Some remarks about Dependent Type Theory

Introduction

The goal of this paper is to describe a calculus designed in 84/85 [19] and later presented in [28]. This calculus was obtained by applying the ideas introduced by N.G. de Bruijn [30, 31] for AUTOMATH to some functional systems created by J.-Y. Girard [40]. There was also strong connections with the work of P. Martin-Löf [64, 66]. This calculus provided quite simple uniform notations for proofs and (functional) programs. Because of this simplicity and uniformity, it was possible to use it for analysing logical problems such as impredicativity [81, 82], paradoxes [20, 26, 23, 7], but also notions of computer science such as parametricity [9, 3]. It could also be used as the basis of implementations of proof and functional systems [27], arguably simpler, or at least competitive, with the ones that were available at the time [46, 17, 76].

One main theme of this work is the importance of notations in mathematics and computer science: new questions were asked and solved only because of the use of AUTOMATH notation, itself a variation of $\lambda$-notation introduced by A. Church [14, 58] for representing functions.

1 General motivations and context

1.1 System $F$

The starting point of this work was a study of system $F$. This is a quite remarkable extension of simply typed $\lambda$-calculus [14]. It was designed independently by Girard [40], motivated by purely logical considerations [42, 40], and by J. Reynolds [85], with the different goal of representing and analysing the notion of “parametric” algorithms.

System $F$ extends simply typed $\lambda$-calculus type variables and notations for parametric functions: $\Lambda\alpha M$ is of type $\forall\alpha T(\alpha)$ if $M$ is of type $T(\alpha)$ and $\alpha$ is a type variable. This system was quite mysterious: the parametric identity function $\Lambda\alpha \lambda x^\alpha x^\alpha$ of type $\forall\alpha (\alpha \to \alpha)$ has no clear set-theoretic semantics for instance. But things were subtle, and Reynolds was conjecturing [84] that it should actually be possible to find a set-theoretic semantics, only to prove one year later [86] a theorem that there cannot be such a semantics. Girard could show that system $F$ satisfies the normalisation property [40], by an ingenious refinement of Tait’s computability method [95]. This result implied in turn a syntactic solution to the famous Takeuti’s conjecture [96].

System $F$ was also formally a complex system compared to simply typed $\lambda$-calculus. In particular, there was a non obvious restriction on typed variables: for forming $\Lambda\alpha M$ we should have that $\alpha$ does not appear free in some type of a variable of $M$. For instance, the term $\Lambda\alpha x^\alpha$ is not allowed. Girard, both in the original paper [40] and in his habilitation thesis [41], presented an extension $F^\omega$ even more complex.

1.2 The system AUTOMATH

At about the same time as system $F$ was designed, N.G. de Bruijn was creating [30] another extension of simply typed $\lambda$-calculus. The goal there was to implement a system which can check the correctness of mathematical proofs. One non standard feature was a uniform treatment of propositions and types, and of programs/terms and proofs. An example in [30] is a statement of the form

---

1For instance, a sorting algorithm is parametric, in the sense that it will act in an uniform way w.r.t. the types of its elements.

2The original notation in [41] was $DT\alpha\lambda x^\alpha, x^\alpha$ of type $\delta\alpha(\alpha \to \alpha)$. 
Theorem 1.1 Let $x$ be a real number such that $f(x) > 1$ and let $n$ be a natural number. If we have $g(x) > x^n$ then $f(x) > n$.

If a mathematician wants to use this statement later on, with $x = \pi$ and $n = 5$, he has to give a proof $p_1$ of $f(\pi) > 1$ and then a proof $p_2$ of $g(x) > x^5$. He can then state $f(\pi) > 5$ by applying the theorem and by giving in this order: $\pi$, the proof $p_1$, then 5 and the proof $p_2$.

In AUTOMATH this will become the definition of a term, representing a corollary of the theorem, as an application, in the sense of $\lambda$-calculus, of a variable $thm$ to some arguments like $thm(\pi, p_1, 5, p_2)$, which is of “type” the statement $f(\pi) > 5$.

As de Bruijn wrote: “Treating propositions as types is definitely not in the way of thinking of ordinary mathematician, yet it is very close to what he actually does”, and to have such a uniform treatment of elements and proofs was a quite original feature. It is interesting, and really surprising, that this feature is a key for representing in a natural way notions from homotopy theory, as was found later by V. Voevodsky [101].

A crucial notion in AUTOMATH, inspired from the notion of block structure in ALGOL 60, is the one of context. It is a sequence of variable declaration with their types and named hypotheses in an arbitrary order. In the previous example, we have the following context

$$x : R, h_1 : f(x) > 1, n : N, h_2 : g(x) > x^n$$

for the previous theorem, and to apply the theorem, we have to find an instantiation of this context.

AUTOMATH introduced also a primitive “sort” for collecting mathematical types, so that we have $\text{real} : \text{type}$ and $\text{nat} : \text{type}$ for types of real numbers and natural numbers.

One also could introduce a primitive sort $\text{prop}$ for collecting mathematical propositions, or, since the notion of types and the notions of propositions are treated uniformly, simply take $\text{prop} = \text{type}$.

Finally, AUTOMATH used the same notation $[x : A]M$ for typed abstraction $\lambda_{x:A}M$ and for dependent product $\Pi_{x:A}B$.

One obtained then a quite minimal calculus

$$M, A ::= x \mid M M \mid [x : A]M \mid \text{type}$$

As usual, implication/function type $A \to B$ can be defined as $[x : A]B$, where $x$ does not occur in $A, B$. We may write $[x_1 \ x_2 : A]B$ for $[x_1 : A][x_2 : A]B$. Also we have a natural notion of convertibility from $\lambda$-calculus which is $\beta$-conversion.

Using the same notation for abstraction and dependent product, we can purely formally compute the type of $M$ in a context $\Gamma$, by simply replacing its head variable by its type. For instance $[A : \text{type}]x : A[[f : [z : A]A] f x$ is of type


which is convertible to $[A : \text{type}]x : A[[f : [z : A]A]A$, that is $[A : \text{type}] A \to (A \to A) \to A$.

