Another introduction to Martin-Löf's Intuitionistic Type Theory

Silvio Valentini
Dip. di Matematica Pura ed Applicata – Univ. di Padova
via G. Belzoni n.7, I–35131 Padova, Italy
e-mail: valentini@pdmat1.math.unipd.it

September 24, 1996

1 Introduction

Since the 70s Martin-Löf has developed, in a number of successive variants, an Intuitionistic Theory of Types [ML84, NPS90] (ITT for short in the following). The initial aim was to provide a formal system for constructive mathematics but the relevance of the theory also in computer science was soon recognized. In fact, from an intuitionistic perspective, to define a constructive set theory is completely equivalent to define a logical calculus [H80] or a language for problem specification [K32], and hence the topic is of immediate relevance both to mathematicians, logicians and computer scientists. Moreover, since an element of a set can also be seen as a proof of the corresponding proposition or as a program which solves the corresponding problem, ITT is also a functional programming language with a very rich type structure and an integrated system to derive correct programs from their specification [PS86]. These pleasant properties of the theory have certainly contributed to the interest for it arisen in the computer science community, especially among those people who believe that program correctness is a major concern in the programming activity [BCMS89].

To develop ITT one has to pass through four steps:

- The first step is the definition of a theory of *expressions* which both allows to abstract on variables and has a decidable equality theory; indeed the first requirement is inevitable to gain the needed expressiveness while the last is essential in order to guarantee a mechanical verification of the correct application of the rules used to present ITT. Here the natural candidate is a sort of simple typed lambda calculus which can be found in the appendix.

- The second step is the definition of the *types* (respectively *sets*, *propositions* or *problems*) one is interested in; here the difference between the classical and the intuitionistic approach is essential: in fact a proposition is not

merely an expression supplied with a truth value but rather an expression such that one knows what counts as one of its *verifications* (respectively one of its *elements* or one of the *programs* which solves the problem).

- The third step is the choice of the *judgments* one can express on the types introduced in the previous step. Four forms of judgment are considered in ITT:

  1. the first form of judgment is obviously the judgment which asserts that an expression is a type;
  2. the second form of judgment states that two types are equal;
  3. the third form of judgment asserts that an expression is an element of a type;
  4. the fourth form of judgment states that two elements of a type are equal.

- The fourth step is the definition of the computation rules which allow to execute the programs defined in the previous point.

In the paper we will show some of the standard sets and propositions in [ML84] and some examples of application of the theory to actual cases. Finally the main meta-mathematical results on ITT will be recalled [BV92].

2 Judgments and Propositions

In a classical approach to the development of a formal theory one usually takes care to define a formal language only to specify the syntax (s)he wants to use while as regard to the intended semantics no *a priori* tie on the used language is required. Here the situation is quite different; in fact we want to develop a *constructive* set theory and hence we can assume no *external* knowledge on sets; then we do not merely develop a suitable syntax to describe something which exists *somewhere*. Hence we have "to put our cards on the table" at the very beginning of the game and to declare the kind of judgments on sets we are going to express. Let us consider the following example: let A be a set,then

$$a \in A$$

which means that a is an element of A, is one of the form of judgment we are interested in (but also the previous "A is a set" is already a form of judgment!). It is important to note that a logical calculus is meaningful only if it allows to derive judgments. Hence one should try to define it only after the choice of the judgments (s)he is interested in. A completely different problem is the definition of the suitable notation to express such a logical calculus; in this case it is probably more correct to speak of a *well-writing* theory instead of a logical calculus (cfr. appendix to see a well-writing theory suitable for ITT [V95]).

Let us show the form of the judgments we are going to use to develop our constructive set theory. The first is

$$\text{type-ness:} \quad A \; type$$

(equivalently A *set*, A *prop* and A *prob*) which reads "A is a type" (respectively "A is a set", "A is a proposition", "A is a problem") and states that A is a type. The second form of judgment is

$$\text{equal types:} \quad A = B \; type$$

which, provided that A and B are types, states that they are equal types. The next judgment is

$$\text{belong-ness:} \quad a \in A$$

which states that a is an element of the type A and finally

$$\text{equal elements:} \quad a = b \in A$$

which, provided that a and b are elements of the type A, states that a and b are equal elements of the type A.

3 Different readings of the judgments

Here we can see a peculiar aspect of a constructive set theory: we can give many different readings of the same judgment. Let us show some of them

A type	**a ∈ A**
A is a set	a is an element of A
A is a proposition	a is a proof of A
A is a problem	a is a method which solves A

The first reading conforms to the definition of a constructive set theory. The second one, which links type theory with intuitionistic logic, is due to Heyting [H56, H80]: it is based on the identification of a proposition with the set of its proofs. Finally the third reading, due to Kolmogorov [K32], consists on the identification of a problem with the set of its solutions.

3.1 On the judgment A *set*

Let us explain the meaning of the various forms of judgment; to this aim we may use many different philosophical approaches: here it is convenient to commit to an epistemological one.

What is a set? A set is defined by prescribing how its elements are formed.

Let us work out an example: the set $\mathcal{N}$ of natural numbers. We state that natural numbers form a set since we know how to construct its elements.

$$0 \in \mathcal{N} \qquad \frac{a \in \mathcal{N}}{s(a) \in \mathcal{N}}$$

i.e. 0 is a natural number and the successor of any natural number is a natural number.

