In the Search of a Naive Type Theory*

Agnieszka Kozubek and Paweł Urzyczyn

Institute of Informatics,
University of Warsaw, Poland
{kozubek,urzy}@mimuw.edu.pl

Abstract. This paper consists of two parts. In the first part we argue that an appropriate "naive type theory" should replace naive set theory (as understood in Halmos' book) in everyday mathematical practice, especially in teaching mathematics to Computer Science students. In the second part we make the first step towards developing such a theory: we discuss a certain pure type system with powerset types. While the system only covers very initial aspects of the intended theory, we believe it can be used as an initial formalism to be further developed. The consistency of this basic system is established by proving strong normalization.

1 Why Not Set Theory?

Set theory is an enormous success in the contemporary mathematics, including the mathematics relevant to Computer Science. Virtually all maths is developed within the framework of set theory, and virtually all books and papers are written under the silent assumption of ZF or ZFC axioms occurring "behind the back". We sometimes feel as if we actually lived in set theory, as if it was the only true and real world.

The set-theoretical background has made its way to education, from the university to the kindergarten level, and what once was a foundational subject on the border of logic and philosophy now has become a part of elementary mathematics.

And indeed, set theory deserves its pride. From an extremely modest background—the notion of "being an element" and the idea of equality—it develops complex notions and objects serving the needs of even most demanding researcher. Enjoying the paradise of sets we tend to forget about the price we pay for that.

Of course, we must avoid paradoxes, and thus the set formation patterns are severely restricted. We must give up Cantor's idea of "putting together" any collection of objects, resigning therefore, at least partly, from the very basic intuition that a set of objects can be selected by any criterion at all.

Universes *vs* **predicates.** In fact, there are *two* very basic intuitions that are glued together into the notion of a "set":

* Partly supported by the Polish Government Grant 3 T11C 002 27, and by the EU Coordination Action 510996 "Types for Proofs and Programs".

M. Miculan, I. Scagnetto, and F. Honsell (Eds.): TYPES 2007, LNCS 4941, pp. 110–124, 2008.

– Set as a domain or universe;
– Set as a result of selection.

We used to treat this identification as natural and obvious. But perhaps only because we were *taught* to do so. These two ideas are in fact different, and this very confusion is responsible for Russel's paradox. In addition, ordinary mathematical practice often makes an explicit difference between the two aspects. Mathematicians have been classifying objects according to their domain, kind, sort or *type* since the antiquity [2,21]. An empty set of numbers and an empty set of apples are intuitively *not* the same, as well as in most cases we do not need and do not want to treat a function in the same way as its arguments.

The difference between domains (types) and predicates is made explicit in type theory. This results in various simplifications. For instance, the difference between operations on universes (product, disjoint sum) and operations on predicates (intersection, set union) becomes immediately apparent and natural. Yet another example is that a union $\bigcup A$ of a family A of sets is typically of the same "type" as members of A rather than as A itself. In set theory, this argument is not sufficient to reject common student's misconceptions like e.g. $A \subseteq \bigcup A$, because classifying sets (a priori) into types is illegal.

Everyday maths *vs* foundations of mathematics. The purpose of set theory was to give a universal foundation for a consistent mathematics. That happened at the beginning of the 20th century, when consistency of elementary notions was a serious issue, threatened by the danger of antinomies, and when modern formal mathematics was in its infancy. It was then important to ensure as much security as possible. Therefore, all the development had to be done from first principles, and the results of it have little to do with ordinary mathematical practice. For instance, using the Axiom of Foundation one derives in set theory the surrealistic conclusion that all the world is built from curly braces.

This foundational tool is now being widely used for a quite different purpose. We use sets as a basic everyday framework for various kinds of mathematics, and we teach set theory to students, beginning at a very elementary level. But that puts us into an awkward situation. On the one hand, we want to use as much common-sense as possible, on the other hand we do not want paradoxes and inconsistency. So if we do not want to cheat, what can we do? One possibility is to hide the problem and pretend that everything is OK: *"Emmm... We assume that all sets under consideration are subsets of a certain large set."* This is what often happens in elementary and high-school textbooks. But is it really different than saying that the world is placed on the back of a giant turtle? An intelligent student must eventually ask on whose back the turtle is standing. And then all we can say is "Sit up straight!"

The other option is to pull the skeleton out of the closet, put all axioms on the table, and pay a heavy overhead by spending a lot of effort on constructing ordered pairs in the style of Kuratowski, integers in the style of von Neumann, and so on. This approach is common at the university level and has been considerably mastered. For half a century, the book [18] by Halmos has been giving guidance to lecturers how to achieve a balance between precision and simplicity.

(Contrary to its title, the book is not about naive set theory. It is about axiomatic set theory taught in a "naive" or "common-sense" style.) But even this didactic masterpiece is a certain compromise.

The idea *vs* the implementation. This is because the overhead is unavoidable. Very basic mathematical ideas must be encoded in set theory before they can be used, and a substantial part of student's attention is paid to the details of the encoding. To a large extent this is a wasted effort and it would be certainly more efficient to concentrate on "top-level" issues. Using an old comparison in a different context, it is like teaching the details of fuel injection in a driving school while we should rather let students practice driving. Getting accustomed to set theoretical "implementation" of mathematics is painful to many students. In set theory the implementation is not "encapsulated" at all and we can smell the fuel in the passenger's cabin. One of the most fundamental God's creations is turned into a transitive set of von Neumann's numbers so we must live with phenomena like $1 \in 2 \subseteq 3 \in 4$ or $\mathbb{N} = \bigcup \mathbb{N}$.

