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 λ-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 λ-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[1].

System F extends simply typed λ-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 α is a type variable. This system was quite mysterious: the parametric identity function[2] $\Lambda\alpha\lambda x^\alpha x^\alpha$ 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 λ-calculus. In particular, there was a non obvious restriction on typed variables: for forming $\Lambda\alpha M$ we should have that α 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_ω 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 λ-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

[1] For 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.

[2] The 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 mathematicien 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 $g(\pi) > \pi^5$. He can then state $f(\pi) > 5$ by *applying* the theorem and by giving *in this order*: π, 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 λ-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 $\mathsf{real} : \mathsf{type}$ and $\mathsf{nat} : \mathsf{type}$ for types of real numbers and natural numbers.

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

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

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

One recovers ordinary implication/function type $A \to B$ 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 λ-calculus which is β-conversion.

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

$$[A : \mathsf{type}][x : A][f : [z : A]A]([z : A]A)\ x$$

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

1.3 Representation of matematical notions in AUTOMATH

One of the first example represented in the system [30] was the notion of equality on a type $A : \mathsf{type}$. This is represented by a collection of constants. One constant $\mathsf{Eq} : [x\ y : A]\mathsf{type}$ represents the notion of equality itself: if a_0 and a_1 are elements of type A, then $\mathsf{Eq}\ a_0\ a_1$ 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

$$\mathsf{refl} : [x : A]\mathsf{Eq}\ x\ x \qquad\qquad \mathsf{eucl} : [x\ y\ z : A]\mathsf{Eq}\ x\ z \to \mathsf{Eq}\ y\ z \to \mathsf{Eq}\ x\ y$$

We can then form the term

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

which is a proof that the relation Eq is symmetric.

This example illustrates the uniform treatment of elements and proofs.

The following other simple 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 $\neg\ A = A \rightarrow \perp$, where $\perp$: type represents the false proposition. It is then no problem to build an element

$$[x : A][f : A \rightarrow \perp]f\ x \quad : \quad A \rightarrow \neg\neg A$$

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 $(A \rightarrow \perp) \rightarrow \perp$.

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_ω

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

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

$$[A : \mathsf{type}][h : \perp]h\ A \quad : \quad [A : \mathsf{type}]\ \perp \rightarrow A$$

Note that we have several proofs of $\perp \rightarrow \perp$, for instance $[x : \perp]x$ or $[x : \perp]x\ \ \perp$ or or $[x : \perp]x\ \ \perp\ \ x$ or $[x : \perp][A : \mathsf{type}]x\ A$. (This last term can be seen as the η-expansion of $[x : \perp]x$.)

We also can represent the notion of equality, not by introducing a new constant, but as a definition, as "Leibnitz' equality"

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

We then define a term which represents the proof of reflexivity

$$\mathsf{refl}\ \ :\ \ [A : \mathsf{type}][x : A]\mathsf{eq}\ A\ x\ x \quad = \quad [A : \mathsf{type}][x : A][P : A \rightarrow \mathsf{type}][h : P\ x]h$$

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

One crucial point is that we don't require[5] type to be of type type; however, like in system F, the type $[A : \mathsf{type}]A$ is of type type. It is possible to define an equality on type, but this is not an instance of eq

$$\mathsf{eq}_1\ \ : \mathsf{type} \rightarrow \mathsf{type} \rightarrow \mathsf{type} \quad = \quad [A\ B : \mathsf{type}][P : \mathsf{type} \rightarrow \mathsf{type}]P\ A \rightarrow P\ B$$

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 $\Lambda\alpha M$ one had to be careful that α was not appearing free in the type of some free variables of M. In this other presentation, using this idea of treating uniformely variable type declarations and variable term declarations, this becomes that we always can form $[A : \mathsf{type}]M$ in a context Γ if M is correct in the context $\Gamma, A : \mathsf{type}$.

[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

$$(\wedge) = [A\ B : \mathsf{type}][X : \mathsf{type}](A \to B \to X) \to X$$
$$(\vee) = [A\ B : \mathsf{type}][X : \mathsf{type}](A \to X) \to (B \to X) \to X$$

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

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

2.3 Encoding of data types

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

$$\mathsf{bool} \ = \ [A : \mathsf{type}]A \to A \to A \qquad \mathsf{nat} \ = \ [A : \mathsf{type}]A \to (A \to A) \to A$$

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

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

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

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

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]

$$\mathsf{list} : \mathsf{type} \to \mathsf{type} \ = \ [A : \mathsf{type}][X : \mathsf{type}]X \to (A \to X \to X) \to X$$

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_ω, 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[6] 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[7].

