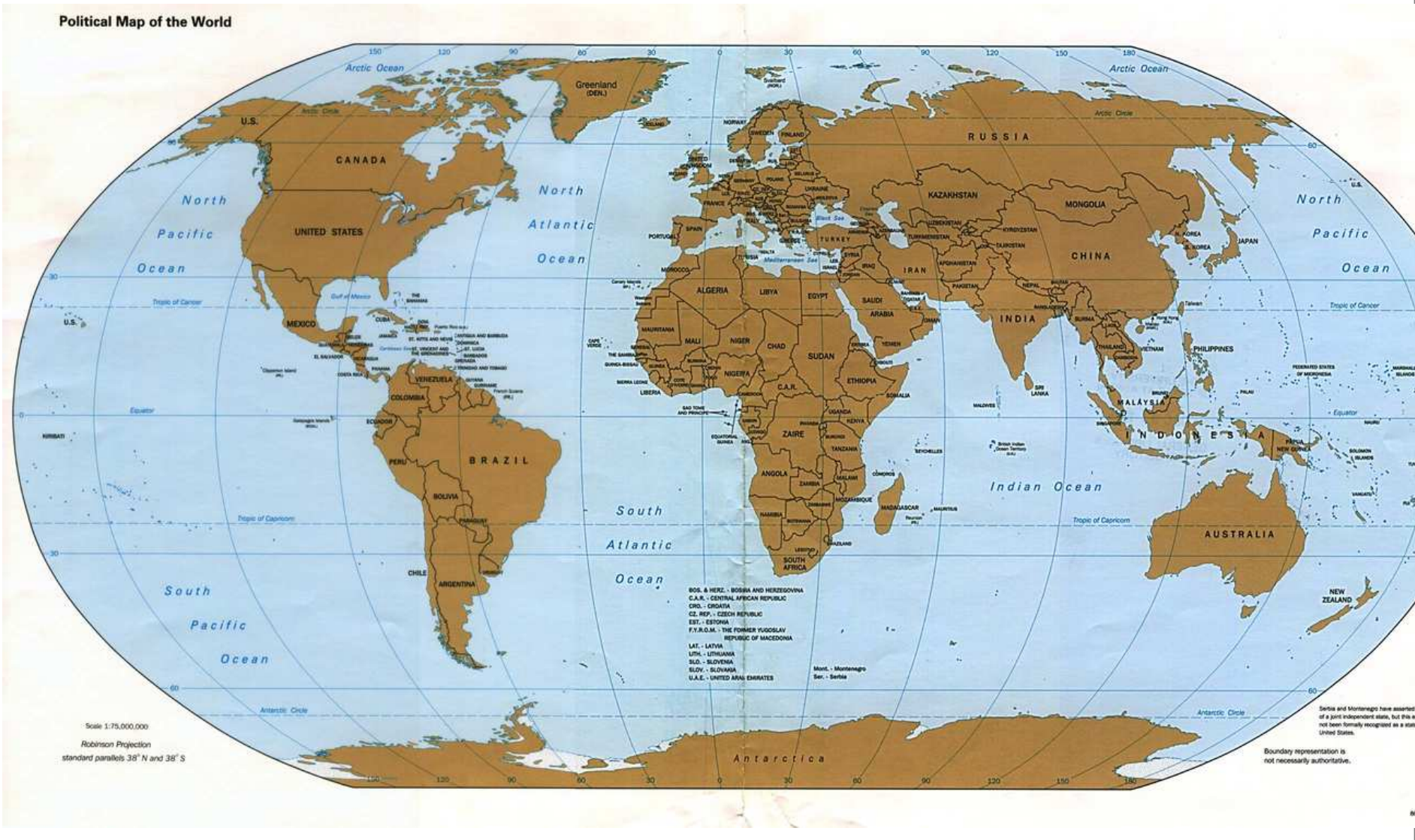
An Introduction to Martin-Lof's Constructive Type Theory and a computer implementation of it.