1.3 Representation of mathematical notions in AUTOMATH

One of the first example represented in the system [30] was the notion of equality on a type $A : \text{type}$. This is represented by a collection of constants. One constant $\text{Eq} : [x y : A] \text{type}$ represents the notion of equality itself: if $a_0$ and $a_1$ are elements of type $A$, then $\text{Eq} a_0 a_1$ is a type/proposition. An element of this type would then be a proof that $a_0$ and $a_1$ are equal. In order to represent that this is an equivalence relation, we introduce two constants

$$\text{refl} : [x : A] \text{Eq } x x \quad \text{eucl} : [x y z : A] \text{Eq } x z \to \text{Eq } y z \to \text{Eq } x y$$

We can then form the term

$$[x y : A][h : \text{Eq } x y] \text{eucl } y x y (\text{refl } y) h : [x y : A] (\text{Eq } x y) \to \text{Eq } y x$$

which is a proof that the relation $\text{Eq}$ is symmetric.
This example illustrates the uniform treatment of elements and proofs.

The following example was also quite illuminating. Heyting’s deduction rules for intuitionistic logic looked quite formal, e.g. why do we have $A \rightarrow \neg \neg A$ and not $(\neg \neg A) \rightarrow A$? With AUTOMATH notation this becomes clear. First, negation can be defined as $\neg A = A \rightarrow \bot$, where $\bot$: type represents the false proposition. It is then no problem to build an element

$$[x : A][f : A \rightarrow \bot]f x : A \rightarrow \neg \neg A$$

while it is at least intuitively clear that we cannot build an element of type $(\neg \neg A) \rightarrow A$, since one cannot build any element of type $A$ simply using an hypothesis of type $(A \rightarrow \bot) \rightarrow \bot$.

A more complex example, also illuminating, is presented by Jutting [54], who gives different versions of a proof of the basic result that any surjective map from $[1, n]$ to itself is injective, first informally and then with more and more formal details, until reaching a proof in AUTOMATH.

2 AUTOMATH and system $F_\omega$

The idea\(^3\) was then to use AUTOMATH powerful notations to represent system $F^4$. We encode $\Lambda M$ by $[\alpha : \text{type}]M$ and $\forall T(\alpha)$ by $[\alpha : \text{type}]T(\alpha)$. We can for instance represent the polymorphic identity as the term $\text{id} = [A : \text{type}]x : A, x$, of type $T = [A : \text{type}]A \rightarrow A$. We can then apply $\text{id}$ to its own type $T$, which should be convertible to $[x : T]x$.

We also can define the type $\bot = [A : \text{type}]A$ and it represents the false proposition, since it implies any type, as shown by the following proof

$$[A : \text{type}]h : \bot \rightarrow A : [A : \text{type}] \bot \rightarrow A$$

Note that we have several proofs of $\bot \rightarrow \bot$, for instance $[x : \bot]x$ or $[x : \bot]x \rightarrow \bot$ or $[x : \bot]x \rightarrow x$ or

$$[x : \bot][A : \text{type}]x A.$$ (This last term can be seen as the $\eta$-expansion of $[x : \bot]x$.)

We also can represent the notion of equality, not by introducing a new constant, but as a definition, as “Leibnitz’ equality”

$\text{eq} = [A : \text{type}]x \rightarrow A \rightarrow P. y$ $x : A \rightarrow P \rightarrow P \rightarrow P$ $y$

We then define a term which represents the proof of reflexivity

$$\text{refl} : [A : \text{type}]x : A \rightarrow A \bullet x = [A : \text{type}]x : A \rightarrow P \rightarrow P \rightarrow P$$

and, following [40], we can prove symmetry and transitivity of the eq relation.

One crucial point is that we don’t require\(^4\) type to be of type type; however, like in system $F$, the type $[A : \text{type}]A$ is of type type. It is possible to define an equality on type, but this is not an instance of eq

$$\text{eq} : \text{type} \rightarrow \text{type} \rightarrow \text{type} = [A B : \text{type}]P : \text{type} \rightarrow \text{type} P A \rightarrow P B$$

2.1 Some remarks

One important feature of this presentation, coming from the use of Algol block structure and expressed by the notion of context, is that, at any point in time, there are only finitely many variables that are “alive”, each declared with its type. This is to be contrasted with usual presentations of logical systems and type systems at the time, which assumed an infinite “pool” of variable.

This notion of context also gives a more natural presentation of type abstraction in system $F$. With the usual presentation, when forming $\Lambda M$ one had to be careful that $\alpha$ was not appearing free in the type of some free variables of $M$. In this other presentation, using this idea of treating uniformly variable type declarations and variable term declarations, this becomes that we always can form $[A : \text{type}]M$ in a context $\Gamma$ if $M$ is correct in the context $\Gamma, A : \text{type}$.

---

3 The formal rules are presented in the Appendix.
4 This possibility of using abstraction over type, was actually suggested by de Bruijn [30] but with the mention: “It is difficult to see what happens if we admit this”.
5 Contrary to the system [62] that I discovered after I had formulated the present system.
2.2 Russell-Prawitz encoding of logical connectives

Since we represent system $F$, it is possible to encode logical connectives [88, 2]. Intuitively, a connective is defined by its elimination rule. For instance

$$(\land) = [A B : \text{type}] [X : \text{type}] (A \rightarrow B \rightarrow X) \rightarrow X$$

$$(\lor) = [A B : \text{type}] [X : \text{type}] (A \rightarrow X) \rightarrow (B \rightarrow X) \rightarrow X$$

It is also possible to encode existential quantification in the same way

$$\exists = [A : \text{type}] [B : A \rightarrow \text{type}] [X : \text{type}] ([x : A] B \ x \rightarrow X) \rightarrow X$$

2.3 Encoding of data types

It was also possible to encode data types, following some ideas of Martin-Löf and Girard [69, 41]

bool $= [A : \text{type}] A \rightarrow A \rightarrow A$

nat $= [A : \text{type}] A \rightarrow (A \rightarrow A) \rightarrow A$

which corresponds to Church’s encoding [15]. The definition of nat expresses how to define a function by iteration. But since we have access to binary products, we can follow Kleene’s encoding of primitive recursion in term of iteration. In order to define

$$f 0 = a \quad f (n + 1) = g n (f n)$$

we define the function $h : n \mapsto (n, f n)$ by iteration

$$h 0 = (0, a) \quad h (n + 1) = (\pi_1 (h n), g' (h n))$$

where $g' z = g (\pi_1 z) (\pi_2 z)$. One could then use this encoding to represent the predecessor function.

Surprisingly, one could also encode data types such as list, as suggested by Böhm and Berrarducci [11]

$$\text{list} : \text{type} \rightarrow \text{type} = [A : \text{type}] [X : \text{type}] X \rightarrow (A \rightarrow X \rightarrow X) \rightarrow X$$

We can also reason about these programs, using the same formalism. Early examples of such proofs about programs can be found in [77].