[6]Martin-Löf had explored an intensional approach [64], and later with P. Hancock [65], which even excluded the ξ-rule, but this was abandonned 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.

[7]There is a discussion at the end of the paper, allunding 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 AUTOMATH's approach, and to prove that type-checking is decidable, as a corollary of a normalisation theorem, which extends the normalisation theorem for system F_ω [19, 99]. The implementation was not trivial however, and AUTOMATH's checking program actually had a wrong treatment of bound variables [98]. The first type checker for the system we describe was written by G. Huet [27], making use of the recent addition of algebraic data types and pattern-matching in ML by G. Cousineau [59]. This implementation was essential in order to test this formal system on examples.

Type-checking reduces to conversion, by building a list of conversion constraints that need to hold for a term to be well-typed. This algorithm adapts naturally for building a proof interactively with place-holders, as was implemented in the work [4].

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 $[x : C]A : \mathsf{type}$ for $A : \mathsf{type}$ in the context $x : C$, and where C is itself of the form $[x_1 : A_1] \ldots [x_n : A_n]\mathsf{type}$ (we can have $n = 0$ in which case C is type itself).

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

$$\mathsf{eq} \quad = \quad [A : \mathsf{type}][x\ y : A][P : A \to \mathsf{type}]P\ x \to P\ y$$

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

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

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

$$[P : A \to \mathsf{type}]P\ z \to P\ x \ \ : \ \ \mathsf{type}$$

There is however another *predicative* proof of the symmetry of eq, which is

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

A simpler example is $\bot{\to}\bot$, which has both an impredicative proof $[x :\bot]x \quad \bot$ and a predicative proof $[x :\bot][A : \mathsf{type}]x\ A$. 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

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

We can program various functional hierarchies $\mathsf{ord} \to (\mathsf{nat} \to \mathsf{nat})$, simply by instantianting $A : \mathsf{type}$ by $\mathsf{nat} \to \mathsf{nat}$. For instance, the Hardy hierarchy $h : \mathsf{ord} \to \mathsf{nat} \to \mathsf{nat}$ 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 ord corresponding to various ordinals ω, ω^ω, $\epsilon_0, \ldots$ [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 $(\mathsf{ord} \to A) \to A$. One can even consider the type (Huet, [27])

$$\mathsf{ord}_\infty = [A : \mathsf{type}]\ ([B : \mathsf{type}]\ (B \to A) \to A) \to A$$

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 example 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: $\varphi \to \forall x\ \psi(x)$ if $\varphi \to \psi(x)$ *and* x is not free in φ. 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 $[x : A]M$ is correct in the context Γ, 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[8] relation defines a linear order. As he emphasized, it is surprising that we can "bring forth judgements that at first sight appear to be possible only on the basis of some intuitions".

Here was Frege's definition of the transitive closure of a relation, using AUTOMATH notations, in the context $A : \mathsf{type},\ R : A \to A \to \mathsf{type}$

$$\begin{array}{lcllcl}
\mathsf{her} & : & (A \to \mathsf{type}) \to \mathsf{type} & = & [P : A \to \mathsf{type}][x : A]\ P\ x \to [y : A]\ R\ x\ y \to P\ y \\
R^+ & : & A \to A \to \mathsf{type} & = & [x\ y : A][P : A \to \mathsf{type}]\ \mathsf{her}\ P\ \to ([z : A]\ R\ x\ z \to P\ z) \to P\ y
\end{array}$$

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

$$\begin{array}{lcl}
\mathsf{lem}_1 & : & [x : A]\ \mathsf{her}\ (R^+\ x) \\
 & = & [x\ y : A][h_1 : R^+\ x\ y][z : A][h_2 : R\ y\ z] \\
 & & [P : A \to \mathsf{type}][h_3 : \mathsf{her}\ P][h_4 : [u : A]R\ x\ u \to P\ u]\ h_3\ y\ (h_1\ P\ h_3\ h_4)\ z\ h_2 \\
\mathsf{lem}_2 & : & [x\ y\ z : A]\ R^+\ x\ y \to R^+\ y\ z \to R^+\ x\ z \\
 & = & [x\ y\ z : A][h_1 : R^+\ x\ y][h_2 : R^+\ y\ z]\ h_2\ (R^+\ x)\ (\mathsf{lem}_1\ x)\ (\mathsf{lem}_1\ x\ y\ h_1)
\end{array}$$

