TYPES Summer School 2005

Proofs of Programs
and
Formalisation of Mathematics

Lecture Notes

Volume I

August 15–26 2005

Göteborg, Sweden

Proofs of Programs and Formalisation of Mathematics

Lecture Notes: Volume I

Introduction to Type Theory

Foundations

TYPES Summer School

August 15–26 2005

Göteborg, Sweden

Foreword

The summer school *Proofs of Programs and Formalisation of Mathematics* was held in Göteborg, Sweden from 15th to 26th of August 2005, under the sponsorship of the IST (Information Society Technologies) program of the European Union, Chalmers University of Technology and Göteborg University.

This school followed three previous TYPES summer schools: Båstad, Sweden in 1993, Giens, France in 1999, and Giens, France in 2002.

The aim of this school was to provide students with the most recent knowledge on formal proofs and interactive proof development.

The program of the school consisted on six parts: Introduction to Type Theory, Foundations, Introduction to the Systems *Agda*, *Coq* and *Isabelle*, Advanced Applications and Tools, Dependently Typed Programming, and Formalisation of Mathematics.

We would like to thank all the researchers who contributed to the success of this school by presenting their work. In addition, we would like to thank all the people who contributed to the organisation of the school.

Göteborg, Sweden, July 2005
Ana Bove and Bengt Nordström,
on behalf of the organising committee

Contents

PART I: Introduction to Type Theory

Herman Geuvers: Lambda-calculus, type systems

Bengt Nordström: Types, propositions and problems

PART II: Foundations

Gilles Dowek: Mixing deductions and computations

PART I:

Introduction to Type Theory

Herman Geuvers:

Lambda-calculus, type systems

Introduction to Lambda Calculus

Henk Barendregt   Erik Barendsen

Revised edition
December 1998, March 2000

Contents

1 Introduction **5**

2 Conversion **9**

3 The Power of Lambda **17**

4 Reduction **23**

5 Type Assignment **33**

6 Extensions **41**

7 Reduction Systems **47**

Bibliography **51**

Chapter 1

Introduction

Some history

Leibniz had as ideal the following.

(1) *Create a 'universal language' in which all possible problems can be stated.*

(2) *Find a decision method to solve all the problems stated in the universal language.*

If one restricts oneself to mathematical problems, point (1) of Leibniz' ideal is fulfilled by taking some form of set theory formulated in the language of first order predicate logic. This was the situation after Frege and Russell (or Zermelo).

Point (2) of Leibniz' ideal became an important philosophical question. 'Can one solve all problems formulated in the universal language?' It seems not, but it is not clear how to prove that. This question became known as the *Entscheidungsproblem*.

In 1936 the *Entscheidungsproblem* was solved in the negative independently by Alonzo Church and Alan Turing. In order to do so, they needed a formalisation of the intuitive notion of 'decidable', or what is equivalent 'computable'. Church and Turing did this in two different ways by introducing two models of computation.

(1) Church (1936) invented a formal system called the *lambda calculus* and defined the notion of computable function via this system.

(2) Turing (1936/7) invented a class of machines (later to be called *Turing machines*) and defined the notion of computable function via these machines.

Also in 1936 Turing proved that both models are equally strong in the sense that they define the same class of computable functions (see Turing (1937)).

Based on the concept of a Turing machine are the present day *Von Neumann computers*. Conceptually these are Turing machines with random access registers. *Imperative programming languages* such as *Fortran*, *Pascal* etcetera as well as all the assembler languages are based on the way a Turing machine is instructed: by a sequence of statements.

Functional programming languages, like *Miranda*, *ML* etcetera, are based on the lambda calculus. An early (although somewhat hybrid) example of such a language is *Lisp*. *Reduction machines* are specifically designed for the execution of these functional languages.

Reduction and functional programming

A *functional program* consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. Reduction consists of replacing a part P of E by another expression P' according to the given rewrite rules. In schematic notation

$$E[P] \to E[P'],$$

provided that $P \to P'$ is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called *normal form* E^* of the expression E consists of the output of the given functional program.

An example:

$$\begin{aligned}(7+4)*(8+5*3) &\to 11*(8+5*3)\\ &\to 11*(8+15)\\ &\to 11*23\\ &\to 253.\end{aligned}$$

In this example the reduction rules consist of the 'tables' of addition and of multiplication on the numerals.

Also symbolic computations can be done by reduction. For example

first of (sort (append ('dog', 'rabbit') (sort (('mouse', 'cat'))))) →
→ first of (sort (append ('dog', 'rabbit') ('cat', 'mouse')))
→ first of (sort ('dog', 'rabbit', 'cat', 'mouse'))
→ first of ('cat', 'dog', 'mouse', 'rabbit')
→ 'cat'.

The necessary rewrite rules for append and sort can be programmed easily in a few lines. Functions like append given by some rewrite rules are called *combinators*.

Reduction systems usually satisfy the *Church-Rosser property*, which states that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example may be reduced as follows:

$$\begin{aligned}(7+4)*(8+5*3) &\to (7+4)*(8+15)\\ &\to 11*(8+15)\\ &\to 11*23\\ &\to 253,\end{aligned}$$

or even by evaluating several expressions at the same time:

$$\begin{aligned}(7+4)*(8+5*3) &\to 11*(8+15)\\ &\to 11*23\\ &\to 253.\end{aligned}$$

Application and abstraction

The first basic operation of the λ-calculus is *application*. The expression

$$F \cdot A$$

or

$$FA$$

denotes the data F considered as algorithm applied to the data A considered as input. This can be viewed in two ways: either as the process of computation FA or as the output of this process. The first view is captured by the notion of conversion and even better of reduction; the second by the notion of models (semantics).

The theory is *type-free*: it is allowed to consider expressions like FF, that is F applied to itself. This will be useful to simulate recursion.

The other basic operation is *abstraction*. If $M \equiv M[x]$ is an expression containing ('depending on') x, then $\lambda x.M[x]$ denotes the function $x \mapsto M[x]$. Application and abstraction work together in the following intuitive formula.

$$(\lambda x.2 * x + 1)3 = 2 * 3 + 1 \quad (= 7).$$

That is, $(\lambda x.2 * x + 1)3$ denotes the function $x \mapsto 2 * x + 1$ applied to the argument 3 giving $2*3+1$ which is 7. In general we have $(\lambda x.M[x])N = M[N]$. This last equation is preferably written as

$$(\lambda x.M)N = M[x := N], \tag{β}$$

where $[x := N]$ denotes substitution of N for x. It is remarkable that although (β) is the only essential axiom of the λ-calculus, the resulting theory is rather involved.

Free and bound variables

Abstraction is said to *bind* the *free* variable x in M. E.g. we say that $\lambda x.yx$ has x as bound and y as free variable. Substitution $[x := N]$ is only performed in the free occurrences of x:

$$yx(\lambda x.x)[x := N] \equiv yN(\lambda x.x).$$

In calculus there is a similar variable binding. In $\int_a^b f(x, y)dx$ the variable x is bound and y is free. It does not make sense to substitute 7 for x: $\int_a^b f(7, y)d7$; but substitution for y makes sense: $\int_a^b f(x, 7)dx$.

For reasons of hygiene it will always be assumed that the bound variables that occur in a certain expression are different from the free ones. This can be fulfilled by renaming bound variables. E.g. $\lambda x.x$ becomes $\lambda y.y$. Indeed, these expressions act the same way:

$$(\lambda x.x)a = a = (\lambda y.y)a$$

and in fact they denote the same intended algorithm. Therefore expressions that differ only in the names of bound variables are identified.

Functions of more arguments

Functions of several arguments can be obtained by iteration of application. The idea is due to Schönfinkel (1924) but is often called *currying*, after H.B. Curry who introduced it independently. Intuitively, if $f(x,y)$ depends on two arguments, one can define

$$\begin{aligned} F_x &= \lambda y.f(x,y), \\ F &= \lambda x.F_x. \end{aligned}$$

Then

$$(Fx)y = F_x y = f(x,y). \qquad (*)$$

This last equation shows that it is convenient to use *association to the left* for iterated application:

$$FM_1 \cdots M_n \text{ denotes } (\cdot\cdot((FM_1)M_2)\cdots M_n).$$

The equation $(*)$ then becomes

$$Fxy = f(x,y).$$

Dually, iterated abstraction uses *association to the right*:

$$\lambda x_1 \cdots x_n.f(x_1,\ldots,x_n) \text{ denotes } \lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.f(x_1,\ldots,x_n))\cdot\cdot)).$$

Then we have for F defined above

$$F = \lambda xy.f(x,y)$$

and $(*)$ becomes

$$(\lambda xy.f(x,y))xy = f(x,y).$$

For n arguments we have

$$(\lambda x_1 \cdots x_n.f(x_1,\cdots,x_n))x_1 \cdots x_n = f(x_1,\ldots,x_n)$$

by using n times (β). This last equation becomes in convenient vector notation

$$(\lambda \vec{x}.f[\vec{x}])\vec{x} = f[\vec{x}];$$

more generally one has

$$(\lambda \vec{x}.f[\vec{x}])\vec{N} = f[\vec{N}].$$

Chapter 2

Conversion

In this chapter, the λ-calculus will be introduced formally.

2.1. DEFINITION. The set of λ-*terms* (notation Λ) is built up from an infinite set of variables $V = \{v, v', v'', \ldots\}$ using application and (function) abstraction.

$$\begin{array}{rcl} x \in V & \Rightarrow & x \in \Lambda, \\ M, N \in \Lambda & \Rightarrow & (MN) \in \Lambda, \\ M \in \Lambda, x \in V & \Rightarrow & (\lambda x M) \in \Lambda. \end{array}$$

In BN-form this is

$$\begin{array}{rcl} \textsf{variable} & ::= & \text{`}v\text{'} \mid \textsf{variable}\ \text{`}'\text{'} \\ \lambda\textsf{-term} & ::= & \textsf{variable} \mid \text{`('}\ \lambda\textsf{-term}\ \lambda\textsf{-term}\ \text{`)'} \mid \text{`(}\lambda\text{'}\ \textsf{variable}\ \lambda\textsf{-term}\ \text{`)'} \end{array}$$

2.2. EXAMPLE. The following are λ-terms.

$$\begin{array}{c} v'; \\ (v'v); \\ (\lambda v(v'v)); \\ ((\lambda v(v'v))v''); \\ (((\lambda v(\lambda v'(v'v)))v'')v'''). \end{array}$$

2.3. CONVENTION. (i) $x, y, z, \ldots$ denote arbitrary variables; $M, N, L, \ldots$ denote arbitrary λ-terms. Outermost parentheses are not written.

(ii) $M \equiv N$ denotes that M and N are the same term or can be obtained from each other by renaming bound variables. E.g.

$$\begin{array}{rcl} (\lambda xy)z & \equiv & (\lambda xy)z; \\ (\lambda xx)z & \equiv & (\lambda yy)z; \\ (\lambda xx)z & \not\equiv & z; \\ (\lambda xx)z & \not\equiv & (\lambda xy)z. \end{array}$$

(iii) We use the abbreviations

$$FM_1 \cdots M_n \equiv (\cdot\cdot((FM_1)M_2) \cdots M_n)$$

and

$$\lambda x_1 \cdots x_n.M \equiv \lambda x_1(\lambda x_2(\cdots(\lambda x_n(M))\cdot\cdot)).$$

The terms in Example 2.2 now may be written as follows.

$$y;$$
$$yx;$$
$$\lambda x.yx;$$
$$(\lambda x.yx)z;$$
$$(\lambda xy.yx)zw.$$

Note that $\lambda x.yx$ is $(\lambda x(yx))$ and not $((\lambda x.y)x)$.

2.4. Definition. (i) The set of *free variables* of M, notation $\mathrm{FV}(M)$, is defined inductively as follows.

$$\begin{array}{rcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(MN) & = & \mathrm{FV}(M) \cup \mathrm{FV}(N); \\ \mathrm{FV}(\lambda x.M) & = & \mathrm{FV}(M) - \{x\}. \end{array}$$

A variable in M is *bound* if it is not free. Note that a variable is bound if it occurs under the scope of a λ.

(ii) M is a *closed λ-term* (or *combinator*) if $\mathrm{FV}(M) = \emptyset$. The set of closed λ-terms is denoted by Λ^o.

(iii) The result of *substituting* N for the free occurences of x in M, notation $M[x := N]$, is defined as follows.

$$\begin{array}{rcl} x[x := N] & \equiv & N; \\ y[x := N] & \equiv & y, \quad \text{if } x \not\equiv y; \\ (M_1 M_2)[x := N] & \equiv & (M_1[x := N])(M_2[x := N]); \\ (\lambda y.M_1)[x := N] & \equiv & \lambda y.(M_1[x := N]). \end{array}$$

2.5. Example. Consider the λ-term

$$\lambda xy.xyz.$$

Then x and y are bound variables and z is a free variable. The term $\lambda xy.xxy$ is closed.

2.6. Variable convention. If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

Note that in the fourth clause of Definition 2.4 (iii) it is not needed to say 'provided that $y \not\equiv x$ and $y \notin \mathrm{FV}(N)$'. By the variable convention this is the case.

Now we can introduce the λ-calculus as formal theory.

2.7. DEFINITION. (i) The principal axiom scheme of the λ-calculus is

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

for all $M, N \in \Lambda$.

(ii) There are also the 'logical' axioms and rules.

Equality:

$$M = M;$$
$$M = N \Rightarrow N = M;$$
$$M = N, N = L \Rightarrow M = L.$$

Compatibility rules:

$$M = M' \Rightarrow MZ = M'Z;$$
$$M = M' \Rightarrow ZM = ZM';$$
$$M = M' \Rightarrow \lambda x.M = \lambda x.M'. \qquad (\xi)$$

(iii) If $M = N$ is provable in the λ-calculus, then we sometimes write $\boldsymbol{\lambda} \vdash M = N$.

As a consequence of the compatibility rules, one can replace (sub)terms by equal terms in any term context:

$$M = N \Rightarrow \boxed{\cdots M \cdots} = \boxed{\cdots N \cdots}.$$

For example, $(\lambda y.yy)x = xx$, so

$$\boldsymbol{\lambda} \vdash \lambda x.x((\lambda y.yy)x)x = \lambda x.x(xx)x.$$

2.8. REMARK. We have identified terms that differ only in the names of bound variables. An alternative is to add to the λ-calculus the following axiom scheme

$$\lambda x.M = \lambda y.M[x := y], \quad \text{provided that } y \text{ does not occur in } M. \qquad (\alpha)$$

We prefer our version of the theory in which the identifications are made on syntactic level. These identifications are done in our mind and not on paper. For implementations of the λ-calculus the machine has to deal with this so called α-conversion. A good way of doing this is provided by the name-free notation of de Bruijn, see Barendregt (1984), Appendix C.

2.9. LEMMA. $\boldsymbol{\lambda} \vdash (\lambda x_1 \cdots x_n.M)X_1 \cdots X_n = M[x_1 := X_1] \cdots [x_n := X_n]$.

PROOF. By the axiom (β) we have

$$(\lambda x_1.M)X_1 = M[x_1 := X_1].$$

By induction on n the result follows. $\square$

2.10. Example (Standard combinators). Define the combinators

$$\begin{aligned} \mathsf{I} &\equiv \lambda x.x; \\ \mathsf{K} &\equiv \lambda xy.x; \\ \mathsf{K}_* &\equiv \lambda xy.y; \\ \mathsf{S} &\equiv \lambda xyz.xz(yz). \end{aligned}$$

Then, by Lemma 2.9, we have the following equations.

$$\begin{aligned} \mathsf{I}M &= M; \\ \mathsf{K}MN &= M; \\ \mathsf{K}_*MN &= N; \\ \mathsf{S}MNL &= ML(NL). \end{aligned}$$

Now we can solve simple equations.

2.11. Example. $\exists G\ \forall X\ GX = XXX$ (there exists $G \in \Lambda$ such that for all $X \in \Lambda$ one has $\boldsymbol{\lambda} \vdash GX = XX$). Indeed, take $G \equiv \lambda x.xxx$ and we are done.

Recursive equations require a special technique. The following result provides one way to represent recursion in the λ-calculus.

2.12. Fixedpoint Theorem. (i) $\forall F\ \exists X\ FX = X$. (*This means: for all $F \in \Lambda$ there is an $X \in \Lambda$ such that $\boldsymbol{\lambda} \vdash FX = X$.*)

(ii) *There is a fixed point combinator*

$$\mathsf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

such that

$$\forall F\ F(\mathsf{Y}F) = \mathsf{Y}F.$$

Proof. (i) Define $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then

$$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX.$$

(ii) By the proof of (i). □

2.13. Example. (i) $\exists G\ \forall X\ GX = \mathsf{S}GX$. Indeed,

$$\begin{aligned} \forall X\ GX = \mathsf{S}GX &\Leftarrow Gx = \mathsf{S}Gx \\ &\Leftarrow G = \lambda x.\mathsf{S}Gx \\ &\Leftarrow G = (\lambda gx.\mathsf{S}gx)G \\ &\Leftarrow G \equiv \mathsf{Y}(\lambda gx.\mathsf{S}gx). \end{aligned}$$

Note that one can also take $G \equiv \mathsf{YS}$.

(ii) $\exists G\ \forall X\ GX = GG$: take $G \equiv \mathsf{Y}(\lambda gx.gg)$. (Can you solve this without using Y?)

In the lambda calculus one can define numerals and represent numeric functions on them.

2.14. DEFINITION. (i) $F^n(M)$ with $F \in \Lambda$ and $n \in \mathbb{N}$ is defined inductively as follows.

$$\begin{aligned} F^0(M) &\equiv M; \\ F^{n+1}(M) &\equiv F(F^n(M)). \end{aligned}$$

(ii) The *Church numerals* $\boldsymbol{c}_0, \boldsymbol{c}_1, \boldsymbol{c}_2, \ldots$ are defined by

$$\boldsymbol{c}_n \equiv \lambda f x. f^n(x).$$

2.15. PROPOSITION (J.B. Rosser). *Define*

$$\begin{aligned} \mathsf{A}_+ &\equiv \lambda xypq.xp(ypq); \\ \mathsf{A}_* &\equiv \lambda xyz.x(yz); \\ \mathsf{A}_{\exp} &\equiv \lambda xy.yx. \end{aligned}$$

Then one has for all $n, m \in \mathbb{N}$

(i) $\mathsf{A}_+\boldsymbol{c}_n\boldsymbol{c}_m = \boldsymbol{c}_{n+m}$.

(ii) $\mathsf{A}_*\boldsymbol{c}_n\boldsymbol{c}_m = \boldsymbol{c}_{n*m}$.

(iii) $\mathsf{A}_{\exp}\boldsymbol{c}_n\boldsymbol{c}_m = \boldsymbol{c}_{(n^m)}$, *except for* $m = 0$ *(Rosser started counting from* 1*).*

In the proof we need the following.

2.16. LEMMA. (i) $(\boldsymbol{c}_n x)^m(y) = x^{n*m}(y)$.

(ii) $(\boldsymbol{c}_n)^m(x) = \boldsymbol{c}_{(n^m)}(x)$, *for* $m > 0$.

PROOF. (i) Induction on m. If $m = 0$, then LHS $= y =$ RHS. Assume (i) is correct for m (Induction Hypothesis: IH). Then

$$\begin{aligned} (\boldsymbol{c}_n x)^{m+1}(y) &= \boldsymbol{c}_n x((\boldsymbol{c}_n x)^m(y)) \\ &= \boldsymbol{c}_n x(x^{n*m}(y)) \quad \text{by IH,} \\ &= x^n(x^{n*m}(y)) \\ &\equiv x^{n+n*m}(y) \\ &\equiv x^{n*(m+1)}(y). \end{aligned}$$

(ii) Induction on $m > 0$. If $m = 1$, then LHS $\equiv \boldsymbol{c}_n x \equiv$ RHS. If (ii) is correct for m, then

$$\begin{aligned} (\boldsymbol{c}_n)^{m+1}(x) &= \boldsymbol{c}_n((\boldsymbol{c}_n)^m(x)) \\ &= \boldsymbol{c}_n(\boldsymbol{c}_{(n^m)}(x)) \quad \text{by IH,} \\ &= \lambda y.(\boldsymbol{c}_{(n^m)}(x))^n(y) \\ &= \lambda y.x^{n^m * n}(y) \quad \text{by (i),} \\ &= \boldsymbol{c}_{(n^{m+1})}x. \end{aligned}$$

PROOF OF THE PROPOSITION. (i) Exercise.

(ii) Exercise. Use Lemma 2.16 (i).

(iii) By Lemma 2.16 (ii) we have for $m > 0$

$$\begin{aligned} \mathsf{A}_{\exp}\boldsymbol{c}_n\boldsymbol{c}_m &= \boldsymbol{c}_m\boldsymbol{c}_n \\ &= \lambda x.(\boldsymbol{c}_n)^m(x) \\ &= \lambda x.\boldsymbol{c}_{(n^m)}x \\ &= \boldsymbol{c}_{(n^m)}, \end{aligned}$$

since $\lambda x.Mx = M$ if $M \equiv \lambda y.M'[y]$ and $x \notin \mathrm{FV}(M)$. Indeed,

$$\begin{aligned} \lambda x.Mx &\equiv \lambda x.(\lambda y.M'[y])x \\ &= \lambda x.M'[x] \\ &\equiv \lambda y.M'[y] \\ &\equiv M. \ \Box \end{aligned}$$

Exercises

2.1. (i) Rewrite according to official syntax

$$M_1 \equiv y(\lambda x.xy(\lambda zw.yz)).$$

(ii) Rewrite according to the simplified syntax

$$M_2 \equiv \lambda v'(\lambda v''((((\lambda vv)v')v'')((v''(\lambda v'''(v'v''')))v''))).$$

2.2. Prove the following *substitution lemma*. Let $x \not\equiv y$ and $x \notin \mathrm{FV}(L)$. Then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

2.3. (i) Prove, using Exercise 2.2,

$$\boldsymbol{\lambda} \vdash M_1 = M_2 \ \Rightarrow \ \boldsymbol{\lambda} \vdash M_1[x := N] = M_2[x := N].$$

(ii) Show

$$\boldsymbol{\lambda} \vdash M_1 = M_2 \,\&\, \boldsymbol{\lambda} \vdash N_1 = N_2 \ \Rightarrow \ \boldsymbol{\lambda} \vdash M_1[x := N_1] = M_2[x := N_2].$$

2.4. Prove Proposition 2.15 (i), (ii).

2.5. Let $\mathsf{B} \equiv \lambda xyz.x(yz)$. Simplify $M \equiv \mathsf{B}XYZ$, that is find a 'simple' term N such that $\boldsymbol{\lambda} \vdash M = N$.

2.6. Simplify the following terms.
(i) $M \equiv (\lambda xyz.zyx)aa(\lambda pq.q)$;
(ii) $M \equiv (\lambda yz.zy)((\lambda x.xxx)(\lambda x.xxx))(\lambda w.\mathsf{I})$;
(iii) $M \equiv \mathsf{SKSKSK}$.

2.7. Show that
(i) $\boldsymbol{\lambda} \vdash \mathsf{KI} = \mathsf{K}_*$;
(ii) $\boldsymbol{\lambda} \vdash \mathsf{SKK} = \mathsf{I}$.

2.8. (i) Write down a closed λ-term $F \in \Lambda$ such that for all $M, N \in \Lambda$

$$FMN = M(NM)N.$$

(ii) Construct a λ-term F such that for all $M, N, L \in \Lambda^o$

$$FMNL = N(\lambda x.M)(\lambda yz.yLM).$$

2.9. Find closed terms F such that
(i) $Fx = x\mathbf{I}$;
(ii) $Fxy = x\mathbf{I}y$.

2.10. Find closed terms F such that
(i) $Fx = F$. This term can be called the 'eater' and is often denoted by $\mathbf{K}_\infty$;
(ii) $Fx = xF$;
(iii) $F\mathbf{IKK} = F\mathbf{K}$.

2.11. Show

$$\forall C[\ ,\ ]\ \exists F\ \forall \vec{x}\ F\vec{x} = C[F, \vec{x}]$$

and take another look at the exercises 2.8, 2.9 and 2.10.

2.12. Let $P, Q \in \Lambda$. P and Q are *incompatible*, notation $P \sharp Q$, if $\boldsymbol{\lambda}$ extended with $P = Q$ as axiom proves every equation between λ-terms, i.e. for all $M, N \in \Lambda$ one has $\boldsymbol{\lambda} + (P = Q) \vdash M = N$. In this case one says that $\boldsymbol{\lambda} + (P = Q)$ is *inconsistent*.
(i) Prove that for $P, Q \in \Lambda$

$$P \sharp Q \ \Leftrightarrow\ \boldsymbol{\lambda} + (P = Q) \vdash \mathbf{true} = \mathbf{false},$$

where $\mathbf{true} \equiv \mathbf{K}$, $\mathbf{false} \equiv \mathbf{K}_*$.
(ii) Show that $\mathbf{I} \sharp \mathbf{K}$.
(iii) Find a λ-term F such that $F\mathbf{I} = x$ and $F\mathbf{K} = y$.
(iv) Show that $\mathbf{K} \sharp \mathbf{S}$.

2.13. Write down a grammar in BN-form that generates the λ-terms exactly in the way they are written in Convention 2.3.

Chapter 3

The Power of Lambda

We have seen that the function plus, times and exponentiation on $\mathbb{N}$ can be represented in the λ-calculus using Church's numerals. We will now show that all computable (recursive) functions can be represented in the λ-calculus. In order to do this we will use first a different system of numerals.

Truth values and a conditional can be represented in the λ-calculus.

3.1. Definition. (i) $\textsf{\textbf{true}} \equiv \textsf{\textbf{K}}$, $\textsf{\textbf{false}} \equiv \textsf{\textbf{K}}_*$.

(ii) If B is considered as a Boolean, i.e. a term that is either **true** or **false**, then

$$\textsf{if } B \textsf{ then } P \textsf{ else } Q$$

can be represented by

$$BPQ.$$

3.2. Definition (Pairing). For $M, N \in \Lambda$ write

$$[M, N] \equiv \lambda z.\textsf{if } z \textsf{ then } M \textsf{ else } N \qquad (\equiv \lambda z.zMN).$$

Then

$$\begin{aligned} [M, N]\textsf{\textbf{true}} &= M, \\ [M, N]\textsf{\textbf{false}} &= N, \end{aligned}$$

and hence $[M, N]$ can serve as an ordered pair.

We can use this pairing construction for an alternative representation of natural numbers due to Barendregt (1976).

3.3. Definition. For each $n \in \mathbb{N}$, the numeral $\ulcorner n \urcorner$ is defined inductively as follows.

$$\begin{aligned} \ulcorner 0 \urcorner &\equiv \textsf{\textbf{I}}, \\ \ulcorner n+1 \urcorner &\equiv [\textsf{\textbf{false}}, \ulcorner n \urcorner]. \end{aligned}$$

3.4. LEMMA (Successor, predecessor, test for zero). *There exist combinators* S^+, P^-, *and* **Zero** *such that*

$$\begin{aligned} \mathsf{S}^+ \ulcorner n \urcorner &= \ulcorner n+1 \urcorner, \\ \mathsf{P}^- \ulcorner n+1 \urcorner &= \ulcorner n \urcorner, \\ \mathsf{Zero} \ulcorner 0 \urcorner &= \mathsf{true}, \\ \mathsf{Zero} \ulcorner n+1 \urcorner &= \mathsf{false}. \end{aligned}$$

PROOF. Take

$$\begin{aligned} \mathsf{S}^+ &\equiv \lambda x.[\mathsf{false}, x], \\ \mathsf{P}^- &\equiv \lambda x.x\, \mathsf{false}, \\ \mathsf{Zero} &\equiv \lambda x.x\, \mathsf{true}. \ \Box \end{aligned}$$

3.5. DEFINITION (Lambda definability). (i) A *numeric function* is a map

$$\varphi : \mathbb{N}^p \to \mathbb{N}$$

for some p. In this case φ is called p-ary.

(ii) A numeric p-ary function φ is called λ-*definable* if for some combinator F

$$F \ulcorner n_1 \urcorner \cdots \ulcorner n_p \urcorner = \ulcorner \varphi(n_1, \ldots, n_p) \urcorner \qquad (*)$$

for all $n_1, \ldots, n_p \in \mathbb{N}$. If $(*)$ holds, then φ is said to be λ-*defined* by F.

3.6. DEFINITION. The *initial functions* are the numeric functions U_i^n, S^+, Z defined by

$$\begin{aligned} U_i^n(x_1, \ldots, x_n) &= x_i, \quad (1 \le i \le n); \\ S^+(n) &= n+1; \\ Z(n) &= 0. \end{aligned}$$

Let P(n) be a numeric relation. As usual

$$\mu m[P(m)]$$

denotes the least number m such that $P(m)$ holds if there is such a number; otherwise it is undefined.

3.7. DEFINITION. Let $\mathcal{A}$ be a class of numeric functions.

(i) $\mathcal{A}$ is *closed under composition* if for all φ defined by

$$\varphi(\vec{n}) = \chi(\psi_1(\vec{n}), \ldots, \psi_m(\vec{n}))$$

with $\chi, \psi_1, \ldots, \psi_m \in \mathcal{A}$, one has $\varphi \in \mathcal{A}$.

(ii) $\mathcal{A}$ is *closed under primitive recursion* if for all φ defined by

$$\begin{aligned} \varphi(0, \vec{n}) &= \chi(\vec{n}), \\ \varphi(k+1, \vec{n}) &= \psi(\varphi(k, \vec{n}), k, \vec{n}) \end{aligned}$$

with $\chi, \psi \in \mathcal{A}$, one has $\varphi \in \mathcal{A}$.

(iii) $\mathcal{A}$ is *closed under minimalization* if for all φ defined by

$$\varphi(\vec{n}) = \mu m[\chi(\vec{n}, m) = 0]$$

with $\chi \in \mathcal{A}$ such that

$$\forall \vec{n}\ \exists m\ \chi(\vec{n}, m) = 0,$$

one has $\varphi \in \mathcal{A}$.

3.8. DEFINITION. The class $\mathcal{R}$ of *recursive functions* is the smallest class of numeric functions that contains all initial functions and is closed under composition, primitive recursion and minimalization.

So $\mathcal{R}$ is an inductively defined class. The proof that all recursive functions are λ-definable is in fact by a corresponding induction argument. The result is originally due to Kleene (1936).

3.9. LEMMA. *The initial functions are λ-definable.*

PROOF. Take as defining terms

$$\begin{array}{rcl} \mathsf{U}_i^n & \equiv & \lambda x_1 \cdots x_n.x_i, \\ \mathsf{S}^+ & \equiv & \lambda x.[\mathsf{false}, x] \quad \text{(see Lemma 3.4)} \\ \mathsf{Z} & \equiv & \lambda x.\ulcorner 0 \urcorner. \ \square \end{array}$$

3.10. LEMMA. *The λ-definable functions are closed under composition.*

PROOF. Let $\chi, \psi_1, \ldots, \psi_m$ be λ-defined by $G, H_1, \ldots, H_m$ respectively. Then

$$\varphi(\vec{n}) = \chi(\psi_1(\vec{n}), \ldots, \psi_m(\vec{n}))$$

is λ-defined by

$$F \equiv \lambda \vec{x}.G(H_1\vec{x}) \cdots (H_m\vec{x}). \ \square$$

As to primitive recursion, let us first consider an example. The addition function can be specified as follows.

$$\begin{array}{rcl} \text{Add}(0, y) & = & y, \\ \text{Add}(x+1, y) & = & 1 + \text{Add}(x, y) \ = \ S^+(\text{Add}(x, y)). \end{array}$$

An intuitive way to compute Add(m, n) us the following.

Test whether $m = 0$.
If yes: give output n;
if no: compute Add$(m-1, n)$ and give its successor as output.

Therefore we want a term **Add** such that

$$\mathsf{Add}\, xy = \mathsf{if}\ \mathsf{Zero}\, x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{S}^+(\mathsf{Add}(\mathsf{P}^- x)y).$$

This equation can be solved using the fixedpoint combinator: take

$$\mathsf{Add} \equiv \mathsf{Y}(\lambda axy.\mathsf{if}\ \mathsf{Zero}\, x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{S}^+(a(\mathsf{P}^- x)y)).$$

The general case is treated as follows.

3.11. LEMMA. *The λ-definable functions are closed under primitive recursion.*

PROOF. Let φ be defined by

$$\begin{aligned} \varphi(0,\vec{n}) &= \chi(\vec{n}), \\ \varphi(k+1,\vec{n}) &= \psi(\varphi(k,\vec{n}),k,\vec{n}), \end{aligned}$$

where χ, ψ are λ-defined by G, H respectively. Now we want a term F such that

$$\begin{aligned} Fx\vec{y} &= \textsf{if } \textbf{Zero}\, x \textsf{ then } G\vec{y} \textsf{ else } H(F(\mathbf{P}^{-}x)\vec{y})(\mathbf{P}^{-}x)\vec{y} \\ &\equiv D(F,x,\vec{y}), \text{ say.} \end{aligned}$$

It is sufficient to find an F such that

$$\begin{aligned} F &= \lambda x\vec{y}.D(F,x,\vec{y}) \\ &= (\lambda f x\vec{y}.D(f,x,\vec{y}))F. \end{aligned}$$

Now such an F can be found by the Fixedpoint Theorem and we are done. □

3.12. LEMMA. *The λ-definable functions are closed under minimalization.*

PROOF. Let φ be defined by

$$\varphi(\vec{n}) = \mu m[\chi(\vec{n},m) = 0],$$

where χ is λ-defined by G. Again by the Fixedpoint Theorem there is a term H such that

$$\begin{aligned} H\vec{x}y &= \textsf{if } \textbf{Zero}(G\vec{x}y) \textsf{ then } y \textsf{ else } H\vec{x}(\mathbf{S}^{+}y) \\ &= (\lambda h\vec{x}y.E(h,\vec{x},y))H\vec{x}y, \text{ say.} \end{aligned}$$

Set $F \equiv \lambda\vec{x}.H\vec{x}\ulcorner 0\urcorner$. Then F λ-defines φ:

$$\begin{aligned} F\ulcorner\vec{n}\urcorner &= H\ulcorner\vec{n}\urcorner\ulcorner 0\urcorner \\ &= \ulcorner 0\urcorner && \text{if } G\ulcorner\vec{n}\urcorner\ulcorner 0\urcorner = \ulcorner 0\urcorner \\ &= H\ulcorner\vec{n}\urcorner\ulcorner 1\urcorner && \text{else} \\ &= \ulcorner 1\urcorner && \text{if } G\ulcorner\vec{n}\urcorner\ulcorner 1\urcorner = \ulcorner 0\urcorner \\ &= H\ulcorner\vec{n}\urcorner\ulcorner 2\urcorner && \text{else} \\ &= \ulcorner 2\urcorner && \text{if} \ldots \\ &= \ldots \quad \square \end{aligned}$$

3.13. THEOREM. *All recursive functions are λ-definable.*

PROOF. By the lemmas 3.9–3.12. □

The converse also holds. So for numeric functions we have φ is recursive iff φ is λ-definable. Moreover also for partial functions a notion of λ-definability exists. If ψ is a partial numeric function, then we have

$$\psi \text{ is partial recursive } \Leftrightarrow \psi \text{ is } \lambda\text{-definable.}$$

3.14. THEOREM. *With respect to the Church numerals $\boldsymbol{c}_n$ all recursive functions can be λ-defined.*

PROOF. Define

$$\begin{aligned} \mathsf{S}^+_{\boldsymbol{c}} &\equiv \lambda xyz.y(xyz), \\ \mathsf{P}^-_{\boldsymbol{c}} &\equiv \lambda xyz.x(\lambda pq.q(py))(\mathsf{K}z)\mathsf{I}^1, \\ \mathsf{Zero}_{\boldsymbol{c}} &\equiv \lambda x.x(\mathsf{K}\,\mathsf{false})\mathsf{true}. \end{aligned}$$

Then these terms represent the successor, predecessor and test for zero. Then as before all recursive functions can be λ-defined. □

An alternative proof uses 'translators' between the numerals $\ulcorner n \urcorner$ and $\boldsymbol{c}_n$.

3.15. PROPOSITION. *There exist terms T, T^{-1} such that for all n*

$$\begin{aligned} T\boldsymbol{c}_n &= \ulcorner n \urcorner; \\ T^{-1}\ulcorner n \urcorner &= \boldsymbol{c}_n. \end{aligned}$$

PROOF. Construct T, T^{-1} such that

$$\begin{aligned} T &\equiv \lambda x.x\mathsf{S}^+\ulcorner 0 \urcorner. \\ T^{-1} &= \lambda x.\mathsf{if}\ \mathsf{Zero}\,x\ \mathsf{then}\ \boldsymbol{c}_0\ \mathsf{else}\ \mathsf{S}^+_{\boldsymbol{c}}(T^{-1}(\mathsf{P}^-x)). \ \square \end{aligned}$$

3.16. COROLLARY (Second proof of Theorem 3.14). *Let φ be a recursive function (of arity 2 say). Let F represent φ with respect to the numerals $\ulcorner n \urcorner$. Define*

$$F_{\boldsymbol{c}} \equiv \lambda xy.T^{-1}(F(Tx)(Ty)).$$

Then $F_{\boldsymbol{c}}$ represents φ with respect to the Church numerals. □

The representation of pairs in the lambda calculus can also be used to solve multiple fixedpoint equations.

3.17. MULTIPLE FIXEDPOINT THEOREM. *Let $F_1, \ldots, F_n$ be λ-terms. Then we can find $X_1, \ldots, X_n$ such that*

$$\begin{aligned} X_1 &= F_1 X_1 \cdots X_n, \\ &\vdots \\ X_n &= F_n X_1 \cdots X_n. \end{aligned}$$

Observe that for $n = 1$ this is the ordinary Fixedpoint Theorem (2.12).

PROOF. We treat the case $n = 2$. So we want

$$\begin{aligned} X_1 &= F_1 X_1 X_2, \\ X_2 &= F_2 X_1 X_2. \end{aligned}$$

[1]Term found by J. Velmans.

The trick is to construct X_1 and X_2 simultaneously, as a pair. By the ordinary Fixedpoint Theorem we can find an X such that

$$X = [F_1(X\textsf{true})(X\textsf{false}), F_2(X\textsf{true})(X\textsf{false})].$$

Now define $X_1 \equiv X\textsf{true}$, $X_2 \equiv X\textsf{false}$. Then the result follows. This can be generalized to arbitrary n. $\square$

3.18. Example. There exist $G, H \in \Lambda$ such that

$$\begin{aligned} Gxy &= Hy(\textsf{K}x), \\ Hx &= G(xx)(\textsf{S}(H(xx))). \end{aligned}$$

Indeed, we can replace the above equations by

$$\begin{aligned} G &= \lambda xy.Hy(\textsf{K}x), \\ H &= \lambda x.G(xx)(\textsf{S}(H(xx))), \end{aligned}$$

and apply the Multiple Fixedpoint Theorem with $F_1 \equiv \lambda ghxy.hy(\textsf{K}x)$ and $F_2 \equiv \lambda ghx.g(xx)(\textsf{S}(h(xx)))$.

Exercises

3.1. (i) Find a λ-term **Mult** such that for all $m, n \in \mathbb{N}$

$$\textsf{Mult}\ulcorner n \urcorner \ulcorner m \urcorner = \ulcorner n \cdot m \urcorner.$$

(ii) Find a λ-term **Fac** such that for all $n \in \mathbb{N}$

$$\textsf{Fac}\ulcorner n \urcorner = \ulcorner n! \urcorner.$$

3.2. The *simple Ackermann function* φ is defined as follows.

$$\begin{aligned} \varphi(0, n) &= n + 1, \\ \varphi(m + 1, 0) &= \varphi(m, 1), \\ \varphi(m + 1, n + 1) &= \varphi(m, \varphi(m + 1, n)). \end{aligned}$$

Find a λ-term F that λ-defines φ.

3.3. Construct λ-terms $M_0, M_1, \ldots$ such that for all n one has

$$\begin{aligned} M_0 &= x, \\ M_{n+1} &= M_{n+2}M_n. \end{aligned}$$

3.4. Verify that $\textsf{P}_c^-$ (see the first proof of Theorem 3.14) indeed λ-defines the predecessor function with respect to the Church numerals.

Chapter 4

Reduction

There is a certain asymmetry in the basic scheme (β). The statement

$$(\lambda x.x^2 + 1)3 = 10$$

can be interpreted as '10 is the result of computing $(\lambda x.x^2 + 1)3$', but not vice versa. This computational aspect will be expressed by writing

$$(\lambda x.x^2 + 1)3 \twoheadrightarrow 10$$

which reads '$(\lambda x.x^2 + 1)3$ *reduces to* 10'.

Apart from this conceptual aspect, reduction is also useful for an analysis of convertibility. The Church-Rosser theorem says that if two terms are convertible, then there is a term to which they both reduce. In many cases the inconvertibility of two terms can be proved by showing that they do not reduce to a common term.

4.1. Definition. (i) A binary relation R on Λ is called *compatible* (with the operations) if

$$\begin{aligned} M\,R\,N \;\Rightarrow\; & (ZM)\,R\,(ZN), \\ & (MZ)\,R\,(NZ) \text{ and} \\ & (\lambda x.M)\,R\,(\lambda x.N). \end{aligned}$$

(ii) A *congruence* relation on Λ is a compatible equivalence relation.

(iii) A *reduction* relation on Λ is a compatible, reflexive and transitive relation.

4.2. Definition. The binary relations $\to_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ on Λ are defined inductively as follows.

(i) 1. $(\lambda x.M)N \to_\beta M[x := N]$;
 2. $M \to_\beta N \;\Rightarrow\; ZM \to_\beta ZN$, $MZ \to_\beta NZ$ and $\lambda x.M \to_\beta \lambda x.N$.

(ii) 1. $M \twoheadrightarrow_\beta M$;
 2. $M \to_\beta N \;\Rightarrow\; M \twoheadrightarrow_\beta N$;
 3. $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \;\Rightarrow\; M \twoheadrightarrow_\beta L$.

(iii) 1. $M \twoheadrightarrow_\beta N \;\Rightarrow\; M =_\beta N$;
2. $M =_\beta N \;\Rightarrow\; N =_\beta M$;
3. $M =_\beta N, N =_\beta L \;\Rightarrow\; M =_\beta$L.

These relations are pronounced as follows.

$$\begin{array}{rcl} M \twoheadrightarrow_\beta N & : & M\beta\text{-}reduces \text{ to } N; \\ M \to_\beta N & : & M\beta\text{-}reduces \text{ to } N in\ one\ step; \\ M =_\beta N & : & M \text{ is } \beta\text{-}convertible \text{ to } N. \end{array}$$

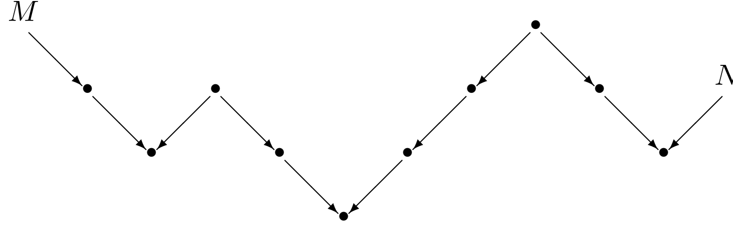
By definition $\to_\beta$ is compatible, $\twoheadrightarrow_\beta$ is a reduction relation and $=_\beta$ is a congruence relation.

4.3. Example. (i) Define

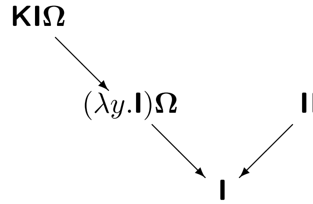
$$\begin{array}{rcl} \boldsymbol{\omega} & \equiv & \lambda x.xx, \\ \boldsymbol{\Omega} & \equiv & \boldsymbol{\omega\omega}. \end{array}$$

Then $\boldsymbol{\Omega} \to_\beta \boldsymbol{\Omega}$.
(ii) $\mathbf{KI\Omega} \twoheadrightarrow_\beta \mathbf{I}$.

Intuitively, $M =_\beta N$ if M is connected to N via $\to_\beta$-arrows (disregarding the directions of these). In a picture this looks as follows.



4.4. Example. $\mathbf{KI\Omega} =_\beta \mathbf{II}$. This is demonstrated by the following reductions.



4.5. Proposition. $M =_\beta N \;\Leftrightarrow\; \boldsymbol{\lambda} \vdash M = N$.

Proof. By an easy induction. □

4.6. Definition. (i) A *β-redex* is a term of the form $(\lambda x.M)N$. In this case $M[x := N]$ is its *contractum*.
(ii) A λ-term M is a *β-normal form* (β-nf) if it does not have a β-redex as subexpression.
(iii) A term M *has* a β-normal form if $M =_\beta N$ and N is a β-nf, for some N.

4.7. EXAMPLE. $(\lambda x.xx)y$ is not a β-nf, but has as β-nf the term yy.

An immediate property of nf's is the following.

4.8. LEMMA. *Let M be a β-nf. Then*

$$M \twoheadrightarrow_\beta N \ \Rightarrow\ N \equiv M.$$

PROOF. This is true if $\twoheadrightarrow_\beta$ is replaced by $\rightarrow_\beta$. Then the result follows by transitivity. □

4.9. CHURCH-ROSSER THEOREM. *If $M \twoheadrightarrow_\beta N_1$, $M \twoheadrightarrow_\beta N_2$, then for some N_3 one has $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$; in diagram*

M
N1 N2
N3

The proof is postponed until 4.19.

4.10. COROLLARY. *If $M =_\beta N$, then there is an L such that $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$.*

An intuitive proof of this fact proceeds by a tiling procedure: given an arrow path showing $M =_\beta N$, apply the Church-Rosser property repeatedly in order to find a common reduct. For the example given above this looks as follows.

M
N

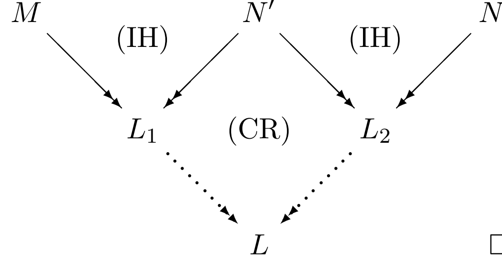
This is made precise below.

PROOF. Induction on the generation of $=_\beta$.

Case 1. $M =_\beta N$ because $M \twoheadrightarrow_\beta N$. Take $L \equiv N$.

Case 2. $M =_\beta N$ because $N =_\beta M$. By the IH there is a common β-reduct L_1 of N, M. Take $L \equiv L_1$.

Case 3. $M =_\beta N$ because $M =_\beta N'$, $N' =_\beta N$. Then



□

4.11. COROLLARY. (i) *If M has N as β-nf, then $M \twoheadrightarrow_\beta N$.*

(ii) *A λ-term has at most one β-nf.*

PROOF. (i) Suppose $M =_\beta N$ with N in β-nf. By Corollary 4.10 $M \twoheadrightarrow_\beta L$ and $N \twoheadrightarrow_\beta L$ for some L. But then $N \equiv L$, by Lemma 4.8, so $M \twoheadrightarrow_\beta N$.

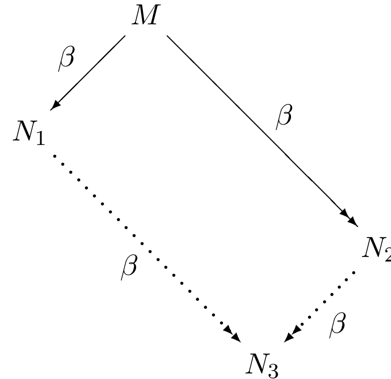
(ii) Suppose M has β-nf's N_1, N_2. Then $N_1 =_\beta N_2$ $(=_\beta M)$. By Corollary 4.10 $N_1 \twoheadrightarrow_\beta L$, $N_2 \twoheadrightarrow_\beta L$ for some L. But then $N_1 \equiv L \equiv N_2$ by Lemma 4.8. □

4.12. SOME CONSEQUENCES. (i) The λ-calculus is consistent, i.e. $\boldsymbol{\lambda} \nvdash \textsf{true} = \textsf{false}$. Otherwise $\textsf{true} =_\beta \textsf{false}$ by Proposition 4.5, which is impossible by Corollary 4.11 since **true** and **false** are distinct β-nf's. This is a syntactic consistency proof.

(ii) $\boldsymbol{\Omega}$ has no β-nf. Otherwise $\boldsymbol{\Omega} \twoheadrightarrow_\beta N$ with N in β-nf. But $\boldsymbol{\Omega}$ only reduces to itself and is not in β-nf.

(iii) In order to find the β-nf of a term M (if it exists), the various subexpressions of M may be reduced in different orders. By Corollary 4.11 (ii) the β-nf is unique.

The proof of the Church-Rosser theorem occupies 4.13–4.19. The idea of the proof is as follows. In order to prove Theorem 4.9, it is sufficient to show the Strip Lemma:



In order to prove this lemma, let $M \to_\beta N_1$ be a one step reduction resulting from changing a redex R in M in its contractum R' in N_1. If one makes a

bookkeeping of what happens with R during the reduction $M \twoheadrightarrow_\beta N_2$, then by reducing all 'residuals' of R in N_2 the term N_3 can be found. In order to do the necessary bookkeeping an extended set $\underline{\Lambda} \supseteq \Lambda$ and reduction $\underline{\beta}$ is introduced. The underlining serves as a 'tracing isotope'.

4.13. DEFINITION (Underlining). (i) $\underline{\Lambda}$ is the set of terms defined inductively as follows.

$$\begin{array}{rcl} x \in V & \Rightarrow & x \in \underline{\Lambda}, \\ M, N \in \underline{\Lambda} & \Rightarrow & (MN) \in \underline{\Lambda}, \\ M \in \underline{\Lambda}, x \in V & \Rightarrow & (\lambda x.M) \in \underline{\Lambda}, \\ M, N \in \underline{\Lambda}, x \in V & \Rightarrow & ((\underline{\lambda} x.M)N) \in \underline{\Lambda}. \end{array}$$

(ii) The underlined reduction relations $\rightarrow_{\underline{\beta}}$ (one step) and $\twoheadrightarrow_{\underline{\beta}}$ are defined starting with the contraction rules

$$\begin{array}{rcl} (\lambda x.M)N & \rightarrow_{\underline{\beta}} & M[x := N], \\ (\underline{\lambda} x.M)N & \rightarrow_{\underline{\beta}} & M[x := N]. \end{array}$$

Then $\rightarrow_{\underline{\beta}}$ is extended in order to become a compatible relation (also with respect to $\underline{\lambda}$-abstraction). Moreover, $\twoheadrightarrow_{\underline{\beta}}$ is the transitive reflexive closure of $\rightarrow_{\underline{\beta}}$.

(iii) If $M \in \underline{\Lambda}$, then $|M| \in \Lambda$ is obtained from M by leaving out all underlinings. E.g. $|(\lambda x.x)((\underline{\lambda} x.x)(\lambda x.x))| \equiv \mathsf{I}(\mathsf{II})$.

4.14. DEFINITION. The map $\varphi : \underline{\Lambda} \rightarrow \Lambda$ is defined inductively as follows.

$$\begin{array}{rcl} \varphi(x) & \equiv & x, \\ \varphi(MN) & \equiv & \varphi(M)\varphi(N), \\ \varphi(\lambda x.M) & \equiv & \lambda x.\varphi(M), \\ \varphi((\underline{\lambda} x.M)N) & \equiv & \varphi(M)[x := \varphi(N)]. \end{array}$$

In other words, φ contracts all redexes that are underlined, from the inside to the outside.

NOTATION. If $|M| \equiv N$ or $\varphi(M) \equiv N$, then this will be denoted by

$$M \xrightarrow[|\;|]{} N \text{ or } M \xrightarrow[\varphi]{} N.$$

4.15. LEMMA.

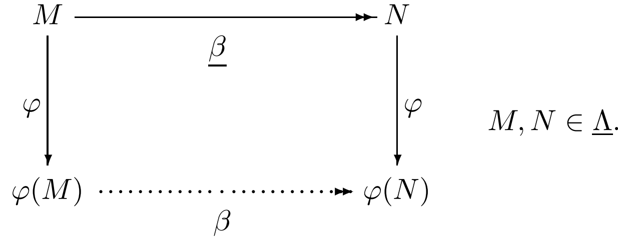
$$\begin{array}{ccc} M' & \overset{\cdots\cdots\cdots\cdots}{\twoheadrightarrow}_{\underline{\beta}} & N' \\ \big\downarrow {\scriptstyle |\,|} & & \big\downarrow {\scriptstyle |\,|} \\ M & \twoheadrightarrow_{\beta} & N \end{array} \qquad \begin{array}{l} M', N' \in \underline{\Lambda}, \\ M, N \in \Lambda. \end{array}$$

PROOF. First suppose $M \to_\beta N$. Then N is obtained by contracting a redex in M and N' can be obtained by contracting the corresponding redex in M'. The general statement follows by transitivity. □

4.16. LEMMA. (i) *Let* $M, N \in \underline{\Lambda}$. *Then*

$$\varphi(M[x := N]) \equiv \varphi(M)[x := \varphi(N)].$$

(ii)

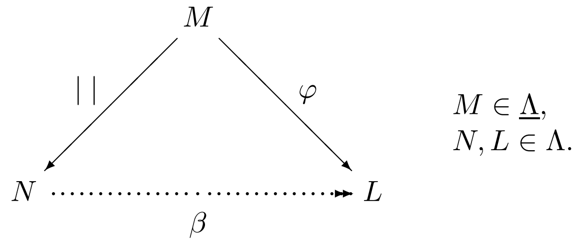


$M, N \in \underline{\Lambda}$.

PROOF. (i) By induction on the structure of M, using the Substitution Lemma (see Exercise 2.2) in case $M \equiv (\underline{\lambda} y.P)Q$. The condition of that lemma may be assumed to hold by our convention about free variables.

(ii) By induction on the generation of $\twoheadrightarrow_{\underline{\beta}}$, using (i). □

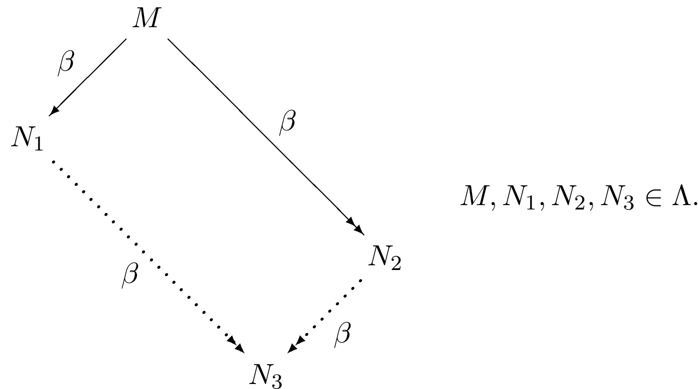
4.17. LEMMA.



$M \in \underline{\Lambda}$,
$N, L \in \Lambda$.

PROOF. By induction on the structure of M. □

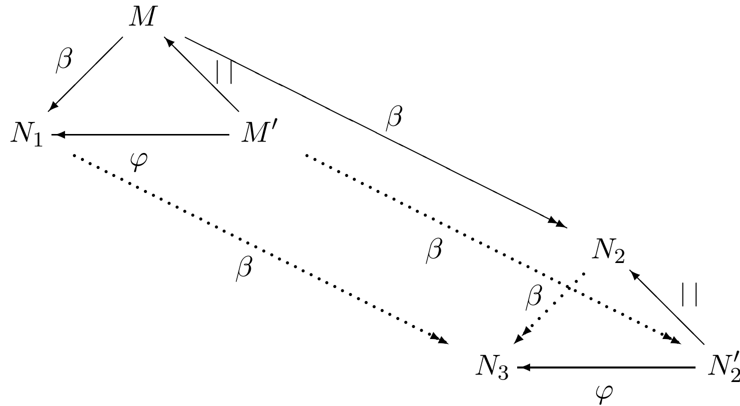
4.18. STRIP LEMMA.



$M, N_1, N_2, N_3 \in \Lambda$.

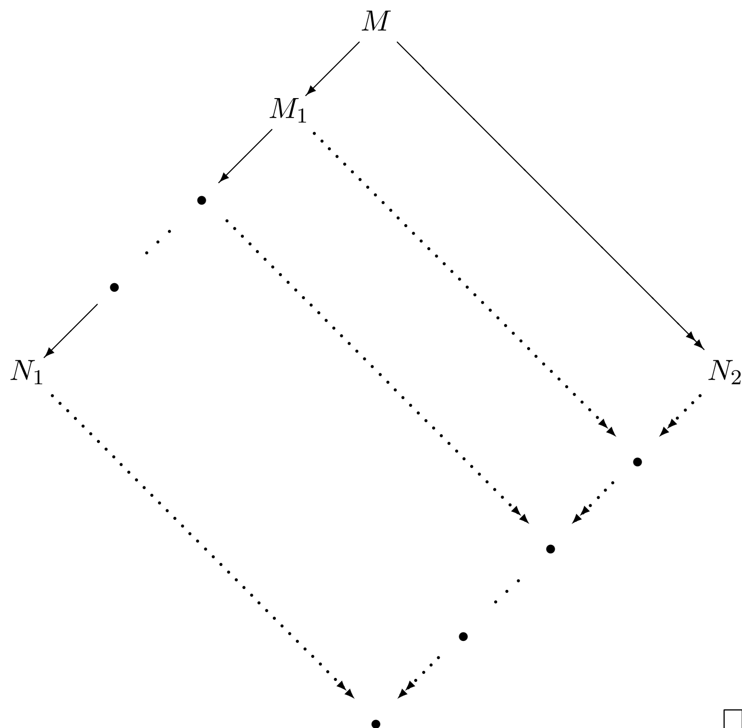
PROOF. Let N_1 be the result of contracting the redex occurrence $R \equiv (\lambda x.P)Q$ in M. Let $M' \in \underline{\Lambda}$ be obtained from M by replacing R by $R' \equiv (\underline{\lambda} x.P)Q$. Then

$|M'| \equiv M$ and $\varphi(M') \equiv N_1$. By the lemmas 4.15, 4.16 and 4.17 we can erect the diagram



which proves the Strip Lemma. □

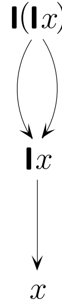
4.19. Proof of the Church-Rosser Theorem. If $M \twoheadrightarrow_\beta N_1$, then $M \equiv M_1 \rightarrow_\beta M_2 \rightarrow_\beta \cdots \rightarrow_\beta M_n \equiv N_1$. Hence the CR property follows from the Strip Lemma and a simple diagram chase:



□

4.20. Definition. For $M \in \Lambda$ the *reduction graph* of M, notation $G_\beta(M)$, is the directed multigraph with vertices $\{N \mid M \twoheadrightarrow_\beta N\}$ and directed by $\rightarrow_\beta$.

4.21. Example. $G_\beta(\mathsf{I}(\mathsf{I}x))$ is



sometimes simply drawn as

It can happen that a term M has a nf, but at the same time an infinite reduction path. Let $\mathbf{\Omega} \equiv (\lambda x.xx)(\lambda x.xx)$. Then $\mathbf{\Omega} \to \mathbf{\Omega} \to \cdots$ so $\mathsf{K}\mathsf{I}\mathbf{\Omega} \to \mathsf{K}\mathsf{I}\mathbf{\Omega} \to \cdots$, and $\mathsf{K}\mathsf{I}\mathbf{\Omega} \twoheadrightarrow \mathsf{I}$. Therefore a so called *strategy* is necessary in order to find the normal form. We state the following theorem; for a proof see Barendregt (1984), Theorem 13.2.2.

4.22. Normalization Theorem. *If M has a normal form, then iterated contraction of the leftmost redex leads to that normal form.*

In other words: the leftmost reduction strategy is *normalizing*. This fact can be used to find the normal form of a term, or to prove that a certain term has no normal form.

