Some remarks about impredicativity

Thierry Coquand

ITC talk, 2020/11/25

This talk

I will try to describe a calculus I designed in 84/85

Sources and questions at the time

Main message: importance of *notations*

New notations introduced by de Bruijn where proofs are first-class objects

de Bruijn had several interesting discussions about notations

Cf. Leibnitz notation $f(x)dx$ or introduction of variables or introduction of the notion of functions

This talk

First part: why it is nice to use impredicativity

Second part: paradoxes

Third part: what happens if we don't have impredicativity

Context

Representation of proofs in a computer with the followin ultimate goal

the computer should generate interesting new concepts and new proofs

(cf. 1958 Advice Taker J. McCarthy)

Partial goal: the computer should check the correctness of a given proof

Also it should provide help in analysing a given proof

-how many time some lemma has been used? (already in Frege 1879)

-can we simplify a general result instantiated to a special situation?

L.S. van Benthem Jutting, *The development of a text in AUT-QE*, 1973

Context: System F

Quite remarkable: *same* extension of simply typed λ-calculus designed independently by a logician Girard and a computer scientist Reynolds

Mysterious calculus: $\Lambda\alpha\lambda x^\alpha x^\alpha$ of type $\forall\alpha\ (\alpha \rightarrow \alpha)$

No apparent set theoretic semantics

But Reynolds was conjecturing (83) that there should be one

Only to find one year later a *theorem* that there cannot be such semantics

Context: System F

System F quite complex system compared to simply typed λ-calculus

Restriction on typed variables: in the rule $\Lambda\alpha.M : \forall\alpha.\sigma$ we should have that α does not appear free in some type of a variable of M, e.g. $\Lambda\alpha.x^\alpha$ is not allowed

Girard had an extension Fω even more complex

It seemed important not to mix the order of type variables and term variables

In (then) presentations of simply typed λ-calculus, one started with an infinite collection of variables for each type

context: AUTOMATH

de Bruijn, see archive papers https://www.win.tue.nl/automath/

N.G. de Bruijn, *AUTOMATH, a language for mathematics*, 1973

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

context: AUTOMATH

Representation of a statement

Theorem 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 (1) of $f(\pi) > 1$ and then a proof (2) of $g(\pi) > \pi^5$

He can then state $f(\pi) > 5$ by *applying* theorem 1 and by giving *in this order*

π, the proof (1), 5 the proof (2)

AUTOMATH

In AUTOMATH this will become

Corollary $=$ Theorem 1(π, (1), 5, (2)) : A

where A is the statement $f(\pi) > 5$

AUTOMATH

A crucial notion in AUTOMATH, inspired from the notion of *block structure* in ALGOL 60, is the one of *context*

Sequence of variable declaration with their types and named hypotheses in an *arbitrary* order

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

AUTOMATH also had a primitive "sort" type

$R : \mathsf{type},\ N : \mathsf{type},\ x : R,\ h_1 : f(x) > 1,\ n : N,\ h_2 : g(x) > x^n$

One also could introduce a primitive sort prop or take prop $=$ type

AUTOMATH

AUTOMATH used 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

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

AUTOMATH

One of the first example was equality on $A : \mathsf{type}$

$\mathsf{Eq} : [x : A][y : A]\mathsf{type}$

$\mathsf{refl} : [x : A]\mathsf{Eq}\ x\ x$

$\mathsf{eucl} : [x : A][y : A][z : A]\mathsf{Eq}\ x\ z \rightarrow \mathsf{Eq}\ y\ z \rightarrow \mathsf{Eq}\ x\ y$

These were introduced as *primitives*

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

is then of type $[x : A][y : A]\ (\mathsf{Eq}\ x\ y) \rightarrow \mathsf{Eq}\ y\ x$

AUTOMATH

I was quite impressed by the following example

Heyting rules for intuitionistic logic looked quite formal

E.g. why $A \to \neg(\neg A)$ and not $\neg(\neg A) \to A$?