Of course in this way we only construct the *typical* elements, we will call them the *canonical elements*, of the set of the natural numbers and we say nothing on elements like $3 + 2$. We can recognize also this element as a natural number if we understand that the operation $+$ is just a method such that, given two natural numbers, provides us, by means of a calculation, with a natural number in canonical form, i.e. $3 + 2 = s(2 + 2)$; this is the reason why, besides the judgment on belong-ness, we also need the equal elements judgment. For the type of the natural numbers we put

$$0 = 0 \in \mathcal{N} \qquad \frac{a = b \in \mathcal{N}}{s(a) = s(b) \in \mathcal{N}}$$

In order to make clear the meaning of the judgment A *set* let us consider another example. Suppose that A and B are sets, then we can construct the type $A \times B$, which corresponds to the cartesian product of the sets A and B, since we know what are its canonical elements:

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A \times B} \qquad \frac{a = c \in A \quad b = d \in B}{\langle a, b\rangle = \langle c, d\rangle \in A \times B}$$

So we explained the meaning of the judgment A *set* but meanwhile we also explained the meaning of two other forms of judgment.

What is an element of a set? An element of a set A is a method which, when executed, yields a canonical element of A as result.

When are two elements equal? Two elements a, b of a set A are equal if, when executed, they yield equal canonical elements of A as results.

It is interesting to note that one cannot construct a set if (s)he does not know how to produce its elements: for instance the subset of the natural numbers whose elements are the code numbers of the recursive functions which do not halt when applied to 0 is *not* a type in ITT, due to the halting problem. Of course it *is* a subset, since there is a way to describe it, and a suitable subset theory is usually sufficient in order to develop a great deal of standard mathematics (see for instance [SV95]).

3.2 On the judgment A *prop*

We can now immediately explain the meaning of the second way to read the judgment A *type*, i.e. to answer to the question: *What is a proposition?*

Since we want to identify a proposition with the set of its proofs, in order to answer to this question we have only to repeat for proposition what we said for sets: a proposition is defined by laying down what counts as a proof of the proposition.

This approach is completely different from the classical one; in the classical case a proposition is an expression provided with a truth value, while in the intuitionistic one to state that an expression is a proposition one has to clearly

state what (s)he is willing to accept as one of its proofs. Consider for instance the proposition $A\&B$: supposing A and B are propositions, then $A\&B$ is a proposition since we can state what is one of its proofs: a proof of $A\&B$ consists of a proof of A together with a proof of B, and so we can state

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A\&B}$$

but then $A\&B \equiv A \times B$ and in general we can identify sets and propositions.

3.2.1 A lot of propositions

Since we identify sets and propositions we can construct a lot of new sets if we know how to construct new propositions, i.e. if we can explain what is one of their proofs. Even if the intended meaning is completely different with respect to the classical case, we can recognize the following propositions.

a proof of	**consists of**
$A\&B$	$\langle a, b\rangle$, where a is a proof of A and b is a proof of B
$A \vee B$	$i(a)$, where a is a proof of A or $j(b)$, where b is a proof of B
$A \to B$	$\lambda(b)$, where b(x) is a proof of B provided that x is a proof of A
$(\forall x \in A)\ B(x)$	$\lambda(b)$, where b(x) is a proof of $B(x)$ provided that x is an element of A
$(\exists x \in A)\ B(x)$	$\langle a, b\rangle$, where a is an element of A and b is a proof of $B(a)$
$\bot$	nothing

It is worth noting that the intuitionistic meaning of the logical connectives allows to recognize that the connective $\to$ is just a special case of the quantifier $\forall$, provided the proposition $B(x)$ does not depend on the variable x, and the connective $\&$ is a special case of the quantifier $\exists$ under the same assumption.

Let us see which sets corresponds to the propositions we have defined so far.

The proposition	**corresponds to the set**
$A\&B$	$A \times B$, the cartesian product of the sets A and B
$A \vee B$	$A + B$, the disjoint union of the sets A and B
$A \to B$	$A \to B$, the set of the functions from A to B
$(\forall x \in A)\ B(x)$	$\Pi(A, B)$, the cartesian product of a family $B(x)$ of types indexed on the type A
$(\exists x \in A)\ B(x)$	$\Sigma(A, B)$, the disjoint union of a family $B(x)$ of types indexed on the type A
$\bot$	$\emptyset$, the empty set

4 Hypothetical judgments

So far we have explained the basic ideas of ITT; now we want to introduce a formal system. To this aim we must introduce the notion of hypothetical judgment: a hypothetical judgment is a judgment expressed under assumptions. Let us explain what is an assumption; here we only give some basic ideas while a formal approach can be found in [BV92]. Let A be a type; then the assumption

$$x : A$$

means both:

1. a variable declaration: the variable x has type A
2. a logical assumption: x is a hypothetical proof of A

The previous is just the simplest form of assumption, but we can also use

$$y : (x : A)\ B$$

which means that y is a function which maps an element $a \in A$ into the element $y(a) \in B$, and so on, by using assumptions of arbitrary complexity (see the appendix to find some explanation on the notation we use).

Let us come back to hypothetical judgments. We start with the simplest example of hypothetical judgment; suppose A *type* then we can state that B is a propositional function on the elements of A by putting

$$B(x)\ \mathit{prop}\ [x : A]$$

provided that we know what it counts as a proof of $B(a)$ for any element $a \in A$. For instance, one could consider the hypothetical judgment $x \neq 0 \to (\frac{3}{x} * x = 3)$ *prop* $[x : \mathcal{N}]$ whose meaning is straightforward.

