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

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

Basic idea

Not only that a proof of P is an object in the type P, but also

the *process of proving* is identified with the *process of building* an object in the type P.

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**.

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

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

- constants are either primitive or defined

Constants

There are two kinds of constants:

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

defined: have a type and a definiens.

- explicitly defined
- implicitly defined

Primitive constants

- have only a type, no definiens.
- computes to themselves (i.e. are values).
- constructors in functional languages.
- introduction rules and formation rules in logic

Examples:

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

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:

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

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 defiendum to the definiens is the contraction rule).

Examples:

$$\&\mathsf{E} \in (A \in \mathsf{Set},\ B \in \mathsf{Set},\ C \in (A,\ B)\mathsf{Set},$$
$$(x \in A,\ y \in B)C(\&\mathsf{I}(x, y)),\ (z \in A \,\&\, B))C(z)$$
$$\&\mathsf{E}(A, B, C, f, \&\mathsf{I}(a, b)) \equiv f(a, b)$$

Place holders

If we are going to edit types and objects, the we must have a way to express incomplete expressions.
We use the notation

$$\Box_1, \ldots \Box_n$$

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 construct an object

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

$$c \in \Box_1$$
$$c = \Box_2$$

We must first give the type of c by filling in $\Box_1$.
We can either enter text from the keyboard, or do it stepwise, replace it by

- $(x \in \Box_3)\Box_4$ — a function type, or
- Set, or
- $C(\Box_3, \ldots \Box_n)$

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

$$\square_0 \in A$$

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 $x \in B$.