To have a notation for proofs can make explicit some interesting phenomena: for instance, the proof of lem_2 uses twice lem_1 but in different contexts, and the proof of 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 β-reduction mechanism of λ-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 $A \to A \to \mathsf{type}$. 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[9]. In particular, to express *definition* by *convertibility* seemed to be preferable in this respect than representing explicitely

[8] A relation R on a type A is *functional* if $R\ x\ y$ and $R\ x\ z$ imply $\mathsf{eq}\ A\ y\ z$.

[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 $R : A \to A \to \mathsf{type}$, in higher-order logic. One other encoding of the transitive closure is the following

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

where $\mathsf{trans}\ =\ [A : \mathsf{type}][S : A \to A \to \mathsf{type}][a\ b\ c : A]\ S\ a\ b \to S\ b\ c \to S\ a\ c$. 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

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

instead of

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

and one can show that equivalence between $\mathsf{eq}\ A\ x\ y$ and $\mathsf{Eq}\ A\ x\ y$.

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

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

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

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

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

$[A : \mathsf{type}]\ A \to P\ A$
$[A : \mathsf{type}][B : \mathsf{type}]\ (A \to P\ B) \to P(A \to B)$
$[A : \mathsf{type}][B : \mathsf{type}]\ P\ (A \to B) \to P\ A \to P\ B$
$[A : \mathsf{type}][C : \mathsf{type} \to \mathsf{type}]\ ([A : \mathsf{type}]\ P(C\ A)) \to P([A : \mathsf{type}]\ C\ A)$
$[A : \mathsf{type}][C : \mathsf{type} \to \mathsf{type}]\ P([A : \mathsf{type}]\ C\ A) \to [A : \mathsf{type}]\ P(C\ A)$

This predicate $P : \mathsf{type} \to \mathsf{type}$ 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

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

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*

$$[A : \mathsf{type}]\ A \to A \simeq A \to A$$

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

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

as a "weak" initial algebra for an operation $T : \mathsf{type} \to \mathsf{type}$.

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

$$[X\ Y : \mathsf{type}]\ (X \to Y) \to (T\ X \to T\ Y)$$

we can build an element of type $T\ A \to A$

$$[u : T\ A][X : \mathsf{type}][v : T\ X \to X]\ v\ (\mathsf{mon}\ ([a : A]\ a\ X\ v)\ u)$$

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

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

Using mon again, we get an element $T\ \mathsf{intro}$ of type $T\ (T\ A) \to T\ A$ and so a function of type $\mathsf{match} : A \to T\ A$, which is a judgemental T-morphism from intro to $T\ \mathsf{intro}$. 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 $\exists_{X:\mathsf{type}} X \to T\ X$ where $\exists_{X:\mathsf{type}} P(X)$ denotes

$$[Y : \mathsf{type}]\ ([X : \mathsf{type}]\ P(X) \to Y) \to Y$$

One could use existential type, such as $\exists_{X:\mathsf{type}} P(X)$, for representing *abstract data types* [73].

One could also form types such as

$$[X : \mathsf{type}]\ ((X \to X) \to X) \to X$$

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

$$[X : \mathsf{type}]\ ((X \to X) \to X) \to (X \to X \to X) \to X$$

which represents a type of syntax for pure λ-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 $\bot = [X : \mathsf{type}]X$.

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 $\mathsf{type} : \mathsf{type}$ [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

$$\exists_{X:\mathsf{type}} P(X) \quad = \quad [Y : \mathsf{type}]\ ([X : \mathsf{type}]\ P(X) \to Y) \to Y$$

[11] If $M : A$ and M reduces to M' by a sequence of β-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

$$\pi_1 : [z : \exists_{X:\mathsf{type}} P(X)]\ \mathsf{type} \qquad\qquad \pi_2 : [z : \exists_{X:\mathsf{type}} P(X)]\ P(\pi_1\ z)$$

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 representation of abstract data types [73].