Of course, we also have:

$$B(x) = D(x)\ [x : A]$$
$$b(x) \in B(x)\ [x : A]$$
$$b(x) = d(x) \in B(x)\ [x : A]$$

The previous are just the simplest forms of hypothetical judgments and in general, supposing

$$\begin{array}{l} A_1\ \mathit{type} \\ A_2(x_1)\ \mathit{type}\ [x_1 : A_1] \\ \quad \vdots \\ A_n(x_1, \ldots, x_{n-1})\ \mathit{type}\ [x_1 : A_1, \ldots, x_{n-1} : A_{n-1}] \end{array}$$

we obtain the hypothetical judgment

$$J\ [x_1 : A_1, \ldots, x_n : A_n(x_1, \ldots, x_{n-1})]$$

where J is one of the four forms of judgment we have considered.

5 The logic of types

We can now describe the full set of rules needed to describe one type. We will consider four kinds of rules: the first rule states the conditions required in order to *form* the type, the second one *introduces* the canonical elements of the type, the third rule explains how to use, and hence *eliminate*, the elements of the type and the last one how to *compute* using the elements of the type. For each kind of rules we will first give an abstract explanation and then we will show the actual rules for the cartesian product of two types.

Formation. How to form a new type (eventually using some previously defined types) and when two types constructed in this way are equal.

Example:

$$\frac{A\ type \quad B\ type}{A \times B\ type} \qquad \frac{A = C \quad B = D}{A \times B = C \times D}$$

which state that the cartesian product of two types is a type.

Introduction. What are the canonical elements of the type and when two canonical elements are equal.

Example:

$$\frac{a \in A \quad b \in B}{\langle a, b\rangle \in A \times B} \qquad \frac{a = c \in A \quad b = d \in B}{\langle a, b\rangle = \langle c, d\rangle \in A \times B}$$

which state that the canonical elements of the cartesian product $A \times B$ are the couples whose first element is in A and second element is in B.

Elimination. How to define functions on the elements of the type defined by the introduction rules.

Example:

$$\frac{c \in A \times B \quad d(x, y) \in C(\langle x, y\rangle)\ [x : A, y : B]}{E(c, d) \in C(c)}$$

which states that to define a function on *all* the elements of the type $A \times B$ it is sufficient to explain how it works on the canonical elements.

Equality. How to compute the function defined by the elimination rule.

Example:

$$\frac{a \in A \quad b \in B \quad d(x, y) \in C(\langle x, y\rangle)\ [x : A, y : B]}{E(\langle a, b\rangle, d) = d(a, b) \in C(\langle a, b\rangle)}$$

which states that to evaluate the function $E(c, d)$, defined by the elimination rule, one has first to evaluate c in order to obtain a canonical element $\langle a, b\rangle \in A \times B$ and then to use the method $d : (x : A)(y : B)\ C(\langle x, y\rangle)$, provided by the second premise of the elimination rule, to obtain the value $d(a, b) \in C(\langle a, b\rangle)$.

The same approach can be used to obtain the rules for a type which is not a *logical* proposition. Let us consider the case of the type $\mathcal{N}$. It is interesting to note that there is no need to change *anything* with respect to the general pattern in order to recover all the usual properties of natural numbers.

Formation:

$$\mathcal{N}\ set$$

Introduction:

$$0 \in \mathcal{N} \qquad \frac{a \in \mathcal{N}}{s(a) \in \mathcal{N}}$$

Elimination:

$$\frac{c \in \mathcal{N} \quad d \in C(0) \quad e(x,y) \in C(s(x))[x : \mathcal{N}, y : C(x)]}{R(c,d,e) \in C(c)}$$

Equality:

$$\frac{d \in C(0) \quad e(x,y) \in C(s(x)) \ [x : \mathcal{N}, y : C(x)]}{R(0,d,e) = d \in C(0)}$$
$$\frac{a \in \mathcal{N} \quad d \in C(0) \quad e(x,y) \in C(s(x))[x : \mathcal{N}, y : C(x)]}{R(s(a),d,e) = e(a, R(a,d,e)) \in C(s(a))}$$

As you see the elimination rule is an old friend: we have re-discovered the induction principle on natural numbers.

We can now consider a new kind of rules which makes explicit the computation process which is only implicit in the equality rule.

Computation:

$$\frac{c \Rightarrow 0 \quad d \Rightarrow g}{R(c,d,e) \Rightarrow g} \qquad \frac{c \Rightarrow s(a) \quad e(a, R(a,d,e)) \Rightarrow g}{R(c,d,e) \Rightarrow g}$$

6 Some programs

Now we can develop some simple programs on natural numbers and look at their execution.

6.1 The sum of natural numbers

Let $x, y \in \mathcal{N}$, then the sum of x and y is defined by means of the following deduction:

$$\frac{x \in \mathcal{N} \quad y \in \mathcal{N} \quad \dfrac{[v : \mathcal{N}]_1}{s(v) \in \mathcal{N}}}{x + y \equiv R(x, y, (u, v)\ s(v)) \in \mathcal{N}}\ 1$$

For instance we can evaluate $3 + 2$ as follows:

$$\frac{3 \Rightarrow s(2) \quad s(R(2,2,(u,v)\ s(v)) \Rightarrow s(2+2)}{3 + 2 \equiv R(3,2,(u,v)\ s(v)) \Rightarrow s(2+2)}$$

This simple example can already suggest that ITT is a functional programming language with a strong typing system.