Bengt Nordström

bengt@cs.chalmers.se

ChungAng University, Seoul, Korea

on leave from Chalmers University, Göteborg, Sweden


Political Map of the World
Arctic Ocean
Greenland (DEN.)
U.S.
CANADA
North Pacific Ocean
North Atlantic Ocean
UNITED STATES
MEXICO
RUSSIA
KAZAKHSTAN
MONGOLIA
CHINA
IRAN
INDIA
JAPAN
ALGERIA
LIBYA
EGYPT
SAUDI ARABIA
MALI
NIGER
CHAD
SUDAN
NIGERIA
ETHIOPIA
C.A.R.
ZAIRE
ANGOLA
VENEZUELA
COLOMBIA
ECUADOR
PERU
BRAZIL
BOLIVIA
CHILE
ARGENTINA
Indian Ocean
South Atlantic Ocean
South Pacific Ocean
AUSTRALIA
INDONESIA
MALAYSIA
PHILIPPINES
NEW ZEALAND
SOUTH AFRICA
Antarctica
Tropic of Cancer
Tropic of Capricorn
Equator
Arctic Circle
Antarctic Circle
BOS. & HERZ. - BOSNIA AND HERZEGOVINA
C.A.R. - CENTRAL AFRICAN REPUBLIC
CRO. - CROATIA
CZ. REP. - CZECH REPUBLIC
EST. - ESTONIA
F.Y.R.O.M. - THE FORMER YUGOSLAV REPUBLIC OF MACEDONIA
LAT. - LATVIA
LITH. - LITHUANIA
SLO. - SLOVENIA
SLOV. - SLOVAKIA
U.A.E. - UNITED ARAB EMIRATES
Mont. - Montenegro
Ser. - Serbia
Scale 1:75,000,000
Robinson Projection
standard parallels 38° N and 38° S
Boundary representation is not necessarily authoritative.

Europe


EUROPE
Greenland Sea
Norwegian Sea
ICELAND
Reykjavík
Jan Mayen (NORWAY)
Barents Sea
NORWAY
SWEDEN
FINLAND
RUSSIA
Oslo
Stockholm
Helsinki
Tallinn
ESTONIA
LATVIA
Riga
LITHUANIA
Vilnius
Minsk
BELARUS
Moscow
North Atlantic Ocean
North Sea
UNITED KINGDOM
IRELAND
Dublin
London
DENMARK
Berlin
Warsaw
POLAND
GERMANY
Prague
UKRAINE
Kiev
Paris
FRANCE
Bay of Biscay
Vienna
Budapest
ROMANIA
Bucharest
Black Sea
BULGARIA
Sofia
ITALY
Rome
Adriatic Sea
Tyrrhenian Sea
Ionian Sea
GREECE
TURKEY
PORTUGAL
Lisbon
SPAIN
Madrid
Mediterranean Sea
Algiers
ALGERIA
TUNISIA
MOROCCO
Rabat
Scale 1: 19,500,000
Lambert Conformal Conic Projection, standard parallels 40°N and 56°N
802910AI (R01083) 6-02

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 identificatons 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	**data** Or A B = Ori1 A | Ori2 B;
$A \,\&\, B$	a proof of A and a proof of B	**data** And A B = Andi A B;
$A \supset B$	a method taking a proof of A to a proof of B	**data** Implies A B = Impi A -> B;
$Falsity$		**data** Falsity = ;

Constructors are introduction rules

Ori1	$\in \quad A \rightarrow A \vee B$	$\dfrac{A}{A \vee B}$
Ori2	$\in \quad B \rightarrow A \vee B$	$\dfrac{B}{A \vee B}$
Andi	$\in \quad A \rightarrow B \rightarrow A \,\&\, B$	$\dfrac{A \quad B}{A \,\&\, B}$
Impli	$\in \quad (A \rightarrow B) \rightarrow A \supset B$	$\dfrac{\begin{matrix}[A] \\ B\end{matrix}}{A \supset B}$