We do not really need these phenomena. The actual use of various objects and notions in mathematics is based on their intensional "specification" rather than formal implementation. We still have to ask students to remember the rule

$$\langle\, a, b\,\rangle = \langle\, c, d\,\rangle \qquad \text{iff} \qquad a = c \wedge b = d, \tag{*}$$

in addition to the definition $\langle\, a, b\,\rangle = \{\{a\}, \{a, b\}\}$. But we must spend time on proving the above equivalence. A doubtful reward is the malicious homework "Prove that $\bigcup(\mathbb{N} \times \mathbb{N}) = \mathbb{N}$." We got used to such homeworks so much that we do not notice that they are nonsense. In a typed framework a substantial part of this nonsense simply disappears.

2 Why Type Theory and What Type Theory?

We believe that an appropriate type theory should give a chance to build a framework for "naive" mathematics that would not exhibit many of the drawbacks mentioned above. In particular, it is reasonable to expect that a "naive type theory" can be more adequate than "naive set theory" from our point of view, in that it should

- be free from both paradoxes and unnecessary artificial formalization;
- distinguish between domains (universes) and sets understood as predicates;
- begin with intensional specifications rather than from bare first principles;
- be closer to the everyday maths and computer science practice;
- be more appropriate for automatic verification.

We do not want to depart from ordinary mathematical practice, and thus our naive type theory should be adequate for classical reasoning, and extensional with respect to functions and predicates. We find it however methodologically appropriate that these choices are made explicit (introduced by appropriate axioms) rather than implicit (built in the design principles). We also would like to include a Curry-Howard flavour, taking seriously De Bruijn's slogan [6]:

> *Treating propositions as types is definitely not in the way of thinking of the ordinary mathematician, yet it is very close to what he actually does.*

The basic idea is of course to separate the two roles played by sets, namely to put apart domains (types) and predicates (selection criteria for objects of a given type). Thus for any type A we need a powerset type $P(A)$, identified with the function space $A \to *$, where $*$ is the sort of propositions. That is, we would like to treat "$M \in \{a : A \mid \varphi(a)\}$" as syntactic sugar for "$\varphi(M)$".

Although our principal aim is a "naive" approach, we should be aware of the necessity of a formalization. Firstly, because we still need some justification for consistency, secondly, because it may be desirable that "naive" reasoning can be computer-assisted. We find it quite natural and straightforward to build such a formalization beginning with a certain pure type system, to be later extended with additional constructs and axioms.

Related systems. *Simple type theory:* In Church's simple type theory [8,21] there are two base types: the type $\mathbf{i}$ of individuals and the type $\mathbf{b}$ of truth values. Expressions have types and formulas are simply expressions of type $\mathbf{b}$. There is no built-in notion of a proof and formulas are not types. In addition to lambda-abstraction, there is another binding operator that can be used to build expressions, namely the *definite description* $\iota x.\, \varphi(x)$, meaning "the only object x that satisfies $\varphi(x)$". While various forms of definite description are often used in the informal language of mathematics, the construct does not occur in most contemporary logical systems. As argued by William Farmer in a series of papers [11,12,13,14], simple type theory could be efficiently used in mathematical practice and teaching. Also the textbook [2] by P.B. Andrews develops a version of simple type theory as a basis for everyday mathematics. This is very much in line with our way of thinking. We choose a slightly different approach, mostly to avoid the inherently two-valued Boolean logic built in Church's type theory.

Quine's New Foundations: Quine's type theory [20,23] is based on an implicit *linear* hierarchy of universes. Full comprehension is possible at each level, but a set always lives at a higher floor than its elements. The idea of a linear hierarchy is of course convenient from a foundational point of view, but is not very intuitive. Also implementing "ordinary" mathematics requires a similar effort as in the usual set theory. The restriction to stratified constructs does not help either: one encounters difficulties when trying to define functions between objects belonging to different levels of the hierarchy.

Constable's computational naive type theory: We have to admit that the title of Halmos' book has already been rephrased by R. Constable [9]. But Constable's idea of a "naive type theory" is quite different than ours. It is inspired by Martin-Löf's theory and based on the idea of a *setoid* type, determined by a domain of objects plus an appropriate notion of equality. (In other words, quotient becomes a basic notion.) For instance, the field $\mathbb{Z}_3$ has the same domain as the set of integers $\mathbb{Z}$, but a different equality. And $\mathbb{Z}_6$ is defined by taking an "intersection" of $\mathbb{Z}_2$ and $\mathbb{Z}_3$. This is very convenient and natural way of dealing with quotient constructions. However (even putting aside the little counterintuitivity of the "contravariant" intersection) we still believe that a "naive" notion of equality should be more strict: two objects should not be considered the same in one context but different in another.

Coq and the calculus of inductive constructions: An almost adequate framework for a naive type theory is the Calculus of Constructions extended with inductive types. This is essentially the basic part of the type theory of the Coq proof assistant [5]. The paper [7] describes an attempt to use Coq in teaching rudiments of set theory. But in Coq, if A is a type (A : Set is provable) then the powerset $A \to$ Prop of A is a kind ($A \to$ Prop : Type is provable). That is, a set and its powerset do not live in the same sort, although they should receive similar treatment.