6.2 The product of natural numbers

For any $x, y \in \mathcal{N}$, we define the product of x and y by means of the following deduction which makes use of the definition of the sum of the previous section.

$$\dfrac{\begin{array}{ccc} & & y:\mathcal{N} \quad [v:\mathcal{N}]_1 \\ & & \vdots \\ x \in \mathcal{N} & 0 \in \mathcal{N} & y+v \in \mathcal{N} \end{array}}{x * y \equiv R(x, 0, (u, v)\ y + v) \in \mathcal{N}}\ 1$$

In general, the recursive equation with unknown f:

$$\begin{cases} f(0) = k \in A \\ f(s(x)) = g(x, f(x)) \in A \end{cases}$$

is solved in ITT by

$$f(x) \equiv R(x, k, (u, v)\ g(u, v))$$

and it is possible to prove that

$$R(x, k, (u, v)\ g(u, v)) \in A\ [x : \mathcal{N}, k : A, g : (u : \mathcal{N})(v : A)\ A]$$

7 All types are similar

Looking at the rules for the types that we have introduced until now it is possible to realize that they always follow the same pattern. First the *formation* rules state how to form the new type. For instance in order to define the type $A \to B$ we put:

$$\dfrac{A\ type \quad B\ type}{A \to B\ type} \qquad \dfrac{A = C \quad B = D}{A \to B = C \to D}$$

The second step is the definition of the canonical elements of the type; this is the step which completely determines the type we are constructing since all the other rules directly depend on these ones. For instance, for the type $A \to B$ we state that its canonical elements are the functions $\lambda(b)$ such that $b(x) \in B\ [x : A]$.

$$\dfrac{b(x) \in B\ [x : A]}{\lambda(b) \in A \to B} \qquad \dfrac{b(x) = d(x) \in B\ [x : A]}{\lambda(b) = \lambda(d) \in A \to B}$$

The elimination rule is now determined; it states that the *only* elements of the type are those introduced by the introduction rule(s) and hence that one can define a function on all the elements of the type if (s)he knows how this function works on the canonical elements. Again let us use the type $A \to B$ as a paradigmatic example; for this type we have only one introduction rule and hence in the elimination rule, besides the premise $c \in A \to B$, we have to consider only another premise, i.e. $d(y) \in C(\lambda(y))\ [y : (x : A)\ B]$ which shows how to obtain a proof of $C(\lambda(y))$ starting from a generic assumptions for the introduction rule. Now $c \in A \to B$ must be equal to $\lambda(b)$ for some

$b(x) \in B\ [x : A]$ and hence by using c one has to be able to construct anything (s)he can construct starting from $b : (x : A)\ B$; since the second premise states that we can obtain a proof $d(b)$ of $C(\lambda(b))$ then the elimination rule states that we have to be able to obtain a proof of $C(c)$, which we call $F(c, d)$, in dependence of the two assumptions $c \in A \to B$ and $d : (y : (x : A)\ B)\ C(\lambda(y))$.

$$\frac{c \in A \to B \quad d(y) \in C(\lambda(y))\ [y : (x : A)\ B]}{F(c, d) \in C(c)}$$

Let us now consider the equality rule: it shows how to compute the function defined by the elimination rule. Since in the elimination rule we have considered a different premise in correspondence with each introduction rule, we will need also a particular equality rule in correspondence with each introduction rule. In fact, consider the evaluation process of a function obtained by an elimination rule for the type A: first the element of the type A which appears in the leftmost premise is evaluated into a canonical element of A in correspondence to a suitable introduction rule; then the premise(s) of this introduction rule can be substituted for the assumption(s) of the corresponding premise of the elimination rule. Let us consider the case of the type $A \to B$: we have only one introduction rule and hence we have to define one elimination rule which explains how to evaluate a function $F(z, d) \in C(z)\ [z : A \to B]$ when it is used on the canonical element $\lambda(b)$; the second premise of the elimination rule states that we obtain a proof $d(b)$ of $C(\lambda(b))$ simply by substituting b for y and hence we put $F(\lambda(b), d) = d(b)$.

$$\frac{b(x) \in B\ [x : A] \quad d(y) \in C(\lambda(y))\ [y : (x : A)\ B]}{F(\lambda(b), d) = d(b) \in C(\lambda(b))}$$

To show now a situation which is a little different, let us analyze the elimination and the equality rules for the type $\mathcal{N}$. In this case we have two introduction rules and hence in the elimination rule, besides the premise $c \in \mathcal{N}$, there will be two other premises. The first, in correspondence with the introduction rule $0 \in \mathcal{N}$ which has no premise, has no assumption and hence, supposing we are proving an instance of the propositional function $C(z)\ prop\ [z : \mathcal{N}]$, it must be a proof d of the proposition $C(0)$. The second premise is more complex here that in the previous case since the second introduction rule for the type $\mathcal{N}$ is inductive, i.e. the premise contains the type $\mathcal{N}$ itself. In this case we can suppose to have proved the property $C(x)$ for the natural number x before we construct the natural number $s(x)$ hence in the elimination rule, when we are going to prove $C(s(x))$, besides the assumption due to the premise of the introduction rule, we can also assume to have a proof of $C(x)$.

Since we have two introduction rules we also have two equality rules. The first concerns $R(c, d, e)$, in correspondence with the first introduction rule, i.e. when the value of c is 0, and in this case the second assumption shows that $R(0, d, e) = d$. The second equality rule concerns the case the value of c is $s(a)$ for some natural number $a \in \mathcal{N}$: in this case we can suppose to know that

$R(a,d,e) \in C(a)$ and hence the third assumption shows that $R(s(a),d,e) = e(a,R(a,d,e))$.