4.23. Example. $\mathsf{K}\mathbf{\Omega}\mathsf{I}$ has an infinite leftmost reduction path, viz.

$$\mathsf{K}\mathbf{\Omega}\mathsf{I} \to_\beta (\lambda y.\mathbf{\Omega})\mathsf{I} \to_\beta \mathbf{\Omega} \to_\beta \mathbf{\Omega} \to_\beta \cdots,$$

and hence does not have a normal form.

The functional language (pure) *Lisp* uses an *eager* or *applicative* evaluation strategy, i.e. whenever an expression of the form FA has to be evaluated, A is reduced to normal form first, before 'calling' F. In the λ-calculus this strategy is not normalizing as is shown by the two reduction paths for $\mathsf{K}\mathsf{I}\mathbf{\Omega}$ above. There is, however, a variant of the lambda calculus, called the λI-calculus, in which the eager evaluation strategy is normalizing. In this λI-calculus terms like K, 'throwing away' $\mathbf{\Omega}$ in the reduction $\mathsf{K}\mathsf{I}\mathbf{\Omega} \twoheadrightarrow \mathsf{I}$ do not exist. The 'ordinary' λ-calculus is sometimes referred to as λK-calculus; see Barendregt (1984), Chapter 9.

Remember the fixedpoint combinator Y. For each $F \in \Lambda$ one has $\mathsf{Y}F =_\beta F(\mathsf{Y}F)$, but neither $\mathsf{Y}F \twoheadrightarrow_\beta F(\mathsf{Y}F)$ nor $F(\mathsf{Y}F) \twoheadrightarrow_\beta \mathsf{Y}F$. In order to solve

reduction equations one can work with A.M. Turing's fixedpoint combinator, which has a different reduction behaviour.

4.24. Definition. Turing's fixedpoint combinator $\mathbf{\Theta}$ is defined by setting

$$\begin{aligned} A &\equiv \lambda xy.y(xxy), \\ \mathbf{\Theta} &\equiv AA. \end{aligned}$$

4.25. Proposition. *For all $F \in \Lambda$ one has*

$$\mathbf{\Theta} F \twoheadrightarrow_\beta F(\mathbf{\Theta} F).$$

Proof.

$$\begin{aligned} \mathbf{\Theta} F &\equiv AAF \\ &\to_\beta (\lambda y.y(AAy))F \\ &\to_\beta F(AAF) \\ &\equiv F(\mathbf{\Theta} F). \ \square \end{aligned}$$

4.26. Example. $\exists G\, \forall X\; GX \twoheadrightarrow X(XG)$. Indeed,

$$\begin{aligned} \forall X\; GX \twoheadrightarrow X(XG) &\Leftarrow G \twoheadrightarrow \lambda x.x(xG) \\ &\Leftarrow G \twoheadrightarrow (\lambda gx.x(xg))G \\ &\Leftarrow G \equiv \mathbf{\Theta}(\lambda gx.x(xg)). \end{aligned}$$

Also the Multiple Fixedpoint Theorem has a 'reducing' variant.

4.27. Theorem. *Let $F_1, \dots, F_n$ be λ-terms. Then we can find $X_1, \dots, X_n$ such that*

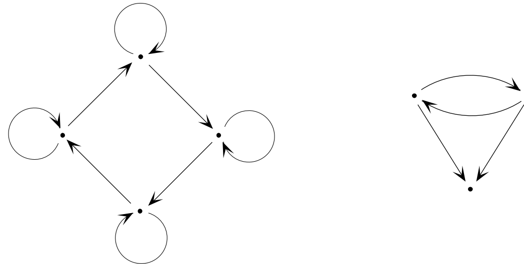
$$\begin{aligned} X_1 &\twoheadrightarrow F_1 X_1 \cdots X_n, \\ &\vdots \\ X_n &\twoheadrightarrow F_n X_1 \cdots X_n. \end{aligned}$$

Proof. As for the equational Multiple Fixedpoint Theorem 3.17, but now using $\mathbf{\Theta}$. $\square$

Exercises

4.1. Show $\forall M\, \exists N\, [N$ in β-nf and $N\mathsf{I} \twoheadrightarrow_\beta M]$.

4.2. Construct four terms M with $G_\beta(M)$ respectively as follows.

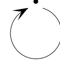


4.3. Show that there is no $F \in \Lambda$ such that for all $M, N \in \Lambda$

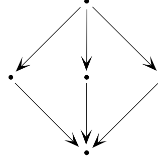
$$F(MN) = M.$$

4.4.* Let $M \equiv AAx$ with $A \equiv \lambda axz.z(aax)$. Show that $G_\beta(M)$ contains as subgraphs an n-dimensional cube for every $n \in \mathbb{N}$.

4.5. (A. Visser)
(i) Show that there is only one redex R such that $G_\beta(R)$ is as follows.



(ii) Show that there is no $M \in \Lambda$ with $G_\beta(M)$ is



[*Hint.* Consider the relative positions of redexes.]

4.6.* (C. Böhm) Examine $G_\beta(M)$ with M equal to
(i) $H\mathsf{I}H$, $H \equiv \lambda xy.x(\lambda z.yzy)x$.
(ii) $LL\mathsf{I}$, $L \equiv \lambda xy.x(yy)x$.
(iii) $Q\mathsf{I}Q$, $Q \equiv \lambda xy.xy\mathsf{I}xy$.

4.7.* (J.W. Klop) Extend the λ-calculus with two constants $\boldsymbol{\delta}$, $\boldsymbol{\varepsilon}$. The reduction rules are extended to include $\boldsymbol{\delta}MM \to \boldsymbol{\varepsilon}$. Show that the resulting system is not Church-Rosser.
[*Hint.* Define terms C, D such that

$$\begin{array}{rcl} Cx & \twoheadrightarrow & \boldsymbol{\delta}x(Cx) \\ D & \twoheadrightarrow & CD \end{array}$$

Then $D \twoheadrightarrow \boldsymbol{\varepsilon}$ and $D \twoheadrightarrow C\boldsymbol{\varepsilon}$ in the extended reduction system, but there is no common reduct.]

4.8. Show that the term $M \equiv AAx$ with $A \equiv \lambda axz.z(aax)$ does not have a normal form.

4.9. (i) Show $\boldsymbol{\lambda} \nvdash WWW = \boldsymbol{\omega}_3\boldsymbol{\omega}_3$, with $W \equiv \lambda xy.xyy$ and $\boldsymbol{\omega}_3 \equiv \lambda x.xxx$.
(ii) Show $\boldsymbol{\lambda} \nvdash B_x = B_y$ with $B_z \equiv A_zA_z$ and $A_z \equiv \lambda p.ppz$.

4.10. Draw $G_\beta(M)$ for M equal to:
(i) WWW, $W \equiv \lambda xy.xyy$.
(ii) $\boldsymbol{\omega\omega}$, $\boldsymbol{\omega} \equiv \lambda x.xx$.
(iii) $\boldsymbol{\omega}_3\boldsymbol{\omega}_3$, $\boldsymbol{\omega}_3 \equiv \lambda x.xxx$.
(iv) $(\lambda x.\mathsf{I}xx)(\lambda x.\mathsf{I}xx)$.
(v) $(\lambda x.\mathsf{I}(xx))(\lambda x.\mathsf{I}(xx))$.
(vi) $\mathsf{II}(\mathsf{III})$.

4.11. The *length* of a term is its number of symbols times 0.5 cm. Write down a λ-term of length < 30 cm with normal form $> 10^{10^{10}}$ light year.
[*Hint.* Use Proposition 2.15 (ii). The speed of light is $c = 3 \times 10^{10}$ cm/s.]

Chapter 5

Type Assignment

The lambda calculus as treated so far is usually referred to as a *type-free* theory. This is so, because every expression (considered as a function) may be applied to every other expression (considered as an argument). For example, the identity function $\mathsf{I} \equiv \lambda x.x$ may be applied to any argument x to give as result that same x. In particular I may be applied to itself.

There are also typed versions of the lambda calculus. These are introduced essentially in Curry (1934) (for the so called Combinatory Logic, a variant of the lambda calculus) and in Church (1940). Types are usually objects of a syntactic nature and may be assigned to lambda terms. If M is such a term and a type A is assigned to M, then we say 'M *has type* A' or 'M *in* A'; the denotation used for this is $M : A$. For example in some typed systems one has $\mathsf{I} : (A{\rightarrow}A)$, that is, the identity I may get as type $A{\rightarrow}A$. This means that if x being an argument of I is of type A, then also the value $\mathsf{I}x$ is of type A. In general, $A{\rightarrow}B$ is the type of functions from A to B.

Although the analogy is not perfect, the type assigned to a term may be compared to the dimension of a physical entity. These dimensions prevent us from wrong operations like adding 3 volt to 2 ampère. In a similar way types assigned to lambda terms provide a partial specification of the algorithms that are represented and are useful for showing partial correctness.

Types may also be used to improve the efficiency of compilation of terms representing functional algorithms. If for example it is known (by looking at types) that a subexpression of a term (representing a funtional program) is purely arithmetical, then fast evaluation is possible. This is because the expression then can be executed by the ALU of the machine and not in the slower way in which symbolic expressions are evaluated in general.

The two original papers of Curry and Church introducing typed versions of the lambda calculus give rise to two different families of systems. In the typed lambda calculi *à la* Curry terms are those of the type-free theory. Each term has a set of possible types. This set may be empty, be a singleton or consist of several (possibly infinitely many) elements. In the systems *à la* Church the terms are annotated versions of the type-free terms. Each term has (up to an equivalence relation) a unique type that is usually derivable from the way the term is annotated.

The Curry and Church approaches to typed lambda calculus correspond to

two paradigms in programming. In the first of these a program may be written without typing at all. Then a compiler should check whether a type can be assigned to the program. This will be the case if the program is correct. A well-known example of such a language is *ML*, see Milner (1984). The style of typing is called *implicit typing*. The other paradigm in programming is called *explicit typing* and corresponds to the Church version of typed lambda calculi. Here a program should be written together with its type. For these languages type-checking is usually easier, since no types have to be constructed. Examples of such languages are *Algol 68* and *Pascal*. Some authors designate the Curry systems as 'lambda calculi *with type assignment*' and the Church systems as 'systems of *typed* lambda calculus'.

Within each of the two paradigms there are several versions of typed lambda calculus. In many important systems, especially those *à la* Church, it is the case that terms that do have a type always possess a normal form. By the unsolvability of the halting problem this implies that not all computable functions can be represented by a typed term, see Barendregt (1990), Theorem 4.2.15. This is not so bad as it sounds, because in order to find such computable functions that cannot be represented, one has to stand on one's head. For example in $\lambda 2$, the second order typed lambda calculus, only those partial recursive functions cannot be represented that happen to be total, but not provably so in mathematical analysis (second order arithmetic).

Considering terms and types as programs and their specifications is not the only possibility. A type A can also be viewed as a proposition and a term M in A as a proof of this proposition. This so called propositions-as-types interpretation is independently due to de Bruijn (1970) and Howard (1980) (both papers were conceived in 1968). Hints in this direction were given in Curry and Feys (1958) and in Läuchli (1970). Several systems of proof checking are based on this interpretation of propositions-as-types and of proofs-as-terms. See e.g. de Bruijn (1980) for a survey of the so called AUTOMATH proof checking system. Normalization of terms corresponds in the formulas-as-types interpretation to normalisation of proofs in the sense of Prawitz (1965). Normal proofs often give useful proof theoretic information, see e.g. Schwichtenberg (1977).

In this section a typed lambda calculus will be introduced in the style of Curry. For more information, see Barendregt (1992).

The system $\lambda\!\to$-Curry

Originally the implicit typing paradigm was introduced in Curry (1934) for the theory of combinators. In Curry and Feys (1958) and Curry et al. (1972) the theory was modified in a natural way to the lambda calculus assigning elements of a given set $\mathbb{T}$ of types to type free lambda terms. For this reason these calculi *à la* Curry are sometimes called *systems of type assignment*. If the type $\sigma \in \mathbb{T}$ is assigned to the term $M \in \Lambda$ one writes $\vdash M : \sigma$, sometimes with a subscript under $\vdash$ to denote the particular system. Usually a set of assumptions Γ is needed to derive a type assignment and one writes $\Gamma \vdash M : \sigma$ (pronounce this as 'Γ yields M in σ'). A particular Curry type assignment system depends on two parameters, the set $\mathbb{T}$ and the rules of type assignment. As an example we

now introduce the system $\lambda\!\to$-Curry.

5.1. DEFINITION. The set of *types* of $\lambda\!\to$, notation Type($\lambda\!\to$), is inductively defined as follows. We write $\mathbb{T} = \text{Type}(\lambda\!\to)$. Let $\mathbb{V} = \{\alpha, \alpha', \ldots\}$ be a set of *type variables*. It will be convenient to allow *type constants* for basic types such as Nat, Bool. Let $\mathbb{B}$ be such a collection. Then

$$\begin{array}{rcl} \alpha \in \mathbb{V} & \Rightarrow & \alpha \in \mathbb{T}, \\ \mathsf{B} \in \mathbb{B} & \Rightarrow & \mathsf{B} \in \mathbb{T}, \\ \sigma, \tau \in \mathbb{T} & \Rightarrow & (\sigma\!\to\!\tau) \in \mathbb{T} \quad \text{(function space types).} \end{array}$$

For such definitions it is convenient to use the following abstract syntax to form $\mathbb{T}$.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T}\!\to\!\mathbb{T}$$

with

$$\mathbb{V} = \alpha \mid \mathbb{V}' \qquad \text{(type variables).}$$

NOTATION. (i) If $\sigma_1, \ldots, \sigma_n \in \mathbb{T}$ then

$$\sigma_1\!\to\!\sigma_2\!\to\!\cdots\!\to\!\sigma_n$$

stands for

$$(\sigma_1\!\to\!(\sigma_2\!\to\!\cdots\!\to\!(\sigma_{n-1}\!\to\!\sigma_n)\cdot\cdot));$$

that is, we use association to the right.

(ii) $\alpha, \beta, \gamma, \ldots$ denote arbitrary type variables.

5.2. DEFINITION. (i) A *statement* is of the form $M : \sigma$ with $M \in \Lambda$ and $\sigma \in \mathbb{T}$. This statement is pronounced as 'M in σ'. The type σ is the *predicate* and the term M is the *subject* of the statement.

(ii) A *basis* is a set of statements with only distinct (term) variables as subjects.

5.3. DEFINITION. Type *derivations* in the system $\lambda\!\to$ are built up from assumptions $x{:}\sigma$, using the following inference rules.

$$\frac{M : \sigma\!\to\!\tau \qquad N : \sigma}{MN : \tau} \qquad\qquad \frac{\begin{array}{c}\not{x : \sigma}\\ \vdots \\ M : \tau\end{array}}{\lambda x.M : \sigma\!\to\!\tau}$$

5.4. DEFINITION. (i) A statement $M : \sigma$ is *derivable from* a basis Γ, notation

$$\Gamma \vdash M : \sigma$$

(or $\Gamma \vdash_{\lambda\to} M : \sigma$ if we wish to stress the typing system) if there is a derivation of $M : \sigma$ in which all non-cancelled assumptions are in Γ.

(ii) We use $\vdash M : \sigma$ as shorthand for $\emptyset \vdash M : \sigma$.

5.5. EXAMPLE. (i) Let $\sigma \in \mathbb{T}$. Then $\vdash \lambda fx.f(fx) : (\sigma\to\sigma)\to\sigma\to\sigma$, which is shown by the following derivation.

$$\dfrac{\dfrac{\cancel{f:\sigma\to\sigma}^{(2)} \qquad \dfrac{\cancel{f:\sigma\to\sigma}^{(2)} \qquad \cancel{x:\sigma}^{(1)}}{fx:\sigma}}{\dfrac{f(fx):\sigma}{\lambda x.f(fx):\sigma\to\sigma}\,(1)}}{\lambda fx.f(fx):(\sigma\to\sigma)\to\sigma\to\sigma}\,(2)$$

The indices (1) and (2) are bookkeeping devices that indicate at which application of a rule a particular assumption is being cancelled.

(ii) One has $\vdash \mathbf{K} : \sigma\to\tau\to\sigma$ for any $\sigma, \tau \in \mathbb{T}$, which is demonstrated as follows.

$$\dfrac{\dfrac{\cancel{x:\sigma}^{(1)}}{\lambda y.x:\tau\to\sigma}}{\lambda xy.x:\sigma\to\tau\to\sigma}\,(1)$$

(iii) Similarly one can show for all $\sigma \in \mathbb{T}$

$$\vdash \mathbf{I} : \sigma\to\sigma.$$

(iv) An example with a non-empty basis is the statement

$$y{:}\sigma \vdash \mathbf{I}y : \sigma.$$

Properties of $\lambda\to$

Several properties of type assignment in $\lambda\to$ are valid. The first one analyses how much of a basis is necessary in order to derive a type assignment.

5.6. DEFINITION. Let $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ be a basis.

(i) Write $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\sigma_i = \Gamma(x_i)$. That is, Γ is considered as a partial function.

(ii) Let V_0 be a set of variables. Then $\Gamma \restriction V_0 = \{x{:}\sigma \mid x \in V_0 \,\&\, \sigma = \Gamma(x)\}$.

(iii) For $\sigma, \tau \in \mathbb{T}$ substitution of τ for α in σ is denoted by $\sigma[\alpha := \tau]$.

5.7. BASIS LEMMA. *Let Γ be a basis.*

(i) *If $\Gamma' \supseteq \Gamma$ is another basis, then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma' \vdash M : \sigma.$$

(ii) $\Gamma \vdash M : \sigma \;\Rightarrow\; \mathrm{FV}(M) \subseteq \mathrm{dom}(\Gamma)$.

(iii) $\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \restriction \mathrm{FV}(M) \vdash M : \sigma$.

PROOF. (i) By induction on the derivation of $M : \sigma$. Since such proofs will occur frequently we will spell it out in this simple situation in order to be shorter later on.

Case 1. $M : \sigma$ is $x{:}\sigma$ and is element of Γ. Then also $x{:}\sigma \in \Gamma'$ and hence $\Gamma' \vdash M : \sigma$.

Case 2. $M : \sigma$ is $(M_1 M_2) : \sigma$ and follows directly from $M_1 : (\tau{\to}\sigma)$ and $M_2 : \tau$ for some τ. By the IH one has $\Gamma' \vdash M_1 : (\tau{\to}\sigma)$ and $\Gamma' \vdash M_2 : \tau$. Hence $\Gamma' \vdash (M_1 M_2) : \sigma$.

Case 3. $M : \sigma$ is $(\lambda x.M_1) : (\sigma_1{\to}\sigma_2)$ and follows directly from $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$. By the variable convention it may be assumed that the bound variable x does not occur in $\mathrm{dom}(\Gamma')$. Then $\Gamma', x{:}\sigma_1$ is also a basis which extends $\Gamma, x{:}\sigma_1$. Therefore by the IH one has $\Gamma', x{:}\sigma_1 \vdash M_1 : \sigma_2$ and so $\Gamma' \vdash (\lambda x.M_1) : (\sigma_1{\to}\sigma_2)$.

(ii) By induction on the derivation of $M : \sigma$. We only treat the case that $M : \sigma$ is $(\lambda x.M_1) : (\sigma_1{\to}\sigma_2)$ and follows directly from $\Gamma, x{:}\sigma_1 \vdash M_1 : \sigma_2$. Let $y \in \mathrm{FV}(\lambda x.M_1)$, then $y \in \mathrm{FV}(M_1)$ and $y \not\equiv x$. By the IH one has $y \in \mathrm{dom}(\Gamma, x{:}\sigma_1)$ and therefore $y \in \mathrm{dom}(\Gamma)$.

(iii) By induction on the derivation of $M : \sigma$. We only treat the case that $M : \sigma$ is $(M_1 M_2) : \sigma$ and follows directly from $M_1 : (\tau{\to}\sigma)$ and $M_2 : \tau$ for some τ. By the IH one has $\Gamma \restriction \mathrm{FV}(M_1) \vdash M_1 : (\tau{\to}\sigma)$ and $\Gamma \restriction \mathrm{FV}(M_2) \vdash M_2 : \tau$. By (i) it follows that $\Gamma \restriction \mathrm{FV}(M_1 M_2) \vdash M_1 : (\tau{\to}\sigma)$and $\Gamma \restriction \mathrm{FV}(M_1 M_2) \vdash M_2 : \tau$ and hence $\Gamma \restriction \mathrm{FV}(M_1 M_2) \vdash (M_1 M_2) : \sigma$. □

The second property analyses how terms of a certain form get typed. It is useful among other things to show that certain terms have no types.

5.8. Generation Lemma. (i) $\Gamma \vdash x : \sigma \;\Rightarrow\; (x{:}\sigma) \in \Gamma$.

(ii) $\Gamma \vdash MN : \tau \;\Rightarrow\; \exists\sigma\,[\Gamma \vdash M : (\sigma{\to}\tau)\,\&\,\Gamma \vdash N : \sigma]$.

(iii) $\Gamma \vdash \lambda x.M : \rho \;\Rightarrow\; \exists\sigma,\tau\,[\Gamma, x{:}\sigma \vdash M : \tau\,\&\,\rho \equiv (\sigma{\to}\tau)]$.

Proof. By induction on the structure of derivations. □

5.9. Proposition (Typability of subterms). *Let M' be a subterm of M. Then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma' \vdash M' : \sigma' \quad \textit{for some } \Gamma' \textit{ and } \sigma'.$$

The moral is: if M has a type, i.e. $\Gamma \vdash M : \sigma$ for some Γ and σ, then every subterm has a type as well.

Proof. By induction on the generation of M. □

5.10. Substitution Lemma.

(i) $\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$.

(ii) *Suppose* $\Gamma, x{:}\sigma \vdash M : \tau$ *and* $\Gamma \vdash N : \sigma$. *Then* $\Gamma \vdash M[x := N] : \tau$.

Proof. (i) By induction on the derivation of $M : \sigma$.

(ii) By induction on the derivation showing $\Gamma, x{:}\sigma \vdash M : \tau$. □

The following result states that the set of $M \in \Lambda$ having a certain type in $\lambda{\to}$ is closed under reduction.

5.11. Subject Reduction Theorem. *Suppose* $M \twoheadrightarrow_\beta M'$. *Then*

$$\Gamma \vdash M : \sigma \;\Rightarrow\; \Gamma \vdash M' : \sigma.$$

PROOF. Induction on the generation of $\twoheadrightarrow_\beta$ using the Generation Lemma 5.8 and the Substitution Lemma 5.10. We treat the prime case, namely that $M \equiv (\lambda x.P)Q$ and $M' \equiv P[x := Q]$. Well, if

$$\Gamma \vdash (\lambda x.P)Q : \sigma$$

then it follows by the Generation Lemma that for some τ one has

$$\Gamma \vdash (\lambda x.P) : (\tau \to \sigma) \text{ and } \Gamma \vdash Q : \tau.$$

Hence once more by the Generation Lemma

$$\Gamma, x{:}\tau \vdash P : \sigma \text{ and } \Gamma \vdash Q : \tau$$

and therefore by the Substitution Lemma

$$\Gamma \vdash P[x := Q] : \sigma. \ \square$$

Terms having a type are not closed under expansion. For example,

$$\vdash \mathsf{I} : (\sigma \to \sigma), \text{ but } \not\vdash \mathsf{KI}\ (\lambda x.xx) : (\sigma \to \sigma).$$

See Exercise 5.1. One even has the following stronger failure of subject expansion, as is observed in van Bakel (1992).

5.12. OBSERVATION. There are $M, M' \in \Lambda$ and $\sigma, \sigma' \in \mathbb{T}$ such that $M' \twoheadrightarrow_\beta M$ and

$$\vdash M : \sigma, \qquad \vdash M' : \sigma',$$

but

$$\not\vdash M' : \sigma.$$

PROOF. Take $M \equiv \lambda xy.y$, $M' \equiv \mathsf{SK}$, $\sigma \equiv \alpha \to (\beta \to \beta)$ and $\sigma' \equiv (\beta \to \alpha) \to (\beta \to \beta)$; do Exercise 5.1. $\square$

All typable terms have a normal form. In fact, the so-called *strong normalization* property holds: if M is a typable term, then all reductions starting from M are finite.

Decidability of type assignment

For the system of type assignment several questions may be asked. Note that for $\Gamma = \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ one has

$$\Gamma \vdash M : \sigma \ \Leftrightarrow \ \vdash (\lambda x_1{:}\sigma_1 \cdots \lambda x_n{:}\sigma_n.M) : (\sigma_1 \to \cdots \to \sigma_n \to \sigma),$$

therefore in the following one has taken $\Gamma = \emptyset$. Typical questions are

(1) Given M and σ, does one have $\vdash M : \sigma$?
(2) Given M, does there exist a σ such that $\vdash M : \sigma$?
(3) Given σ, does there exist an M such that $\vdash M : \sigma$?

These three problems are called *type checking*, *typability* and *inhabitation* respectively and are denoted by $M : \sigma?$, $M : ?$ and $? : \sigma$.

Type checking and typability are decidable. This can be shown using the following result, independently due to Curry (1969), Hindley (1969), and Milner (1978).

5.13. THEOREM. (i) *It is decidable whether a term is typable in* $\lambda\!\rightarrow$.

(ii) *If a term* M *is typable in* $\lambda\!\rightarrow$, *then* M *has a principal type scheme, i.e. a type* σ *such that every possible type for* M *is a substitution instance of* σ. *Moreover* σ *is computable from* M.

5.14. COROLLARY. *Type checking for* $\lambda\!\rightarrow$ *is decidable.*

PROOF. In order to check $M : \tau$ it suffices to verify that M is typable and that τ is an instance of the principal type of M. □

For example, a principal type scheme of **K** is $\alpha\to\beta\to\alpha$.

Polymorphism

Note that in $\lambda\!\rightarrow$ one has

$$\vdash \mathbf{I} : \sigma\to\sigma \qquad \text{for all } \sigma \in \mathbb{T}.$$

In the polymorphic lambda calculus this quantification can be internalized by stating

$$\vdash \mathbf{I} : \forall\alpha.\alpha\to\alpha.$$

The resulting system is the *polymorphic* of *second-order* lambda calculus due to Girard (1972) and Reynolds (1974).

5.15. DEFINITION. The set of *types* of $\lambda 2$ (notation $\mathbb{T} = \mathrm{Type}(\lambda 2)$) is specified by the syntax

$$\mathbb{T} = \mathbb{V} \mid \mathbb{B} \mid \mathbb{T}\to\mathbb{T} \mid \forall\mathbb{V}.\mathbb{T}.$$

5.16. DEFINITION. The rules of type assignment are those of $\lambda\!\rightarrow$, plus

$$\frac{M : \forall\alpha.\sigma}{M : \sigma[\alpha := \tau]} \qquad\qquad \frac{M : \sigma}{M : \forall\alpha.\sigma}$$

In the latter rule, the type variable α may not occur free in any assumption on which the premiss $M : \sigma$ depends.

5.17. EXAMPLE. (i) $\vdash \mathbf{I} : \forall\alpha.\alpha\to\alpha$.

(ii) Define $\mathrm{Nat} \equiv \forall\alpha.(\alpha\to\alpha)\to\alpha\to\alpha$. Then for the Church numerals $\boldsymbol{c}_n \equiv \lambda fx.f^n(x)$ we have $\vdash \boldsymbol{c}_n : \mathrm{Nat}$.

The following is due to Girard (1972).

5.18. THEOREM. (i) *The Subject Reduction property holds for* $\lambda 2$.

(ii) $\lambda 2$ *is strongly normalizing.*

Typability in $\lambda 2$ is *not* decidable; see Wells (1994).

Exercises

5.1. (i) Give a derivation of

$$\vdash \mathbf{SK} : (\alpha{\to}\beta){\to}(\alpha{\to}\alpha).$$

(ii) Give a derivation of

$$\vdash \mathbf{KI} : \beta{\to}(\alpha{\to}\alpha).$$

(iii) Show that $\nvdash \mathbf{SK} : (\alpha{\to}\beta{\to}\beta)$.

(iv) Find a common β-reduct of $\mathbf{SK}$ and $\mathbf{KI}$. What is the most general type for this term?

5.2. Show that $\lambda x.xx$ and $\mathbf{KI}(\lambda x.xx)$ have no type in $\lambda{\to}$.

5.3. Find the most general types (if they exist) for the following terms.

(i) $\lambda xy.xyy$.

(ii) $\mathbf{SII}$.

(iii) $\lambda xy.y(\lambda z.z(yx))$.

5.4. Find terms $M, N \in \Lambda$ such that the following hold in $\lambda{\to}$.

(i) $\vdash M : (\alpha{\to}\beta){\to}(\beta{\to}\gamma){\to}(\alpha{\to}\gamma)$.

(ii) $\vdash N : (((\alpha{\to}\beta){\to}\beta){\to}\beta){\to}(\alpha{\to}\beta)$.

5.5. Find types in $\lambda 2$ for the terms in the exercises 5.2 and 5.3.

Chapter 6

Extensions

In Chapter 3 we have seen that all computable functions can be expressed in the lambda calculus. For reasons of efficiency, reliability and convenience this language will be extended. The set of λ-terms Λ will be extended with constants. Some of the constants are selected to represent primitive data (such as numbers) and operations on these (such as addition). Some new reduction rules (the so called δ-rules) are introduced to express the operational semantics of these operations. Even if these constants and operations can be implemented in the lambda calculus, it is worthwhile to have primitive symbols for them. The reason is that in an implementation of the lambda calculus addition of the Church numerals runs less efficient than the usual implementation in hardware of addition of binary represented numbers. Having numerals and addition as primitives therefore creates the possibility to interprete these efficiently.

From now on we allow constants in λ-terms. Let $\mathbb{C}$ be a set of constants.

6.1. Definition. The set of *lambda terms with constants*, notation $\Lambda(\mathbb{C})$, is defined inductively as follows.

$$\begin{array}{rcl} \boldsymbol{C} \in \mathbb{C} & \Rightarrow & \boldsymbol{C} \in \Lambda(\mathbb{C}), \\ x \in V & \Rightarrow & x \in \Lambda(\mathbb{C}), \\ M, N \in \Lambda(\mathbb{C}) & \Rightarrow & (MN) \in \Lambda(\mathbb{C}), \\ M \in \Lambda(\mathbb{C}), x \in V & \Rightarrow & (\lambda x.M) \in \Lambda(\mathbb{C}). \end{array}$$

This definition given as an abstract syntax is as follows.

$$\Lambda(\mathbb{C}) ::= \mathbb{C} \mid V \mid \Lambda(\mathbb{C})\,\Lambda(\mathbb{C}) \mid \lambda V\,\Lambda(\mathbb{C}).$$

6.2. Definition (δ-reduction). Let $X \subseteq \Lambda(\mathbb{C})$ be a set of closed normal forms. Usually we take $X \subseteq \mathbb{C}$. Let $f : X^k \to \Lambda$ be an 'externally defined' function. In order to represent f, a so-called δ-rule may be added to the λ-calculus. This is done as follows.

(1) A special constant in $\mathbb{C}$ is selected and is given some name, say $\boldsymbol{\delta}$ ($= \boldsymbol{\delta}_f$).
(2) The following contraction rules are added to those of the λ-calculus:

$$\boldsymbol{\delta} M_1 \cdots M_k \to f(M_1, \ldots, M_k),$$

for $M_1, \ldots, M_k \in X$.

Note that for a given function f this is not one contraction rule but in fact a rule *schema*. The resulting extension of the λ-calculus is called $\lambda\delta$. The corresponding notion of (one step) reduction is denoted by ($\to_{\beta\delta}$) $\twoheadrightarrow_{\beta\delta}$.

So δ-reduction is not an absolute notion, but depends on the choice of f.

6.3. Theorem (G. Mitschke). *Let f be a function on closed normal forms. Then the resulting notion of reduction $\twoheadrightarrow_{\beta\delta}$ satisfies the Church-Rosser property.*

Proof. Follows from Theorem 15.3.3 in Barendregt (1984). □

The notion of normal form generalises to $\beta\delta$-normal form. So does the concept of leftmost reduction. The $\beta\delta$-normalforms can be found by a leftmost reduction (notation $\twoheadrightarrow_{\ell\beta\delta}$).

6.4. Theorem. *If $M \twoheadrightarrow_{\beta\delta} N$ and N is in $\beta\delta$-nf, then $M \twoheadrightarrow_{\ell\beta\delta} N$.*

Proof. Analogous to the proof of the theorem for β-normal forms (4.22). □

6.5. Example. One of the first versions of a δ-rule is in Church (1941). Here X is the set of all closed normal forms and for $M, N \in X$ we have

$$\begin{array}{rcll} \boldsymbol{\delta}_{\mathrm{C}} MN & \to & \mathsf{true}, & \text{if } M \equiv N; \\ \boldsymbol{\delta}_{\mathrm{C}} MN & \to & \mathsf{false}, & \text{if } M \not\equiv N. \end{array}$$

Another possible set of δ-rules is for the Booleans.

6.6. Example. The following constants are selected in $\mathbb{C}$.

$$\boldsymbol{true}, \boldsymbol{false}, \boldsymbol{not}, \boldsymbol{and}, \boldsymbol{ite} \text{ (for } \textit{if then else}\text{)}.$$

The following δ-rules are introduced.

$$\begin{array}{rcll} \boldsymbol{not\ true} & \to & \boldsymbol{false}; & \\ \boldsymbol{not\ false} & \to & \boldsymbol{true}; & \\ \boldsymbol{and\ true\ true} & \to & \boldsymbol{true}; & \\ \boldsymbol{and\ true\ false} & \to & \boldsymbol{false}; & \\ \boldsymbol{and\ false\ true} & \to & \boldsymbol{false}; & \\ \boldsymbol{and\ false\ false} & \to & \boldsymbol{false}; & \\ \boldsymbol{ite\ true} & \to & \mathsf{true} & (\equiv \lambda xy.x); \\ \boldsymbol{ite\ false} & \to & \mathsf{false} & (\equiv \lambda xy.x). \end{array}$$

It follows that

$$\begin{array}{rcl} \boldsymbol{ite\ true}\, x\, y & \twoheadrightarrow & x, \\ \boldsymbol{ite\ false}\, x\, y & \twoheadrightarrow & y. \end{array}$$

Now we introduce as δ-rules some operations on the set of integers

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

6.7. EXAMPLE. For each $n \in \mathbb{Z}$ a constant in $\mathbb{C}$ is selected and given the name $\boldsymbol{n}$. (We will express this as follows: for each $n \in \mathbb{Z}$ a constant $\boldsymbol{n} \in \mathbb{C}$ is chosen.) Moreover the following constants in $\mathbb{C}$ are selected:

$$\boldsymbol{plus}, \boldsymbol{minus}, \boldsymbol{times}, \boldsymbol{divide}, \boldsymbol{equal}, \boldsymbol{error}.$$

Then we introduce the following δ-rules (schemes). For $m, n \in \mathbb{Z}$

$$\begin{array}{rcll}
\boldsymbol{plus}\,\boldsymbol{n}\,\boldsymbol{m} & \to & \boldsymbol{n+m}; & \\
\boldsymbol{minus}\,\boldsymbol{n}\,\boldsymbol{m} & \to & \boldsymbol{n-m}; & \\
\boldsymbol{times}\,\boldsymbol{n}\,\boldsymbol{m} & \to & \boldsymbol{n*m}; & \\
\boldsymbol{divide}\,\boldsymbol{n}\,\boldsymbol{m} & \to & \boldsymbol{n \div m}, & \text{if } m \neq 0; \\
\boldsymbol{divide}\,\boldsymbol{n}\,\boldsymbol{0} & \to & \boldsymbol{error}; & \\
\boldsymbol{equal}\,\boldsymbol{n}\,\boldsymbol{n} & \to & \boldsymbol{true}; & \\
\boldsymbol{equal}\,\boldsymbol{n}\,\boldsymbol{m} & \to & \boldsymbol{false}, & \text{if } n \neq m.
\end{array}$$

We may add rules like

$$\boldsymbol{plus}\,\boldsymbol{n}\,\boldsymbol{error} \to \boldsymbol{error}.$$

Similar δ-rules can be introduced for the set of reals.

Again another set of δ-rules is concerned with characters.

6.8. EXAMPLE. Let Σ be some linearly ordered alphabet. For each symbol $s \in \Sigma$ we choose a constant 's' $\in \mathbb{C}$. Moreover we choose two constants $\boldsymbol{\delta}_{\leq}$ and $\boldsymbol{\delta}_{=}$ in $\mathbb{C}$ and formulate the following δ-rules.

$$\begin{array}{rcll}
\boldsymbol{\delta}_{\leq}`s_1\text{'}`s_2\text{'} & \to & \boldsymbol{true}, & \text{if } s_1 \text{ precedes } s_2 \text{ in the ordering of } \Sigma; \\
\boldsymbol{\delta}_{\leq}`s_1\text{'}`s_2\text{'} & \to & \boldsymbol{false}, & \text{otherwise.} \\
\boldsymbol{\delta}_{=}`s_1\text{'}`s_2\text{'} & \to & \boldsymbol{true}, & \text{if } s_1 = s_2; \\
\boldsymbol{\delta}_{=}`s_1\text{'}`s_2\text{'} & \to & \boldsymbol{false}, & \text{otherwise.}
\end{array}$$

It is also possible to represent 'multiple valued' functions F by putting as δ-rule

$$\boldsymbol{\delta n} \to \boldsymbol{m}, \quad \text{provided that } F(n) = m.$$

Of course the resulting $\lambda\delta$-calculus does not satisfy the Church-Rosser theorem and can be used to deal with non-deterministic computations. We will not pursue this possibility, however.

We can extend the type assignment system $\lambda{\to}$ to deal with constants by adding typing *axioms* of the form

$$\boldsymbol{C} : \sigma.$$

For the system with integers this would result in the following. Let $\mathsf{Z}, \mathsf{B} \in \mathbb{B}$ be basic type constants (with intended interpretation $\mathbb{Z}$ and booleans, respectively). Then one adds the following typing axioms to $\lambda\!\to$.

$$\begin{gathered}
\boldsymbol{true} : \mathsf{B}, \quad \boldsymbol{false} : \mathsf{B}, \\
\boldsymbol{not} : \mathsf{B}\to\mathsf{B}, \quad \boldsymbol{and} : \mathsf{B}\to\mathsf{B}\to\mathsf{B}, \\
\boldsymbol{n} : \mathsf{Z}, \quad \boldsymbol{error} : \mathsf{Z}, \\
\boldsymbol{plus} : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z}, \quad \boldsymbol{minus} : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z}, \quad \boldsymbol{times} : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z}, \quad \boldsymbol{divide} : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z}, \\
\boldsymbol{equal} : \mathsf{Z}\to\mathsf{Z}\to\mathsf{B}.
\end{gathered}$$

6.9. EXAMPLE. $\vdash \lambda xy.\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z}$, as is shown by the following derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\boldsymbol{times} : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z} \qquad \cancel{x : \mathsf{Z}}^{(2)}}{\boldsymbol{times}\, x : \mathsf{Z}\to\mathsf{Z}} \qquad \dfrac{\dfrac{\boldsymbol{plus} : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z} \qquad \cancel{x : \mathsf{Z}}^{(2)}}{\boldsymbol{plus}\, x : \mathsf{Z}\to\mathsf{Z}} \qquad \cancel{y : \mathsf{Z}}^{(1)}}{\boldsymbol{plus}\, xy : \mathsf{Z}}}{\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}}}{\lambda y.\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}\to\mathsf{Z}}\,(1)}{\lambda xy.\boldsymbol{times}\, x(\boldsymbol{plus}\, xy) : \mathsf{Z}\to\mathsf{Z}\to\mathsf{Z}}\,(2)$$

The Strong Normalization property for (plain) $\lambda\!\to$ implies that not all recursive functions are definable in the system. The same holds for the above $\lambda\delta$-calculus with integers. The following system of type assignment is such that all computable functions are representable by a typed term. Indeed, the system also assigns types to non-normalizing terms by introducing a primitive fixedpoint combinator $\boldsymbol{Y}$ having type $(\sigma\to\sigma)\to\sigma$ for every σ.

6.10. DEFINITION. (i) The $\lambda\boldsymbol{Y}\delta$*-calculus* is an extension of the $\lambda\delta$-calculus in which there is a constant $\boldsymbol{Y}$ with reduction rule

$$\boldsymbol{Y} f \to f(\boldsymbol{Y} f).$$

(ii) Type assignment to $\lambda\boldsymbol{Y}\delta$-terms is defined by adding the axioms

$$\boldsymbol{Y} : (\sigma\to\sigma)\to\sigma$$

for each $\sigma \in \mathbb{T}$. The resulting system is denoted by $\lambda\boldsymbol{Y}\delta\!\to$.

Because of the presence of $\boldsymbol{Y}$, not all terms have a normal form. Without proof we state the following.

6.11. THEOREM. (i) *The* $\lambda\boldsymbol{Y}\delta$*-calculus satisfies the Church-Rosser property.*

(ii) *If a term in the* $\lambda\boldsymbol{Y}\delta$*-calculus has a normal form, then it can be found using leftmost reduction.*

(iii) *The Subject Reduction property holds for* $\lambda\boldsymbol{Y}\delta\!\to$.

6.12. THEOREM. *All computable functions can be represented in the* $\lambda\boldsymbol{Y}\delta$-*calculus by a term typable in* $\lambda\boldsymbol{Y}\delta{\to}$.

PROOF. The construction uses the primitive numerals $\boldsymbol{n}$. If we take $\mathsf{S}^+_{\boldsymbol{Y}\delta} \equiv \lambda x.\boldsymbol{plus}\,x\,\mathbf{1}$, $\mathsf{P}^-_{\boldsymbol{Y}\delta} \equiv \lambda x.\boldsymbol{minus}\,x\,\mathbf{1}$, and $\mathsf{Zero}_{\boldsymbol{Y}\delta} \equiv \lambda x.\boldsymbol{equal}\,x\,\mathbf{0}$, then the proof of Theorem 3.13 can be imitated using $\boldsymbol{Y}$ instead of the fixedpoint combinator Y. The types for the functions defined using $\boldsymbol{Y}$ are natural. □

One could also add $\boldsymbol{Y}$ to the system $\lambda 2$ using the (single) axiom

$$\boldsymbol{Y} : \forall\alpha.(\alpha{\to}\alpha){\to}\alpha.$$

Exercises

6.1. Let $\boldsymbol{k}_n$ be defined by $\boldsymbol{k}_0 \equiv \mathsf{I}$ and $\boldsymbol{k}_{n+1} \equiv \mathsf{K}(\boldsymbol{k}_n)$. Show that on the $\boldsymbol{k}_n$ the recursive functions can be represented by terms in the $\lambda\delta_{\mathrm{C}}$-calculus.

6.2. Write down a $\lambda\delta$-term F in the system of Example 6.7 such that

$$F\boldsymbol{n} \twoheadrightarrow \boldsymbol{n!} + \boldsymbol{n}.$$

6.3. Write down a $\lambda\delta$-term F in the system of Example 6.8 such that for $s_1, s_2, t_1, t_2 \in \Sigma$ we have

$$\begin{array}{rcll} F[`s_1\text{'}, `t_1\text{'}][`s_2\text{'}, `t_2\text{'}] & \twoheadrightarrow & \boldsymbol{true}, & \text{if } (s_1,t_1) \text{ precedes } (s_2,t_2) \text{ in the} \\ & & & \text{lexicographical ordering of } \Sigma\times\Sigma; \\ & \twoheadrightarrow & \boldsymbol{false}, & \text{otherwise.} \end{array}$$

6.4. Give suitable typing axioms (in $\lambda{\to}$ and $\lambda 2$) for the constants in Example 6.6.

Chapter 7

Reduction Systems

In this chapter we consider some alternative models of computation based on rewriting. The objects in these models are terms built up from *constants* with *arity* in $\mathbb{N}$ and variables, using application.

7.1. DEFINITION. Let $\mathcal{C}$ be a set of constants. The set of *terms over* $\mathcal{C}$ (notation $\mathcal{T} = \mathcal{T}(\mathcal{C})$) is defined as follows.

$$\begin{array}{rcl} x \in V & \Rightarrow & x \in \mathcal{T}, \\ \boldsymbol{C} \in \mathcal{C}, t_1 \ldots, t_k \in \mathcal{T} & \Rightarrow & \boldsymbol{C}(t_1, \ldots, t_n) \in \mathcal{T}, \end{array}$$

where $n = \text{arity}(\boldsymbol{C})$.

Recursive programming schemes

The simplest reduction systems are *recursive programming schemes* (RPS).

The general form of an RPS has as language the terms $\mathcal{T}(\mathcal{C})$. On these a reduction relation is defined as follows.

$$\begin{array}{rcl} \boldsymbol{C}_1(x_1, \ldots, x_{n_1}) & \rightarrow & t_1, \\ & \vdots & \\ \boldsymbol{C}_k(x_1, \ldots, x_{n_k}) & \rightarrow & t_k, \end{array}$$

where $n_i = \text{arity}(\boldsymbol{C}_i)$. Here we have

(1) The $\boldsymbol{C}$'s are all different constants.
(2) The free variables in t_i are among the $x_1, \ldots, x_{n_i}$.
(3) In the t's there may be arbitrary $\boldsymbol{C}$'s.

For example, the system

$$\begin{array}{rcl} \boldsymbol{C}(x, y) & \rightarrow & \boldsymbol{D}(\boldsymbol{C}(x, x), y), \\ \boldsymbol{D}(x, y) & \rightarrow & \boldsymbol{C}(x, \boldsymbol{D}(x, y)) \end{array}$$

is an RPS.

The λ-calculus is powerful enough to 'implement' all these RPS's. We can find λ-terms with the specified reduction behaviour.

7.2. Theorem. *Each RPS can be represented in* λ*-calculus. For example (see above), there are terms* C *and* D *such that*

$$\begin{array}{rcl} \mathsf{C}xy & \twoheadrightarrow_\beta & \mathsf{D}(\mathsf{C}xx)y, \\ \mathsf{D}xy & \twoheadrightarrow_\beta & \mathsf{C}x(\mathsf{D}xy). \end{array}$$

Proof. By the reducing variant 4.27 of the Multiple Fixedpoint Theorem. $\square$

Without proof we mention the following.

7.3. Theorem. *Every RPS satisfies the Church-Rosser theorem.*

Term rewrite systems

More general than the RPS's are the so called *term rewrite systems* (TRS's), which use *pattern matching* in function definitions. A typical example is

$$\begin{array}{rcl} \boldsymbol{A}(\boldsymbol{0}, y) & \to & y, \\ \boldsymbol{A}(\boldsymbol{S}(x), y) & \to & \boldsymbol{S}(\boldsymbol{A}(x, y)). \end{array}$$

Then, for example, $\boldsymbol{A}(\boldsymbol{S}(\boldsymbol{0}), \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{0}))) \twoheadrightarrow \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{S}(\boldsymbol{0})))$.

The difference with RPS's is that in a TRS the arguments of a rewrite rule may have some structure. A constant in a TRS that does not have a contraction rule (i.e. no rewrite rule starts with that constant) is called a *constructor*. The other constants are called *functions*.

Not all TRS's satisfy the Church-Rosser property. Consider the system

$$\begin{array}{rcl} \boldsymbol{A}(x) & \to & \boldsymbol{B}, \\ \boldsymbol{A}(\boldsymbol{B}) & \to & \boldsymbol{C}. \end{array}$$

Then $\boldsymbol{A}(\boldsymbol{B})$ reduces both to $\boldsymbol{B}$ and to $\boldsymbol{C}$. It is said that the two rules *overlap*. The following rule overlaps with itself:

$$\boldsymbol{D}(\boldsymbol{D}(x)) \to \boldsymbol{E}.$$

Then $\boldsymbol{D}(\boldsymbol{D}(\boldsymbol{D}(\boldsymbol{D})))$ reduces to $\boldsymbol{E}$ and to $\boldsymbol{D}(\boldsymbol{E})$.

See Klop (1992) for a survey and references on TRS's.

Combinatory logic (CL) is a reduction system related to λ-calculus. Terms in CL consist of (applications of) constants $\boldsymbol{I}$, $\boldsymbol{K}$, $\boldsymbol{S}$ and variables, without arity restrictions. The contraction rules are

$$\begin{array}{rcl} \boldsymbol{I}x & \to & x, \\ \boldsymbol{K}xy & \to & x, \\ \boldsymbol{S}xyz & \to & xz(yz). \end{array}$$

(Note that $\boldsymbol{KI}$ is a nf.) Then $\boldsymbol{KII} \to \boldsymbol{I}$, and $\boldsymbol{SII}(\boldsymbol{SII})$ has no normal form. This CL can be represented as a TRS by considering $\boldsymbol{I}, \boldsymbol{K}, \boldsymbol{S}$ as (0-ary) constructors, together with a function $\boldsymbol{Ap}$ with arity 2, as follows.

$$\begin{array}{rcl} \boldsymbol{Ap}(\boldsymbol{I}, x) & \to & x, \\ \boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{K}, x), y) & \to & x, \\ \boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{S}, x), y), z) & \to & \boldsymbol{Ap}(\boldsymbol{Ap}(x, z), \boldsymbol{Ap}(y, z)). \end{array}$$

The CL-term $\boldsymbol{SII}(\boldsymbol{SII})$ is translated into $\underline{\boldsymbol{\Omega}} \equiv \boldsymbol{Ap}(\underline{\boldsymbol{\omega}}, \underline{\boldsymbol{\omega}})$ where $\underline{\boldsymbol{\omega}} \equiv \boldsymbol{Ap}(\boldsymbol{Ap}(\boldsymbol{S}, \boldsymbol{I}), \boldsymbol{I})$.

The Normalization Theorem does not extend to TRS's. Consider the above TRS-version of CL, together with the rules

$$\begin{array}{rcl} \boldsymbol{or}(x, \boldsymbol{true}) & \to & \boldsymbol{true}, \\ \boldsymbol{or}(\boldsymbol{true}, x) & \to & \boldsymbol{true}, \\ \boldsymbol{or}(\boldsymbol{false}, \boldsymbol{false}) & \to & \boldsymbol{false}. \end{array}$$

The expression

$$\boldsymbol{or}(A, B)$$

can, in general, not be normalized by contracting always the leftmost redex. In fact A and B have to be evaluated in parallel. Consider e.g. the terms

$$\boldsymbol{or}(\underline{\boldsymbol{\Omega}}, \boldsymbol{Ap}(\boldsymbol{I}, \boldsymbol{true}))$$

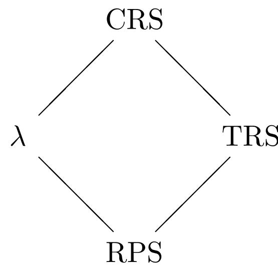
and

$$\boldsymbol{or}(\boldsymbol{Ap}(\boldsymbol{I}, \boldsymbol{true}), \underline{\boldsymbol{\Omega}}).$$

Therefore this system is called *non-sequential*.

Combinatory reduction systems

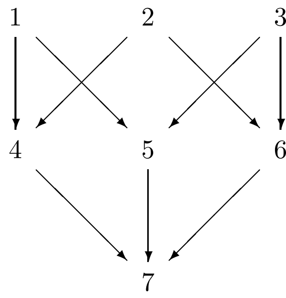
Even more general than TRS's are the combinatory reduction systems (CRS) introduced in Klop (1980). These are TRS's together with arbitrary variable binding operations. We have in fact



Exercises

7.1. (Toyama et al. (1989a), see also (1989b)) A TRS is called *strongly normalizing* (SN) if there is no term that has an infinite reduction path. So (the TRS version of) $\mathrm{CL}(\boldsymbol{S}, \boldsymbol{K})$ is not SN, but $\mathrm{CL}(\boldsymbol{I}, \boldsymbol{K})$ (with the obvious reduction rules) is. Define the following two TRS's.
$\mathcal{R}_1$:

$$\begin{array}{rcl} F(4, 5, 6, x) & \to & F(x, x, x, x), \\ F(x, y, z, w) & \to & 7, \end{array}$$



$\mathcal{R}_2$:

$$\begin{array}{rcl} G(x,x,y) & \to & x, \\ G(x,y,x) & \to & x, \\ G(y,x,x) & \to & x. \end{array}$$

Show that both $\mathcal{R}_1$ and $\mathcal{R}_2$ are SN, but the union $\mathcal{R}_1 \cup \mathcal{R}_2$ is not.

Bibliography

Abramsky, S., D.M. Gabbay and T.S.E. Maibaum (eds.) (1992). *Handbook of Logic in Computer Science*, Vol. II, Oxford University Press.

van Bakel, S.J. (1992). Complete restrictions of the intersection type discipline, *Theoretical Computer Science* **102**, pp. 135–163.

Barendregt, H.P. (1976). A global representation of the recursive functions in the lambda calculus, *Theoretical Computer Science* **3**, pp. 225–242.

Barendregt, H.P. (1984). *The Lambda Calculus: Its Syntax and Semantics*, Studies in Logic 103, second, revised edition, North-Holland, Amsterdam.

Barendregt, H.P. (1990). Functional programming and lambda calculus, *in:* J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol. II, Elsevier/MIT Press.

Barendregt, H.P. (1992). Lambda calculi with types, *in:* Abramsky et al. (1992), pp. 117–309.

de Bruijn, N.G. (1970). The mathematical language AUTOMATH, its usage and some of its extensions, *in:* M. Laudet, D. Lacombe and M. Schuetzenberger (eds.), *Symposium on Automatic Demonstration*, INRIA, Versailles, Lecture Notes in Computer Science 125, Springer-Verlag, Berlin, pp. 29–61. Also in Nederpelt et al. (1994).

de Bruijn, N.G. (1980). A survey of the AUTOMATH project, *in:* Hindley and Seldin (1980), pp. 580–606.

Church, A. (1936). An unsolvable problem of elementary number theory, *American Journal of Mathematics* **58**, pp. 354–363.

Church, A. (1940). A formulation of the simple theory of types, *Journal of Symbolic Logic* **5**, pp. 56–68.

Church, A. (1941). *The Theory of Lambda Conversion*, Princeton University Press.

Curry, H.B. (1934). Functionality in combinatory logic, *Proceedings of the National Academy of Science USA* **20**, pp. 584–590.

Curry, H.B. (1969). Modified basic functionality in combinatory logic, *Dialectica* **23**, pp. 83–92.

Curry, H.B. and R. Feys (1958). *Combinatory Logic*, Vol. I, North-Holland, Amsterdam.

Curry, H.B., J.R. Hindley and J.P. Seldin (1972). *Combinatory Logic*, Vol. II, North-Holland, Amsterdam.

Girard, J.-Y. (1972). *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*, Dissertation, Université Paris VII.

Hindley, J.R. (1969). The principal typescheme of an object in combinatory logic, *Transactions of the American Mathematical Society* **146**, pp. 29–60.

Hindley, J.R. and J.P. Seldin (eds.) (1980). *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, New York.

Howard, W.A. (1980). The formulae-as-types notion of construction, *in:* Hindley and Seldin (1980), pp. 479–490.

Kleene, S.C. (1936). λ-definability and recursiveness, *Duke Mathematical Journal* **2**, pp. 340–353.

Klop, J.W. (1980). *Combinatory Reduction Systems*, Dissertation, Utrecht University. CWI Tract, Amsterdam.

Klop, J.W. (1992). Term rewrite systems, *in:* Abramsky et al. (1992).

Läuchli, H. (1970). An abstract notion of realizability for which intuitionistic predicate logic is complete, *in:* G. Myhill, A. Kino and R. Vesley (eds.), *Intuitionism and Proof Theory: Proceedings of the Summer School Conference*, Buffalo, New York, North-Holland, Amsterdam, pp. 227–234.

Milner, R. (1978). A theory of type polymorphism in programming, *Journal of Computer and Systems Analysis* **17**, pp. 348–375.

Milner, R. (1984). A proposal for standard ML, *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, pp. 184–197.

Nederpelt, R.P., J.H. Geuvers and R.C. de Vrijer (eds.) (1994). *Selected Papers on Automath*, Studies in Logic 133, North-Holland, Amsterdam.

Prawitz, D. (1965). *Natural Deduction: A Proof-Theoretical Study*, Almqvist and Wiksell, Stockholm.

Reynolds, J.C. (1974). Towards a theory of type structure, *Colloque sur la Démonstration*, Paris, Lecture Notes in Computer Science 19, Springer-Verlag, Berlin, pp. 408–425.

Schönfinkel, M. (1924). Über die Bausteinen der mathematische Logik, *Mathematische Annalen* **92**, pp. 305–316.

Schwichtenberg, H. (1977). Proof theory: applications of cut-elimination, *in:* J. Barwise (ed.), *Handbook of Mathematical Logic*, North-Holland, Amsterdam, pp. 867–895.

Toyama, Y., J.W. Klop and H.P. Barendregt (1989a). Termination for the direct sum of left-linear term rewriting systems, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, Chapel Hill, Lecture Notes in Computer Science 355, Springer-Verlag, Berlin, pp. 477–491.

Toyama, Y., J.W. Klop and H.P. Barendregt (1989b). Termination for the direct sum of left-linear term rewriting systems, *Technical Report CS-R8923*, Centre for Mathematics and Computer Science (CWI), Amsterdam.

Turing, A.M. (1936/7). On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* **42**, pp. 230–265.

Turing, A.M. (1937). Computability and λ-definability, *Journal of Symbolic Logic* **2**, pp. 153–163.

Wells, J.B. (1994). Typability and type-checking in the second-order λ-calculus are equivalent and undecidable, *Proceedings of the 9th Annual Symposium on Logic in Computer Science*, Paris, France, IEEE Computer Society Press, pp. 176–185.

Exercises of the class of Herman Geuvers

Exercises 1a: Simple Type Theory

1. Find inhabitants (i.e. *closed* terms) of the following types (in STT)
   (a) $(\alpha\to\beta)\to(\beta\to\gamma)\to\alpha\to\gamma$
   (b) $\alpha\to\beta\to(\alpha\to\beta\to\gamma)\to\gamma$
   (c) $((\alpha\to\beta\to\alpha)\to\alpha)\to\alpha$
   (d) $\beta\to((\alpha\to\beta)\to\gamma)\to\gamma$

2. The type $\alpha\to(\alpha\to\alpha)\to\alpha$ is also called nat.
   (a) Show that there are infinitely many closed terms (inhabitants) of type nat.
   (b) Describe a term $0 : \mathsf{nat}$ and the succesor $\mathsf{succ} : \mathsf{nat}\to\mathsf{nat}$.
   (c) Describe the derivations that the (infinitely many) terms under (a) correspond to.
   (d) Construct a derivation of $((\alpha\to\beta)\to\gamma)\to\beta\to\gamma$ and the associated typed λ-term.

3. Add *product types* to $\lambda\to$, that is add $\sigma\times\tau$ to the types and
   (a) Add the appropriate term constructors to extend the term language of $\lambda\to$.
   (b) Give typing rules for terms of type $\sigma\times\tau$, by giving an *elimination rule* and an *introduction rule*. (A term of type $\sigma\times\tau$ should be built up from a term of type σ and a term of type τ.)
   (c) Give a reduction rule for the new term constructors. Try to give an "β-like" rule and an "η-like" rule.

4. Prove the claim made in the proof of the Weak Normalization theorem (page 16 of the slides of lesson 2): If we reduce in P a redex of maximum height (height $h(P)$) that does not contain any other redex of height $h(P)$, obtaining the term Q, then $m(Q) <_l m(P)$.