Weyl's predicative mathematics and Luo's logic-enriched theories: Zhaohui Luo in [22] considers „logic-enriched type theories" where the logical aspect is separated by design from the data-type aspect (in particular a separate kind $Prf(P)$ is used for proofs of any proposition P). Within that framework one can introduce both predicative and impredicative notion of a set, so that the kind *Type* is closed under the powerset construction. This approach is used by Adams and Luo [1] to formalize the predicative mathematics of Weyl [25], who long ago made an explicit distinction between "categories" and sets, understood respectively as universes and predicates. Weyl's theory is strictly predicative, and this certainly departs from our "naive" understanding of sets, but the impredicative version mentioned in [22] is very much consistent with it.

In this paper. In the next section we collect a few postulates concerning the possible exposition of a naive type theory. With this exposition we would like to initiate a discussion to help establish a new approach to both teaching and using mathematics in a way that will avoid the set-theoretic "overheads" and remain sufficiently precise and paradox-free.

We realize that a naive approach to type theory can result in an inconsistency, as it happened to naive set theory and many other ideas. Therefore we consider it necessary to build the naive approach on top of a rigourous formal system, to be developed in parallel. The relation between the formal language and the naive theory should be similar to the relation between the first-order ZFC formal theory, and Halmos' book [18].

In the present paper we do not attempt to solve the problem in general but rather to formulate it explicitly and highlight its importance. On the technical side, we only address here one very initial but important problem. A set X and its powerset $P(X)$ should be objects of the same sort, and we also assume that subsets of X should be identified with predicates on X. In the language of pure type systems that leads to the idea of a type assignment of the form $X{\to}* : *$, which turns out to imply inconsistency. In Section 4 we show that this inconsistency can be eliminated if the difference between propositions and types is made explicit. More precisely, we prove strong normalization (and thus consistency) of an appropriate PTS.

Of course, that is only the first step. We need a much richer consistent system to back up our "practical" exposition of sets, functions, and composite types, as sketched in the previous section. This will most likely require extending our system LNTT by various additional constructs, in particular a general scheme for inductive types, and additional axioms. All this is future work.

3 Informal Exposition

In this section we sketch some basic ideas of how a "naive" informal presentation of basic mathematics could look when set theory is replaced by type theory. As we said, these ideas go far beyond the initial formalism of Section 4.

Types. Every object is assigned a *type*. Types themselves are not objects.[1] Certain types are postulated by axioms, and many of these should be special cases of a general scheme for introducing inductive (perhaps also co-inductive) types. In particular, the following should be assumed:

- A unit type with a single element *nil*.
- Product types $A \times B$ and co-product types $A + B$, for any types A, B.
- The type $\mathbb{N}$ of integers.
- The powerset type $P(A)$, for any type A.
- Function types $A \to B$, perhaps as a special case of a more general product.

In particular, a powerset $P(A)$ of a type A should form a type and not a kind (i.e. it should be in the same sort as A) so that operations on types can be applied equally to both. Otherwise the classification of compound objects becomes unreasonably complicated: just think of a product $A \times P(A) \times (A \to P(A))$. Types come together with their constructors, eliminators etc., their properties postulated by axioms. For instance, the equivalence (*) should be an axiom.

Equality. In the "ordinary" mathematics two objects are equal iff they are the same object; one can do the same in the typed framework. As in common mathematical practice, equality between sets, functions, etc. should be extensional. In the formal model Leibniz's equality should probably be an axiom.

Sets. A predicate $\varphi(x)$, where $x : A$, is identified with a *subset* $\{x : A \mid \varphi(x)\}$ of type A. Subsets are assumed to be extensional, i.e.,

$$\varphi = \varphi' \qquad \text{iff} \qquad \forall x{:}A.\, \varphi(x) \leftrightarrow \varphi'(x).$$

Inclusion is defined as usual by $\varphi \subseteq \varphi'$ iff $\forall x{:}A.\, \varphi(x) \to \varphi'(x)$. Set union and intersection as well as the complement $-\varphi = \{x : A \mid \neg\varphi(x)\}$ are well-defined operations on sets. Note the difference between operations on *sets* (like union) and on *types* (like disjoint sum).

An indexed family of sets is given by any 2-argument predicate, so that e.g. we can write the ordinary definition $\bigcap_{y:Y} A_y = \{x : X \mid \forall y{:}Y.\, A_y(x)\}$. Should we need an intersection indexed by elements of a *set* rather than a type we must explicitly include it in the definition by writing

$$\bigcap_{y \in \psi} A_y = \{x : X \mid \forall y{:}Y\, (\psi(y) \to A_y(x))\}.$$

[1] At least not yet. We may have to relax this restriction, if we want to deal with e.g. objects of type "semigroup". This may lead to an infinite hierarchy of universes.

At this stage, one can prove standard results about the properties of the algebra of sets. Subsets of a Cartesian product $A \times A$ are of course called relations, and we can discuss properties of relations and introduce constructions like transitive closure and so on.