2.4 Evaluation and comparison with other formal systems

We get a formal system which uses a simple and uniform notation. It contains not only system $F_\omega$, but also higher-order logic such as the one presented by A. Church [14, 74]. Inheriting from AUTOMATH the uniform treatment of functions and proofs, we don’t have to describe first the terms, and then deduction rules for formulae, as was done in [14]. As in AUTOMATH, proof-checking becomes type-checking and we get a formal system which can naturally be represented on a computer.

2.4.1 Decidable Type Checking

A distinctive feature of our system is that the judgement $M : A$ is decidable. This is crucial in order to reduce proof-checking to type-checking. The importance of this criteria for a formal system to be able to recognize if a given potential proof is correct or not was stressed by Kreisel [55].

Despite this, all formulations of dependent type theory at the time, by Martin-Löf, Scott and Constable [66, 68, 17, 90], were not following this criteria. Both NuPrl and Martin-Löf’s system were using an equality reflection rule, which had as consequence that the judgement $M : A$ was no longer decidable. Similarly, Scott’s system [90] used some undecidable notion of conversion, despite the fact that it was strongly inspired by AUTOMATH.

---

6Martin-Löf had explored an intensional approach [64], and later with P. Hancock [65], which even excluded the $\xi$-rule, but this was abandoned around 1979 [66, 68]. I was told by Constable that the choice of intensional versus extensional equality was the topic of several discussions in the NuPrl group, but they also opted for the extensional approach.

7There is a discussion at the end of the paper, alluding to objections from Gödel and Kreisel about this issue.
Conceptually, one can argue that this non decidability is not satisfactory: if $M$ is of type $A$, it is not possible to consider anymore $M$ as a proof of $A$ (some crucial information for checking the correctness of the proof may be missing in $M$). But this also implied that the formal system was more complex to implement. Both NuPrl and K. Petersson’s system [76] were using a LCF approach [46], with an abstract data type of theorems. For the present formal system, it was possible to follow the arguably simpler implementation. Both NuPrl and K. Petersson’s system [76] were using a LCF approach [46], with an abstract of the proof may be missing in

A simpler example is $\bot \rightarrow \bot$, which has both an impredicative proof $[x : \bot] x \bot$ and a predicative proof $[x : \bot][A : \text{type}] x A$. Russell, in the introduction of the second edition of Principia Mathematica [104], analyses such examples but without an explicit notation for proofs.

Another example of a data type was the type of ordinals of second class

$$\text{ord} = [A : \text{type}][x : A][f : A \rightarrow A][l : (\text{nat} \rightarrow A) \rightarrow A] A$$

We can program various functional hierarchies $\text{ord} \rightarrow (\text{nat} \rightarrow \text{nat})$, simply by instantiating $A : \text{type}$ by $\text{nat} \rightarrow \text{nat}$. For instance, the Hardy hierarchy $h : \text{ord} \rightarrow \text{nat} \rightarrow \text{nat}$ is obtained by instantiating $x$ to the identity function, $f$ to the functional $f u n = u (n + 1)$ and $l$ to the diagonal function $l v n = v n n$.

One can represent directly elements of type $\text{ord}$ corresponding to various ordinals $\omega$, $\omega^\omega$, $\epsilon_0$, ..., [19]. The paper [33] had a predicative representation of the Hardy function $h_{\epsilon_0}$ and it was possible to compare it with the (much simpler formally) impredicative representation. As it happens, such a discussion was already present in Hilbert’s papers [47, 48] for his attempt to prove the continuum hypothesis.

2.4.2 Impredicativity in logic and functional system

One other feature of the formal system we get is the use of impredicativity, coming from system $F$. This use of impredicativity is crucial in order to be able to define logical connectives and data types from “nothing”. This illustrates the “generative” character of impredicative definitions, which is the “advantage” and mystery of impredicativity; this aspect is described vividly by Poincaré [81, 82].

We get a quite concrete representation of what is going on with proofs using impredicativity. An argument is impredicative precisely when we use the product rule $[x : C] A : \text{type}$ for $A : \text{type}$ in the context $x : C$, and where $C$ is itself of the form $[x_1 : A_1] \ldots [x_n : A_n]$ type (we can have $n = 0$ in which case $C$ is type itself).

An example is the notion of equality which can be defined by

$$\text{eq} = [A : \text{type}][x y : A][P : A \rightarrow \text{type}] P x \rightarrow P y$$

and can be proved to be a symmetric relation. In a context $A : \text{type}$, $x, y : A$, $h : \text{eq} A x y$ we can prove $\text{eq} A y x$ by

$$h ([z : A]\text{eq} A z x) (\text{refl} A x)$$

which would be an impredicative proof: we use $\text{eq}$ as a relation on $A$, while $\text{eq}$ is defined by quantification over all predicates on $A$. Indeed, the correctness of the application $h ([z : A]\text{eq} A z x)$ requires $\text{eq} A z x$ to have the type $\text{type}$, and so we make use of the impredicative quantification

$$[P : A \rightarrow \text{type}] P z \rightarrow P x : \text{type}$$

There is however another predicative proof of the symmetry of $\text{eq}$, which is

$$[P : A \rightarrow \text{type}] [h_1 : P y] h ([z : A] P z \rightarrow P x) ([h_2 : P x] h_2) h_1$$

A simpler example is $\bot \rightarrow \bot$, which has both an impredicative proof $[x : \bot] x \bot$ and a predicative proof $[x : \bot][A : \text{type}] x A$. Russell, in the introduction of the second edition of Principia Mathematica [104], analyses such examples but without an explicit notation for proofs.

Another example of a data type was the type of ordinals of second class

$$\text{ord} = [A : \text{type}][x : A][f : A \rightarrow A][l : (\text{nat} \rightarrow A) \rightarrow A] A$$

We can program various functional hierarchies $\text{ord} \rightarrow (\text{nat} \rightarrow \text{nat})$, simply by instantiating $A : \text{type}$ by $\text{nat} \rightarrow \text{nat}$. For instance, the Hardy hierarchy $h : \text{ord} \rightarrow \text{nat} \rightarrow \text{nat}$ is obtained by instantiating $x$ to the identity function, $f$ to the functional $f u n = u (n + 1)$ and $l$ to the diagonal function $l v n = v n n$. One can represent directly elements of type $\text{ord}$ corresponding to various ordinals $\omega$, $\omega^\omega$, $\epsilon_0$, ..., [19]. The paper [33] had a predicative representation of the Hardy function $h_{\epsilon_0}$ and it was possible to compare it with the (much simpler formally) impredicative representation. As it happens, such a discussion was already present in Hilbert’s papers [47, 48] for his attempt to prove the continuum hypothesis.
There is no problem to define ordinals of third class, introducing further a “constructor” of type \((\text{ord} \to A) \to A\). One can even consider the type \(\text{Huet} [27]\)
\[
\text{ord}_\infty = [A : \text{type}] \{(B : \text{type}) (B \to A) \to A\}
\]

One could as well represent directly in this “minimal” higher-order logic the proof of Tarski’s fixed-point theorem for monotone maps on complete lattice \([51]\).

