Just What is it that Makes Martin-Löf's Type Theory so Different, so Appealing?*

Neil Leslie

May 24, 1999

Class. Here come our friends Form and Letter. Boys, we are having a most interesting discussion on intuitionism.

Letter. Could you speak about anything else with good old Int? He is completely submerged in it.

Int. Once you have been struck by the beauty of a subject devote your life to it!

Form. Quite so! Only I wonder how there can be beauty in so indefinite a thing as intuitionism. None of your terms are well-defined, nor do you give exact rules of derivation. Thus one for ever remains in doubt as to which reasonings are correct and which are not.

A. Heyting *Intuitionism: An introduction.*

1 Introduction

Martin-Löf's type theory (M-LTT) was developed by Per Martin-Löf in the early 1970s, as a constructive foundation for mathematics.

> "The theory of types with which we shall be concerned is intended to be a full scale system for formalizing intuitionistic mathematics as developed, for example, in the book by Bishop 1967"
>
> P. Martin-Löf *An Intuitionistic Theory of Types*

So we have two of the key features. The theory is:

- constructive (intuitionistic, but we will blur the distinction);
- foundational.

*With apologies to both Per Martin-Löf and Richard Hamilton

Because the theory is foundational we should not expect a formal explanation of it. If we were to give a formal explanation of the theory then we should be appealing to another theory. Then *that* would be a better theory to use as a foundation.

Other key features are:

- The theory is open-ended, in the sense that we are free to add new types.
- The theory is justified in an "informal, but rigorous" fashion, usually based on a Dummett-Prawitz style theory of meaning. "Informal" does not mean "sloppy" or "careless": far from it. The theory is developed with a great deal of care and attention to detail.
- We make extensive use of the Curry-Howard "propositions-as-types" analogy.
- The theory is clear and simple. Because of this it has been used for everything from program derivation to handling donkey sentences.
- The theory is also known as:
  - intuitionistic type theory;
  - constructive set theory.

2 Before formalisation:judgement, proof, proposition

The informal notion of *proof* and *proposition* are intertwined and we must grasp these together. A proposition is something of which we can understand an arbitrary (direct) proof. We make a judgement when we assert that we have a proof of a proposition. The rules that we present will (generally) be rules for manipulating *judgements*

We immediately see an analogy between propositions and types. A type is something of which we can recognise an arbitrary (canonical) element.

Notice that we have not said that we can list *all* the possible proofs of a proposition, just that we can recognise an arbitrary (direct) one. We identify the meaning of a proposition with its proofs.

We must be careful to distinguish direct and indirect proofs (or canonical and non-canonical elements). We will accept that $10^{10^{10}}$ is a natural number. To see this we have to put it into a form where we can *immediately* see that it is a natural number. That is, we have to *evaluate* it. We call canonical objects *values* of a type. Now we can identify the meaning of a proposition with its *canonical* proofs.

We will write

$$a : A$$

for the judgement that a is a proof (witness) of the proposition A.

We can give alternative readings of this judgement. The two most obvious are:

- a has type A;
- a is a program which meets the specification A.

Having made this very strong identification of types, propositions and specifications we shall not worry about, for example, what **proposition** the **type** of natural numbers is.

An important point to note (for computational purposes anyway) is that proofs *are* programs. These programs are expressed in a lazy functional language. The semantics we give for the theory is an operational one (it's a constructive theory, so we assume that we know what computation is about before we started!) and laziness is quite important.

3 The judgements required for the theory

There are four forms of judgement that we need to use.

- A is a type, written $A\ type$.

  To justify this judgement we need to explain what we need to know in order to form the type A. The theory is predicative, so we do **not** have $A : Type$.

- a is an element of type A, written $a : A$.

  To justify this judgement we need to explain how to evaluate a.

- a and b are equal elements of type A, written $a = b : A$ or sometimes $a =_A b$.

  To justify this judgement we need to explain how to evaluate a and b to values of A.

- A and B are equal types, written $A = B$.