We can now play a little game and see how these formal considerations work even if one has no a priori comprehension of the type (s)he is defining. Let us suppose to have the following formation and introduction rules:

Formation:

$$\frac{A\ type \quad B(x)\ type\ [x:A]}{W(A,B)\ type}$$

Introduction:

$$* \in W(A,B) \qquad \frac{a \in A \quad b(y) \in W(A,B)\ [y:B(a)]}{\circ(a,b) \in W(A,B)}$$

Can you find the correct elimination and equality rules?

The elimination rule is completely determined by the introduction rules: there are two introduction rules and hence, besides the premise $c \in W(A,B)$, there would be two minor premises. The first will have no assumption because the first introduction rule has no premise, while the second premise will have an assumption in correspondence with any premise of the second introduction rule plus an inductive assumption since this introduction rule is inductive.

Elimination:

$$\frac{\begin{array}{c}[x:A, w:(y:B(x))W(A,B), z:(y:B(x))C(w(y))]_1 \\ \vdots \\ c \in W(A,B) \quad d \in C(*) \qquad e(x,w,z) \in C(\circ(x,w))\end{array}}{\square(c,d,e) \in C(c)}\ 1$$

Finally we have to define two equality rules in correspondence with the two introduction rules.

Equality:

$$\frac{\begin{array}{c}[x:A, w:(y:B(x))W(A,B), z:(y:B(x))C(w(y))]_1 \\ \vdots \\ d \in C(*) \qquad e(x,w,z) \in C(\circ(x,w))\end{array}}{\square(*,d,e) = d \in C(*)}\ 1$$

$$\frac{\begin{array}{c}a \in A \quad b(y) \in W(A,B)[y:B(a)] \quad d \in C(*) \quad e(x,w,z) \in C(\circ(x,w)) \\ \vdots \\ [x:A, w:(y:B(x))W(A,B), z:(y:B(x))C(w(y))]_1\end{array}}{\square(\circ(a,b),d,e) = e(a,b,(y)\ \square(b(y),d,e)) \in C(\circ(a,b))}\ 1$$

Can you guess what this type is? It is the type of the labeled tree: the element $* \in W(A,B)$ is a leaf and $\circ(a,b) \in W(A,B)$ is a node which has the label $a \in A$ and a branch which arrives at the tree $b(y) \in W(A,B)$ in correspondence with each element $y \in B(a)$ (see figure 1).

Figure 1: the tree $\circ(a, b)$

8 Some applications

To understand the expressiveness of ITT as a programming language, let us show some simple programs: the first is the description of a computer memory and the second one is a Turing machine.

8.1 A computer memory

To describe a computer memory it is convenient to introduce the type *Boole* of the boolean values.

Formation:

$$Boole\ set$$

Introduction:

$$\top \in Boole \qquad \bot \in Boole$$

Elimination:

$$\frac{c \in Boole \quad d \in C(\top) \quad e \in C(\bot)}{\text{if } c \text{ then } d \text{ else } e \text{ endif} \in C(c)}$$

Equality:

$$\frac{d \in C(\top) \quad e \in C(\bot)}{\text{if } \top \text{ then } d \text{ else } e \text{ endif} = d \in C(\top)} \qquad \frac{d \in C(\top) \quad e \in C(\bot)}{\text{if } \bot \text{ then } d \text{ else } e \text{ endif} = e \in C(\bot)}$$

Then we obtain the type $Value$ of a m-bits memory word by using m times the cartesian product of *Boole* with itself.

$$Value \equiv Boole^m \equiv \underbrace{Boole \times \ldots \times Boole}_{m}$$

By using the same construction we can define also the address space by putting

$$Address \equiv Boole^n$$

if we want to specify a computer with an n-bits address bus.

The definition of a computer memory is now straightforward: a memory is a map which returns a value in correspondence with any address.

$$mem \in Address \to Value$$

The first operation one have to define on a computer memory is its initialization at startup; the following position is one of the possible solution of the problem of memory initialization:

$$mem \equiv \lambda((x)\ \langle \bot, \ldots, \bot \rangle)$$

where $\langle a_1, \ldots, a_n \rangle$ is the obvious generalization of the operation of couple formation we used for the cartesian product.

We can now define the two standard operations on a computer memory: reading and writing. Suppose $add \in Address$ and $mem \in Address \to Value$, then the function $read(mem, add)$, which returns the value at the memory location add of the memory mem, consists simply in applying the function mem to the address add:

$$read(mem, add) \equiv mem[add] \equiv F(mem, (y)\ y(add))$$

As regard to the function $write(mem, add, value)$, which returns the new memory configuration after writing the value val at the location add of the memory mem, we put:

$$write(mem, add, val) \equiv \lambda((x)\ \text{if } x = add \text{ then } val \text{ else } read(mem, x) \text{ endif})$$

It is obvious that the computer memory we proposed here is not very realistic, but we can improve it a bit if we introduce the type $extBoole$.

Formation:

$$extBoole\ set$$

Introduction:

$$\top \in extBoole \quad \bot \in extBoole \quad \omega \in extBoole$$

where the new canonical element $\omega \in extBoole$ can be used to specify the presence of a value still undefined in the memory.

Elimination:

$$\frac{c \in extBoole \quad d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } c \text{ of } \top : d_1; \bot : d_2; \omega : d_3 \text{ endcase} \in C(c)}$$

Equality:

$$\frac{d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } \top \text{ of } \top : d_1; \bot : d_2; \omega : d_3 \text{ endcase} = d_1 \in C(\top)}$$

$$\frac{d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } \bot \text{ of } \top : d_1; \bot : d_2; \omega : d_3 \text{ endcase} = d_2 \in C(\bot)}$$

$$\frac{d_1 \in C(\top) \quad d_2 \in C(\bot) \quad d_3 \in C(\omega)}{\text{case } \omega \text{ of } \top : d_1; \bot : d_2; \omega : d_3 \text{ endcase} = d_3 \in C(\omega)}$$

In this way we can improve the initialization function by putting

$$mem \equiv \lambda((x)\ \langle \omega, \ldots, \omega \rangle)$$

which states that at startup the state of the memory is completely unknown. In this way it becomes clear that we must write a *real* value at a particular address before we can read something meaningful at that address.

