

# Types, Propositions and Problems

## an introduction to type theoretical ideas

Bengt Nordström

Computing Science, Chalmers and University of Göteborg

Types Summer School, Hisingen, 15 August 2005

## Classical logic, truth tables

### Conjunction

$A$	$B$	$A \& B$
$T$	$T$	$T$
$T$	$F$	$F$
$F$	$T$	$F$
$F$	$F$	$F$

### Disjunction

$A$	$B$	$A \vee B$
$T$	$T$	$T$
$T$	$F$	$T$
$F$	$T$	$T$
$F$	$F$	$F$

### Implication

$A$	$B$	$A \supset B$
$T$	$T$	$T$
$T$	$F$	$F$
$F$	$T$	$T$
$F$	$F$	$T$

This assumes that a proposition is either true or false!

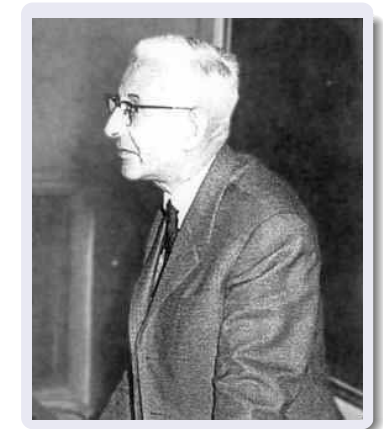
## Brouwer

Brouwer rejected the idea that the meaning of a mathematical proposition is its truth value. Mathematical propositions do not exist independently of us. We cannot say that a proposition is true without having a proof of it.



## Heyting

Heyting was a student of Brouwer. He gave the following explanation of the logical constants.

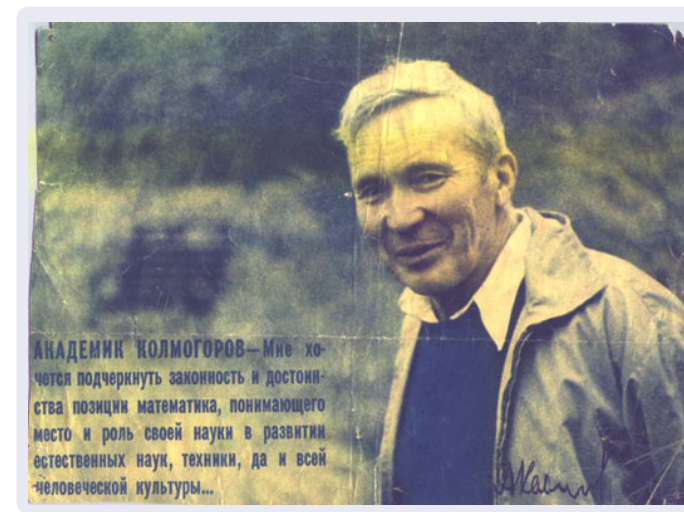


## Heyting's explanation of the logical constants (1930)

A proof of:	consists of:
$A \& B$	a proof of $A$ and a proof of $B$
$A \vee B$	a proof of $A$ or a proof of $B$
$A \supset B$	a method which takes any proof of $A$ to a proof of $B$
$\neg A$	a method which takes any proof of $A$ to a proof of absurdity
$\perp$	has no proof
$\exists x \in A. B$	an element $a$ in $A$ and a proof of $B[x := a]$
$\forall x \in A. B$	a method, which takes any element $x$ in $A$ to a proof of $B[x := a]$

## Kolmogorov

Independently of Heyting, Kolmogorov interpreted propositions as problems.



## Kolmogorov understood the logical constants as problems (1932)

The problem:	is solved if we can:
$A \& B$	solve $A$ and solve $B$
$A \vee B$	solve $A$ or solve $B$
$A \supset B$	reduce the solution of $B$ to the solution of $A$
$\neg A$	show that there is no solution of $A$
$\perp$	has no solution

## Heyting's and Kolmogorov's explanation

A proof (solution) of:	consists of:
$A \& B$	a proof (solution) of $A$ and a proof (solution) of $B$
$A \vee B$	a proof (solution) of $A$ or a proof (solution) of $B$
$A \supset B$	a method which takes any proof (solution) of $A$ to a proof (solution) of $B$
$\neg A$	a method which takes any proof (solution) of $A$ to a proof (solution) of absurdity
$\perp$	has no proof (solution)
$\exists x \in A. B$	an element $a$ in $A$ and a proof (solution) of $B[x := a]$
$\forall x \in A. B$	a method, which takes any element $x$ in $A$ to a proof (solution) of $B[x := a]$

### Question:

Is this correct? Could not a proof (solution) of  $A \& B$  be obtained by induction, for instance?