### 2.4.3 Comparison with Frege’s Begriffsschrift

One of the first examples encoded in this calculus \([19]\) was the result proved by Frege in his remarkable 1879 book *Begriffsschrift* \([34]\). In this book, Frege introduced not only the notion of *quantifiers* but also *higher-order logic*.

As emphasized in von Plato’s book \([102]\), and also in \([89]\), one crucial insight of Frege was the formulation of the rule of \(\forall\) introduction: \(\varphi \rightarrow \forall x \psi(x)\) if \(\varphi \rightarrow \psi(x)\) and \(x\) is not free in \(\varphi\). It is indeed remarkable that one can capture in a finite way the laws for quantification over a maybe infinite collection!

In the present system, following AUTOMATH, this rule simply becomes that \([\forall x : A] M\) is correct in the context \(\Gamma\), if \(M\) is correct in the extended context \(\Gamma, x : A\), i.e. a shift of the abstraction \(x : A\) between the context and the term.

Frege also explained how to encode the transitive closure of a relation using an impredicative definition. His main goal was to show formally that the transitive closure of a functional \(R\) between the context and the term.

\[
\text{her} : (A \to \text{type}) \to \text{type} = [P : A \to \text{type}] [x : A] P x \to [y : A] R x y \to P y
\]

\[
R^+ : A \to A \to \text{type} = [x y : A] [P : A \to \text{type}] \text{her} P \rightarrow ([z : A] R x z \to P z) \to P y
\]

Here are two of the first lemmas, also following Frege \([34]\)

\[
\text{lem}_1 : [x : A] \text{her} (R^+ x) = [x y : A] h_1 : R^+ x y [z : A] h_2 : R y z
\]

\[
P : A \to \text{type} [h_1 : \text{her} P] [h_4 : [u : A] R x u \to P u] h_3 y (h_1 P h_3 h_4) z h_2
\]

\[
\text{lem}_2 : [x y z : A] R^+ x y z \to R^+ x z
\]

\[
= [x y z : A] [h_1 : R^+ x y] [h_2 : R^+ y z] h_2 (R^+ x) (\text{lem}_1 x) (\text{lem}_1 x y h_1)
\]

To have a notation for proofs can make explicit some interesting phenomena: for instance, the proof of \(\text{lem}_2\) uses twice \(\text{lem}_1\) but in different contexts, and the proof of \(\text{lem}_1\) uses twice the hypothesis \(h_3\). One similar application, appearing already in Frege \([34]\) is to record how many times a Lemma is referred to in later proofs, giving hints of what may be key facts in some mathematical developments. One other possibility, which has not really been exploited yet, is that it is now possible to *instantiate* abstract proofs on some concrete arguments, using the \(\beta\)-reduction mechanism of \(\lambda\)-calculus. These instances may then be simplified further, and this could be helpful in order to understand better, or to “run”, a given proof.

F. Pfenning \([80]\) had yet another suggestion of using this notation to find possible generalizations of proofs and concepts.

One important difference with the formalisation of Frege was, once again, the treatment of equality. As in AUTOMATH, we use here the combinatory logic notion of convertibility to represent definitions. If one wants to reason *internally* about equality, one has to use what de Bruijn called “book equality” which is a term of type \(A \to A \to \text{type}\). Frege instead used book equality itself, introduced as a primitive, in order to represent definitions.

When translating Frege’s proof using AUTOMATH notations, I really felt that these notations represented rather faithfully what is going on when one is trying to understand a proof\(^8\). In particular, to express *definition* by convertibility seemed to be preferable in this respect than representing explicitly

\(^8\) A relation \(R\) on a type \(A\) is *functional* if \(R x y\) and \(R x z\) imply \(eq A y z\).

\(^9\) As shown independently by S. Berardi \([8]\) and H. Geuvers \([37]\) the natural embedding of higher logic in this calculus is *not* conservative; this issue is however solved by introducing further universes.
in the proof term itself the process of unfolding definitions. Besides de Bruijn’s work, the importance of
the notion of *definitional equality* is emphasized by Gödel [43], Tait [95] and Martin-Löf [67].

### 3 Inductive Definitions and data types

Frege [34] explained how to encode inductive definitions such as the transitive closure of a relation
\( R : A \to A \to \text{type} \), in higher-order logic. One other encoding of the transitive closure is the following

\[
[x \ y : A][S : A \to A \to \text{type}][(a \ b : A)R \ a \ b \to S \ a \ b) \to \text{trans \ } A \ S \to S \ x \ y]
\]

where \( \text{trans} = [A : \text{type}][S : A \to A \to \text{type}][a \ b \ c : A] \ S \ a \ b \to S \ b \ c \to S \ a \ c. \) This expresses that the
transitive closure of \( R \) is the intersection of all transitive relations containing \( R \).

Similarly the relation of equality can be represented alternatively as the intersection of all reflexive relations

\[
\text{Eq} = [A : \text{type}][x \ y : A][S : A \to A \to \text{type}][(z : A) S \ z \ z) \to S \ x \ y]
\]

instead of

\[
\text{eq} = [A : \text{type}][x \ y : A] [P : A \to \text{type} \ P \ x \to P \ y]
\]

and one can show that equivalence between \( \text{eq} \ A \ x \ y \) and \( \text{Eq} \ A \ x \ y \).