  To justify this judgement we need to show that values of A are values of B, equal values of A are equal values of B, and *vice versa*.

Since "meaning is use" using these judgements will make them clearer.

4 Starting the theory

Now we will start to define some types. Before we do this we have to introduce a notation for writing expressions.

4.1 Aritied expressions

We need to be able to write expressions down formally. We use a syntactic theory of aritied expressions. The arities just tell us how to plug bits of syntax together. Each variable and (primitive) constant has an arity.

- Single, saturated expressions have arity o;
- Combined expressions will have arity $\alpha_1 \otimes \ldots \otimes \alpha_n$, where $\alpha_1 \ldots \alpha_n$ are arities.
- unsaturated expressions will have arity $\alpha \longrightarrow \beta$, where α and β are arities.

We shall skip this material, except to note two things.

- We have a *syntactic* abstraction: if x is a variable of arity α and b an expression of arity β then $(x)b$ is an expression of arity $\alpha \longrightarrow \beta$; and
- We will drop brackets and so on, and will abuse notation and freely write, for example, $f(x)$ in place of $(y)f(x, y)$.

4.2 Defining a type

As an example we shall define the type of the natural numbers. When we define a type we (typically) present these rules:

- formation rule;
- introduction rules;
- introduction of equal elements;
- computation rules;
- elimination rule;
- equality rules.

The rules are written in a natural deduction format, and we have skipped a great deal of material to get here.

5 Digression

At this point we should have a long and detailed discussion about proof-theoretic justification of logical laws. Sadly, however we do not have the time to do this. We shall simply assert that we are using a Prawitz-Dummett style, proof-theoretic meaning theory, and make a few of minor comments here. It is worth noting that Dummett starts off by considering what an adequate theory of meaning might be. Having constructed such a thing we find that constructive logic comes out naturally. Dummett does start out to justify claims for a special place for constructive logic. However, he seems quite happy with the revisionist aspect of his work. Somewhere or other he writes:

> "If the price of this solution to the problem of the basis of those theories is that argumentation within mathematics is compelled to become more cautious than that which classical mathematicians have been accustomed to use, and more sensitive to distinctions to which they have accustomed to be indifferent, it is a price worth paying, especially if the resulting versions of the theories indeed prove more apt for their applications"

and:

> "The prejudices of mathematicians do not constitute an argument, however: the important question for us is whether constructive mathematics is adequate for applications."

5.1 The form of the rules

An *introduction rule* for a logical constant $*$ is a rule which has $*$ as the principal connective in its conclusion.

An *elimination rule* for a logical constant $*$ is a rule which has a premiss whose principle connective is $*$.

Thus a given rule can be both an introduction and an elimination rule, and can be an elimination rule for more than one connective.

However we shall restrict our attention to introduction and elimination rules of particular forms, and we will typically refer to these as *the* introduction and elimination rules for the connective.

The introduction rules that we present will be taken to be self-justifying. In this we are following Gentzen's observation in §5.13 of "Investigations into Logical Deduction":

> "The introductions represent, as it were, the 'definitions' of the symbols concerned, and the eliminations are no more, in the final analysis, than the consequences of these definitions."

This is all very well, but we must avoid *tonk*. The rule for *tonk* introduction is:

$$\frac{A}{A \textit{ tonk } B}\ \text{Tonk I}$$

Rule 1: Tonk Intro

and the rule for *tonk* elimination is:

$$\frac{A\ tonk\ B}{B}\ \text{Tonk E}$$

Rule 2: Tonk Elim

And now we can prove:

$$\dfrac{\dfrac{\dfrac{[A]^1}{A\ tonk\ B}\ \text{Tonk I}}{B}\ \text{Tonk E}}{A \supset B}\ \text{Hook I}^1$$

Proof 1: $A \supset B$

We are rescued at this point (I believe) by the computation rules. One way to look at *tonk* is to consider it as allowing us to write non-normalisable proofs. Since we know that proof normalisation and program evaluation (computation) are intimately connected, we should not be surprised that the computation rules are what rescues us.