## Direct and indirect proofs

When we say that we have a proof of a proposition, then we mean that we have a method which when computed yields a direct proof of it.

Compare this with mathematics and programming: When we say that  $2 + 4$  and  $\text{fst}(\langle 45^2, -9 \rangle)$  are natural numbers, then we mean that they can be *computed* to a natural number.

### Terminology:

proofs:	objects:
direct vs. indirect proof	value vs. expression
canonical vs. non-canonical proof	canonical vs. non-canonical element
introduction vs. elimination proof	

## What is a proposition (problem)?

To summarize Heyting's and Kolmogorov's explanations:

### What does it mean to understand a proposition?

I understand a proposition when I understand what a direct proof of it is.

This looks very similar to:

### What does it mean to understand a set?

I understand a set when I understand what a canonical element of it is.

## Examples of indirect proofs

### And-elimination

$$\frac{A \& B}{A}$$

If we have a proof of  $A \& B$ , then we can compute it to a direct proof. This always consists of a proof of  $A$  and a proof of  $B$ . Hence we may always obtain a proof of  $A$  from a proof of  $A \& B$ .

### Mathematical induction

$$\frac{n \in \mathbb{N} \quad P(0) \quad (\forall n \in \mathbb{N}) P(n) \supset P(\text{succ}(n))}{P(i)}$$

### Propositions and sets

A proof (element) of:	consists of:
$A \& B$	a proof (solution) of $A$ and a proof (solution) of $B$
$A \times B$	an element in $A$ and an element in $B$
$A \vee B$	a proof (solution) of $A$ or a proof (solution) of $B$
$A + B$	an element in $A$ or an element in $B$
$A \supset B$	a method which takes any proof (solution) of $A$ to a proof (solution) of $B$
$A \rightarrow B$	a method which takes any element in $A$ to an element in $B$
$\perp$	has no proof (solution)
$\emptyset$	has no elements
$\exists x \in A. B$	an element $a$ in $A$ and a proof (solution) of $B[x := a]$
$\forall x \in A. B$	a method, which takes any element $x$ in $A$ to a proof (solution) of $B[x := a]$

This similarity leads to the

### Curry-Howard isomorphism

$$A \& B = A \times B$$

$$A \vee B = A + B$$

$$A \supset B = A \rightarrow B$$

$$\perp = \emptyset$$

$$\neg A = A \rightarrow \emptyset$$

## Curry's contribution

Curry noticed the formal similarity between the axioms of positive implicational logic:

$$A \supset B \supset A \\ (A \supset B \supset C) \supset (A \supset B) \supset C$$

and the type of the basic combinators:

$$K \in A \rightarrow B \rightarrow A \\ S \in (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow C$$

and modus ponens corresponds to the typing rule for application:

$$\frac{A \supset B \quad A}{A} \quad \frac{f \in A \rightarrow B \quad a \in A}{f \ a \in B}$$

## Proofs as Programs in a functional programming language

A direct proof of:	consists of:	As a type:
$A \vee B$	a proof of $A$ or a proof of $B$	<b>data</b> Or $A \ B = \text{Ori1 } A \   \ \text{Ori2 } B$ ;
$A \& B$	a proof of $A$ and a proof of $B$	<b>data</b> And $A \ B = \text{Andi } A \ B$ ;
$A \supset B$	a method taking a proof of $A$ to a proof of $B$	<b>data</b> Implies $A \ B = \text{Impi } A \ \rightarrow \ B$ ;
<i>Falsity</i>		<b>data</b> Falsity = ;

## Constructors are introduction rules

$$\frac{A}{A \vee B} \quad \text{Ori1} \in A \rightarrow A \vee B$$

$$\frac{B}{A \vee B} \quad \text{Ori2} \in B \rightarrow A \vee B$$

$$\frac{A \quad B}{A \& B} \quad \text{Andi} \in A \rightarrow B \rightarrow A \& B$$

$$\frac{[A]}{A \supset B} \quad \text{Impi} \in (A \rightarrow B) \rightarrow A \supset B$$

