

# An Introduction to Martin-Lof's Constructive Type Theory and a computer implementation of it.

Bengt Nordström

`bengt@cs.chalmers.se`

ChungAng University, Seoul, Korea

on leave from Chalmers University, Göteborg, Sweden

Political Map of the World





# Background

Martin-Löf developed his type theory during 1970 – 1980 as a foundational language for mathematics. It is based on Constructive Mathematics and a proposition is the set of all its proofs. The following identifications can be made:

- $a \in A$
- $a$  is a *proof* of the *proposition*  $A$
- $a$  is an *object* in the *type*  $A$
- $a$  is a *program* with *specification*  $A$
- $a$  is a *solution* to the *problem*  $A$

# Proofs as Programs

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 = Ori1 A   Ori2 B;
$A \& B$	a proof of $A$ and a proof of $B$	<b>data</b> And A B = Andi A B;
$A \supset B$	a method taking a proof of $A$ to a proof of $B$	<b>data</b> Implies A B = Impi A -> B;
<i>Falsity</i>		<b>data</b> Falsity = ;

# Constructors are introduction rules

**Ori1**  $\in A \rightarrow A \vee B$

$$\frac{A}{A \vee B}$$

**Ori2**  $\in B \rightarrow A \vee B$

$$\frac{B}{A \vee B}$$

**Andi**  $\in A \rightarrow B \rightarrow A \& B$

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

**Impli**  $\in (A \rightarrow B) \rightarrow A \supset B$

$$\frac{[A] \quad B}{A \supset B}$$

# 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**  $b$ )  $f g = g b$

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

# 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**  $a b) f = f a b$

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

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



# 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**  $f) a = f a$

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

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

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

# 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 ( $\lambda x.$ Andi (andel  $x$   $\lambda y.$  $\lambda z.$  $z$ )  
        (andel  $x$   $\lambda y.$  $\lambda z.$  $y$ ))
```

# Two questions:

- Is it possible to extend this to more powerful logics, like predicate logic?
- We have that a proof of  $P$  is an object in the type  $P$ . But is it possible to identify the *process of proving*  $P$  with the *process of building* an object in the type  $P$ ?

The answer to these two questions is yes.

# Overview of 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
- If  $A \in \text{Set}$  then  $EI(A)$  is a type
- If  $A$  is a type and  $B$  a family of types for  $x \in A$  then  $(x \in A)B$  is a type.

# What programs are there?

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

- Application

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

- Abstraction

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

- constants are either primitive or defined

# Constants

There are two kinds of constants:

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

identifier  $\in$  Type

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

identifier = expr  $\in$  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:

$$\mathbb{N} \in \text{Set}$$
$$0 \in \mathbb{N}$$
$$s \in (\mathbb{N})\mathbb{N}$$
$$\& \in (\text{Set}, \text{Set})\text{Set}$$
$$\&I \in (A \in \text{Set}, B \in \text{Set}, A, B)A \& B$$
$$\Pi \in (A \in \text{Set}, (A)\text{Set}) \text{Set}$$
$$\lambda \in (A \in \text{Set}, B \in (A)\text{Set}, (x \in A)B(x))$$
$$\Pi(A, B)$$



# 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:

$$2 \equiv \text{succ}(\text{succ}(0)) \in \mathbf{N}$$

$$\forall \equiv \Pi \in (A \in \mathbf{Set}, (A)\mathbf{Set}) \mathbf{Set}$$

$$+ \equiv [x, y] \text{natrec}([x] \mathbf{N}, x, y, [u, v] \text{succ}(v)) \in (\mathbf{N}, \mathbf{N}) \mathbf{N}$$

$$\rightarrow \equiv [A, B] \Pi(A, [x] B) \in (A, B \in \mathbf{Set}) \mathbf{Set}$$

# 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:

$$\begin{aligned} &\&E \in (A \in \text{Set}, B \in \text{Set}, C \in (A, B)\text{Set}, \\ &\quad (x \in A, y \in B)C(\&I(x, y)), (z \in A \& B))C(z) \\ &\&E(A, B, C, f, \&I(a, b)) \equiv f(a, b) \end{aligned}$$

# The editing process

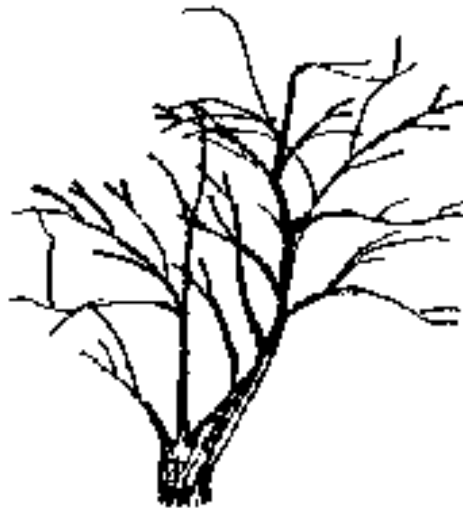
The idea is to build expressions from incomplete expressions with holes (placeholders). Each editing step replaces a place holder with another incomplete expression

# 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 (pruning a tree goes in the other direction).



Before  
Pruning



After First  
Years Pruning



After Second  
Years Pruning

# 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).

# To prove is to build

- To apply a rule **c** is to construct an application of the constant **c**.
- To assume **A** is to construct an abstraction of a variable of type **A**.
- To refer to an assumption of **A** is to use a variable of type **A**.

# To construct an object

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

$$c \in \square_1$$

$$c = \square_2$$

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

•  $(x \in \square_3)\square_4$  — a function type, or

• Set, or

•  $C(\square_3, \dots, \square_n)$

# Refinement of an object

When we have given the type, we can build the object:

$$c \in C$$

$$c = \square_0$$

where 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.



# 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  $n$  and the types of the new place holders as well as 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  $(x \in B)C$  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  $(x \in B)C$  to the problem  $C$  using the assumption  $x \in B$ .