With AUTOMATH notation this becomes clear $\neg\ A = A \to\perp$

We have $A \to (A \to\perp) \to\perp$

Proof $[x : A][f : A \to\perp]f\ x$

We don't have $((A \to\perp) \to\perp) \to A$

AUTOMATH and system Fω

$$\frac{}{\Gamma \vdash x : A} \qquad x : A \; in \; \Gamma$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash [x : A]M : [x : A]B} \qquad \frac{\Gamma \vdash N : [x : A]B \qquad \Gamma \vdash M : A}{\Gamma \vdash N \;\; M : B(M/x)}$$

AUTOMATH and system Fω

$$C \quad ::= \quad \mathsf{type} \mid [x:A]C$$

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

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

AUTOMATH and system Fω

We allow quantification over sorts of the form $[x_1 : A_1] \ldots [x_n : A_n]\mathsf{type}$

E.g. $[A : \mathsf{type}]A$ is considered to be a type

Like in system F!

This possibility was suggested by de Bruijn (1968) but with the mention: “It is difficult to see what happens if we admit this”

AUTOMATH and system Fω

Even more suggestive formulation at the time with $\tau(M)$ type of M

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

if $\tau(A) \leqslant \text{type}$

$$\frac{\Gamma[x : A] \vdash M}{\Gamma \vdash [x : A]M} \qquad \frac{\Gamma \vdash N \qquad \Gamma \vdash M}{\Gamma \vdash N\ M}$$

if $\tau(N) = [x : A]B$ and $\tau(M) \leqslant A$

with $\Gamma[x_1 : A_1] \dots [x_n : A_n]\text{type} \leqslant \Gamma\text{type}$ and for *predicative systems* we should have $A_i : \text{type}$

Notation

The computation of $\tau(M)$ is remarkably simple if, like in Automath, we write $(M)N$ for $M\ N$, the argument is on the left of the function

Then $\tau(M)$ is simply obtained by replacing the head variable by its type

$\tau([A:\mathsf{type}][x:A]x) = [A:\mathsf{type}][x:A]A$

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

A redex is then $(M)[x:A]N$ and we can have $(M)[x:A](N)[y:B]\dots$

If x not free in N this can be simplified to N

More traditional presentation

$$A, M \ ::= \ x \mid \lambda_{x:A} M \mid M\ N \mid \Pi_{x:A} B \mid \mathsf{type} \qquad C \ ::= \ \mathsf{type} \mid \Pi_{x:A} C$$

$$\frac{\Gamma, x : A \vdash B : \mathsf{type}}{\Gamma \vdash \Pi_{x:A} B : \mathsf{type}} \qquad \frac{\Gamma, x : A \vdash C}{\Gamma \vdash \Pi_{x:A} C}$$

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

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda_{x:A} M : \Pi_{x:A} B} \qquad \frac{\Gamma \vdash N : \Pi_{x:A} B \qquad \Gamma \vdash M : A}{\Gamma \vdash N\ M : B(M/x)}$$

AUTOMATH and system Fω

We write $A \rightarrow B$ pour $[x : A]B$ if x is not free in B

$[A : \mathsf{type}]A \rightarrow A$ is the type of the polymorphic identity

$$[A : \mathsf{type}][x : A]x \ : \ [A : \mathsf{type}]A \rightarrow A$$

simply because $[A : \mathsf{type}][x : A] \vdash x : A$

No problem with restriction on variables for $\forall$ introduction

This is taken care of by the notion of context

Also, at each given time, only finitely many variables that are “alive”

AUTOMATH and system Fω

Russell-Prawitz encoding

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

$\mathsf{Eq} \quad : [A : \mathsf{type}]A \to A \to \mathsf{type}$

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

Encoding of Church, Girard, Martin-Löf, Böhm-Berarducci (85)

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

AUTOMATH and system Fω

Simple and *uniform* notation

Contains system Fω

But also Church's 1940 system for higher-order logic

Inherits from AUTOMATH the uniform treatment of functions and proofs

Proof-checking becomes type-checking

Can be represented on a computer

AUTOMATH and system Fω

$\bot = [A : \mathsf{type}]A$

We have $[B : \mathsf{type}][x : \bot]B$

Proof $[B : \mathsf{type}][x : \bot](B)x$

Type $[B : \mathsf{type}][x : \bot](B)[A : \mathsf{type}]A$ which is $[B : \mathsf{type}][x : \bot]B$

We have two proofs of $\bot \rightarrow \bot$

$[x : \bot]x$ and $[x : \bot](\bot)x$

AUTOMATH and system Fω

Contrary to Martin-Löf system, we don't have to assume anything

Everything can be built from "nothing"