Elimination rules can be defined

orel $\in A \vee B \to (A \to C) \to (B \to C) \to C$

$$\frac{A \vee B \qquad \begin{matrix}[A] \\ C\end{matrix} \qquad \begin{matrix}[B] \\ C\end{matrix}}{C}$$

orel $(\textbf{Ori1}\ a)\ f\ g = f\ a$

orel $(\textbf{Ori2}\ b)\ f\ g = g\ b$

Elimination rules can be defined

orel $\in A \vee B \to (A \to C) \to (B \to C) \to C$

orel $(\textbf{Ori1}\ a)\ f\ g = f\ a$

orel $(\textbf{Ori2}\ b)\ f\ g = g\ b$

$$\frac{A \vee B \qquad \begin{matrix}[A] \\ C\end{matrix} \qquad \begin{matrix}[B] \\ C\end{matrix}}{C}$$

andel $\in A\,\&\,B \to (A \to B \to C) \to C$

andel $(\textbf{Andi}\ a\ b)\ f = f\ a\ b$

$$\frac{A\,\&\,B \qquad \begin{matrix}[A, B] \\ C\end{matrix}}{C}$$

Elimination rules can be defined

orel $\in A \vee B \to (A \to C) \to (B \to C) \to C$

$$\frac{A \vee B \qquad \begin{matrix}[A] \\ C\end{matrix} \qquad \begin{matrix}[B] \\ C\end{matrix}}{C}$$

orel $(\textbf{Ori1}\ a)\ f\ g = f\ a$

orel $(\textbf{Ori2}\ b)\ f\ g = g\ b$

andel $\in A\,\&\,B \to (A \to B \to C) \to C$

$$\frac{A\,\&\,B \qquad \begin{matrix}[A, B] \\ C\end{matrix}}{C}$$

andel $(\textbf{Andi}\ a\ b)\ f = f\ a\ b$

implel $\in A \supset B \to A \to B$

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

implel $(\textbf{Impli}\ f)\ a = f\ a$

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:

$$\textbf{Impli}\ (\lambda x.\textbf{Andi}\ (\textbf{andel}\ x\ \lambda y.\lambda z.z)\ (\textbf{andel}\ x\ \lambda y.\lambda z.y))$$

Two questions:

- Is is 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 $A \in$ Set then *El*(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

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

- constants are either primitive or defined

Constants

There are two kinds of constants:

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

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

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:

$$
\begin{array}{rcl}
\mathsf{N} & \in & \mathsf{Set} \\
0 & \in & \mathsf{N} \\
\mathsf{s} & \in & (\mathsf{N})\mathsf{N} \\
\& & \in & (\mathsf{Set},\ \mathsf{Set})\mathsf{Set} \\
\&\mathbf{I} & \in & (A \in \mathsf{Set},\ B \in \mathsf{Set},\ A,\ B)A \,\&\, B \\
\Pi & \in & (A \in \mathsf{Set},\ (A)\mathsf{Set})\ \mathsf{Set} \\
\lambda & \in & (A \in \mathsf{Set}, B \in (A)\mathsf{Set}, (x \in A)B(x)) \\
 & & \Pi(A, B)
\end{array}
$$

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:

$$
\begin{aligned}
2 &\equiv \mathsf{succ}(\mathsf{succ}(0)) \in \mathsf{N} \\
\forall &\equiv \Pi \in (A \in \mathsf{Set},\ (A)\mathsf{Set})\,\mathsf{Set} \\
+ &\equiv [x, y]\mathsf{natrec}([x]\mathsf{N}, x, y, [u, v]\mathsf{succ}(v)) \in (\mathsf{N}, \mathsf{N})\mathsf{N} \\
\rightarrow &\equiv [A, B]\Pi(A, [x]B)) \in (A, B \in \mathsf{Set})\mathsf{Set}
\end{aligned}
$$

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:

$$\&\mathbf{E} \in (A \in \mathsf{Set},\ B \in \mathsf{Set},\ C \in (A,\ B)\mathsf{Set},$$