Equivalences and quotients. While a definition of an equivalence relation (possibly partial) over a type A presents no difficulty, the notion of a quotient type must be postulated separately. Clearly, for every $a : A$ we could consider a set $[a]_r = \{b : A \mid b\, r\, a\}$, and form a subset of $P(A)$ consisting of all such sets. However, that would be inconsistent with our main idea: a domain of interpretation is always a *type* and not a set. Also, there is no actual reason to *define* the abstract objects, the *classes of abstraction*, as *equivalence sets*, as it is done in set theory. There is a difference between abstraction and implementation. For instance, we define rationals from integers, but we do not think of 1/2 as of a set.

The quotient type $A/_r$ induced by a (partial) equivalence r should be equipped with a canonical (partial) map $\mathit{abstract} : A \to A/_r$ and (as a form of the axiom of choice) one could also postulate another map $\mathit{select} : A/_r \to A$ satisfying $\mathit{abstract} \circ \mathit{select} = \mathrm{id}_{A/_r}$. The one-to-one correspondence between the quotient type and the set of equivalence classes should then be proven as a theorem ("the principle of equivalence").

Functions: total or partial? The notion of a function brings the first serious difficulty. In typed systems, once we assert $f : A \to B$ and $a : A$ we usually conclude $f(a) : B$. That means we treat $f(a)$ as a legitimate, well-defined object of type B. Everything works well as long as we can assume that all functions from A to B are total. However, it can happen that a function is defined only on a certain subset A' of a given domain. In set theory this is not a problem, because both the type of arguments and the actual domain are simply sets, and we can always take $f : A' \to B$ rather than $f : A \to B$. In the typed framework, we would like to still say that e.g. $\lambda x{:}\mathbb{R}.\, 1/x$ maps reals to reals, but the *domain* of the function is a proper subset of the type $\mathbb{R}$. There are several possible solutions of this problem, see [11,15].

Perhaps the most adequate solution for our needs is to distinguish between the function space $A \to B$ and the type $A \multimap B$ of partial functions. Then one assigns a domain predicate $\mathrm{dom}(f)$ to every partial function f, and imposes a restriction on the use of the expression $f(a)$ to the cases when $a \in \mathrm{dom}(f)$. This seems to be quite consistent with the ordinary mathematical practice.

The old idea of a definite description may turn out useful in this respect. A standard function definition may have the form $f(x) = \iota y.\varphi(x, y)$, or equivalently $f = \lambda x\, \iota y.\varphi(x, y)$, and we would postulate an axiom of the form

$$x \in \mathrm{dom}(\lambda x\, \iota y.\varphi(x, y)) \qquad \text{iff} \qquad \exists! y\, \varphi(x, y).$$

Extensionality for partial functions would then be stated as

$$f = g \quad \leftrightarrow \quad (\mathrm{dom}(f) = \mathrm{dom}(g)) \wedge \forall x\, (x \in \mathrm{dom}(f) \to f(x) = g(x)).$$

This approach assumes that $f(a)$ is a well-formed expression of the appropriate type only if $a \in \mathrm{dom}(f)$, a problem that does not formally occur[2] in set theory,

[2] But it occurs in practice: e.g. $f(x) \neq y$ can be understood differently than $\langle x, y \rangle \notin f$.

where $f(x) = y$ is syntactic sugar for $\langle x, y \rangle \in f$. In type theory, it is more natural to refrain from entering this level of extensionality, and to assume function application as a primitive.

Mathematics is not an exact science. Various identifications are common in mathematical practice. In a strictly typed framework such identifications are unavoidable. For instance, we would like to treat total functions as special cases of partial functions, even if these are of two different types. There is a natural coercion from $A \to B$ to $A \multimap\!\!\to B$, which can, at the meta-level, be treated as identity without creating confusion.

Also the difference between types and subsets becomes inconvenient in certain situations. One specific example is when we have an algebra with a domain represented as a type, and we need to consider a subalgebra based on a subset of that domain. Then we would prefer to have the "large" and the "small" domain living in the same sort. To overcome this difficulty, one may have to postulate a selection scheme: for every subset S of type A there exists a type $A|S$, such that objects of type $A|S$ are in a bijective correspondence with elements of S. This partially brings back the identification of domains and predicates, but in a controlled way.

4 Naive Type Theory as a Pure Type System

The assumption that a set and a powerset should live in the same sort leads naturally to the following idea: consider a pure type system with the usual axiom $* : \Box$ and with the rule $(*, \Box, *)$. This rule makes possible to build products of the form $\Pi x{:}A.\, \kappa$, where $A : *$ and $\kappa : \Box$, and the product itself is then a type (is assigned the sort $*$). In particular, the function space $A \to *$ is a type, and this is exactly the powerset of A. A subset of A is then represented by any abstraction $\lambda x{:}A.\, \varphi(x)$, where $\varphi(x)$ is a (dependent) proposition.

Unfortunately, this idea is too naive. As pointed out by A.J.C. Hurkens and H. Geuvers, this theory suffers from Girard's paradox, and thus it is inconsistent.

Theorem 1 (Geuvers, Hurkens [17]). *Let* VNTT (*Very Naive Type Theory*) *be an extension of* λP *by the additional rule* $(*, \Box, *)$. *Then every type is inhabited in* VNTT (*every proposition has a proof*).