Data types $N = [A : \mathsf{type}]A \to (A \to A) \to A$

Logical notions $\mathsf{Eq} = [A : \mathsf{type}][x : A][y : A][P : A \to \mathsf{type}]P\ x \to P\ y$

$[A : \mathsf{type}][x : A]\mathsf{Eq}\ A\ x\ x$ proved by $[A : \mathsf{type}][x : A][P : A \to \mathsf{type}][h : P\ x]h$

This is the advantage of impredicativity

AUTOMATH and system Fω

Also $M : A$ is *decidable*

So we reduce proof-checking to type-checking

This was not the case for Martin-Löf system at the time (79-86) which used equality reflection and where the judgement $M : A$ was *not* decidable

Hence this system was much more complex to implement

Kent Petersson, 1981, using a LCF approach

Same for NuPrl (the NuPrl groups had a lot of discussions if one should use or not the equality reflection rule)

Unclear status for terms: they *cannot* be considered as proofs

Frege

One of the first example encoded in this calculus was the result proved by Frege in his remarkable 1879 book *Begriffsschrift*

Frege introduced not only the notion of quantifiers but also higher-order logic

Axioms

$\varphi \rightarrow \psi \rightarrow \varphi$ $\quad\quad$ $(\varphi \rightarrow \psi \rightarrow \delta) \rightarrow (\varphi \rightarrow \psi) \rightarrow \varphi \rightarrow \delta$

$(\forall x \phi(x)) \rightarrow \phi(t/x)$

Frege

The crucial rule is the following

$\varphi \rightarrow \forall x\ \psi(x)$ if $\varphi \rightarrow \psi(x)$ *and* x is not free in φ

Absolutely remarkable that one can capture in a finite way the laws about quantification over a maybe infinite collection!

AUTOMATH

$$\frac{\Gamma[x : A] \vdash M}{\Gamma \vdash [x : A]M}$$

I found it also remarkable that this simply becomes a shift of $[x : A]$

Frege

Frege shows that the transitive closure of a functional 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"

Transitive closure of a relation $R : A \rightarrow A \rightarrow \mathsf{type}$

$$
\begin{aligned}
&[x : A][y : A][S : A \rightarrow A \rightarrow \mathsf{type}]\\
&([a : A][b : A]R\ a\ b \rightarrow S\ a\ b) \rightarrow\\
&([a : A][b : A][c : A]S\ a\ b \rightarrow S\ b\ c \rightarrow S\ a\ c) \rightarrow\\
&S\ x\ y
\end{aligned}
$$

Inductive definitions

In higher-order logic, we can represent inductive/co-inductive definitions

Another example was the proof of Newman's Lemma by Gérard Huet

Use the notion of Noetherian relation, which can also be represented

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

We encode this as the prinicple of Noetherian induction

Inductive definitions

Two encoding of equality

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

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

Inductive definitions

type of ordinals $Ord = [A : \text{type}][x : A][f : A \to A][l : (N \to A) \to A]A$

We can program functions $Ord \to (N \to N)$

We can define $\omega,\ \epsilon_0,\ \ldots$ without any problems

Inductive definitions

It is not so easy to define a predecessor function $N \to N$

We get *iteration* and not directly *primitive recursion*

$f(S\ n) = g(f(n)) \qquad f(0) = a$

$f(S\ n) = g(n, f(n)) \qquad f(0) = a$

But we can build the product of types

hence we define instead $n \mapsto (n, f(n))$ by iteration

In this way, we recover Kleene's encoding in λ-calculus

Inductive definitions

It is *not* possible to show the induction principle

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

H. Geuvers (2001) has even proved that we *cannot* find an encoding of natural numbers where induction principle is provable

But this did not seem to be such a big issue: we can instead reduce ourselves to natural number satisfying the predicate

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

Inductive definitions

It is clear how to build C systematically from N

Internalisation of computability/parametricity predicate

We cannot show $\neg(\mathsf{Eq}\ N\ 0\ (S\ 0))$ however

Inductive definitions

Inductively Defined Types in the Calculus of Constructions,
Ch. Paulin-Mohring, F. Pfenning, MFPS 1989

An example: encoding of F_2 in F_3

Quite clear with these notations, and mixing propositions and types

Inductive definitions

We introduce a *predicate* P on type with constructors

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

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

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

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

This is thought as a predicate but it can also be seen as a type and this provides an encoding of F_2 in F_3