## Elimination rules can be defined

**orel**  $\in A \vee B \rightarrow (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C$

**orel (Ori1)**  $a$   $f$   $g = f a$

**orel (Ori2)**  $b$   $f$   $g = g b$

**andel**  $\in A \& B \rightarrow (A \rightarrow B \rightarrow C) \rightarrow C$

**andel (Andi)**  $a$   $b$   $f = f a b$

**implel**  $\in A \supset B \rightarrow A \rightarrow B$

**implel (Impli)**  $f$   $a = f a$

$$\frac{A \vee B \quad \begin{array}{c} [A] \\ C \end{array} \quad \begin{array}{c} [B] \\ C \end{array}}{C} \text{ orel}$$

$$\frac{A \& B \quad \begin{array}{c} [A, B] \\ C \end{array}}{C} \text{ andel}$$

$$\frac{A \supset B \quad A}{B} \text{ implel}$$

## Proof checking = Type checking

In this way we can prove propositional formulas in a typed functional programming language. The problem of proving for instance

$$(A \& B) \supset (B \& A)$$

is then the problem of finding a program in this type. The type checker will check if the proof is correct. In this case, we can use the following program:

```
Impli (λx.Andi (andel x λy.λz.z)
          (andel x λy.λz.y))
```

## What about the quantifiers?

### Propositions and sets

A proof (element) of:	consists of:
$\exists x \in A. B$	an element $a$ in $A$ and a proof (solution) of $B[x := a]$
$\Sigma x \in A. B$	an element $a$ in $A$ and an element in $B[x := a]$
$\forall x \in A. B$	a method, which takes any element $x$ in $A$ to a proof (solution) of $B[x := a]$
$\Pi x \in A. B$	a method, which takes any element $x$ in $A$ to an element in $B[x := a]$

## Overview of Martin Löf's type theory

- Type theory is a small typed functional language with one basic type and two type forming operation.
- It is a **framework** for defining logics.
- A new logic is introduced by definitions.

## What types are there?

- Set is a type
- $E(A)$  is a type, if  $A \in \text{Set}$ .
- $(x \in A) \rightarrow B$  is a type, if  $A$  is a type and  $B$  a family of types for  $x \in A$ .

## What programs are there?

Programs are formed from variables and constants using abstraction and application:

- Application

$$\frac{c \in (x \in A) \rightarrow B \quad a \in A}{c \ a \in B[x := a]}$$

- Abstraction

$$\frac{b \in B \ [x \in A]}{[x]b \in (x \in A) \rightarrow B}$$

- constants are either primitive or defined

## Constants

There are two kinds of constants:

**primitive:** (not defined) have a type but no definiens (RHS):

$$\text{identifier} \in \text{Type}$$

**defined:** have a type and a definiens:

$$\text{identifier} = \text{expr} \in \text{Type}$$

There are two kinds of defined constants:

- explicitly defined
- implicitly defined

## Primitive constants

- computes to themselves (i.e. are values).
- constructors in functional languages.
- introduction rules and formation rules in logic
- postulates

Examples:

$$\mathbb{N} \in \text{Set}$$

$$0 \in \mathbb{N}$$

$$s \in \mathbb{N} \rightarrow \mathbb{N}$$

$$\& \in \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$$

$$\&I \in (A \in \text{Set}) \rightarrow (B \in \text{Set}) \rightarrow A \rightarrow B \rightarrow A \& B$$

$$\Pi \in (A \in \text{Set}) \rightarrow (A \rightarrow \text{Set}) \rightarrow \text{Set}$$

$$\lambda \in (A \in \text{Set}) \rightarrow (B \in A \rightarrow \text{Set}) \rightarrow ((x \in A) \rightarrow B(x)) \rightarrow \Pi(A, B)$$

## Explicitly defined constants

- have a type and a definiens (RHS).
- the definiens is a welltyped expression
- abbreviation
- derived rule in logic.
- names for proofs and theorems in math.

Examples:

$$\begin{aligned}
 2 \in \mathbb{N} &= \text{succ}(\text{succ } 0) \\
 \forall(A \in \text{Set})(B \in A \rightarrow \text{Set}) \in \text{Set} &= \prod A B \\
 +(x \in \mathbb{N})(y \in \mathbb{N}) \in \mathbb{N} &= \text{natrec } [x] \mathbb{N} \times y [u, v](\text{succ } v) \\
 \supset (A \in \text{Set})(B \in \text{Set}) \in \text{Set} &= \prod A [x] B
 \end{aligned}$$

## Two approaches to the usage of implicit constants:

- The conservative approach: Use them only to define induction principles for sets (elimination rules). These are functions, which for an inductively defined set  $A$  produces a function in  $(x \in A) \rightarrow (C z)$  for a family of sets  $C \in A \rightarrow \text{Set}$ .
- The liberal approach: Use them when they are convenient.