Proof. The proof is essentially the same as Hurkens' proof in [19], (cf. the version given in [24, chapter 14]) for the system λU^-. There are two essential factors that imply that Russel's paradox can be implemented in a theory:

- A powerset $P(x)$ of a domain x of a sort s lives in the same sort s.
- There is enough polymorphism available in s to implement a construction of an inductive object $\mu x{:}s.P(s)$.

In λU^- we have $s = \Box$ and polymorphism on the kind level is directly available. But almost the same can happen in VNTT, for $s = *$. Indeed, the powerset $A \to *$ of any type A is a type, and although type polymorphism as such is not present, it sneaks in easily by the back door. Instead of quantifying over types, one can quantify over object variables of type $T \to *$, where T is any type. Thus instead of using $\mu t : *.P(t) = \forall t(\forall u{:}\, * \, ((u \to t) \to P(u) \to t) \to t)$ one takes $a : T$ and then defines

$$\mu t : *.P(t) = \forall x : T \to * \, (\forall y : T \to * \, ((ya \to xa) \to P(ya) \to xa) \to xa),$$

with essentially the same effect. □

It follows that our naive type theory cannot be too naive, and must avoid the danger of Girard's paradox. The solution is to distinguish between propositions and sets, like in Coq.

Define a pure type system LNTT (Less Naive Type Theory) with four sorts

$$*^t, \quad *^p, \quad \Box^t, \quad \Box^p,$$

with axioms $(*^t : \Box^t)$ and $(*^p : \Box^p)$ and with the following rules:

$$(*^t, *^t, *^t), \quad (*^p, *^p, *^p), \quad (*^t, \Box^t, \Box^t), \quad (*^t, *^p, *^p), \quad (*^t, \Box^p, *^t).$$

The first and second rule represent, respectively, the formation of function types, and logical implication; the third rule is for dependent types and the fourth one permits quantification over objects of any type. The last rule is for the powerset.

Note that there is no polymorphism involved, as rule $(\Box^t, *^t, *^t)$ can be fatal; however impredicativity is still present because of rule $(*^t, \Box^p, *^t)$.

Strong Normalization

First note that, as all PTSs with only β-reduction, the system LNTT has the Church-Rosser property and subject reduction property on well-typed terms [16]. Moreover, LNTT is a singly sorted PTS [3], so the uniqueness of types also holds.

Definition 2. In a fixed context Γ we use the following terminology.

1. A is a *term* if and only if there exists B such that $\Gamma \vdash A : B$ or $\Gamma \vdash B : A$.
2. A is a *kind* if and only if $\Gamma \vdash A : \Box^t$.
3. A is a *constructor* if and only if there exists B such that $\Gamma \vdash A : B : \Box^t$.
4. A is a *type* if and only if $\Gamma \vdash A : *^t$.
5. A is a *formula* if and only if $\Gamma \vdash A : *^p$.
6. A is an *object* if and only if there exists B such that $\Gamma \vdash A : B : *^t$.
7. A is a *proof* if and only if there exists B such that $\Gamma \vdash A : B : *^p$.

The classification of terms in LNTT is more complicated than e.g. in the calculus of constructions λC. While in λC there is a simple „linear" hierarchy (from objects via types/constructors to kinds), in LNTT we also have a separate hierarchy from proofs via formulas to $*^p$. The relation between the two hierarchies is not straightforward: in some respects formulas correspond to types, in other to objects. This is why we need two translations in the proof of strong normalization. We use the notation $Term_\Gamma$, $Kind_\Gamma$, $Constr_\Gamma$, $Type_\Gamma$, $Prop_\Gamma$, Obj_Γ, and $Proof_\Gamma$ for, respectively, terms, kinds, constructors, types, formulas, objects, and proofs of the context Γ. The following lemma explains the various cases.

Lemma 3. *Assume a fixed context Γ*

- *If A is a term such that $\Gamma \vdash A : \Box^p$ then $A = *^p$.*
- *If A is a kind then A is of the following form*
  - *$A = *^t$ or*
  - *$A = \Pi x : \tau.B$ where τ is a type and B is a kind.*
- *If A is a constructor then*
  - *A is a type, or*
  - *A is a variable, or*
  - *A is of the form $\lambda x : \tau.\kappa$ where τ is a type and κ is a constructor, or*
  - *A is of the form κM where M is an object and κ is a constructor.*
- *If A is a type then*
  - *A is a type variable, or*
  - *A is of the form $\Pi x : \tau.\sigma$ where τ and σ are types, or*
  - *A is of the form $\Pi x : \tau.*^p$ where τ is a type, or*
  - *A is of the form κM where M is an object and κ is a constructor.*
- *If A is a formula then*
  - *A is a propositional variable, or*
  - *A is of the form $\Pi x : \varphi.\psi$ where φ and ψ are formulas, or*
  - *A is of the form $\Pi x : \tau.\varphi$ where τ is a type and φ is a formula, or*
  - *A is of the form MN where M and N are objects.*
- *If A is an object then*
  - *A is an object variable, or*
  - *A is of the form $\lambda x : \tau.N$ where τ is a type and N is an object, or*
  - *A is of the form $\lambda x : \tau.\varphi$ where τ is a type and φ is a formula, or*
  - *A is of the form MN where M and N are objects.*
- *If A is a proof then*
  - *A is a proof variable, or*
  - *A is of the form $\lambda x : \tau.D$ where τ is a type and D is a proof, or*
  - *A is of the form $\lambda x : \varphi.D$ where φ is a formula and D is a proof, or*
  - *A is of the form $D_1 D_2$ where D_1 and D_2 are proofs, or*
  - *A is of the form DN where D is a proof and N is an object.*

Lemma 4. *If A is a term which is not a proof and B is a subterm of A then B is not a proof.*

Proof. This is an immediate consequence of Lemma 3. □

Note that it follows from Lemma 4 that all formulas of the form $\Pi x : \varphi.\, \psi$, where φ and ψ are formulas, are actually implications (can be written as $\varphi \to \psi$) because the proof variable x cannot occur in ψ.