5. Fill the three gaps in the proof of Strong Normalization (page 17 of the slides of lesson 2). That is, prove
   (a) $[\![\sigma]\!] \subseteq \mathsf{SN}$ (by induction on σ)
   (b) If $M[N/x]\vec{P} \in [\![\tau]\!]$, $N \in [\![\sigma]\!]$, then $(\lambda x.M)N\vec{P} \in [\![\tau]\!]$ (by induction on σ)
   (c) (By induction on the derivation of $\Gamma \vdash M : \sigma$).

$$\left.\begin{array}{r} x_1{:}\tau_1,\ldots,x_n{:}\tau_n \vdash M : \sigma \\ N_1 \in [\![\tau_1]\!],\ldots,N_n \in [\![\tau_n]\!] \end{array}\right\} \Rightarrow M[N_1/x_1,\ldots N_n/x_n] \in [\![\sigma]\!]$$

6. Prove the Substitution Lemma (by induction on the derivation of $\Gamma, x : \tau, \Delta \vdash M : \sigma$). (That is, prove that if $\Gamma, x : \tau, \Delta \vdash M : \sigma$ and $\Gamma \vdash P : \tau$, then $\Gamma, \Delta \vdash M[P/x] : \sigma$.)

Exercises 1b: Polymorphic Lambda Calculus

1. $\perp$ is the type $\forall\alpha.\alpha$. Give the typing derivations of the following typing. $\lambda x{:}\perp.\lambda\alpha.x(\alpha\to\alpha)(x\alpha)$.

2. Find terms of the following types in $\lambda 2$. (See the slides for the definitions.)

   (a) $\sigma\to\sigma\vee\tau$. Now make this term polymorphic in σ and τ.

   (b) $\sigma\to\tau\to\sigma\wedge\tau$

   (c) $\forall\beta.\sigma\to\exists\alpha.\sigma[\alpha/\beta]$. Which logical rule does this term correspond to?

   (d) Given $M:\exists\alpha.\sigma$ and $F:\forall\alpha.\sigma\to\tau$, with $\alpha\notin\mathrm{FV}(\tau)$, construct a term of type τ. Which logical rule does this term correspond to?

3. Define the type of booleans bool in $\lambda 2$ as $\mathsf{bool} := \forall\alpha.\alpha\to\alpha\to\alpha$

   (a) Define $\mathsf{true} : \mathsf{bool}$ and $\mathsf{false} : \mathsf{bool}$.

   (b) Define conjunction and disjunction over the booleans

4. Recall the natural numbers in $\lambda 2$.

   (a) Define exponentiation $\mathsf{exp} : \mathsf{nat}\to\mathsf{nat}\to\mathsf{nat}$ on the natural numbers in $\lambda 2$. (Use the iterator and already defined functions.)

   (b) Define the function $\mathsf{Z?} : \mathsf{nat}\to\mathsf{bool}$ such that $\mathsf{Z?}0 =_\beta \mathsf{true}$ and $\mathsf{Z?}(Sx) =_\beta \mathsf{false}$.

5. The type of lists over A is defined by $\mathsf{list}_A := \forall\alpha.\alpha\to(A\to\alpha\to\alpha)\to\alpha$.

   (a) Define the "head" function over list_A. This function requires a "default value" for the case of the nil-list:

   $$\mathsf{head} : A\to\mathsf{list}_A\to A.$$

   NB. The tail function is not so easy to define. It can't be defined directly by iteration.

   (b) Define the function $\mathsf{suclist} : \mathsf{list}_{\mathsf{nat}}\to\mathsf{nat}$ that adds 1 to each element in a list of natural numbers. (See the "map" function on the slides.)

6. Consider the type of Binary trees with nodes in A and leaves in B, as given in the lecture:

   $$\mathsf{tree}_{A,B} := \forall\alpha.(B\to\alpha)\to(A\to\alpha\to\alpha\to\alpha)\to\alpha$$

   (a) Define the functions $\mathsf{leaf} : B\to\mathsf{tree}_{A,B}$ and $\mathsf{join} : A\to\mathsf{tree}_{A,B}\to\mathsf{tree}_{A,B}\to\mathsf{tree}_{A,B}$.

   (b) Define the *iterator* for tree:

   $$\mathsf{it} : \forall\gamma.(B\to\gamma)\to(A\to\gamma\to\gamma\to\gamma)\to\mathsf{tree}_{A,B}\to\gamma.$$

   (Given a type γ and functions $f : (\gamma\to B)$ and $g : (\gamma\to A\to A)$, it should produce a function from $\mathsf{tree}_{A,B}$ to γ.)

(c) Take $B :=$ nat and write a function tsuml that computes the sum of all leaves.

(d) Take $A :=$ nat and write a function tsumln that computes the sum of all leaves and nodes.

(e) Take $A :=$ bool and write a function tend that computes the leave (a term of type B) that is found by going "left" if the boolean in the node is true and "right" if it's false.

(f) Take $A, B :=$ bool and write a function tpath that computes the path (as a term of type $\text{List}_{\mathsf{bool}}$ to the leaf by going "left" if the boolean in the node is true and "right" if it's false.

7. Prove Strong Normalization for $\lambda 2$ by proving the following by induction on the derivation.
Proposition

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \sigma \Rightarrow M[P_1/x_1, \ldots, P_n/x_n] \in [\![\sigma]\!]_\rho$$

for all valuations ρ and $P_1 \in [\![\tau_1]\!]_\rho, \ldots, P_n \in [\![\tau_n]\!]_\rho$
See the slides or the Handbook article by Barendregt (Def 4.1.7, page 50) for the derivation rules of $\lambda 2$. See the slides for how this fits in the proof of SN.

Exercises of the class of Herman Geuvers

Exercises 2a: Higher Order Logic

1. Define the Leibniz equality on A as $t =_A q := \forall P{:}A{\to}\mathsf{Prop}.(P\,t){\to}(P\,q)$. Prove the following by finding terms of the associated types.

   (a) reflexivity of $=_A$: $\forall x{:}A.x =_A x$.

   (b) transitivity of $=_A$: $\forall x, y, z{:}A.x =_A y \to y =_A z \to x =_A z$.

   (c) symmetry of $=_A$: $\forall x, y{:}A.x =_A y \to y =_A x$.

2. The transitive closure of a relation R is defined as follows.

   $\mathsf{trclos} := \lambda R{:}A{\to}A{\to}\mathsf{Prop}.\lambda x, y{:}A.(\forall Q{:}A{\to}A{\to}\mathsf{Prop}.(\mathsf{trans}(Q){\to}(R \subseteq Q){\to}(Q\,x\,y)))$.

   So trclos is of type $(A{\to}A{\to}\mathsf{Prop}){\to}(A{\to}A{\to}\mathsf{Prop})$

   (a) Define the notions trans and $\subseteq$ in the definition of trclos.

   (b) Prove that the transitive closure is transitive. (Find a term of type $\mathsf{trans}(\mathsf{trclos}\,R)$).

   (c) Prove that the transitive closure of R contains R. (Find a term of type $R \subseteq (\mathsf{trclos}\,R)$).

3. In this exercises we will prove in higher order logic a variant of the Knaster-Tarski fixed-point theorem.
   Given a domain A, we identify $A{\to}\mathsf{Prop}$ with the collection of subsets of A. In this exercise we consider maps $\Phi : (A{\to}\mathsf{Prop}){\to}(A{\to}\mathsf{Prop})$, mapping subsets of A to subsets of A.
   Φ is asumed to be *monotone*: $\forall P, Q{:}A{\to}\mathsf{Prop}.P \subseteq Q \to (\Phi\,P) \subseteq (\Phi\,Q)$.
   ($P \subseteq Q$ is an abbreviation for $\forall x{:}A.(P\,x){\to}(Q\,x)$, which is also gives away the answer to the exercise above.)
   $P : A{\to}\mathsf{Prop}$ is called Φ-*closed* if $(\Phi\,P) \subseteq P$.

   (a) Define (formally) $X : A{\to}\mathsf{Prop}$ as the *smallest* Φ-*closed* subset of A.

   (b) Prove (for arbitrary $P : A{\to}\mathsf{Prop}$): if P is Φ-closed, then $X \subseteq P$.
   (Find a term of type $\forall P{:}A{\to}\mathsf{Prop}.(\Phi\,P) \subseteq P \to X \subseteq P$.)

   (c) Prove $(\Phi\,X) \subseteq X$.

   (d) Prove $X \subseteq (\Phi\,X)$.

   (e) Conclude that X is the *least fixed point of* Φ:

      i. $X \approx (\Phi\,X)$,

      ii. $P \approx (\Phi\,P) \to X \subseteq P$.

   where we take the equality $\approx$ to be defined in the set-theoretical way as
   $P \approx Q := P \subseteq Q \wedge Q \subseteq P$.

4. Recall the induction principle over natural numbers as a higher order formula. Given a domain N and $0 : N, S : N \to N$, Ind_N is

$$\forall P{:}N\to\mathsf{Prop}.(P\,0) \to (\forall x{:}N.(P\,x)\to(P(S\,x))) \to \forall x{:}N.(P\,x)$$

   (a) Consider a datatype of lists over a base domain A. So we have two base domains A and L and we let $\mathrm{Nil} : L, \mathrm{Cons} : A\to L\to L$. Define the induction principle over lists.

   (b) Consider a datatype of binary trees with leaves in base type A and node labels in base type B. So So we have three base domains A, B and $T : \mathsf{Set}$ and we let $\mathrm{Leaf} : A\to T, \mathrm{Join} : B\to T\to T\to T$. Define the induction principle over these trees.

Exercises 2b: Extensions of λHOL; the λ cube; PTSs

1. (a) Explain for every $\to$ and Π in the following judgment which Π-rule (of λHOL) is needed to make it a valid construction.

$$A : \mathsf{Type}, R : A\to A\to\mathsf{Prop} \vdash \Pi x{:}A.\Pi Q{:}(A\to\mathsf{Prop})\to\mathsf{Prop}.Q(R\,x)\to R\,x\,x : \mathsf{Prop}$$

   (b) Do the same for the following judgment in CC.

$$A : \mathsf{Prop} \vdash \Pi F{:}(\Pi\alpha{:}\mathsf{Prop}.\Pi Q{:}\alpha\to\mathsf{Prop}.\Pi y{:}\alpha Q\,y\to Q\,y).F\,A \to \mathsf{Prop} : \mathsf{Type}$$

2. Give a context Γ and a term M of the type

$$(\Pi x{:}A.(R\,x\,a\to R\,a\,(f\,x)))\to R\,a\,a\to R\,a\,(f\,a)$$

   in this context.
   What is the simplest system of the λ cube in which this typing is valid?

3. (a) Recall the polymorphic type of *lists over* A, List_A and define it in $\lambda 2$. (So $A : \mathsf{Prop} \vdash \mathrm{List}_A : \mathsf{Prop}$; verify that this is indeed possible in the λ-cube system $\lambda 2$.)

   (b) Define *induction over lists* as a proposition in λP2. (So $A : \mathsf{Prop} \vdash \mathsf{ind}_{\mathrm{List}} : \mathsf{Prop}$; verify that this is indeed possible in the λ-cube system λP2.)

4. Define in CC, $\varphi := \forall x{:}A.x = a$, $\psi := \forall x{:}B.\exists y{:}B.x \neq y$ (with $A, B : \mathsf{Prop}$) and define

$$\mathrm{EXT} := \forall\alpha,\beta{:}\mathsf{Prop}.(\alpha \Leftrightarrow \beta) \Rightarrow (\alpha =_{\mathsf{Prop}} \beta).$$

   Give a term of type $\bot$ in CC in the following context

$$e : \mathrm{EXT}, A, B : \mathsf{Prop}, h_1 : \varphi, h_2 : \psi$$

   Alternatively you may try to find this term in Coq, see the file `coq_ex7.v8`.

5. Prove the following basic property for any Pure Type System $(\mathcal{S}, \mathcal{A}, \mathcal{R})$. (By induction on the derivation.)
   (Variable Lemma)
   If $\Gamma \vdash M : A$, then $\Gamma \vdash x : B$ for all $x : B \in \Gamma$.

6. Prove the Substitution Lemma for PTSs. (By induction on the derivation; do the cases for the last rule being (weak) or (λ).

Types Summer School
Gothenburg Sweden August 2005

Type Systems

Herman Geuvers
Radboud University Nijmegen, NL

Lecture 1: Simple and Polymorphic Type Theory

Simplest system: $\lambda\!\!\to$ just arrow types

$$\text{Typ} := \text{TVar} \mid (\text{Typ}{\to}\text{Typ})$$

- Examples: $(\alpha{\to}\beta){\to}\alpha$, $(\alpha{\to}\beta){\to}((\beta{\to}\gamma){\to}(\alpha{\to}\gamma))$
- Brackets associate to the right and outside brackets are omitted:
  $(\alpha{\to}\beta){\to}(\beta{\to}\gamma){\to}\alpha{\to}\gamma$
- Types are denoted by $\sigma, \tau, \ldots$.

Terms:

- typed variables $x_1^\sigma, x_2^\sigma, \ldots$, countably many for every σ.
- application: if $M : \sigma{\to}\tau$ and $N : \sigma$, then $(MN) : \tau$
- abstraction: if $P : \tau$, then $(\lambda x^\sigma.P) : \sigma{\to}\tau$

Examples:

$$\begin{aligned} \lambda x^\sigma.\lambda y^\tau.x &: \sigma{\to}\tau{\to}\sigma \\ \lambda x^{\alpha\to\beta}.\lambda y^{\beta\to\gamma}.\lambda z^\alpha.y(xz) &: (\alpha{\to}\beta){\to}(\beta{\to}\gamma){\to}\alpha{\to}\gamma \\ \lambda x^\alpha.\lambda y^{(\beta\to\alpha)\to\alpha}.y(\lambda z^\beta.x)) &: \alpha{\to}((\beta{\to}\alpha){\to}\alpha){\to}\alpha \end{aligned}$$

For every type there is a term of that type:

$$x^\sigma : \sigma$$

Not for every type there is a closed term of that type:

$(\alpha{\to}\alpha){\to}\alpha$ is not inhabited

Formulation with contexts to declare the free variables:

$$x_1 : \sigma_1, x_2 : \sigma_2, \ldots, x_n : \sigma_n$$

is a context, usually denoted by Γ.

Derivation rules of $\lambda\!\!\to$:

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash M : \sigma{\to}\tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \qquad \frac{\Gamma, x{:}\sigma \vdash P : \tau}{\Gamma \vdash \lambda x{:}\sigma.P : \sigma{\to}\tau}$$

$\Gamma \vdash_{\lambda\to} M : \sigma$ if there is a derivation using these rules with conclusion $\Gamma \vdash M : \sigma$

Typical problems one would like to have an algorithm for:

$\Gamma \vdash M : \sigma?$	Type Checking Problem	TCP
$\Gamma \vdash M : ?$	Type Synthesis Problem	TSP
$\vdash ? : \sigma$	Type Inhabitation Problem (by a closed term)	TIP

Typical problems one would like to have an algorithm for:

$\Gamma \vdash M : \sigma?$	Type Checking Problem	TCP
$\Gamma \vdash M : ?$	Type Synthesis Problem	TSP
$\vdash ? : \sigma$	Type Inhabitation Problem (by a closed term)	TIP

For $\lambda{\rightarrow}$, all these problems are decidable.
Remarks:

- TCP and TSP are (usually) equivalent:
  To solve $MN : \sigma$, one has to solve $N :?$ (and if this gives answer τ, solve $M : \tau{\rightarrow}\sigma$).
- TIP is undecidable for most extensions of $\lambda{\rightarrow}$, as it corresponds to provability in some logic.

Formulas-as-Types (Curry, Howard):

There are two readings of a judgement $M : \sigma$

1. term as algorithm/program, type as specification:
   M is a function of type σ
2. type as a proposition, term as its proof:
   M is a proof of the proposition σ

Formulas-as-Types (Curry, Howard):

There are two readings of a judgement $M : \sigma$

1. term as algorithm/program, type as specification:
   M is a function of type σ
2. type as a proposition, term as its proof:
   M is a proof of the proposition σ

- There is a one-to-one correspondence between
  - typable terms in $\lambda{\rightarrow}$
  - derivations in minimal proposition logic
- The judgement

  $$x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n \vdash M : \sigma$$

  can be read as

  M is a proof of σ from the assumptions $\tau_1, \tau_2, \ldots, \tau_n$.

Example

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[\alpha\to\beta\to\gamma]^3 \quad [\alpha]^1}{\beta\to\gamma} \qquad \cfrac{[\alpha\to\beta]^2 \quad [\alpha]^1}{\beta}}{\gamma}}{\alpha\to\gamma}\,1}{(\alpha\to\beta)\to\alpha\to\gamma}\,2}{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}\,3 \quad \simeq \quad \begin{array}{c}\lambda x{:}\alpha\to\beta\to\gamma.\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz)\\ :(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma\end{array}$$

Example

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[\alpha\to\beta\to\gamma]^3 \quad [\alpha]^1}{\beta\to\gamma} \qquad \cfrac{[\alpha\to\beta]^2 \quad [\alpha]^1}{\beta}}{\gamma}}{\alpha\to\gamma}\,1}{(\alpha\to\beta)\to\alpha\to\gamma}\,2}{(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}\,3 \quad \simeq \quad \begin{array}{c}\lambda x{:}\alpha\to\beta\to\gamma.\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz)\\ :(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma\end{array}$$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[x:\alpha\to\beta\to\gamma]^3 \quad [z:\alpha]^1}{xz:\beta\to\gamma} \qquad \cfrac{[y:\alpha\to\beta]^2 \quad [z:\alpha]^1}{yz:\beta}}{xz(yz):\gamma}}{\lambda z{:}\alpha.xz(yz):\alpha\to\gamma}\,1}{\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz):(\alpha\to\beta)\to\alpha\to\gamma}\,2}{\lambda x{:}\alpha\to\beta\to\gamma.\lambda y{:}\alpha\to\beta.\lambda z{:}\alpha.xz(yz):(\alpha\to\beta\to\gamma)\to(\alpha\to\beta)\to\alpha\to\gamma}\,3$$

Computation:

- β-reduction: $(\lambda x{:}\sigma.M)P \longrightarrow_\beta M[P/x]$
- η-reduction: $\lambda x{:}\sigma.Mx \longrightarrow_\eta M$ if $x \notin \mathsf{FV}(M)$

Cut-elimination in minimal logic $=$ β-reduction in $\lambda\!\to$.

$$\cfrac{\cfrac{\begin{array}{c}[\sigma]^1\\ \mathcal{D}_1\\ \tau\end{array}}{\sigma\to\tau}\,1 \qquad \begin{array}{c}\mathcal{D}_2\\ \sigma\end{array}}{\tau} \quad \rightsquigarrow \quad \begin{array}{c}\mathcal{D}_2\\ \sigma\\ \mathcal{D}_1\\ \tau\end{array}$$

$$\cfrac{\cfrac{\begin{array}{c}[x:\sigma]^1\\ \mathcal{D}_1\\ M:\tau\end{array}}{\lambda x{:}\sigma.M:\sigma\to\tau}\,1 \qquad \begin{array}{c}\mathcal{D}_2\\ P:\sigma\end{array}}{(\lambda x{:}\sigma.M)P:\tau} \quad \rightsquigarrow \quad \begin{array}{c}\mathcal{D}_2\\ P:\sigma\\ \mathcal{D}_1\\ M[P/x]:\tau\end{array}$$

Properties of $\lambda\!\to$.

- Uniqueness of types
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.
- Subject Reduction
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.
- Strong Normalization
  If $\Gamma \vdash M : \sigma$, then all $\beta\eta$-reductions from M terminate.

Properties of $\lambda\!\rightarrow$.

- Uniqueness of types
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.
- Subject Reduction
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.
- Strong Normalization
  If $\Gamma \vdash M : \sigma$, then all $\beta\eta$-reductions from M terminate.
- Substitution property
  If $\Gamma, x : \tau, \Delta \vdash M : \sigma$, $\Gamma \vdash P : \tau$, then $\Gamma, \Delta \vdash M[P/x] : \sigma$.
- Thinning
  If $\Gamma \vdash M : \sigma$ and $\Gamma \subseteq \Delta$, then $\Delta \vdash M : \sigma$.
- Strengthening
  If $\Gamma, x : \tau \vdash M : \sigma$ and $x \notin \mathsf{FV}(M)$, then $\Gamma \vdash M : \sigma$.

Strong Normalization of β for $\lambda\!\rightarrow$.
Note:

- Terms may get larger under reduction
  $(\lambda f.\lambda x.f(fx))P \longrightarrow_\beta \lambda x.P(Px)$
- Redexes may get multiplied under reduction.
  $(\lambda f.\lambda x.f(fx))((\lambda y.M)Q) \longrightarrow_\beta \lambda x.((\lambda y.M)Q)(((\lambda y.M)Q)x)$
- New redexes may be created under reduction.
  $(\lambda f.\lambda x.f(fx))(\lambda y.N) \longrightarrow_\beta \lambda x.(\lambda y.N)((\lambda y.N)x)$

Strong Normalization of β for $\lambda\!\rightarrow$ à la Curry is proved by constructing a model of $\lambda\!\rightarrow$.
Definition

- $[\![\alpha]\!] :=$ SN (the set of strongly normalizing λ-terms).
- $[\![\sigma\!\rightarrow\!\tau]\!] := \{M \mid \forall N \in [\![\sigma]\!](MN \in [\![\tau]\!])\}$.

Lemma (both by induction on σ)

- $[\![\sigma]\!] \subseteq$ SN
- If $M[N/x]\vec{P} \in [\![\tau]\!]$, $N \in [\![\sigma]\!]$, then $(\lambda x.M)N\vec{P} \in [\![\tau]\!]$.

Proposition

$$\left.\begin{array}{l} x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash M : \sigma \\ N_1 \in [\![\tau_1]\!], \ldots, N_n \in [\![\tau_n]\!] \end{array}\right\} \Rightarrow M[N_1/x_1, \ldots N_n/x_n] \in [\![\sigma]\!]$$

Proof By induction on the derivation of $\Gamma \vdash M : \sigma$.
Corollary $\lambda\!\rightarrow$ is SN
Proof By taking $N_i := x_i$ in the Proposition.

Polymorphic λ-calculus

Why Polymorphic λ-calculus?

- Simple type theory $\lambda\!\rightarrow$ is not very expressive
- In simple type theory, we can not 'reuse' a function.
  E.g. $\lambda x{:}\alpha.x : \alpha\!\rightarrow\!\alpha$ and $\lambda x{:}\beta.x : \beta\!\rightarrow\!\beta$.

We want to define functions that can treat types polymorphically: add types $\forall\alpha.\sigma$:
Examples

- $\forall\alpha.\alpha\!\rightarrow\!\alpha$
  If $M : \forall\alpha.\alpha\!\rightarrow\!\alpha$, then M can map any type to itself.
- $\forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$
  If $M : \forall\alpha.\forall\beta.\alpha\!\rightarrow\!\beta\!\rightarrow\!\alpha$, then M can take two inputs (of arbitrary types) and return a value of the first input type.

Derivation rules of $\lambda 2$:
Full (system F-style) polymorphism:

$$\mathsf{Typ} := \mathsf{TVar} \mid (\mathsf{Typ}{\rightarrow}\mathsf{Typ}) \mid \forall\alpha.\mathsf{Typ}.$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.\sigma}\ \alpha \notin \mathsf{FV}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall\alpha.\sigma}{\Gamma \vdash M\tau : \sigma[\tau/\alpha]}\ \text{for } \tau \text{ any type}$$

Examples:

- $\lambda\alpha.\lambda\beta.\lambda x{:}\alpha.\lambda y{:}\beta.x : \forall\alpha.\forall\beta.\alpha{\rightarrow}\beta{\rightarrow}\alpha.$
- $\lambda x{:}(\forall\alpha.\alpha).\lambda y{:}\sigma.x\tau : (\forall\alpha.\alpha){\rightarrow}\sigma{\rightarrow}\tau.$
- $\lambda x{:}(\forall\alpha.\alpha).x\,(\sigma{\rightarrow}\tau)\,(x\,\sigma) : (\forall\alpha.\alpha){\rightarrow}\tau.$

Recall: Important Properties

$\Gamma \vdash M : \sigma?$	TCP
$\Gamma \vdash M : ?$	TSP
$\vdash ? : \sigma$	TIP

Properties of $\lambda 2$

- TIP is undecidable,
- TCP and TSP are equivalent & decidable.

Formulas-as-types for $\lambda 2$:
There is a formulas-as-types isomorphism between $\lambda 2$ and second order proposition logic, PROP2

Derivation rules of PROP2:

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \forall\alpha.\sigma}\ \alpha \notin \mathsf{FV}(\Gamma) \qquad \frac{\Gamma \vdash \forall\alpha.\sigma}{\Gamma \vdash \sigma[\tau/\alpha]}$$

NB This is constructive second order proposition logic:

$$\forall\alpha.\forall\beta.((\alpha{\rightarrow}\beta){\rightarrow}\alpha){\rightarrow}\alpha \quad \text{Peirce's law}$$

is not derivable.

Definability of the other connectives:

$$\begin{aligned}
\bot &:= \forall\alpha.\alpha \\
\sigma\wedge\tau &:= \forall\alpha.(\sigma{\rightarrow}\tau{\rightarrow}\alpha){\rightarrow}\alpha \\
\sigma\vee\tau &:= \forall\alpha.(\sigma{\rightarrow}\alpha){\rightarrow}(\tau{\rightarrow}\alpha){\rightarrow}\alpha \\
\exists\alpha.\sigma &:= \forall\beta.(\forall\alpha.\sigma{\rightarrow}\beta){\rightarrow}\beta
\end{aligned}$$

and all the standard constructive derivation rules are derivable.

Example ($\wedge$-elimination):

$$\dfrac{\dfrac{\forall\alpha.(\sigma{\rightarrow}\tau{\rightarrow}\alpha){\rightarrow}\alpha}{(\sigma{\rightarrow}\tau{\rightarrow}\sigma){\rightarrow}\sigma} \qquad \dfrac{\dfrac{[\sigma]^1}{\tau{\rightarrow}\sigma}}{\sigma{\rightarrow}\tau{\rightarrow}\sigma}\,1}{\sigma}$$

Idea:
The definition of a connective is an encoding of the elimination rule.

Data types in $\lambda 2$

$$\text{Nat} := \forall\alpha.\alpha{\to}(\alpha{\to}\alpha){\to}\alpha$$

This type can be used as the type of natural numbers, using the encoding of $\mathbb{N}$ as Church numerals in the λ-calculus.

$$n \mapsto \lambda x.\lambda f.f(\ldots(f x)) \quad n\text{-times } f$$

- $0 := \lambda\alpha.\lambda x{:}\alpha.\lambda f{:}\alpha{\to}\alpha.x$
- $S := \lambda n{:}\text{Nat}.\lambda\alpha.\lambda x{:}\alpha.\lambda f{:}\alpha{\to}\alpha.f(n\alpha x f)$
- Iteration: if $c{:}\sigma$ and $g{:}\sigma{\to}\sigma$, then define $\text{It}\, c\, g : \text{Nat}{\to}\sigma$ as

  $$\lambda n{:}\text{Nat}.n\,\sigma\, c\, g$$

  Then

  $$\text{It}\, c\, g\, n = g(\ldots(g c)) \ \ (n \text{ times } g)$$

  $\Rightarrow$ Define $+$, $\times$, $\ldots$ using iteration.

Properties of $\lambda 2$.

- Uniqueness of types
  If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.
- Subject Reduction
  If $\Gamma \vdash M : \sigma$ and $M \longrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.
- Strong Normalization
  If $\Gamma \vdash M : \sigma$, then all $\beta\eta$-reductions from M terminate.

Strong Normalization of β for $\lambda 2$.

Note:

- There are two kinds of β-reductions
  - $(\lambda x{:}\sigma.M)P \longrightarrow_\beta M[P/x]$
  - $(\lambda\alpha.M)\tau \longrightarrow_\beta M[\tau/\alpha]$
- The second doesn't do any harm: we can just look at the underlying untyped λ-terms

Recall the proof for $\lambda{\to}$:

- $[\![\alpha]\!] := \text{Term}(\alpha) \cap \text{SN}$.
- $[\![\sigma{\to}\tau]\!] := \{M : \sigma{\to}\tau | \forall N \in [\![\sigma]\!](MN \in [\![\tau]\!])\}$.

Question:
How to define $[\![\forall\alpha.\sigma]\!]$??

$$[\![\forall\alpha.\sigma]\!] := \Pi_{X \in U} [\![\sigma]\!]_{\alpha:=X}??$$

Strong Normalization of β for $\lambda 2$.

Question:
How to define $[\![\forall\alpha.\sigma]\!]$??

$$[\![\forall\alpha.\sigma]\!] := \Pi_{X \in U} [\![\sigma]\!]_{\alpha:=X}??$$

- What is U?
  The collection of all 'possible' interpretations of types (?)
- $\Pi_{X \in U} [\![\sigma]\!]_{\alpha:=X}$ may get very (too?) big.

Girard:

- $[\![\forall\alpha.\sigma]\!]$ should be small

  $$\bigcap_{X \in U} [\![\sigma]\!]_{\alpha:=X}$$

- Characterization of U.

U := SAT, the collection of saturated sets of (untyped) λ-terms.
$X \subset \Lambda$ is saturated if

- $xP_1 \ldots P_n \in X$ (for all $x \in$ Var, $P_1, \ldots, P_n \in$ SN)
- $X \subseteq$ SN
- If $M[N/x]\vec{P} \in X$ and $N \in$ SN, then $(\lambda x.M)N\vec{P} \in X$.

Let ρ : TVar $\rightarrow$ SAT be a valuation of type variables.
Define the interpretation of types $[\![\sigma]\!]_\rho$ as follows.

- $[\![\alpha]\!]_\rho := \rho(\alpha)$
- $[\![\sigma{\rightarrow}\tau]\!]_\rho := \{M | \forall N \in [\![\sigma]\!]_\rho (MN \in [\![\tau]\!]_\rho)\}$
- $[\![\forall\alpha.\sigma]\!]_\rho := \cap_{X \in \mathsf{SAT}} [\![\sigma]\!]_{\rho,\alpha:=X}$

Proposition

$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \sigma \Rightarrow M[P_1/x_1, \ldots, P_n/x_n] \in [\![\sigma]\!]_\rho$
for all valuations ρ and $P_1 \in [\![\tau_1]\!]_\rho, \ldots, P_n \in [\![\tau_n]\!]_\rho$

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

Corollary $\lambda 2$ is SN

(Proof: take P_1 to be x_1, ..., P_n to be x_n.)

Types Summer School
Gothenburg Sweden August 2005

Type Systems
Herman Geuvers
Radboud University Nijmegen, NL

Lecture 2: Higher Order Logic and Type Theory

The original motivation of Church to introduce simple type theory was:

to define higher order (predicate) logic

In $\lambda\!\rightarrow$ we add the following

- prop as a basic type
- $\Rightarrow$: prop$\rightarrow$prop$\rightarrow$prop
- $\forall_\sigma$: $(\sigma\rightarrow$prop$)\rightarrow$prop (for each type σ)

This defines the language of higher order logic $\mathbf{HOL}$.

- Induction

$$\forall_{N\rightarrow\mathsf{prop}}(\ \lambda P{:}N\rightarrow\mathsf{prop}.(P\,0) \Rightarrow (\forall_N(\lambda x{:}N.(P x \Rightarrow P(S\,x))) \Rightarrow \forall_N(\lambda x{:}N.P x)))$$

Notation:

$$\forall P{:}N\rightarrow\mathsf{prop}(\ (P\,0) \Rightarrow (\forall x{:}N.(P x \Rightarrow P(S\,x))) \Rightarrow \forall x{:}N.P x)$$

- Higher order predicates/functions

transitive closure of a relation R

$$\lambda R{:}\ A\rightarrow A\rightarrow\mathsf{prop}.\lambda x,y{:}A.\ (\forall Q{:}A\rightarrow A\rightarrow\mathsf{prop}.(\mathsf{trans}(Q) \Rightarrow (R \subseteq Q) \Rightarrow Q\,x\,y))$$

of type

$$(A\rightarrow A\rightarrow\mathsf{prop})\rightarrow(A\rightarrow A\rightarrow\mathsf{prop})$$

Derivation rules for Higher Order Logic $\mathbf{HOL}$ (following Church)

- Natural deduction style.
- Rules are ‘on top’ of the simple type theory.
- Judgements are of the form

$$\Delta \vdash_\Gamma \varphi$$

  - $\Delta = \psi_1, \ldots, \psi_n$
  - Γ is a $\lambda\!\rightarrow$-context
  - $\Gamma \vdash \varphi$: prop, $\Gamma \vdash \psi_1$: prop,..., $\Gamma \vdash \psi_n$: prop
  - Γ is usually left implicit: $\Delta \vdash \varphi$

(axiom)	$\dfrac{}{\Delta \vdash \varphi}$	if $\varphi \in \Delta$
($\Rightarrow$ -introduction)	$\dfrac{\Delta \cup \varphi \vdash \psi}{\Delta \vdash \varphi \Rightarrow \psi}$	
($\Rightarrow$ -elimination)	$\dfrac{\Delta \vdash \varphi \Rightarrow \psi \quad \Delta \vdash \varphi}{\Delta \vdash \psi}$	
($\forall$-introduction)	$\dfrac{\Delta \vdash \varphi}{\Delta \vdash \forall x{:}\sigma.\varphi}$	if $x{:}\sigma \notin \mathsf{FV}(\Delta)$
($\forall$-elimination)	$\dfrac{\Delta \vdash \forall x{:}\sigma.\varphi}{\Delta \vdash \varphi[t/x]}$	if $t : \sigma$
(conversion)	$\dfrac{\Delta \vdash \varphi}{\Delta \vdash \psi}$	if $\varphi =_\beta \psi$

Church has additional things that we will not consider now:

- Negation connective with rules
- Classical logic

$$\frac{\Delta \vdash \neg\neg\varphi}{\Delta \vdash \varphi}$$

- Define other connectives in terms of $\Rightarrow, \forall, \neg$ (classically).
- Choice operator $\iota_\sigma : (\sigma \to \mathsf{prop}) \to \sigma$
- Rule for ι:

$$\frac{\Delta \vdash \exists! x{:}\sigma. P\, x}{\Delta \vdash P(\iota_\sigma P)}$$

This (Church' original higher order logic) is basically the logic of the theorem prover HOL (Gordon, Melham, Harrison) and of Isabelle-HOL (Paulson, Nipkow).
We will here restrict to the basic constructive core ($\forall, \Rightarrow$) of $\mathbf{HOL}$.

Important in $\mathbf{HOL}$:
Conversion rule:

$$\cfrac{\cfrac{\Delta \vdash \forall P{:}N \to \mathsf{prop}.(\ldots Pc \ldots)}{\Delta \vdash (\ldots (\lambda y{:}N.y > 0)c \ldots)}\ \forall\text{-elim}}{\Delta \vdash (\ldots c > 0 \ldots)}\ \text{conv}$$

Definability of other connectives (constructively):

$$\begin{aligned}
\bot &:= \forall \alpha{:}\mathsf{prop}.\alpha \\
\varphi \wedge \psi &:= \forall \alpha{:}\mathsf{prop}.(\varphi \Rightarrow \psi \Rightarrow \alpha) \Rightarrow \alpha \\
\varphi \vee \psi &:= \forall \alpha{:}\mathsf{prop}.(\varphi \Rightarrow \alpha) \Rightarrow (\psi \Rightarrow \alpha) \Rightarrow \alpha \\
\exists x{:}\sigma.\varphi &:= \forall \alpha{:}\mathsf{prop}.(\forall x{:}\sigma.\varphi \Rightarrow \alpha) \Rightarrow \alpha
\end{aligned}$$

Equality is definable in higher order logic:

> t and q terms are equal if they share the same properties (Leibniz equality)

Definition in $\mathbf{HOL}$ (for $t, q : A$):

$$t =_A q := \forall P{:}A \to \mathsf{prop}.(Pt \Rightarrow Pq)$$

- This equality is reflexive and transitive (easy)
- It is also symmetric(!) Trick: find a "smart" predicate P

Exercise: Prove reflexivity, transitivity and symmetry of $=_A$.

Exercise: Proof of symmetry of $=_A$.
(Trick: take $\lambda y{:}A.y =_A t$ for P.)

$$\dfrac{\dfrac{\dfrac{\Delta \vdash t =_A q}{\Delta \vdash \forall P{:}A\to\mathsf{prop}.(Pt \Rightarrow Pq)}}{\Delta \vdash (t =_A t) \Rightarrow (q =_A t)} \qquad \dfrac{\dots}{\Delta \vdash t =_A t}}{\Delta \vdash q =_A t}$$

One more exercise on Higher Order Logic
The transitive closure of a binary relation R on A has been defined as follows.

$$\begin{aligned}\mathsf{trclos}\, R := \lambda x, y{:}A.\\ (\forall Q{:}A\to A\to\mathsf{Prop}.(\mathsf{trans}(Q)\to(R \subseteq Q)\to(Q\,x\,y))).\end{aligned}$$

1. Prove that the transitive closure is transitive.
2. Prove that the transitive closure of R contains R.

(axiom)	$\dfrac{}{\Delta \vdash \varphi}$	if $\varphi \in \Delta$
($\Rightarrow$ -introduction)	$\dfrac{\Delta \cup \varphi \vdash \psi}{\Delta \vdash \varphi \Rightarrow \psi}$	
($\Rightarrow$ -elimination)	$\dfrac{\Delta \vdash \varphi \Rightarrow \psi \quad \Delta \vdash \varphi}{\Delta \vdash \psi}$	
($\forall$-introduction)	$\dfrac{\Delta \vdash \varphi}{\Delta \vdash \forall x{:}\sigma.\varphi}$	if $x{:}\sigma \notin \mathsf{FV}(\Delta)$
($\forall$-elimination)	$\dfrac{\Delta \vdash \forall x{:}\sigma.\varphi}{\Delta \vdash \varphi[t/x]}$	if $t : \sigma$
(conversion)	$\dfrac{\Delta \vdash \varphi}{\Delta \vdash \psi}$	if $\varphi =_\beta \psi$

Why not introduce a λ-term notation for the derivations?

This gives a type theory λHOL

- Let prop be a new 'universe' of propositional types.
- Direct encoding (deep embedding) of **HOL** into the type theory λHOL

(axiom) $\dfrac{}{\Delta \vdash_\Gamma x : \varphi}$ if $x{:}\varphi \in \Delta$

($\Rightarrow$ -introduction) $\dfrac{\Delta, x{:}\varphi \vdash_\Gamma M : \psi}{\Delta \vdash_\Gamma \lambda x{:}\varphi.M : \varphi \Rightarrow \psi}$

($\Rightarrow$ -elimination) $\dfrac{\Delta \vdash_\Gamma M : \varphi \Rightarrow \psi \quad \Delta \vdash_\Gamma N : \varphi}{\Delta \vdash_\Gamma M\,N : \psi}$

($\forall$-introduction) $\dfrac{\Delta \vdash_{\Gamma, x:\sigma} M : \varphi}{\Delta \vdash_\Gamma \lambda x{:}\sigma.M : \forall x{:}\sigma.\varphi}$ if $x{:}\sigma \notin$ FV(Δ)

($\forall$-elimination) $\dfrac{\Delta \vdash_\Gamma M : \forall x{:}\sigma.\varphi}{\Delta \vdash_\Gamma M\,t : \varphi[t/x]}$ if $\Gamma \vdash t : \sigma$

(conversion) $\dfrac{\Delta \vdash_\Gamma M : \varphi}{\Delta \vdash_\Gamma M : \psi}$ if $\varphi =_\beta \psi$

Now we have two 'levels' of type theories

- The (simple) type theory describing the language of **HOL**
- The type theory for the proof-terms of **HOL**

NB Many rules, many similar rules.

We put these levels together into one type theory λHOL.
Pseudoterms:

T ::= Prop | Type | Type$'$ | Var | (ΠVar:T.T) | (λVar:T.T) | TT

{Prop, Type, Type$'$} is the set of sorts, $\mathcal{S}$.

Some of the typing rules are parametrized

(axiom) $\vdash$ Prop : Type $\qquad \vdash$ Type : Type$'$

(var) $\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$ (weak) $\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$

(Π) $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$ if $(s_1, s_2) \in$ { (Type, Type), (Prop, Prop), (Type, Prop) }

(λ) $\dfrac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$

(app) $\dfrac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B[N/x]}$

(conv) $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$ if $A =_\beta B$

(Π) $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$ if $(s_1, s_2) \in$ { (Type, Type), (Prop, Prop), (Type, Prop) }

- The combination (Type,Type) forms the function types $A {\to} B$ for A, B:Type.
  This comprises the unary predicate types and binary relations types: $A {\to}$Prop and $A {\to} A {\to}$Prop.
  Also: higher order predicate types like $(A {\to} A {\to}$Prop$) {\to}$Prop.
  NB A Π-type formed by (Type,Type) is always an $\to$-type.
- (Prop,Prop) forms the propositional types $\varphi {\to} \psi$ for φ, ψ:Prop; implicational formulas.
  NB A Π-type formed by (Type,Type) is always an $\to$-type.
- (Type,Prop) forms the dependent propositional type $\Pi x{:}A.\varphi$ for A:Type, φ:Prop; universally quantified formulas.

Example: Deriving irreflexivity from anti-symmetry

$$\begin{aligned} \text{Rel} &:= \lambda X{:}\text{Type}.X \to X \to \text{Prop} \\ \text{AntiSym} &:= \lambda X{:}\text{Type}.\lambda R{:}(\text{Rel } X).\forall x, y{:}X.(Rxy) \Rightarrow (Ryx) \Rightarrow \bot \\ \text{Irrefl} &:= \lambda X{:}\text{Type}.\lambda R{:}(\text{Rel } X).\forall x{:}X.(Rxx) \Rightarrow \bot \end{aligned}$$

Derivation in $\mathbf{HOL}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\forall x^A y^A R\,x\,y \Rightarrow R\,y\,x \Rightarrow \bot}{\forall y^A R\,x\,y \Rightarrow R\,y\,x \Rightarrow \bot}}{R\,x\,x \Rightarrow R\,x\,x \Rightarrow \bot} \quad [R\,x\,x]}{R\,x\,x \Rightarrow \bot} \quad [R\,x\,x]}{\bot}}{R\,x\,x \Rightarrow \bot}}{\forall x^A.R\,x\,x \Rightarrow \bot}$$

Derivation in $\mathbf{HOL}$, with terms:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{z : \forall x^A y^A R\,x\,y \Rightarrow R\,y\,x \Rightarrow \bot}{zx : \forall y^A R\,x\,y \Rightarrow R\,y\,x \Rightarrow \bot}}{zxx : R\,x\,x \Rightarrow R\,x\,x \Rightarrow \bot} \quad [q : R\,x\,x]}{zxxq : R\,x\,x \Rightarrow \bot} \quad [q : R\,x\,x]}{zxxqq : \bot}}{\lambda q{:}(R\,x\,x).zxxqq : R\,x\,x \Rightarrow \bot}}{\lambda x{:}A.\lambda q{:}(R\,x\,x).zxxqq : \forall x^A.R\,x\,x \Rightarrow \bot}$$

Typing judgement in λHOL:

$$A{:}\text{Type}, R{:}A{\to}A{\to}\text{Prop},\ z : \Pi x, y{:}A.(R\,x\,y{\to}R\,y\,x{\to}\bot) \vdash \lambda x{:}A\lambda q{:}(R\,x\,x).z\,x\,x\,q\,q : (\Pi x{:}A.R\,x\,x{\to}\bot)$$

Question: is the type theory λHOL really isomorphic with $\mathbf{HOL}$?

Yes:Disambiguation Lemma Given

$$\Gamma \vdash M : T \text{ in } \lambda\text{HOL}$$

there is a permutation of Γ: $\Gamma_D, \Gamma_L, \Gamma_P$ such that

1. $\Gamma_D, \Gamma_L, \Gamma_P \vdash M : T$
2. Γ_D consists only of declarations A : Type
3. Γ_L consists only of declarations $x : \sigma$ with $\Gamma_D \vdash \sigma$: Type
4. Γ_P consists only of declarations $z : \varphi$ with $\Gamma_D, \Gamma_L \vdash \varphi$: Prop

So, if $\Gamma \vdash M : T$, we also have

$$\underbrace{A_1{:}\text{Type}, \ldots, A_n{:}\text{Type}}_{\substack{\Gamma_D \\ \text{domainvar.}}}, \underbrace{x{:}\sigma_1, \ldots, x_m{:}\sigma_m}_{\substack{\Gamma_L \\ \text{termvar.}}}, \underbrace{z_1{:}\varphi_1, \ldots z_p{:}\varphi_p}_{\substack{\Gamma_P \\ \text{proofvar.}}} \vdash M : T$$

Properties of λHOL.

- Uniqueness of types
  If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A{=_\beta}B$.
- Subject Reduction
  If $\Gamma \vdash M : A$ and $M \longrightarrow_\beta N$, then $\Gamma \vdash N : A$.
- Strong Normalization
  If $\Gamma \vdash M : A$, then all β-reductions from M terminate.

Proof of SN is a higher order extension of the one for $\lambda 2$ (using the saturated sets).

Decidability Questions:

$$\Gamma \vdash M : \sigma? \quad \text{TCP}$$
$$\Gamma \vdash M : ? \quad \text{TSP}$$
$$\Gamma \vdash ? : \sigma \quad \text{TIP}$$

For λHOL:

- TIP is undecidable
- TCP/TSP: simultaneously.

Type Checking

Define algorithms $\mathrm{Ok}(-)$ and $\mathrm{Type}_{_}(-)$ simultaneously:

- $\mathrm{Ok}(-)$ takes a context and returns 'true' or 'false'
- $\mathrm{Type}_{_}(-)$ takes a context and a term and returns a term or 'false'.

$$\mathrm{Ok}(<>) = \text{'true'}$$

$$\mathrm{Ok}(\Gamma, x{:}A) = \mathrm{Type}_\Gamma(A) \in \{\mathsf{Prop}, \mathsf{Type}\},$$

$$\mathrm{Type}_\Gamma(x) = \text{if } \mathrm{Ok}(\Gamma) \text{ and } x{:}A \in \Gamma \text{ then } A \text{ else 'false'},$$

$$\mathrm{Type}_\Gamma(\mathsf{Prop}) = \text{if } \mathrm{Ok}(\Gamma) \text{then } \mathsf{Type} \text{ else 'false'},$$

$$\mathrm{Type}_\Gamma(\mathsf{Type}) = \text{if } \mathrm{Ok}(\Gamma) \text{then } \mathsf{Type}' \text{ else 'false'},$$

$$\mathrm{Type}_\Gamma(\mathsf{Type}') = \text{'false'},$$

$$\begin{aligned}
\mathrm{Type}_\Gamma(MN) = {} & \text{if } \mathrm{Type}_\Gamma(M) = C \text{ and } \mathrm{Type}_\Gamma(N) = D \\
& \text{then} \quad \text{if } C \twoheadrightarrow_\beta \Pi x{:}A.B \text{ and } A =_\beta D \\
& \qquad\quad \text{then } B[N/x] \text{ else 'false'} \\
& \text{else} \quad \text{'false'},
\end{aligned}$$

$$\begin{aligned}
\mathrm{Type}_\Gamma(\lambda x{:}A.M) = {} & \text{if } \mathrm{Type}_{\Gamma,x:A}(M) = B \\
& \text{then} \qquad \text{if } \mathrm{Type}_\Gamma(\Pi x{:}A.B) \in \{\mathsf{Prop}, \mathsf{Type}\} \\
& \qquad\qquad \text{then } \Pi x{:}A.B \text{ else 'false'} \\
& \text{else 'false'},
\end{aligned}$$

$$\begin{aligned}
\mathrm{Type}_\Gamma(\Pi x{:}A.B) = {} & \text{if } \mathrm{Type}_\Gamma(A) = \mathsf{Type} \\
& \qquad \text{and } \mathrm{Type}_{\Gamma,x:A}(B) = s \in \{\mathsf{Prop}/\mathsf{Type}\} \\
& \text{then } s \text{ else} \\
& \text{if } \mathrm{Type}_\Gamma(A) = \mathsf{Prop} \text{ and } \mathrm{Type}_{\Gamma,x:A}(B) = \mathsf{Prop} \\
& \text{then } \mathsf{Prop} \text{ else 'false'}
\end{aligned}$$

Soundness

$$\text{Type}_\Gamma(M) = A \quad\Rightarrow\quad \Gamma \vdash M : A$$

for all Γ, M.

Completeness

$$\Gamma \vdash M : A \quad\Rightarrow\quad \text{Type}_\Gamma(M) =_\beta A$$

for all Γ, M and A.

This implies that, if $\text{Type}_\Gamma(M) =$ 'false', then M is not typable in Γ.

Completeness only makes sense if we have uniqueness of types
(Otherwise: let $\text{Type}_{_}(-)$ generate a set of possible types)

Termination

We want $\text{Type}_{_}(-)$ to terminate on all inputs.
(Not guaranteed by soundness and completeness)

Interesting case (1): application:

$$\begin{array}{rcl} \text{Type}_\Gamma(MN) & = & \text{if } \text{Type}_\Gamma(M) = C \text{ and } \text{Type}_\Gamma(N) = D \\ & & \text{then}\quad \text{if } C \twoheadrightarrow_\beta \Pi x{:}A.B \text{ and } A =_\beta D \\ & & \qquad\quad \text{then } B[N/x] \text{ else 'false'} \\ & & \text{else}\quad \text{'false'}, \end{array}$$

For this case, termination follows from the decidability of equality on well-typed terms (using SN and CR).

Termination

Interesting case(2): λ-abstraction:

$$\begin{array}{rcl} \text{Type}_\Gamma(\lambda x{:}A.M) & = & \text{if } \text{Type}_{\Gamma,x:A}(M) = B \\ & & \text{then}\qquad \text{if } \text{Type}_\Gamma(\Pi x{:}A.B) \in \{\text{Prop}, \text{Type}\} \\ & & \qquad\qquad \text{then } \Pi x{:}A.B \text{ else 'false'} \\ & & \text{else 'false'}, \end{array}$$

Replace the side condition

$$\text{Type}_\Gamma(\Pi x{:}A.B) \in \{\text{Prop}, \text{Type}\}$$

by

$\text{Type}_\Gamma(A) = \text{Prop}$ and $B \equiv \Pi\vec{y}{:}\vec{C}.D$ with $D \neq \text{Prop}/\text{Type}/\text{Type}'$

or

$\text{Type}_\Gamma(A) = \text{Type}$ and $B \equiv \Pi\vec{y}{:}\vec{C}.D$ with $D \neq \text{Type}/\text{Type}'$.

Types Summer School
Gothenburg Sweden August 2005

Type Systems
Herman Geuvers
Radboud University Nijmegen, NL

Lecture 3: Extensions of λHOL; the λ-cube; Pure Type Systems

(axiom) $\vdash$ Prop : Type $\qquad \vdash$ Type : Type$'$

(var) $\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$ (weak) $\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$

(Π) $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$ if $(s_1, s_2) \in$ { (Type, Type), (Prop, Prop), (Type, Prop) }

(λ) $\dfrac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$

(app) $\dfrac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$

(conv) $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$ if $A =_\beta B$

λHOL contains $\lambda 2$ and $\lambda\!\rightarrow$.

(Π) $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$ if $(s_1, s_2) \in$ { (Type, Type), (Prop, Prop), (Type, Prop) }

This rule allows to form

- $\rightarrow$-types on the Type-level (one copie of $\lambda\!\rightarrow$)
- $\rightarrow$-types on the Prop-level (second copie of $\lambda\!\rightarrow$)
- $\Pi\alpha$:Prop.$\alpha\!\rightarrow\!\alpha$: polymorphic types on the Prop-level (one copie of $\lambda 2$)

(Π) $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2}$ if $(s_1, s_2) \in$ { (Type, Type), (Prop, Prop), (Type, Prop) }

Why not extend λHOL to include

- Higher order logic over polymorphic domains?
  like ΠA : Type.$A\!\rightarrow\!A$
- Quantification over all domains?
  like in ΠA : Type.$\Pi P{:}A\!\rightarrow$Prop.$\Pi x{:}A.P\,x\!\rightarrow\!P\,x$

$$(\Pi)\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2} \text{ if } (s_1, s_2) \in \{\ (\text{Type}, \text{Type}), (\text{Prop}, \text{Prop}), (\text{Type}, \text{Prop})\ \}$$

Why not extend λHOL to include

- Higher order logic over polymorphic domains?
  like $\Pi A : \text{Type}. A \rightarrow A$
- Quantification over all domains?
  like in $\Pi A : \text{Type}. \Pi P{:}A \rightarrow \text{Prop}. \Pi x{:}A. P\, x \rightarrow P\, x$

This can easily be done by allowing in the Π-rule

- $(s_1, s_2) \in \{\ (\text{Type}', \text{Type})\ \}$ to obtain higher order logic over polymorphic domains $\rightsquigarrow$ system λU^-
- $(s_1, s_2) \in \{\ (\text{Type}', \text{Prop})\ \}$ to allow quantification over all domains $\rightsquigarrow$ system λU

Problem:

- λU (λHOL + (Type',Type) and (Type',Prop)) is inconsistent (Girard)
- λU^- (λHOL + (Type',Type)) is inconsistent (Coquand, Hurkens)

NB λHOL + (Type',Prop) is consistent.

Problem:

- λU (λHOL + (Type',Type) and (Type',Prop)) is inconsistent (Girard)
- λU^- (λHOL + (Type',Type)) is inconsistent (Coquand, Hurkens)

NB λHOL + (Type',Prop) is consistent.

Implications

- λU^- can't be used as a logic.
- In λU^-, there is a closed term M with $\vdash M : \bot$
- This M can not be in normal form (by some syntactic reasoning)
- So, λU^- is not SN

Type Checking in λU^- is still decidable:
All types (terms of type Prop, Type or Type$'$) are strongly normalizing

$$\begin{array}{rcl} \text{Type}_\Gamma(MN) & = & \text{if } \text{Type}_\Gamma(M) = C \text{ and } \text{Type}_\Gamma(N) = D \\ & & \text{then} \quad \text{if } C \twoheadrightarrow_\beta \Pi x{:}A.B \text{ and } A =_\beta D \\ & & \qquad\quad \text{then } B[N/x] \text{ else 'false'} \\ & & \text{else} \quad \text{'false'}, \end{array}$$

In the type synthesis algorithm we only check equality of types

Variations on the rules of λHOL:

- There are many type system with (slightly) different rules
- Many (proofs of) properties are similar
- Plan: Study these type systems in one general framework:
  - The cube of typed λ-calculi (Barendregt)
  - Pure Type Systems (Terlouw, Berardi)

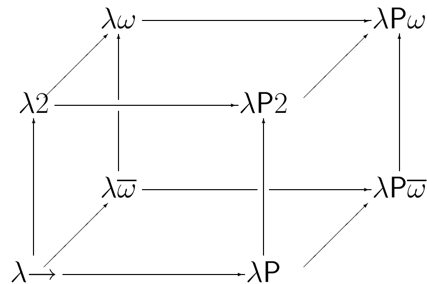
The cube of typed λ-calculi: (forget about Type$'$ for the moment)
Vary on all possible combinations for

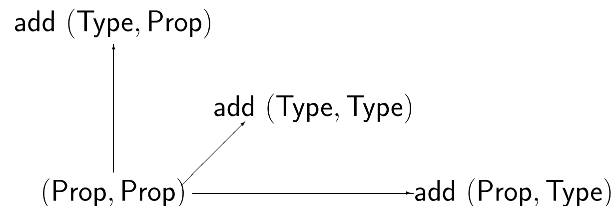
$\mathcal{R} \subseteq \{$ (Prop, Prop), (Type, Prop), (Type, Type), (Prop, Type) $\}$

in the Π-rule:

$$(\Pi)\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2} \text{ if } (s_1, s_2) \in \mathcal{R}$$

We take (Prop, Prop) in every $\mathcal{R}$





$$(\Pi)\ \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_2} \text{ if } (s_1, s_2) \in \mathcal{R}$$

System	$\mathcal{R}$			
$\lambda\!\rightarrow$	(Prop, Prop)			
$\lambda 2$ (system F)	(Prop, Prop)	(Type, Prop)		
λP (LF)	(Prop, Prop)		(Prop, Type)	
$\lambda\overline{\omega}$	(Prop, Prop)			(Type, Type)
λP2	(Prop, Prop)	(Type, Prop)	(Prop, Type)	
$\lambda\omega$ (system Fω)	(Prop, Prop)	(Type, Prop)		(Type, Type)
$\lambda P\overline{\omega}$	(Prop, Prop)		(Prop, Type)	(Type, Type)
λPω (CC)	(Prop, Prop)	(Type, Prop)	(Prop, Type)	(Type, Type)

$\lambda\!\rightarrow$ in this presentation is equivalent to $\lambda\!\rightarrow$ in the way we've presented before. Similarly for $\lambda 2$, λP, ...

This cube also gives a fine structure for the

Calculus of Constructions, CC (Coquand and Huet)

CC has:

- Polymorphic data types on the Prop-level, e.g. $\Pi\alpha{:}\mathsf{Prop}.\alpha{\to}(\alpha{\to}\alpha){\to}\alpha$.
- Predicate domains on the Type-level, e.g. $N{\to}N{\to}\mathsf{Prop}$
- Logic on the Prop-level, e.g. $\varphi \wedge \psi := \Pi\alpha{:}\mathsf{Prop}.(\varphi{\to}\psi{\to}\alpha){\to}\alpha$.
- Universal quantification (first and higher order), e.g. $\Pi P{:}N{\to}\mathsf{Prop}.\Pi x{:}N.Px{\to}Px$.

One can do higher order predicate logic in CC, in a slightly unusual way:

- 'propositions' and first order 'sets' are both of type Prop
- propositions and sets are completely mixed

Is it faithful to do higher order predicate logic in CC??

Answer: No!
There are non-provable formulas of **HOL** that become inhabited in CC

Consider extensionality of propostions:

$$\mathsf{EXT} := \forall\alpha,\beta{:}\mathsf{prop}.(\alpha \Leftrightarrow \beta) \Rightarrow (\alpha =_{\mathsf{prop}} \beta)$$

In CC, this becomes $\Pi\alpha,\beta{:}\mathsf{Prop}.(\alpha \leftrightarrow \beta) \to (\alpha =_{\mathsf{Prop}} \beta)$

Consider extensionality of propostions:

$$\mathsf{EXT} := \forall\alpha,\beta{:}\mathsf{prop}.(\alpha \Leftrightarrow \beta) \Rightarrow (\alpha =_{\mathsf{prop}} \beta)$$

In CC, this becomes $\Pi\alpha,\beta{:}\mathsf{Prop}.(\alpha \leftrightarrow \beta) \to (\alpha =_{\mathsf{Prop}} \beta)$

Suppose two base domains A and B and constants $a : A$, $b : B$.
In **HOL**, the following formulas are consistent.

- $\varphi := \forall x{:}A.x = a$, $\psi := \forall x{:}B.\exists y{:}B.x \neq y$

Consider extensionality of propostions:

$$\text{EXT} := \forall \alpha, \beta{:}\text{prop}.(\alpha \Leftrightarrow \beta) \Rightarrow (\alpha =_{\text{prop}} \beta)$$

In CC, this becomes $\Pi\alpha, \beta{:}\text{Prop}.(\alpha \leftrightarrow \beta) \rightarrow (\alpha =_{\text{Prop}} \beta)$

Suppose two base domains A and B and constants $a : A$, $b : B$.
In **HOL**, the following formulas are consistent.

- $\varphi := \forall x{:}A.x = a$, $\psi := \forall x{:}B.\exists y{:}B.x \neq y$

But in CC, EXT also applies to the base sets A and B.
$A \leftrightarrow B$ (both are non-empty) so $A =_{\text{Prop}} B$
so property ψ (of B) also applies to A
so $\forall x{:}A.\exists y{:}A.x \neq y$
contradicting φ
So, in CC, φ and ψ are inconsistent

We have to be careful when doing higher order logic in CC.