Another example [19] was an encoding of the proof of Newman’s Lemma by G. Huet [52] which was based on the notion of Noetherian relation. This notion is represented by expressing the principle of Noetherian induction

\[
[P : A \to \text{type}][(x : A) (y : A) R \ x \ y \to P \ y) \to P \ x) \to [x : A] P \ x
\]

Note that we obtain the predicate of being accessible for the relation \( R \) [75] simply by shifting the abstraction \( [x : A] \)

\[
[x : A][P : A \to \text{type}][(x : A) (y : A) R \ x \ y \to P \ y) \to P \ x) \to x : A]
\]

One systematic study of inductive definitions represented in this system was carried out in [79]. One quite remarkable example there was the encoding of the system \( F_2 \) in \( F_3 \), which is a good illustration of the use of the correspondance between logical and functional system. One represents a predicate \( P \) on the sort type with constructors of types

\[
[A : \text{type}][B : \text{type}](A \to P \ B) \to P(A \to B)
\]

\[
[A : \text{type}][B : \text{type}](P \ A \to B) \to P \ A \to P \ B
\]

\[
[A : \text{type}][C : \text{type} \to \text{type}][(A : \text{type}) P(C \ A)] \to P([A : \text{type} \ C \ A]
\]

\[
[A : \text{type}][C : \text{type} \to \text{type}][(A : \text{type} \ C \ A)] \to [A : \text{type} \ P(C \ A)
\]

This predicate \( P : \text{type} \to \text{type} \) can also be read as a family of types, and it represents an encoding of \( F_2 \) in \( F_3 \).

Such encoding of data types can also be interesting in an univalent setting as shown in the work\(^{10}\) [5]. For instance the circle is described as

\[
[A : \text{type}][a : A] \ a =_A a \to A
\]

where \( a_0 =_A a_1 \) is a primitive equality type on \( A \). Another example, not mentionned in [5], is the following representation of the type of *integers*

\[
[A : \text{type} \ A \to A \simeq A \to A
\]

where \( A \simeq B \) is the type of equivalences between two types.

\(^{10}\)A model of this system extended with univalence for type is described in [97].
As explained in a letter from G. Plotkin to J. C. Reynolds, commenting on [86] and which became the paper [87], the general pattern for representing inductive types is to use

\[ A = \exists \text{type} \ (T \ X \rightarrow X) \rightarrow X \]

as a “weak” initial algebra for an operation \( T : \text{type} \rightarrow \text{type} \).

As soon as \( T \) is monotone, i.e. we have a term mon of type

\[ [X \ Y : \text{type}] \ (X \rightarrow Y) \rightarrow (T \ X \rightarrow T \ Y) \]

we can build an element of type \( T A \rightarrow A \)

\[ [u : T \ A][X : \text{type}][v : T \ X \rightarrow X] \ v \ (\text{mon} ([a : A] \ a \ X \ v) \ u) \]

We can also build, for any \( T \)-algebra \( \alpha : T \ X \rightarrow X \) a map \( i_\alpha : A \rightarrow X \) by \( h = [x : A] x \ X \ f \).

Let us define \( g \circ f \) as \([x : X] g \ (f \ x)\) for \( f : X \rightarrow Y \) and \( g : Y \rightarrow Z \). If we have two \( T \)-algebras \( \alpha : T \ X \rightarrow X \) and \( \beta : T \ Y \rightarrow Y \), we can say that a map \( f : X \rightarrow Y \) is a judgemental \( T \)-morphism if \( f \circ \alpha \) and \( \beta \circ (T \ f) \) are convertible. One can then check that the map \( i_\alpha \) is a judgemental \( T \)-morphism from intro to \( \alpha \).

Using mon again, we get an element \( T \ \text{intro} \) of type \( T \ (T \ A) \rightarrow T \ A \) and so a function of type \( \text{match} : A \rightarrow T \ A \), which is a judgemental \( T \)-morphism from intro to \( T \ \text{intro} \). These are some steps in Lambek’s Theorem [57] for building the initial algebra of an endofunctor, and we think that these notations are well adapted to express what is going on.

As discovered by G. Wraith [105], such an encoding also works for coinductive definitions. The “weak” final coalgebra for \( T \) can be encoded as \( \exists \text{type} X \rightarrow T \ X \) where \( \exists \text{type} P(X) \) denotes

\[ [Y : \text{type}] \ ([X : \text{type}] \ P(X) \rightarrow Y) \rightarrow Y \]

One could use existential type, such as \( \exists \text{type} P(X) \), for representing abstract data types [73].

One could also form types such as

\[ [X : \text{type}] \ ((X \rightarrow X) \rightarrow X) \rightarrow X \]

which can be thought of as a type of de Bruijn indices, or the type

\[ [X : \text{type}] \ ((X \rightarrow X) \rightarrow (X \rightarrow X)) \rightarrow (X \rightarrow X \rightarrow X) \rightarrow X \]

which represents a type of syntax for pure \( \lambda \)-terms [27].

### 4 Consistency and Paradoxes

Is the calculus consistent? If we look at it as a logical system, this amounts to the question whether or not we can find a proof of \( \bot = [X : \text{type}] X \).

This is a direct consequence of normalisation and subject reduction 11, since it is clear that there is no term of type \( \bot \) in normal form. The normalisation proof is quite subtle [19, 99]. As explained by Girard [40], normalisation implies consistency of higher order arithmetic.

I found later [20] that there should be a finitary proof of consistency, by interpreting type by the finite set \( \{0, 1\} \), with a truth-table interpretation of types/propositions. This is however not completely trivial, and it involves a non standard encoding of functions in set theory described later in [1] (see also [71]).

Consistency was however a little surprising at first since the calculus was very close to Martin-Löf system with type : type [62] which was shown to be inconsistent by Girard [41]. This paradox involved the collection of all well-ordered relations which we can define using the encoding

\[ \exists \text{type} P(X) = [Y : \text{type}] \ ([X : \text{type}] \ P(X) \rightarrow Y) \rightarrow Y \]

\[ ^{11} \text{If } M : A \text{ and } M \text{ reduces to } M' \text{ by a sequence of } \beta \text{-reduction then } M' : A. \]
Looking at this argument \[41\], it was clear \[19\] that one could reproduce this paradox if we could encode the two projections

\[
\pi_1: [z : \exists X \text{type} P(X)] \rightarrow \text{type} \quad \pi_2: [z : \exists X \text{type} P(X)] P(\pi_1 z)
\]

So a corollary of the consistency proof was that it was not possible to have such encoding. Consistency of the present calculus is closely connected to the fact that we cannot encode “strong” sums, following W. Howard’s terminology \[49\]. This can also be seen as an indirect justification of the impredicative type:

So a corollary of the consistency proof was that it was not possible to have such encoding. Consistency of the present calculus is closely connected to the fact that we cannot encode “strong” sums, following W. Howard’s terminology \[49\]. This can also be seen as an indirect justification of the impredicative representation of abstract data types \[73\].