The first part of our strong normalization proof applies to all terms but proofs. For a fixed context Γ we define the translation

$$T_\Gamma : Term_\Gamma - Proof_\Gamma \;\to\; Term(\lambda P2)$$

from terms of LNTT into the system $\lambda P2$. Special variables *Bool*, *Forall* and *Impl* will be used in the definition of T. Types for these variables are given by the following context:

$$\Gamma_0 = \{Bool : *, Forall : \Pi\tau{:}*.\, (\tau \to Bool) \to Bool, Impl : Bool \to Bool \to Bool\}.$$

Definition of the translation T_Γ follows:

- $T_\Gamma(\Box^t) = \Box$;
- $T_\Gamma(\Box^p) = *$;
- $T_\Gamma(*^t) = *$;
- $T_\Gamma(*^p) = Bool$;
- $T_\Gamma(x) = x$, when x is a variable;
- $T_\Gamma(\Pi x : A.B) = \Pi x : T_\Gamma(A).T_{\Gamma,x:A}(B)$, for products created with the rules $(*^t, *^t, *^t)$, $(*^t, \Box^p, *^t)$, $(*^t, \Box^t, \Box^t)$;
- $T_\Gamma(\Pi x : \tau.\varphi) = Forall\ T_\Gamma(\tau)\ (\lambda x : T_\Gamma(\tau).T_{\Gamma,x:\tau}(\varphi))$, for products created with the rule $(*^t, *^p, *^p)$;
- $T_\Gamma(\Pi x : \varphi.\psi) = Impl\ T_\Gamma(\varphi)\ T_\Gamma(\psi)$, for products created with the rule $(*^p, *^p, *^p)$;
- $T_\Gamma(\lambda x : A.B) = \lambda x : T_\Gamma(A).T_{\Gamma,x:A}(B)$;
- $T_\Gamma(AB) = T_\Gamma(A)\ T_\Gamma(B)$.

For the sake of simplicity we omit the subscript Γ if it is clear which context we are using.[3] Note that we cannot apply the translation T to proofs. Formulas of LNTT get translated by T_Γ into objects of $\lambda P2$. Thus each abstraction of the form $\lambda x : \varphi.N$ would have to be translated into an expression $\lambda x : T(\varphi).T(N)$. But $T(\varphi)$ is an object so this expression would be ill-formed.
The translation T is extended to contexts as follows:

- $T(\langle\rangle) = \Gamma_0$,
- $T(\Gamma, x : A) = T(\Gamma), x : T_\Gamma(A)$, if A is a kind, a type, or $*^p$,
- $T(\Gamma, x : A) = T(\Gamma)$, if A is a formula.

We now state some technical lemmas which are used in the proof of soundness of the translation T.

Definition 5. *We say that contexts Γ and Γ' are* equivalent *with respect to the set of variables $X = \{x_1, \ldots, x_n\}$ if and only if Γ and Γ' are legal contexts and for all $x \in X$ we have $\Gamma(x) =_\beta \Gamma'(x)$.*

Lemma 6. *If Γ and Γ' are equivalent with respect to X, and $N \in Term_\Gamma$ is such that $FV(N) \subseteq X$ and $\Gamma \vdash N : A$, then $\Gamma' \vdash N : A'$ where $A =_\beta A'$. In particular, if $N \in Term_\Gamma$ then $N \in Term_{\Gamma'}$.*

Proof. Induction with respect to the structure of the derivation $\Gamma \vdash N : A$. □

Lemma 7. *If Γ and Γ' are equivalent with respect to $FV(M)$ and $M \in Term_\Gamma$ then $T_\Gamma(M) = T_{\Gamma'}(M)$.*

Proof. Induction with respect to the structure of M. □

Lemma 8. *If $\Gamma \vdash a : A$ and $\Gamma, x : A \vdash B : C$ for some C and a, B are not proofs then $T_\Gamma(B[x := a]) = T_{\Gamma,x:A}(B)[x := T_\Gamma(a)]$.*

Proof. Induction with respect to the structure of B, using Lemma 7. □

[3] I.e., when the context is clear from the context ;)

Lemma 9. *If B and B' are not proofs and $B \to_\beta B'$ then $T_\Gamma(B) \twoheadrightarrow_\beta^+ T_\Gamma(B')$.*

Proof. The proof is by a routine induction with respect to $B \to_\beta B'$. If B is a redex then, by Lemma 4, it must be of one of the following forms:

$$(\lambda x : \tau.\varphi)N, \qquad (\lambda x : \tau.M)N, \qquad (\lambda x : \tau.\kappa)N,$$

where τ is a type, φ is a formula, and M, N are objects. In each of these cases we apply Lemma 8. If B is not a redex, we apply the induction hypothesis, using Lemma 7. □