8.2 A Turing machine on a two symbols alphabet

Our second example is the definition of a Turing machine which uses a two symbols alphabet. [1]

Let us begin with some definitions which makes clearer what follows. The first definition concerns the movements of the head.

$$\begin{aligned} Move &\equiv Boole \\ Left &\equiv \top \\ Right &\equiv \bot \end{aligned}$$

Then, as regard the alphabet we put

$$\begin{aligned} Symb &\equiv Boole \\ 0 &\equiv \top \\ 1 &\equiv \bot \end{aligned}$$

Finally we define our Turing machine. As usual a Turing machine is made of a *tape* where a *head* can read and write under the control of a program whose instructions are scanned by a program counter which indicates the execution *state*. Thus we can formally identify a Turing machine with a quadruple

$$TM \equiv \langle prog, tape, head, state \rangle$$

where, supposing

$$\begin{aligned} TapeType &\equiv \mathcal{N} \to Symb \\ HeadPos &\equiv \mathcal{N} \\ States &\equiv \mathcal{N} \\ ProgType &\equiv (States \times Symb) \to (Symb \times Move \times States) \end{aligned}$$

we have

$$tape \in TapeType$$

[1] It is well known that the use of such a simple alphabet does not limit the expressive power of the machine; here we prefer to limit the alphabet to avoid to introduce new types.

i.e. the tape is a function which gives the symbol at the n-th location of the tape if applied to the natural number n; [2]

$$head \in HeadPos$$

i.e. *head* is the current position of the head over the tape;

$$state \in States$$

i.e. *state* is the current execution state of the machine, finally

$$prog \in ProgType$$

i.e. *prog* is the transition function of the machine which, given the actual state and the symbol at the current position of the head over the tape, returns the symbol to write at the current position, the next movement of the head and the new state of the machine.

We can now describe the execution process of the Turing machine. Of course we have to start by considering the initial conditions, i.e. the initial configuration of the tape, the initial position of the head and the initial state of the machine. For instance we put

$$\begin{array}{lll} initTape & \equiv \lambda x. \text{ if } x = 0 \text{ then } 1 \\ & \qquad\qquad \text{else if } x = 1 \text{ then } 1 \\ & \qquad\qquad\quad \vdots \\ & \qquad\qquad \text{else if } x = n \text{ then } 1 \\ & \qquad\qquad \text{else } 0 \text{ endifs} \in TapeType \\ initHead & \equiv 0 \in HeadPos \\ initState & \equiv 0 \in States \end{array}$$

to state that the machine works on the tape $11\ldots1000\ldots$, which contains the symbol 1 in its first n locations and 0 in all the other ones, and starts in the execution state 0 with its head over the location 0.

Let us now show how a computation step can be described. Let *prog* be a program and suppose to have an actual description of the machine determined by the actual tape, the actual position of the head and the actual state; then we want to obtain the new configuration of the tape, the new position of the head and the new state of the machine which result after the execution of one computation step. Hence we need a function

$$\begin{array}{lll} execStep \in ProgType \to (& TapeType \times HeadPos \times States \\ & \to TapeType \times HeadPos \times States) \end{array}$$

such that

$$execStep[prog][\langle inTape, inHead, inState\rangle] = \langle outTape, outHead, outState\rangle$$

[2] In order to use the type of the natural numbers which we have already defined, we consider here Turing machines whose tape is non-finite only at the right instead of the more standard machines whose tape has both sides not finite; it is well-known that this requirement does not change the class of the computable functions.

To define *execStep* it is convenient to put

$$\Pi_1(c) \equiv E(c, (x, y)\ x) \qquad \Pi_2(c) \equiv E(c, (x, y)\ y)$$

so that

$$\Pi_1(\langle a, b\rangle) = a \qquad \Pi_2(\langle a, b\rangle) = b$$

and hence

$$\Pi^3_1(x) \equiv \Pi_1(\Pi_1(x)) \qquad \Pi^3_2(x) \equiv \Pi_2(\Pi_1(x)) \qquad \Pi^3_3(x) \equiv \Pi_2(x)$$

In fact we can now define $outTape$, $outHead$, $outState$ as a function of $inTape$, $inHead$ and $inState$, besides the program $prog$, as follows:

$$\begin{array}{rl} outTape & \equiv \lambda x.\ \text{if } x = inHead \\ & \quad \text{then } \Pi^3_1(prog[inState, inTape[inHead]]) \\ & \quad \text{else } inTape[x] \text{ endif} \\ outHead & \equiv \text{if } \Pi^3_2(prog[inState, inTape[inHead]]) \\ & \quad \text{then } inHead - 1 \text{ else } inHead + 1 \text{ endif} \\ outState & \equiv \Pi^3_3(prog[inState, inTape[inHead]]) \end{array}$$

Hence we can define the function *execStep* by putting

$$execStep \equiv \lambda prog.\lambda\langle inTape, inHead, inState\rangle.\langle outTape, outHead, outState\rangle$$