Or: we may try to improve on this: taking the sets and the propositions apart:
System λPREDω:

- Sorts: Prop, Set, Type^p, Type^s
- Axioms for these sorts: Prop : Type^p, Set : Type^s
- Rules R:
  - (Prop, Prop): implication
  - (Set, Prop): first order quantification
  - (Type^p, Prop): higher order quantification
  - (Set, Set): function types
  - (Set, Type^p): predicate types
  - (Type^p, Type^p): higher order types

Pure Type Systems
Determined by a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ with

- $\mathcal{S}$ the set of sorts
- $\mathcal{A}$ the set of axioms, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$
- $\mathcal{R}$ the set of rules, $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$

If $s_2 = s_3$ in $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$.

pseudoterms:

$$\mathsf{T} ::= \mathcal{S} \mid \mathsf{Var} \mid (\Pi\mathsf{Var}{:}\mathsf{T}.\mathsf{T}) \mid (\lambda\mathsf{Var}{:}\mathsf{T}.\mathsf{T}) \mid \mathsf{TT}.$$

(sort) $\vdash s_1 : s_2$ if $(s_1, s_2) \in \mathcal{A}$ (var) $\dfrac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A}$ if $x \notin \Gamma$

(weak) $\dfrac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C}$ if $x \notin \Gamma$

(Π) $\dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3}$ if $(s_1, s_2, s_3) \in \mathcal{R}$

(λ) $\dfrac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$

(app) $\dfrac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$

(conv$_\beta$) $\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}$ $A =_\beta B$

Examples of PTSs

λPREDω	
$\mathcal{S}$	Set, Type^s, Prop, Type
$\mathcal{A}$	Set : Type^s, Prop : Type
$\mathcal{R}$	(Set, Set), (Set, Type), (Type, Type), (Prop, Prop), (Set, Prop), (Type, Prop)

CC	
$\mathcal{S}$	Prop, Type
$\mathcal{A}$	Prop : Type
$\mathcal{R}$	(Prop, Prop), (Prop, Type), (Type, Prop), (Type, Type)

λHOL	
$\mathcal{S}$	Prop, Type, Type$'$
$\mathcal{A}$	Prop : Type, Type : Type$'$
$\mathcal{R}$	(Prop, Prop), (Type, Type), (Type, Prop)

λU	
$\mathcal{S}$	Prop, Type, Type$'$
$\mathcal{A}$	Prop : Type, Type : Type$'$
$\mathcal{R}$	(Prop, Prop), (Type, Type), (Type$'$, Type), (Type$'$, Prop), (Type, Prop)

$\lambda\star$	
$\mathcal{S}$	$\star$
$\mathcal{A}$	$\star : \star$
$\mathcal{R}$	$(\star, \star)$

A PTS-morphism from $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ to $\lambda(\mathcal{S}', \mathcal{A}', \mathcal{R}')$ is an $f : \mathcal{S} \to \mathcal{S}'$ that preserves the axioms and rules:

- if $(s_1, s_2) \in \mathcal{A}$ then $(f(s_1), f(s_2)) \in \mathcal{A}'$
- if $(s_1, s_2, s_3) \in \mathcal{R}$ then $(f(s_1), f(s_2), f(s_3)) \in \mathcal{R}'$

f extends to pseudoterms and contexts
Proposition:

If $\Gamma \vdash M : A$ then $f(\Gamma) \vdash f(M) : f(A)$

Examples:

- $f : \lambda(\mathcal{S}, \mathcal{A}, \mathcal{R}) \to \lambda\star$, $f(s) := \star$. ("initial" PTS)
- g : λPRED$\omega \to$ CC, g(Prop) $= g$(Set) $:=$ Prop, g(Type^p) $= g$(Type^s) $:=$ Type.

Corollary: SN for CC $\Rightarrow$ SN for λPREDω

There are now two type systems for higher order predicate logic: λPREDω and λHOL.

λPREDω	
$\mathcal{S}$	Set, Type^s, Prop, Type
$\mathcal{A}$	Set : Type^s, Prop : Type
$\mathcal{R}$	(Set, Set), (Set, Type), (Type, Type), (Prop, Prop), (Set, Prop), (Type, Prop)

λHOL	
$\mathcal{S}$	Prop, Type, Type$'$
$\mathcal{A}$	Prop : Type, Type : Type$'$
$\mathcal{R}$	(Prop, Prop), (Type, Type), (Type, Prop)

They are equivalent:
The PTS-morphism h : λPRED$\omega \to \lambda$HOL, given by

$$\begin{array}{llll} h(\text{Prop}) & := \text{Prop} & h(\text{Set}) & := \text{Type} \\ h(\text{Type}^p) & := \text{Type} & h(\text{Type}^s) & := \text{Type}' \end{array}$$

constitutes an isomorphism between the derivable sequents.

CC^∞	
$\mathcal{S}$	Prop, $\{\text{Type}_i\}_{i \in \mathbb{N}}$
$\mathcal{A}$	Prop : Type, $\text{Type}_i : \text{Type}_{i+1}$
$\mathcal{R}$	(Prop, Prop), (Prop, Type_i), (Type_i, Prop)
	(Type_i, Type_j, $\text{Type}_{\max(i,j)}$)

NB: ($\text{Type}_{i+1}\text{Type}_i$, Type_i)) would be inconsistent.

CC^∞	
$\mathcal{S}$	Prop, $\{\text{Type}_i\}_{i \in \mathbb{N}}$
$\mathcal{A}$	Prop : Type, $\text{Type}_i : \text{Type}_{i+1}$
$\mathcal{R}$	(Prop, Prop), (Prop, Type_i), (Type_i, Prop)
	(Type_i, Type_j, $\text{Type}_{\max(i,j)}$)

NB: ($\text{Type}_{i+1}\text{Type}_i$, Type_i)) would be inconsistent.
The Extended Calculus of Constructions has in addition

- Cumulativity: Prop $\subseteq \text{Type}_0 \subseteq \text{Type}_1 \subseteq \ldots$
- Σ-types:

$$\frac{\Gamma \vdash A : \text{Prop} \quad \Gamma, x{:}A \vdash B : \text{Prop}}{\Gamma \vdash \Sigma x{:}A.B : \text{Prop}} \qquad \frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x{:}A \vdash B : \text{Type}_j}{\Gamma \vdash \Sigma x{:}A.B : \text{Type}_{\max(i,j)}}$$

NB: We have $\Pi A{:}\text{Type}_i.\varphi$: Prop, but not $\Sigma A{:}\text{Type}_i.\varphi$: Prop
NB: Coq has in addition Set : Type and rules (Set, Set), (Type_i, Set), (Set, Prop) and inductive types.

What is the use of the abstract framework of PTSs?

- Present (the kernel of) systems in a uniform way
- Compare systems (e.g. λHOL, λPREDω, CC) within one framework
- Prove properties for many systems at once.

Properties of PTSs.

- Uniqueness of types
  If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.
  Holds if $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subseteq (\mathcal{S} \times \mathcal{S}) \times \mathcal{S}$ are functions.
- Subject Reduction
  If $\Gamma \vdash M : A$ and $M \longrightarrow_\beta N$, then $\Gamma \vdash N : A$.
- Substitution property
  If $\Gamma, x : B, \Delta \vdash M : A$, $\Gamma \vdash P : B$, then
  $\Gamma, \Delta[P/x] \vdash M[P/x] : A[P/x]$.
- Thinning
  If $\Gamma \vdash M : A$ and $\Gamma \subseteq \Delta$, Δ well-formed, then $\Delta \vdash M : A$.
- Strengthening
  If $\Gamma, x : \tau, \Delta \vdash M : A$ and $x \notin \text{FV}(M, A, \Delta)$, then
  $\Gamma, \Delta \vdash M : A$.

Strong Normalization:

If $\Gamma \vdash M : A$, then all β-reductions from M terminate.

SN holds for some PTSs (all subsystems of CC,...), and for some not (λU^-, $\lambda\star$, ...).

SN for CC can be proved by a higher order extension of the saturated sets argument (for $\lambda 2$).

Types Summer School
Gothenburg Sweden August 2005

Formalising Mathematics in Type Theory

Herman Geuvers
Radboud University Nijmegen, NL

Dogma of Type Theory

- Everything has a type

$$M : A$$

- Types are a bit like sets, but: . . .
  - types give "syntactic information"

$$3 + (7 * 8)^5 : \text{nat}$$

  - sets give "semantic information"

$$3 \in \{n \in \mathbb{N} \mid \forall x, y, z > 0 (x^n + y^n \neq z^n)\}$$

Per Martin-Löf:
A type comes with
construction principles: how to build objects of that type? and
elimination principles: what can you do with an object of that type?

This fits well with the Brouwerian view of mathematics:

"there exists an x" means
"we have a method of constructing x"

In short: a type is characterised by the construction principles for its objects.

Examples

- A summer school is constructed from students, teachers, a team of good organisers and good weather.
- A phrase is constructed from a noun and a verb or from two phrases with the word "and" between them.
  So any phrase has the shape
  "noun verb and noun verb and . . . and noun and verb".
- A natural number is either 0 or the successor S applied to a natural number.
  So the natural numbers are the objects of the shape $S(\ldots S(0) \ldots)$.

Note:

Checking whether an object belongs to an alleged type is decidable!

But if type checking should be decidable, there is not much information one can encode in a type (?)

$$X := \{n \in \mathbb{N} \mid \forall x, y, z > 0(x^n + y^n \neq z^n)\}$$

is X a type?
The proper question is: what are the objects of X? (How does one construct them?)

One constructs an object of the type X by giving an $N \in \mathbb{N}$ and a proof of the fact that $\forall x, y, z > 0(x^N + y^N \neq z^N)$.

The type X consists of pairs $\langle N, p\rangle$, with

- $N \in \mathbb{N}$
- p a proof of $\forall x, y, z > 0(x^N + y^N \neq z^N)$

$\langle N, p\rangle : X$ is decidable (if proof-checking is decidable).

More technically.
(Especially related to the type theory of Coq, but more widely applicable.)

- A data type (or set) is a term A : Set
- A formula is a term φ : Prop
- An object is a term $t : A$ for some A : Set
- A proof is a term $p : \varphi$ for some φ : Prop.
- Set and Prop are both "universes" or "sorts".

Slogan: (Curry-Howard isomorphism)

Propositions as Types
Proofs as Terms

Judgement

$$\Gamma \vdash M : U$$

- Γ is a context
- M is a term
- U is a type

Two readings

- M is an object (expression) of data type U (if U : Set)
- M is a proof (deduction) of proposition U (if U : Prop)

Γ contains

- variable declarations $x : T$
  - $x : A$ with A : Set $\rightsquigarrow$ 'declaring x in A'
  - $x : \varphi$ with φ : Prop $\rightsquigarrow$ 'assuming φ' (axiom)
- definitions $x := M : T$
  - $x := t : A$ with A : Set $\rightsquigarrow$ 'defining x as the expression t'
  - $x := p : \varphi$ with φ : Prop $\rightsquigarrow$ 'defining x as the proof p of φ'
    ($\simeq$ declaring x as a "reference" to the lemma φ)

Type theory as a basis for theorem proving

- Interactive theorem proving = interactive term construction
  Proving φ = (interactively) constructing a *proof term* $p : \varphi$
- Proof checking = Type checking
  Type checking is decidable and hence proof checking is.

NB Proof terms are first class citizens.

Type theory as a basis for theorem proving

- Interactive theorem proving = interactive term construction
  Proving φ = (interactively) constructing a *proof term* $p : \varphi$
- Proof checking = Type checking
  Type checking is decidable and hence proof checking is.

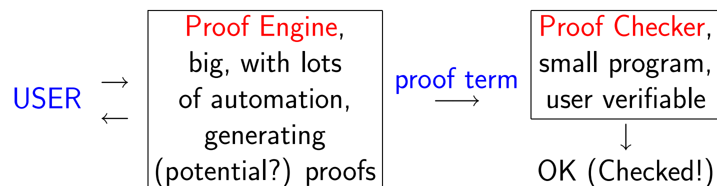
Decidability problems:

$\Gamma \vdash M : A?$	Type Checking Problem	TCP
$\Gamma \vdash M : ?$	Type Synthesis Problem	TSP
$\Gamma \vdash ? : A$	Type Inhabitation Problem	TIP

TCP and TSP are decidable
TIP is undecidable

De Bruijn criterion for theorem provers / proof checkers:
How to check the checker?

Interactive Theorem Prover:



A TP satisfies the De Bruijn criterion if a small, 'easily' verifiable, independent proof checker can be written.

How proof terms occur (in Coq):

```
Lemma trivial : forall x:A, P x -> P x.
intros x H.
exact H.
Qed.
```

- Using the tactic script a term of type `forall x:A, P x -> P x` has been created.
- Using `Qed`, `trivial` is defined as this term and added to the global context.

...

Computation

- (β):

$$(\lambda x{:}A.M)N \to_\beta M[N/x]$$

- (ι): primitive recursion reduction rules (later)
- (δ): definition unfolding: if $x := t : A \in \Gamma$, then

$$M(x) \to_\delta M(t)$$

- Transitive, reflexive, symmetric closure: $=_{\beta\iota\delta}$

NB: Types that are equal modulo $=_{\beta\iota\delta}$ have the same inhabitants (definitional equality):

$$\text{(conversion)}\ \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B}\ A =_{\beta\iota\delta} B$$

This is also called the Poincaré principle:
"(computational) equalities do not require a proof"

The Poincaré principle says that if $x : A(n) \to B$ and $y : A(f\,m)$, then

$$x\,y : B \text{ iff } f\,m = n.$$

But: type checking should be decidable, so $f\,m = n$ should be decidable.

So: the definable functions in our type theory must be restricted: all computations should terminate.

Data types and executable programs in type theory
Data types:

```
Inductive nat : Set :=
    0 : nat
  | S : nat->nat.
```

This definition yields

- The constructors 0 and S
- Induction principle:
  nat_ind : $\forall P{:}\text{nat}{\to}\text{Prop}.(P\,0) \to (\forall n{:}\text{nat}.(P\,n){\to}(P(S\,n))) \to \forall n{:}\text{nat}(P\,n)$
- Recursion scheme (primitive recursion over higher types)

Example of the recursion scheme (1 abbreviates (S 0) etc.)

```
Fixpoint nfib (n:nat) :nat :=
match n with
 | 0     => 1
 | S m => match m with
             | 0     => 1
             | S p   =>  nfib p + nfib m
            end
end.
```

NB: Recursive calls should be 'smaller' (according to some rather general syntactic measure)

- Coq includes a (small, functional) programming language in which executable functions can be written.

. . .

Dependently typed data types: vectors of length n over A

```
Inductive vect (A:Set) : nat -> Set :=
    |  nnil   : vect A 0
    | ccons : forall (n:nat)(a:A), vect A n -> vect A (S n).
```

Now define, for example,

- head : forall (A:Set)(n:nat), vect A (S n) $\rightarrow$ A
- tail : forall (A:Set)(n:nat), vect A (S n) $\rightarrow$ vect A n

...

Let the type checker do the work for you!
Implicit Syntax
If the type checker can infer some arguments, we can leave them out:

Write $f __ a\, b$ in stead of $f\, S\, T\, a\, b$ if
$f : \Pi S, T{:}\mathsf{Set}. S \rightarrow T \rightarrow T$

Also: define $F := f __$ and write $F\, a\, b$.

...

Inductive types are also used to define the logical connectives:
(Notation: `A\/B` denotes `or A B` etc.)
Inductive or (A : Prop)(B : Prop) : Prop :=
    `or_introl` : A $\rightarrow$ A\/B |
    `or_intror` : B $\rightarrow$ A\/B.
Inductive and (A : Prop)(B : Prop) : Prop :=
    `conj` : A $\rightarrow$ B $\rightarrow$ A/\B.

Inductive ex (A : Set)(P : A$\rightarrow$Prop) : Prop :=
    `ex_intro` : (x:A)(P x) $\rightarrow$ (Ex P).
Inductive True : Prop := I : True.
Inductive False : Prop := .
All (constructive) logical rules are now derivable.

...

Proof terms in intensional type theory

- The 'subtype' $\{t : A \mid (P\ t)\}$ is defined as the type of pairs $\langle t, p\rangle$ where $t : A$ and $p : (P\ t)$.
- A partial function is a function on a subtype
  E.g. $(-)^{-1} : \{x{:}\mathbb{R} \mid x \neq 0\} \rightarrow \mathbb{R}$.
  If $x : \mathbb{R}$ and $p : x \neq 0$, then $\frac{1}{\langle x,p\rangle} : \mathbb{R}$.
- Usually we only consider partial functions that are proof-irrelevant, i.e.
  if $p : t \neq 0$ and $q : t \neq 0$, then $\frac{1}{\langle t,p\rangle} = \frac{1}{\langle t,q\rangle}$.

Use Σ-types for mathematical structures:
theory of groups: Given A : Type, a group over A is a tuple consisting of

$$\begin{aligned} \circ &: A \to A \to A \\ e &: A \\ \mathsf{inv} &: A \to A \end{aligned}$$

such that the following types are inhabited.

$$\begin{aligned} \forall x, y, z{:}A.(x \circ y) \circ z &= x \circ (y \circ z), \\ \forall x{:}A.e \circ x &= x, \\ \forall x{:}A.(\mathsf{inv}\ x) \circ x &= e. \end{aligned}$$

Type of group-structures over A, Group-Str(A), is

$$(A \to A \to A) \times (A \times (A \to A))$$

The type of groups over A, Group(A), is

$$\begin{aligned} \mathsf{Group}(A) := \Sigma \circ{:}A \to A \to A.\Sigma e{:}A.\Sigma \mathsf{inv}{:}A \to A. \\ (\forall x, y, z{:}A.(x \circ y) \circ z = x \circ (y \circ z)) \wedge \\ (\forall x{:}A.e \circ x = x) \wedge \\ (\forall x{:}A.(\mathsf{inv}\ x) \circ x = e). \end{aligned}$$

If t : Group(A), we can extract the elements of the group structure by projections: $\pi_1 t : A \to A \to A$, $\pi_1(\pi_2 t) : A$
If $f : A \to A \to A$, $a : A$ and $h : A \to A$ with p_1, p_2 and p_3 proof-terms of the associated group-axioms, then

$$\langle f, \langle a, \langle h, \langle p_1, \langle p_2, p_3 \rangle\rangle\rangle\rangle\rangle : \mathsf{Group}(A).$$

We would like to use names for the projections:
Coq has labelled record types (type dependent)

•

```
Record My_type : Set :=
  { l_1   : type_1 ;
    l_2   : type_2 ;
    l_3   : type_3   }.
```

If `X : My_type`, then `(l_1 X) : type_1`.
- Basically, `My_type` consists of labelled tuples:
  `[l_1:= value_1, l_2:=value_2, l_3:=value_3]`
- Also with dependent types: `l_1` may occur in `type_2`.
  If `X : My_type`, then

  (l_2 X) : type_2 [(l_1 X)/l_1]

•

```
Record Group : Type :=
  { crr    :  Set;
    op     :  crr -> crr -> crr;
    unit   :  crr;
    inv    :  crr -> crr;
    assoc  :  (x,y,z:crr)
              (op (op x y) z) = (op x (op y z))
    ...       ...
 }.
```

If `X : Group`, then `(op X) : (crr X) -> (crr X) -> (crr X)`.

The record types can be defined in Coq using inductive types.
Note: `Group` is in `Type` and not in `Set`

Let the checker infer even more for you! Coercions

- The user can tell the type checker to use specific terms as coercions.
  `Coercion k : A >-> B` declares the term `k : A -> B` as a coercion.
  - If `f a` can not be typed, the type checker will try to type check `(k f) a` and `f (k a)`.
  - If we declare a variable `x:A` and `A` is not a type, the type checker will check if `(k A)` is a type.

  Coercions can be composed.

Coercions and structures

```
Record CMonoid : Type :=
  { m_crr    :> CSemi_grp;
    m_proof :  (Commutative m_crr (sg_op m_crr))
        /\ (IsUnit m_crr (sg_unit m_crr) (sg_op m_crr))
  }.
```

- A monoid is now a tuple $\langle\langle\langle S, =_S, r\rangle, a, f, p\rangle, q\rangle$
  If `M : Monoid`, the carrier of `M` is `(crr(sg_crr(m_crr M)))`
  Nasty !!
  $\Rightarrow$ We want to use the structure `M` as synonym for the carrier set `(crr(sg_crr(m_crr M)))`.
  $\Rightarrow$ The maps `crr`, `sg_crr`, `m_crr` should be left implicit.
- The notation `m_crr :> Semi_grp` declares the coercion `m_crr : Monoid >-> Semi_grp`.

Functions and Algorithms

- Set theory (and logic): a function $f : A{\rightarrow}B$ is a relation $R \subset A \times B$ such that $\forall x{:}A.\exists! y{:}B.R\,x\,y$.
  "functions as graphs"
- In Type theory, we have functions-as-graphs ($R : A{\rightarrow}B{\rightarrow}$Prop), but also functions-as-algorithms: $f : A{\rightarrow}B$.

Functions as algorithms also compute: β and ι rules:

$$\begin{aligned}
(\lambda x{:}A.M)N &\longrightarrow_\beta M[N/x],\\
\text{Rec}\,b\,f\,0 &\longrightarrow_\iota b,\\
\text{Rec}\,b\,f\,(S\,x) &\longrightarrow_\iota f\,x\,(\text{Rec}\,b\,f\,x).
\end{aligned}$$

Terms of type $A{\rightarrow}B$ denote algorithms, whose operational semantics is given by the reduction rules.
(Type theory as a small programming language)

Intensionality versus Extensionality

The equality in the side condition in the (conversion) rule can be intensional or extensional.

Extensional equality requires the following rules:

$$(\text{ext})\quad \frac{\Gamma \vdash M, N : A{\rightarrow}B \quad \Gamma \vdash p : \Pi x{:}A.(M x = N x)}{\Gamma \vdash M = N : A{\rightarrow}B}$$

$$(\text{conv})\quad \frac{\Gamma \vdash P : A \quad \Gamma \vdash A = B : s}{\Gamma \vdash P : B}$$

- Intensional equality of functions = equality of algorithms (the way the function is presented to us (syntax))
- Extensional equality of functions = equality of graphs (the (set-theoretic) meaning of the function (semantics))

Adding the rule (ext) renders TCP undecidable:

Suppose $H : (A{\rightarrow}B){\rightarrow}\mathrm{Prop}$ and $x : (H\ f)$; then

$$x : (H\ g) \text{ iff there is a } p : \Pi x{:}A.f\ x = g\ x$$

So, to solve this TCP, we need to solve a TIP.

The interactive theorem prover Nuprl is based on extensional type theory.

Setoids

How to represent the notion of set?

Note: A set is not just a type, because $M : A$ is decidable whereas $t \in X$ is undecidable

A setoid is a pair $[A, =]$ with

- A : Set,
- $= : A{\rightarrow}(A{\rightarrow}\mathrm{Prop})$ an equivalence relation over A

Function space setoid (the setoid of setoid functions) $[A \stackrel{s}{\rightarrow} B, =_{A \stackrel{s}{\rightarrow} B}]$ is defined by

$$A \stackrel{s}{\rightarrow} B := \Sigma f{:}A{\rightarrow}B.(\Pi x, y{:}A.(x =_A y){\rightarrow}((f\ x) =_B\ (f\ y))),$$
$$f =_{A \stackrel{s}{\rightarrow} B} g := \Pi x, y{:}A.(x =_A y){\rightarrow}(\pi_1\ f\ x) =_B (\pi_1\ g\ y).$$

Two mathematical constructions: quotient and subset for setoids.

Q is an equivalence relation over the setoid $[A, =_A]$ if

- $Q : A{\rightarrow}(A{\rightarrow}\mathrm{Prop})$ is an equivalence relation,
- $=_A \subset Q$, i.e. $\forall x, y{:}A.(x =_A y){\rightarrow}(Q\ x\ y)$.

The quotient setoid $[A, =_A]/Q$ is defined as

$$[A, Q]$$

Easy exercise:
If the setoid function $f : [A, =_A] \rightarrow [B, =_B]$ respects Q
(i.e. $\forall x, y{:}A.(Q\ x\ y){\rightarrow}((f\ x) =_B (f\ y))$)
it induces a setoid function from $[A, =_A]/Q$ to $[B, =_B]$.

Given $[A, =_A]$ and predicate P on A define the sub-setoid

$$[A, =_A] \mid P \ := \ [\Sigma x{:}A.(P\ x), =_A|P]$$

$=_A|P$ is $=_A$ restricted to P: for $q, r : \Sigma x{:}A.(P\ x)$,

$$q\ (=_A|P)\ r := (\pi_1\ q) =_A (\pi_1\ r)$$

Proof-irrelevance is "embedded" in the subsetoid construction:

Setoid functions are proof-irrelevant.

Bengt Nordström:

Types, propositions and problems

Martin-Löf's Type Theory

B. Nordström, K. Petersson and J. M. Smith

[1]

Contents

1 Introduction 1
  1.1 Different formulations of type theory 3
  1.2 Implementations 4
2 Propositions as sets 4
3 Semantics and formal rules 7
  3.1 Types 7
  3.2 Hypothetical judgements 9
  3.3 Function types 12
  3.4 The type Set 14
  3.5 Definitions 15
4 Propositional logic 16
5 Set theory 20
  5.1 The set of Boolean values 21
  5.2 The empty set 21
  5.3 The set of natural numbers 22
  5.4 The set of functions (Cartesian product of a family of sets) 24
  5.5 Propositional equality 27
  5.6 The set of lists 29
  5.7 Disjoint union of two sets 29
  5.8 Disjoint union of a family of sets 30
  5.9 The set of small sets 31
6 ALF, an interactive editor for type theory 33

1 Introduction

The type theory described in this chapter has been developed by Martin-Löf with the original aim of being a clarification of constructive mathematics. Unlike most other formalizations of mathematics, type theory is not based on predicate logic. Instead, the logical constants are interpreted within type

[1]This is a chapter from Handbook of Logic in Computer Science, Vol 5, Oxford University Press, October 2000

theory through the Curry-Howard correspondence between propositions and sets [10, 22]: a proposition is interpreted as a set whose elements represent the proofs of the proposition.

It is also possible to view a set as a problem description in a way similar to Kolmogorov's explanation of the intuitionistic propositional calculus [25]. In particular, a set can be seen as a specification of a programming problem; the elements of the set are then the programs that satisfy the specification.

An advantage of using type theory for program construction is that it is possible to express both specifications and programs within the same formalism. Furthermore, the proof rules can be used to derive a correct program from a specification as well as to verify that a given program has a certain property. As a programming language, type theory is similar to typed functional languages such as ML [19, 32] and Haskell [23], but a major difference is that the evaluation of a well-typed program always terminates.

The notion of constructive proof is closely related to the notion of computer program. To prove a proposition $(\forall x \in A)(\exists y \in B)P(x, y)$ constructively means to give a function f which when applied to an element a in A gives an element b in B such that $P(a, b)$ holds. So if the proposition $(\forall x \in A)(\exists y \in B)P(x, y)$ expresses a specification, then the function f obtained from the proof is a program satisfying the specification. A constructive proof could therefore itself be seen as a computer program and the process of computing the value of a program corresponds to the process of normalizing a proof. It is by this computational content of a constructive proof that type theory can be used as a programming language; and since the program is obtained from a proof of its specification, type theory can be used as a programming logic. The relevance of constructive mathematics for computer science was pointed out already by Bishop [4].

Recently, several implementations of type theory have been made which can serve as logical frameworks, that is, different theories can be directly expressed in the implementations. The formulation of type theory we will describe in this chapter form the basis for such a framework, which we will briefly present in the last section.

The chapter is structured as follows. First we will give a short overview of different formulations and implementations of type theory. Section 2 will explain the fundamental idea of propositions as sets by Heyting's explanation of the intuitionistic meaning of the logical constants. The following section will give a rather detailed description of the basic rules and their semantics; on a first reading some of this material may just be glanced at, in particular the subsection on hypothetical judgements. In section 4 we illustrate type theory as a logical framework by expressing propositional logic in it. Section 5 introduces a number of different sets and the final section give a short description of ALF, an implementation of the type theory

of this chapter.

Although self-contained, this chapter can be seen as complement to our book, *Programming in Type Theory. An Introduction* [33], in that we here give a presentation of Martin-Löf's monomorphic type theory in which there are two basic levels, that of types and that of sets. The book is mainly concerned with a polymorphic formulation where instead of a level of types there is a theory of expressions. One major difference between these two formulations is that in the monomorphic formulation there is more type information in the terms, which makes it possible to implement a type checker [27]; this is important when type theory is used as a logical framework where type checking is the same as proof checking.

1.1 Different formulations of type theory

One of the basic ideas behind Martin-Löf's type theory is the Curry-Howard interpretation of propositions as types, that is, in our terminology, propositions as sets. This view of propositions is closely related to Heyting's explanation of intuitionistic logic [21] and will be explained in detail below.

Another source for type theory is proof theory. Using the identification of propositions and sets, normalizing a derivation corresponds to computing the value of the proof term expressing the derivation. One of Martin-Löf's original aims with type theory was that it could serve as a framework in which other theories could be interpreted. And a normalization proof for type theory would then immediately give normalization for a theory expressed in type theory.

In Martin-Löf's first formulation of type theory from 1971 [28], theories like first order arithmetic, Gödel's T [18], second order logic and simple type theory [5] could easily be interpreted. However, this formulation contained a reflection principle expressed by a universe V and including the axiom $\mathsf{V} \in \mathsf{V}$, which was shown by Girard to be inconsistent. Coquand and Huet's Calculus of Constructions [8] is closely related to the type theory in [28]: instead of having a universe V, they have the two types Prop and Type and the axiom $\mathsf{Prop} \in \mathsf{Type}$, thereby avoiding Girard's paradox.

Martin-Löf's later formulations of type theory have all been predicative; in particular second order logic and simple type theory cannot be interpreted in them. The strength of the theory considered in this chapter instead comes from the possibility of defining sets by induction.

The formulation of type theory from 1979 in *Constructive Mathematics and Computer Programming* [30] is polymorphic and extensional. One important difference with the earlier treatments of type theory is that normalization is not obtained by metamathematical reasoning; instead, a direct semantics is given, based on Tait's computability method. A consequence

of the semantics is that a term, which is an element in a set, can be computed to normal form. For the semantics of this theory, lazy evaluation is essential. Because of a strong elimination rule for the set expressing the propositional equality, judgemental equality is not decidable. This theory is also the one in *Intuitionistic Type Theory* [31]. It is also the theory used in the Nuprl system [6] and by the group in Groningen [3].

The type theory presented in this chapter was put forward by Martin-Löf in 1986 with the specific intention that it should serve as a logical framework.

1.2 Implementations

One major application of type theory is to use it as a programming logic in which you derive programs from specifications. Such derivations easily become long and tedious and, hence, error prone; so, it is essential to formalize the proofs and to have computerized tools to check them.

There are several examples of computer implementations of proof checkers for formal logics. An early example is the AUTOMATH system [11, 12] which was designed by de Bruijn to check proofs of mathematical theorems. Quite large proofs were checked by the system, for example the proofs in Landau's book *Grundlagen der Analysis* [24]. Another system, which is more intended as a proof assistant, is the Edinburgh (Cambridge) LCF system [19, 34]. The proofs are constructed in a goal directed fashion, starting from the proposition the user wants to prove and then using tactics to divide it into simpler propositions. The LCF system also introduced the notion of metalanguage (ML) in which the user could implement her own proof strategies. Based on the LCF system, a system for Martin-Löf's type theory was implemented in Göteborg 1982 [35]. Another, more advanced, system for type theory was developed by Constable et al at Cornell University [6].

During the last years, several logical frameworks based on type theory have been implemented: the Edinburgh LF [20], Coq from INRIA [13], LEGO from Edinburgh [26], and ALF from Göteborg [1, 27]. Coq and LEGO are both based on Coquand and Huet's calculus of constructions, while ALF is an implementation of the theory we describe in this chapter. A brief overview of the ALF system is given in section 6.

2 Propositions as sets

The basic idea of type theory to identify propositions with sets goes back to Curry [10], who noticed that the axioms for positive implicational calculus,

formulated in the Hilbert style,

$$A \supset B \supset A$$

$$(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$$

correspond to the types of the basic combinators K and S

$$K \in A \rightarrow B \rightarrow A$$

$$S \in (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$$

Modus ponens then corresponds to functional application. Tait [39] noticed the further analogy that removing a cut in a derivation corresponds to a reduction step of the combinator representing the proof. Howard [22] extended these ideas to first-order intuitionistic arithmetic. Another way to see that propositions can be seen as sets is through Heyting's [21] explanations of the logical constants. The constructive explanation of logic is in terms of proofs: a proposition is true if we know how to prove it. For implication we have

> A proof of $A \supset B$ is a function (method, program) which to each proof of A gives a proof of B.

The notion of function or method is primitive in constructive mathematics and a function from a set A to a set B can be viewed as a program which when applied to an element in A gives an element in B as output. The idea of propositions as sets is now to identify a proposition with the set of its proofs. In case of implication we get

> $A \supset B$ is identified with $A \rightarrow B$, the set of functions from A to B.

The elements in the set $A \rightarrow B$ are of the form $\lambda x.b$, where $b \in B$ and b may depend on $x \in A$.

Heyting's explanation of conjunction is that a proof of $A \wedge B$ is a pair whose first component is a proof of A and whose second component is a proof of B. Hence, we get the following interpretation of a conjunction as a set.

> $A \wedge B$ is identified with $A \times B$, the cartesian product of A and B.

The elements in the set $A \times B$ are of the form $\langle a, b \rangle$ where $a \in A$ and $b \in B$.

A disjunction is constructively true if and only if we can prove one of the disjuncts. So a proof of $A \vee B$ is either a proof of A or a proof of B together with the information of which of A or B we have a proof. Hence,

$A \vee B$ is identified with $A + B$, the disjoint union of A and B.

The elements in the set $A+B$ are of the form $\mathsf{inl}(a)$ and $\mathsf{inr}(b)$, where $a \in A$ and $b \in B$.

The negation of a proposition A can be defined by:

$$\neg A \quad \equiv \quad A \supset \bot$$

where $\bot$ stands for absurdity, that is a proposition which has no proof. If we let $\emptyset$ denote the empty set, we have

$\neg A$ is identified with the set $A \rightarrow \emptyset$

using the interpretation of implication.

In order to interpret propositions defined using quantifiers, we need operations defined on families of sets, i.e. sets B depending on elements x in some set A. We let $B\,[x \leftarrow a]$ denote the expression obtained by substituting a for all free occurrences of x in B. Heyting's explanation of the existential quantifier is the following.

A proof of $(\exists x \in A)B$ consists of a construction of an element a in the set A together with a proof of $B\,[x \leftarrow a]$.

So, a proof of $(\exists x \in A)B$ is a pair whose first component a is an element in the set A and whose second component is a proof of $B\,[x \leftarrow a]$. The set corresponding to this is the disjoint union of a family of sets, denoted by $(\Sigma x \in A)B$. The elements in this set are pairs $\langle a, b\rangle$ where $a \in A$ and $b \in B\,[x \leftarrow a]$. We get the following interpretation of the existential quantifier.

$(\exists x \in A)B$ is identified with the set $(\Sigma x \in A)B$.

Finally, we have the universal quantifier.

A proof of $(\forall x \in A)B$ is a function (method, program) which to each element a in the set A gives a proof of $B\,[x \leftarrow a]$.

The set corresponding to the universal quantifier is the cartesian product of a family of sets, denoted by $(\Pi x \in A)B$. The elements in this set are functions which, when applied to an element a in the set A gives an element in the set $B\,[x \leftarrow a]$. Hence,

$(\forall x \in A)B$ is identified with the set $(\Pi x \in A)B$.

The elements in the set $(\Pi x \in A)B$ are of the form $\lambda x.b$ where $b \in B$ and both b and B may depend on $x \in A$. Note that if B does not depend on x then $(\Pi x \in A)B$ is the same as $A \to B$, so $\to$ is not needed as a primitive when we have cartesian products over families of sets. In the same way, $(\Sigma x \in A)B$ is nothing but $A \times B$ when B does not depend on x.

Except the empty set, we have not yet introduced any sets that correspond to atomic propositions. One such set is the equality set $a =_A b$, which expresses that a and b are equal elements in the set A. Recalling that a proposition is identified with the set of its proofs, we see that this set is nonempty if and only if a and b are equal. If a and b are equal elements in the set A, we postulate that the constant $\mathsf{id}(a)$ is an element in the set $a =_A b$.

When explaining the sets interpreting propositions we have used an informal notation to express elements of the sets. This notation differs from the one we will use in type theory in that that notation will be monomorphic in the sense that the constructors of a set will depend on the set. For instance, an element of $A \to B$ will be of the form $\lambda(A, B, b)$ and an element of $A \times B$ will be of the form $\langle A, B, a, b\rangle$.

3 Semantics and formal rules

We will in this section first introduce the notion of type and the judgement forms this explanation give rise to. We then explain what a family of types is and introduce the notions of variable, assumption and substitution together with the rules that follow from the semantic explanations. Next, the function types are introduced with their semantic explanation and the formal rules which the explanation justifies. The rules are formulated in the style of natural deduction [36].

3.1 Types

The basic notion in Martin-Löf's type theory is the notion of type. A type is explained by saying what an object of the type is and what it means for two objects of the type to be identical. This means that we can make the judgement

$$A \text{ is a type,}$$

which we in the formal system write as

$$A \ \mathit{type},$$

when we know the conditions for asserting that something is an object of type A and when we know the conditions for asserting that two objects

of type A are identical. We require that the conditions for identifying two objects must define an equivalence relation.

When we have a type, we know from the semantic explanation of what it means to be a type what the conditions are to be an object of that type. So, if A is a type and we have an object a that satisfies these conditions then

$$a \text{ is an object of type } A,$$

which we formally write

$$a \in A.$$

Furthermore, from the semantics of what it means to be a type and the knowledge that A is a type we also know the conditions for two objects of type A to be identical. Hence, if A is a type and a and b are objects of type A and these objects satisfies the equality conditions in the semantic explanation of A then

$$a \text{ and } b \text{ are identical objects of type } A,$$

which we write

$$a = b \in A.$$

Two types are equal when an arbitrary object of one type is also an object of the other and when two identical objects of one type are identical objects of the other. If A and B are types we know the conditions for being an object and the conditions for being identical objects of these types. Then we can investigate if all objects of type A are also objects of type B and if all identical objects of type A are also objects of type B and vice versa. If these conditions are satisfied then

$$A \text{ and } B \text{ are identical types},$$

which we formally write

$$A = B.$$

The requirement that the equality between objects of a type must be an equivalence relation is formalized by the rules:

Reflexivity of objects

$$\frac{a \in A}{a = a \in A}$$

Symmetry of objects

$$\frac{a = b \in A}{b = a \in A}$$

Transitivity of objects

$$\frac{a = b \in A \qquad b = c \in A}{a = c \in A}$$

The corresponding rules for types are easily justified from the meaning of what it means to be a type.

Reflexivity of types

$$\frac{A \ \mathit{type}}{A = A}$$

Symmetry of types

$$\frac{A = B}{B = A}$$

Transitivity of types

$$\frac{A = B \qquad B = C}{A = C}$$

The meaning of the judgement forms $a \in A$, $a = b \in A$ and $A = B$ immediately justifies the rules

Type equality rules

$$\frac{a \in A \qquad A = B}{a \in B} \qquad\qquad \frac{a = b \in A \qquad A = B}{a = b \in B}$$

3.2 Hypothetical judgements

The judgements we have introduced so far do not depend on any assumptions. In general, a hypothetical judgement is made in a *context* of the form

$$x_1 \in A_1,\ x_2 \in A_2,\ \ldots,\ x_n \in A_n$$

where we already know that A_1 is a type, A_2 is a type in the context $x_1 \in A_1$, ..., and A_n is a type in the context $x_1 \in A_1$, $x_2 \in A_2$, ..., $x_{n-1} \in A_n$. The explanations of hypothetical judgements are made by induction on the length of a context. We have already given the meaning of the judgement forms in the empty context; hence we could now directly explain the judgement forms in a context of length n. However, in order not to hide the explanations by heavy notation, we will give them for hypothetical judgements only depending on one assumption and then illustrate the general case with the judgement that A is a type in a context of length n.

Let C be a type which does not depend on any assumptions. That A is a type when $x \in C$, which we write

$$A \; type \; [x \in C],$$

means that, for an arbitrary object c of type C, $A\,[x \leftarrow c]$ is a type, that is, A is a type when c is substituted for x. Furthermore we must also know that if c and d are identical objects of type C then $A\,[x \leftarrow c]$ and $A\,[x \leftarrow d]$ are the same types. When A is a type depending on $x \in C$ we say that A is *a family of types over the type C*.

That A and B are identical families of types over the type C,

$$A = B \; [x \in C],$$

means that $A\,[x \leftarrow c]$ and $A\,[x \leftarrow c]$ are equal types for an arbitrary object c of type C.

That a is an object of type A when $x \in C$,

$$a \in A \; [x \in C],$$

means that we know that $a\,[x \leftarrow c]$ is an object of type $A\,[x \leftarrow c]$ for an arbitrary object c of type C. We must also know that $a\,[x \leftarrow c]$ and $a\,[x \leftarrow d]$ are identical objects of type $A\,[x \leftarrow c]$ whenever c and d are identical objects of type C.

That a and b are identical objects of type A depending on $x \in C$,

$$a = b \in A \; [x \in C],$$

means that $a\,[x \leftarrow c]$ and $b\,[x \leftarrow c]$ are the same objects of type $A\,[x \leftarrow c]$ for an arbitrary object c of type C.

We will illustrate the general case by giving the meaning of the judgement that A is a type in a context of length n; the other hypothetical judgements are explained in a similar way. We assume that we already know the explanations of the judgement forms in a context of length $n-1$. Let

$$x_1 \in A_1, \; x_2 \in A_2, \; \ldots \; , \; x_n \in A_n$$

be a context of length n. We then know that

$$\begin{array}{l} A_1 \; type \\ A_2 \; type \; [x_1 \in A_1] \\ \qquad \vdots \\ A_n \; type \; [x_1 \in A_1, \; x_2 \in A_2, \; \ldots, \; x_{n-1} \in A_{n-1}] \end{array}$$

To know the hypothetical judgement

$$A\ type\ [x_1 \in A_1,\ x_2 \in A_2,\ \ldots,\ x_n \in A_n]$$

means that we know that the judgement

$$\begin{array}{l} A\,[x_1 \leftarrow a]\ type \\ \quad [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,] \end{array}$$

holds for an arbitrary object a of type A_1 in the empty context. We must also require that if a and b are arbitrary identical objects of type A_1 then the judgement

$$\begin{array}{l} A\,[x_1 \leftarrow a] = A\,[x_1 \leftarrow b] \\ \quad [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,] \end{array}$$

holds. This explanation justifies two rules for substitution of objects in types. We can formulate these rules in different ways, simultaneously substituting objects for one or several of the variables in the context. Formulating the rules so they follow the semantical explanation as closely as possible gives us:

Substitution in types

$$\frac{A\ type\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{A\,[x_1 \leftarrow a]\ type\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

$$\frac{A\ type\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a = b \in A_1}{A\,[x_1 \leftarrow a] = A\,[x_1 \leftarrow b]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

The explanations of the other hypothetical judgement forms give the following substitution rules. Let A and B be types in the context $x_1 \in A_1$, $x_2 \in A_2,\ \ldots,\ x_n \in A_n$.

Substitution in equal types

$$\frac{A = B\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{A\,[x_j \leftarrow a] = B\,[x_j \leftarrow a]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

Let A be a type in the context $x_1 \in A_1,\ x_2 \in A_2,\ \ldots,\ x_n \in A_n$.

Substitution in objects

$$\frac{a \in A\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{a\,[x_1 \leftarrow a] \in A\,[x_1 \leftarrow a]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

Let A be a type and c and d be objects of type A in the context $x_1 \in A_1$, $x_2 \in A_2,\ \ldots,\ x_n \in A_n$.

Substitution in equal objects

$$\frac{c = d \in A\ [x_1 \in A_1, x_2 \in A_2, \ldots, x_n \in A_n] \qquad a \in A_1}{c\,[x_1 \leftarrow a] = d\,[x_1 \leftarrow a] \in A\,[x_1 \leftarrow a]\ [x_2 \in A_2\,[x_1 \leftarrow a],\ \ldots,\ x_n \in A_n\,[x_1 \leftarrow a]\,]}$$

The explanations of the hypothetical judgement forms justifies the following rule for introducing assumptions.

Assumption

$$\frac{\begin{array}{ll} A_1\ \mathit{type} & \\ A_2\ \mathit{type} & [x_1 \in A_1] \\ \quad\vdots & \\ A_n\ \mathit{type} & [x_1 \in A_1, \ldots, x_{n-1} \in A_{n-1}] \\ A\ \mathit{type} & [x_1 \in A_1, \ldots, x_{n-1} \in A_{n-1}, x_n \in A_n] \end{array}}{x \in A\ [x_1 \in A_1, \ldots x_n \in A_n, x \in A]}$$

In this rule all premises are explicit. In order to make the rules shorter and more comprehensible we will often leave out that part of the context which is the same in the conclusion and each premise.

The rules given in the previous section without assumptions could be justified also for hypothetical judgements.

3.3 Function types

One of the basic ways to form a new type from old ones is to form a function type. So, if we we have a type A and a family B of types over A, we want to form the dependent function type $(x \in A)B$ of functions from A to B. In order to do this, we must explain what it means to be an object of type $(x \in A)B$ and what it means for two objects of type $(x \in A)B$ to be identical. The function type is explained in terms of application.

To know that an object c is of type $(x \in A)B$ means that we know that when we apply it to an arbitrary object a of type A we get an object $c(a)$ in $B\,[x \leftarrow a]$ and that we get identical objects in $B\,[x \leftarrow a]$ when we apply it to identical objects a and b of A.

That two objects c and d of $(x \in A)B$ are identical means that when we apply them on an arbitrary object a of type A we get identical objects of type $B\,[x \leftarrow a]$.

Since we now have explained what it means to be an object of a function type and the conditions for two objects of a function type to be equal, we can justify the rule for forming the function type.

Function type

$$\frac{A\ \mathit{type} \qquad B\ \mathit{type}\ [x \in A]}{(x \in A)B\ \mathit{type}}$$

We also obtain the rule for forming equal function types.

Equal function types

$$\frac{A = A' \qquad B = B'\ [x \in A]}{(x \in A)B = (x \in A')B'}$$

We will use the abbreviation $(A)B$ for $(x \in A)B$ when B does not depend on x. We will also write $(x \in A; y \in B)C$ instead of $(x \in A)(y \in B)C$ and $(x, y \in A)B$ instead of $(x \in A; y \in A)C$.

We can also justify the following two rules for application

Application

$$\frac{c \in (x \in A)B \qquad a \in B}{c(a) \in B\,[x \leftarrow a]} \qquad\qquad \frac{c \in (x \in A)B \qquad a = b \in A}{c(a) = c(b) \in B\,[x \leftarrow a]}$$

We also have the following rules for showing that two functions are equal.

Application

$$\frac{c = d \in (x \in A)B \qquad a \in A}{c(a) = d(a) \in B\,[x \leftarrow a]}$$

Extensionality

$$\frac{c \in (x \in A)B \quad d \in (x \in A)B \quad c(x) = d(x) \in B\ [x \in A]}{c = d \in (x \in A)B}$$

x must occur free neither in c nor in d

Instead of writing repeated applications as $c(a_1)(a_2)\cdots(a_n)$ we will use the simpler form $c(a_1, a_2, \ldots, a_n)$.

One fundamental way to introduce a function is to abstract a variable from an expression:

Abstraction

$$\frac{b \in B\ [x \in A]}{([x]b) \in (x \in A)B}$$

We will write repeated abstractions as $[x_1, x_2, \ldots, x_n]b$ and also exclude the outermost parentheses when there is no risk of confusion.

How do we know that this rule is correct, i.e. how do we know that $[x]b$ is a function of the type $(x \in A)B$? By the semantics of function types, we must know that when we apply $[x]b$ of type $(x \in A)B$ on an object a of type A, then we get an object of type $B[x \leftarrow a]$; the explanation is by β-conversion:

β-conversion

$$\frac{a \in A \qquad b \in B\ [x \in A]}{([x]b)(a) = b\,[x \leftarrow a] \in B\,[x \leftarrow a]}$$

We must also know that when we apply an abstraction $[x]b$, where $b \in B\ [x \in A]$, on two identical objects a_1 and a_2 of type A, then we get identical results of type $B\,[x \leftarrow a]$ as results. We can see this in the following way. By β-conversion we know that $([x]b)(a_1) = b\,[x \leftarrow a_1] \in B\,[x \leftarrow a_1]$ and $([x]b)(a_2) = b\,[x \leftarrow a_2] \in B\,[x \leftarrow a_2]$. By the meaning of the judgements B *type* $[x \in A]$ and $b \in B\ [x \in A]$ we know that $B[x \leftarrow a_1] = B[x \leftarrow a_2]$ and that $b\,[x \leftarrow a_1] = b\,[x \leftarrow a_2] \in B\,[x \leftarrow a_1]$. Hence, by symmetry and transitivity, we get $([x]b)(a_1) = ([x]b(a_2) \in B\,[x \leftarrow a_1]$ from $a_1 = a_2 \in A$.

To summarize: to be an object f in a functional type $(x \in A)B$ means that it is possible to make an application $f(a)$ if $a \in A$. Then by looking at β-conversion as the definition of what it means to apply an abstracted expression to an object it is possible to give a meaning to an abstracted expression. Hence, application is more primitive then abstraction on this type level. Later we will see that for the *set* of functions, the situation is different.

By the rules we have introduced, we can derive the rules

η-conversion

$$\frac{c \in (x \in A)B}{([x]c(x)) = c \in (x \in A)B}$$

x must not occur free in c

ξ-rule

$$\frac{b = d \in B\ [x \in A]}{[x]b = [x]d \in (x \in A)B}$$

3.4 The type Set

The objects in the type Set consist of inductively defined sets. In order to explain a type we have to explain what it means to be an object in it and what it means for two such objects to be the same. So, to know that Set is a type we must explain what a set is and what it means for two sets to be the same: to know that A is an object in Set (or equivalently that A is a set) is to know how to form canonical elements in A and when two canonical elements are equal. A canonical element is an element on constructor form; examples are zero and the successor function for natural numbers.

Two sets are the same if an element of one of the sets is also an element of the other and if two equal elements of one of the sets are also equal elements of the other.

This explanation justifies the following rule

Set-formation

$$\mathsf{Set}\ \mathit{type}$$

If we have a set A we may form a type $El(A)$ whose objects are the elements of the set A:

El-formation

$$\frac{A : \mathsf{Set}}{El(A)\ \mathit{type}}$$

Notice that it is required that the sets are built up inductively: we know exactly in what ways we can build up the elements in the set and different ways corresponding to different constructors. As an example, for the ordinary definition of natural numbers, there are precisely two ways of building up elements, one using zero and the other one using the successor function. This is in contrast with types which in general are not built up inductively. For instance, the type Set can obviously not be defined inductively. It is always possible to introduce new sets (i.e. objects in Set). The concept of type is open; it is possible to add more types to the language, for instance by adding a new object A to Set gives the new type $\mathsf{El}(A)$).

In the sequel, we will often write A instead of $El(A)$ since it will always be clear from the context if A stands for the set A or the type of elements of A.

3.5 Definitions

Most of the generality and usefulness of the language comes from the possibilities of introducing new constants. It is in this way that we can introduce the usual mathematical objects like natural numbers, integers, functions, tuples etc. It is also possible to introduce more complicated inductive sets like sets for proof objects: it is in this way rules and axioms of a theory is represented in the framework.

A distinction is made between primitive and defined constants. The value of a *primitive constant* is the constant itself. So, the constant has only a type and not a definition; instead it gets its meaning by the semantics of the theory. Such a constant is also called a constructor. Examples of primitive constants are N, succ and $\mathsf{0}$; they can be introduced by the following declarations:

$$\mathsf{N} \in \mathsf{Set}$$

$$\begin{array}{rcl} \mathsf{succ} & \in & \mathsf{N} \rightarrow \mathsf{N} \\ \mathsf{0} & \in & \mathsf{N} \end{array}$$

A defined constant is defined in terms of other objects. When we apply a defined constant to all its arguments in an empty context, for instance, $c(e_1, \ldots, e_n)$, then we get an expression which is a definiendum, that is, an expression which computes in one step to its definiens (which is a well-typed object).

A defined constant can either be explicitly or implicitly defined. We declare an *explicitly defined constant* c by giving it as an abbreviation of an object a in a type A:

$$c = a \in A$$

For instance, we can make the following explicit definitions:

$$\begin{array}{rcl} 1 & = & \mathsf{succ}(\mathsf{0}) \in \mathsf{N} \\ I_{\mathsf{N}} & = & [x]x \in \mathsf{N} \rightarrow \mathsf{N} \\ I & = & [A, x]x \in (A \in \mathsf{Set}; A)A \end{array}$$

The last example is the monomorphic identity function which when applied to an arbitrary set A yields the identity function on A. It is easy to see if an explicit definition is correct: you just check that the definiens is an object in the correct type.

We declare an *implicitly defined constant* by showing what definiens it has when we apply it to its arguments. This is done by pattern-matching and the definition may be recursive. Since it is not decidable if an expression defined by pattern-matching on a set really defines a value for each element of the set, the correctness of an implicit definition is in general a semantical issue. We must be sure that all well-typed expressions of the form $c(e_1, \ldots, e_n)$ is a definiendum with a unique well-typed definiens. Here are two examples, addition and the operator for primitive recursion in arithmetic:

$$\begin{array}{rcl} + & \in & \mathsf{N} \rightarrow \mathsf{N} \rightarrow \mathsf{N} \\ +(\mathsf{0}, y) & = & y \\ +(\mathsf{succ}(x), y) & = & \mathsf{succ}(+(x, y)) \\ \mathsf{natrec} & \in & \mathsf{N} \rightarrow (\mathsf{N} \rightarrow \mathsf{N} \rightarrow \mathsf{N}) \rightarrow \mathsf{N} \rightarrow \mathsf{N} \\ \mathsf{natrec}(d, e, \mathsf{0}) & = & d \\ \mathsf{natrec}(d, e, \mathsf{succ}(a)) & = & e(a, \mathsf{natrec}(d, e, a)) \end{array}$$

4 Propositional logic

Type theory can be used as a logical framework, that is, it can be used to represent different theories. In general, a theory is presented by a list of typings

$$c_1 \in A_1, \ldots,\ c_n \in A_n$$

where $c_1, \ldots, c_n$ are new primitive constants, and a list of definitions

$$d_1 = e_1 \in B_1, \ldots,\ d_m = e_m \in A_m$$

where $d_1, \ldots, d_m$ are new defined constants.

The basic types of Martin-Löf's type theory are Set and the types of elements of the particular sets we introduce. In the next section we will give a number of examples of sets, but first we use the idea of propositions as sets to express that propositional logic with conjunction and implication; the other connectives can be introduced in the same way. When viewed as the type of propositions, the semantics of Set can be seen as the constructive explanation of propositions: a proposition is defined by laying down what counts as a direct (or canonical) proof of it; or differently expressed: a proposition is defined by its introduction rules. Given a proposition A, that is, an object of the type Set, then $El(A)$ is the type of proofs of A. From the semantics of sets we get that two proofs are the same if they have the same form and identical parts; we also get that two propositions are the same if a proof of one of the propositions is also a proof of the other and if identical proofs of one of the propositions are also identical proofs of the other.

The primitive constant & for conjunction is introduced by the following declaration

$$\& \in (\mathsf{Set}; \mathsf{Set})\mathsf{Set}$$

From this declaration we obtain, by repeated function application, the clause for conjunction in the usual inductive definition of formulas in the propositional calculus:

& -formation

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set}}{A \& B \in \mathsf{Set}}$$

where we have used infix notation, that is, we have written $A\&B$ instead of $\&(A, B)$.

We must now define what counts as a proof of a conjunction, and that is done by the following declaration of the primitive constant $\&_I$.

$$\&_I \in (A, B \in \mathsf{Set}; A; B)A\&B$$

This declaration is the inductive definition of the set $\&(A, B)$, that is, all elements in the set is equal to an element of the form $\&_I(A, B, a, b)$, where A and B are sets and $a \in A$ and $b \in B$. A proof of the syntactical form $\&_I(A, B, a, b)$ is called a *canonical proof* of $A\&B$.

By function application, we obtain the introduction rule for conjunction from the declaration of $\&_I$.

& -introduction

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad a \in A \qquad b \in B}{\&_I(A, B, a, b) \in A\&B}$$

To obtain the two elimination rules for conjunction, we introduce the two defined constants

$$\&_{E1} \in (A, B \in \mathsf{Set}; A\&B)A$$

and

$$\&_{E2} \in (A, B \in \mathsf{Set}; A\&B)B$$

by the defining equations

$$\&_{E1}(A, B, \&_I(A, B, a, b)) = a \in A$$

and

$$\&_{E2}(A, B, \&_I(A, B, a, b)) = b \in B$$

respectively. Notice that it is the definition of the constants which justifies their typings. To see that the typing of $\&_{E1}$ is correct, assume that A and B are sets, and that $p \in A\&B$. We must then show that $\&_{E1}(A, B, p)$ is an element in A. But since $p \in A\&B$, we know that p is equal to an element of the form $\&_I(A, B, a, b)$, where $a \in A$ and $b \in B$. But then we have that $\&_{E1}(A, B, p) = \&_{E1}(A, B, \&_I(A, B, a, b))$ which is equal to a by the defining equation of $\&_{E1}$.

From the typings of $\&_{E1}$ and $\&_{E2}$ we obtain, by function application, the elimination rules for conjunction:

& -elimination 1

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad c \in A\&B}{\&_{E1}(A, B, c) \in A}$$

and

& -elimination 2

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad c \in A\&B}{\&_{E2}(A, B, c) \in B}$$

The defining equations for $\&_{E1}$ and $\&_{E2}$ correspond to Prawitz' reduction rules in natural deduction:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ A\end{array}}{A\&B}}{A} \quad = \quad \begin{array}{c}\vdots\\ A\end{array}$$

and

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ B\end{array}}{A\&B}}{B} \quad = \quad \begin{array}{c}\vdots\\ B\end{array}$$

respectively. Notice the role which these rules play here. They are used to justify the correctness, that is, the well typings of the elimination rules. The elimination rules are looked upon as methods which can be executed, and it is the reduction rules which defines the execution of the elimination rules.

The primitive constant $\supset$ for implication is introduced by the declaration

$$\supset\, \in (\mathsf{Set}; \mathsf{Set})\mathsf{Set}$$

As for conjunction, we obtain from this declaration the clause for implication in the inductive definition of formulas in the propositional calculus:

$\supset$ -formation

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set}}{A \supset B \in \mathsf{Set}}$$

A canonical proof of an implication is formed by the primitive constant $\supset_I$, declared by

$$\supset_I\, \in (A, B \in \mathsf{Set}; (A)B)A \supset B$$

By function application, the introduction rule for implication is obtained from the declaration of $\supset_I$:

$\supset$ -introduction

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad b(x) \in B \ [x \in A]}{\supset_I (A, B, b) \in A \supset B}$$

So, to get a canonical proof of $A \supset B$ we must have a function b which when applied on a proof of A gives a proof of B, and the proof then obtained is $\supset_I (A, B, b)$.

To obtain modus ponens, the elimination rule for $\supset$, we introduce the defined constant

$$\supset_E \in (A, B \in \mathsf{Set}; A \supset B; A)B$$

which is defined by the equation

$$\supset_E (A, B, \supset_I (A, B, b, a)) = b(a) \in B$$

In the same way as for conjunction, we can use this definition to show that $\supset_E$ is well-typed.