We still can define the introduction rule

$$i : [X : \mathsf{type}]\ P(X) \rightarrow \exists_{X:\mathsf{type}} P(X) \ = \ [X : \mathsf{type}][h : P(X)][Y : \mathsf{type}][h_1 : [X : \mathsf{type}]P(X) \rightarrow Y]\ h_1\ X\ h$$

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

$$\mathsf{eq}\ S\ (i\ X_1\ p_1)\ (i\ X_2\ p_2)$$

where eq is Leibnitz' equality, then the two structures X_1, p_1 and X_2, p_2 are isomorphic[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 $\mathsf{type} : \mathsf{type}$ 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 $T = [X : \mathsf{type}]\ X \rightarrow X$ 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 $\mathsf{type} : \mathsf{type}$ by Martin-Löf, and its normalisation proof [62].

Girard found his paradox not by looking directly at a contradiction in a system with $\mathsf{type} : \mathsf{type}$, 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 $\mathsf{prop} : \mathsf{type}$. Girard added then the following quantifications

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

The resulting system was called System U. One might expect this system to be inconsistent, since system F has no set theoretic semantics[13]. Girard defined then what he called System U^-, the system obtained by leaving the first quantification clause. He wrote that this system was "maybe consistent". I found

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

[13] Since one can directly translate System U in the system with $\mathsf{type} : \mathsf{type}$ by defining $\mathsf{prop} = \mathsf{type}$, this implies that $\mathsf{type} : \mathsf{type}$ is inconsistent as well.

Pow	:	$\mathsf{type} \to \mathsf{type}$	=	$[X : \mathsf{type}]X \to \mathsf{prop}$
T	:	$\mathsf{type} \to \mathsf{type}$	=	$[X : \mathsf{type}]Pow\ (Pow\ X)$
U	:	type	=	$[X : \mathsf{type}](T\ X \to X) \to X$
intro	:	$T\ U \to U$	=	$[t : T\ U][X : \mathsf{type}][f : T\ X \to X]f\ ([g : Pow\ X]t\ ([z : U]g\ (z\ X\ f)))$
match	:	$U \to T\ U$	=	$[z : U]z\ (T\ U)\ ([t : T\ (T\ U)][g : Pow\ U]t\ ([x : T\ U]g\ (\mathsf{intro}\ x)))$
δ	:	$U \to U$	=	$[z : U]\mathsf{intro}\ (\mathsf{match}\ z)$
Q	:	$T\ U$	=	$[p : Pow\ U][z : U]\mathsf{match}\ z\ p \to p\ z$
A	:	$Pow\ U$	=	$[z : U][p : Pow\ U]\mathsf{match}\ z\ p \to p\ (\delta\ z)$
B	:	$Pow\ U$	=	$[z : U]\neg\ (A\ z)$
lem_1	:	$Q\ B$	=	$[z : U][k : \mathsf{match}\ z\ B][l : A\ z]l\ B\ k\ ([p : Pow\ U]l\ ([z_1 : U]p\ (\delta\ z_1))$
D	:	prop	=	$[p : Pow\ U]Q\ p \to p\ (\mathsf{intro}\ Q)$
lem_2	:	$\neg\ D$	=	$[h : D]h\ B\ lem_1\ ([p : Pow\ U]h\ ([z : U]p\ (\delta\ z))$
lem_3	:	D	=	$[p : Pow\ U][h : Q\ p]h\ (\mathsf{intro}\ Q)\ ([x : U]h\ (\delta\ x))$
$loop$	:	$\bot$	=	$lem_2\ lem_3$

Figure 1: A variation of Reynolds and Hurkens Paradox

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

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

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

This a priori defines only a *weak* initial algebra. However, following the reasoning in [86], and essentially[14] 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

$$\mathsf{rel} : \mathsf{type} \to \mathsf{type} = [X : \mathsf{type}]\ X \to X \to \mathsf{prop} \qquad (\equiv) : \mathsf{rel}\ \mathsf{prop} = [p\ q : \mathsf{prop}]\ (p \to q) \wedge (q \to p)$$

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

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

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

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

$$[X : \mathsf{type}]\ (T\ X \to X) \to T\ X \tag{1}$$

instead of

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

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[15] 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 lem_1 and 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 $\mathsf{prop} : \mathsf{type}$ and $\mathsf{type} : \mathsf{type}_1$. They differ for quantifications[16]:

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

When expressing Reynolds' argument type-theoretically, we never need the rule $(\mathsf{type}_1, \mathsf{prop})$, 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 $(\mathsf{type}_1, \mathsf{prop})$) 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 $(\mathsf{prop}, \mathsf{prop}),\ (\mathsf{type}, \mathsf{prop}),\ (\mathsf{type}, \mathsf{type})$.