Inductive definitions

This encoding can also be interesting in a univalent setting

Cf. work of Awodey, Frey, Speight LICS 2018

Encoding of the circle $[X : \mathsf{type}][x : X]\ x =_X x \to X$

An example they don't mention may be the encoding of $\mathbb{Z}$

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

Inductive definitions

Letter from Plotkin to Reynolds 84

General pattern $A = [X : \mathsf{type}](T\ X \to X) \to X$ weak initial algebra for $T : \mathsf{type} \to \mathsf{type}$

provided we have $(X \to Y) \to T\ X \to T\ Y$

If we have $u : T\ A$ and $f : T\ X \to X$ we have $A \to X$ hence $T\ A \to T\ X$ hence $T\ X$ and X

So $\mathsf{intro} : T\ A \to A$ and so $T\ (T\ A) \to T\ A$ and so $\mathsf{match} : A \to T\ A$

This is Lambek's argument, but it is quite concrete with these notations

Inductive/Co-Inductive definitions

For $T\ X = 1 + X$

We write $(T\ X \to X) \to X$ as $((1 + X) \to X) \to X$ which becomes $X \to (X \to X) \to X$

This works for streams! (Wraith 1989)

$S = \exists_{X:\text{type}} A \times (X \to X)$

In general $\exists_{X:\text{type}} X \to T\ X$

Extremely flexible, without syntactic restrictions

Inductive/Co-Inductive definitions

We can encode existential quantification

$\exists = [A : \mathsf{type}][P : A \rightarrow \mathsf{type}][Q : \mathsf{type}]([x : A]P\ x \rightarrow Q) \rightarrow Q$

Encoding of $\exists$ by how we *use* an existential statement

We can then program $\pi_1 : \exists\ A\ P \rightarrow A$

but *not* $\pi_2 : [z : \exists\ A\ P]P\ (\pi_1\ z)$

Inductive/Co-Inductive definitions

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

There we don't even have the first projection!

Mitchell-Plotkin encoding of *abstract* data types 1988

AUTOMATH, system F and Higher-Order Logic

AUTOMATH introduces a new notion of *definitional* equality

This is β, η-conversion

It is different from the term Eq introduced as a type the *book equality*

This notion of definitional equality is not present in Frege (nor in HOL)

Frege has a primitive notion of equality which is used for representing definitions, like in more recent presentations of HOL

AUTOMATH, system F and Higher-Order Logic

While representing proofs in Frege I thought that the terms expressed exactly what I had in mind when going through/trying to understand these proofs

This seems to provide a quite good system of notations for proofs

For instance, one of the first proof in Frege uses twice the same lemma in two different ways

Furthermore, we can do operations on proofs, e.g. instantiations, using β-reduction

Consistency and expressiveness

Is this calculus *consistent* ?

Not clear at the time, though one year later I found out that there should be a *finite model* interpreting type as $\{0,1\}$

This became clear only with Aczel's encoding of product

Type Theory and Set Theory, 2001

I found out also that Martin-Löf had introduced a quite similar calculus but with $\tau(\text{type}) = \text{type}$

Consistency and expressiveness

Girard had found a contradiction in this system; not easy to follow

But, looking at his proof, it was clear that the calculus becomes *inconsistent* if one introduces a Σ type with two projections!

$$\frac{\Gamma, x : A \vdash B : \mathsf{type}}{\Gamma \vdash \Sigma_{x:A} B : \mathsf{type}}$$

So consistency is connected to the fact that we cannot encode "strong" sums (as it was called by Howard already in 1969)!

Girard’s Paradox

How did Girard discover his paradox?

Martin-Löf had a *consistency proof* for his system with type : type

How is this possible?

Consistency Proof

Russell found a paradox in Frege's system 1901

Stratification in types for avoiding this paradox

Still the impredicativity was not something so obvious

Quantification over all propositions, still a proposition $[A : \mathsf{type}]A : \mathsf{type}$

This problem is more apparent in system F where we have $[A : \mathsf{type}]A \rightarrow A$

We can form terms such as $[x : \perp]x\ (\perp \rightarrow \perp)\ x$ that look dangerous

Consistency Proof

Takeuti (1945), Prawitz, Martin-Löf, Andrews

Consistency for higher-order logic?