Thus for each connective we will have:

- a number of introduction rules;
- a number of computation rules;
- exactly one elimination rule, justified by reference to the introduction and computation rules;
- a number of rules for equality.

6 Defining the type of the natural numbers

As a first example we will look at the type of the natural numbers. The formation rules for the natural numbers is:

$$\frac{}{Nat\ type}\ \text{Nat formation}$$

Rule 3: Nat formation

The introduction rules for the natural numbers are:

$$\frac{}{zero : Nat}\ \text{Nat Intro}$$

Rule 4: Nat Intro

and:

$$\frac{n : Nat}{succ(n) : Nat}\text{ Nat Intro}$$

Rule 5: Nat Intro

Rules 4 and 5 tell us that *zero* is a *canonical* element of Nat, and that if n is an element of Nat then $succ(n)$ is a *canonical* element of Nat. We are lazy here as n does not have to be a canonical element of Nat.

The rules for introducing equal canonical elements are:

$$\frac{}{zero = zero : Nat}\text{ Nat Intro}$$

Rule 6: Nat Intro

and:

$$\frac{n = m : Nat}{succ(n) = succ(m) : Nat}\text{ Nat Intro}$$

Rule 7: Nat Intro

Typically, when we present rules for a type we do not present the rules for introducing equal canonical elements as these are usually obvious.

How do we compute with a natural number? We use *structural* recursion, because this reflects the way that we have (inductively) defined the type. It is worth noting that all functions in M-LTT are total, and that typing implies termination.

We call the structural recursion operator for natural numbers *natrec*, and it has the following computation rules:

$$\frac{n \longrightarrow zero \qquad d \longrightarrow d'}{natrec(d, e, n) \longrightarrow d'}\text{ natrec comp}$$

Rule 8: natrec comp

and:

$$\frac{n \longrightarrow succ(m) \qquad e(m, natrec(d, e, m)) \longrightarrow e'}{natrec(d, e, n) \longrightarrow e'}\text{ natrec comp}$$

Rule 9: natrec comp

We call $natrec$ the *non-canonical constant* associated with the natural numbers.

The elimination rule is now, effectively, a typing rule for $natrec(d, e, n)$, and it is:

$$\dfrac{n : Nat \qquad \begin{array}{c}[x : Nat] \\ \vdots \\ C(x)\ type\end{array} \qquad d : C(zero) \qquad \begin{array}{c}\left[\begin{array}{c} y : Nat \\ z : C(y)\end{array}\right] \\ \vdots \\ e(y, z) : C(succ(y))\end{array}}{natrec(d, e, n) : C(n)}\ \text{Nat elim}$$

Rule 10: Nat elim

This rule is justified by a consideration of the introduction and computation rules.

And we also get rules for equality between expressions formed by *natrec*:

$$\dfrac{\begin{array}{c}[x : Nat] \\ \vdots \\ C(x)\ type\end{array} \qquad d : C(zero)}{natrec(d, e, zero) = d : C(n)}\ \text{Natrec eq 1}$$

Rule 11: natrec eq 1

and:

$$\dfrac{n : Nat \qquad \begin{array}{c}[x : Nat] \\ \vdots \\ C(x)\ type\end{array} \qquad d : C(zero) \qquad \begin{array}{c}\left[\begin{array}{c} y : Nat \\ z : C(y)\end{array}\right] \\ \vdots \\ e(y, z) : C(succ(y))\end{array}}{natrec(d, e, succ(n)) = e(n, natrec(d, e, n)) : C(succ(n))}\ \text{Natrec eq 1}$$

Rule 12: natrec eq 2

Again we typically don't give the rules for equality between expression formed by the non-canonical constant because they are usually so straightforward.

6.1 Summarising how we defined the natural numbers

In order to define the natural numbers we:

- gave the formation rule;
- gave introduction rules which describe the *canonical* elements;
- explained how to compute with values of *Nat*;

- presented an elimination rule, which we justified by reference to the introduction and computation rules.