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 $\mathsf{type} : \mathsf{type}$.

4.2 Parametricity and Normalisation Proofs

How was it possible for Martin-Löf to have a normalisation proof for $\mathsf{type} : \mathsf{type}$, 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 $\perp = [X : \mathsf{type}]X$.

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

$$\mathsf{id} : [X : \mathsf{type}]X \to X = [X : \mathsf{type}][x : X]x$$

is parametric is expressed by a term of type

$$[X : \mathsf{type}][X' : X \to \mathsf{type}][x : X]X'\ x \to X'\ (\mathsf{id}\ X\ x)$$

which is

$$\mathsf{id}' = [X : \mathsf{type}][X' : X \to \mathsf{type}][x : X][x' : X'\ x]x'.$$

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 $\mathsf{type} : \mathsf{type}$, defining $\mathsf{type}' : \mathsf{type} \to \mathsf{type}$ to be $\mathsf{type}' = [X : \mathsf{type}]X \to \mathsf{type}$. This observation can be seen as underlying Martin-Löf's (formally correct) normalisation proof for $\mathsf{type} : \mathsf{type}$.

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 β, η-conversion.

[16] E.g. to have the rule $(\mathsf{type}, \mathsf{prop})$ means that, if $A : \mathsf{type}$, then we have $[x : A]B : \mathsf{prop}$ if $B : \mathsf{prop}$ 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 $\mathsf{type}_1 : \mathsf{type}_2 : \ldots$ with quantifications $[x : A]B : \mathsf{type}_n$ if $A : \mathsf{type}_n$ and $B : \mathsf{type}_n$ for $x : A$. It is then natural to rename the base impredicative sort as prop. We have $[x : A]B : \mathsf{prop}$ if $B : \mathsf{prop}$ 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

$$\mathsf{nat} : \mathsf{type}_2 = [X : \mathsf{type}_1]\ X \to (X \to X) \to X$$

we have $\mathsf{zero} : \mathsf{nat} = [X : \mathsf{type}_1][x : X][f : X \to X]\ x$ and

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

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

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

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 prop, type_1, 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 $\mathsf{nat} = [X : \mathsf{type}]\ X \to (X \to X) \to X$, then it is not possible to prove the induction principle[19]

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

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

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

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

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

[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. Fridlender's work [35]. It is a curious remark that for the data type $\bot = [A : \mathsf{type}]A$, it *is* possible to prove the induction principle $[x :\bot]R\ x$ for $R :\bot \to \mathsf{type}$.

[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 $f : \mathsf{nat} \rightarrow \mathsf{type}$ such that $f\ \mathsf{zero} = \mathsf{nat}$ and $f\ (\mathsf{succ}\ n) = (f\ n) \rightarrow \mathsf{nat}$.

All these remarks suggested to extend the sytem 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,
-*where* expressions correspond to lemmas local to a proof
-*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 correspondance 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].

[21] An example of a proof using inductive definitions in this paper was Wainer's proof of Girard's Theorem on functional hierarchy [103], 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

$$\mathsf{sup} : [X : \mathsf{type}](X \rightarrow U) \rightarrow U$$

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

[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

$$\Gamma \quad ::= \quad () \mid \Gamma[x:A] \qquad M, A \quad ::= \quad \mathsf{type} \mid x \mid [x:A]M \mid M\ M \qquad C \quad ::= \quad \mathsf{type} \mid [x:A]C$$

$$\frac{}{() \vdash \mathsf{type}} \qquad \frac{\Gamma \vdash C}{\Gamma[x:C] \vdash \mathsf{type}} \qquad \frac{\Gamma \vdash A : \mathsf{type}}{\Gamma[x:A] \vdash \mathsf{type}}$$

$$\frac{\Gamma[x:A] \vdash B : \mathsf{type}}{\Gamma \vdash [x:A]B : \mathsf{type}} \qquad \frac{\Gamma[x:A] \vdash C}{\Gamma \vdash [x:A]C}$$

$$\frac{\Gamma \vdash \mathsf{type}}{\Gamma \vdash x : A}\ (x : A\ in\ \Gamma) \qquad \frac{\Gamma[x:A] \vdash M : B}{\Gamma \vdash [x:A]M : [x:A]B} \qquad \frac{\Gamma \vdash N : [x:A]B \qquad \Gamma \vdash M : A}{\Gamma \vdash N\ M : B(M/x)}$$

Terms are considered up to β-conversions. For study of systems with η-conversions see [39, 7, 24, 25].