We still can define the introduction rule

\[
i : [X : \text{type}] P(X) \rightarrow \exists X \text{type} P(X) = [X : \text{type}] [h : P(X)] [Y : \text{type}] [h_1 : [X : \text{type}] P(X) \rightarrow Y] h_1 X h
\]

Girard was actually not using the two projections but the following “key” remark instead: if \(S = \exists X \text{type} P(X)\), thought of as a type of “structures” (in his case, well-ordering), and if we have a proof of

\[
eq S (i X_1 p_1) (i X_2 p_2)
\]

where \(\equiv\) is Leibnitz’ equality, then the two structures \(X_1, p_1\) and \(X_2, p_2\) are isomorphic\(^\text{12}\). It is interesting that this remark is a crucial component in the formulation of the axiom of univalence \[101\].

4.1 Girard’s Paradox

Another surprising feature of the paradox discovered by Girard for the type : type system was that Martin-Löf had previously found a formally correct proof of normalisation for this system \[62\]. How was this possible?

One application of Girard’s elegant normalisation proof for system \(F\) \[40\] was a solution of Takeuti’s conjecture \[96\]. These works were direct descendants of the debate between Russell and Poincaré \[89, 81\] about the status of impredicativity. Takeuti’s conjecture expressed cut-elimination for a sequent calculus formulation of higher-order logic. This is a quite surprising conjecture, since cut-elimination proofs are usually done by induction on the cut formula, but, with impredicativity, such induction is not possible. For instance, the polymorphic identity of type \(T = \exists X \text{type} X \rightarrow X\) can itself be applied to a more complex type than its own type, e.g. \(T \rightarrow T\). This “circularity” illustrates further the special status of impredicative definitions. Also, normalisation of impredicative system \(F\), or cut-elimination of Takeuti’s system, implies consistency of the strong system of second-order arithmetic \[40\]. Girard’s normalisation proof and design of system \(F\) were direct motivations for the formulation of a type system with type : type by Martin-Löf, and its normalisation proof \[62\].

Girard found his paradox not by looking directly at a contradiction in a system with type : type, but by looking instead at a “logical” extension of system \(F\). In the same way that higher-order logic, as formulated by Church \[14\], extends simple type theory by a type of propositions, constants for connectives and quantifiers, and some logical rules, it is quite natural to look for a similar extension of system \(F\), adding a type of propositions, and this was the calculus Girard was considering.

With the present notation, this amounts to add a new sort \(\text{prop} : \text{type}\). Girard added then the following quantifications

1. \([X : \text{type}] B : \text{prop} \text{ if } B : \text{prop} \text{ for } X : \text{type}\)
2. \([x : A] B : \text{prop} \text{ if } A : \text{type} \text{ and } B : \text{prop} \text{ for } x : A\)
3. \(A \rightarrow B : \text{prop} \text{ if } A : \text{prop} \text{ and } B : \text{prop}\)

The resulting system was called System \(U\). One might expect this system to be inconsistent, since system \(F\) has no set theoretic semantics\(^\text{13}\). Girard defined then what he called System \(\neg U\), the system obtained by leaving the first quantification clause. He wrote that this system was “maybe consistent”. I found

\(^\text{12}\) One needs type : type for building \(Q_1 : S \rightarrow \text{type}\) such that \(Q_1 (i X p)\) expresses that \(X_1, p_1\) and \(X, p\) are isomorphic.

\(^\text{13}\) Since one can directly translate System \(U\) in the system with type : type by defining \(\text{prop} = \text{type}\), this implies that type : type is inconsistent as well.
out [23] that this was not the case: we get a contradiction already in System $U^{-}$ using only the two last clauses, looking at another paradox. I will now try to describe this result in more details.

This other paradox was a direct translation in this formal system of Reynolds’ Theorem [86] that there is no set theoretic model of system $F$. This was using the type

$$A : \text{type} = [X : \text{type}] (T \times X) \to X$$

for $T X = (X \to \text{prop}) \to \text{prop}$.

This a priori defines only a weak initial algebra. However, following the reasoning in [86], and essentially Bishop’s idea of interpreting a set as a type with an equivalence relation [10], one can define an equivalence relation $E$ on $A$ such that the “set” $A, E$ becomes isomorphic to the set $T A, T E$ where $T E$ is the equivalence relation on $T A$ induced from $E$ by the technique of logical relations [23]. We introduce

$$\text{rel} : \text{type} \to \text{type} = [X : \text{type}] X \times X \to \text{prop} \quad (\equiv) : \text{rel} \text{prop} = [p \ q : \text{prop}] (p \to q) \land (q \to p)$$

and we can define $\text{pow} : [X : \text{type}] \text{rel} X \times X \to \text{rel} (\text{Pow} X)$ by

$$\text{pow} = [X : \text{type}][E : \text{rel} X][P_0 \ P_1 : \text{Pow} X][x_0 \ x_1 : X] E x_0 \ x_1 \to (P_0 \ x_1) \equiv (P_1 \ x_1)$$

and $T E$ is defined to be $\text{pow} (\text{Pow} A) (\text{pow} A E)$.

One can then translate the set theoretic result that a set cannot be in bijection with the power set of its power set, and get a proof of $\bot$ [23].

In [53], A. Hurkens presents a short variation of this paradox, using

(1) $$[X : \text{type}] (T \times X) \to T X$$

instead of

(2) $$[X : \text{type}] (T X \to X) \to X$$

As it turned out, his argument can be used almost as such using the definition (2) as in [23] instead of (1). This is presented\(^{14}\) in Figure 1.

Looking at the formal system in which we express this paradox, one finds that this never uses the first quantification clause, and this thus answers Girard’s question: already System $U^{-}$ is inconsistent. I only realized that this was the case however by using the notion of Pure Type System introduced by H. Barendregt [6, 91], following S. Berardi PhD thesis [8]. With this notation, we can describe Systems

\(^{14}\)Technically, one has to work with partial equivalence relations, going back to ideas from Gandy’s PhD thesis [36].