Lemma 10. *If $B =_\beta B'$ and B, B' are kinds, types or objects then $T_\Gamma(B) =_\beta T_\Gamma(B')$.*

Proof. By Church-Rosser property there exists a well-typed term C such that $B \twoheadrightarrow_\beta C$ and $B' \twoheadrightarrow_\beta C$. We have $T_\Gamma(B) \twoheadrightarrow_\beta T_\Gamma(C)$ and $T_\Gamma(B') \twoheadrightarrow_\beta T_\Gamma(C)$, by Lemma 9, whence $T_\Gamma(B) =_\beta T_\Gamma(B')$. □

Lemma 11 (Soundness of the translation T). *If $\Gamma \vdash A : B$ and A is not a proof then $T(\Gamma) \vdash T_\Gamma(A) : T_\Gamma(B)$ in $\lambda P2$.*

Proof. Induction with respect to the structure of the derivation of $\Gamma \vdash A : B$ using Lemmas 7, 8 and 10. □

Corollary 12. *If M is not a proof then M is strongly normalizing.*

Proof. Assume that there is an infinite reduction $M \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots$ By Lemma 9 then $T(M) \twoheadrightarrow_\beta^+ T(M_1) \twoheadrightarrow_\beta^+ T(M_2) \twoheadrightarrow_\beta^+ \cdots$ But $T(M)$ is a valid term of $\lambda P2$, by Lemma 11, thus it is strongly normalizing. The contradiction shows that also M is strongly normalizing. □

To show strong normalization for proofs we use another translation t from LNTT to the calculus of constructions λC. This translation depends on a given context Γ. Observe that the classification of a term A in LNTT does not a priori determine the classification of $t(A)$ in λC. For instance, some types of LNTT are translated to types and some (those which have $*^p$ as a "target") to kinds, cf. Lemma 18. Similarly, some object terms of LNTT are translated as type constructors. Note that we do not define the translation for $\Box^t$ and $\Box^p$ as it is not needed for soundness.

- $t_\Gamma(*^t) = *$;
- $t_\Gamma(*^p) = *$;
- $t_\Gamma(x) = x$, if x is a variable;
- $t_\Gamma(\Pi x : \tau.B) = t_{\Gamma,x:\tau}(B)$, for products constructed using the rule $(*^t, \Box^t, \Box^t)$;
- $t_\Gamma(\Pi x : A.B) = \Pi x : t_\Gamma(A).t_{\Gamma,x:A}(B)$, for all other products;
- $t_\Gamma(\lambda x : \tau.\kappa) = t_{\Gamma,x:\tau}(\kappa)$, if κ is a constructor and τ is a type;
- $t_\Gamma(\lambda x : A.B) = \lambda x : t_\Gamma(A).t_{\Gamma,x:A}(B)$, for all other abstractions;
- $t_\Gamma(\kappa N) = t_\Gamma(\kappa)$, if κ is a constructor;
- $t_\Gamma(AB) = t_\Gamma(A)t_\Gamma(B)$, for all other applications.

We extend the translation t to contexts by taking

$$t(\langle\rangle) = \langle\rangle \quad \text{and} \quad t(\Gamma, x : A) = t(\Gamma), x : t_\Gamma(A).$$

Lemma 13. *If Γ and Γ' are equivalent with respect to $FV(M)$ and M is a term in Γ then $t_\Gamma(M) = t_{\Gamma'}(M)$.*

Proof. Induction with respect to the structure of M. □

Lemma 14. *Assume that $\Gamma, x : A \vdash B : C$ and $\Gamma \vdash N : A$ and N is an object or a proof.*

- *If N is an object and B is a type or a constructor then*
$$t_\Gamma(B[x := N]) = t_{\Gamma,x:A}(B).$$
- *If B is neither a type nor a constructor then*
$$t_\Gamma(B[x := N]) = t_{\Gamma,x:A}(B)[x := t_\Gamma(N)].$$

Proof. Induction with respect to the structure of B, using Lemma 7. □

Definition 15. A reduction step $A \to_\beta A'$ is *silent* if

- $A = (\lambda x : \tau.\kappa)N \to_\beta \kappa[x := N] = A'$, where κ is a constructor and N is an object, or
- $A = \Pi x : \tau.B \to_\beta \Pi x : \tau'.B = A'$, where $\tau \to_\beta \tau'$ and B is a kind, or
- $A = \kappa N \to_\beta \kappa N' = A'$, where $N \to_\beta N'$ and κ is a constructor, or
- $A = \lambda x : \tau.\kappa \to_\beta \lambda x : \tau'.\kappa = A'$, where κ is a constructor, or
- $A = C[B] \to_\beta C[B'] = A'$, where $C[\,]$ is any context and $B \to_\beta B'$ is a silent reduction.

Lemma 16. *If $A \to_\beta B$ then $t_\Gamma(A) \twoheadrightarrow_\beta t_\Gamma(B)$. In addition, if the reduction $A \to_\beta B$ is not silent then $t_\Gamma(A) \twoheadrightarrow_\beta^+ t_\Gamma(B)$.*

Proof. Induction with respect to $A \to_\beta B$, using Lemma 14 when A is a redex, and Lemma 13 in the other cases. □

Corollary 17. *If $B =_\beta B'$ then $t_\Gamma(B) =_\beta t_\Gamma(B')$.*

The following lemma states soundness of the translation t_Γ. In particular, item 2 implies that all the rules in the calculus of constructions are needed.