References

[1] Peter Aczel. On Relating Type Theories and Set Theories. In T. Altenkirch, B. Reus, and W. Naraschewski, editors, *Types for Proofs and Programs*, pages 33–46. Springer, 1998.

[2] Peter Aczel. The Russell-Prawitz modality. *Math. Struct. Comput. Sci.*, 11(4):541–554, 2001.

[3] M. Algehed and J.-Ph. Bernardy. Simple noninterference from parametricity. *Proc. ACM Program. Lang.*, 3(ICFP):89:1–89:22, 2019.

[4] L. Augustsson, Th. Coquand, and B. Nordström. Another logical framework. In *Annual Workshop on Logical Frameworks, 1990*, 1990.

[5] S. Awodey, J. Frey, and S. Speight. Impredicative encodings of (higher) inductive types. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 76–85. ACM, 2018.

[6] Henk Barendregt. Introduction to Generalized Type Systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[7] G. Barthe and Th. Coquand. Remarks on the equational theory of non-normalizing pure type systems. *J. Funct. Program.*, 16(2):137–155, 2006.

[8] S. Berardi. *Type Dependence and Constructive Mathematics*. PhD thesis, University of Torino, 1989. PhD thesis.

[9] J.-Ph. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 345–356. ACM, 2010.

[10] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.

[11] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comput. Sci.*, 39:135–154, 1985.

[12] Sylvie Boldo and Guillaume Melquiond. *Computer arithmetic and formal proofs. Verifying floating-point algorithms with the Coq system*. Amsterdam: Elsevier/ISTE Press, 2017.

[13] K. Buzzard, J. Commelin, and P. Massot. Formalising perfectoid spaces. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 299–312. ACM, 2020.

[14] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[15] Alonzo Church. *The calculi of Lambda-Conversion*, volume 6. Princeton University Press, Princeton, NJ, 1941.

[16] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without K. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 257–268. ACM, 2014.

[17] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[18] Robert L. Constable and N. P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings*, volume 193 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 1985.

[19] Th. Coquand. *Une théorie des constructions*. Phd thesis, Université Paris VII, 1985.

[20] Th. Coquand. An Analysis of Girard's Paradox. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, pages 227–236. IEEE Computer Society, 1986.

[21] Th. Coquand. Pattern matching with dependent types. In *Annual Workshop on Logical Frameworks, 1992*, 1992.

[22] Th. Coquand. The Paradox of Trees in Type Theory. *BIT*, 32(1):10–14, 1992.

[23] Th. Coquand. A new paradox in type theory. In *Logic, methodology and philosophy of science IX. Proceedings of the ninth international congress of logic, methodology and philosophy of science, Uppsala, Sweden, August 7-14, 1991*, pages 555–570. Amsterdam: North-Holland, 1994.

[24] Th. Coquand. Canonicity and normalization for dependent type theory. *Theor. Comput. Sci.*, 777:184–191, 2019.

[25] Th. Coquand. Reduction free normalisation for a proof irrelevant type of propositions. *CoRR*, abs/2103.04287, 2021.

[26] Th. Coquand and H. Herbelin. *A*-translation and looping combinators in pure type systems. *J. Funct. Program.*, 4(1):77–88, 1994.

[27] Th. Coquand and G. P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *EUROCAL '85, European Conference on Computer Algebra, Linz, Austria, April 1-3, 1985, Proceedings Volume 1: Invited Lectures*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184. Springer, 1985.

[28] Th. Coquand and Gérard P. Huet. The Calculus of Constructions. *Inf. Comput.*, 76:95–120, 1988.

[29] Th. Coquand and Ch. Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[30] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. Sympos. Automatic Demonstrat., Versailles/France, 1968, Lect. Notes Math. 125, 29-61 (1970)., 1970.

[31] N. G. de Bruijn. Automath a language for mathematics. Seminaire de mathematiques superieures - ete 1971. No. 52. Montreal: Les Presses de l'Universite de Montreal, 1973.