In order to define the execution of a Turing machine we only have to apply the function *execStep* for an arbitrary number of times. To this aim it is useful to define the function *iterate* such that, supposing $a \in A$ and $f \in A \to A$,

$$\begin{cases} iterate(0, f, a) = a \in A \\ iterate(s(x), f, a) = f[iterate(x, f, a)] \in A \end{cases}$$

i.e. $iterate(n, f, a) \equiv f^n(a)$. To solve this recursive equation within ITT, we put

$$iterate(n, f, a) \equiv It(n)[f, a]$$

so that

$$\begin{cases} It(0) = \lambda f.\lambda a.a \\ It(s(x)) = \lambda f.\lambda a.f[It(x)[f, a]] \end{cases}$$

which is solved by

$$It(n) \equiv R(n, \lambda f.\lambda a.a, (u, v)\ \lambda f.\lambda a.f[v[f, a]])$$

Now we can use the function *iterate* to obtain

$$(execStep[prog])^n[\langle initTape, initHead, initState\rangle]$$

i.e. we can calculate the status of the Turing machine after any finite number of steps, but ...

9 Some meta-mathematical results on ITT

In this section we will recall some of the basic results on ITT [BV92]: in particular we will see that it is not possible to apply the function *iterate* we defined in the previous paragraph a non-finite number of times to simulate the execution of a non-terminating computation.

The first theorem we state show that ITT is a theory we can trust on.

Theorem 9.1 (Consistency) *The Intuitionistic Theory of Types is consistent.*

The proof of this theorem, as usual in proof theory [P65], is based on the following theorem of normal form.

Theorem 9.2 (Normalization) *Any provable judgment has a canonical proof.*

Since in ITT any proof of the truth of a proposition is coded by an element which belongs to that proposition, the normalization theorem has also another interesting corollary about the execution of all the programs which have a type.

Theorem 9.3 (Computability) *Let $a \in A$ be a derivable judgment. Then the valuation of the program a terminates; hence any program partially corrected, i.e. well-typed, is totally corrected.*

This is a strong result on a general property of the programs to which one can assign a type within ITT, but it is meanwhile a strong limitation to the number of programs one can write within ITT. In particular it shows that it is not possible to represent all the recursive functions, and hence that there is no possibility to simulate the execution of a general Turing machine. Anyhow one can prove the following theorem.

Theorem 9.4 *A function (from $\mathcal{N}$ to $\mathcal{N}$) has a type in the Intuitionistic Type Theory if and only if it is provably total in first order Peano arithmetic.*

Thus the class of functions that one can describe is surely not too small!

A The multi-level typed λ-calculus

This appendix contains some considerations on the well-writing theory we need in order to present ITT. We will not try to give a detailed description here but the willing reader could refer to [V95].

The basic idea is to use a kind of simple typed λ-calculus since this calculus allows to abstract on variables while maintaining a decidable equality theory, which is essential in order to describe any logical system since one needs to recognize the correct application of the rules. On the other hand in order to describe ITT a simple typed λ-calculus, as in [B84], is not sufficient. This is the reason way we define the following multi-level typed λ-calculus; the basic idea

for its definition is to construct a *tower* of simple typed λ-calculi, each one over another, marked by a *level*. Hence the rules for the multi-level typed λ-calculus are those of a simple typed λ-calculus enriched by a label which specifies the level.

$$\text{assumption} \quad \frac{\Gamma \vdash N :_i M}{\Gamma, x :_{i-1} N \vdash x :_{i-1} N} \quad i \geq 1$$

$$\text{weakening} \quad \frac{\Gamma \vdash N :_i M \quad \Gamma \vdash K :_j L}{\Gamma, x :_{i-1} N \vdash K :_j L} \quad i \geq 1$$

$$\text{abstraction} \quad \frac{\Gamma, x :_j N \vdash K :_i L}{\Gamma \vdash ((x :_j N)K) :_i ((x :_j N)L)}$$

$$\text{application} \quad \frac{\Gamma \vdash N :_i ((x :_j L)M) \qquad \Gamma \vdash K :_j L}{\Gamma \vdash N(K) :_i M[x := K]}$$

These rules by themselves are almost useless since no expression can be assigned a type because to prove the conclusion of a rule one should have already proved the premise(s). So in order to start we need some axiom. The first thing one has to do is to settle the maximum level (s)he wants to use when describing a particular theory; to this aim we will use the symbol $*$ to indicate the only type of the highest level. We can then define all the other types downward from $*$. In the case of ITT, the basic idea is to define a chain

$$a :_0 A :_1 \mathit{type} :_2 *$$

to mean that a is an element of A which is a type, i.e. an element of *type*, which is the only element of $*$. Thus our first axiom is:

$$\vdash \mathit{type} :_2 *$$

We can now begin our description of ITT; to this aim we will follow the explanation we already used during the informal introduction of the rules in the previous sections. We start by stating an axiom which introduces a constant for each type constructor in correspondence with each formation rule. For instance suppose we want to describe the type $A \to B$; to this aim we will add the axiom

$$\vdash \to :_1 (X :_1 \mathit{type})(Y :_1 \mathit{type})\ \mathit{type}.$$

which means that $\to$ is a type-constructor constant which gives a new type when applied to the types X and Y.

The next step corresponds to the introduction rules: we will add a new axiom for each kind of canonical element. Let us consider again the case of the type $A \to B$; then we put

$$\vdash \lambda :_0 (X : \mathit{type})(Y : \mathit{type})(y : (x : X)\ Y)\ X \to Y.$$

which states that, supposing X and Y be two types and y be a function from X to Y, $\lambda(X, Y, y)$ is an element of the type $X \to Y$.