Lemma 18. *Assume a fixed environment Γ.*

1. *If M is a proof, an object, or a formula and $\Gamma \vdash M : B$ holds in* LNTT *then $t(\Gamma) \vdash t_\Gamma(M) : t_\Gamma(B)$ in λC.*
2. *If M is a type or a constructor then $t(\Gamma) \vdash t_\Gamma(M) : *$ or $t(\Gamma) \vdash t_\Gamma(M) : \Box$.*
3. *If M is a kind then $t(\Gamma) \vdash t_\Gamma(M) : \Box$.*

Proof. Simultaneous induction with respect to the structure of the appropriate derivation, using Lemma 13. □

Theorem 19. *System* LNTT *has the strong normalization property.*

Proof. We already know that all expressions except proofs are strongly normalizing. Arguing as in the proof of Corollary 12, and using Lemma 16, we conclude that almost all steps in an infinite reduction sequence must be silent. Thus it suffices to prove that if D is a proof than there is no infinite silent reduction of D. This goes by induction with respect to the size of D, by cases depending on its shape. □

No Conclusion

The above is by no means a complete proposal of either theoretical or didactic character. It is essentially a collection of questions and partial suggestions of how such a proposal should be eventually designed. These questions are of double nature, and we would like to pursue the two directions. The first one is to find means to talk about basic mathematics without referring to set theory in either a naive (i.e., inconsistent) or axiomatic way, using instead an appropriate type-based language. That should happen in a possibly non-invasive way, keeping as much linguistic compatibility with the "standard" style as possible.

The second problem is to give a formal foundation to this informal type-based language. This formalization is to be used for two purposes: to guarantee logical consistency of the naive exposition and to facilitate computer assisted verification and teaching. That requires building a complex system, of which our PTS-style Less Naive Type Theory is just a very basic core. This system must involve various extensions as in [4], perhaps include a hierarchy of sorts, etc. All this is future work.

Acknowledgement

Thanks to Herman Geuvers and Christophe Raffalli for helpful discussions. Also thanks to the anonymous referees for their suggestions.

References

1. Adams, R., Luo, Z.: Weyl's predicative classical mathematics as a logic-enriched type theory. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, Springer, Heidelberg (2007)
2. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, 2nd edn. Applied Logic Series, vol. 27. Kluwer Academic Publishers, Dordrecht (2002)
3. Barendregt, H.P.: Lambda calculi with types. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. II, pp. 117–309. Oxford University Press, Oxford (1992)
4. Barthe, G.: Extensions of pure type systems. In: Dezani-Ciancaglini and Plotkin [10], pp. 16–31
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2004)
6. de Bruijn, N.G.: A survey of the project AUTOMATH. In: Seldin, J.P., Hindley, J.R. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 579–606. Academic Press, London (1980)
7. Chrząszcz, J., Sakowicz, J.: Papuq: A Coq assistant (manuscript, 2007)
8. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic 5(2), 56–68 (1940)

9. Constable, R.L.: Naive computational type theory. In: Schwichtenberg, H., Steinbruggen, R. (eds.) Proof and System-Reliability, pp. 213–259. Kluwer Academic Press, Dordrecht (2002)
10. Dezani-Ciancaglini, M., Plotkin, G. (eds.): TLCA 1995. LNCS, vol. 902. Springer, Heidelberg (1995)
11. Farmer, W.M.: A partial functions version of Church's simple theory of types. Journal of Symbolic Logic 55(3), 1269–1291 (1990)
12. Farmer, W.M.: A simple type theory with partial functions and subtypes. Annals of Pure and Applied Logic 64, 211–240 (1993)
13. Farmer, W.M.: A basic extended simple type theory. Technical Report 14, McMaster University (2003)
14. Farmer, W.M.: The seven virtues of simple type theory. Technical Report 18, McMaster University (2003)
15. Farmer, W.M.: Formalizing undefinedness arising in calculus. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 475–489. Springer, Heidelberg (2004)
16. Geuvers, H.: The Church-Rosser property for beta-eta-reduction in typed lambda calculi. In: Logic in Computer Science, pp. 453–460 (1992)
17. Geuvers, H.: Private communication (2006)
18. Halmos, P.R.: Naive Set Theory. Van Nostrand, 1960. Reprinted by Springer, Heidelberg (1998)
19. Hurkens, A.J.C.: A simplification of Girard's paradox. In: Dezani-Ciancaglini and Plotkin [10], pp. 266–278
20. Jensen, R.B.: On the consistency of a slight(?) modification of Quine's NF. Synthese 19, 250–263 (1969)
21. Kamareddine, F., Laan, T., Nederpelt, R.: Types in logic and mathematics before 1940. Bulletin of Symbolic Logic 8(2), 185–245 (2002)
22. Luo, Z.: A type-theoretic framework for formal reasoning with different logical foundations. In: Okada, M., Satoh, J. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 214–222. Springer, Heidelberg (2006)
23. Quine, W.V.: New foundations for mathematical logic. American Mathematical Monthly 44, 70–80 (1937)
24. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism. Elsevier, Amsterdam (2006)
25. Weyl, H.: The Continuum. Dover, Mineola, NY (1994)