[32] Th. Ehrhard. A categorical semantics of constructions. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 264–273. IEEE Computer Society, 1988.

[33] Steven Fortune, Daniel Leivant, and Michael O'Donnell. The expressiveness of simple and second-order type structures. *J. ACM*, 30(1):151–185, 1983.

[34] G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Nebert, 1879.

[35] Daniel Fridlender. A proof-irrelevant model of martin-löf's logical framework. *Math. Struct. Comput. Sci.*, 12(6):771–795, 2002.

[36] R. O. Gandy. *On axiomatic systems in mathematics and theories in physics*. PhD thesis, 1952. PhD thesis.

[37] Herman Geuvers. Conservativity between logics and typed lambda calculi. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 79–107. Springer, 1993.

[38] Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181. Springer, 2001.

[39] Herman Geuvers and Benjamin Werner. On the Church-Rosser Property for Expressive Type Systems and its Consequences for their Metatheoretic Study. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*, pages 320–329. IEEE Computer Society, 1994.

[40] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types.(An extension of Gödel's interpretation to analysis and its application to cut elimination in analysis and type theory). Proc. 2nd Scandinav. Logic Sympos. 1970, Studies Logic Foundations Math. 63, 63-92 (1971)., 1971.

[41] J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, 1972. Thèse de Doctorat d'Etat.

[42] J.-Y. Girard. The system $\mathcal{F}$ of variable types, fifteen years later. *Theor. Comput. Sci.*, 45:159–192, 1986.

[43] Kurt Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[44] G. Gonthier. The Four-Color Theorem. *Notices Amer. Math. Soc.*, 55:1382–1393, 2008.

[45] G. Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[46] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[47] D. Hilbert. Über das unendliche. *Mathematische Annalen*, 95:161–190, 1926.

[48] D. Hilbert. Die Grundlagen der Mathematik. *Abhandlungen aus dem Seminar der Hamburgischen Universität*, 6:65–85, 1928.

[49] William A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, λ-calculus and Formalism*, pages 479–490. Academic Press, 1980.

[50] D. J. Howe. The Computational Behaviour of Girard's Paradox. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 205–214. IEEE Computer Society, 1987.

[51] Gérard P. Huet. A Mechanization of Type Theory. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 139–146. William Kaufmann, 1973.

[52] Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[53] A. J. C. Hurkens. A simplification of Girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[54] L. S. Jutting. The development of a text in aut-qe. In P. Braffort et L. Michel, editor, *Symposium d'Orsday sur la manipulation des symboles et l'utilisation d'APL*, 1973.

[55] G. Kreisel. Church's thesis: A kind of reducibility axiom for constructive mathematics. In *Intuitionism and Proof Theory (Proc. Conf., Buffalo, N.Y., 1968)*, pages 121–150. North-Holland, Amsterdam, 1970.

[56] J.-L. Krivine. Un algorithme non typable dans le système F. (An algorithm not typable in the system F). *C. R. Acad. Sci., Paris, Sér. I*, 304:123–126, 1987.

[57] J. Lambek. A fixpoint theorem for complete categories. *Math. Z.*, 103:151–161, 1968.

[58] Peter J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Commun. ACM*, 8(2):89–101, 1965.

[59] The Caml Language. A History of Caml, 2005.

[60] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 42–54, 2006.

[61] Paul Lorenzen. *Einführung in die operative Logik und Mathematik*, volume 78 of *Grundlehren Math. Wiss.* Springer, 1955.

[62] P. Martin-Löf. A Theory of Types. Rep., Dep. Math., Univ. Stockh., 1971.

[63] P. Martin-Löf. On the strength of intuitionsitic reasoning. Rep., Dep. Math., Univ. Stockh., 1971.

[64] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. Sheperdson, editors, *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975.

[65] P. Martin-Löf. Syntax and Semantics of Mathematical Language. Notes written by P. Hancock, Rep., Dep. Math., Univ. Stockh., 1975.

[66] P. Martin-Löf. Constructive mathematics and computer programming. Rep., Dep. Math., Univ. Stockh., 1979.

[67] Per Martin-Löf. Hauptsatz for the theory of species. Proc. 2nd Scandinav. Logic Sympos. 1970, Studies Logic Foundations Math. 63, 217-233, 1971.