This is the pattern that we follow for all types.

7 "Logical" types

In this section we will present the rules for the types which we can interpret as logical connectives.[1]

7.1 + types

The formation rule for the + types is:

$$\frac{A\ type \qquad B\ type}{A + B\ type} + \text{Form}$$

Rule 13: + Form

The introduction rules are:

$$\frac{a : A}{inl(a) : A + B} + \text{Intro L}$$

Rule 14: + Intro L

$$\frac{b : B}{inr(b) : A + B} + \text{Intro R}$$

Rule 15: + Intro R

The non-canonical constant associated with the + types is called *when*, and the rules for its evaluation are as follows:

$$\frac{f \longrightarrow inl(l) \qquad d(l) \longrightarrow d'}{when(d, e, f) \longrightarrow d'} \text{when Comp}$$

Rule 16: when Comp

and:

$$\frac{f \longrightarrow inr(r) \qquad e(r) \longrightarrow e'}{when(d, e, f) \longrightarrow e'} \text{when Comp}$$

Rule 17: when Comp

[1] Of course we mean the connectives of constructive logic.

The elimination rule for $+$ is:

$$\frac{f : A + B \qquad \begin{array}{c}[z : A + B] \\ \vdots \\ C(z)\ type\end{array} \qquad \begin{array}{c}[x : A] \\ \vdots \\ d(x) : C(inl(x))\end{array} \qquad \begin{array}{c}[y : B] \\ \vdots \\ e(y) : C(inr(y))\end{array}}{when(d, e, f) : C(f)} + \text{Elim}$$

Rule 18: + elim

Now we shall explicitly think of types as propositions, and we shall ignore the dependency of C on $A + B$ in Rule 18 to produce:

$$\frac{f : A + B \qquad C\ prop \qquad \begin{array}{c}[x : A] \\ \vdots \\ d(x) : C\end{array} \qquad \begin{array}{c}[y : B] \\ \vdots \\ e(y) : C\end{array}}{when(d, e, f) : C} + \text{Elim Prop}$$

Rule 19: + elim prop

Apart from the premiss C *prop*, and the decoration with the formal proof objects, this is just the rule for $\vee$ elimination. Rules 14 and 15 are just the introduction rules for $\vee$. Hence we make the definition:

$$A \vee B =_{\text{def}} A + B$$

The other connectives are handled in a similar fashion.

7.2 Σ types

If B is a family of types over A we can form $\Sigma(A, B)$:

$$\frac{A\ type \qquad \begin{array}{c}[x : A] \\ \vdots \\ B(x)\ type\end{array}}{\Sigma(A, B)\ type} \Sigma\ \text{Form}$$

Rule 20: Σ Form

The values in $\Sigma(A, B)$ are pairs:

$$\frac{a : A \qquad \begin{array}{c}[x : A] \\ \vdots \\ b(x) : B(x)\end{array}}{pair(a, b) : \Sigma\,(A, B)} \Sigma\ \text{I}$$

Rule 21: Σ Intro

The non-canonical constant associated with the Σ types is called $split$ and its computation rule is:

$$\frac{c \longrightarrow pair(a, b) \qquad d(a, b) \longrightarrow d'}{split(d, c) \longrightarrow d'} \text{ split Comp}$$

Rule 22: split Comp

The elimination rule is:

$$\frac{c : \Sigma(A, B) \qquad \begin{array}{c}[z : \Sigma(A, B)] \\ \vdots \\ C(z)\ type\end{array} \qquad \begin{array}{c}\left[\begin{array}{c} x : A \\ y(w) : B(w)[w : A]\end{array}\right] \\ \vdots \\ d(x, y) : C(pair(x, y))\end{array}}{split(d, c) : C(c)} \ \Sigma \text{ Elim}$$

Rule 23: Σ elim

The elimination rule for the Σ types introduces a new feature we have not yet seen, the "hypothetical assumption".