The defining equation corresponds to the reduction rule

$$\dfrac{\dfrac{\begin{array}{c} A \\ \vdots \\ B \end{array}}{A \supset B} \qquad \begin{array}{c}\vdots \\ A\end{array}}{B} \quad = \quad \begin{array}{c} \vdots \\ A \\ \vdots \\ B \end{array}$$

By function application, we obtain from the typing of $\supset_E$

$\supset$ -elimination

$$\frac{A \in \mathsf{Set} \qquad B \in \mathsf{Set} \qquad b \in A \supset B \qquad a \in A}{\supset_E (A, B, b, a) \in B}$$

5 Set theory

We will in this section introduce a theory of sets with natural numbers, lists, functions, etc. which could be used when specifying and implementing computer programs. We will also show how this theory is represented in the type theory framework.

When defining a set, we first introduce a primitive constant for the set and then give the primitive constants for the constructors, which express the different ways elements of the set can be constructed. The typing rule for constant denoting the set is called the *formation rule* of the set and the typing rules for the constructors are called the *introduction rules*. Finally, we introduce a selector as an implicitly defined constant to express the induction principle of the set; the selector is defined by pattern-matching and may be recursive. The type rule for the selector is called the *elimination rule* and the defining equations are called *equality rules*.

Given the introduction rules, it is possible to mechanically derive the elimination rule and the equality rules for a set; how this can be done have been investigated by Martin-Löf [29], Backhouse [2], Coquand and Paulin [9], and Dybjer[14].

5.1 The set of Boolean values

The set of Boolean values is an example of an enumeration set. The values of an enumeration set are exactly the same as the constructors of the set and all constructors yield different elements. For the Booleans this means that there are two ways of forming an element and therefore also two constructors; true and false. Since we have given the elements of the set and their equality, we can introduce a constant for the set and make the type declaration

$$\mathsf{Bool} \in \mathsf{Set}$$

We can also declare the types of the constructor constants

$$\mathsf{true} \in \mathsf{Bool}$$
$$\mathsf{false} \in \mathsf{Bool}$$

The principal selector constant of an enumeration set is a function that performs case analysis on Boolean values. For the Booleans we introduce the if constant with the type

$$\mathsf{if} \in (C \in (\mathsf{Bool})\mathsf{Set}; b \in \mathsf{Bool}; C(\mathsf{true}); C(\mathsf{false}))\, C(b)$$

and the defining equations

$$\mathsf{if}(C, \mathsf{true}, a, b) = a$$

$$\mathsf{if}(C, \mathsf{false}, a, b) = b$$

In these two definitional equalities we have omitted the types since they can be obtained immediately from the typing of if. In the sequel, we will often write just $a = b$ instead of $a = b \in A$ when the type A is clear from the context.

5.2 The empty set

To introduce the empty set, $\{\}$, we just define a set with no constructors at all. First we make a type declaration for the set

$$\{\} \in \mathsf{Set}$$

Since there are no constructors we immediately define the selector case and its type by the declaration

$$\mathsf{case} \in (C \in (\{\})\mathsf{Set}; a \in \{\})\, C(a)$$

The empty set corresponds to the absurd proposition and the selector corresponds to the natural deduction rule for absurdity

$$\frac{\bot \ \mathit{true} \qquad C \ \mathit{prop}}{C \ \mathit{true}}$$

5.3 The set of natural numbers

In order to introduce the set of natural numbers, N, we must give the rules for forming all the natural numbers as well as all the rules for forming two equal natural numbers. These are the introduction rules for natural numbers.

There are two ways of forming natural numbers, $\mathsf{0}$ is a natural number and if n is a natural number then $\mathsf{succ}(n)$ is a natural number. There are also two corresponding ways of forming equal natural numbers, the natural number $\mathsf{0}$ is equal to $\mathsf{0}$ and if the natural number n is equal to m, then $\mathsf{succ}(n)$ is equal to $\mathsf{succ}(m)$. So we have explained the meaning of the natural numbers as a set, and can therefore make the type declaration

$$\mathsf{N} \in \mathsf{Set}$$

and form the introduction rules for the natural numbers, by declaring the types of the constructor constants $\mathsf{0}$ and succ

$$\mathsf{0} \in \mathsf{N}$$

$$\mathsf{succ} \in (n \in \mathsf{N})\,\mathsf{N}$$

The general rules in the framework makes it possible to give the introduction rules in this simple form.

We will introduce a very general form of selector for natural numbers, natrec, as a defined constant. It could be used both for expressing elements by primitive recursion and proving properties by induction. The functional constant natrec takes four arguments; the first is a family of sets that determines the set which the result belongs to, the second and third are the results for the zero and successor case, respectively, and the fourth argument, finally, is the natural number which is the principal argument of the selector. Formally, the type of natrec is

$$\begin{array}{ll} \mathsf{natrec} \in & (C \in (\mathsf{N})\,\mathsf{Set}; \\ & \;d \in C(\mathsf{0}); \\ & \;e \in (x \in \mathsf{N}; y \in C(x))\,C(\mathsf{succ}(x)); \\ & \;n \in \mathsf{N}) \\ & C(n) \end{array}$$

The defining equations for the natrec constant are

$$\mathsf{natrec}(C, d, e, \mathsf{0}) = d$$

$$\mathsf{natrec}(C, d, e, \mathsf{succ}(m)) = e(n, \mathsf{natrec}(C, d, e, m))$$

The selector for natural numbers could, as we already mentioned, be used for introducing ordinary primitive recursive functions. Addition and multiplication could, for example, be introduced as two defined constants

$$\mathsf{plus} \in (\mathsf{N};\mathsf{N})\,\mathsf{N}$$

$$\mathsf{mult} \in (\mathsf{N};\mathsf{N})\,\mathsf{N}$$

by the defining equations

$$\mathsf{plus}(m,n) = \mathsf{natrec}([x]\mathsf{N}, n, [x,y]\mathsf{succ}(y), m)$$

$$\mathsf{mult}(m,n) = \mathsf{natrec}([x]\mathsf{N}, \mathsf{0}, [x,y]\mathsf{plus}(y,n), m)$$

Using the rules for application together with the type and the definitional equalities for the constant natrec it is easy to derive the type of the right hand side of the equalities above as well as the following equalities for addition and multiplication:

$$\mathsf{plus}(\mathsf{0},n) = n \in \mathsf{N} \quad [n \in \mathsf{N}]$$
$$\mathsf{plus}(\mathsf{succ}(m),n) = \mathsf{succ}(\mathsf{plus}(m,n)) \in \mathsf{N} \quad [m \in \mathsf{N}, n \in \mathsf{N}]$$

$$\mathsf{mult}(\mathsf{0},n) = \mathsf{0} \in \mathsf{N} \quad [n \in \mathsf{N}]$$
$$\mathsf{mult}(\mathsf{succ}(m),n) = \mathsf{plus}(\mathsf{mult}(m,n),n) \in \mathsf{N} \quad [m \in \mathsf{N}, n \in \mathsf{N}]$$

In general, if we have a primitive recursive function f from N to A

$$\begin{array}{rcl} f(\mathsf{0}) & = & d \\ f(\mathsf{succ}(n)) & = & e(n, f(n)) \end{array}$$

where $d \in A$ and e is a function in $(\mathsf{N};A)A$, we can introduce it as a defined constant

$$f' \in (\mathsf{N})A$$

using the defining equation

$$f'(n) = \mathsf{natrec}([x]A, d', e', n)$$

where d' and e' are functions in type theory which correspond to d and e in the definition of f.

The type of the constant natrec represents the usual elimination rule for natural numbers

$$\frac{\begin{array}{l} C(x)\ \mathsf{Set}\ [x \in \mathsf{N}] \\ d \in C(\mathsf{0}) \\ e \in C(\mathsf{succ}(x))\ [x \in \mathsf{N}, y \in C(x)] \\ x \in \mathsf{N} \end{array}}{\mathsf{natrec}(C,d,e,x) \in C(x)}$$

which can be obtained by assuming the arguments and then apply the constant natrec on them. Note that, in the conclusion of the rule, the expression $\mathsf{natrec}(C,d,e,x)$ contains the family C. This is a consequence of the explicit declaration of natrec in the framework.

5.4 The set of functions (Cartesian product of a family of sets)

We have already introduced the *type* $(x \in A)B$ of functions from the type A to the type B. We need the corresponding *set* of functions from a set to another set. If we have a set A and a family of sets B over A, we can form the cartesian product of a family of sets, which is denoted $\Pi(A, B)$. The elements of this set are functions which when applied to an element a in A yield an element in $B(a)$. The elements of the set $\Pi(A, B)$ are formed by applying the constructor λ to the sets A and B and an object of the corresponding function type.

The constant Π is introduced by the type declaration

$$\Pi \in (A \in \mathsf{Set}; B \in (x \in A)\mathsf{Set})\,\mathsf{Set}$$

and the constant λ by

$$\lambda \in (A \in \mathsf{Set}; B \in (x \in A)\,\mathsf{Set}; f \in (x \in A)B)\,\Pi(A, B)$$

These constant declarations correspond to the rules

$$\frac{A \in \mathsf{Set} \qquad B(x) \in \mathsf{Set}\ [x \in A]}{\Pi(A, B) \in \mathsf{Set}}$$

$$\frac{A \in \mathsf{Set} \qquad B(x) \in \mathsf{Set}\ [x \in A] \qquad f \in B(x)\ [x \in A]}{\lambda(A, B, f) \in \Pi(A, B)}$$

Notice that the elements of a cartesian product of a family of sets, $\Pi(A, B)$, are more general than ordinary functions from A to B in that the result of applying an element of $\Pi(A, B)$ to an argument can be in a set which may depend on the value of the argument.

The most important defined constant in the Π-set is the constant for application. In type theory this selector takes as arguments not only an element of $\Pi(A, B)$ and an object of type A but also the sets A and B themselves. The constant is introduced by the type declaration

$$\mathsf{apply} \in (A \in \mathsf{Set}; B \in (x \in A)\mathsf{Set}; g \in \Pi(A, B); a \in A)\,B(a)$$

and the definitional equality

$$\mathsf{apply}(A, B, \lambda(f), a) = f(a)$$

The cartesian product of a family of sets is, when viewed as a proposition the same as universal quantification. The type of the constructor corresponds to the introduction rule

$$\frac{B(x)\ \mathit{true}\ [x \in A]}{(\forall\, x \in A)\, B(x)\ \mathit{true}}$$

and the type of the selector corresponds to the elimination rule

$$\frac{(\forall x \in A)\, B(x)\ \mathit{true} \qquad a \in A}{B(a)\ \mathit{true}}$$

The cartesian product of a family of sets is a generalization of the ordinary function set. If the family of sets B over A is the same for all elements of A, then the cartesian product is just the set of ordinary functions. The constant $\rightarrow$ is introduced by the following explicit definition:

$$\rightarrow \in (A, B \in \mathsf{Set})\, \mathsf{Set}$$
$$\rightarrow = [A, B]\Pi(A, [x]B)$$

The set of functions is, when viewed as a proposition, the same as implication since the type of the constructor is the same as the introduction rule for implication

$$\frac{B\ \mathit{true}\ [A\ \mathit{true}]}{A \supset B\ \mathit{true}}$$

and the type of the selector is the same as the elimination rule

$$\frac{A \supset B\ \mathit{true} \qquad A\ \mathit{true}}{B\ \mathit{true}}$$

Given the empty set and the set of function we can define a constant for negation in the following way

$$\neg \in (A \in \mathsf{Set})\, \mathsf{Set}$$
$$\neg(A) = A \rightarrow \{\}$$

Example 1. Let us see how to prove the proposition $A \supset \neg\neg A$. In order to prove the proposition we must find an element in the set

$$A \rightarrow (\neg(\neg A)) \ \equiv\ A \rightarrow ((A \rightarrow \{\}) \rightarrow \{\})$$

We start by making the assumptions $x \in A$ and $y \in A \rightarrow \{\}$ and then obtain an element in $\{\}$

$$\mathsf{apply}(A \rightarrow \{\}, A, y, x) \in \{\}$$

and therefore

$$\begin{array}{l} \lambda(A, (A \rightarrow \{\}) \rightarrow \{\}, \\ \quad [x]\lambda(A \rightarrow \{\}, \{\}, \\ \qquad\quad [y]\mathsf{apply}(A \rightarrow \{\}, A, y, x))) \\ \in A \rightarrow ((A \rightarrow \{\}) \rightarrow \{\}) \ \equiv\ A \rightarrow (\neg(\neg A)) \end{array}$$

Example 2. Using the rules for natural numbers, booleans and functions we will show how to define a function, $\mathsf{eqN} \in (\mathsf{N}, \mathsf{N})\mathsf{Bool}$, that decides if two natural numbers are equal. We want the following equalities to hold:

$$\begin{array}{rcl} \mathsf{eqN}(0,0) & = & \mathsf{true} \\ \mathsf{eqN}(0,\mathsf{succ}(n)) & = & \mathsf{false} \\ \mathsf{eqN}(\mathsf{succ}(m),0) & = & \mathsf{false} \\ \mathsf{eqN}(\mathsf{succ}(m),\mathsf{succ}(n)) & = & \mathsf{eqN}(m,n) \end{array}$$

It is impossible to define eqN directly just using natural numbers and recursion on the arguments. We have to do recursion on the arguments separately and first use recursion on the first argument to compute a *function* which when applied to the second argument gives us the result we want. So we first define a function $f \in (\mathsf{N})\,(\mathsf{N} \to \mathsf{Bool})$ which satisfies the equalities

$$\begin{array}{rcl} f(0) & = & \lambda([m]\,\mathsf{iszero}(m)) \\ f(\mathsf{succ}(n)) & = & \lambda([m]\,\mathsf{natrec}(m, \mathsf{false}, [x,y]\,\mathsf{apply}(f(n),x))) \end{array}$$

where

$$\mathsf{iszero}(m) \quad = \quad \mathsf{natrec}(m, \mathsf{true}, [x,y]\,\mathsf{false})$$

If we use the recursion operator explicitly, we can define f as

$$\begin{array}{rcl} f(n) & = & \mathsf{natrec}(n, \\ & & \qquad \lambda([m]\mathsf{iszero}(m)), \\ & & \qquad [u,v]\,\lambda((m)\mathsf{natrec}(m, \mathsf{false}, [x,y]\,\mathsf{apply}(v,x)))) \end{array}$$

The function f is such that $f(n)$ is equal to a function which gives true if is applied to n and false otherwise, that is, we can use it to define eqN as follows

$$\mathsf{eqN}(m,n) \quad = \quad \mathsf{apply}(f(m),n)$$

It is a simple exercise to show that

$$\mathsf{eqN} \in (\mathsf{N}, \mathsf{N})\,\mathsf{Bool}$$

and that it satisfies the equalities we want it to satisfy.

5.5 Propositional equality

The equality on the judgement level $a = b \in A$ is a definitional equality and two objects are equal if they have the same normal form. In order to express that, for example, addition of natural numbers is a commutative operation it is necessary to introduce a set for propositional equality.

If a and b are elements in the set A, then $\mathsf{Id}(A, a, b)$ is a set. We express this by introducing the constant Id and its type

$$\mathsf{Id} \in (X \in \mathsf{Set}; a \in X; b \in X)\,\mathsf{Set}$$

The only constructor of elements in equality sets is id and it is introduced by the type declaration

$$\mathsf{id} \in (X \in \mathsf{Set}; x \in X)\,\mathsf{Id}(X, x, x)$$

To say that id is the only constructor for $\mathsf{Id}(A, a, b)$ is the same as to say that $\mathsf{Id}(A, a, b)$ is the least reflexive relation. Transitivity, symmetry and congruence can be proven from this definition. We use the name idpeel for the selector and it is introduced by the type declaration

$$\begin{aligned} \mathsf{idpeel} \in \ & (A \in \mathsf{Set}; \\ & \ C \in (x, y \in A; e \in \mathsf{Id}(A, x, y))\,\mathsf{Set}; \\ & \ a, b \in A; \\ & \ e \in \mathsf{Id}(A, a, b); \\ & \ d \in (x \in A)\,C(x, x, \mathsf{id}(A, x))) \\ & C(a, b, e) \end{aligned}$$

and the equality

$$\mathsf{idpeel}(A, C, a, b, \mathsf{id}(A, a), d) = d(a)$$

The intuition behind this constant is that it expresses a substitution rule for elements which are propositionally equal.

Example 3. The type of the constructor in the set $\mathsf{Id}(A, a, b)$ corresponds to the reflexivity rule of equality. The symmetry and transitivity rules can easily be derived.

Let A be a set and a and b two elements of A. Assume that

$$d \in \mathsf{Id}(A, a, b)$$

In order to prove symmetry, we must construct an element in $\mathsf{Id}(A, b, a)$. By applying idpeel on A, $[x, y, e]\mathsf{Id}(A, y, x)$, a, b, d and $[x]\mathsf{id}(A, x)$ we get, by simple typechecking, an element in the set $\mathsf{Id}(A, b, a)$.

$$\mathsf{idpeel}(A, [x, y, e]\mathsf{Id}(A, y, x), a, b, d, [x]\mathsf{id}(A, x)) \in \mathsf{Id}(A, b, a)$$

The derived rule for symmetry can therefore be expressed by the constant symm defined by

$$\mathsf{idsymm} \in (A \in \mathsf{Set}; a, b \in A; d \in \mathsf{Id}(A, a, b))\, \mathsf{Id}(A, b, a)$$

$$\mathsf{idsymm}(A, a, b, d) = \mathsf{idpeel}(A, [x, y, e]\mathsf{Id}(A, y, x), a, b, d, [x]\mathsf{id}(A, x))$$

Transitivity is proved in a similar way. Let A be a set and a, b and c elements of A. Assume that

$$d \in \mathsf{Id}(A, a, b) \text{ and } e \in \mathsf{Id}(A, b, c)$$

By applying idpeel on A, $[x, y, z]\mathsf{Id}(A, y, c) \rightarrow \mathsf{Id}(A, z, c)$, a, b, d and the identity function $[x]\lambda(\mathsf{Id}(A, x, c), \mathsf{Id}(A, x, c), [w]w)$ we get an element in the set $\mathsf{Id}(A, b, c) \rightarrow \mathsf{Id}(A, a, c)$. This element is applied on e in order to get the desired element in $\mathsf{Id}(A, a, c)$.

$$\mathsf{idtrans} \in (A \in \mathsf{Set}; a, b, c \in A; d \in \mathsf{Id}(A, a, b); e \in \mathsf{Id}(A, b, c))\, \mathsf{Id}(A, a, c)$$

$$\begin{aligned}\mathsf{idtrans}(A, a, b, c, d, e) = \mathsf{apply}(&\mathsf{Id}(A, b, c), \mathsf{Id}(A, a, c),\\ &\mathsf{idpeel}(A, [x, y, z]\mathsf{Id}(A, y, c) \rightarrow \mathsf{Id}(A, x, c),\\ &\qquad a, b, d,\\ &\qquad [x]\lambda(\mathsf{Id}(A, x, c), \mathsf{Id}(A, x, c), [w]w)),\\ &e)\end{aligned}$$

Example 4. Let us see how we can derive a rule for substitution in set expressions. We want to have a rule

$$\frac{P(x) \in \mathsf{set}\ [x \in A] \qquad a \in A \qquad b \in A \qquad c \in \mathsf{Id}(A, a, b) \qquad p \in P(a)}{\mathsf{subst}(P, a, b, c, p) \in P(b)}$$

To derive such a rule, first assume that we have a set A and elements a and b of A. Furthermore assume that $c \in \mathsf{Id}(A, a, b)$, $P(x) \in \mathsf{Set}\ [x \in A]$ and $p \in P(a)$. Type checking gives us that

$$\lambda(P(x), P(x), [w]w) \in P(x) \rightarrow P(x)\ [x \in A]$$

$$\begin{aligned}&\mathsf{idpeel}(A, [x, y, z](P(x) \rightarrow P(y)), a, b, c, [x]\lambda(P(x), P(x), [w]w))\\ &\in P(a) \rightarrow P(b)\end{aligned}$$

We can now apply the function above on p to obtain an element in $P(b)$. So we can define a constant subst that expresses the substitution rule above. The type of subst is

$$\begin{aligned}\mathsf{subst} \in\ &(A \in \mathsf{Set};\\ &\ P \in (A)\mathsf{Set};\\ &\ a, b \in A;\\ &\ c \in \mathsf{Id}(A, a, b);\\ &\ p \in P(a))\\ &P(b)\end{aligned}$$

and the defining equation

$$\begin{aligned}\mathsf{subst}(A, P, a, b, c, p) = \mathsf{apply}(&P(a), P(b),\\ &\mathsf{idpeel}(A, [x, y, z](P(x) \to P(y)),\\ &\qquad a, b, c,\\ &\qquad [x]\lambda(P(x), P(x), [w]w)),\\ &p)\end{aligned}$$

5.6 The set of lists

The set of lists $\mathsf{List}(A)$ is introduced in a similar way as the natural numbers, except that there is a parameter A that determines which set the elements of a list belongs to. There are two constructors to build a list, nil for the empty list and cons to add an element to a list. The constants we have introduced so far have the following types:

$$\mathsf{List} \in (A \in \mathsf{set})\,\mathsf{Set}$$

$$\begin{aligned}&\mathsf{nil} \in (A \in \mathsf{set})\,\mathsf{List}(A)\\ &\mathsf{cons} \in (A \in \mathsf{set}; a \in A; l \in \mathsf{List}(A))\,\mathsf{List}(A)\end{aligned}$$

The selector $\mathsf{listrec}$ for types is a constant that expresses primitive recursion for lists. The selector is introduced by the type declaration

$$\begin{aligned}\mathsf{listrec} \in\ &(A \in \mathsf{Set};\\ &\ C \in (\mathsf{List}(A))\,\mathsf{Set};\\ &\ c \in C(\mathsf{nil}(A));\\ &\ e \in (x \in A; y \in \mathsf{List}(A); z \in C(y))\,C(\mathsf{cons}(A, x, y));\\ &\ l \in \mathsf{List}(A))\\ &C(l)\end{aligned}$$

The defining equations for the $\mathsf{listrec}$ constant are

$$\mathsf{listrec}(A, C, c, e, \mathsf{nil}(A)) = c$$

$$\mathsf{listrec}(A, C, c, e, \mathsf{cons}(A, a, l)) = e(l, a, \mathsf{listrec}(A, C, c, e, l))$$

5.7 Disjoint union of two sets

If we have two sets A and B we can form the disjoint union $A+B$. The elements of this set are either of the form $\mathsf{inl}(A, B, a)$ or of the form $\mathsf{inr}(A, B, b)$ where $a \in A$ and $b \in B$. In order to express this in the framework we introduce the constants

$$+ \in (A, B \in \mathsf{Set})\,\mathsf{Set}$$

$$\mathsf{inl} \in (A, B \in \mathsf{Set}; A)\,A + B$$

$$\mathsf{inr} \in (A, B \in \mathsf{Set}; B)\,A + B$$

The selector when is introduced by the type declaration

$$\begin{array}{ll}\mathsf{when} \in & (A, B \in \mathsf{Set};\\ & \;C \in (A + B)\,\mathsf{Set};\\ & \;e \in (x \in A)\,C(\mathsf{inl}(A, B, x));\\ & \;f \in (y \in B)\,C(\mathsf{inr}(A, B, y));\\ & \;p \in A + B)\\ & C(p)\end{array}$$

and defined by the equations

$$\mathsf{when}(A, B, C, e, f, \mathsf{inl}(A, B, a)) = e(a)$$

$$\mathsf{when}(A, B, C, e, f, \mathsf{inr}(A, B, b)) = f(b)$$

Seen as a proposition the disjoint union of two sets expresses disjunction. The constructors correspond to the introduction rules

$$\frac{A\ \mathit{true}}{A \vee B\ \mathit{true}} \qquad\qquad \frac{B\ \mathit{true}}{A \vee B\ \mathit{true}}$$

and the selector when corresponds to the elimination rule.

$$\frac{A \vee B\ \mathit{true} \qquad C\ \mathit{prop} \qquad C\ \mathit{true}\ [A\ \mathit{true}] \qquad C\ \mathit{true}\ [B\ \mathit{true}]}{C\ \mathit{true}}$$

5.8 Disjoint union of a family of sets

In order to be able to deal with the existential quantifier and to have a set of ordinary pairs, we will introduce the disjoint union of a family of sets. The set is introduced by the type declaration

$$\Sigma \in (A \in \mathsf{Set}; B \in (A)\mathsf{Set})\,\mathsf{Set}$$

There is one constructor in this set, pair, which is introduced by the type declaration

$$\mathsf{pair} \in (A \in \mathsf{Set}; B \in (A)\mathsf{Set}; a \in A; B(a))\,\Sigma(A, B)$$

The selector of a set $\Sigma(A, B)$ splits a pair into its parts. It is defined by the type declaration

$$\begin{array}{ll}\mathsf{split} \in & (A \in \mathsf{Set}; B \in (A)\,\mathsf{Set};\\ & \;C \in (\Sigma(A, B))\,\mathsf{Set};\\ & \;d \in (a \in A; b \in B(a))\,C(\mathsf{pair}(A, B, a, b));\\ & \;p \in \Sigma(A, B))\\ & C(p)\end{array}$$

and the defining equation

$$\mathsf{split}(A, B, C, d, \mathsf{pair}(A, B, a, b)) = d(a, b)$$

Given the selector split it is easy to define the two projection functions that give the first and second component of a pair.

$$\mathsf{fst} \in (A \in \mathsf{Set}; B \in (A)\,\mathsf{Set}; p \in \Sigma(A, B))\,A$$
$$\mathsf{fst}(A, B, p) = \mathsf{split}(A, B, [x]A, [x, y]x, p)$$

$$\mathsf{snd} \in (A \in \mathsf{Set}; B \in (A)\,\mathsf{Set}; p \in \Sigma(A, B))\,B(\mathsf{fst}(A, B, p)$$
$$\mathsf{snd}(A, B, p) = \mathsf{split}(A, B, [x]B(\mathsf{fst}(A, B, p)), [x, y]y, p)$$

When viewed as a proposition the disjoint union of a family of sets Σ(A,B) corresponds to the existential quantifier $(\exists\, x \in A)B(x)$. The types of constructor pair and when correspond to the natural deduction rules for the existential quantifier

$$\frac{a \in A \qquad B(a)\ \mathit{true}}{(\exists\, x \in A)\,B(x)\ \mathit{true}}$$

$$\frac{(\exists\, x \in A)\,B(x)\ \mathit{true} \qquad C\ \mathit{prop} \qquad C\ \mathit{true}\ [x \in A, B(x)\ \mathit{true}]}{C\ \mathit{true}}$$

5.9 The set of small sets

A set of small sets U, or a universe, is a set that reflects some part of the set structure on the object level. It is of course necessary to introduce this set if one wants to do some computation using sets, for example to specify and prove a type checking algorithm correct, but it is also necessary in order to prove inequalities such as $0 \neq \mathsf{succ}(0)$. Furthermore, the universe can be used for defining families of sets using recursion, for example non-empty lists and sets such as N^n.

We will introduce the universe simultanously with a function S that maps an element of U to the set the element encodes. The universe we will introduce has one constructor for each set we have defined. The constants for sets are introduced by the type declaration

$$\mathsf{U} \in \mathsf{Set}$$
$$\mathsf{S} \in (\mathsf{U})\mathsf{Set}$$

Then we introduce the constructors in U and the defining equations for S

$$\mathsf{Bool}_U \in \mathsf{U}$$
$$\mathsf{S}(\mathsf{Bool}_U) = \mathsf{Bool}$$

$$\{\}_U \in \mathsf{U}$$
$$\mathsf{S}(\{\}_U) = \{\}$$

$$\mathsf{N}_U \in \mathsf{U}$$
$$\mathsf{S}(\mathsf{N}_U) = \mathsf{N}$$

$$\Pi_U \in (A \in \mathsf{U}; B \in (\mathsf{S}(A))\mathsf{U})\,\mathsf{U}$$
$$\mathsf{S}(\Pi_U(A, B)) = \Pi(\mathsf{S}(A), [h]\mathsf{S}(B(h)))$$

$$\mathsf{Id}_U \in (A \in \mathsf{U}; a \in \mathsf{S}(A); b \in \mathsf{S}(A))\,\mathsf{U}$$
$$\mathsf{S}(\mathsf{Id}_U(A, a, b)) = \mathsf{Id}(\mathsf{S}(A), a, b)$$

$$\mathsf{List}_U \in (A \in \mathsf{U})\,\mathsf{U}$$
$$\mathsf{S}(\mathsf{List}_U(A)) = \mathsf{List}(\mathsf{S}(A))$$

$$+_U \in (A \in \mathsf{U}; B \in \mathsf{U})\,\mathsf{U}$$
$$\mathsf{S}(+_U(A, B)) = +(\mathsf{S}(A), \mathsf{S}(B))$$

$$\Sigma_U \in (A \in \mathsf{U}; B \in (\mathsf{S}(A))\mathsf{U})\,\mathsf{U}$$
$$\mathsf{S}(\Sigma_U(A, B)) = \Sigma(\mathsf{S}(A), [h]\mathsf{S}(B(h)))$$

Example 5. Let us see how we can derive an element in the set

$$\neg\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$$

or, in other words, how we can find an expression in the set

$$\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)) \rightarrow \{\}$$

We start by assuming that

$$x \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$$

Then we construct a function, Iszero, that maps a natural number to an element in the universe.

$$\mathsf{Iszero} \in (\mathsf{N})\,\mathsf{U}$$
$$\mathsf{Iszero}(m) = \mathsf{natrec}(m, \mathsf{Bool}_U, [y, z]\{\}_U)$$

It is easy to see that

$$\mathsf{Iszero}(0) = \mathsf{S}(\mathsf{Bool}_U) = \mathsf{Bool}$$
$$\mathsf{Iszero}(\mathsf{succ}(0)) = \mathsf{S}(\{\}_U) = \{\}$$

and therefore

$$\mathsf{true} \in \mathsf{Bool} = \mathsf{Iszero}(0)$$

$$\mathsf{subst}(x, \mathsf{true}) \in \mathsf{Iszero}(\mathsf{succ}(0)) = \{\}$$

Finally, we have the element we are looking for

$$\lambda(\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)), \{\}, [x]\mathsf{subst}(x, \mathsf{true})) \in \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)) \rightarrow \{\}$$

It is shown in Smith [37] that without a universe no negated equalities can be proved.

6 ALF, an interactive editor for type theory

At the department of Computing Science in Göteborg, we have developed an interactive editor for objects and types in type theory. The editor is based on direct manipulation, that is, the things which are being built are shown on the screen, and editing is done by pointing and clicking on the screen.

The proof object is used as a true representative of a proof. The process of proving the proposition A is represented by the process of building a proof object of A. The language of type theory is extended with place holders (written as indexed question marks). The notation $? \in A$ stands for the problem of finding an object in A. An object is edited by replacing the placeholders by expressions which may contain placeholders. It is also possible to delete a subpart of an object by replacing it with a placeholder.

There is a close connection between the individual steps in proving A and the steps to build a proof object of A. When we are making a top-down proof of a proposition A, then we try to reduce the problem A to some subproblems $B_1, \ldots, B_n$ by using a rule c which takes proofs of $B_1, \ldots, B_n$ to a proof of A. Then we continue by proving $B_1, \ldots, B_n$. For instance, we can reduce the problem A to the two problems $C \supset A$ and C by using modus ponens. In this way we can continue until we have only axioms and assumptions left. This process corresponds exactly to how we can build a mathematical object from the outside and in. If we have a problem

$$? \in A$$

then it is possible to refine the place holder in the following ways:

- The placeholder can be replaced by an application $c(?_1, \ldots ?_n)$ where c is a constant, or $x(?_1, \ldots ?_n)$, where x is a variable. In the case that we have a constant, we must have that $c(?_1, \ldots ?_n) \in A$, which holds

if the type of the constant c is equal to $(x_1 \in A_1; \ldots; x_n \in A_n)B$ and $?_1 \in A_1, ?_2 \in A_2\,[x_1 \leftarrow ?_1], \ldots, x_n \in A_n[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}]$ and

$$B[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}] = A$$

So, we have reduced the problem A to the subproblems A_1, $A_2\,[x_1 \leftarrow ?_1]$, $\ldots$, $A_n[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}]$ and further refinements must satisfy the constraint $B[x_1 \leftarrow ?_1, \ldots, x_{n-1} \leftarrow ?_{n-1}] = A$. The number n of new placeholders can be computed from the arity of the constant c and the expected arity of the placeholder. As an example, if we start with $? \in A$ and A is not a function type and if we apply the constant c of type $(x \in B)C$, then the new term will be

$$c(?_1) \in A$$

where the new placeholder $?_1$ must have the type B (since all arguments to c must have that type) and furthermore the type of $c(?_1)$ must be equal to A, that is, the following equality must hold:

$$C\,[x \leftarrow ?_1] = A.$$

These kind of constraints will in general be simplified by the system. So, the editing step from $? \in A$ to $c(?_1) \in A$ is correct if $?_1 \in B$ and $C\,[x \leftarrow ?_1] = A$. This operation corresponds to applying a rule when we are constructing a proof. The rule c reduces the problem A to the problem B.

- The placeholder is replaced by an abstraction $[x]?_1$. We must have that

  $$[x]?_1 \in A$$

  which holds if A is equal to a function type $(y \in B)C$. The type of the variable x must be B and we must keep track of the fact that $?_1$ may be substituted by an expression which may depend on the variable x. This corresponds to making a new assumption, when we are constructing a proof. We reduce the general problem $(y \in B)C$ to the problem $C\,[y \leftarrow x]$ under the assumption that $x \in B$. The assumed object x can be used to construct a solution to C, that is, we may use the knowledge that we have a solution to the problem B when we are constructing a solution to the problem C.

- The placeholder is replaced by a constant c. This is correct if the type of c is equal to A.

- The placeholder is replaced by a variable x. The type of x must be equal to A. But we cannot replace a placeholder with any variable of the correct type, the variable must have been abstracted earlier.

To delete a part of a proof object corresponds to regretting some earlier steps in the proof. Notice that the deleted steps do not have to be the last steps in the derivation, by moving the pointer around in the proof object it is possible to undo any of the preceeding steps without altering the effect of following steps. However, the deletion of a sub-object is a non-trivial operation; it may cause the deletion of other parts which are depending on this.

The proof engine, which is the abstract machine representing an ongoing proof process (or an ongoing construction of a mathematical object) has two parts: the theory (which is a list of constant declarations) and the scratch area. Objects are built up in the scratch area and moved to the theory part when they are completed. There are two basic operations which are used to manipulate the scratch area. The *insertion* command replaces a placeholder by a new (possible incomplete) object and the *deletion* command replaces a sub-object by a placeholder.

When implementing type theory we have to decide what kind of inductive definitions and definitional equalities to allow. The situation is similar for both, we could give syntactic restrictions which guarantees that only meaningful definitions and equalities are allowed. We could for instance impose that an inductive definition has to be strictly positive and a definional equality has to be primitive recursive. We know, however, that any such restriction would disallow meaningful definitions. We have therefore – for the moment – no restrictions at all. This means that the correctness of a definition is the user's responsibility.

Among the examples developed in ALF, we can mention a proof that Ackermann's function is not primitive recursive [38], functional completeness of combinatorial logic [16], Tait's normalization proof for Gödel's T [17], the fundamental theorem of arithmetic [40], a constructive version of Ramsey's theorem [15], and a semantical analysis of simply typed lambda calculus with explicit substitution [7].

References

[1] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[2] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. In *Proceedings of the Workshop on General Logic, Edinburgh.* Laboratory for the Foundations of Computer Science, University of Edinburgh, February 1987.

[3] R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[4] E. Bishop. Mathematics as a numerical language. In Myhill, Kino, and Vesley, editors, *Intuitionism and Proof Theory*, pages 53–71, Amsterdam, 1970. North Holland.

[5] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[7] C. Coquand. From Semantics to Rules: a Machine Assisted Analysis. In Börger, Gurevich, and Meinke, editors, *CSL'93*, pages 91–105. Springer-Verlag, LNCS 832, 1994.

[8] T. Coquand and G. Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[9] T. Coquand and C. Paulin-Mohring. Inductively defined types. In *Proceedings of the Workshop on Programming Logic, Båstad*, 1989.

[10] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[11] N. G. de Bruijn. The Mathematical Language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61, Versailles, France, 1968. IRIA, Springer-Verlag.

[12] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606, New York, 1980. Academic Press.

[13] G. Dowek, A. Felty, H. Herbelin, H. Huet, G. P. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide version 5.6. Technical report, Rapport Technique 134, INRIA, December 1991.

[14] P. Dybjer. Inductive families. *Formal Aspects of Computing*, pages 440–465, 1994.

[15] D. Fridlender. Ramsey's theorem in type theory. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, October 1993.

[16] V. Gaspes. Formal Proofs of Combinatorial Completeness. In *To appear in the informal proceedings from the logical framework workshop at Båstad*, June 1992.

[17] V. Gaspes and J. M. Smith. Machine Checked Normalization Proofs for Typed Combinator Calculi. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[18] K. Gödel. Über eine bisher noch nicht benutze erweitrung des finiten standpunktes. *Dialectica*, 12, 1958.

[19] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[20] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

[21] A. Heyting. *Intuitionism: An Introduction.* North-Holland, Amsterdam, 1956.

[22] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[23] P. Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[24] L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system*, volume 83 of *Mathematical Centre Tracts.* Mathematisch Centrum, Amsterdam, 1979.

[25] A. N. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Matematische Zeitschrift*, 35:58–65, 1932.

[26] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical report, LFCS Technical Report ECS-LFCS-92-211, 1992.

[27] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[28] P. Martin-Löf. A Theory of Types. Technical Report 71–3, University of Stockholm, 1971.

[29] P. Martin-Löf. Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland Publishing Company, 1971.

[30] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[31] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[32] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[33] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[34] L. C. Paulson. *Logic and Computation.* Cambridge University Press, 1987.

[35] K. Petersson. A Programming System for Type Theory. PMG report 9, Chalmers University of Technology, S–412 96 Göteborg, 1982, 1984.

[36] D. Prawitz. *Natural Deduction.* Almquist & Wiksell, Stockholm, 1965.

[37] J. M. Smith. The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes. *Journal of Symbolic Logic*, 53(3), 1988.

[38] N. Szasz. A Machine Checked Proof that Ackermann's Function is not Primitive Recursive. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, June 1991. Also in G. Huet and G. Plotkin, editors, Logical Frameworks, Cambridge University Press.

[39] W. Tait. Infinitely long terms of transfinite type. In *Formal systems and recursive functions*, pages 176–185, Amsterdam, 1965. North-Holland.

[40] B. von Sydow. A machine-assisted proof of the fundamental theorem of arithmetic. PMG Report 68, Chalmers University of Technology, June 1992.

PART II:

Foundations

Gilles Dowek:

Mixing deductions and computations

Introduction to Proof Theory

Gilles Dowek

École polytechnique and INRIA
LIX, École polytechnique
91128 Palaiseau Cedex, France
Gilles.Dowek@polytechnique.fr
http://www.lix.polytechnique.fr/~dowek

Contents

1 Predicate Logic **7**
- 1.1 Languages . . . 7
  - 1.1.1 Terms and propositions . . . 9
  - 1.1.2 Variables and substitutions . . . 10
- 1.2 Proofs . . . 13
  - 1.2.1 Proofs *à la* Hilbert . . . 13
  - 1.2.2 The deduction lemma . . . 15
  - 1.2.3 Natural deduction . . . 16
  - 1.2.4 Constructive proofs . . . 21
- 1.3 Models . . . 23

2 Extensions of predicate logic **27**
- 2.1 Many-sorted predicate logic . . . 27
- 2.2 Predicate logic modulo . . . 29
  - 2.2.1 Deduction rules . . . 29
  - 2.2.2 Congruences defined by rewrite rules . . . 31
- 2.3 Binding logic . . . 33

3 Type theory **35**
- 3.1 Naive set theory . . . 35
- 3.2 Set theory . . . 38
- 3.3 Simple type theory . . . 39
- 3.4 Infinity . . . 43
- 3.5 More axioms . . . 45
  - 3.5.1 Extensionality . . . 45
  - 3.5.2 Descriptions . . . 45
- 3.6 Type theory with a binder . . . 46

4 Cut elimination in predicate logic **51**
- 4.1 Uniform proofs . . . 51
- 4.2 Cuts and cut elimination . . . 53
- 4.3 Proofs as terms . . . 56
- 4.4 Cut elimination . . . 61
- 4.5 Harrop theories . . . 66

5 Cut elimination in predicate logic modulo **69**
5.1 Congruences defined by a system rewriting atomic propositions . 69
5.2 Proof as terms 70
5.3 Counterexamples 71
5.4 Reducibility candidates 72
5.5 Pre-model 74
5.6 Pre-model construction 79
5.6.1 The term case 79
5.6.2 Quantifier free rewrite systems 80
5.6.3 Positive rewrite systems 80
5.6.4 Type theory and type theory with infinity 81

Introduction

There is something special about the mathematical discourse : each assertion must be justified by a proof. A proof is a sequence of assertions produced from the previous ones by deduction rules. The deduction rules are thus the "rules of the game" that mathematicians play.

Euclid's *Elements* (III$^{\text{rd}}$ century B.C.) are usually considered as the first systematic development where each assertion is given a proof, however, the precise definition of the notion of proof has only been formulated at the beginning of the XX$^{\text{th}}$ century. Having a definition, and not just an informal idea of what a correct proof is, is important in several areas. First, since the middle of the XX$^{\text{th}}$ century, proofs have been used not only by mathematicians, but also by computerized proof processing systems such as proof checkers and proof search systems, and designing such a system requires a precise definition.

Having a definition is also necessary to solve some problems about proofs. This is what proof theory is about. A first type of results proof theory permits to prove is independence results: results asserting that some proposition cannot be proved in some theory, for instance that the axiom of parallels cannot be proved from the other axioms of geometry.

However, proof theory is not concerned only with the provable propositions but also with the structure of proofs themselves, for instance with the comparison of different proofs of the same theorem. One key notion in proof theory is that of *canonical*, *direct* or *cut free* proof. For instance, if we first prove two propositions A and B, to deduce the proposition $A \wedge B$ (A *and* B) and at last the proposition A, we build a proof that is not canonical, because it contains an unnecessary detour by the proposition $A \wedge B$, that has nothing to do with the problem. Such a detour is called a *cut*. The main results we prove in these course notes are that in some cases, such cuts can be eliminated and thus that all provable propositions have canonical proofs. Moreover non canonical proofs can be transformed into canonical ones in an algorithmic way.

From a philosophical point of view, these results show that proving a theorem does not require to use ideas external to the statement of the theorem, or more precisely, that the use of such external ideas is only required in some specific cases, depending on the theory. Another application of cut elimination is that studying the structure of canonical proofs permits to show that some propositions have no canonical proofs. Hence, from the cut elimination theorem, we can

deduce that they have no proof at all. We get this way independence results. Cut elimination is also used to reduce dramatically the search space of proof search algorithms, by restricting to canonical proofs. Finally, cut elimination permits to prove the witness property for constructive proofs, *i.e.* that each time we have a proof of a special form of the existence of an object verifying a property P, there is also a mathematical object, called *a witness*, for which the property P can be proved to hold. Moreover, with the cut elimination algorithm, a description of this object can be computed from the proof. This allows to use mathematics as a programming language: the cut elimination process is the execution process of this programming language.

Very often, a proof is defined as a succession of reasonning steps starting from the axioms and ending at a conclusion. With such a definition, deduction rules are just reasonning rules. This definition hides the fact that, in mathematics, proofs are not only formed with reasonning steps but also with computation steps. *Deduction modulo* is a reformulation of the axiomatic method where reasonning and computation are both fully taken into account. We can, for instance, take advantage of this distinction between reasonning and computation when designing proof seach methods. More surprisingly, we can also take advantage of this distinction in proof theory. In particular, several cut elimination theorems can then be seen as corollaries of a single general cut elimination theorem for deduction modulo. Thus deduction modulo can be used as a unifying framework to present the basic results of proof theory. This is the point of view we have taken in these course notes.

Chapter 1

Predicate Logic

1.1 Languages

A *language* permits to designate things (The Moon, the number 2, the set of even numbers, ...) and to express facts (The Moon is a satellite of the Earth, the number 2 is a member of the set of even numbers, the set of even numbers is infinite, ...). A phrase that designates a thing is called a *term*, one that expresses a fact is called a *proposition*.

The easiest way to designate a thing is to use an *individual symbol* (also called a *proper name*) such as "2". Thus, a language contains individual symbols and individual symbols are terms. But, if we want to be able to designate an infinite number of objects with a finite number of symbols, we cannot give a proper name to each object. Thus, a language must contain an other kind of symbols, called *function symbols*. A function symbol alone is not a term, but it permits to construct a term when it is applied to already constructed terms. For instance, with the individual symbol 0 and the function symbol Su (for "successor") we can designate all the natural numbers. The number zero is designated by the term 0, the number one by the term $Su(0)$ obtained by applying the function symbol Su to the term 0, the number two by the term $Su(Su(0))$, ... Some function symbols must be applied to several arguments to construct a term, for instance the symbol $+$ must be applied to two arguments. The function symbol $+$ is said to have two arguments, while the symbol Su is said to have one argument. Individual symbols can be seen as special function symbols that have zero arguments.

The simplest way to form a proposition is to apply a *predicate symbol* to one or several terms. For instance, we can form this way the proposition

$$satellite(Moon,Earth)$$

that expresses that the Moon is a satellite of the Earth. Thus, a language contains predicate symbols. The predicate symbol *satellite* that must be applied to two terms to form a proposition is said to have two arguments. A proposition

formed by application of a predicate symbol to terms is called *atomic*. More propositions can be formed with the connectors $\neg$ (not), $\wedge$ (and), $\vee$ (or) and $\Rightarrow$ (implies). It is also convenient to consider propositions $\top$ (truth) and $\bot$ (falsity). We can for instance form this way the proposition

$$prime(Su(Su(0))) \wedge \neg prime(Su(Su(Su(Su(0)))))$$

that expresses that the number two is prime and that the number four is not.

A last construction is needed for propositions such as "all men are mortal" or "some number is prime", where we express that all objects verify some predicate or that some object verify some predicate without expliciting this object. We could introduce symbols *all* and *some* and let them replace a term as an argument of a predicate symbol or a function symbol. For instance we would write

$$prime(some)$$

to express that some number is prime, in the same way that we write

$$prime(Su(Su(0)))$$

to express that the number two is prime. But, such a construction is ambiguous. Indeed, the proposition

$$some \geq all$$

may express that for all numbers there is some greater number (which is true) but also that there is some number greater than all numbers (which is false). A more precise construction is to apply the predicate symbol to a variable and indicate in a second step if this variable is universal or existential with a quantifier $\forall$ (for all) or $\exists$ (there exists). The fact that some number is prime is then expressed

$$\exists x \; prime(x)$$

The order in which these quantifiers are applied permits to resolve the ambiguities. The fact that for all numbers there is some greater number is expressed by the proposition

$$\forall x \; \exists y \; y \geq x$$

while the fact that some number is greater than all numbers (which is false) is expressed by the proposition

$$\exists y \; \forall x \; y \geq x$$

Among all the symbols used to form terms and propositions, some are the same in all languages: the connectors $\top$, $\bot$, $\neg$, $\wedge$, $\vee$ and $\Rightarrow$, the quantifiers $\forall$ and $\exists$ and the variables, while the function symbols (including the individual symbols) and the predicate symbols are specific to a given language. For instance the symbol *Moon* is used in the language of astronomy, but not in the language of geometry.

1.1.1 Terms and propositions

Definition 1.1.1 (Language) *A* language *is a set of function symbols and a set of predicate symbols. To each symbol is associated a number, called its* number of arguments.

Definition 1.1.2 (Term) *Let $\mathcal{L}$ be a language and $\mathcal{V}$ be an infinite set whose elements are called* variables. *The* terms *of the language $\mathcal{L}$ with variables $\mathcal{V}$ are defined by the following rules*

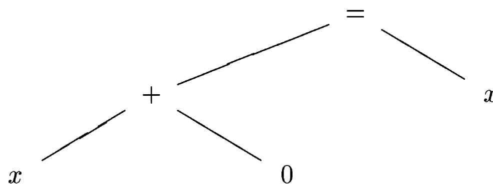
- *if x is a variable then the tree whose root is labeled by x and that has no sub-tree is a term,*
- *if f is a function symbol of n arguments and $t_1, ..., t_n$ are terms then the tree whose root is labeled by f and whose sub-trees are $t_1, ..., t_n$ is a term.*

Definition 1.1.3 (Proposition) *Let $\mathcal{L}$ be a language and $\mathcal{V}$ be an infinite set. The* propositions *of the language $\mathcal{L}$ with variables $\mathcal{V}$ are defined by the following rules*

- *if P is a predicate symbol of n arguments and $t_1, ..., t_n$ are terms then the tree whose root is labeled by P and whose sub-trees are $t_1, ..., t_n$ is a proposition,*
- *the trees whose root are labeled by $\top$ and $\bot$ and that have no sub-tree are propositions,*
- *if A is a proposition then the tree whose root is labeled by $\neg$ and whose sub-tree is A is a proposition,*
- *if A and B are propositions then the trees whose root are labeled by $\wedge$, $\vee$ or $\Rightarrow$ and whose sub-trees are A and B are propositions,*
- *if A is a proposition and x a variable then the trees whose root are labeled $\forall x$ and $\exists x$ and whose sub-tree is A are propositions.*

Remark. In several places, we shall use the notation $A \Leftrightarrow B$. There is no connector $\Leftrightarrow$ in our definition of the notion of proposition. Thus the proposition $A \Leftrightarrow B$ is just a notation for the proposition $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

Example 1.1.1 *If $=$ is a predicate symbol of two arguments, $+$ a function symbol of two arguments, 0 a function symbol of zero arguments (*i.e. *an individual symbol) and x a variable then the tree*



is a proposition.

Remark. Terms and propositions have been defined as trees whose nodes are labeled by symbols. Some authors prefer to define terms and propositions as strings, *i.e.* as sequences of symbols. The proposition of example 1.1.1 would then be written

$$= (+(x, 0), x)$$

or

$$x + 0 = x$$

This is difference is just a matter of taste.

However, the advantage of considering trees instead of strings is that this permits to disregard the shallow properties of expressions: whether + is written before, between or after its arguments, whether parentheses or brackets are used, ... and to focus on the logical structure of expressions.

1.1.2 Variables and substitutions

Definition 1.1.4 (Variables) *The set of variables of a term (resp. proposition) is defined by induction over its height as follows*

- $Var(x) = \{x\}$,
- $Var(f(t_1, ..., t_n)) = Var(t_1) \cup ... \cup Var(t_n)$,
- $Var(P(t_1, ..., t_n)) = Var(t_1) \cup ... \cup Var(t_n)$,
- $Var(\top) = Var(\bot) = \emptyset$,
- $Var(\neg A) = Var(A)$,
- $Var(A \wedge B) = Var(A \vee B) = Var(A \Rightarrow B) = Var(A) \cup Var(B)$,
- $Var(\forall x\ A) = Var(\exists x\ A) = Var(A) \cup \{x\}$.

The set of free variables of a term (resp. a proposition) is defined by induction over its height as follows

- $FV(x) = \{x\}$,
- $FV(f(t_1, ..., t_n)) = FV(t_1) \cup ... \cup FV(t_n)$,
- $FV(P(t_1, ..., t_n)) = FV(t_1) \cup ... \cup FV(t_n)$,
- $FV(\top) = FV(\bot) = \emptyset$,
- $FV(\neg A) = FV(A)$,
- $FV(A \wedge B) = FV(A \vee B) = FV(A \Rightarrow B) = FV(A) \cup FV(B)$,
- $FV(\forall x\ A) = FV(\exists x\ A) = FV(A) \setminus \{x\}$.

Definition 1.1.5 (Closed and open) *A term (resp. a proposition) that contain no free variables is said to be* closed, *otherwise it is said to be* open.

We now want to define the operation of substitution. For instance, substituting the term $y + 2$ for the variable x in the proposition $x \times 2 = 4$ yields the proposition $(y + 2) \times 2 = 4$. The result of the substitution of the term u for the variable x in the term or proposition t is written $(u/x)t$. When we substitute a term u for a variable x in a term or a proposition t, we want to substitute only the free occurrences of x. A first attempt to define substitution is the following.

Definition 1.1.6 (Replacement) *Let t be a term (resp. a proposition), x be a variable and u be a term. The term (resp. the proposition) $\langle u/x\rangle t$ is defined by induction over the height of t as follows.*

- $\langle u/x\rangle x = u$,
  if y is a variable different from x, then $\langle u/x\rangle y = y$,
  $\langle u/x\rangle f(t_1, ..., t_n) = f(\langle u/x\rangle t_1, ..., \langle u/x\rangle t_n)$,
- $\langle u/x\rangle P(t_1, ..., t_n) = P(\langle u/x\rangle t_1, ..., \langle u/x\rangle t_n)$,
  $\langle u/x\rangle \top = \top$,
  $\langle u/x\rangle \bot = \bot$,
  $\langle u/x\rangle(\neg A) = \neg\langle u/x\rangle A$,
  $\langle u/x\rangle(A \wedge B) = \langle u/x\rangle A \wedge \langle u/x\rangle B$,
  $\langle u/x\rangle(A \vee B) = \langle u/x\rangle A \vee \langle u/x\rangle B$,
  $\langle u/x\rangle(A \Rightarrow B) = \langle u/x\rangle A \Rightarrow \langle u/x\rangle B$,
  $\langle u/x\rangle(\forall x\ A) = \forall x\ A$,
  $\langle u/x\rangle(\forall y\ A) = \forall y\ \langle u/x\rangle A$ *if* $y \neq x$,
  $\langle u/x\rangle(\exists x\ A) = \exists x\ A$,
  $\langle u/x\rangle(\exists y\ A) = \exists y\ \langle u/x\rangle A$ *if* $y \neq x$.

But there is still a problem with this definition : when we replace $y + 0$ for x in $\forall y\ P(x, y)$ we obtain $\forall y\ P(y + 0, y)$ where the variable y in $y + 0$ is now quantified, while originally, this variable y had nothing to do with the variable y quantified in $\forall y\ P(x, y)$. To perform a correct substitution, we must first rename the variable y quantified in $\forall y\ P(x, y)$ to get, for instance, $\forall z\ P(x, z)$ and then substitute the variable x by $y + 0$ to get $\forall z\ P(y + 0, z)$. The choice of the variable z is arbitrary, and we could also have obtained $\forall w\ P(y + 0, w)$.

Thus, to define the substitution operation, we must first define the equivalence of two propositions modulo bound variable renaming and define substitution on the quotient of the set of propositions modulo this relation.

Definition 1.1.7 (Alphabetic equivalence) *The alphabetic equivalence between propositions is defined as follows*

- *if A and B are atomic propositions then $A \sim B$ if and only if $A = B$,*
  $\top \sim \top$,
  $\bot \sim \bot$,
  $(\neg A) \sim (\neg A')$ *if and only if* $A \sim A'$,

$(A \wedge B) \sim (A' \wedge B')$ *if and only if* $A \sim A'$ *and* $B \sim B'$,
$(A \vee B) \sim (A' \vee B')$ *if and only if* $A \sim A'$ *and* $B \sim B'$,
$(A \Rightarrow B) \sim (A' \Rightarrow B')$ *if and only if* $A \sim A'$ *and* $B \sim B'$,
$(\forall x\ A) \sim (\forall y\ A')$ *if and only if for some variable* z *not appearing in* $\forall x\ A$ *nor in* $\forall y\ A'$ $\langle z/x \rangle A \sim \langle z/y \rangle A'$,
$(\exists x\ A) \sim (\exists y\ A')$ *if and only if for some variable* z *not appearing in* $\exists x\ A$ *nor in* $\exists y\ A'$ $\langle z/x \rangle A \sim \langle z/y \rangle A'$.

From now on, propositions will be considered up to alphabetic equivalence, *i.e.* we consider only classes of propositions modulo alphabetic equivalence. So the proposition $\forall x\ (0 \leq x)$ and $\forall y\ (0 \leq y)$ are equal.

Definition 1.1.8 (Substitution) *Let* t *be a term (resp. a proposition),* x *be a variable and* u *be a term. The term (resp. the proposition)* $(u/x)t$ *is defined by induction over the height of* t *as follows*

- $(u/x)x = u$,
  if y *is a variable different from* x*, then* $(u/x)y = y$,
  $(u/x)f(t_1, ..., t_n) = f((u/x)t_1, ..., (u/x)t_n)$,
- $(u/x)P(t_1, ..., t_n) = P((u/x)t_1, ..., (u/x)t_n)$,
  $(u/x)\top = \top$,
  $(u/x)\bot = \bot$,
  $(u/x)(\neg A) = \neg (u/x)A$,
  $(u/x)(A \wedge B) = (u/x)A \wedge (u/x)B$,
  $(u/x)(A \vee B) = (u/x)A \vee (u/x)B$,
  $(u/x)(A \Rightarrow B) = (u/x)A \Rightarrow (u/x)B$,
  $(u/x)(\forall y\ A) = \forall z\ (u/x)(z/y)A$ *where* z *is a variable not appearing in* $\forall y\ A$*, not appearing in* u *and distinct from* x,
  $(u/x)(\exists y\ A) = \exists z\ (u/x)(z/y)A$ *where* z *is a variable not appearing in* $\exists y\ A$*, not appearing in* u *and distinct from* x.

We can in the same way define *simultaneous substitution.*

Definition 1.1.9 (Simultaneous substitution) *Let* t *be a term (resp. a proposition),* $x_1, ..., x_n$ *be variables and* $u_1, ..., u_n$ *be terms. Let* σ *be the finite function mapping* x_i *to* u_i*. The term (resp. the proposition)* σt *is defined by induction over the height of* t *as follows*

- $\sigma x_i = u_i$,
  if y *is a variable different from the* x_i*'s, then* $\sigma y = y$,
  $\sigma f(t_1, ..., t_n) = f(\sigma t_1, ..., \sigma t_n)$,
- $\sigma P(t_1, ..., t_n) = P(\sigma t_1, ..., \sigma t_n)$,
  $\sigma \top = \top$,
  $\sigma \bot = \bot$,
  $\sigma(\neg A) = \neg \sigma A$,
  $\sigma(A \wedge B) = \sigma A \wedge \sigma B$,

$\sigma(A \vee B) = \sigma A \vee \sigma B$,
$\sigma(A \Rightarrow B) = \sigma A \Rightarrow \sigma B$,
$\sigma(\forall y\ A) = \forall z\ \sigma(z/y)A$ *where* z *is a variable not appearing in* $\forall y\ A$ *and not appearing in* σ,
$\sigma(\exists y\ A) = \exists z\ \sigma(z/y)A$ *where* z *is a variable not appearing in* $\exists y\ A$ *and not appearing in* σ.

1.2 Proofs

We are now ready to define the tools that permit to prove propositions.

1.2.1 Proofs *à la* Hilbert

Definition 1.2.1 (Theory) *A* theory *is a set of propositions, called* axioms, *such that the membership of some proposition to this set can be decided in an algorithmic way.*

Definition 1.2.2 (Deduction rule) *A* Deduction rule *is a set of* $n+1$*-uples of propositions, such that the membership of some* $n+1$*-uples of propositions to this set can be decided in an algorithmic way. The* $n+1$*-uple* $\langle A_1, ..., A_n, B\rangle$ *is written*

$$\frac{A_1\ ...\ A_n}{B}$$

The propositions $A_1, ..., A_n$ *are called* the premises *and the proposition* B the conclusion *of the* $n+1$*-uple.*

Definition 1.2.3 (Proof) *Let* D *a set of deduction rules. A* proof *of a proposition* B *in* D *is a tree whose root is labeled by the proposition* B*, whose sub-trees are proofs of propositions* $A_1, ..., A_n$ *and such that the* $n+1$*-uple*

$$\frac{A_1\ ...\ A_n}{B}$$

is an element of one of the deduction rules of D.

