on the set of terms of $\boldsymbol{\lambda}_{\rightarrow}$ defined in 1.2.2. More explicitly, $\boldsymbol{\lambda}_{\rightarrow}$ has the term system as described, with the following axioms and rules for $\preceq$ (is $\preceq_\beta$) and $=$ (is $=_\beta$):

$$t \succeq t \qquad (\lambda x^A.t^B)s^A \succeq t^B[x^A/s^A]$$

$$\frac{t \succeq s}{rt \succeq rs} \qquad \frac{t \succeq s}{tr \succeq sr} \qquad \frac{t \succeq s}{\lambda x.t \succeq \lambda x.s} \qquad \frac{t \succeq s \quad s \succeq r}{t \succeq r}$$

$$\frac{t \succeq s}{t = s} \qquad \frac{t = s}{s = t} \qquad \frac{t = s \quad s = r}{t = r}$$

The *extensional simple typed lambda calculus* $\boldsymbol{\lambda\eta}_{\rightarrow}$ is the calculus of $\beta\eta$-reduction and $\beta\eta$-equality $=_{\beta\eta}$ and the set of terms $\boldsymbol{\lambda}_{\rightarrow}$; in addition to the axioms and rules already stated for the calculus $\boldsymbol{\lambda}_{\rightarrow}$ there is the axiom

$$\lambda x.tx \succeq t \quad (x \notin \mathrm{FV}(t)).$$ ⊠

1.2.10. Lemma. *(Substitutivity of $\succeq_\beta$ and $\succeq_{\beta\eta}$) For $\succeq$ either $\succeq_\beta$ or $\succeq_{\beta\eta}$ we have*

$$\textit{if } s \succeq s' \textit{ then } s[y/s''] \succeq s'[y/s''].$$

Proof. By induction on the depth of a proof of $s \succeq s'$. It suffices to check the crucial basis step, where s is $(\lambda x.t)t'$, and s' is $t[x/t']$: $(\lambda x.t)t'[y/s''] = (\lambda x.(t[y/s''])t'[y/s''] = t[y/s''][x/t'[y/s'']] = t[x/t'][y/s'']$ using (1.2.4). Here it is assumed that $x \not\equiv y$, $x \notin \mathrm{FV}(s'')$ (if not, rename x). ⊠

1.2.11. Proposition. *$\succ_{\beta,1}$ and $\succ_{\beta\eta,1}$ are weakly confluent.*

Proof. By distinguishing cases. If the conversions leading from t to t' and from t to t'' concern disjoint redexes, then t''' is simply obtained by converting both redexes. More interesting are the cases where the redexes are nested.