If we follow the same reasoning that allowed us to define $\vee$ we see that we can make this definition:

$$(\exists x : A)B(x) =_{\text{def}} \Sigma(A, B)$$

Further, suppose that B is a type, and not a family of types over A. We can then make this definition:

$$A\&B =_{\text{def}} \Sigma(A, B)$$

If we were used to classical logic we would be a bit surprised here: classically, the "existential quantifier" is thought of as a generalisation of "disjunction".

The elimination rule for & looks like this (suppressing the well-formedness premiss and dependency of C on $A\&B$):

$$\frac{c : A\&B \qquad \begin{array}{c}\left[\begin{array}{c} x : A \\ y : B\end{array}\right] \\ \vdots \\ d(x, y) : C\end{array}}{split(d, c) : C} \ \& \text{ Elim}$$

Rule 24: & elim

We can recover the "usual" elimination rules. First we replace C by A in Rule 24 to get:

$$\cfrac{c : A\&B \qquad \begin{array}{c} \left[\begin{array}{c} x : A \\ y : B \end{array}\right] \\ \vdots \\ d(x, y) : A \end{array}}{split(d, c) : A}\ \&\ \text{Elim l}$$

Rule 25: & elim l

If we choose d to be $(a, b)a$, then the second premiss is immediate, and so we get this rule:

$$\cfrac{c : A\&B}{split((a, b)a, c) : A}\ \&\ \text{Elim l}$$

Rule 26: & elim l

We can now define:

$$fst(p) =_{\text{def}} split((a, b)a, p)$$

We can define:

$$snd(p) =_{\text{def}} split((a, b)b, p)$$

and recover the other "usual" elimination rule for &.

7.3 Π types

If B is a family of types over A we can form $\Pi(A, B)$:

$$\cfrac{A\ type \qquad \begin{array}{c} [x : A] \\ \vdots \\ B(x)\ type \end{array}}{\Pi(A, B)\ type}\ \Pi\ \text{Form}$$

Rule 27: Π Form

The values in $\Pi(A, B)$ are functions:

$$\cfrac{\begin{array}{c} [x : A] \\ \vdots \\ b(x) : B(x) \end{array}}{\lambda(b) : \Pi(A, B)}\ \Pi\ \text{I}$$

Rule 28: Π Intro

Here we see function abstraction, which allows us to form functions. This is a different notion from the notion of syntactic abstraction that we have been using.

The rule for introducing equal values of $\Pi(A, B)$ is:

$$\frac{\begin{array}{c}[x : A] \\ \vdots \\ b(x) = c(x) : B(x)\end{array}}{\lambda(b) = \lambda(c) : \Pi(A, B)} \text{ Equal } \Pi \text{ I}$$

Rule 29: Equal Π Intro

The computation rule for $funsplit$ is:

$$\frac{f \longrightarrow \lambda(b) \qquad e(b) \longrightarrow e'}{funsplit(e, f) \longrightarrow e'} \text{ funsplit Comp}$$

Rule 30: funsplit Comp

Notice that this computation rule is uniform with the other ones we have given. It is *not* the rule for using $apply$, which is this rule:

$$\frac{f \longrightarrow \lambda(b) \qquad b(x) \longrightarrow b'}{apply(f, x) \longrightarrow b'} \text{ apply Comp}$$

Rule 31: apply Comp

We can define $apply$ using $funsplit$:

$$apply(f, x) =_{\text{def}} funsplit((y)(y(x)), f)$$

The elimination rule for the Π types looks like this:

$$\frac{f : \Pi(A, B) \qquad \begin{array}{c}[x : \Pi(A, B)] \\ \vdots \\ C(x)\ type\end{array} \qquad \begin{array}{c}[y(z) : B(z)[z : A]] \\ \vdots \\ e(y) : C(\lambda(y))\end{array}}{funsplit(e, f) : C(f)} \ \Pi \text{ Elim}$$

Rule 32: Π elim

Once again a little thought will convince us that we can define:

$$(\forall a : A)B(x) =_{\text{def}} \Pi(A, B)$$

If B is not dependent on A then we can define:

$$A \supset B =_{\text{def}} \Pi(A, B)$$

Modus ponens comes out as a case of the Π elimination rule.