Definition 1.2.4 (Logical axioms) *A* logical axiom *is a proposition of the following form where* A*,* B*,* C *are arbitrary propositions and* x *an arbitrary variable.*

$$A \Rightarrow (B \Rightarrow A)$$

$$(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$$

$$(\forall x\ (A \Rightarrow B)) \Rightarrow (A \Rightarrow \forall x\ B) \quad (\textit{if } x \notin FV(A))$$

$$\top$$

$$\bot \Rightarrow A$$

$$A \Rightarrow (\neg A \Rightarrow \bot)$$

$$(A \Rightarrow \bot) \Rightarrow \neg A$$

$$(A \wedge B) \Rightarrow A$$

$$(A \wedge B) \Rightarrow B$$

$$A \Rightarrow B \Rightarrow (A \wedge B)$$

$$A \Rightarrow (A \vee B)$$

$$B \Rightarrow (A \vee B)$$

$$(A \vee B) \Rightarrow ((A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow C))$$

$$\forall x\ A \Rightarrow (t/x)A$$

$$(t/x)A \Rightarrow \exists x\ A$$

$$\exists x\ A \Rightarrow ((\forall x\ (A \Rightarrow B)) \Rightarrow B) \quad \textit{(if } x \notin FV(B))$$

$$A \vee \neg A$$

Definition 1.2.5 (Deduction rules *à la* Hilbert) *Given a theory Γ, the deduction rules* à la *Hilbert for Γ are the following:*

- *the rule* Axiom *containing all the* 1*-uples*

$$\overline{A}$$

  where A is an element of Γ or a logical axiom,

- *the rule* Modus ponens *containing all the* 3*-uples*

$$\frac{A \Rightarrow B \quad A}{B}$$

- *the rule* Generalization *containing all the* 2*-uples*

$$\frac{A}{\forall x\ A}$$

  where x does not appear free in Γ.

These rules should be understood as follows: axioms have trivial proofs, if we have already proved $A \Rightarrow B$ and A we can deduce B, if we have already proved A with no assumption on x, we can deduce $\forall x\ A$.

Example 1.2.1 *Consider the language formed with the four proposition symbols (*i.e. *predicate symbol of zero arguments) P, Q, R and S. Consider the theory formed with the propositions*

$$P$$

$$Q$$

$$Q \Rightarrow R$$

$$P \Rightarrow (R \Rightarrow S)$$

we have the following proof of the proposition S

$$\dfrac{\dfrac{\dfrac{}{P \Rightarrow (R \Rightarrow S)}\,\textit{Axiom} \quad \dfrac{}{P}\,\textit{Axiom}}{R \Rightarrow S}\,\textit{Modus ponens} \quad \dfrac{\dfrac{}{Q \Rightarrow R}\,\textit{Axiom} \quad \dfrac{}{Q}\,\textit{Axiom}}{R}\,\textit{Modus ponens}}{S}\,\textit{Modus ponens}$$

Remark. Some authors prefer to define proofs as sequences of propositions rather than as trees. Again, this is just a matter of taste.

1.2.2 The deduction lemma

We now want to prove that a proposition $A \Rightarrow B$ has a proof in the theory Γ if and only if the proposition B has a proof in the theory Γ, A.

Proposition 1.2.1 *Let A be a proposition, the proposition $A \Rightarrow A$ has a proof in the empty theory.*

Proof. The propositions

$$(A \Rightarrow ((A \Rightarrow A) \Rightarrow A)) \Rightarrow ((A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A))$$

$$A \Rightarrow ((A \Rightarrow A) \Rightarrow A)$$

$$A \Rightarrow (A \Rightarrow A)$$

are logical axioms. Hence, the proposition $A \Rightarrow A$ has the proof

$$\dfrac{\dfrac{B \qquad A \Rightarrow ((A \Rightarrow A) \Rightarrow A)}{(A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A)}\,\text{Modus ponens} \quad A \Rightarrow (A \Rightarrow A)}{A \Rightarrow A}\,\text{Modus ponens}$$

where B is $(A \Rightarrow ((A \Rightarrow A) \Rightarrow A)) \Rightarrow ((A \Rightarrow (A \Rightarrow A)) \Rightarrow (A \Rightarrow A))$.

Proposition 1.2.2 (Deduction lemma) *The proposition $A \Rightarrow B$ has a proof in the theory Γ is and only if the proposition B has a proof in the theory Γ, A.*

Proof. If the proposition $A \Rightarrow B$ has a proof in the theory Γ, then it has a proof in the theory Γ, A. So does the proposition A. Thus, the proposition B has a proof built with the *Modus ponens* rule.

Conversely, we prove by induction over the height of the proof of B in Γ, A that there is a proof of $A \Rightarrow B$ in Γ.

- If the root of the proof is a *Axiom*, then either $B = A$ and we have a proof of $A \Rightarrow B$ by the proposition 1.2.1, or B an element of Γ and we have the proof

$$\frac{B \Rightarrow (A \Rightarrow B) \quad B}{A \Rightarrow B}\text{ Modus ponens}$$

- If the root of the proof is a *Modus ponens* then B is deduced from $C \Rightarrow B$ and C, that have smaller proofs. By induction hypothesis, there are proofs π_1 and π_2 of $A \Rightarrow (C \Rightarrow B)$ and $A \Rightarrow C$ in Γ and we take the proof

$$\dfrac{\dfrac{(A \Rightarrow (C \Rightarrow B)) \Rightarrow ((A \Rightarrow C) \Rightarrow (A \Rightarrow B)) \quad \dfrac{\pi_1}{A \Rightarrow (C \Rightarrow B)}}{(A \Rightarrow C) \Rightarrow (A \Rightarrow B)}\text{ Modus p.} \quad \dfrac{\pi_2}{A \Rightarrow C}}{A \Rightarrow B}\text{ Modus p.}$$

- If the root of the proof is a *Generalization* then we have $B = \forall x\ C$, x does not appear in Γ nor in A and C has a smaller proof. By induction hypothesis, there is a proof π of $A \Rightarrow C$ in Γ and we take the proof

$$\dfrac{(\forall x\ (A \Rightarrow C)) \Rightarrow (A \Rightarrow \forall x\ C) \quad \dfrac{\dfrac{\pi}{A \Rightarrow C}}{\forall x\ (A \Rightarrow C)}\text{ Generalization}}{A \Rightarrow \forall x\ C}\text{ Modus ponens}$$

1.2.3 Natural deduction

Introducing an hypothesis seems to be a natural step in a proof. To prove, for instance, the proposition $(n = 0) \Rightarrow (n + 1 = 1)$ we want to assume that $n = 0$ and then to prove that $n + 1 = 1$.

Proofs *à la* Hilbert do not permit to do that directly: if we have a proof of the proposition $n + 1 = 1$ using the hypothesis $n = 0$, the deduction lemma permits to transform this proof into one of the proposition $(n = 0) \Rightarrow (n + 1 = 1)$, but this proof is much longer than the proof we started with and it is not very natural.

Natural deduction is an alternative definition of the notion of proof where the introduction of an hypothesis is deduction rule. In Natural deduction, a deduction step can modify not only the proved proposition but also the theory Γ, hence a proof is not a tree of propositions, but a tree of ordered pairs $\langle \Gamma, A \rangle$ where Γ is a theory and A a proposition. Such an ordered pair is called a *sequent*

and is written $\Gamma \vdash A$ (read "Γ *entails* A"). The *Introduction* rule that permits to introduce an hypothesis transforms the sequent $\Gamma, A \vdash B$ into the sequent $\Gamma \vdash A \Rightarrow B$.

The notions of deduction rule and proof adapt straightforwardly to sequents.

Definition 1.2.6 (Deduction rule on sequents) *A* Deduction rule *is a set of $n+1$-uples of sequents, such that the membership of some $n+1$-uples of sequents to this set can be decided in an algorithmic way. The $n+1$-uple $\langle \Gamma_1 \vdash A_1, ..., \Gamma_n \vdash A_n, \Delta \vdash B \rangle$ is written*

$$\frac{\Gamma_1 \vdash A_1 \; ... \; \Gamma_n \vdash A_n}{\Delta \vdash B}$$

The sequents $\Gamma_1 \vdash A_1, ..., \Gamma_n \vdash A_n$ are called the premises *and the sequent $\Delta \vdash B$* the conclusion *of the $n+1$-uple.*

Definition 1.2.7 (Proof on sequents) *Let D a set of deduction rules. A* proof *of a sequent $\Delta \vdash B$ in D is a tree whose root is labeled by the sequent $\Delta \vdash B$, whose sub-trees are proofs of sequents $\Gamma_1 \vdash A_1, ..., \Gamma_n \vdash A_n$ and such that the $n+1$-uple*

$$\frac{\Gamma_1 \vdash A_1 \; ... \; \Gamma_n \vdash A_n}{\Delta \vdash B}$$

is an element of one of the deduction rule of D.

With the introduction rule, the three first logical axioms are now redundant, indeed the sequent $\Gamma \vdash A \Rightarrow (B \Rightarrow A)$ can be proved as follows

$$\dfrac{\dfrac{\Gamma, A, B \vdash A}{\Gamma, A \vdash B \Rightarrow A}\text{Intro}}{\Gamma \vdash A \Rightarrow (B \Rightarrow A)}\text{Intro}$$

The sequent $\Gamma \vdash (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$ can be proved as follows

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta \vdash A \Rightarrow (B \Rightarrow C) \quad \Delta \vdash A}{\Delta \vdash B \Rightarrow C}\text{Modus p.} \quad \dfrac{\Delta \vdash A \Rightarrow B \quad \Delta \vdash A}{\Delta \vdash B}\text{Modus p.}}{\Gamma, A \Rightarrow (B \Rightarrow C), A \Rightarrow B, A \vdash C}\text{Modus p.}}{\Gamma, A \Rightarrow (B \Rightarrow C), A \Rightarrow B \vdash A \Rightarrow C}\text{Intro}}{\Gamma, A \Rightarrow (B \Rightarrow C) \vdash (A \Rightarrow B) \Rightarrow (A \Rightarrow C)}\text{Intro}}{\Gamma \vdash (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))}\text{Intro}$$

where $\Delta = \Gamma, A \Rightarrow (B \Rightarrow C), A \Rightarrow B, A$. And, if the variable x appears free neither in Γ nor in A, the sequent $\Gamma \vdash (\forall x \; (A \Rightarrow B)) \Rightarrow (A \Rightarrow \forall x \; B)$ can be proved as follows

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Delta \vdash (\forall x \; (A \Rightarrow B)) \Rightarrow (A \Rightarrow B) \quad \Delta \vdash \forall x \; (A \Rightarrow B)}{\Delta \vdash A \Rightarrow B}\text{Modus p.} \quad \Delta \vdash A}{\Delta \vdash B}\text{Modus p.}}{\Gamma, \forall x \; (A \Rightarrow B), A \vdash \forall x \; B}\text{Generalization}}{\Gamma, \forall x \; (A \Rightarrow B) \vdash A \Rightarrow \forall x \; B}\text{Intro}}{\Gamma \vdash (\forall x \; (A \Rightarrow B)) \Rightarrow (A \Rightarrow \forall x \; B)}\text{Intro}$$

where $\Delta = \Gamma, \forall x \; (A \Rightarrow B), A$.

Using proof *à la* Hilbert, when we have proved the propositions A and B and we want to deduce the proposition $A \wedge B$, we must use the logical axiom $A \Rightarrow (B \Rightarrow (A \wedge B))$ and deduce $B \Rightarrow (A \wedge B)$ and then $A \wedge B$ with the *Modus ponens* rule. It is more natural to take a rule allowing to deduce directly $\Gamma \vdash A \wedge B$ from $\Gamma \vdash A$ and $\Gamma \vdash B$. As we have the rule *Introduction* this logical axiom and this rule are equivalent. As we have just seen, in a system where we have the logical axiom, we can simulate any instance of the rule and conversely, in a system where we have the rule, the axiom can be proved as follows

$$\dfrac{\dfrac{\dfrac{\Gamma, A, B \vdash A \quad \Gamma, A, B \vdash B}{\Gamma, A, B \vdash A \wedge B}\text{New rule}}{\Gamma, A \vdash B \Rightarrow (A \wedge B)}\text{Intro}}{\Gamma \vdash A \Rightarrow (B \Rightarrow (A \wedge B))}\text{Intro}$$

Excercise 1.2.1 *With proof à la Hilbert, are the logical axiom and the rule equivalent ? Hint: try to prove the Deduction lemma.*

We can suppress in a similar way all the logical axioms and replace them by deduction rules. Let us take another example. The logical axiom

$$(A \vee B) \Rightarrow ((A \Rightarrow C) \Rightarrow ((B \Rightarrow C) \Rightarrow C))$$

can be replaced by the rule

$$\dfrac{\Gamma \vdash A \vee B \quad \Gamma \vdash A \Rightarrow C \quad \Gamma \vdash B \Rightarrow C}{\Gamma \vdash C}$$

But, as it is equivalent to prove the sequent $\Gamma \vdash A \Rightarrow C$ or the sequent $\Gamma, A \vdash C$ we can transform this rule further into

$$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

In this rule, $\vee$ is the only connector or quantifier that appears explicitly. In most rules, only one connector or quantifier occurs. This permits to classify the rules according to the connector or quantifier that appears in this rule. The rules of a connector or quantifier can further be classified according to the position of this connector or quantifier. If it appears in the conclusion of the rule, then the rules is called an *introduction rule*, if it appears in a premise, then the rule is an *elimination rule*. For instance, the connector $\vee$ has two introduction rules

$$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\vee\text{-intro}$$

$$\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\vee\text{-intro}$$

and one elimination rule

$$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\vee\text{-elim}$$

The *Modus ponens*

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

is the elimination rule of implication. The *Generalization*

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A} \text{ if } x \notin FV(\Gamma)$$

is the introduction rule of the universal quantifier $\forall$. And the rule *Introduction*

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

is the introduction rule of the implication.

The system obtained this way is called *Natural Deduction.*

Definition 1.2.8 (Natural deduction)

$$\frac{}{\Gamma \vdash A}\ \mathit{Axiom} \quad \mathit{if}\ A \in \Gamma$$

$$\frac{}{\Gamma \vdash \top}\ \top\text{-}\mathit{intro}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\text{-}\mathit{elim}$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\text{-}\mathit{intro}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg\text{-}\mathit{elim}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-}\mathit{intro}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-}\mathit{elim}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-}\mathit{elim}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-}\mathit{intro}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-}\mathit{intro}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-}\mathit{elim}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-}\mathit{intro}$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \Rightarrow\text{-}elim$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash \forall x\ A} \forall\text{-}intro \quad if\ x \notin FV(\Gamma)$$

$$\frac{\Gamma \vdash \forall x\ A}{\Gamma \vdash (t/x)A} \forall\text{-}elim$$

$$\frac{\Gamma \vdash (t/x)A}{\Gamma \vdash \exists x\ A} \exists\text{-}intro$$

$$\frac{\Gamma \vdash \exists x\ A \quad \Gamma, A \vdash B}{\Gamma \vdash B} \exists\text{-}elim \quad if\ x \notin FV(\Gamma, B)$$

$$\frac{}{\Gamma \vdash A \vee \neg A} \textit{Excluded middle}$$

Proposition 1.2.3 *A proposition A has a proof* à la *Hilbert in the theory Γ if and only if the sequent $\Gamma \vdash A$ has a proof in natural deduction.*

Proof. By induction on the height of proofs.

Definition 1.2.9 (Contradictory, consistent) *A theory Γ is* contradictory *if all propositions have a proof in Γ. It is* consistent *otherwise.*

Excercise 1.2.2 *Prove that a theory Γ is contradictory if and only the proposition $\bot$ has a proof. Prove that a theory Γ is contradictory if and only there is a proposition A such that A and $\neg A$ have a proof.*

Excercise 1.2.3 *Let A be a proposition, prove that a theory that proves the proposition $A \Leftrightarrow \neg A$ is contradictory.*

Example 1.2.2 (Equality) *Given a language $\mathcal{L}$ containing a predicate symbol $=$ of two arguments, the theory of equality in this language is formed with the following axioms.*
Identity axiom:

$$\forall x\ (x = x)$$

Leibniz' axiom scheme: for each proposition A, the axiom

$$\forall x\ \forall y\ ((x = y) \Rightarrow ((x/z)A \Rightarrow (y/z)A))$$

Excercise 1.2.4 *In the theory of equality, give a proof of the proposition*

$$\forall x\ \forall y\ (x = y \Rightarrow y = x)$$

Example 1.2.3 (Arithmetic) *The language of arithmetic is formed with*

- *an individual symbol 0, a function symbol Su of one argument and two function symbols $+$ and $\times$ of two arguments*

- *a predicate symbol* $=$ *of two arguments.*

The axioms of arithmetic are the axioms of equality and the axioms:

$$\forall x\ \forall y\ (Su(x) = Su(y) \Rightarrow x = y)$$

$$\forall x\ \neg(0 = Su(x))$$

induction scheme: for each proposition A *the axiom*

$$((0/z)A \wedge (\forall x\ ((x/z)A \Rightarrow (Su(x)/z)A))) \Rightarrow \forall y\ (y/z)A$$

and the axioms

$$\forall y\ (0 + y = y)$$

$$\forall x\ \forall y\ (Su(x) + y = Su(x + y))$$

$$\forall y\ (0 \times y = 0)$$

$$\forall x\ \forall y\ (Su(x) \times y = (x \times y) + y)$$

Excercise 1.2.5 *Write a proof in arithmetic of the propositions*

$$Su(0) + Su(0) = Su(Su(0))$$

$$\forall x\ (x + 0 = x)$$

1.2.4 Constructive proofs

Definition 1.2.10 (Constructive proof) *A proof is* constructive *if it does not use the* excluded middle *rule.*

We want to prove that constructive provability and general provability are equivalent. This does not mean, of course, that all propositions that have a proof have a constructive proof, but that for each proposition A we can compute a proposition A' such that the proposition A has a proof if and only if the proposition A' has a constructive proof.

Definition 1.2.11 (Negative translation) *Let* A *be a proposition, the proposition* A' *is defined by induction over the height of* A *as follows.*

- $A' = \neg\neg A$ *if* A *is atomic,*
- $\top' = \neg\neg\top$,
- $\bot' = \neg\neg\bot$,
- $(\neg A)' = \neg\neg\neg A'$,
- $(A \wedge B)' = \neg\neg(A' \wedge B')$,
- $(A \vee B)' = \neg\neg(A' \vee B')$,

- $(A \Rightarrow B)' = \neg\neg(A' \Rightarrow B')$,
- $(\forall x\ A)' = \neg\neg(\forall x\ A')$,
- $(\exists x\ A)' = \neg\neg(\exists x\ A')$.

Proposition 1.2.4 *The proposition A has a proof if and only if A' has a constructive proof.*

Proof. (1) If a sequent $\Gamma \vdash A$ has a constructive proof π, then the sequent $\Gamma \vdash \neg\neg A$ has a constructive proof. First, we can add the hypothesis $\neg A$ to all sequents of the proof π, we obtain a proof π' of the sequent $\Gamma, \neg A \vdash A$. Then we have the following proof.

$$\dfrac{\dfrac{\Gamma, \neg A \vdash \neg A \qquad \dfrac{\pi'}{\Gamma, \neg A \vdash A}}{\Gamma, \neg A \vdash \bot}\ \neg\text{-elim}}{\Gamma \vdash \neg\neg A}\ \neg\text{-intro}$$

Thus, we can build a constructive proof of $\neg\neg\top$. From a constructive proof of $\Gamma, A \vdash \bot$ we can build a constructive proof of $\Gamma \vdash \neg\neg\neg A$. From constructive proofs of $\Gamma \vdash A$ and $\Gamma \vdash B$, we can build a constructive proof of $\Gamma \vdash \neg\neg(A \wedge B)$. From a constructive proof of $\Gamma \vdash A$, we can build a constructive proof of $\Gamma \vdash \neg\neg(A \vee B)$. From a constructive proof of $\Gamma \vdash B$, we can build a constructive proof of $\Gamma \vdash \neg\neg(A \vee B)$. From a constructive proof of $\Gamma, A \vdash B$, we can build a constructive proof of $\Gamma \vdash \neg\neg(A \Rightarrow B)$. From a constructive proof of $\Gamma \vdash A$, we can build a constructive proof of $\Gamma \vdash \neg\neg\forall x\ A$ provided x does not appear free in Γ. From a constructive proof of $\Gamma \vdash (t/x)A$, we can build a constructive proof of $\Gamma \vdash \neg\neg\exists x\ A$.

(2) Then, we check that from a constructive proofs of $\Gamma \vdash \neg\neg\bot$, we can build a constructive proof of $\Gamma \vdash \neg\neg A$. From constructive proofs of $\Gamma \vdash \neg\neg\neg A$ and $\Gamma \vdash \neg\neg A$, we can build a constructive proof of $\Gamma \vdash \neg\neg\bot$. From a constructive proof of $\Gamma \vdash \neg\neg(\neg\neg A \wedge \neg\neg B)$, we can build a constructive proof of $\Gamma \vdash \neg\neg A$ and a constructive proof of $\Gamma \vdash \neg\neg B$. From a constructive proofs of $\Gamma \vdash \neg\neg(\neg\neg A \vee \neg\neg B)$, $\Gamma, \neg\neg A \vdash \neg\neg C$ and $\Gamma, \neg\neg B \vdash \neg\neg C$ we can build a constructive proof of $\Gamma \vdash \neg\neg C$. From constructive proofs of $\Gamma \vdash \neg\neg(\neg\neg A \Rightarrow \neg\neg B)$ and $\Gamma \vdash \neg\neg A$, we can build a constructive proof of $\Gamma \vdash \neg\neg B$. From constructive proofs of $\Gamma \vdash \neg\neg(\forall x\ \neg\neg A)$, we can build a constructive proof of $\Gamma \vdash \neg\neg(t/x)A$. From constructive proofs of $\Gamma \vdash \neg\neg\exists x\ A$ and $\Gamma, \neg\neg A \vdash \neg\neg B$ we can build a constructive proof of $\Gamma \vdash \neg\neg B$ provided that x does not appear free in Γ nor in B.

As an example we show that from constructive proofs of $\Gamma \vdash \neg\neg(\neg\neg A \Rightarrow \neg\neg B)$ and $\Gamma \vdash \neg\neg A$, we can build a constructive proof of $\Gamma \vdash \neg\neg B$.

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, \neg B \vdash \neg\neg(\neg\neg A \Rightarrow \neg\neg B)} \qquad \dfrac{\dfrac{\dfrac{\Gamma, \neg B, \neg\neg A \Rightarrow \neg\neg B \vdash \neg\neg A \Rightarrow \neg\neg B \qquad \dfrac{\pi'}{\Gamma, \neg B, \neg\neg A \Rightarrow \neg\neg B \vdash \neg\neg A}}{\Gamma, \neg B, \neg\neg A \Rightarrow \neg\neg B \vdash \neg\neg B}\ \Rightarrow\text{-elim} \qquad \Gamma, \neg B, \neg\neg A \Rightarrow \neg\neg B \vdash \neg B}{\Gamma, \neg B, \neg\neg A \Rightarrow \neg\neg B \vdash \bot}\ \neg\text{-elim}}{\Gamma, \neg B \vdash \neg(\neg\neg A \Rightarrow \neg\neg B)}\ \neg\text{-intro}}{\Gamma, \neg B \vdash \bot}\ \neg\text{-elim}}{\Gamma \vdash \neg\neg B}\ \neg\text{-intro}$$

(3) We check that if A is a proposition, then the proposition $\neg\neg(A \vee \neg A)$ has a constructive proof.

$$\dfrac{\dfrac{\neg(A \vee \neg A) \vdash \neg(A \vee \neg A) \qquad \dfrac{\dfrac{\dfrac{\neg(A \vee \neg A), A \vdash \neg(A \vee \neg A) \qquad \dfrac{\neg(A \vee \neg A), A \vdash A}{\neg(A \vee \neg A), A \vdash A \vee \neg A}\ \vee\text{-i.}}{\neg(A \vee \neg A), A \vdash \bot}\ \neg\text{-e.}}{\neg(A \vee \neg A) \vdash \neg A}\ \neg\text{-intro}}{\neg(A \vee \neg A) \vdash A \vee \neg A}\ \vee\text{-intro}}{\neg(A \vee \neg A) \vdash \bot}\ \neg\text{-elim}}{\vdash \neg\neg(A \vee \neg A)}\ \neg\text{-intro}$$

(4) Then, we show that if $\Gamma \vdash A$ has a proof π then $\Gamma' \vdash A'$ has a constructive proof, by induction over the height of π. If the last rule of π is an axiom then we use the axiom rule, if the last rule is an introduction rule then we use lemma (1), if it is an elimination rule then we use lemma (2), if it the excluded middle rule, we use lemma (3).

(5) Conversely, we show that the proposition $A \Leftrightarrow \neg\neg A$ has a (not necessarily constructive) proof and we deduce that $A \Leftrightarrow A'$ has a (non necessarily constructive) proof and that if $\Gamma' \vdash A'$ has a constructive proof then $\Gamma \vdash A$ has a (not necessarily constructive) proof.

Remark. In these course notes, we shall mainly focus on constructive proofs. This does not mean that we renounce the non constructive proofs, but that non constructive proofs of a proposition A are understood as constructive proofs of its negative translation.

1.3 Models

Definition 1.3.1 (Structure) *Let $\mathcal{L}$ be a language formed with the function symbols $f_0, f_1, \ldots$ of number or arguments $n_0, n_1, \ldots$ and the predicate symbols $P_0, P_1, \ldots$ of number of arguments $m_0, m_1, \ldots$. A* structure *$\mathcal{M}$ built on $\mathcal{L}$ is a n-uple formed with*

- *a non empty set M,*
- *a function $\hat{f}_0$ from M^{n_0} to M, a function $\hat{f}_1$ from M^{n_1} to M, ...*
- *a function $\hat{P}_0$ from M^{m_0} to $\{0,1\}$, a function $\hat{P}_1$ from M^{m_1} to $\{0,1\}$, ...*

Definition 1.3.2 (Assignment) *An* assignment *over the set of variables $\mathcal{V}$ is a function from $\mathcal{V}$ to M. If ϕ is an assignment, x a variable and a an element of M, then $\phi + \langle x, a\rangle$ is the assignment mapping x to a and y to $\phi(y)$ when y is distinct from x.*

Definition 1.3.3 (Denotation) *Let $\mathcal{L}$ be a language, $\mathcal{V}$ be a set of variables and $\mathcal{M}$ be a structure built on $\mathcal{L}$. Let ϕ be an assignment and t be a term (resp. a proposition), the denotation of t in $\mathcal{M}$ modulo ϕ is defined by induction over the height of t.*

- $|x|_\phi = \phi(x)$,
  $|f_i(t_1, ..., t_{n_i})|_\phi = \hat{f}_i(|t_1|_\phi, ..., |t_{n_i}|_\phi)$,

- $|P_i(t_1, ..., t_{n_i})|_\phi = \hat{P}_i(|t_1|_\phi, ..., |t_{n_i}|_\phi)$,
  $|\top|_\phi = 1$,
  $|\bot|_\phi = 0$,
  $|\neg A|_\phi = 1$ *if* $|A|_\phi = 0$, *and* 0 *otherwise,*
  $|A \wedge B|_\phi = 1$ *if* $|A|_\phi = 1$ *and* $|B|_\phi = 1$, *and* 0 *otherwise,*
  $|A \vee B|_\phi = 1$ *if* $|A|_\phi = 1$ *or* $|B|_\phi = 1$, *and* 0 *otherwise,*
  $|A \Rightarrow B|_\phi = 1$ *if* $|A|_\phi = 0$ *or* $|B|_\phi = 1$, *and* 0 *otherwise,*
  $|\forall x\ A|_\phi = 1$ *if for all elements* a *of* M, $|A|_{\phi+\langle x,a\rangle} = 1$, *and* 0 *otherwise*
  $|\exists x\ A|_\phi = 1$ *if there is an element* a *of* M *such that* $|A|_{\phi+\langle x,a\rangle} = 1$, *and* 0 *otherwise.*

Definition 1.3.4 (Validity, model) *Let* $\mathcal{L}$ *be a language,* $\mathcal{V}$ *be a set of variables and* $\mathcal{M}$ *be a structure built on* $\mathcal{L}$. *A proposition* P *is* valid *in* $\mathcal{M}$ *is for all assignments* ϕ, $|P|_\phi = 1$. *A theory* Γ *is valid in* $\mathcal{M}$ *if all its axioms are valid. The structure* $\mathcal{M}$ *is a* model *of* Γ *if* Γ *is valid in* $\mathcal{M}$.

Proposition 1.3.1 (Soundness) *Let* Γ *be a theory. If the proposition* P *has a proof in* Γ, *then it is valid in all the models of* Γ.

Proof. By induction over the height of a proof of P in Γ.

Corollary 1.3.2 *If the theory* Γ *has a model in which* P *is not valid then* P *has no proof in* Γ.

Corollary 1.3.3 *If* Γ *has a model then* Γ *is consistent.*

Example 1.3.1 *Consider the language containing two predicate symbol* $=$ *and* $\leq$ *of two arguments. Consider the theory* $\mathcal{O}$ *formed with the axioms of equality and*

$$\forall x\ (x \leq x)$$

$$\forall x\ \forall y\ ((x \leq y \wedge y \leq x) \Rightarrow x = y)$$

$$\forall x\ \forall y\ \forall z\ ((x \leq y \wedge y \leq z) \Rightarrow x \leq z)$$

From these axiom we cannot deduce the proposition

$$\forall x\ \forall y\ (x \leq y \vee y \leq x)$$

Indeed, consider the structure $\mathcal{M} = \langle \mathbb{N}, I, | \rangle$ *where* $I(n, m) = 1$ *if* $n = m$ *and* 0 *otherwise,* $|(n, m) = 1$ *if* n *is a divisor of* m *and* 0 *otherwise. The structure* $\mathcal{M}$ *is a model of* $\mathcal{O}$. *But it is not a model of the proposition* $\forall x\ \forall y\ (x \leq y \vee y \leq x)$, *because* 2 *is not a divisor of* 3 *and* 3 *is not a divisor of* 2.

Remark. The first use of the notion of model to prove that some proposition has no proof in a theory is probably that of F. Klein who has built in 1871 a model of all the axioms of Euclid's geometry except the axiom of parallels, showing that the axiom of parallels cannot be deduced from the other axioms of Euclid's geometry. (However the notion of model has only been defined by A. Tarski, more than fifty years later, in 1936).

The soundness theorem has a converse we shall not prove here.

Proposition 1.3.4 (Gödel's completeness theorem) *Let* Γ *be a theory. If the proposition* P *is valid in all the models of* Γ *then it has a proof in* Γ.

Remark. The soundness theorem holds also for constructive proofs. But not the completeness theorem. For instance, let P be a proposition symbol (*i.e.* a predicate symbol of zero arguments). We shall see (exercise 4.1.1) that the proposition $P \vee \neg P$ has no constructive proof, but it is valid in all models. The notion of model needs to be adapted for constructive proofs.

Remark. In proof theory, the notion of model is mostly used to prove independence results, *i.e.* that some propositions have no proof in some theories. The notion of model is also used in algebra. For instance, ordered sets can be defined as the models of the theory $\mathcal{O}$ of example 1.3.1. Groups can also be defined as the models of some theory, but it can be shown that Archimedian complete ordered fields cannot be defined as the models of some theory. This fact may be used to prove, for instance, that there are ordered sets or groups of all infinite cardinals, while it is known that all Archimedian complete ordered fields are isomorphic to $\mathbb{R}$ and thus that they all have cardinal $2^{\aleph_0}$. The branch of mathematics that studies these applications of logic to algebra is called *model theory.*

Remark. A common misconception is that the notion of model can be used, as an alternative to the notion of proof, to define the notion of mathematical truth, *i.e.* that instead of saying that a proposition is true if it has a proof, we could say that it is true if it is valid in all models. The problem with such a definition of truth is that, unlike the fact that a tree is a proof of some proposition, the fact that a proposition is valid in all models is not self evident, *i.e.* it cannot be checked in an algorithmic way. Thus, the fact that some proposition is valid in all models must itself be justified by some argument. Thus, such a definition of truth reduces the question of the truth of the proposition "P" to that of the proposition "the proposition P is valid in all models" and trying to justify some proposition we enter into an infinite regression.

Remark. (Many-valued model) In the definition 1.3.1, the truth value 0 is used as denotation of non valid propositions, and the truth value 1 as denotation of valid propositions. This definition can be extended by adding other truth values. A common extension is to take a third value for propositions whose validity is unknown in this model.

Chapter 2

Extensions of predicate logic

2.1 Many-sorted predicate logic

In some theories, we want to distinguish several sorts of objects. For instance, in a language with the individual symbols $German$, $English$, $French$, $Germany$, $United-Kingdom$, $Ireland$, $France$ and a predicate L, we can form the propositions

$$L(German, Germany)$$

$$L(English, United-Kingdom)$$

$$L(English, Ireland)$$

$$L(French, France)$$

expressing that $German$ is an official language of $Germany$, ... In this theory, we can also form the unwanted proposition

$$L(Germany, Germany)$$

An extension of predicate logic permits to restrict the term and proposition formation rules, in such a way that such unwanted propositions are avoided.

Definition 2.1.1 (Many-sorted language) *A* language *is a set of sorts, a set of function symbols and a set of predicate symbols. To each function symbol is associated a $n+1$-uple of sorts $\langle s_1, ..., s_n, s_{n+1}\rangle$ called its* rank *and to each predicate symbol is associated a n-uple of sorts $\langle s_1, ..., s_n\rangle$ called its* rank.

Definition 2.1.2 (Term in a many-sorted language) *Let $\mathcal{L}$ be a many-sorted language and $\mathcal{V}_s$ be a a family of disjoint infinite sets indexed by sorts. The* terms *of the language $\mathcal{L}$ with variables $\mathcal{V}_s$ are defined by the following rules*

- *if x is a variable of $\mathcal{V}_s$ then the tree whose root is labeled by x and that has no sub-tree is a term of sort s,*
- *if f is a function symbol of rank $\langle s_1, ..., s_n, s_{n+1}\rangle$ and $t_1, ..., t_n$ are terms of sort $s_1, ..., s_n$ then the tree whose root is labeled by f and whose sub-trees are $t_1, ..., t_n$ is a term of sort s_{n+1}.*

Definition 2.1.3 (Proposition in a many-sorted language) *Let $\mathcal{L}$ be a many-sorted language and $\mathcal{V}_s$ be a a family of disjoint infinite sets indexed by sorts. The* propositions *of the language $\mathcal{L}$ with variables $\mathcal{V}_s$ are defined by the following rules*

- *if P is a predicate symbol of rank $\langle s_1, ..., s_n\rangle$ and $t_1, ..., t_n$ are terms of sort $s_1, ..., s_n$, then the tree whose root is labeled by P and whose sub-trees are $t_1, ..., t_n$ is a proposition,*
- *the trees whose root are labeled by $\top$ and $\bot$ and that have no sub-tree are propositions,*
- *if A is a proposition then the tree whose root is labeled by $\neg$ and whose sub-tree is A is a proposition,*
- *if A and B are propositions then the trees whose root are labeled by $\wedge$, $\vee$ or $\Rightarrow$ and whose sub-trees are A and B are propositions,*
- *if A is a proposition and x a variable then the trees whose root are labeled $\forall x$ and $\exists x$ and whose sub-tree is A are propositions.*

The definition of a substitution is restricted in such a way that a variable of sort s can only be substituted by a term of sort s. The proof rules are the same than in ordinary predicate logic.

Definition 2.1.4 (Structure in a many-sorted language) *Let $\mathcal{L}$ be a language formed with the sorts $s_0, s_1, ...$, the function symbols $f_0, f_1, ...$ of number or arguments and the predicate symbols $P_0, P_1, ...$. A* structure *$\mathcal{M}$ built on $\mathcal{L}$ is a n-uple formed with*

- *a family of non empty sets $M_{s_0}, M_{s_1}, ...$,*
- *a function $\hat{f}_0$ from $M_{s_1} \times ... \times M_{s_n}$ to $M_{s_{n+1}}$ where $\langle s_1, ..., s_n, s_{n+1}\rangle$ is the rank of f_0, a function $\hat{f}_1$ ...*
- *a function $\hat{P}_0$ from $M_{s_1} \times ... \times M_{s_n}$ to $\{0, 1\}$ where $\langle s_1, ..., s_n\rangle$ is the rank of P_0, a function $\hat{P}_1$ ...*

The denotation of a term and a proposition is defined in the same way as in ordinary predicate logic, with the extra condition that in the case of quantifiers, the object a belongs to M_s where s is the sort of the quantified variable.

Proposition 2.1.1 (Soundness and completeness) *A proposition has a proof in a theory if and only if it is valid in all the models of this theory.*

Remark. Predicate logic is a particular case of many-sorted predicate logic with a single sort.

2.2 Predicate logic modulo

In predicate logic, proofs are sequences of deduction steps. The idea of predicate logic modulo is that a proof is not a sequence of deduction steps, but a sequence of deduction steps *and of computation steps.* For instance, in arithmetic, to prove the proposition

$$\exists x\ (2 \times x = 4)$$

we use the $\exists$-intro rule and we are reduced to prove the proposition $2 \times 2 = 4$. Then, we have to use the axioms of addition and multiplication to prove this proposition. In predicate logic modulo, we can simply compute the term 2×2 and obtain the proposition $4 = 4$ that can easily be proved with the identity axiom.

2.2.1 Deduction rules

Definition 2.2.1 *A relation $\equiv$ defined on terms and propositions of a language is a* congruence *if*

- *it is an equivalence relation,*
- *it is compatible with all function symbols, predicate symbols, connectors and quantifiers,* i.e. *if $t \equiv u$ then $f(t) \equiv f(u)$, if $A \equiv B$ and $A' \equiv B'$ then $A \wedge A' \equiv B \wedge B'$, if $A \equiv B$ then $\forall x\ A \equiv \forall x\ B$, ...*

In predicate logic modulo a *theory* is formed with a set of axioms Γ such that the membership of some proposition to this set can be decided in an algorithmic way *and* a congruence $\equiv$ on terms and propositions such that the equivalence of two propositions can be decided in an algorithmic way. Before or after each deduction step, we can transform the proved proposition into any equivalent one. The deduction rules are thus modified to take these computations into account. These rules permit to prove sequents of the form $\Gamma \vdash_{\equiv} A$. A proposition is said to have a proof in the theory $\Gamma, \equiv$ if the sequent $\Gamma \vdash_{\equiv} A$ has a proof with the following deduction rules.

Definition 2.2.2 (Deduction rules modulo)

$$\frac{}{\Gamma \vdash_{\equiv} B}\ \textit{Axiom if } A \in \Gamma \textit{ and } A \equiv B$$

$$\frac{}{\Gamma \vdash_{\equiv} A}\ \top\textit{-intro if } A \equiv \top$$

$$\frac{\Gamma \vdash_{\equiv} B}{\Gamma \vdash_{\equiv} A} \perp\text{-}elim \textit{ if } B \equiv \perp$$

$$\frac{\Gamma, A \vdash_{\equiv} B}{\Gamma \vdash_{\equiv} C} \neg\text{-}intro \textit{ if } B \equiv \perp \textit{ and } C \equiv \neg A$$

$$\frac{\Gamma \vdash_{\equiv} C \quad \Gamma \vdash_{\equiv} A}{\Gamma \vdash_{\equiv} B} \neg\text{-}elim \textit{ if } C \equiv \neg A \textit{ and } B \equiv \perp$$

$$\frac{\Gamma \vdash_{\equiv} A \quad \Gamma \vdash_{\equiv} B}{\Gamma \vdash_{\equiv} C} \wedge\text{-}intro \textit{ if } C \equiv (A \wedge B)$$

$$\frac{\Gamma \vdash_{\equiv} C}{\Gamma \vdash_{\equiv} A} \wedge\text{-}elim \textit{ if } C \equiv (A \wedge B)$$

$$\frac{\Gamma \vdash_{\equiv} C}{\Gamma \vdash_{\equiv} B} \wedge\text{-}elim \textit{ if } C \equiv (A \wedge B)$$

$$\frac{\Gamma \vdash_{\equiv} A}{\Gamma \vdash_{\equiv} C} \vee\text{-}intro \textit{ if } C \equiv (A \vee B)$$

$$\frac{\Gamma \vdash_{\equiv} B}{\Gamma \vdash_{\equiv} C} \vee\text{-}intro \textit{ if } C \equiv (A \vee B)$$

$$\frac{\Gamma \vdash_{\equiv} D \quad \Gamma, A \vdash_{\equiv} C \quad \Gamma, B \vdash_{\equiv} C}{\Gamma \vdash_{\equiv} C} \vee\text{-}elim \textit{ if } D \equiv (A \vee B)$$

$$\frac{\Gamma, A \vdash_{\equiv} B}{\Gamma \vdash_{\equiv} C} \Rightarrow\text{-}intro \textit{ if } C \equiv (A \Rightarrow B)$$

$$\frac{\Gamma \vdash_{\equiv} C \quad \Gamma \vdash_{\equiv} A}{\Gamma \vdash_{\equiv} B} \Rightarrow\text{-}elim \textit{ if } C \equiv (A \Rightarrow B)$$

$$\frac{\Gamma \vdash_{\equiv} A}{\Gamma \vdash_{\equiv} B} \langle x, A\rangle \ \forall\text{-}intro \textit{ if } B \equiv (\forall x \ A) \textit{ and } x \notin FV(\Gamma)$$

$$\frac{\Gamma \vdash_{\equiv} B}{\Gamma \vdash_{\equiv} C} \langle x, A, t\rangle \ \forall\text{-}elim \textit{ if } B \equiv (\forall x \ A) \textit{ and } C \equiv (t/x)A$$

$$\frac{\Gamma \vdash_{\equiv} C}{\Gamma \vdash_{\equiv} B} \langle x, A, t\rangle \ \exists\text{-}intro \textit{ if } B \equiv (\exists x \ A) \textit{ and } C \equiv (t/x)A$$

$$\frac{\Gamma \vdash_{\equiv} C \quad \Gamma, A \vdash_{\equiv} B}{\Gamma \vdash_{\equiv} B} \langle x, A\rangle \ \exists\text{-}elim \textit{ if } C \equiv (\exists x \ A) \textit{ and } x \notin FV(\Gamma, B)$$

$$\frac{}{\Gamma \vdash_{\equiv} A} B \textit{ Excluded middle if } A \equiv (B \vee \neg B)$$

Proposition 2.2.1 (Equivalence) *For every congruence $\equiv$ there is a theory $\mathcal{T}$ such that $\Gamma \vdash_{\equiv} A$ if and only if $\mathcal{T}\Gamma \vdash A$.*

Proof. We take, for instance, all the axioms of the form $\forall x_1 \ldots \forall x_n \ (A \Leftrightarrow B)$ where $A \equiv B$.

Definition 2.2.3 (Model of a theory modulo) *A structure $\mathcal{M}$ is a model of a theory modulo $\Gamma, \equiv$ if all the axioms of Γ are valid in $\mathcal{M}$ and each time two terms (resp. propositions) are congruent they have the same denotation in $\mathcal{M}$.*

Proposition 2.2.2 (Soundness and completeness) *A proposition has a proof in a theory if and only if it is valid in all the models of this theory.*

2.2.2 Congruences defined by rewrite rules

Congruences used in predicate logic modulo are often defined by rewrite systems.

Definition 2.2.4 (Rewrite rule, rewrite system) *A* rewrite rule *is an ordered pair of terms or an ordered pair of propositions $\langle l, r \rangle$ written $l \longrightarrow r$. A* rewrite system *is a set of rewrite rules.*

Definition 2.2.5 (Redex) *Let $\mathcal{R}$ be a rewrite system and t be a term. The term t is a* redex *(reducible expression) if there exists a rule $l \longrightarrow r$ in $\mathcal{R}$ and a substitution σ such that $t = \sigma l$. A term t is said to contain a redex if one of its sub-terms is a redex.*

Definition 2.2.6 (One step reduction) *Let $\mathcal{R}$ be a rewrite system. A term (resp. a proposition) t reduces to a term (resp. a proposition) u in one step ($t \longrightarrow^1 u$) if there is a sub-term t' of t and a substitution σ such that $t' = \sigma l$ and u is obtained by replacing in t the sub-term t' by the term σu.*

Definition 2.2.7 (Reduction sequence) *Let $\mathcal{R}$ be a rewrite system. A* reduction sequence *is a finite or infinite sequence of terms (resp. propositions) $t_0, t_1, ...$ such that for every i, $t_i \longrightarrow^1 t_{i+1}$.*

Definition 2.2.8 (Reduction) *Let $\mathcal{R}$ be a rewrite system. A term (resp. a proposition) t reduces to a term (resp. a proposition) u ($t \longrightarrow u$) if there is a finite reduction sequence starting on t and ending on u.*

Definition 2.2.9 (Congruence sequence) *Let $\mathcal{R}$ be a rewrite system. A* congruence sequence *is a finite or infinite sequence of terms (resp. propositions) $t_0, t_1, ...$ such that for every i, $t_i \longrightarrow^1 t_{i+1}$ or $t_{i+1} \longrightarrow^1 t_i$.*

Definition 2.2.10 (Congruence) *Let $\mathcal{R}$ be a rewrite system. Two terms (resp. two propositions) t and u are congruent if there is a finite congruence sequence starting on t and ending on u.*

Definition 2.2.11 (Normal term) *A term (resp. a proposition) is* normal *if it contains no redex. A term (resp. a proposition) u is a* normal form *of a term (resp. a proposition) t if $t \longrightarrow u$ and u is normal.*

Definition 2.2.12 (Terminating) *A term (resp. a proposition) is* terminating *if it has a normal form,* i.e. *if there exists a finite reduction sequence starting on this term and ending on a normal term. It is* strongly terminating *if all reduction sequences issued from this term are finite.*

A rewrite system is terminating (resp. strongly terminating) if all terms and all propositions are terminating (resp. strongly terminating).

Definition 2.2.13 (Confluent) *A rewrite system is* confluent *if whenever a term (resp. proposition) t reduces to two terms (resp. proposition) u_1 and u_2, then there exists a term (resp. proposition) v such that u_1 reduces to v and u_2 reduces to v.*

Proposition 2.2.3 *In a confluent rewrite system, two terms (resp. two propositions) are congruent if and only if they reduce to a common term.*

Proof. By induction on the length of the congruence sequence.

Proposition 2.2.4 *In a confluent rewrite system a term has at most one normal form.*

Proof. If u_1 and u_2 are normal forms of t, then $t \longrightarrow u_1$ and $t \longrightarrow u_2$. By confluence, there exists a term v such that $u_1 \longrightarrow v$ and $u_2 \longrightarrow v$. As u_1 and u_2 are normal $u_1 = v = u_2$.

Proposition 2.2.5 *In a terminating and confluent rewrite system a term has exactly one normal form. And this normal form can be computed form the term.*

Proof. Termination yields existence and confluence unicity. To compute the normal form, it is sufficient to reduce the term until a normal form is reached.

Proposition 2.2.6 *In a terminating and confluent rewrite system two terms (resp. propositions) are congruent if they have the same normal form.*

Proof. If the two terms have the same normal form, then they are congruent. If they are congruent, so are their normal forms and these two normal forms reduce to a common term. Hence they are equal.

Proposition 2.2.7 *In a terminating and confluent rewrite system, the congruence can be checked in an algorithmic way.*

Proof. Congruence can be checked by computing the normal forms and checking their identity.

Example 2.2.1 (A presentation of arithmetic in predicate logic modulo) *To formulate arithmetic in predicate logic modulo, we can keep the axioms of equality and the axioms*

$$\forall x\ \forall y\ (Su(x) = Su(y) \Rightarrow x = y)$$

$$\forall x\ \neg(0 = Su(x))$$

$$((0/z)A \wedge (\forall x\ ((x/z)A \Rightarrow (Su(x)/z)A))) \Rightarrow \forall y\ (y/z)A$$

and replace the axioms

$$\forall y\ (0 + y = y)$$

$$\forall x\ \forall y\ (Su(x) + y = Su(x + y))$$

$$\forall y\ (0 \times y = 0)$$

$$\forall x\ \forall y\ (Su(x) \times y = (x \times y) + y)$$

by the rewrite rules

$$0 + y \longrightarrow y$$
$$Su(x) + y \longrightarrow Su(x + y)$$
$$0 \times y \longrightarrow 0$$
$$Su(x) \times y \longrightarrow x \times y + y$$

Excercise 2.2.1 *Give a proof of the proposition* $\exists x\ (2 \times x = 4)$.

2.3 Binding logic

In mathematics, we use the notation $x \mapsto x + 2$ to designate the function that maps x to $x + 2$. Such a symbol is said to be a *binder*, because the variable x that is free in $x + 2$ is bound in $x \mapsto x + 2$. In predicate logic the only binders are the quantifiers $\forall$ and $\exists$ that bind variables in propositions, but there is no way to bind variables in terms and so, there is no way to form a term such as $x \mapsto t$.

Binding logic is an extension of predicate logic where function symbols and predicate symbols can bind variables in their arguments. To each function symbol or predicate symbol of n arguments is associated a *rank* $\langle k_1, ..., k_n \rangle$ where $k_1, ..., k_n$ are natural numbers. Then, if f has the rank $\langle k_1, ..., k_n \rangle$ and $t_1, ..., t_n$ are terms, we can form the term

$$f(x_1^1 ... x_{k_1}^1\ t_1, ..., x_1^n ... x_{k_n}^n\ t_n)$$

where $x_1^1, ..., x_{k_1}^1$ are bound in the term t_1, ..., $x_1^n, ..., x_{k_n}^n$ are bound in the term t_n.

In many-sorted binding logic a rank is a sequence of sequences of sorts. Then, when a function symbol f has the rank

$$\langle \langle s_1^1, ..., s_{k_1}^1, s_{k_1+1}^1 \rangle, ... \langle s_1^n, ..., s_{k_n}^n, s_{k_n+1}^n \rangle, s^{n+1} \rangle$$

$x_1^1, ..., x_{k_1}^1$ are variables of sorts $s_1^1, ..., s_{k_1}^1$, ..., $x_{k_n}^n, ..., x_{k_n}^n$ are variables of sorts $s_1^n, ..., s_{k_n}^n$ and $t_1, ..., t_n$ are terms or sorts $s_{k_1+1}^1, ..., s_{k_n+1}^n$ then the sort of the term $f(x_1^1 ... x_{k_1}^1\ t_1, ..., x_1^n ... x_{k_n}^n\ t_n)$ is s^{n+1}.

Substitution is modified in such a way that bound variables are renamed to avoid capture. Proof rules are the same than in predicate logic or predicate logic modulo. A notion of model can also be defined for binding logic, but we shall not present it here.

Chapter 3

Type theory

In arithmetic, (example 1.2.3), we can speak about the natural numbers but not about the functions mapping natural numbers to natural numbers nor about the sets of natural numbers. Thus, arithmetic is not sufficient to express mathematics and we need to build more expressive theories. Set theory and type theory (also called higher-order logic) are such theories.

3.1 Naive set theory

In the language of arithmetic, the symbol Su is a function symbol, thus, it may be used to form terms, such as $Su(0)$, but it is not itself a term. If we want to be able to speak about the function Su, we need the symbol Su to be a term and hence an individual symbol. When Su is an individual symbol, we cannot form the term $Su(0)$ anymore. Hence, we need to introduce a new function symbol α for the application of a function to its argument and write this term $\alpha(Su, 0)$.

We could also introduce a function symbol α_2 for functions of two arguments, but this is not needed. Indeed, a function f of two arguments can always be seen as a function of one argument that maps x to the function that maps y to $f(x, y)$. Thus instead of writing $\alpha_2(f, x, y)$ we can write $\alpha(\alpha(f, x), y)$.

To ease notations we shall write $(f\ x)$ for the term $\alpha(f, x)$ and $(f\ x_1\ ...\ x_n)$ for the term $(...(f\ x_1)...x_n)$.

In the same way, we want the symbols designating predicates (sets), to be terms and hence individual symbols, for instance if the individual symbol *prime* designates the set of prime numbers, to express that the number 2 is prime, we cannot write *prime*(2), but we need to introduce a new predicate symbol $\in$ and write this proposition $2 \in$ *prime*.

For terms expressing predicates of several arguments to be terms, we must also introduce symbols $\in_2$, $\in_3$, ... For predicates of zero arguments (*i.e.* propositions) to be terms, we must introduce a predicate symbol $\in_0$, also written ε. The proposition $\in_2 (R, x, y)$ expresses that x and y are related by the predicate of two arguments (relation) R. The proposition $\varepsilon(E)$ expresses that the pred-

icate of zero argument E is true. The only difference between E and $\varepsilon(E)$ is that E is a term (designating an object) while $\varepsilon(E)$ is a proposition (expressing a fact). The object E may be called the *propositional content* of the proposition $\varepsilon(E)$.

The notions of function and set are redundant. We can express a function as a functional relation (its graph), *i.e.* as a set of ordered pairs. In this case, we just need the symbol $\in$.

Conversely, we can define a set as its characteristic function, *i.e.* as the function mapping its argument to the propositional content of the fact that x belongs to the set. In this case, we just need the symbols α and ε. If E is a set and x an object, the propositional content of the fact that x belongs to E is designated by the term $(E\ x)$ and the fact that x belongs to E is expressed by the proposition $\varepsilon(E\ x)$. Thus, the proposition $x \in E$ is thus written $\varepsilon(E\ x)$. In the same way, the proposition $\in_2 (R, x, y)$ is written $\varepsilon(R\ x\ y)$, ...

Let us now turn to the making of functions and sets. Whenever we have a term t and variables $x_1, ..., x_n$, we want to consider the function $x_1, ..., x_n \mapsto t$, for instance the function $x \mapsto (3 \times x)$. This function is such that we get back t when we apply it to $x_1, ..., x_n$. Whenever we have a proposition P and variables $x_1, ..., x_n$, we want to build the predicate $\{x_1, ..., x_n \mid P\}$, for instance the set $\{x \mid \exists y\ (x = 2 \times y)\}$. This predicate is such that we get back P when we apply it to $x_1, ..., x_n$.

A solution would be to introduce for each term t and sequence of variables $x_1, ..., x_n$ an individual symbol $C_{x_1,...,x_n,t}$ and an axiom

$$(C_{x_1,...,x_n,t}\ x_1\ ...\ x_n) = t$$

and for each proposition P and sequence of variables $x_1, ..., x_n$ an individual symbol $E_{x_1,...,x_n,P}$ and an axiom

$$\varepsilon(E_{x_1,...,x_n,P}\ x_1\ ...\ x_n) \Leftrightarrow P$$

In predicate logic modulo, these axioms can be transformed into rewrite rules

$$(C_{x_1,...,x_n,t}\ u_1\ ...\ u_n) \longrightarrow (u_1/x_1, ..., u_n/x_n)t$$

$$\varepsilon(E_{x_1,...,x_n,P}\ u_1\ ...\ u_n) \longrightarrow (u_1/x_1, ..., u_n/x_n)P$$

But, not all these symbols are necessary, and we can restrict to a much smaller language.

Definition 3.1.1 (Naive set theory) *The language of naive set theory is formed with*

- *a predicate symbol ε of one argument.*
- *a function symbol α of two arguments,*
- *individual symbols S, K, $\dot{\top}$, $\dot{\bot}$, $\dot{\neg}$, $\dot{\wedge}$, $\dot{\vee}$, $\dot{\Rightarrow}$, $\dot{\forall}$ and $\dot{\exists}$.*

and the congruence defined by the rewrite rules

$$(S\ x\ y\ z) \longrightarrow ((x\ z)\ (y\ z))$$

$$(K\ x\ y) \longrightarrow x$$

$$\varepsilon(\dot{\top}) \longrightarrow \top$$

$$\varepsilon(\dot{\bot}) \longrightarrow \bot$$

$$\varepsilon(\dot{\neg}\ x) \longrightarrow \neg\varepsilon(x)$$

$$\varepsilon(\dot{\wedge}\ x\ y) \longrightarrow (\varepsilon(x) \wedge \varepsilon(y))$$

$$\varepsilon(\dot{\vee}\ x\ y) \longrightarrow (\varepsilon(x) \vee \varepsilon(y))$$

$$\varepsilon(\dot{\Rightarrow}\ x\ y) \longrightarrow (\varepsilon(x) \Rightarrow \varepsilon(y))$$

$$\varepsilon(\dot{\forall}\ x) \longrightarrow \forall y\ \varepsilon(x\ y)$$

$$\varepsilon(\dot{\exists}\ x) \longrightarrow \exists y\ \varepsilon(x\ y)$$

Proposition 3.1.1 (Comprehension) *For each term t and sequence of variables $x_1, ..., x_n$ there is a term u such that*

$$(u\ x_1\ ...\ x_n) \equiv t$$

and for each proposition P and sequence of variables $x_1, ..., x_n$ there is a term u such that

$$\varepsilon(u\ x_1\ ...\ x_n) \equiv P$$

Proof. By induction over the height of t (resp. P).

Many variants of this theory have been proposed in the History of mathematics: Cantor's set theory (1872), Frege's *Begriffschrift* (1879), Church's pure λ-calculus (1932), ... Unfortunately, all these systems are contradictory. A contradiction is given by Russell's paradox.

By proposition 3.1.1 there exists a term R such that

$$\forall x\ (\varepsilon(R\ x) \Leftrightarrow \neg\varepsilon(x\ x))$$

(take for instance $R = (S\ (K\ \dot{\neg})\ (S\ (S\ K\ K)\ (S\ K\ K)))$). The set R is the set of all sets that do not contain themselves. By definition, this set contains itself if and only if it does not, which is contradictory. More precisely, with the elimination rule of the universal quantifier $\forall$, we can deduce from this proposition the proposition

$$\varepsilon(R\ R) \Leftrightarrow \neg\varepsilon(R\ R)$$

and we have seen (exercise 1.2.3) that from such a proposition, we can prove a contradiction.

3.2 Set theory

In naive set theory, it is possible to construct functions defined on all the universe and to construct sets in comprehension with any property P. To restrict naive set theory and avoid paradoxes, we may restrict function construction in such a way that functions are defined with a domain of definition and, similarly, only subsets of already constructed sets are constructed in comprehension. Such ideas are exploited in several theories, including set theory and simple type theory.

In Zermelo's set theory and in its extension Zermelo-Fraenkel set theory, the basic notion is that of set and functions are defined as relations. Thus the language does not contain symbols α and ε, but a symbol $\in$.

When P is a proposition, it is not always possible to form the set of objects verifying the property P. This is only allowed in four cases.

- If x and y are two sets, we can form the set $\{x, y\}$ containing exactly x and y (the symbol $\{,\}$ is a function symbol),
- If x is a set we can form the set $\bigcup(x)$ containing the elements of the elements of x,
- If x is a set, we can form a set $\wp(x)$ containing the subsets of x.
- If x is a set and P is a proposition containing variables $y, z_1, ..., z_n$, we can form the subset of x of the elements y verifying P. This set can be written $f_{y,z_1,\ldots,z_n,P}(x, z_1, ..., z_n)$ where $f_{y,z_1,\ldots,z_n,P}$ is a function symbol.

The axioms are

$$z \in \{x, y\} \Leftrightarrow (z = x \vee z = y)$$

$$y \in \bigcup(x) \Leftrightarrow (\exists z\ (y \in z \wedge z \in x))$$

$$y \in \wp(x) \Leftrightarrow (\forall z\ (z \in y \Rightarrow z \in x))$$

$$y \in f_{y,z_1,\ldots,z_n,P}(x, z_1, ..., z_n) \Leftrightarrow (y \in x \wedge P)$$

There is no way to construct the set of sets that do not belong to themselves and Russell's paradox is avoided.