\(^{15}\)If we perform head-reduction on loop, the head variable eventually has a periodic behavior, oscillating between $\text{lem}_1$ and $\text{lem}_3$ with growing types in some abstractions.
$U$ and $U^-$ in the following way. Both have 3 sorts: prop, type and type$_1$, and the same typing relations $\text{prop} : \text{type}$ and $\text{type} : \text{type}_1$. They differ for quantifications\(^{16}\):

System $U$ : $(\text{prop}, \text{prop})$, $(\text{type}, \text{prop})$, $(\text{type}, \text{type})$, $(\text{type}_1, \text{type})$, $(\text{type}_1, \text{prop})$

System $U^-$ : $(\text{prop}, \text{prop})$, $(\text{type}, \text{prop})$, $(\text{type}, \text{type})$, $(\text{type}_1, \text{type})$

When expressing Reynolds’ argument type-theoretically, we never need the rule $(\text{type}_1, \text{prop})$, i.e. we only use System $U^-$. This is a further illustration of the importance of introducing explicit names and notations for notions: the notion of Pure Type System provides a good explicit way to express the quantification structure of a formal system. Once we have named a notion (in this case use of the rule $(\text{type}_1, \text{prop})$) we can more easily notice if this notion appears or not.

As explain in [6], we can describe higher-order logic as the system with the same sorts and typing relations as for System $U$ but with the rules $(\text{prop}, \text{prop})$, $(\text{type}, \text{prop})$, $(\text{type}, \text{type})$.

In [20], I analysed another paradox, closer to Girard’s original formulation. It was expressed in System $U$, but, as was found out later by H. Herbelin and A. Miquel, a slight variation of this paradox is actually expressible in System $U^-$. I implemented this paradox and looked at its computational behavior [20]. At about the same time, A. Meyer and M. Reinholdt [70], suggested a clever use of Girard’s paradox for expressing a fixed-point combinator. Using my implementation, I could check that, contrary to what [70] was hinting, the term representing this paradox was not reducing to itself\(^{17}\). As it turned out, and as was advocated by A. Meyer, it was however possible to use this paradox and produce a family of looping combinators instead, i.e. a term which has the same Böhm tree as one of a fixed-point combinator [50, 26]. A corollary, following [70], is that type-checking is undecidable for type $: \text{type}$.

### 4.2 Parametricity and Normalisation Proofs

How was it possible for Martin-Löf to have a normalisation proof for type $: \text{type}$, which implies consistency, while this system is contradictory? The answer is simple: Martin-Löf was formulating his proof using as meta-language a system which itself had a type of all types.

This is a general phenomenon. One can argue that the most elegant way to prove normalisation for type systems following Tait/Girard computability methods is to use as meta-language a type system which is as close as possible to the object system itself. (This applies as well to the predicative version of type theory [64, 24].) This also points out to a “weakness” of such consistency proof: to be conclusive, it has to rely on the consistency of the meta-language\(^{18}\). An inconsistency, on the other hand, is something concrete and witnessed by a term of type $\bot = [X : \text{type}]X$.

These proofs of normalisation are also very close formally to proofs of parametricity. The work [9] presents an elegant formulation of parametricity, as a purely internal syntactic translation of the system into itself. For instance the fact that the polymorphic identity function

$$\text{id} : [X : \text{type}]X \rightarrow X = [X : \text{type}][x : X]x$$

is parametric is expressed by a term of type

$$[X : \text{type}][X' : X \rightarrow \text{type}][x : X]X' x \rightarrow X' (\text{id} X x)$$

which is

$$\text{id}' = [X : \text{type}][X' : X \rightarrow \text{type}][x : X][x' : X']x'. $$

This can be seen as a syntactical counterpart of the ingenious notion of “reducibility candidate”, introduced by Girard to prove normalisation of system $F$ [40]. In general, a term $M$ is transformed to a term $M'$ and we have $M' : A' \ M$ if $M : A$.

What is remarkable is that such a transformation works as well for type $: \text{type}$, defining type$' : \text{type} \rightarrow \text{type}$ to be type$' = [X : \text{type}]X \rightarrow \text{type}$. This observation can be seen as underlying Martin-Löf’s (formally correct) normalisation proof for type $: \text{type}$.

The paper [24], simplifying and generalizing in some way [64], presents a normalisation proof for a predicative version of type theory, similar to this parametricity interpretation, which works for a cumulative hierarchy of universes with $\beta, \eta$-conversion.

\(^{16}\)E.g. to have the rule $(\text{type}, \text{prop})$ means that, if $A : \text{type}$, then we have $[x : A]B : \text{prop}$ if $B : \text{prop}$ for $x : A$.

\(^{17}\)Intuitively, if one sees head-reduction as a process of understanding a proof, the paradox was becoming more and more complex while trying to understand it!

\(^{18}\)This should be connected to the distinction between metamathematical and simple minded consistency proofs in [68].
5 Consistency and expressiveness

The work on paradoxes has shown, roughly speaking, that we cannot have a consistent system with two levels of impredicative universes. Instead, as suggested in [20], we should extend the system with a hierarchy of predicative universes $\text{type}_1 : \text{type}_2 : \ldots$ with quantifications $[x : A]B : \text{type}_n$ if $A : \text{type}_n$ and $B : \text{type}_n$ for $x : A$. It is then natural to rename the base impredicative sort as $\text{prop}$. We have $[x : A]B : \text{prop}$ if $B : \text{prop}$ for $x : A$, without any condition on $A$. This is a quite natural extension of higher-order logic.

A remark is that one can prove the axiom of infinity in such a system. Indeed, if we redefine

$$\text{nat} : \text{type}_1 = [X : \text{type}_1] X \to (X \to X) \to X$$

we have

$$\text{zero} : \text{nat} = [X : \text{type}_1][x : X][f : X \to X] x$$

$$\text{succ} : \text{nat} \to \text{nat} = [n : \text{nat}][X : \text{type}_1][x : X][f : X \to X] f (n \ X \ x \ f).$$

We can then build terms of type $[n : \text{nat}] \neg (\text{eq } \text{nat } \text{zero } \text{(succ } n))$ and

$$[n \ m : \text{nat}] \text{eq nat } \text{(succ } n) \ (\text{succ } m) \to \text{eq nat } n \ m$$

which provide a formulation of the axiom of infinity. This is somewhat surprising since, for instance, Russell thought that to have a purely logical proof of this axiom of infinity should be impossible [89].

A natural question is how this system compare with set theory? I conjectured [20] that it should be stronger than Zermelo set theory. This question was solved in an elegant way by A. Miquel [71, 72].