Again this is in contrast with the situation in classical logic where $\forall$ is seen as a generalisation of &.

7.4 The empty type

The empty type is a special case of an enumeration type.

$$\frac{}{\{\}\ type}\ \text{Empty form}$$

Rule 33: Empty form

The empty type is unusual in that it (obviously) has no introduction rules.

The general form of the non-canonical constant for the enumeration types is a *case* expression. In general $case_{\{x_1,\ldots,x_n\}}$ will take $n+1$ arguments and will have n computation rules like this:

$$\frac{a \longrightarrow x_i \qquad d_i(x_i) \longrightarrow d'}{case_{\{x_1,\ldots,x_n\}}(d_1,\ldots,d_n,a) \longrightarrow d'}\ case_{\{x_1,\ldots,x_n\}}\ \text{comp}$$

Rule 34: case comp

The computation rule for $case_{\{\}}$ breaks this pattern slightly. The constant $case_{\{\}}$ takes one argument and has one computation rule. We need to suppose that ω is a hypothetical value of the empty type. Then the computation rule tells us that if we could evaluate a to ω then we could evaluate $case_{\{\}}(a)$ to anything:

$$\frac{a \longrightarrow \omega \qquad d(\omega) \longrightarrow d'}{case_{\{\}}(a) \longrightarrow d'}\ \text{case}_{\{\}}\ \text{comp}$$

Rule 35: case$_{\{\}}$ comp

So we get this elimination rule:

$$\frac{a : \{\} \qquad \begin{matrix}[x:\{\}]\\ \vdots \\ C(x)\ type\end{matrix}}{case_{\{\}}(a) : C(a)}\ \{\}\ \text{elim}$$

Rule 36: $\{\}$ elim

Now we can define:

$$\bot =_{\text{def}} \{\}$$

and then define negation:

$$\neg A =_{\text{def}} A \supset \bot$$

8 Example: Some simple logic

For historical reasons we shall prove:

$$A \supset \neg\neg A$$

The proof looks like

$$\dfrac{\dfrac{\dfrac{[A] \quad [A \supset \bot]}{\bot}\text{Modus Ponens}}{(A \supset \bot) \supset \bot} \supset \text{I}}{A \supset \neg\neg A} \supset \text{I}$$

Proof 2: A proof

Proving a proposition is just finding an inhabitant of the type. We can annotate the proof with the formal proof objects:

$$\dfrac{\dfrac{\dfrac{[x : A] \quad [f : A \supset \bot]}{apply(f, x) : \bot}\text{Modus Ponens}}{\lambda((f)apply(f, x)) : (A \supset \bot) \supset \bot} \supset \text{I}}{\lambda((x)\lambda((f)apply(f, x))) : A \supset \neg\neg A} \supset \text{I}$$

Proof 3: A proof with formal proof objects

If we interpret this typically we see that we have shown:

$$\lambda((a)(\lambda((f)(apply(f, a))))) : \Pi(A, \Pi(\Pi(A, \{\}), \{\}))$$

Had we used the "basic" Π elimination rule (Rule 32) we would have shown:

$$\lambda((a)(\lambda((f)(funsplit((x)(x(a), f)))))) : \Pi(A, \Pi(\Pi(A, \{\}), \{\}))$$

It should be obvious at this point that some form of mechanical support is useful.

9 Summary

In this note we have attempted to present and give a flavour of Martin-Löf's Type Theory. We have tried to highlight the use of proof theoretical justification of the rules, without explaining this in full detail. We presented the rules for various types, trying to emphasise the uniformity of the presentation. The types that we presented included those needed to express the logical constants $\perp$, $\vee$, &, $\supset$, $\exists$, and $\forall$.