In predicate logic modulo, these axioms may be transformed into rewrite rules

$$t \in \{u, v\} \longrightarrow t = u \vee t = v$$

$$t \in \bigcup(u) \longrightarrow \exists z\ (t \in z \wedge z \in u)$$

$$t \in \wp(u) \longrightarrow \forall z\ (z \in t \Rightarrow z \in u)$$

$$t \in f_{y,z_1,\ldots,z_n,P}(u, v_1, ..., v_n) \longrightarrow t \in u \wedge (t/y, v_1/z_1, ..., v_n/z_n)P$$

This system does not terminate as the proposition $f_{y,\neg y \in y}(x) \in f_{y,\neg y \in y}(x)$ reduces to $f_{y,\neg y \in y}(x) \in x \wedge \neg f_{y,\neg y \in y}(x) \in f_{y,\neg y \in y}(x)$. Thus, if we call A the

proposition $f_{y,\neg y \in y}(x) \in f_{y,\neg y \in y}(x)$ and B the proposition $f_{y,\neg y \in y}(x) \in x$ we have

$$A \longrightarrow B \wedge \neg A$$

The decidability of the congruence relation generated by these rule is an open problem.

3.3 Simple type theory

Simple type theory originates from the work of A.N. Whitehead and B. Russell. It is another way to restrict naive set theory to avoid paradoxes. In this theory, the basic notion is that of function. Each function has a domain of definition and the application $(f\ t)$ can be constructed only when t belongs to the domain of the function f, otherwise it is prohibited by the syntax. Hence simple type theory is a many-sorted theory. Taking all sets as possible function domains, *i.e.* all sets as sorts, makes it difficult to decide if a term $(f\ t)$ is well-formed or not because we need to decide if the term t designates an object that belongs to the domain of f or not. Moreover as an object can belong to several set, it should have several sorts. In type theory, an object has only one sort that is the maximal set it belongs to. It is called the *type* of this object. There is one type ι for atoms and one type o for propositional contents, then each time we have two types T and U, we can form the type $T \rightarrow U$ of functions mapping objects of sort T to objects of sort U.

Definition 3.3.1 (Simple types) Simple types *are closed terms formed with the individual symbols ι and o and the function symbol $\rightarrow$ of two arguments.*

To ease notation, we write $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow U$ for the type $(T_1 \rightarrow (T_2 ... \rightarrow (T_n \rightarrow U)...))$.

Definition 3.3.2 (Language of type theory) *The language of simple type theory in predicate logic modulo is formed with*

- *a predicate symbol ε of rank $\langle o \rangle$,*
- *for each pair of type T, U, a function symbol $\alpha_{T,U}$ of rank $\langle T \rightarrow U, T, U \rangle$,*
- *for each triple of types T, U, V an individual symbol $S_{T,U,V}$ of sort $(T \rightarrow U \rightarrow V) \rightarrow (T \rightarrow U) \rightarrow T \rightarrow V$,*
  for each pair of types T, U an individual symbol $K_{T,U}$ of sort $T \rightarrow U \rightarrow T$,
  individual symbols $\dot{\top}$ and $\dot{\bot}$ of sort o,
  an individual symbol $\dot{\neg}$ of sort $o \rightarrow o$,
  individual symbols $\dot{\wedge}$, $\dot{\vee}$, $\dot{\Rightarrow}$ of sort $o \rightarrow o \rightarrow o$,
  for each type T, individual symbols $\dot{\forall}_T$ and $\dot{\exists}_T$ of type $(T \rightarrow o) \rightarrow o$.

Definition 3.3.3 (Rewrite system of type theory) *The rewrite system $\mathcal{T}$ is defined by the rules*

$$(S_{T,U,V}\ x\ y\ z) \longrightarrow ((x\ z)\ (y\ z))$$

$$(K_{T,U}\ x\ y) \longrightarrow x$$
$$\varepsilon(\dot{\top}) \longrightarrow \top$$
$$\varepsilon(\dot{\bot}) \longrightarrow \bot$$
$$\varepsilon(\dot{\neg}\ x) \longrightarrow \neg\varepsilon(x)$$
$$\varepsilon(\dot{\wedge}\ x\ y) \longrightarrow \varepsilon(x) \wedge \varepsilon(y)$$
$$\varepsilon(\dot{\vee}\ x\ y) \longrightarrow \varepsilon(x) \vee \varepsilon(y)$$
$$\varepsilon(\dot{\Rightarrow}\ x\ y) \longrightarrow \varepsilon(x) \Rightarrow \varepsilon(y)$$
$$\varepsilon(\dot{\forall}_T\ x) \longrightarrow \forall y\ \varepsilon(x\ y)$$
$$\varepsilon(\dot{\exists}_T\ x) \longrightarrow \exists y\ \varepsilon(x\ y)$$

Proposition 3.3.1 (Comprehension) *For each term t there is a term u not containing the variable x such that $(u\ x) \equiv t$. For each proposition P there is a term u such that $\varepsilon(u) \equiv A$.*

Proof. By induction over the height of t.

- If $t = x$ then we take $u = (S\ K\ K)$, we have $(u\ x) = (S\ K\ K\ x) \equiv (K\ x\ (K\ x)) \equiv x$.
- If t is a variable different from x or an individual symbol, we take $u = (K\ t)$, we have $(u\ x) = (K\ t\ x) \equiv t$.
- If $t = (t_1\ t_2)$, then by induction hypothesis, there are terms u_1 and u_2 such that $(u_1\ x) \equiv t_1$ and $(u_2\ x) \equiv t_2$. We take $u = (S\ u_1\ u_2)$. We have $(u\ x) = (S\ u_1\ u_2\ x) \equiv ((u_1\ x)\ (u_2\ x)) \equiv (t_1\ t_2) = t$.

By induction over the height of A.

- If $A = \varepsilon(t)$, we take $u = t$.
- If $A = B \wedge C$, then by induction hypothesis, there are terms v and w such that $\varepsilon(v) \equiv B$ and $\varepsilon(w) \equiv C$. We take $u = (\dot{\wedge}\ v\ w)$. We proceed the same way if $A = \top, \bot, \neg B, B \vee C$ or $B \Rightarrow C$.
- If $A = \forall x\ B$, then by induction hypothesis, there is a term v such that $\varepsilon(v) \equiv B$ and there is a term w not containing x such that $(w\ x) \equiv v$ and hence $\varepsilon(w\ x) \equiv \varepsilon(v) \equiv B$. We take $u = (\dot{\forall}\ w)$. We have $\varepsilon(u) \equiv \forall x\ \varepsilon(w\ x) \equiv \forall x\ B$. We proceed the same way if $A = \exists x\ B$.

Definition 3.3.4 (Leibniz' Equality) *By the proposition 3.3.1 there is a term $\doteq$ such that*

$$\varepsilon(\doteq\ x\ y) \equiv \forall p\ (\varepsilon(p\ x) \Rightarrow \varepsilon(p\ y))$$

Excercise 3.3.1 *Prove*

$$\forall x\ \varepsilon(x \dot{=} x)$$

and for each proposition A

$$\forall x\ \forall y\ (\varepsilon(x \dot{=} y) \Rightarrow ((x/z)A \Rightarrow (y/z)A))$$

To prove that the rewrite system $\mathcal{T}$ is terminating, we first focus on the two first rules.

Proposition 3.3.2 (Tait's theorem) *The rewrite system*

$$(S_{T,U,V}\ x\ y\ z) \longrightarrow ((x\ z)\ (y\ z))$$

$$(K_{T,U}\ x\ y) \longrightarrow x$$

is strongly terminating.

Proof. The set of *reducible* terms of type T is defined by induction over the height of T.

- If T is ι or o then t is reducible of type T if and only if it is strongly terminating.
- If $T = T_1 \to T_2$ then t is reducible of type T if and only if for every reducible term u of type T_1, the term $(t\ u)$ is reducible of type T_2.

We prove by induction over the height of T that

- (1) all reducible terms are strongly terminating and
- (2) variables and individual symbols other than S and K are reducible terms.

Let $T = U_1 \to ... \to U_n \to V$ ($V = \iota$ or $V = o$). (1) If t is a reducible term of type T, then let x_1, ..., x_n be variables of types $U_1, ..., U_n$. By induction hypothesis, the variables $x_1, ..., x_n$ are reducible. Hence, the term $(t\ x_1\ ...\ x_n)$ is reducible and its type is either ι or o. Hence it is strongly terminating and so is t. (2) If x is a variable of type T or an individual symbol of type T different from S and K, then let u_1, ..., u_n be reducible terms of types $U_1, ..., U_n$. By induction hypothesis the terms $u_1, ..., u_n$ are strongly terminating. A reduction sequence starting from $(x\ u_1\ ...\ u_n)$ reduces redexes in the terms $u_1, ..., u_n$. Hence, it is finite. The term $(x\ u_1\ ...\ x_n)$ is strongly terminating and its type is ι or o, hence it is reducible. Thus, x is reducible.

Then, we prove by induction over the height of t that every term is reducible.

- If t is a variable or an individual symbol different from S and K then it is reducible.
- If $t = (u\ v)$, then the terms u and v are reducible by induction hypothesis, and the term t is reducible.

- If $t = K$ (resp. $t = S$) then let $U_1 \rightarrow ... \rightarrow U_n \rightarrow V$ ($V = \iota$ or $V = o$) be the type of t and let $u_1, ..., u_n$ be reducible terms of types $U_1, ..., U_n$. We have to prove that the term $(K\ u_1\ ...\ u_n)$ (resp. $(S\ u_1\ ...\ u_n)$) is strongly terminating. Consider a reduction sequence $t_0, t_1, t_2, ...$ starting from the term $(K\ u_1\ ...\ u_n)$ (resp. $(S\ u_1\ ...\ u_n)$). We have to prove that this reduction sequence is finite. If the root redex is never reduced, all reductions take place in $u_1, ..., u_n$, these terms are reducible and hence strongly terminating and the reduction sequence is finite. If the root redex is reduced at step m, then the term t_m has the form $(K\ u'_1\ u'_2\ u'_3 ...\ u'_n)$ (resp. $(S\ u'_1\ u'_2\ u'_3 ...\ u'_n)$) and the term t_{m+1} is $(u'_1\ u'_3\ ...\ u'_n)$ (resp. $(u'_1\ u'_3\ (u'_2\ u'_3)\ u'_4\ ...\ u'_n)$) where u'_1 is a reduct of u_1, ..., u'_n is a reduct of u_n. The term $(u_1\ u_3\ ...\ u_n)$ (resp. $(u_1\ u_3\ (u_2\ u_3)\ u_4\ ...\ u_n)$) is reducible, hence it is strongly terminating and the term $(u'_1\ u'_3\ ...\ u'_n)$ (resp. $(u'_1\ u'_3\ (u'_2\ u'_3)\ u'_4\ ...\ u'_n)$) is strongly terminating, thus the reduction sequence $t_0, t_1, t_2, ...$ is finite. Therefore, the term K (resp. S) is reducible.

All terms are reducible, hence all terms are strongly terminating.

Proposition 3.3.3 *The rewrite system $\mathcal{T}$ is strongly terminating.*

Proof. We reduce termination in $\mathcal{T}$ to termination in the system SK. We define a translation $\|\ \|$ of the terms and the propositions of type theory into terms of type theory. In each type T, we choose a variable z_T.

- $\|x\| = x$,
- $\|S_{T,U,V}\| = S_{T,U,V}$,
  $\|K_{T,U}\| = K_{T,U}$,
- $\|(t\ u)\| = (\|t\|\ \|u\|)$,
- $\|\dot{\top}\| = \|\dot{\bot}\| = ((S\ K\ K)\ z_o)$,
  $\|\dot{\neg}\| = (S\ K\ K)$,
  $\|\dot{\wedge}\| = \|\dot{\vee}\| = \|\dot{\Rightarrow}\| = ((S\ K\ K)\ z_{o \rightarrow o \rightarrow o})$,
  $\|\dot{\forall}_T\| = \|\dot{\exists}_T\| = (S\ (S\ K\ K)\ (K\ z_T))$,
- $\|\varepsilon(t)\| = \|t\|$,
  $\|\top\| = \|\bot\| = z_o$,
  $\|\neg A\| = \|A\|$,
  $\|A \wedge B\| = \|A \vee B\| = \|A \Rightarrow B\| = (z_{o \rightarrow o \rightarrow o}\ \|A\|\ \|B\|)$,
  $\|\forall x\ A\| = \|\exists x\ A\| = \|(z_T/x)A\|$.

We check that if A rewrites in one step to B in $\mathcal{T}$, then $\|A\|$ rewrites in at least one step to $\|B\|$ in SK. If $A_0, A_1, A_2, ...$ is a reduction sequence in $\mathcal{T}$, then the sequence $\|A_0\|, \|A_1\|, \|A_2\|, ...$ is a reduction sequence in SK, thus it is finite.

Proposition 3.3.4 *The rewrite system $\mathcal{T}$ is confluent.*

Proposition 3.3.5 *Each term (resp. proposition) has a unique normal form for the rewrite system $\mathcal{T}$ and the congruence generated by this system can be checked in an algorithmic way.*

Proof. It is terminating and confluent.

Proposition 3.3.6 *Type theory has a model.*

Proof. Consider the model

$$\begin{array}{rcl} M_\iota & = & \{0\} \\ M_o & = & \{0,1\} \\ M_{T\to U} & = & M_U^{M_T} \end{array}$$

$$\begin{array}{rcl} \hat{S}_{T,U,V} & = & a \mapsto (b \mapsto (c \mapsto a(c)(b(c)))) \\ \hat{K}_{T,U} & = & a \mapsto (b \mapsto a) \\ \hat{\alpha}(a,b) & = & a(b) \\ \hat{\varepsilon}(a) & = & a \end{array}$$

$$\begin{array}{rcl} \hat{\dot{\top}} & = & 1 \\ \hat{\dot{\bot}} & = & 0 \\ \hat{\dot{\neg}}(a) & = & 1 \text{ if } a=0 \text{ and } 0 \text{ otherwise} \\ \hat{\dot{\wedge}}(a,b) & = & 1 \text{ if } a=1 \text{ and } b=1 \text{ and } 0 \text{ otherwise} \\ \hat{\dot{\vee}}(a,b) & = & 1 \text{ if } a=1 \text{ or } b=1 \text{ and } 0 \text{ otherwise} \\ \hat{\dot{\Rightarrow}}(a,b) & = & 1 \text{ if } a=0 \text{ or } b=1 \text{ and } 0 \text{ otherwise} \\ \hat{\dot{\forall}}_T(a) & = & 1 \text{ if for all } b \text{ in } M_T\ a(b)=1 \text{ and } 0 \text{ otherwise} \\ \hat{\dot{\exists}}_T(a) & = & 1 \text{ if there exists a } b \text{ in } M_T \text{ such that } a(b)=1 \text{ and } 0 \text{ otherwise} \end{array}$$

It is easy to check that $|A|_\phi = |B|_\phi$ when $A \equiv B$.

3.4 Infinity

A set is said E to be infinite if there is function f mapping elements of E to elements of E that is injective, but not surjective. In type theory this proposition $Infinite(E)$ is expressed as follows.

$$\exists a\ \exists f\ \forall x\ (\varepsilon(E\ x) \Rightarrow (E\ (f\ x))) \wedge \forall x\ \forall y\ ((\varepsilon(E\ x) \wedge \varepsilon(E\ y)$$
$$\wedge \varepsilon((f\ x) \dot{=} (f\ y))) \Rightarrow \varepsilon(x \dot{=} y)) \wedge (\forall x\ (\varepsilon(E\ x) \Rightarrow \neg\varepsilon(a \dot{=} (f\ x))))$$

Notice that the proposition $\exists E\ Infinite(E)$ is not valid in the model of proposition 3.3.6, hence it is not provable. If we replace M_ι by the set $\mathbb{N}$ in the model of proposition 3.3.6, we keep a model of type theory and the proposition $\exists E\ Infinite(E)$ is valid in this model. Thus, the proposition $\neg\exists E\ Infinite(E)$ is not valid in this model and therefore it is not provable either. Indeed, so far neither in type theory nor in set theory we have given an axiom that permits to

construct an infinite set. To be able to formalize mathematics we need to add such an axiom.

In type theory, we add an axiom expressing that the set of objects of type ι is infinite. Thus, the set E is such that $\varepsilon(E\ x) \equiv \top$ and we can formulate the axiom

$$\exists a\ \exists f\ \forall x\ \forall y\ (\varepsilon((f\ x) \dot{=} (f\ y)) \Rightarrow \varepsilon(x \dot{=} y)) \wedge (\forall x\ \neg\varepsilon(a \dot{=} (f\ x)))$$

Instead of taking an existential axiom, we can give a name to the function and to the element that is not in its image. For instance, we can call them Su and 0 and we get the two axioms

$$\forall x\ \forall y\ (\varepsilon((Su\ x) \dot{=} (Su\ y)) \Rightarrow \varepsilon(x \dot{=} y))$$

$$\forall x\ \neg\varepsilon(0 \dot{=} (Su\ x))$$

that are two of Peano's axioms.

These axioms become theorems if we add some symbols and rewrite rules.

Definition 3.4.1 (Type theory with infinity) Type theory with infinity *is the extension of type theory with individual symbols* 0 *of type* ι*,* Su *and* $Pred$ *of type* $\iota \to \iota$*, an individual symbol* $Null$ *of type* $\iota \to o$ *and the rules*

$$(Pred\ (Su\ x)) \longrightarrow x$$

$$(Null\ 0) \longrightarrow \dot{\top}$$

$$(Null\ (Su\ 0)) \longrightarrow \dot{\bot}$$

Excercise 3.4.1 *In simple type theory with infinity, prove the propositions*

$$\forall x\ \forall y\ (\varepsilon((Su\ x) \dot{=} (Su\ y)) \Rightarrow \varepsilon(x \dot{=} y))$$

$$\forall x\ \neg\varepsilon(0 \dot{=} (Su\ x))$$

Proposition 3.4.1 *Type theory with infinity has a model.*

Proof. Consider the model

$$\begin{aligned}
M_\iota &= \mathbb{N} \\
M_o &= \{0, 1\} \\
M_{T \to U} &= M_U^{M_T}
\end{aligned}$$

$$\begin{aligned}
\hat{0} &= 0, \\
\hat{Su} &= n \mapsto n + 1, \\
\hat{Pred} &= n \mapsto \text{if } n = 0 \text{ then } 0 \text{ else } n - 1, \\
\hat{Null} &= n \mapsto \text{if } n = 0 \text{ then } 1 \text{ else } 0, \\
\hat{S}_{T,U,V} &= a \mapsto (b \mapsto (c \mapsto a(c)(b(c)))) \\
\hat{K}_{T,U} &= a \mapsto (b \mapsto a) \\
\hat{\alpha}(a, b) &= a(b) \\
\hat{\varepsilon}(a) &= a
\end{aligned}$$

$$\begin{aligned}
\hat{\dot{\top}} &= 1\\
\hat{\dot{\bot}} &= 0\\
\hat{\dot{\neg}}(a) &= 1 \text{ if } a = 0 \text{ and } 0 \text{ otherwise}\\
\hat{\dot{\wedge}}(a,b) &= 1 \text{ if } a = 1 \text{ and } b = 1 \text{ and } 0 \text{ otherwise}\\
\hat{\dot{\vee}}(a,b) &= 1 \text{ if } a = 1 \text{ or } b = 1 \text{ and } 0 \text{ otherwise}\\
\hat{\dot{\Rightarrow}}(a,b) &= 1 \text{ if } a = 0 \text{ or } b = 1 \text{ and } 0 \text{ otherwise}\\
\hat{\dot{\forall}}_T(a) &= 1 \text{ if for all } b \text{ in } M_T\ a(b) = 1 \text{ and } 0 \text{ otherwise}\\
\hat{\dot{\exists}}_T(a) &= 1 \text{ if there exists a } b \text{ in } M_T \text{ such that } a(b) = 1 \text{ and } 0 \text{ otherwise}
\end{aligned}$$

It is easy to check that $|A|_\phi = |B|_\phi$ when $A \equiv B$.

There are many ways to construct the natural numbers in type theory with infinity (as finite cardinals, ...). An easy way is simply to take 0 for zero and $(Su\ n)$ for the successor of n.

Then the type ι contains all the natural numbers, but possibly also other objects. The set of natural numbers can be defined as the smallest set containing 0 and closed by successor, *i.e.* as the intersection of all such sets. An object is a member of $\mathbb{N}$ if it is a member of all sets E containing 0 and closed by successor. Thus

$$\varepsilon(\mathbb{N}\ n) = \forall E\ ((\varepsilon(E\ 0) \wedge (\forall x\ (\varepsilon(E\ x) \Rightarrow \varepsilon(E\ (Su\ x))))) \Rightarrow \varepsilon(E\ n))$$

The existence of such an object given by proposition 3.3.1.

Excercise 3.4.2 *Prove the induction theorem*

$$\forall E\ (\varepsilon(E\ 0) \wedge \forall x\ (\varepsilon(E\ x) \Rightarrow \varepsilon(E\ (Su\ x)))) \Rightarrow \forall n\ (\varepsilon(\mathbb{N}\ n) \Rightarrow \varepsilon(E\ n))$$

3.5 More axioms

3.5.1 Extensionality

In mathematics, it is usual to consider that two sets that have the same elements are equal and that two functions that are point-wise equal are equal. This leads, both in set theory and in type theory to the *axiom of extensionality.* In type theory, this axiom is stated

$$\forall f\ \forall g\ ((\forall x\ \varepsilon((f\ x) \dot{=} (g\ x))) \Rightarrow \varepsilon(f \dot{=} g))$$

$$\forall x\ \forall y\ (\varepsilon(x) \Leftrightarrow \varepsilon(y)) \Rightarrow \varepsilon(x \dot{=} y)$$

3.5.2 Descriptions

The proposition 3.3.1 permits for instance to prove the existence of a function that adds two to its arguments, *i.e.* the proposition

$$\exists f\ \forall x\ \varepsilon((f\ x) \dot{=} (Su\ (Su\ x)))$$

but, it does not permit to prove the existence of a function that takes the value 1 on 1 and the value 0 anywhere else. Indeed, it can be proved that the proposition

$$\exists f\ \forall x\ ((\varepsilon(x\dot{=}(Su\ 0)) \Rightarrow \varepsilon((f\ x)\dot{=}(Su\ 0))) \wedge (\neg\varepsilon(x\dot{=}(Su\ 0)) \Rightarrow \varepsilon((f\ x)\dot{=}0)))$$

has no proof in type theory.

In contrast, with the proposition 3.3.1, it is easy to prove the existence of the graph of this function, *i.e.* the proposition

$$\exists R\ \forall x\ \forall y\ (\varepsilon(R\ x\ y) \Leftrightarrow ((\varepsilon(x\dot{=}1) \Rightarrow \varepsilon(y\dot{=}1)) \wedge (\neg\varepsilon(x\dot{=}1) \Rightarrow \varepsilon(y\dot{=}0))))$$

and we can also prove, for instance by induction, that this relation is functional, *i.e.* that

$$\forall x\ (\varepsilon(\mathbb{N}\ x) \Rightarrow \exists^1 y\ \varepsilon(R\ x\ y))$$

But to conclude to the existence of the function we need the following axiom (descriptions axiom)

$$\forall P\ \forall Q\ (\forall x\ (\varepsilon(P\ x) \Rightarrow \exists^1 y\ \varepsilon(Q\ x\ y)) \Rightarrow \exists f\ \forall x\ (\varepsilon(P\ x) \Rightarrow \varepsilon(Q\ x\ (f\ x)))$$

that relates functions and functional relations.

In set theory, functions are functional relations, thus they need no axiom to be related.

3.6 Type theory with a binder

We have seen in proposition 3.3.1 that to have a language containing the function symbols $\alpha_{T,U}$ and the individual symbols $S_{T,U,V}$ and $K_{T,U}$ and the related rewrite rules is sufficient to prove that, for each term t and variable x there is a term u not containing the variable x such that $(u\ x) \equiv t$. But, the term u is sometimes cumbersome to compute. It is more comfortable to have a symbol $\mapsto$ such that the function mapping x to t can simply be written $x \mapsto t$. The symbol $\mapsto$ is a function symbol of one argument binding one variable in its argument. When we take the symbol $\mapsto$, the symbols S and K become superfluous ($S = x \mapsto y \mapsto z \mapsto ((x\ z)\ (y\ z))$, $K = x \mapsto y \mapsto x$). We thus get the following theory.

Definition 3.6.1 (Language of type theory with a binder) *The language of simple type theory with a binder is formed with*

- *a predicate symbol* ε *of rank* $\langle o \rangle$,
- *for each pair of type* T, U, *a function symbol* $\alpha_{T,U}$ *of rank* $\langle T \to U, T, U \rangle$,
  for each pair of types T, U *a function symbol* $\mapsto$ *of rank* $\langle \langle T, U \rangle, T \to U \rangle$,
- *individual symbols* $\dot{\top}$ *and* $\dot{\bot}$ *of sort* o,
  an individual symbol $\dot{\neg}$ *of sort* $o \to o$,
  individual symbols $\dot{\wedge}$, $\dot{\vee}$, $\dot{\Rightarrow}$ *of sort* $o \to o \to o$,
  for each type T, *individual symbols* $\dot{\forall}_T$ *and* $\dot{\exists}_T$ *of type* $(T \to o) \to o$.

Definition 3.6.2 (Rewrite system of type theory with a binder) *The rewrite system $\mathcal{T}'$ is defined by the rules*

$$((x \mapsto t)\ u) \longrightarrow (u/x)t$$

$$\varepsilon(\dot{\top}) \longrightarrow \top$$

$$\varepsilon(\dot{\bot}) \longrightarrow \bot$$

$$\varepsilon(\dot{\neg}\ x) \longrightarrow \neg\varepsilon(x)$$

$$\varepsilon(\dot{\wedge}\ x\ y) \longrightarrow \varepsilon(x) \wedge \varepsilon(y)$$

$$\varepsilon(\dot{\vee}\ x\ y) \longrightarrow \varepsilon(x) \vee \varepsilon(y)$$

$$\varepsilon(\dot{\Rightarrow}\ x\ y) \longrightarrow \varepsilon(x) \Rightarrow \varepsilon(y)$$

$$\varepsilon(\dot{\forall}_T\ x) \longrightarrow \forall y\ \varepsilon(x\ y)$$

$$\varepsilon(\dot{\exists}_T\ x) \longrightarrow \exists y\ \varepsilon(x\ y)$$

To prove that the rewrite system $\mathcal{T}'$ is terminating, we first focus on the first rule.

Proposition 3.6.1 (Tait's theorem with a binder) *The rewrite system*

$$((x \mapsto t)\ u) \longrightarrow (u/x)t$$

is strongly terminating.

Proof. The set $|T|$ of *reducible* terms of type T is defined by induction over the height of T.

- If T is ι or o then t is in $|T|$ if and only if it is strongly terminating.
- If $T = T_1 \to T_2$ then t is in $|T|$ if and only if it is strongly terminating and when its reduces to a term of the form $x \mapsto t'$ then for every term u in $|T_1|$, $(u/x)t'$ is in $|T_2|$.

To prove that all terms of type T are strongly terminating, we prove that all terms of type T are in $|T|$. More generally, we prove, by induction over the height of t, that if t is a term of type T, σ a substitution mapping variables of type U to elements of $|U|$, then σt is in $|T|$.

- If $t = y$, then if y is in the domain of σ then σt is in $|T|$. Otherwise, $\sigma t = y$, the variable y is normal, hence it is strongly terminating and it cannot reduce to a term of the form $x \mapsto t'$, hence it is in $|T|$.

- If $t = x \mapsto u$, then $T = T_1 \to T_2$. Modulo alphabetic equivalence, we can chose the variable x not appearing in σ, thus $\sigma t = x \mapsto \sigma u$. This term is strongly terminating because a reduction sequence issued from it can only reduce the term σu and, by induction hypothesis, this term is in $|T_2|$ and thus it is strongly terminating. Then, if σt reduces to the term $x \mapsto t'$, then t' is a reduct of σu. Let v be a term of $|T_2|$, the term $(v/x)t'$ is a reduct of $((v/x) \circ \sigma)u$, that is in $|T_2|$ by induction hypothesis. It is easy to check that $|T_2|$ is closed by reduction. Thus the term $(v/x)t'$ is in $|T_2|$.

  Hence, the term σt is in $|T|$.

- If $t = (t_1\ t_2)$ and t_1 is a term of type $U \to T$ and t_2 a term of type U. We have $\sigma t = (\sigma t_1\ \sigma t_2)$. By induction hypothesis σt_1 and σt_2 are in the sets $|U \to T|$ and $|U|$. To prove that σt is in $|T|$, we prove that if u_1 is in $|U \to T|$ and u_2 is in U then $(u_1\ u_2)$ is in $|T|$.

  The terms u_1 and u_2 are strongly terminating. Let n be the maximum length of a reduction sequence issued from u_1 and n' the maximum length of a reduction sequence issued from u_2. We prove that $(u_1\ u_2)$ is in $|T|$ by induction on $n + n'$.

  First we prove that $(u_1\ u_2)$ is strongly terminating. Consider a reduction sequence issued from this term. If the first redex is in u_1 or u_2 then we apply the induction hypothesis, otherwise the redex is at the root of the term $(u_1\ u_2)$, u_1 has the form $x \mapsto u'$ and the first step of the reduction sequence reduces $(u_1\ u_2)$ to $(u_2/x)u'$. This term is in $|T|$, hence it is strongly terminating and the reduction sequence is finite. Then, we prove that if $T = U_1 \to U_2$ and $(u_1\ u_2)$ reduces to a term of the form $y \mapsto v$, then for every term w in $|U_1|$, $(w/y)v$ is in $|U_2|$. As $(u_1\ u_2)$ is an application, the reduction sequence is not empty. If the first redex is in u_1 or u_2, we apply the induction hypothesis, otherwise the redex is at the root of the term $(u_1\ u_2)$, u_1 has the form $x \mapsto u'$ and the first step of the reduction sequence reduces $(u_1\ u_2)$ to $(u_2/x)u'$. This term is in $|T|$ and it reduces to $y \mapsto v$, hence for every term w in $|U_1|$, $(w/y)v$ is in $|U_2|$. Thus the term $(u_1\ u_2)$ is in $|T|$.

Proposition 3.6.2 *The rewrite system $\mathcal{T}'$ is strongly terminating.*

Proof. We follow the lines of the proof of proposition 3.3.3 and reduce termination in $\mathcal{T}'$ to termination in the system formed with the first rule. We define a translation $\|\ \|$ of the terms and the propositions of type theory into terms of type theory. In each type T, we choose a variable z_T.

- $\|x\| = x$,
- $\|x \mapsto t\| = x \mapsto \|t\|$,
- $\|(t\ u)\| = (\|t\|\ \|u\|)$,

- $||\dot{\top}|| = ||\dot{\bot}|| = ((x \mapsto x)\ z_o)$,
  $||\dot{\neg}|| = x \mapsto x$,
  $||\dot{\wedge}|| = ||\dot{\vee}|| = ||\dot{\Rightarrow}|| = ((x \mapsto x)\ z_{o \to o \to o})$,
  $||\dot{\forall}_T|| = ||\dot{\exists}_T|| = x \mapsto (x\ z_T)$,
- $||\varepsilon(t)|| = ||t||$,
  $||\top|| = ||\bot|| = z_o$,
  $||\neg A|| = ||A||$,
  $||A \wedge B|| = ||A \vee B|| = ||A \Rightarrow B|| = (z_{o \to o \to o}\ ||A||\ ||B||)$,
  $||\forall x\ A|| = ||\exists x\ A|| = ||(z_T/x)A||$.

We check that if A rewrites in one step to B in $\mathcal{T}$, then $||A||$ rewrites in at least one step to $||B||$ in the system formed with the first rule. If $A_0, A_1, A_2, ...$ is a reduction sequence in $\mathcal{T}$, then the sequence $||A_0||, ||A_1||, ||A_2||, ...$ is a reduction sequence in the system formed with the first rule, thus it is finite.

Proposition 3.6.3 *The rewrite system $\mathcal{T}'$ is confluent.*

Remark. If we add the axiom of extensionality to both formulations of type theory we get equivalent theories, *i.e.* each language can be translated into the other preserving provability. When we do not take the extensionality axioms, there are subtle differences between these theories, we shall not discuss here.

Remark. Some authors use the notation $\lambda x\ t$ for $x \mapsto t$, hence the name λ-*calculus* for this language.

Chapter 4

Cut elimination in predicate logic

4.1 Uniform proofs

A natural deduction proof built without the excluded middle rule is said to be *constructive*. The choice of this name comes from the fact that, as we shall see, from a constructive proof in the empty theory of a proposition of the form $\exists x\ A$, it is possible to compute a term t and a proof of the proposition $(t/x)A$. Such a term t is called a *witness* of the proposition $\exists x\ A$. Thus, explicitly or implicitly, a constructive existence proof contains a witness.

Conversely, from a term t and a proof of $(t/x)A$, the rule $\exists$-intro permits to build a proof of the proposition $\exists x\ A$. A proof ended by an introduction rule is said to be *uniform*. Witnesses are explicit in uniform existence proofs. Thus, it is equivalent to have a term t and a proof of $(t/x)A$ or a uniform proof of the proposition $\exists x\ A$. To prove that from a constructive proof of a proposition of the form $\exists x\ A$ we can compute a witness, we shall prove that all proofs can be transformed into uniform ones. For instance, the non uniform proof of the proposition $\exists x\ (P(x) \Rightarrow P(x))$

$$\dfrac{\dfrac{\exists x\ (P(x) \Rightarrow P(x)) \vdash \exists x\ (P(x) \Rightarrow P(x))}{\vdash \exists x\ (P(x) \Rightarrow P(x)) \Rightarrow \exists x\ (P(x) \Rightarrow P(x))}\ \Rightarrow\text{-intro} \quad \dfrac{\dfrac{P(c) \vdash P(c)}{\vdash P(c) \Rightarrow P(c)}\ \Rightarrow\text{-intro}}{\vdash \exists x\ (P(x) \Rightarrow P(x))}\ \exists\text{-intro}}{\vdash \exists x\ (P(x) \Rightarrow P(x))}\ \Rightarrow\text{-elim}$$

will be transformed into

$$\dfrac{\dfrac{P(c) \vdash P(c)}{\vdash P(c) \Rightarrow P(c)}\ \Rightarrow\text{-intro}}{\vdash \exists x\ (P(x) \Rightarrow P(x))}\ \exists\text{-intro}$$

From the fact that all proofs can be transformed into uniform ones, we will deduce that

- if A is an atomic proposition then it has no proof,

- $\bot$ has no proof,
- if $\neg A$ has a proof then $\bot$ has a proof from the axiom A,
- if $A \wedge B$ has a proof then A has a proof and B has a proof,
- if $A \vee B$ has a proof then A has a proof or B has a proof,
- if $A \Rightarrow B$ has a proof then B has a proof from the axiom A,
- if $\forall x\ A$ has a proof then A has a proof,
- if $\exists x\ A$ has a proof then there is a term t such that $(t/x)A$ has a proof.

The results obtained for the case of $\top$, $\neg$, $\wedge$, $\Rightarrow$ and $\forall$ are trivial, they can simply be established with the elimination rules. The interesting results are thus for $\bot$, $\vee$ and $\exists$. The result in the case of the existential quantifier $\exists$ is the witness property. The result obtained in the case of the disjunction $\vee$ is called the *disjunction property.* The result obtained in the case of the contradiction $\bot$ is the consistency of the empty theory. Thus, like model constructions, proof transformation results permit to prove consistency and independence results.

Excercise 4.1.1 (Independence of the Excluded middle rule) *Consider a language formed with a proposition symbol P and a theory containing no axioms and no rewrite rules. Construct a model where the proposition P is not valid. Does this proposition have a proof ? Construct a model where the proposition $\neg P$ is not valid. Does this proposition have a proof ? Does the proposition $P \vee \neg P$ have a constructive proof ?*

Excercise 4.1.2 *Consider a language formed with a proposition symbol P and a theory containing no axioms and no rewrite rules. Construct a model where the proposition P is not valid. Does this proposition have a proof ? Construct a model where the proposition $\neg P$ is not valid. Does this proposition have a proof ? Does the proposition $P \vee \neg P$ have a proof (possibly using the excluded middle rule) ? Does natural deduction with the excluded middle have the disjunction property ?*

Excercise 4.1.3 *Consider a language formed with a proposition symbol P, a predicate symbol Q of one argument and two individual symbols* 0 *and* 1 *and a theory containing no axioms and no rewrite rules. Construct a model where the proposition*

$$(((Q(0) \Rightarrow Q(0)) \wedge P) \vee (Q(1) \Rightarrow Q(0) \wedge \neg P))$$

is not valid. Does this proposition have a proof ? Construct a model where the proposition

$$(((Q(0) \Rightarrow Q(1)) \wedge P) \vee (Q(1) \Rightarrow Q(1) \wedge \neg P))$$

is not valid. Does this proposition have a proof ? Does the proposition

$$\exists x\ (((Q(0) \Rightarrow Q(x)) \wedge P) \vee (Q(1) \Rightarrow Q(x) \wedge \neg P))$$

have a proof (possibly using the excluded middle) ? Does natural deduction with the excludes middle rule have the witness property ?

Remark. Some problems in mathematics have the form "Find an object x such that A". One way to solve such a problem is to prove constructively the proposition $\exists x\ A$, to transform this proof into a uniform one and to read the witness in the proof. For instance, finding the quotient of the division of 9 by 2 can be done in the following way: fist prove constructively the proposition

$$\exists q\ \exists r\ (9 = 2 \times q + r \wedge r < 2)$$

then transform this proof into a uniform one and read the witness in the proof. One advantage of proceeding this way, compared to other division algorithms, is that the result cannot be wrong. Indeed, a uniform proof of

$$\exists q\ \exists r\ (9 = 2 \times q + r \wedge r < 2)$$

not only contains the witness 4 but also a proof of the proposition

$$\exists r\ (9 = 2 \times 4 + r \wedge r < 2)$$

Of course, finding a proof of the proposition

$$\exists q\ \exists r\ (9 = 2 \times q + r \wedge r < 2)$$

may be tedious, but it is not if we prove once for all the proposition

$$\forall n\ \forall p\ (\neg(p = 0) \Rightarrow \exists q\ \exists r\ (n = p \times q + r \wedge r < p))$$

Notice that when we apply this theorem to 9 and 2 and to a proof of $\neg 2 = 0$ we get a proof of

$$\exists q\ \exists r\ (9 = 2 \times q + r \wedge r < 2)$$

that is not uniform. Thus, this proof needs to be transformed before the witness can be read. The quotient 4 is computed during this transformation. Thus cut elimination is the execution process of mathematics seen as a programming language.

4.2 Cuts and cut elimination

Definition 4.2.1 (Cut, cut free) *A cut is a proof ended with an elimination rule whose left premise is proved by an introduction rule on the same symbol. Here are the different cases*

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A \vdash \bot}}{\Gamma \vdash \neg A}\ \neg\text{-}intro \qquad \dfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash \bot}\ \neg\text{-}elim$$

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \qquad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-}intro}{\Gamma \vdash A}\ \wedge\text{-}elim$$

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A} \quad \cfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-}intro}{\Gamma \vdash B}\ \wedge\text{-}elim$$

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A \vee B}\ \vee\text{-}intro \quad \cfrac{\pi'}{\Gamma, A \vdash C} \quad \cfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\ \vee\text{-}elim$$

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash B}}{\Gamma \vdash A \vee B}\ \vee\text{-}intro \quad \cfrac{\pi'}{\Gamma, A \vdash C} \quad \cfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\ \vee\text{-}elim$$

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-}intro \quad \cfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash B}\ \Rightarrow\text{-}elim$$

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash \forall x\ A}\ \forall\text{-}intro}{\Gamma \vdash (t/x)A}\ \forall\text{-}elim$$

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma \vdash (t/x)A}}{\Gamma \vdash \exists x\ A}\ \exists\text{-}intro \quad \cfrac{\pi'}{\Gamma, A \vdash B}}{\Gamma \vdash B}\ \exists\text{-}elim$$

A proof contains a cut if one of its sub-trees is a cut. Otherwise it is cut free.

It is easy to check that cut free proofs in the empty theory are uniform.

Proposition 4.2.1 *In the empty theory, a cut free proof ends with an introduction rule.*

Proof. By induction over the height of the proof. The last rule cannot be an axiom rule, because the theory contains no axioms. If the last rule is an elimination, then the left premise of the elimination is proved with a cut free proof. Hence it ends by an introduction and the proof is a cut contradicting the fact that it is cut free.

Thus to prove that all proofs can be transformed into uniform ones we will prove that all proofs can be transformed into cut free ones. To do so, we define a process that eliminates cuts step by step. A cut of the form

$$\cfrac{\cfrac{\cfrac{\pi}{\Gamma, A \vdash \bot}}{\Gamma \vdash \neg A}\ \neg\text{-intro} \quad \cfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash \bot}\ \neg\text{-elim}$$

is replaced by the proof obtained this way: in the proof π we suppress the hypothesis A in all sequents, then each time the axiom rule is used with this

proposition, we replace it with the proof π'. A cut of the form

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \quad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}}{\Gamma \vdash A}\ \wedge\text{-elim}$$

is replaced by the proof π. A cut of the form

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A} \quad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro}}{\Gamma \vdash B}\ \wedge\text{-elim}$$

is replaced by the proof π'. A cut of the form

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A \vee B}\ \vee\text{-intro} \quad \dfrac{\pi'}{\Gamma, A \vdash C} \quad \dfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\ \vee\text{-elim}$$

is replaced by the proof obtained this way: in the proof π' we suppress the hypothesis A in all sequents, then each time the axiom rule is used with this proposition, we replace it by the proof π. A cut of the form

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash B}}{\Gamma \vdash A \vee B}\ \vee\text{-intro} \quad \dfrac{\pi'}{\Gamma, A \vdash C} \quad \dfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\ \vee\text{-elim}$$

is replaced by the proof obtained this way: in the proof π'' we suppress the hypothesis B in all sequents, then each time the axiom rule is used with this proposition, we replace it by the proof π. A cut of the form

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro} \quad \dfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash B}\ \Rightarrow\text{-elim}$$

is replaced by the proof obtained this way: in the proof π we suppress the hypothesis A in all sequents, then each time the axiom rule is used with this proposition, we replace it with the proof π'. A cut of the form

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash \forall x\ A}\ \forall\text{-intro}}{\Gamma \vdash (t/x)A}\ \forall\text{-elim}$$

is replaced by the proof π where the variable x is substituted by the term t everywhere. A cut of the form

$$\dfrac{\dfrac{\dfrac{\pi}{\Gamma \vdash (t/x)A}}{\Gamma \vdash \exists x\ A}\ \exists\text{-intro} \quad \dfrac{\pi'}{\Gamma, A \vdash B}}{\Gamma \vdash B}\ \exists\text{-elim}$$

is replaced by the proof obtained this way: in the proof π', we substitute the variable x by the term t everywhere, then we suppress the hypothesis $(t/x)A$ in all sequents and each time the axiom rule is used with this proposition, we replace it with the proof π.

Excercise 4.2.1 *Eliminate the cuts in the proof*

$$\dfrac{\dfrac{\exists x\ (P(x) \Rightarrow P(x)) \vdash \exists x\ (P(x) \Rightarrow P(x))}{\vdash \exists x\ (P(x) \Rightarrow P(x)) \Rightarrow \exists x\ (P(x) \Rightarrow P(x))}\Rightarrow\text{-}intro \quad \dfrac{\dfrac{P(c) \vdash P(c)}{\vdash P(c) \Rightarrow P(c)}\Rightarrow\text{-}intro}{\vdash \exists x\ (P(x) \Rightarrow P(x))}\exists\text{-}intro}{\vdash \exists x\ (P(x) \Rightarrow P(x))}\Rightarrow\text{-}elim$$

When a proof contains a cut, it is always simple to remove it, thus the cut elimination process is not difficult to define. But removing a cut may create new cuts, so the main question is that of the termination of this process.

4.3 Proofs as terms

The cut elimination process of the previous section is still cumbersome to express. This is due to the fact that we use a too cumbersome notation for natural deduction proof. The goal of this section is to introduce another notation for these proofs.

As we have seen, one of the key operations in this proof transformation process is the substitution of a variable by a term. Another key operation is the following: in a proof π of the sequent $\Gamma, A \vdash B$, remove the hypothesis A in all sequents and replace the axiom rules on this proposition by a proof π' of the sequent $\Gamma \vdash A$. To be able to express smoothly this operation, it is better to use a notation where proofs are expressed by terms containing special variables standing for proofs of the hypotheses. Thus to express a proof of a sequent $A_1, ..., A_n \vdash B$ we shall first introduce variables $\xi_1, ..., \xi_n$ standing for proofs of the propositions $A_1, ..., A_n$. If B is the proposition A_i and the sequent $A_1, ..., A_n \vdash A_i$ is proved with the axiom rule, we shall write this proof ξ_i.

Now a proof π of the sequent $\Gamma, A \vdash B$ is expressed by a term containing one variable for each proposition of Γ and a variable ξ for A and the proof obtained by removing the hypothesis A in all sequents of π and replacing the axiom rules on this proposition by a proof π' of the sequent $\Gamma \vdash A$ is simply obtained by substituting the proof π' for the variable ξ in π.

For each natural deduction rule, we introduce a function symbol. To express a proof such as

$$\dfrac{\dfrac{\pi}{\Gamma \vdash A} \quad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\wedge\text{-intro}$$

we express first the proofs π and π' as terms, then we apply the function symbol of two arguments associated to the rule $\wedge$-intro to π and π'.

In the case of the rule $\Rightarrow$-intro, we transform a proof π of the sequent $\Gamma, A \vdash B$ into one of the sequent $\Gamma \vdash A \Rightarrow B$ containing less hypotheses. The proof π

is expressed by a term containing a variable ξ standing for a proof of A. This variable must not appear in the proof of $\Gamma \vdash A \Rightarrow B$. Thus the function symbol associated to the rule $\Rightarrow$-intro must be a binder.

From now on, to simplify proofs, we shall drop the negation symbol $\neg$. Everything works for the proposition $\neg A$ as for the proposition $A \Rightarrow \bot$.

Definition 4.3.1 (Term notation for proofs) *We express proofs as terms in a language with two sorts: one for terms of the theory and the other for proof-terms. Terms of the theory will be written with Latin letters (t, u, ...) while proof-terms will be written with Greek letters (π, ...).*

- *The proof*

$$\frac{}{A_1, ..., A_n \vdash A_i}\ Axiom$$

  is expressed by the term ξ_i.

- *The proof*

$$\frac{}{\Gamma \vdash \top}\ \top\text{-}intro$$

  is expressed by the term I, where I is an individual symbol.

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash \bot}}{\Gamma \vdash A}\ \bot\text{-}elim$$

  is expressed by the term $\delta_\bot(\pi)$, where $\delta_\bot$ is a function symbol of one argument.

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash A} \quad \dfrac{\pi'}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\ \wedge\text{-}intro$$

  is expressed by the term $\langle \pi, \pi' \rangle$, where $\langle , \rangle$ is a function symbol of two arguments.

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A}\ \wedge\text{-}elim$$

  is expressed by the term $fst(\pi)$ and the proof

$$\frac{\dfrac{\pi}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A}\ \wedge\text{-}elim$$

  is expressed by the term $snd(\pi)$ where fst and snd are function symbols of one argument.

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash A \vee B}\vee\text{-}intro$$

is expressed by the term $i(\pi)$ *and the proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash B}}{\Gamma \vdash A \vee B}\vee\text{-}intro$$

is expressed by the term $j(\pi)$, *where* i *and* j *are function symbols of one argument.*

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash A \vee B} \quad \dfrac{\pi'}{\Gamma, A \vdash C} \quad \dfrac{\pi''}{\Gamma, B \vdash C}}{\Gamma \vdash C}\vee\text{-}elim$$

is expressed by the term $\delta(\pi, \xi\ \pi', \chi\ \pi'')$, *where* δ *is a function symbol of three arguments binding one variable in its second argument and one in its third.*

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\Rightarrow\text{-}intro$$

is expressed by the term $\xi \mapsto \pi$, *where* $\mapsto$ *is a function symbol of one argument binding one variable in its argument.*

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash A \Rightarrow B} \quad \dfrac{\pi'}{\Gamma \vdash A}}{\Gamma \vdash B}\Rightarrow\text{-}elim$$

is expressed by the term $\alpha(\pi, \pi')$, *where* α *is a function symbol of two arguments. This term is also simply written* $(\pi\ \pi')$.

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash A}}{\Gamma \vdash \forall x\ A}\forall\text{-}intro$$

is expressed by the term $x \mapsto \pi$, *where* $\mapsto$ *is a function symbol of one argument binding one variable in its argument.*

- *The proof*

$$\frac{\dfrac{\pi}{\Gamma \vdash \forall x\ A}}{\Gamma \vdash (t/x)A}\forall\text{-}elim$$

is expressed by the term $\alpha(\pi, t)$ *where* α *is a function symbol of two arguments. This term is also simply written* $(\pi\ t)$.

- *The proof*

$$\dfrac{\dfrac{\pi}{\Gamma \vdash (t/x)A}}{\Gamma \vdash \exists x\ A}\ \exists\text{-}intro$$

is expressed by the term $\langle t, \pi \rangle$ *where* $\langle , \rangle$ *is a function symbol of two arguments.*

- *The proof*

$$\dfrac{\dfrac{\pi}{\Gamma \vdash \exists x\ A} \quad \dfrac{\pi'}{\Gamma, A \vdash B}}{\Gamma \vdash B}\ \exists\text{-}elim$$

is expressed by the term $\delta_\exists(\pi, x\xi\ \pi')$ *where* $\delta_\exists$ *is a function symbol of two arguments binding two variables in its second argument.*

Excercise 4.3.1 *Write the term associated to the proof*

$$\dfrac{\dfrac{\exists x\ (P(x) \Rightarrow P(x)) \vdash \exists x\ (P(x) \Rightarrow P(x))}{\vdash \exists x\ (P(x) \Rightarrow P(x)) \Rightarrow \exists x\ (P(x) \Rightarrow P(x))}\ \Rightarrow\text{-}intro \quad \dfrac{\dfrac{P(c) \vdash P(c)}{\vdash P(c) \Rightarrow P(c)}\ \Rightarrow\text{-}intro}{\vdash \exists x\ (P(x) \Rightarrow P(x))}\ \exists\text{-}intro}{\vdash \exists x\ (P(x) \Rightarrow P(x))}\ \Rightarrow\text{-}elim$$

Remark.(An historical note on the choice of symbols) The choice of these symbols comes from a tradition due to Brouwer, Heyting and Kolmogorov, according to which

- there is only one proof of $\top$,
- there is no proof of $\bot$,
- a proof of $A \wedge B$ is an ordered pair formed with a proof of A and a proof of B,
- a proof of $A \vee B$ is a boolean value together with a proof of A or B according to the value of the boolean,
- a proof of $A \Rightarrow B$ is a function mapping proofs of A to proofs of B,
- a proof of $\forall x\ A$ is a function mapping any object t to a proof of $(t/x)A$,
- a proof of $\exists x\ A$ is an ordered pair formed with a term t and a proof of $(t/x)A$.

Remark. (Types of proofs) If π is a proof of B under the hypothesis A then $\xi \mapsto \pi$ is a proof of $A \Rightarrow B$. As all proofs have the same sort, the proof-term $\xi \mapsto \pi$ does not have a type, but if we wanted to give a type to it, it would get the type $A' \rightarrow B'$ where A' is the type of proofs of A and B' the type of proofs of B. Thus the type of a proof would be isomorphic to the proposition proved by the proof-term. This isomorphism is called Curry-de Bruijn-Howard isomorphism. In particular it can be proved that a type contains a closed term

in the language of definition 3.3.2 or 3.6.1 if and only if this type is isomorphic to proposition that has a constructive proof.

As proof-terms have no type, there are proof-terms that are proof of no proposition. For instance, if P is a proposition symbol and ξ a variable standing for a proof of P then the proof-term $(\xi\ \xi)$ does not corresponds to any proof. The natural deduction rules are now used to express which proof-terms is a a proof of which proposition. We use a notation $\xi_1 : A_1, ..., \xi_n : A_n \vdash \pi : B$ to express that π is a proof of the sequent $A_1, ..., A_n \vdash B$ where $\xi_1, ..., \xi_n$ are the names given to the variables of standing for proofs of the propositions $A_1, ..., A_n$. The rules are the following.

Definition 4.3.2 (Deduction rules with proofs)

$$\frac{}{\Gamma \vdash \xi : A}\ \textit{Axiom if}\ \xi : A \in \Gamma$$

$$\frac{}{\Gamma \vdash I : \bot}\ \top\textit{-intro}$$

$$\frac{\Gamma \vdash \pi : \bot}{\Gamma \vdash \delta_\bot(\pi) : A}\ \bot\textit{-elim}$$

$$\frac{\Gamma \vdash \pi : A \quad \Gamma \vdash \pi' : B}{\Gamma \vdash \langle \pi, \pi' \rangle : A \wedge B}\ \wedge\textit{-intro}$$

$$\frac{\Gamma \vdash \pi : A \wedge B}{\Gamma \vdash \mathit{fst}(\pi) : A}\ \wedge\textit{-elim}$$

$$\frac{\Gamma \vdash \pi : A \wedge B}{\Gamma \vdash \mathit{snd}(\pi) : B}\ \wedge\textit{-elim}$$

$$\frac{\Gamma \vdash \pi : A}{\Gamma \vdash i(\pi) : A \vee B}\ \vee\textit{-intro}$$

$$\frac{\Gamma \vdash \pi : B}{\Gamma \vdash j(\pi) : A \vee B}\ \vee\textit{-intro}$$

$$\frac{\Gamma \vdash \pi : A \vee B \quad \Gamma, \xi : A \vdash \pi' : C \quad \Gamma, \chi : B \vdash \pi'' : C}{\Gamma \vdash \delta(\pi, \xi\pi', \chi\pi'') : C}\ \vee\textit{-elim}$$

$$\frac{\Gamma, \xi : A \vdash \pi : B}{\Gamma \vdash \xi \mapsto \pi : A \Rightarrow B}\ \Rightarrow\textit{-intro}$$

$$\frac{\Gamma \vdash \pi : A \Rightarrow B \quad \Gamma \vdash \pi' : A}{\Gamma \vdash (\pi\ \pi') : B}\ \Rightarrow\textit{-elim}$$

$$\frac{\Gamma \vdash \pi : A}{\Gamma \vdash x \mapsto \pi : \forall x\ A}\ \forall\textit{-intro if}\ x \notin FV(\Gamma)$$

$$\frac{\Gamma \vdash \pi : \forall x\ A}{\Gamma \vdash (\pi\ t) : (t/x)A}\ \forall\textit{-elim}$$

$$\frac{\Gamma \vdash \pi : (t/x)A}{\Gamma \vdash \langle t, \pi \rangle : \exists x\ A}\ \exists\text{-}intro$$

$$\frac{\Gamma \vdash \pi : \exists x\ A \quad \Gamma, \xi : A \vdash \pi' : B}{\Gamma \vdash \delta_{\exists}(\pi, x\xi\pi') : B}\ \exists\text{-}elim\ \text{if } x \notin FV(\Gamma, B)$$

Proposition 4.3.1 *A sequent $A_1, \ldots, A_n \vdash B$ is derivable in natural deduction if and only if there exists a term π such that the judgment $\xi_1 : A_1, \ldots, \xi_n : A_n \vdash \pi : B$ is derivable in this system.*

The cut elimination rules can now be rephrased on the proof-terms

Definition 4.3.3 (Cut elimination rules)

$$fst(\langle \pi_1, \pi_2 \rangle) \longrightarrow \pi_1$$

$$snd(\langle \pi_1, \pi_2 \rangle) \longrightarrow \pi_2$$

$$\delta(i(\pi_1), \xi\pi_2, \chi\pi_3) \longrightarrow (\pi_1/\xi)\pi_2$$

$$\delta(j(\pi_1), \xi\pi_2, \chi\pi_3) \longrightarrow (\pi_1/\chi)\pi_3$$

$$((\xi \mapsto \pi_1)\ \pi_2) \longrightarrow (\pi_2/\xi)\pi_1$$

$$((x \mapsto \pi)\ t) \longrightarrow (t/x)\pi$$

$$\delta_{\exists}(\langle t, \pi_1 \rangle, \xi x \pi_2) \longrightarrow (t/x, \pi_1/\xi)\pi_2$$

Proposition 4.3.2 (Subject reduction) *If $\Gamma \vdash \pi : P$ and $\pi \longrightarrow \pi'$ then $\Gamma \vdash \pi' : P$.*

4.4 Cut elimination

We now want to prove that if a proof-term is a proof of some proposition then it is strongly terminating. Following the idea of Curry-de Bruijn-Howard isomorphism, this proof extends that of proposition 3.6.1.

Definition 4.4.1 (Reducible proof-terms) *Let A be a proposition. We define the set $|A|$ of reducible proof-terms of A by induction over the height of A.*

- *If A is an atomic proposition then a proof-term π is an element of $|A|$ if it is strongly terminating.*
- *A proof-term π is an element of $|\top|$ if it is strongly terminating.*
- *A proof-term π is an element of $|\bot|$ if it is strongly terminating.*
- *A proof-term π is an element of $|A \wedge B|$ if it is strongly terminating and when π reduces to a proof-term of the form $\langle \pi_1, \pi_2 \rangle$ then π_1 is an element of $|A|$ and π_2 is an element of $|B|$.*

- *A proof-term π is an element of $|A \vee B|$ if it is strongly terminating and when π reduces to a proof-term of the form $i(\pi_1)$ (resp. $j(\pi_2)$) then π_1 (resp. π_2) is an element of $|A|$ (resp. $|B|$).*

- *A proof-term π is element of $|A \Rightarrow B|$ if it is strongly terminating and when π reduces to a proof-term of the form $\xi \mapsto \pi_1$ then for every π' in $|A|$, $(\pi'/\xi)\pi_1$ is an element of $|B|$.*

- *A proof-term π is an element of $|\forall x\ A|$ if it is strongly terminating and when π reduces to a proof-term of the form $x \mapsto \pi_1$ then for every term t $(t/x)\pi_1$ is an element of $|(t/x)A|$ (which is equal to $|A|$).*

- *A proof-term π is an element of $|\exists x\ A|$ if it is strongly terminating and when π reduces to a proof-term of the form $\langle t, \pi_1 \rangle$ then π_1 is an element of $|(t/x)A|$ (which is equal to $|A|$).*

Lemma 4.4.1 *Elements of $|A|$ are strongly terminating.*

Proof. By definition.

Lemma 4.4.2 *If π is an element of $|A|$ and $\pi \longrightarrow \pi'$ then π' is an element of $|A|$.*

Proof. By definition.

Lemma 4.4.3 *All variables are members of $|A|$.*

Proof. By definition.

Lemma 4.4.4 *If π is an elimination and if for every π' such that $\pi \longrightarrow^1 \pi'$, $\pi' \in |A|$ then $\pi \in |A|$.*

Proof. We first prove that π is strongly terminating. Let $\pi = \pi_1, \pi_2, \ldots$ be a reduction sequence issued from π. If this sequence is empty it is finite. Otherwise we have $\pi \longrightarrow^1 \pi_2$ and hence π_2 is an element of $|A|$ thus it is strongly terminating and the reduction sequence is finite.

Then, we prove that if π reduces to an introduction then the sub-terms belong to the appropriate sets. Let $\pi = \pi_1, \pi_2, \ldots \pi_n$ be a reduction sequence issued from π and such that π_n is an introduction. This sequence cannot be empty because π is an elimination. Thus $\pi \longrightarrow^1 \pi_2 \longrightarrow \pi_n$. We have $\pi_2 \in |A|$ and thus if π_n is an introduction the sub-terms belong to the appropriate sets.

Proposition 4.4.5 (Gentzen-Prawitz theorem) *If $\Gamma \vdash \pi : A$ then the proof-term π is strongly terminating.*

Proof. By lemma 4.4.1, it is sufficient to prove that if $\Gamma \vdash \pi : A$ then the proof-term π is an element of $|A|$. More generally, we prove, by induction over the height of the proof-assignment tree, that if $\Gamma \vdash \pi : A$, θ is a substitution mapping the term variable to terms and σ is a substitution mapping some proof variables associated to a proposition B in Γ to an element of $|B|$, then $\sigma\theta\pi$ is an element of $|A|$.