$$(x \in A,\ y \in B)C(\&\mathbf{I}(x, y)),\ (z \in A \,\&\, B))C(z)$$

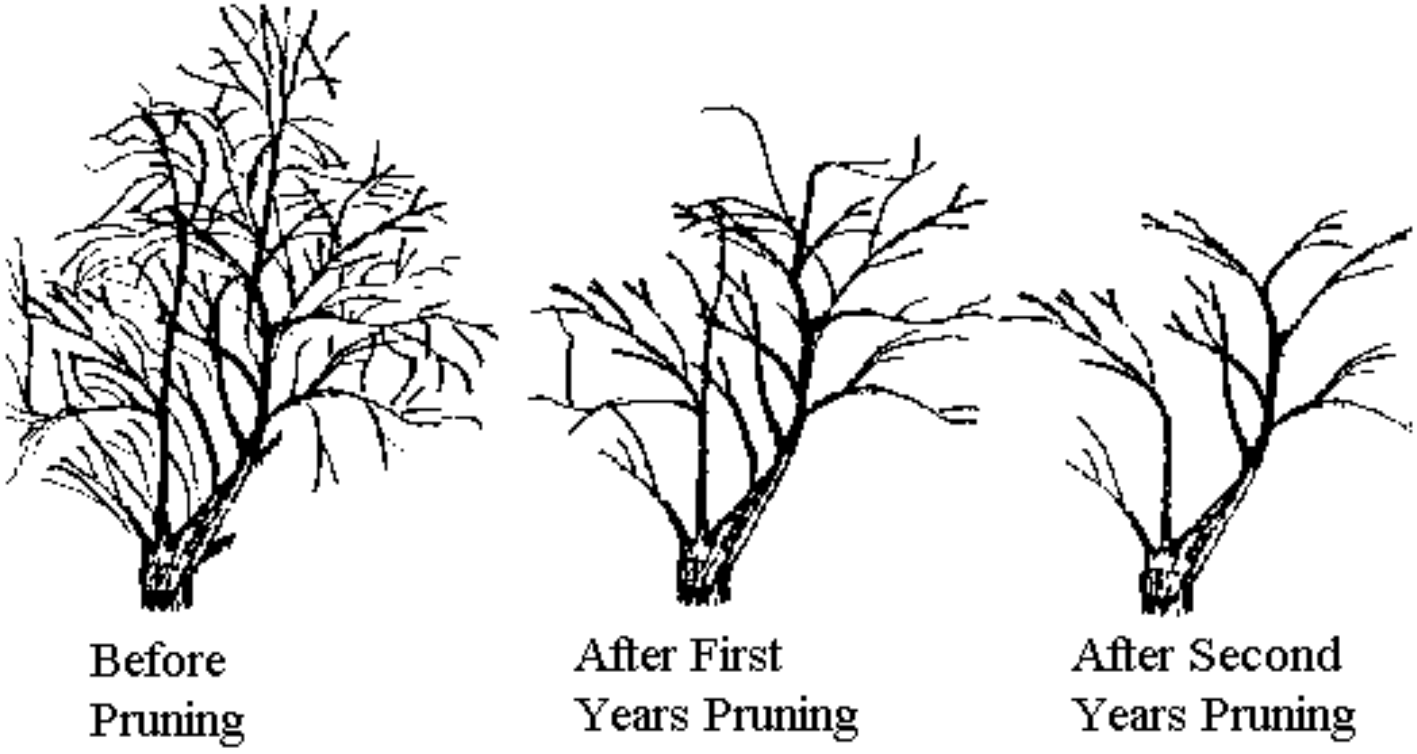
$$\&\mathbf{E}(A, B, C, f, \&\mathbf{I}(a, b)) \equiv f(a, b)$$

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).



Place holders

We use the notation

$$\square_1, \ldots \square_n$$

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
- $C(\square_3, \ldots \square_n)$

Refinement of an object

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

$$c \in C$$

$$c = \Box_0$$

where the expected type of $\Box_0$ is C .
In general, we are in a situation like

$$c = \ldots \Box_1 \ldots \Box_2 \ldots$$

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

$$c(\square_1, \ldots \square_n) \in A$$

where $\square_1 \in B_1, \ldots, \square_n \in B_n$. The system computes n and the types of the new place holders as well as some constraints from the condition that the type of $c(\square_1, \ldots \square_n)$ must be equal to A.

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

Refinement of an object: abstraction

To refine a place holder

$$\Box_0 \in A$$

with an abstraction is to replace it by

$$[x]\Box_1 \in A$$

The system checks that A is a functional type $(x \in B)C$ and the expected type of $\Box_1$ 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 $x \in B$.