The idea is first to encode pointed graphs as binary relations. Since, as we have seen, co-induction is definable in an impredicative system, it is possible to define bissimulation of pointed graphs. A set can then be encoded as a pointed graph up to bissimulation. We obtain in this way a model of non well-founded set theory, and one can check [71] that all the axioms of Zermelo set theory are satisfied. Using [26], a double negation interpretation gives an interpretation of set theory with classical logic [71]. This is refined in [72], which shows that the system with only $\text{prop}$, $\text{type}_1$, $\text{type}_2$ is equiconsistent with Zermelo’s set theory.

6 Intensional Expressiveness and Inductive Definitions

While the class of numerical functions representable in system $F$ is quite large, since it coincides with the class of functions provably total in second-order arithmetic, one can further ask if such representations are good in an “intensional” way. The representation of the predecessor function, for instance, though possible, does not look so natural. This study was initiated by J.-L. Krivine [56], who gave an example of a quite natural algorithm for comparing two natural numbers that does not have a good intensional representation in system $F$.

One other problem is that, if we define $\text{nat} = [X : \text{type}] X \to (X \to X) \to X$, then it is not possible to prove the induction principle\(^{19}\)

$$[P : N \to \text{type}] P \ 0 \to ([x : N] \ P \ x \to P \ (S \ x)) \to [n : N] \ P \ n$$

One general solution [19] was to restrict oneself to natural numbers satisfying the following predicate, simply obtained by shifting the abstraction $n : N$

$$C : N \to \text{type} = [n : N][P : N \to \text{type}] P \ 0 \to ([x : N] \ P \ x \to P \ (S \ x)) \to P \ n$$

by a technique similar to the internalisation of parametricity.

What was more problematic was that, even in this way, it was not possible\(^{20}\) to prove

$$[x : \text{nat}] \ C \ x \to \neg (\text{eq nat } \text{zero } \text{(succ } x))$$

\(^{19}\)The system Cedille [93] suggests another encoding than Church encoding of natural numbers where we have a nice representation of primitive recursion. This alternative representation is also used in D. Friflender’s work [35]. It is a curious remark that for the data type $\perp = [A : \text{type}]A$, it is possible to prove the induction principle $[x : \perp]R \ x$ for $R : \perp \to \text{type}$.

\(^{20}\)We saw above that such a proof is possible in an extension with predicative universes and a predicative encoding of natural numbers.
as shown by the “truth-table” model where we interpret type by \{0, 1\}.

H. Geuvers has even proved that we cannot find an encoding of natural numbers where induction principle is provable [38].

Yet another problem was that, with this definition of natural numbers, it was not possible to have large eliminations, e.g. to define a function \(f : \text{nat} \to \text{type}\) such that \(f \text{ zero} = \text{nat}\) and \(f (\text{succ } n) = (f n) \to \text{nat}\).

All these remarks suggested to extend the system by adding data types with computation rules as primitives, like in Martin-Löf system [63, 64]. This extension was also motivated the work of Constable and N. Mendler [18], that I reformulated without using a subtyping relation. This was carried out in [21] [29]. It was then possible to further extend this system with a predicative hierarchy of universes [22]. We then get a system weaker than Zermelo-Fraenkel, as shown by M. Rathjen [83].

To extend the system with primitive data types and computation rules is definitively less elegant than trying to derive them from more primitive notions, and it is not yet clear how to represent and implement this extension without introducing some arbitrary syntactical conditions. The work [21] was motivated by these questions, trying to see type theory as a total fragment of a programming language with dependent type and definitions by pattern-matching. This stressed further the close connections between notions used in functional programming and in proof theory, for instance:

- constructors of a data type correspond to introduction rules,
- proofs by induction correspond to case analysis and recursive definitions of functions,
- derived rules are represented as (maybe recursively) defined constants,
- \textit{where} expressions correspond to lemmas local to a proof
- \textit{pattern-matching} corresponds to Lorenzen’s inversion principle [61].

We refer to J. Cockx’s paper [16] for recent work in this direction.

Conclusion

This work can be seen as a synthesis of the work on AUTOMATH and Martin-Löf type theory. From AUTOMATH, it uses the notion of context and the idea of reducing proof-checking to type-checking. From Martin-Löf system, we use the idea of having data types as primitive, and the correspondence between the notion of constructors and introduction rules. Thanks to G. Huet, these ideas could be connected to the active development of functional programming in the 80s, and this approach was the basis of several implementations of proof systems that have been quite successful [23] and have been tested on non trivial examples, both in formalisation of mathematics [44, 45, 13] and in checking correctness of software [60, 12]. It was also relevant for initiating important works on the semantics of type theory, such as [92, 32, 1], and more recently [78, 94].

One real surprise was that this language, with its uniform treatment of propositions and types, which we found so well suited for proofs in higher-order logic, has turned out to be quite convenient for expressing concepts from homotopy theory and higher category theory, such as the axiom of univalence [100, 101].

\[ \text{sup : } [X : \text{type}](X \to U) \to U \]

since this type has an element \(\text{sup } U (\{x : U|x\})\) which is both well-founded and has itself has a subtree [22].

21 An example of a proof using inductive definitions in this paper was Wainer’s proof of Girard’s Theorem on functional hierarchy [100], where I used an inductively definition relation instead of a relation defined by recursion as in [103].

22 If we allow inductive definitions as primitives, Girard’s paradox becomes very simple, using the type \(U\) with a constructor of type

23 It should be stressed that the main component in these successes has been an enduring impressive collaborative work on software development of type theoretic systems.
Appendix

\[
\Gamma ::= () | \Gamma[x:A] \\
M, A ::= \text{type} | x | [x:A]M | M M \\
C ::= \text{type} | [x:A]C
\]

\[
\begin{align*}
\Gamma &\vdash C & \Gamma &\vdash A : \text{type} \\
() &\vdash \text{type} & \Gamma[x:C] &\vdash \text{type} & \Gamma[x:A] &\vdash \text{type} \\
\Gamma[x:A] &\vdash B : \text{type} & \Gamma[x:A] &\vdash C \\
\Gamma &\vdash [x:A]B : \text{type} & \Gamma &\vdash [x:A]C \\
\Gamma &\vdash \text{type} & (x : A \text{ in } \Gamma) &\vdash x : A \\
\Gamma &\vdash [x:A]M : [x:A]B \\
\Gamma &\vdash N : [x:A]B & \Gamma &\vdash M : A \\
\Gamma &\vdash N M : B(M/x)
\end{align*}
\]

Terms are considered up to \(\beta\)-conversions. For study of systems with \(\eta\)-conversions see [39, 7, 24, 25].
References


[100] V. Voevodsky. The equivalence axiom and univalent models of type theory. notes from a talk at CMU, 2010.