## Implicitly defined constants

The definiens (RHS) may contain pattern matching and may contain occurrences of the constant itself. The correctness of the definition must in general be decided outside the system

- Recursively defined programs
- Elimination rules (the step from the definiendum to the definiens is the contraction rule).

Examples:

$$\begin{aligned}
 \mathbf{add}(x \in \mathbb{N})(y \in \mathbb{N}) \in \mathbb{N} \\
 \mathbf{add } 0 y = y \\
 \mathbf{add} (\text{succ } u) y = \text{succ} (\mathbf{add } u y) \\
 \mathbf{\&E}(A \in \text{Set})(B \in \text{Set})(C \in A \rightarrow B \rightarrow \text{Set}) \\
 (f \in (x \in A) \rightarrow (y \in B) \rightarrow C(\mathbf{\&I } x y)) \\
 (z \in A \mathbf{\&B}) \\
 \in C(z) \\
 \mathbf{\&E } A B C f (\mathbf{\&I } a b) = f a b
 \end{aligned}$$

## The editing process

The idea is to build expressions from incomplete expressions with holes (placeholders). Each editing step replaces a place holder with another incomplete expression

## Place holders

We use the notation

$$\square_1, \dots, \square_n$$

for place holders (holes).

Each place holder has an **expected type** and a **local context** (variables which may be used to fill in the hole).

## Refinement of an object

When we have constructed the type of the constant  $c$ , we can start to define it:

$$\begin{aligned} c &\in C \\ c &= \square_0 \end{aligned}$$

Here, *the expected type* of  $\square_0$  is  $C$ .

In general, we are in a situation like

$$c = \dots \square_1 \dots \square_2 \dots$$

where we know the expected type of the place holders.

## To construct an object

We start to give the name of the object to define, and the computer responds with

$$\begin{aligned} c &\in \square_1 \\ c &= \square_2 \end{aligned}$$

We must first give the type of  $c$  by refining  $\square_1$ .

We can either enter text from the keyboard, or do it stepwise, replace it by

- $(x \in \square_3) \rightarrow \square_4$
- Set, or
- $C \square_3 \dots \square_n$

## Refinement of an object: application

To refine a place holder

$$\square_0 \in A$$

with a constant  $c$  (or a variable) is to replace it by

$$c \square_1 \dots \square_n \in A$$

where  $\square_1 \in B_1, \dots, \square_n \in B_n$ .

The system computes the expected types of the new place holders and some constraints from the condition that the type of  $c \square_1 \dots \square_n$  must be equal to  $A$ .

We have reduced the problem  $A$  to the subproblems  $B_1, \dots, B_n$  using the rule  $c$ .

## Refinement of an object: abstraction

To refine a place holder

$$\square_0 \in A$$

with an abstraction is to replace it by

$$[x]\square_1 \in A$$

The system checks that  $A$  is a functional type  $(x \in B) \rightarrow C$  and the expected type of  $\square_1$  is  $C$  and the local context for it will contain the assumption  $x \in B$ .

We have reduced the problem  $(x \in B) \rightarrow C$  to the problem  $C$  by using the assumption  $x \in B$ .

## Hence: to prove is to build a proof object

- To apply a rule  $\mathbf{c}$  is to construct an application of the constant  $\mathbf{c}$ .
- To assume  $\mathbf{A}$  is to construct an abstraction of a variable of type  $\mathbf{A}$ .
- To refer to an assumption of  $\mathbf{A}$  is to use a variable of type  $\mathbf{A}$ .

## Summary: Type Theory

- Types:

$$T ::= \text{Set} \mid E!(e) \mid (x \in T) \rightarrow T'$$

- Programs:

$$e ::= e e' \mid [x]e \mid x \mid c$$

- Constants:

- Primitive (without a definition):

$$c \in T$$

- Explicitly defined:

$$c = e \in T$$

- Implicitly defined:

$$\begin{aligned} c \ p_1 \ \dots \ p_n &= e \\ &\vdots \\ c \ p'_1 \ \dots \ p'_n &= e' \end{aligned}$$