- Axiom. If π is a variable ξ, we have $(\xi : A) \in \Gamma$. If ξ is in the domain of definition of σ, then $\sigma\theta\xi = \sigma\xi$ is an element of $|A|$, otherwise $\sigma\theta\xi = \sigma\xi = \xi$ is an element of $|A|$ by proposition 4.4.3.

- $\top$-intro. The proof-term π has the form I. We have $\sigma\theta\pi = I$. This proof-term is normal and thus it is strongly terminating. Hence, the proof-term $\sigma\theta I$ is in $|A|$.

- $\wedge$-intro. The proof-term π has the form $\langle\rho_1, \rho_2\rangle$ where ρ_1 is a proof of some proposition B and ρ_2 a proof of some proposition C. We have $\sigma\theta\pi = \langle\sigma\theta\rho_1, \sigma\theta\rho_2\rangle$. Consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-terms $\sigma\theta\rho_1$ and $\sigma\theta\rho_2$. By induction hypothesis these proof-terms are in $|B|$ and $|C|$. Thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $\langle\rho'_1, \rho'_2\rangle$ where ρ'_1 is a reduct of $\sigma\theta\rho_1$ and ρ'_2 one of $\sigma\theta\rho_2$. The proof-terms ρ'_1 and ρ'_2 are in $|B|$ and $|C|$ by proposition 4.4.2.

  Hence, the proof-term $\sigma\theta\langle\rho_1, \rho_2\rangle$ is in $|A|$.

- $\vee$-intro. The proof-term π has the form $i(\rho)$ (resp. $j(\rho)$) and ρ is a proof of some proposition B. We have $\sigma\theta\pi = i(\sigma\theta\rho)$ (resp. $j(\sigma\theta\rho)$). Consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-terms $\sigma\theta\rho$. By induction hypothesis this proof-term is an element of $|B|$. Thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $i(\rho')$ (resp. $j(\rho')$) where ρ' is a reduct of $\sigma\theta\rho$. The proof-term ρ' is an element of $|B|$ by proposition 4.4.2.

  Hence, the proof-term $\sigma\theta i(\rho)$ (respectively $\sigma\theta j(\rho)$) is an element of $|A|$.

- $\Rightarrow$-intro. The proof-term π has the form $\xi \mapsto \rho$ where ξ is a proof variable of some proposition B and ρ a proof of some proposition C. We have $\sigma\theta\pi = \xi \mapsto \sigma\theta\rho$, consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-term $\sigma\theta\rho$. By induction hypothesis, the proof-term $\sigma\theta\rho$ is an element of $|C|$, thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $\xi \mapsto \rho'$ where ρ' is a reduct of $\sigma\theta\rho$. Let τ be any proof of $|B|$, the proof-term $(\tau/\xi)\rho'$ can be obtained by reduction from $((\tau/\xi) \circ \sigma)\theta\rho$. By induction hypothesis, the proof-term

$((\tau/\xi) \circ \sigma)\theta\rho$ is an element of $|C|$. The proof term $(\tau/\xi)\rho'$ is an element of $|C|$, by proposition 4.4.2.

Hence, the proof-term $\sigma\theta(\xi \mapsto \rho)$ is an element of $|A|$.

- $\forall$-intro. The proof-term π has the form $x \mapsto \rho$ where ρ is a proof of some proposition B. We have $\sigma\theta\pi = x \mapsto \sigma\theta\rho$. Consider a reduction sequence issued from the proof-term $\sigma\theta\pi = x \mapsto \sigma\theta\rho$. This sequence can only reduce the proof-term $\sigma\theta\rho$. By induction hypothesis, the proof-term $\sigma\theta\rho$ is an element of $|B|$, thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $x \mapsto \rho'$ where ρ' is a reduct of $\sigma\theta\rho$. The proof-term $(t/x)\rho'$ is obtained by reducing the proof-term $((t/x)\sigma)((t/x) \circ \theta)\rho$. By induction hypothesis again, the proof-term $((t/x)\sigma)((t/x) \circ \theta)\rho$ is an element of $|B|$. The proof-term $(t/x)\rho'$ is an element of $|B|$, by proposition 4.4.2.

  Hence $\sigma\theta(x \mapsto \rho)$ is an element of $|A|$.

- $\exists$-intro. The proof-term π has the form $\langle t, \rho \rangle$, where ρ is a proof of some proposition B. We have $\sigma\theta\pi = \langle \theta t, \sigma\theta\rho \rangle$. Consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-term $\sigma\theta\rho$. By induction hypothesis this proof-term is in $|B|$. Thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $\langle \theta t, \rho' \rangle$ where ρ' is a reduct of $\sigma\theta\rho$. The proof-term ρ' is an element of $|B|$, by proposition 4.4.2.

  Hence, the proof-term $\sigma\theta\langle t, \rho \rangle$ is an element of $|A|$.

- $\bot$-elim. The proof-term π has the form $\delta_\bot(\rho)$ where ρ is a proof of $\bot$. We have $\sigma\theta\pi = \delta_\bot(\sigma\theta\rho)$. By induction hypothesis, the proof-term $\sigma\theta\rho$ is an element of $|\bot|$. Hence, it is strongly terminating. Let n be the maximum length of reduction sequences issued from this proof-term. We prove by induction on n that $\delta_\bot(\sigma\theta\rho)$ is in $|A|$. Since this proof-term is an elimination, by proposition 4.4.4, we only need to prove that every of its one step reducts is in $|A|$. The reduction can only take place in $\sigma\theta\rho$ and we apply the induction hypothesis.

  Hence, the proof-term $\sigma\theta\delta_\bot(\rho)$ is an element of $|A|$.

- $\wedge$-elim. We only detail the case of left elimination. The proof-term π has the form *fst*(ρ) where ρ is a proof of some proposition $A \wedge B$. We have $\sigma\theta\pi =$ *fst*$(\sigma\theta\rho)$. By induction hypothesis the proof-term $\sigma\theta\rho$ is in $|A \wedge B|$. Hence, it is strongly terminating. Let n be the maximum length of a reduction sequence issued from this proof-term. We prove by induction on n that *fst*$(\sigma\theta\rho)$ is in the set $|A|$. Since this proof-term is a elimination, by proposition 4.4.4, we only need to prove that every of its one step reducts is in $|B|$. If the reduction takes place in $\sigma\theta\rho$ then we apply the induction hypothesis. Otherwise $\sigma\theta\rho$ has the form $\langle \rho'_1, \rho'_2 \rangle$ and the reduct is ρ'_1. By the definition of $|A \wedge B|$ this proof-term is in $|A|$.

  Hence, the proof-term $\sigma\theta$*fst*(ρ) is an element of $|A|$.

- $\vee$-elim. The proof-term π has the form $\delta(\rho_1, \xi\rho_2\ \chi\rho_3)$ where ρ_1 is a proof of some proposition $B \vee C$ and ρ_2 and ρ_3 are proofs of A. We have $\sigma\theta\pi = \delta(\sigma\theta\rho_1, \xi\sigma\theta\rho_2, \chi\sigma\theta\rho_3)$. By induction hypothesis, the proof-term $\sigma\theta\rho_1$ is in the set $|B \vee C|$, and the proof-terms $\sigma\theta\rho_2$ and $\sigma\theta\rho_3$ are in the set $|A|$. Hence, these proof-terms are strongly terminating. Let n, n' and n'' be the maximum length of reduction sequences issued from these proof-terms. We prove by induction on $n + n' + n''$ that $\delta(\sigma\theta\rho_1, \sigma\theta\xi\rho_2, \chi\sigma\theta\rho_3)$ is in $|A|$. Since this proof-term is an elimination, by proposition 4.4.4, we only need to prove that every of its one step reducts is in $|A|$. If the reduction takes place in $\sigma\theta\rho_1$, $\sigma\theta\rho_2$ or $\sigma\theta\rho_3$ then we apply the induction hypothesis. Otherwise, if $\sigma\theta\rho_1$ has the form $i(\rho')$ (resp. $j(\rho')$) and the reduct is $((\rho'/\xi)\circ\sigma)\theta\rho_2$ (resp. $((\rho'/\chi)\circ\sigma)\theta\rho_3$). By the definition of $|B \vee C|$ the proof-term ρ' is in $|B|$ (resp. $|C|$). Hence by induction hypothesis $((\rho'/\xi) \circ \sigma)\theta\rho_2$ (resp. $((\rho'/\chi) \circ \sigma)\theta\rho_3$) is in $|A|$.

  Hence, the proof-term $\sigma\theta\delta(\rho_1, \xi\rho_2, \chi\rho_3)$ is an element of $|A|$.

- $\Rightarrow$-elim. The proof-term π has the form $(\rho_1\ \rho_2)$ and ρ_1 is a proof of some proposition $B \Rightarrow A$ and ρ_2 a proof of the proposition B. We have $\sigma\theta\pi = (\sigma\theta\rho_1\ \sigma\theta\rho_2)$. By induction hypothesis $\sigma\theta\rho_1$ and $\sigma\theta\rho_2$ are in the sets $|B \Rightarrow A|$ and $|B|$. Hence these proof-terms are strongly terminating. Let n be the maximum length of a reduction sequence issued from $\sigma\theta\rho_1$ and n' the maximum length of a reduction sequence issued from $\sigma\theta\rho_2$. We prove by induction on $n + n'$ that $(\sigma\theta\rho_1\ \sigma\theta\rho_2)$ is in the set $|A|$. Since this proof-term is an elimination, by proposition 4.4.4, we only need to prove that every of its one step reducts is in $|A|$. If the reduction takes place in $\sigma\theta\rho_1$ or in $\sigma\theta\rho_2$ then we apply the induction hypothesis. Otherwise $\sigma\theta\rho_1$ has the form $\xi \mapsto \rho'$ and the reduct is $(\sigma\theta\rho_2/\xi)\rho'$. By the definition of $|B \Rightarrow A|$ this proof-term is in $|A|$.

  Hence, the proof-term $\sigma\theta(\rho_1\ \rho_2)$ is an element of $|A|$.

- $\forall$-elim. The proof-term π has the form $(\rho\ t)$ where ρ is a proof of some proposition $\forall x\ B$ and $A = (t/x)B$. We have $\sigma\theta\pi = (\sigma\theta\rho\ \theta t)$. By induction hypothesis, the proof-term $\sigma\theta\rho$ is in $|\forall x\ B|$. Hence, it is strongly terminating. Let n be the maximum length of a reduction sequence issued from this proof-term. We prove by induction on n that $(\sigma\theta\rho\ \theta t)$ is in the set $|A|$. As this proof-term is an elimination, by proposition 4.4.4, we only need to prove that every of its one step reducts is in $|A|$. If the reduction takes place in $\sigma\theta\rho$ then we apply the induction hypothesis. Otherwise $\sigma\theta\rho$ has the form $x \mapsto \rho'$ and the reduct is $(\theta t/x)\rho'$. By the definition of $|\forall x\ B|$ this proof-term is in $|A|$.

  Hence, the proof-term $\sigma\theta(\rho\ t)$ is an element of $|A|$.

- $\exists$-elim. The proof-term π has the form $\delta_\exists(\rho_1, x\xi\rho_2)$ where ρ_1 is a proof of some proposition $\exists x\ B$ and ρ_2 is a proof of A. We have $\sigma\theta\pi = \delta_\exists(\sigma\theta\rho_1, x\xi\sigma\theta\rho_2)$. By induction hypothesis, the proof-term $\sigma\theta\rho_1$ is in the

set $|\exists x\ B|$ and the proof-term $\sigma\theta\rho_2$ is in the set $|A|$. Hence, these proof-terms are strongly terminating. Let n and n' be the maximum length of reduction sequences issued from these proof-terms. We prove by induction on $n + n'$ that $\delta_\exists(\sigma\theta\rho_1, x\xi\sigma\theta\rho_2)$ is in $|A|$. As this proof-term is an elimination, by proposition 4.4.4, we only need to prove that every of its one step reducts is in $|A|$. If the reduction takes place in $\sigma\theta\rho_1$ or $\sigma\theta\rho_2$ then we apply the induction hypothesis. Otherwise, $\sigma\theta\rho_1$ has the form $\langle t, \rho'\rangle$ and the reduct is $(\rho'/\xi)(t/x)\sigma\theta\rho_2 = ((\rho'/\xi) \circ (t/x)\sigma)((t/x) \circ \theta)\rho_2$. By the definition of $|\exists x\ B|$, the proof-term ρ' is in $|B|$. Thus, by induction hypothesis, the proof-term $((\rho'/\xi) \circ (t/x)\sigma)((t/x) \circ \theta)\rho_2$ is in $|A|$.

Hence, the proof-term $\sigma\theta\delta_\exists(\rho_1, \xi x\rho_2)$ is an element of $|A|$.

4.5 Harrop theories

We have seen that constructive cut free proofs in the empty theory are uniform, and we have deduced the disjunction property and the witness property for the empty theory. Of course these properties do not extend to all theories, but they extended to *Harrop theories.*

Definition 4.5.1 (Harrop theory) *The set of* Harrop propositions *is inductively defined as follows:*

- *atomic propositions, $\top$ and $\bot$ are Harrop propositions,*
- *$\neg A$ is a Harrop proposition,*
- *$A \wedge B$ is a Harrop proposition if A and B are Harrop propositions,*
- *$A \Rightarrow B$ is a Harrop proposition if B is a Harrop proposition,*
- *$\forall x\ A$ is a Harrop proposition if A is a Harrop proposition,*

A Harrop theory *is a theory whose axioms are all Harrop propositions.*

Proposition 4.5.1 *Let Γ be a Harrop theory. If $A \vee B$ has a constructive proof in Γ, then A or B has a proof in Γ and this proof is constructive. If $\exists x\ A$ has a constructive proof in Γ, then there is a term t such that $(t/x)A$ has a proof in Γ and this proof is constructive.*

Proof. By induction over the height of the proof.

If the proofs ends with an introduction, then the result is trivial.

The proof cannot end with an axiom because Γ contains only Harrop propositions and the conclusion is not a Harrop proposition.

We prove now that if the proof ends with an elimination then the theory Γ is contradictory and hence the result is trivial.

Let C_1 be the conclusion of the proof and C_2 be the left premise of this elimination, the proof of C_2 cannot end with an introduction because the proof

is cut free, hence it ends with an axiom rule or an elimination, if it ends with an elimination rule, then let C_3 be the left premise of this rule, ... Thus the rule ends with a sequence of elimination rules on propositions $C_1, ..., C_n$ and C_n is an axiom.

We prove that at least one of the propositions $C_1, ..., C_n$ is $\bot$. Assume this is not the case. Then the proposition C_n is a Harrop proposition because it is an element of Γ. Let us prove that the proposition C_{n-1} is also a Harrop proposition. The proposition C_{n-1} has been produced from C_n with an elimination rule. This elimination rule cannot be $\vee$-elim or $\exists$-elim because C_n is a Harrop proposition, it cannot be $\bot$-elim, because none of the propositions $C_1, ..., C_n$ is $\bot$. Hence it is either $\wedge$-elim, $\Rightarrow$-elim or $\forall$-elim, thus C_{n-1} is a Harrop proposition. We prove this way by induction that all the propositions $C_n, ..., C_1$ are Harrop propositions. Hence C_1 is a Harrop proposition which is contradictory.

Thus one of the propositions $C_1, ..., C_n$ is $\bot$, thus the theory Γ is contradictory, it proves all propositions and the result is trivial.

Excercise 4.5.1 *Show that proofs of propositions of the form $A \vee B$ and $\exists x\ A$ in consistent Harrop theories end with an introduction rule.*

Corollary 4.5.2 *Let P and Q be two proposition symbols, the proposition*

$$\neg\neg(P \vee Q) \Rightarrow (P \vee Q)$$

does not have a constructive proof in the empty theory.

Proof. Assume that the proposition $\neg\neg(P \vee Q) \Rightarrow (P \vee Q)$ has a proof. Let Γ be the Harrop theory formed with the axiom $\neg\neg(P \vee Q)$, the proposition $P \vee Q$ has a proof in Γ. Thus either the proposition P or the proposition Q has proof in Γ and it is easy to construct a model of Γ where P is not valid and a model of Γ where Q is not valid.

Corollary 4.5.3 *Let P be a proposition symbol, the proposition*

$$\neg\neg P \Rightarrow P$$

does not have a constructive proof in the empty theory.

Proof. If it had, so would the proposition. $\neg\neg(P \vee Q) \Rightarrow (P \vee Q)$.

Corollary 4.5.4 *Let P be a predicate symbol of one argument, the proposition*

$$(\neg\forall x\ P(x)) \Rightarrow \exists x\ \neg P(x)$$

does not have a constructive proof in the empty theory.

Proof. Assume that the proposition $(\neg\forall x\ P(x)) \Rightarrow \exists x\ \neg P(x)$ has a proof. Let Γ be the Harrop theory formed with the axiom $\neg\forall x\ P(x)$. Then the proposition $\exists x\ \neg P(x)$ has a proof in Γ. Thus there is a term t such that the proposition $\neg P(t)$ has a proof in Γ. Consider a model $\mathcal{M}$ with two elements and let $\hat{P}$ hold form the denotation of t but not for the other element. This model is a model of Γ but not of $\neg P(t)$. Thus, the proposition $\neg P(t)$ does not have a proof in Γ which is contradictory.

Chapter 5

Cut elimination in predicate logic modulo

We have seen that from the cut elimination theorem we could deduce the consistency, the disjunction property and the witness property for the empty theory. Of course, not many theorems can be proved in the empty theory. When we add axioms, cut free proofs need not be uniform anymore. For instance adding the axiom $\exists x\ P(x)$, allows a non uniform proof of the proposition $\exists x\ P(x)$. We have already seen that the disjunction property and the witness property extended to Harrop theories. We are now interested in other theories: theories modulo with no axioms, such as simple type theory and simple type theory with infinity.

5.1 Congruences defined by a system rewriting atomic propositions

Proposition 5.1.1 *Consider a congruence $\equiv$ defined by a confluent rewrite system rewriting terms to terms and atomic propositions to arbitrary propositions. If A and B are not atomic and $A \equiv B$ then A and B have the same root connector or quantifier.*

Proposition 5.1.2 *Consider a congruence $\equiv$ defined by a confluent rewrite system rewriting terms to terms and atomic propositions to arbitrary propositions. Consider the theory modulo formed with no axioms and the congruence $\equiv$. A cut free proof in this theory ends with an introduction rule.*

Proof. By induction over the height of the proof. The last rule cannot be an axiom rule, because there is no axiom. If the last rule is an elimination, then the left premise of the elimination is proved with a cut free proof. Hence it ends by an introduction. By proposition 5.1.2, this introduction concerns

the same connector or quantifier as the elimination rule and the proof is a cut contradicting the fact that it is cut free.

Thus, if cut elimination holds for such a theory, then consistency, the disjunction property and the witness property also.

5.2 Proof as terms

Proof-terms are defined as in predicate logic and the reduction rules are the same. But the proof assignments rules have to be modified to take the congruence into account.

Definition 5.2.1 (Deduction rules with proofs)

$$\frac{}{\Gamma \vdash_{\equiv} \xi : B} \text{ Axiom if } \xi : A \in \Gamma \text{ and } A \equiv B$$

$$\frac{}{\Gamma \vdash_{\equiv} I : A} \top\text{-intro if } A \equiv \top$$

$$\frac{\Gamma \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} \delta_{\bot}(\pi) : A} \bot\text{-elim if } B \equiv \bot$$

$$\frac{\Gamma \vdash_{\equiv} \pi : A \quad \Gamma \vdash_{\equiv} \pi' : B}{\Gamma \vdash_{\equiv} \langle \pi, \pi' \rangle : C} \wedge\text{-intro if } C \equiv (A \wedge B)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : C}{\Gamma \vdash_{\equiv} \mathit{fst}(\pi) : A} \wedge\text{-elim if } C \equiv (A \wedge B)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : C}{\Gamma \vdash_{\equiv} \mathit{snd}(\pi) : B} \wedge\text{-elim if } C \equiv (A \wedge B)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : A}{\Gamma \vdash_{\equiv} i(\pi) : C} \vee\text{-intro if } C \equiv (A \vee B)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} j(\pi) : C} \vee\text{-intro if } C \equiv (A \vee B)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : D \quad \Gamma, \xi : A \vdash_{\equiv} \pi' : C \quad \Gamma, \chi : B \vdash_{\equiv} \pi'' : C}{\Gamma \vdash_{\equiv} \delta(\pi, \xi\pi', \chi\pi'') : C} \vee\text{-elim if } D \equiv (A \vee B)$$

$$\frac{\Gamma, \xi : A \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} \xi \mapsto \pi : C} \Rightarrow\text{-intro if } C \equiv (A \Rightarrow B)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : C \quad \Gamma \vdash_{\equiv} \pi' : A}{\Gamma \vdash_{\equiv} (\pi\ \pi') : B} \Rightarrow\text{-elim if } C \equiv (A \Rightarrow B)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : A}{\Gamma \vdash_{\equiv} x \mapsto \pi : B} \langle x, A \rangle\ \forall\text{-intro if } B \equiv (\forall x\ A) \text{ and } x \notin FV(\Gamma)$$

$$\frac{\Gamma \vdash_{\equiv} \pi : B}{\Gamma \vdash_{\equiv} (\pi\ t) : C} \langle x, A, t \rangle\ \forall\text{-elim if } B \equiv (\forall x\ A) \text{ and } C \equiv (t/x)A$$

$$\frac{\Gamma \vdash_{\equiv} \pi : C}{\Gamma \vdash_{\equiv} \langle t, \pi \rangle : B} \langle x, A, t \rangle \text{ } \exists\text{-}intro \text{ } if \text{ } B \equiv (\exists x \text{ } A) \text{ } and \text{ } C \equiv (t/x)A$$

$$\frac{\Gamma \vdash_{\equiv} \pi : C \quad \Gamma, \xi : A \vdash_{\equiv} \pi' : B}{\Gamma \vdash_{\equiv} \delta_{\exists}(\pi, x\xi\pi') : B} \langle x, A \rangle \text{ } \exists\text{-}elim \text{ } if \text{ } C \equiv (\exists x \text{ } A) \text{ } and \text{ } x \notin FV(\Gamma, B)$$

Proposition 5.2.1 *A sequent* $A_1, \ldots, A_n \vdash_{\equiv} B$ *is derivable in natural deduction modulo if and only if there exists a term* π *such that the judgment* $\xi_1 : A_1, \ldots, \xi_n : A_n \vdash_{\equiv} \pi : B$ *is derivable in this system.*

Proposition 5.2.2 (Subject reduction) *If* $\Gamma \vdash_{\equiv} \pi : P$ *and* $\pi \longrightarrow \pi'$ *then* $\Gamma \vdash \pi' : P$.

5.3 Counterexamples

Cut elimination fails for very simple rewrite systems.

Example 5.3.1 (Russell's counterexample) *We have seen that in naive set theory, if we call* A *the proposition* $\varepsilon(R \ R)$ *(or* $R \in R$*) we have*

$$A \longrightarrow \neg A$$

Modulo this rule, the proposition $\neg A$ *has the proof*

$$\xi \mapsto (\xi \ \xi)$$

and the proposition A *also thus the proposition* $\bot$ *has the proof*

$$((\xi \mapsto (\xi \ \xi)) \ (\xi \mapsto (\xi \ \xi)))$$

This proof only reduces to itself and thus it does not terminate. It is easy to check that more generally, there are no cut free proofs of $\bot$ *because there no uniform proofs of this proposition.*

Example 5.3.2 (Crabbé's counterexample) *Set theory is an example of a theory modulo that does not have the cut elimination property. We have seen that there are two propositions* A *and* B *in set theory such that*

$$A \longrightarrow B \wedge \neg A$$

Thus under the assumption $\chi : B$, *the proposition* $\neg A$ *has the proof*

$$\xi \mapsto (snd(\xi) \ \xi)$$

and the proposition A *has the proof*

$$\langle \chi, \xi \mapsto (snd(\xi) \ \xi) \rangle$$

thus the proposition $\bot$ *has the proof*

$$((\xi \mapsto (snd(\xi) \ \xi)) \ \langle \chi, \xi \mapsto (snd(\xi) \ \xi) \rangle)$$

and the proposition $\neg B$ has the proof

$$\chi \mapsto ((\xi \mapsto (snd(\xi)\ \xi))\ \langle \chi, \xi \mapsto (snd(\xi)\ \xi)\rangle)$$

It is easy to check that this proof does not terminate and more generally that the proposition $\neg B$ has no cut free proof.

Example 5.3.3 (A terminating counterexample) *Cut elimination may be lost even with a confluent and terminating rewrite system. The example is a refined version of Russell's counterexample. Instead of taking the non terminating rule $R \in R \longrightarrow \neg R \in R$, we take the terminating rule*

$$R \in R \longrightarrow \forall y\ (y \simeq R \Rightarrow \neg y \in R)$$

where $y \simeq z$ stands for $\forall x\ (y \in x \Rightarrow z \in x)$. Modulo this rule, the proposition $\neg R \in R$ has the proof

$$\pi = \xi \mapsto (\xi\ R\ (x \mapsto (\chi \mapsto \chi))\ \xi)$$

and the proposition $R \in R$ has the proof

$$\pi' = y \mapsto (\xi \mapsto (\chi \mapsto (\pi\ (\xi\ R\ \chi))))$$

The proposition $\perp$ has the proof

$$(\pi\ \pi')$$

This proof only reduces to itself and thus it does not terminate. It is easy to check that more generally, there are no cut free proofs of $\perp$ because there no uniform proofs of this proposition.

5.4 Reducibility candidates

Let us try to characterize some congruences for which cut elimination holds.

We wish to use a cut elimination proof similar to that of predicate logic. The main problem is that we cannot take the set of all strongly terminating proof-terms for the set of reducible proof-terms of an atomic proposition. For instance if P, Q and R are three proposition symbol and we have the rule

$$P \longrightarrow Q \Rightarrow R$$

then a proof of P is also a proof of $Q \Rightarrow R$ and thus, to belong to $|P|$, besides being strongly terminating, a proof-term must be such that whenever it reduces to an introduction $\xi \mapsto \pi'$ for all proof π'' of $|Q|$, the proof $(\pi''/\xi)\pi'$ belongs to $|R|$. In this case we can take the set of all strongly terminating proofs for $|Q|$ and $|R|$ and the set $|Q \Rightarrow R|$ for $|P|$ and a proof similar to that of predicate logic permits to establish cut elimination modulo this rule.

However, generalizing this method may be difficult when we have non terminating rules or rules introducing quantifiers. For instance consider the proposition symbols P and Q and the rule

$$Q \longrightarrow P \wedge Q$$

defining $|Q|$ as $|P \wedge Q|$ would be circular, as to know $|P \wedge Q|$ we need to know $|P|$ and $|Q|$. In the same way, consider a predicate symbol P of one argument, an individual symbol c and the rule

$$P(c) \longrightarrow \forall x\ P(x)$$

Defining $|P(c)|$ as the set $|\forall x\ P(x)|$ would be circular as to know $|\forall x\ P(x)|$ we need to know $|P(t)|$ for all terms t, including c.

Thus we shall prove in a first step that cut elimination holds provided we know how to assign a set of proofs $|A|$ to each atomic proposition A in such a way that the sets of reducible proofs - defined relatively to these sets - of two equivalent propositions are identical. In a second step we shall give examples where such sets can be constructed including the two examples above and simple type theory.

Not any set of proof-terms is a good candidate for $|A|$. Indeed, we have seen that to let the cut elimination proof go through we needed the sets of reducible proofs to verify the properties of propositions 4.4.1, 4.4.2, 4.4.3 and 4.4.4 that are used in the cut elimination proof. Thus, at least, the sets of reducible proofs of atomic propositions must verify these properties. This leads to the following definition.

Definition 5.4.1 (Girard's reducibility candidate) *A set R of proof-terms is a* reducibility candidate *if*

- *if $\pi \in R$, then π is strongly terminating,*
- *if $\pi \in R$ and $\pi \longrightarrow \pi'$ then $\pi' \in R$,*
- *all variables belong to R,*
- *if π is an elimination and if for every π' such that $\pi \longrightarrow^1 \pi'$, $\pi' \in R$ then $\pi \in R$.*

Let $\mathcal{C}$ be the set of all reducibility candidates.

Assigning a reducibility candidate to each atomic proposition A, is equivalent to assign to each predicate symbol P of n arguments a function $\hat{P}$ that maps n-uples of terms to reducibility candidates. Then, we define the set $|P(t_1, ..., t_n)|$ as $\hat{P}(t_1, ..., t_n)$. Thus we want to prove that if we know how to assign such a function to each predicate symbol, in such a way that the sets of reducible proofs defined relatively to these functions are such that two equivalent propositions have the same set of reducible proofs, then cut elimination holds modulo this congruence.

This can be generalized: to have cut elimination it is sufficient to assign, to each predicate symbol P of n arguments, a function $\hat{P}$ that maps n-uples of elements of an arbitrary set M to reducibility candidates and to associate to each term t an element $|t|$ of M. Then we define $|P(t_1, ..., t_n)|$ as $\hat{P}(|t_1|, ..., |t_n|)$. If the sets of reducible proofs defined relatively to these functions are such that two equivalent propositions have the same set of reducible proofs, then cut elimination holds modulo this congruence.

There are many similarities between this definition and the definition of a model. In particular the fact that if $A \equiv B$ then $|A| = |B|$ can be read as the validity of the congruence in this structure. The only difference with the notion of model is that the functions $\hat{P}$ do not map n-uples of elements of M to truth values 0 or 1, but to reducibility candidates. Hence such structures are many-valued models where truth values are reducibility candidates. We shall call them *pre-models*. As we want to apply this result to many-sorted theories also, we directly give the definition for many-sorted predicate logic modulo.

5.5 Pre-model

Definition 5.5.1 (Pre-model) *Let $\mathcal{L}$ be a many sorted first-order language. A* pre-model *for $\mathcal{L}$ is given by:*

- *for every sort T, a set M_T,*
- *for every function symbol f of rank $\langle T_1, \ldots, T_n, U \rangle$, a function $\hat{f}$ from $M_{T_1} \times \ldots \times M_{T_n}$ to M_U,*
- *for every predicate symbol P of rank $\langle T_1, \ldots, T_n \rangle$, a function $\hat{P}$ from $M_{T_1} \times \ldots \times M_{T_n}$ to $\mathcal{C}$.*

Definition 5.5.2 *Let t be a term and ϕ an assignment mapping all the free variables of t of sort T to elements of M_T. We define the object $|t|_\phi$ by induction over the height of t.*

- $|x|_\phi = \phi(x)$,
- $|f(t_1, \ldots, t_n)|_\phi = \hat{f}(|t_1|_\phi, \ldots, |t_n|_\phi)$.

Definition 5.5.3 *Let A be a proposition and ϕ an assignment mapping all the free variables of A of sort T to elements of M_T. We define the set $|A|_\phi$ of proof-terms by induction over the height of A.*

- *A proof-term π is an element of $|P(t_1, \ldots, t_n)|_\phi$ if it is in $\hat{P}(|t_1|_\phi, \ldots, |t_n|_\phi)$.*
- *A proof-term π is an element of $|\top|_\phi$ if π is strongly terminating.*
- *A proof-term π is an element of $|\bot|_\phi$ if π is strongly terminating.*

- *A proof-term π is an element of $|A \wedge B|_\phi$ if π is strongly terminating and when π reduces to a proof-term of the form $\langle \pi_1, \pi_2 \rangle$ then π_1 and π_2 are elements of $|A|_\phi$ and $|B|_\phi$.*

- *A proof-term π is an element of $|A \vee B|_\phi$ if π is strongly terminating and when π reduces to a proof-term of the form $i(\pi_1)$ (resp. $j(\pi_2)$) then π_1 (resp. π_2) is an element of $|A|_\phi$ (resp. $|B|_\phi$).*

- *A proof-term π is element of $|A \Rightarrow B|_\phi$ if it is strongly terminating and when π reduces to a proof-term of the form $\xi \mapsto \pi_1$ then for every π' in $|A|_\phi$, $(\pi'/\xi)\pi_1$ is an element of $|B|_\phi$.*

- *A proof-term π is an element of $|\forall x\ A|_\phi$ if it is strongly terminating and when π reduces to a proof-term of the form $x \mapsto \pi_1$ then for every term t of sort T (where T is the sort of x) and every element E of M_T, $(t/x)\pi_1$ is an element of $|A|_{\phi+\langle x,E\rangle}$.*

- *A proof-term π is an element of $|\exists x\ A|_\phi$ if π is strongly terminating and whenever π reduces to a proof-term of the form $\langle t, \pi_1 \rangle$ there exists an element E of M_T (where T is the sort of x) such that π_1 is an element of $|A|_{\phi+\langle x,E\rangle}$.*

Definition 5.5.4 *A pre-model is a pre-model of a congruence $\equiv$ if, whenever $A \equiv B$, then for every assignment ϕ, $|A|_\phi = |B|_\phi$.*

Proposition 5.5.1 *For every proposition A and assignment ϕ, $|A|_\phi$ is a reducibility candidate*

Proof. By induction over the height of A.

If A is an atomic proposition, $|A|_\phi$ is a reducibility candidate by definition.

If A is a composed proposition, then, by definition, $|A|_\phi$ contains only terminating proof-terms. It is routine to prove closure by reduction. It is also routine to check that all variables are members of $|A|_\phi$.

Now, we assume that π is a an elimination and that for every π' such that $\pi \longrightarrow^1 \pi'$, $\pi' \in |A|_\phi$. We want to prove that π is in $|A|_\phi$. Following the definition of $|A|_\phi$, we first prove that π is strongly terminating and then that if it reduces to an introduction, the sub-proofs belong to the appropriate sets.

We first prove that π is strongly terminating. Let $\pi = \pi_1, \pi_2, \ldots$ be a reduction sequence issued from π. If this sequence is empty it is finite. Otherwise we have $\pi \longrightarrow^1 \pi_2$ and hence π_2 is an element of $|A|_\phi$ thus it is strongly terminating and the reduction sequence is finite.

Then we prove that if π reduces to an introduction then the sub-proofs belong to the appropriate sets. Let $\pi = \pi_1, \pi_2, \ldots \pi_n$ be a reduction sequence issued from π and such that π_n is an introduction. This sequence cannot be empty because π is an elimination and hence not an introduction. Thus $\pi \longrightarrow^1 \pi_2 \longrightarrow \pi_n$. We have $\pi_2 \in |A|_\phi$ and thus if π_n is an introduction the sub-proofs belong to the appropriate sets.

Proposition 5.5.2 (Substitution) *Given any proposition A, term t and variable x we have*

$$|(t/x)A|_\phi = |A|_{\phi+\langle x,|t|_\phi\rangle}$$

Proof. By induction on the height of A.

We can now prove the main theorem of this chapter: if a system has a pre-model then proof-terms modulo this system terminate.

Proposition 5.5.3 *Let $\equiv$ be a congruence and $\mathcal{M}$ be a pre-model of $\equiv$. If $\Gamma \vdash_\equiv \pi : A$ then the proof-term π is strongly terminating.*

Proof. As $|A|_\emptyset$ is a reducibility candidate, it is sufficient to prove that if $\Gamma \vdash \pi : A$ then the proof-term π is an element of $|A|_\emptyset$. More generally, we prove, by induction over the height of the proof-assignment tree, that if $\Gamma \vdash \pi : A$,

- θ is a substitution mapping term variables to terms,
- ϕ is an assignment mapping variables to elements of the model,
- σ is a substitution mapping some proof variables associated to proposition B in Γ to an element of $|B|_\phi$,

then $\sigma\theta\pi$ is an element of $|A|_\phi$.

- Axiom. If π is a variable ξ, we have $(\xi : B) \in \Gamma$ with $B \equiv A$. If ξ is in the domain of definition of σ, then $\sigma\theta\xi = \sigma\xi$ is an element of $|B|_\phi = |A|_\phi$, otherwise $\sigma\theta\xi = \sigma\xi = \xi$ is an element of $|A|_\phi$ because $|A|_\phi$ is a candidate.
- $\top$-intro. The proof-term π has the form I. We have $\sigma\theta\pi = I$. This proof-term is normal, hence it is strongly terminating. Hence, the proof-term $\sigma\theta I$ is in $|A|_\phi$.
- $\wedge$-intro. The proof-term π has the form $\langle\rho_1,\rho_2\rangle$ where ρ_1 is a proof of some proposition B and ρ_2 a proof of some proposition C. We have $\sigma\theta\pi = \langle\sigma\theta\rho_1,\sigma\theta\rho_2\rangle$. Consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-terms $\sigma\theta\rho_1$ and $\sigma\theta\rho_2$. By induction hypothesis these proof-terms are in $|B|_\phi$ and $|C|_\phi$. Thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $\langle\rho'_1,\rho'_2\rangle$ where ρ'_1 is a reduct of $\sigma\theta\rho_1$ and ρ'_2 one of $\sigma\theta\rho_2$. The proof-terms ρ'_1 and ρ'_2 are in $|B|_\phi$ and $|C|_\phi$ because these sets are candidates.

  Hence, the proof-term $\sigma\theta\langle\rho_1,\rho_2\rangle$ is in $|A|_\phi$.
- $\vee$-intro. The proof-term π has the form $i(\rho)$ (resp. $j(\rho)$) and ρ is a proof of some proposition B. We have $\sigma\theta\pi = i(\sigma\theta\rho)$ (resp. $j(\sigma\theta\rho)$). Consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-terms $\sigma\theta\rho$. By induction hypothesis this proof-term is an element of $|B|_\phi$. Thus the reduction sequence is finite.

Furthermore, all reducts of $\sigma\theta\pi$ have the form $i(\rho')$ (resp. $j(\rho')$) where ρ' is a reduct of $\sigma\theta\rho$. The proof-term ρ' is an element of $|B|_\phi$ because this set is a candidate.

Hence, the proof-term $\sigma\theta i(\rho)$ (respectively $\sigma\theta j(\rho)$) is an element of $|A|_\phi$.

- $\Rightarrow$-intro. The proof-term π has the form $\xi \mapsto \rho$ where ξ is a proof variable of some proposition B and ρ a proof of some proposition C. We have $\sigma\theta\pi = \xi \mapsto \sigma\theta\rho$, consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-term $\sigma\theta\rho$. By induction hypothesis, the proof-term $\sigma\theta\rho$ is an element of $|C|_\phi$, thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $\xi \mapsto \rho'$ where ρ' is a reduct of $\sigma\theta\rho$. Let τ be any proof of $|B|_\phi$, the proof-term $(\tau/\xi)\rho'$ can be obtained by reduction from $((\tau/\xi) \circ \sigma)\theta\rho$. By induction hypothesis, the proof-term $((\tau/\xi) \circ \sigma)\theta\rho$ is an element of $|C|_\phi$. The proof-term $(\tau/\xi)\rho'$ is an element of $|C|_\phi$ because this set is a candidate.

  Hence, the proof-term $\sigma\theta\xi \mapsto \rho$ is an element of $|A|_\phi$.

- $\forall$-intro. The proof-term π has the form $x \mapsto \rho$ where ρ is a proof of some proposition B. We have $\sigma\theta\pi = x \mapsto \sigma\theta\rho$.

  Consider a reduction sequence issued from the proof-term $\sigma\theta\pi = x \mapsto \sigma\theta\rho$. This sequence can only reduce the proof-term $\sigma\theta\rho$. Let E be an element of M_T (where T is the sort of x). By induction hypothesis, the proof-term $\sigma\theta\rho$ is an element of $|B|_{\phi+\langle x,E\rangle}$, thus the reduction sequence is finite.

  Furthermore, all reducts of $\sigma\theta\pi$ have the form $x \mapsto \rho'$ where ρ' is a reduct of $\sigma\theta\rho$. The proof-term $(t/x)\rho'$ is obtained by reducing the proof-term $((t/x)\sigma)((t/x) \circ \theta)\rho$. By induction hypothesis again, the proof-term $((t/x)\sigma)((t/x) \circ \theta)\rho$ is an element of $|B|_{\phi+\langle x,E\rangle}$. The proof-term $(t/x)\rho'$ is an element of $|B|_{\phi+\langle x,E\rangle}$, because this set is a candidate.

  Hence $\sigma\theta(x \mapsto \rho)$ is an element of $|A|_\phi$.

- $\exists$-intro. The proof-term π has the form $\langle t, \rho\rangle$, $A \equiv \exists x\ B$ and ρ is a proof of $(t/x)B$. We have $\sigma\theta\pi = \langle \theta t, \sigma\theta\rho\rangle$. Consider a reduction sequence issued from this proof-term. This sequence can only reduce the proof-term $\sigma\theta\rho$. By induction hypothesis this proof-term is in $|(t/x)B|_\phi$. Thus the reduction sequence is finite.

  Furthermore, let $E = |t|_\phi$. Any reduct of $\sigma\theta\pi$ has the form $\langle \theta t, \rho'\rangle$ where ρ' is a reduct of $\sigma\theta\rho$. The proof-term ρ' is an element of $|(t/x)B|_\phi$, *i.e.* of $|B|_{\phi+\langle x,E\rangle}$, because $|B|_{\phi+\langle x,E\rangle}$ is a candidate.

  Hence, the proof-term $\sigma\theta\langle t, \rho\rangle$ is an element of $|A|_\phi$.

- $\bot$-elim. The proof-term π has the form $\delta_\bot(\rho)$ where ρ is a proof of $\bot$. We have $\sigma\theta\pi = \delta_\bot(\sigma\theta\rho)$. By induction hypothesis, the proof-term $\sigma\theta\rho$ is an element of $|\bot|_\phi$. Hence, it is strongly terminating. Let n be the maximum length of reduction sequences issued from this proof-term. We

prove by induction on n that $\delta_\bot(\sigma\theta\rho)$ is in $|A|_\phi$. Since this proof-term is an elimination, we only need to prove that every of its one step reducts is in $|A|_\phi$. The reduction can only take place in $\sigma\theta\rho$ and we apply the induction hypothesis.

Hence, the proof-term $\sigma\theta\delta_\bot(\rho)$ is an element of $|A|_\phi$.

- $\wedge$-elim. We only detail the case of left elimination. The proof-term π has the form $\mathit{fst}(\rho)$ where ρ is a proof of some proposition $A \wedge B$. We have $\sigma\theta\pi = \mathit{fst}(\sigma\theta\rho)$. By induction hypothesis the proof-term $\sigma\theta\rho$ is in $|A \wedge B|_\phi$. Hence, it is strongly terminating. Let n be the maximum length of a reduction sequence issued from this proof-term. We prove by induction on n that $\mathit{fst}(\sigma\theta\rho)$ is in the set $|A|_\phi$. Since this proof-term is a elimination we only need to prove that every of its one step reducts is in $|B|_\phi$. If the reduction takes place in $\sigma\theta\rho$ then we apply the induction hypothesis. Otherwise $\sigma\theta\rho$ has the form $\langle\rho_1', \rho_2'\rangle$ and the reduct is ρ_1'. By the definition of $|A \wedge B|_\phi$ this proof-term is in $|A|_\phi$.

  Hence, the proof-term $\sigma\theta \mathit{fst}(\rho)$ is an element of $|A|_\phi$.

- $\vee$-elim. The proof-term π has the form $\delta(\rho_1, \xi\rho_2\ \chi\rho_3)$ where ρ_1 is a proof of some proposition $B \vee C$ and ρ_2 and ρ_3 are proofs of A. We have $\sigma\theta\pi = \delta(\sigma\theta\rho_1, \xi\sigma\theta\rho_2, \chi\sigma\theta\rho_3)$. By induction hypothesis, the proof-term $\sigma\theta\rho_1$ is in the set $|B \vee C|_\phi$, and the proof-terms $\sigma\theta\rho_2$ and $\sigma\theta\rho_3$ are in the set $|A|_\phi$. Hence, these proof-terms are strongly terminating. Let n, n' and n'' be the maximum length of reduction sequences issued from these proof-terms. We prove by induction on $n + n' + n''$ that $\delta(\sigma\theta\rho_1, \sigma\theta\xi\rho_2, \chi\sigma\theta\rho_3)$ is in $|A|_\phi$. Since this proof-term is an elimination we only need to prove that every of its one step reducts is in $|A|_\phi$. If the reduction takes place in $\sigma\theta\rho_1$, $\sigma\theta\rho_2$ or $\sigma\theta\rho_3$ then we apply the induction hypothesis. Otherwise, if $\sigma\theta\rho_1$ has the form $i(\rho')$ (resp. $j(\rho')$) and the reduct is $(\rho'/\xi)\sigma\theta\rho_2$ (resp. $(\rho'/\chi)\sigma\theta\rho_3$). By the definition of $|B{\vee}C|_\phi$ the proof-term ρ' is in $|B|_\phi$ (resp. $|C|_\phi$). Hence by induction hypothesis $((\rho'/\xi)\circ\sigma)\theta\rho_2$ (resp. $((\rho'/\chi)\circ\sigma)\theta\rho_3$) is in $|A|_\phi$.

  Hence, the proof-term $\sigma\theta\delta(\rho_1, \xi\rho_2, \chi\rho_3)$ is an element of $|A|_\phi$.

- $\Rightarrow$-elim. The proof-term π has the form $(\rho_1\ \rho_2)$ and ρ_1 is a proof of some proposition $B \Rightarrow A$ and ρ_2 a proof of the proposition B. We have $\sigma\theta\pi = (\sigma\theta\rho_1\ \sigma\theta\rho_2)$. By induction hypothesis $\sigma\theta\rho_1$ and $\sigma\theta\rho_2$ are in the sets $|B \Rightarrow A|_\phi$ and $|B|_\phi$. Hence these proof-terms are strongly terminating. Let n be the maximum length of a reduction sequence issued from $\sigma\theta\rho_1$ and n' the maximum length of a reduction sequence issued from $\sigma\theta\rho_2$. We prove by induction on $n + n'$ that $(\sigma\theta\rho_1\ \sigma\theta\rho_2)$ is in the set $|A|_\phi$. Since this proof-term is an elimination we only need to prove that every of its one step reducts is in $|A|_\phi$. If the reduction takes place in $\sigma\theta\rho_1$ or in $\sigma\theta\rho_2$ then we apply the induction hypothesis. Otherwise $\sigma\theta\rho_1$ has the form $\xi \mapsto \rho'$ and the reduct is $(\sigma\theta\rho_2/\xi)\rho'$. By the definition of $|B \Rightarrow A|_\phi$ this proof-term is in $|A|_\phi$.

Hence, the proof-term $\sigma\theta(\rho_1\ \rho_2)$ is an element of $|A|_\phi$.

- $\forall$-elim. The proof-term π has the form $(\rho\ t)$ where ρ is a proof of some proposition $\forall x\ B$ and $A = (t/x)B$. We have $\sigma\theta\pi = (\sigma\theta\rho\ \theta t)$. By induction hypothesis, the proof-term $\sigma\theta\rho$ is in $|\forall x\ B|_\phi$. Hence, it is strongly terminating. Let n be the maximum length of a reduction sequence issued from this proof-term. We prove by induction on n that $(\sigma\theta\rho\ \theta t)$ is in the set $|A|_\phi$. As this proof-term is an elimination, we only need to prove that every of its one step reducts is in $|A|_\phi$. If the reduction takes place in $\sigma\theta\rho$ then we apply the induction hypothesis. Otherwise $\sigma\theta\rho$ has the form $x \mapsto \rho'$ and the reduct is $\mathrm{n}(\theta t/x)\rho'$. By the definition of $|\forall x\ B|_\phi$ this proof-term is in $|B|_{\phi+\langle x,E\rangle}$ for all E. Thus, it is in is in $|B|_{\phi+\langle x,|t|_\phi\rangle} = |(t/x)B|_\phi = |A|_\phi$.

  Hence, the proof-term $\sigma\theta(\rho\ t)$ is an element of $|A|_\phi$.

- $\exists$-elim. The proof-term π has the form $\delta_\exists(\rho_1, x\xi\rho_2)$ where ρ_1 is a proof of some proposition $\exists x\ B$ and ρ_2 is a proof of A. We have $\sigma\theta\pi = \delta_\exists(\sigma\theta\rho_1, x\xi\sigma\theta\rho_2)$. By induction hypothesis, the proof-term $\sigma\theta\rho_1$ is in the set $|\exists x\ B|_\phi$ and the proof-term $\sigma\theta\rho_2$ is in the set $|A|_\phi$. Hence, these proof-terms are strongly terminating. Let n and n' be the maximum length of reduction sequences issued from these proof-terms. We prove by induction on $n + n'$ that $\delta_\exists(\sigma\theta\rho_1, x\xi\sigma\theta\rho_2)$ is in $|A|_\phi$. As this proof-term is an elimination, we only need to prove that every of its one step reducts is in $|A|_\phi$. If the reduction takes place in $\sigma\theta\rho_1$ or $\sigma\theta\rho_2$ then we apply the induction hypothesis. Otherwise, $\sigma\theta\rho_1$ has the form $\langle t, \rho'\rangle$ and the reduct is $(\rho'/\xi)(t/x)\sigma\theta\rho_2 = ((\rho'/\xi) \circ (t/x)\sigma)((t/x) \circ \theta)\rho_2$. By the definition of $|\exists x\ B|_\phi$, there exists an element E of such that the proof-term ρ' is in $|B|_{\phi+\langle x,E\rangle}$. Thus, by induction hypothesis, the proof-term $((\rho'/\xi) \circ (t/x)\sigma)((t/x) \circ \theta)\rho_2$ is in $|A|_{\phi+\langle x,E\rangle}$, *i.e.* in $|A|_\phi$.

  Hence, the proof-term $\sigma\theta\delta_\exists(\rho_1, \xi x\rho_2)$ is an element of $|A|_\phi$.

5.6 Pre-model construction

5.6.1 The term case

Proposition 5.6.1 *If a congruence is defined by a rewrite system or a set of equalities on terms, but not on propositions, then it has a pre-model and hence proof reduction terminates modulo this congruence.*

Proof. We associate the set of strongly terminating proofs for all atomic propositions.

Corollary 5.6.2 *All equational theories are consistent, have the disjunction property and the witness property.*

5.6.2 Quantifier free rewrite systems

Definition 5.6.1 (Quantifier free) *A rewrite system is* quantifier free *if no quantifier appears on the right hand side of any of its rules.*

Proposition 5.6.3 *A quantifier free, confluent, and terminating rewrite systems has a pre-model, hence proof reduction terminates modulo such a rewrite system.*

Proof. By induction over proposition height, we associate a set of proof-terms to each each normal closed quantifier free proposition.

$$\begin{array}{rcl}
\Psi(A) & = & \{\pi \mid \pi \text{ st. ter.}\} \quad \text{if } A \text{ is atomic} \\
\Psi(\top) & = & \{\pi \mid \pi \text{ st. ter.}\} \\
\Psi(\bot) & = & \{\pi \mid \pi \text{ st. ter.}\} \\
\Psi(A \wedge B) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \langle \pi_1, \pi_2 \rangle \Rightarrow \pi_1 \in \Psi(A) \wedge \pi_2 \in \Psi(B)\} \\
\Psi(A \vee B) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow i(\pi_1) \Rightarrow \pi_1 \in \Psi(A) \wedge \pi \longrightarrow i(\pi_2) \Rightarrow \pi_2 \in \Psi(B)\} \\
\Psi(A \Rightarrow B) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \xi \mapsto \pi_1 \Rightarrow \forall \pi' \in \Psi(A)\ (\pi'/\xi)\pi_1 \in \Psi(B)\}
\end{array}$$

We define a pre-model as follows. Let M_T be the set of normal closed terms of sort T.

$$\begin{array}{rcl}
\hat{f}(t_1, \ldots, t_n) & = & f(t_1, \ldots, t_n) \downarrow \\
\hat{P}(t_1, \ldots, t_n) & = & \Psi((P(t_1, \ldots, t_n)) \downarrow).
\end{array}$$

where $A \downarrow$ (resp. $t \downarrow$) is the normal form of the proposition A (resp. term t).

We prove, by an easy induction, that $|A|_\phi = |B|_\phi$ when $A \equiv B$.

5.6.3 Positive rewrite systems

For some rewrite systems, pre-models can be built by a fixed point construction.

Definition 5.6.2 *A rewrite system is* positive *if it rewrites atomic propositions to propositions containing only positive occurrences of atomic propositions.*

Definition 5.6.3 *A pre-model is* syntactical *if*

- $M_T = \mathcal{T}_T / \equiv$ *where* $\mathcal{T}_T$ *is the set of closed terms of sort* T,
- *if* f *is a function symbol,* $\hat{f}$ *is the function that maps the classes* $e_1, ..., e_n$ *to the class of the term* $f(t_1, \ldots, t_n)$ *where* $t_1, ..., t_n$ *are elements of* $e_1, ..., e_n$ *(since the relation* $\equiv$ *is a congruence, this does not depend of the choice of representatives).*

A syntactical pre-model is defined solely by the interpretation of predicate variables.

Definition 5.6.4 *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two syntactical pre-models. We write $\hat{P}_1$ for the denotation of P in $\mathcal{M}_1$ and $\hat{P}_2$ for the denotation of P in $\mathcal{M}_2$*

We say that $\mathcal{M}_1 \leq \mathcal{M}_2$ if and only if for any predicate symbol P and closed terms $t_1, \ldots, t_n$ we have

$$\hat{P}_1(t_1, \ldots, t_n) \subseteq \hat{P}_2(t_1, \ldots, t_n)$$

The set of syntactical pre-models is a complete lattice for the order $\leq$.

Proposition 5.6.4 *Let $\mathcal{R}$ be a confluent and terminating rewrite system. If the system $\mathcal{R}$ is positive then it has a pre-model, hence proof reduction terminates modulo $\mathcal{R}$.*

Proof. Let $\mathcal{F}$ be the function mapping syntactical pre-models to syntactical pre-models defined by

$$\mathcal{F}(\mathcal{M})(P)(t_1, \ldots, t_n) = |P(t_1, \ldots, t_n) \downarrow|_{\mathcal{M},\emptyset}.$$

As the system $\mathcal{R}$ is positive the function $\mathcal{F}$ is monotone. Hence, as the set of syntactical pre-models is a complete lattice, it has a fixed point. This fixed point is a pre-model of the rewrite system.

Proposition 5.6.5 *Let $\mathcal{R}$ be a rewrite system such that any atomic proposition has at most one one-step reduct. If the system $\mathcal{R}$ is positive then it has a pre-model, hence proof reduction terminates modulo $\mathcal{R}$.*

Proof. Let $\mathcal{F}$ be the function mapping syntactical pre-models to syntactical pre-models defined by

$$\mathcal{F}(\mathcal{M})(P)(t_1, \ldots, t_n) = |P(t_1, \ldots, t_n) + |_{\mathcal{M},\emptyset}$$

where $A+$ is the unique one-step reduct of A if it exists and A otherwise. Again, since the system $\mathcal{R}$ is positive the function $\mathcal{F}$ is monotone and again, since the set of syntactical pre-models is a complete lattice, it has a fixed point. This fixed point is a pre-model of the rewrite system.

5.6.4 Type theory and type theory with infinity

Proposition 5.6.6 (Girard's theorem) *Simple type theory has a pre-model, hence proof reduction terminate in simple type theory.*

Proof. We construct a pre-model as follows. The essential point is that we anticipate the fact that objects of sort o actually represent propositions, by interpreting them as reducibility candidates.

$$\begin{aligned} M_\iota &= \{0\} \\ M_o &= \mathcal{C} \\ M_{T \to U} &= M_U^{M_T} \end{aligned}$$

$$\begin{array}{rcl}
\hat{S}_{T,U,V} & = & a \mapsto (b \mapsto (c \mapsto a(c)(b(c)))) \\
\hat{K}_{T,U} & = & a \mapsto (b \mapsto a) \\
\hat{\alpha}(a,b) & = & a(b) \\
\hat{\varepsilon}(a) & = & a \\
\hat{\top} & = & \{\pi \mid \pi \text{ st. ter.}\} \\
\hat{\bot} & = & \{\pi \mid \pi \text{ st. ter.}\} \\
\hat{\wedge}(a,b) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \langle \pi_1, \pi_2 \rangle \Rightarrow \pi_1 \in a \wedge \pi_2 \in b\} \\
\hat{\vee}(a,b) & = & \{\pi \mid \pi \text{ st. ter.} \wedge (\pi \longrightarrow i(\pi_1) \Rightarrow \pi_1 \in a) \wedge (\pi \longrightarrow i(\pi_2) \Rightarrow \pi_2 \in b)\} \\
\hat{\Rightarrow}(a,b) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \xi \mapsto \pi_1 \Rightarrow \forall \pi' \in a\ (\pi'/\xi)\pi_1 \in b\} \\
\hat{\forall}_T(a) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow x \mapsto \pi_1 \Rightarrow \forall t \text{ of type } T\ \forall E \in M_T\ (t/x)\pi_1 \in a(E)\} \\
\hat{\exists}_T(a) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \langle t, \pi_2 \rangle \Rightarrow \exists E \in M_T\ \pi_2 \in a(E)\}
\end{array}$$

It is easy to check that $|A|_\phi = |B|_\phi$ when $A \equiv B$.

Proposition 5.6.7 *Simple type theory with infinity has a pre-model, hence proof reduction terminates in simple type theory with infinity.*

Proof.

$$\begin{array}{rcl}
M_\iota & = & \mathbb{N} \\
M_o & = & \mathcal{C} \\
M_{T\to U} & = & M_U^{M_T} \\
\hat{0} & = & 0, \\
\hat{Su} & = & n \mapsto n+1, \\
\hat{Pred} & = & n \mapsto \text{if } n = 0 \text{ then } 0 \text{ else } n-1, \\
\hat{Null} & = & n \mapsto \{\pi \mid \pi \text{ st. ter.}\}, \\
\hat{S}_{T,U,V} & = & a \mapsto (b \mapsto (c \mapsto a(c)(b(c)))) \\
\hat{K}_{T,U} & = & a \mapsto (b \mapsto a) \\
\hat{\alpha}(a,b) & = & a(b) \\
\hat{\varepsilon}(a) & = & a \\
\hat{\top} & = & \{\pi \mid \pi \text{ st. ter.}\} \\
\hat{\bot} & = & \{\pi \mid \pi \text{ st. ter.}\} \\
\hat{\wedge}(a,b) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \langle \pi_1, \pi_2 \rangle \Rightarrow \pi_1 \in a \wedge \pi_2 \in b\} \\
\hat{\vee}(a,b) & = & \{\pi \mid \pi \text{ st. ter.} \wedge (\pi \longrightarrow i(\pi_1) \Rightarrow \pi_1 \in a) \wedge (\pi \longrightarrow i(\pi_2) \Rightarrow \pi_2 \in b)\} \\
\hat{\Rightarrow}(a,b) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \xi \mapsto \pi_1 \Rightarrow \forall \pi' \in a\ (\pi'/\xi)\pi_1 \in b\} \\
\hat{\forall}_T(a) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow x \mapsto \pi_1 \Rightarrow \forall t \text{ of type } T\ \forall E \in M_T\ (t/x)\pi_1 \in a(E)\} \\
\hat{\exists}_T(a) & = & \{\pi \mid \pi \text{ st. ter.} \wedge \pi \longrightarrow \langle t, \pi_2 \rangle \Rightarrow \exists E \in M_T\ \pi_2 \in a(E)\}
\end{array}$$

It is easy to check that $|A|_\phi = |B|_\phi$ when $A \equiv B$.

Remark. In the pre-model above $\hat{\top}$ and $\hat{\bot}$ are interpreted by the same reducibility candidate (while in a model they are interpreted by a different truth value) hence the interpretation of $Null$ is simply the constant function equal to this

candidate. Thus it is not necessary to interpret the type ι as $\mathbb{N}$ and we could also take $M_\iota = \{0\}$.