Also the elimination rule becomes a new axiom; it introduces the constant we used in the elimination rule. For instance for the type $A \to B$ we put

$$\begin{array}{ll} \vdash F :_0 & (X : \mathit{type})(Y : \mathit{type})(Z : (z : X \to Y)\ \mathit{type}) \\ & (c : X \to Y)(d : (y : (x : X)Y)Z(\lambda(X, Y, y))) \\ & Z(c) \end{array}$$

which states that, supposing X and Y are two types, Z is a propositional function on elements of $X \to Y$, c is an element of $X \to Y$ and d is a method which maps any function y from X to Y into an element of $Z(\lambda(X, Y, y))$, $F(X, Y, c, d)$ is an element of $Z(c)$. In this way any rule of ITT becomes an axiom of the multi-level typed λ-calculus.

By means of the rules we have considered so far we can not deal with the equality rules of ITT. To deal also with them let us add some equality rules to our multi-level typed λ-calculus. We can start from the equality rules for the simple typed λ-calculus and enrich them by specifying the levels.

var-equality
$$\frac{\Gamma \vdash x :_i N}{\Gamma \vdash x = x :_i N}$$

const-equality
$$\frac{\Gamma \vdash c :_i N}{\Gamma \vdash c = c :_i N}$$

app-equality
$$\frac{\Gamma \vdash K_1 = K_2 :_i ((x :_j N)M) \quad \Gamma \vdash L_1 = L_2 :_j N}{\Gamma \vdash K_1(L_1) = K_2(L_2) :_i M[x := K_1]}$$

ξ-equality
$$\frac{\Gamma, x :_j N \vdash K = L :_i M}{\Gamma \vdash ((x :_j N)K) = ((x :_j N)L) :_i ((x :_j N)M)}$$

α-equality
$$\frac{\Gamma, y :_j N \vdash K :_i M}{\Gamma \vdash ((y :_j N)K) = ((x :_j N)K[y := x]) :_i ((y :_j N)M)}$$

β-equality
$$\frac{\Gamma, x :_j N \vdash K :_i M \quad \Gamma \vdash L :_j N}{\Gamma \vdash ((x :_j N)K)(L) = K[x := L] :_i M[x := L]}$$

η-equality
$$\frac{\Gamma \vdash K :_i ((x :_j N)M)}{\Gamma \vdash ((x :_j N)K(x)) = K :_i ((x :_j N)M)}$$

where in the α-equality rule and in the η-equality rule we assume that x does not appear in K.

Using these rules we will obtain all the rules of ITT which state that the type-constructor constants and the element-constructor constants are functions. On the other hand to deal with the equality rules characteristic of ITT we have to add specific axioms. For instance we add

$$X : type, Y : type, b : (x : X)\ Y, d : (y : (x : X)\ Y)Z(\lambda(X, Y, y)) \vdash F(\lambda(X, Y, b), d) = d(b) : Z(\lambda(X, Y, b))$$

if we want to deal with the equality rule for the type $A \to B$.

We can show that the multi-level typed λ-calculus satisfies all our requirements on a well-writing theory [V95]. Let us recall that the β-reduction relation between expressions is obtained extending in the obvious way the relation defined by

$$((x : N)\ K)(M) \Rightarrow K[x := M]$$

Then, provided we say that an expression is in *normal form* if it contains no sub-expression which can be β-reduced, we can prove the following theorem.

Theorem A.1 (Normalization) *Any expression can be reduced into an equivalent one in normal form.*

One of the consequences of this result is a theorem on the decidability of the equality between expressions. In fact, supposing to consider only the general rules on equality, we can prove the following theorem.

Theorem A.2 (Decidability of equality) *Given two expressions M and N of the same type, one can effectively decide if M and N are equal.*

References

[BCMS89] R. Backhouse, P. Chisholm, G. Malcom, E. Saaman, *Do-it-yourself type theory*, Formal Aspects of Computing 1, 1989, pp.19-84.

[B84] H.P. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, North-Holland, Amsterdam, 1984.

[BV92] A. Bossi, S. Valentini, *An intuitionistic theory of types with assumptions of high-arity variables*, Annals of Pure and Applied Logic 57, North Holland, 1992, pp.93-149.

[H56] A. Heyting, *Intuitionism, an introduction*, North-Holland, Amsterdam, 1956.

[H80] W.A. Howard, *The formula-as-types notion of construction*, To H.B. Curry: Essays on combinatory Logic, Lambda Calculus and Formalism (R. Hindley and J.P. Seldin, eds.), Academic Press, London, 1980, pp. 479-490.

[K32] A.N. Kolmogorov, *Zur Deutung der intuitionistischen Logik*, Mathematische Zeitschrift, vol. 35, 1932, pp. 58-65.

[ML84] P. Martin-Löf, "Intuitionistic type theory", notes by Giovanni Sambin of a series of lectures given in Padua, June 1980, Bibliopolis, Napoli, 1984.

[NPS90] B. Nordstrom, K. Petersson, J.M. Smith, "Programming in Martin-Löf's Type Theory, an introduction", Oxford Univ. Press, Oxford, 1990.

[PS86] K. Petersson, J.M. Smith, *Program derivation in type theory: a partitioning problem*, Comput. Languages 11 (3/4), 1986, pp.161-172.

[P65] D. Prawitz, *Natural Deduction*, Almquist and Wiksell, Stockolm, 1965.

[SV95] G. Sambin, S. Valentini, *Building up a tool-box for Martin-Löf intuitionistic type theory*, in preparation.

[V95] S. Valentini, *The multi-level typed λ-calculus*, in preparation.