[68] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory. Lecture Notes.* Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[69] Per Martin-Löf. A construction of the provable wellorderings of the theory of species. In *Logic, meaning and computation. Essays in memory of Alonzo Church*, pages 343–351. Dordrecht: Kluwer Academic Publishers, 2001.

[70] A. R. Meyer and M. B. Reinhold. "Type" Is Not A Type. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 287–295. ACM Press, 1986.

[71] A. Miquel. *Le Calcul des Constructions Implicite: Syntaxe et Sémantique.* PhD thesis, Université Paris VII, 2001. PhD thesis.

[72] A. Miquel. Lamda-Z: Zermelo's Set Theory as a PTS with 4 Sorts. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 232–251. Springer, 2004.

[73] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In Mary S. Van Deusen, Zvi Galil, and Brian K. Reid, editors, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, pages 37–51. ACM Press, 1985.

[74] Rob Nederpelt and Herman Geuvers. *Type Theory and Formal Proof: An Introduction.* Cambridge University Press, USA, 1st edition, 2014.

[75] B. Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, 1988.

[76] B. Nordström and K. Petersson. Types and specifications. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 915–920. North-Holland/IFIP, 1983.

[77] Christine Paulin-Mohring. Algorithm development in the calculus of constructions. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, pages 84–91. IEEE Computer Society, 1986.

[78] P.-M. Pédrot and N. Tabareau. Failure is not an option - an exceptional type theory. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018*, pages 245–271, 2018.

[79] F. Pfenning and Ch. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 1989.

[80] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 74–85. IEEE Computer Society, 1991.

[81] Henri Poincaré. Les mathématiques et la logique. *Revue de métaphysique et de morale*, 13:815–835, 1905.

[82] Henri Poincaré. La logique de l'infini. *Scientia (Rivista di Scienza)*, 12(24):1–11, 1912.

[83] Michael Rathjen. Constructive Zermelo–Fraenkel Set Theory, Power Set, and the Calculus of Constructions. In P. Dybjer, S. Lindström, E. Palmgren, and B. G. Sundholm, editors, *Epistemology versus Ontology. Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf.* Springer Netherlands, Dordrecht, 2012.

[84] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[85] John C. Reynolds. Towards a theory of type structure. In Bernard Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[86] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984.

[87] John C. Reynolds and Gordon D. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Inf. Comput.*, 105(1):1–29, 1993.

[88] Bertrand Russell. *The Principles of Mathematics.* W. W. Norton & Company, Berlin, 2 edition, 1903.

[89] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908.

[90] D.S. Scott. Constructive validity. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, pages 237–275. Springer Berlin Heidelberg, 1970.

[91] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Amsterdam: Elsevier, 2006.

[92] Thomas Streicher. *Semantics of Type Theory.* Progress in Theoretical Computer Science. Birkhaeuser Boston, Inc., 1991.

[93] Aaron Stump. The calculus of dependent lambda eliminations. *J. Funct. Program.*, 27:e14, 2017.

[94] N. Tabareau, E. Tanter, and M. Sozeau. The marriage of univalence and parametricity. *J. ACM*, 68(1):5:1–5:44, 2021.

[95] W. W. Tait. Intensional interpretations of functionals of finite type. I. *J. Symb. Log.*, 32:198–212, 1967.

[96] G. Takeuti. On the fundamental conjecture of GLC. I, II. *J. Math. Soc. Japan*, 7:249–275, 394–408, 1955.

[97] Taichi Uemura. Cubical assemblies and the independence of the propositional resizing axiom, 2018. Preprint.

[98] L.S. van Benthem Jutting. *Checking Landau' "Grundlagen" in the AUTOMATH system.* PhD thesis, University of Eindhoven, 1977. PhD thesis.

[99] L.S. van Benthem Jutting. Normalization in Coquand's system, 1986. Internal Report.

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

[101] Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Math. Structures Comput. Sci*, 25:1278–1294, 2015.

[102] Jan von Plato. *The great formal machinery works.* Princeton University Press, Princeton, NJ, 2017. Theories of deduction and computation at the origins of the digital age.

[103] S. S. Wainer. Slow growing versus fast growing. *J. Symb. Log.*, 54(2):608–614, 1989.

[104] A. N. Whitehead and B. A. W. Russell. *Principia Mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.

[105] G. C. Wraith. A note on categorical datatypes. In D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, pages 118–127, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.