Cannot be elementary (Gödel)

Normalisation of system Fω implies consistency of higher-order logic

Girard 1970: found a very elegant proof of normalisation for system F

Analysed by Martin-Löf

One should use a stronger system in the meta logic (Gödel)

Consistency Proof

The most elegant normalisation proof is obtained when the metalogic is as close as possible to the system itself

Cf. 2010 work of Bernardy, Jansson, Paterson on parametricity

$[A : \mathsf{type}][x : A]x$ becomes $[A : \mathsf{type}][A' : A \to \mathsf{type}][x : A][x' : A'\ x]x'$

This works with $\mathsf{type} : \mathsf{type}$ with $\mathsf{type}' = [A : \mathsf{type}]A \to \mathsf{type}$

This is essentially what is going on with Martin-Löf's proof

Cf. Canonicity and normalisation for dependent types, T.C. 2018

Using previous works by Altenkirch, Hofmann and Streicher

Consistency Proof

So we cannot rely on consistency proofs for “strong” system

Even for predicative systems, such as Martin-Löf’s present system

On the other side, when we have a paradox this is something concrete which can be tested

Girard's Paradox

Girard did not find his paradox while looking for a contradiction with type : type

He found it for an extension of system F with a type of propositions

Given that Haskell uses system Fω this is something quite natural to have

His result was that the type of propositions cannot be a type in the system itself (the logic of Haskell has to be "external")

Girard’s Paradox

In his system, we had prop : type and we can quantify

(1) $[A : \mathsf{type}]B : \mathsf{prop}$ if $A : \mathsf{type} \vdash B : \mathsf{prop}$

(2) $[x : A]B : \mathsf{prop}$ if $A : \mathsf{type}$ and $x : A \vdash B : \mathsf{prop}$

(3) $A \rightarrow B : \mathsf{prop}$ if $A : \mathsf{prop}$ and $B : \mathsf{prop}$

Girard wrote that the system obtained by leaving (1) was “maybe consistent”

I found out later 1989 that this was not the case: we got a contradiction even using only (2) and (3)

Girard and Reynolds

Reynolds was using

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

Since this is covariant we can define $\mathsf{intro} : A \to T\ A$ and $\mathsf{match} : T\ A \to A$

This does not give an isomorphism between A and $T\ A$ but if we use a parametricity interpretation we get such an isomorphism and a contradiction

As noticed by Plotkin 84, this is essentially what is used when proving Freyd's Adjoint Functor Theorem

Girard and Reynolds

Reynolds was doing the reasoning in set theory, but his proof works as well directly in type theory

I found out only that this applies to Girard's question after Barendregt had introduced notations for Pure Type Systems

Girard and Reynolds

Later 1994 Hurkens found out that we have a direct paradox without going via parametricity but using

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

While preparing this talk, I discovered that exactly the same argument works with $A = [X : \mathsf{type}](T\ X \to X) \to X$

The proof is quite short and can be checked in Agda –type-in-type

Though we don't have intro as a definitional retract there are still some definitional equalities valid that would be interesting to analyse further

Girard and Reynolds

$$
\begin{array}{llll}
Pow & : & \mathsf{type} \to \mathsf{type} & = & \lambda X.X \to \mathsf{prop} \\
T & : & \mathsf{type} \to \mathsf{type} & = & \lambda X.Pow\ (Pow\ X) \\
U & : & \mathsf{type} & = & \Pi_{X:\mathsf{type}}\ (T\ X \to X) \to X \\
\tau & : & T\ U \to U & = & \lambda t \lambda X \lambda f \lambda p.t\ (\lambda x\ (p\ (f\ (x\ X\ f)))) \\
\sigma & : & U \to T\ U & = & \lambda s.s\ U\ \tau \\
Q & : & T\ U & = & \lambda p.\Pi_{x:U}\ \sigma\ x\ p \to p\ x \\
B & : & Pow\ U & = & \lambda y.\neg\ \Pi_{p:Pow\ U}\ \sigma\ y\ p \to p\ (\tau\ (\sigma\ y)) \\
C & : & U & = & \tau\ Q \\
lem_1 & : & Q\ B & = & \lambda x \lambda k \lambda l.l\ B\ k\ (\lambda p.l\ (\lambda y.p\ (\tau\ (\sigma\ y)))) \\
A & : & \mathsf{prop} & = & \Pi_{p:Pow\ U}\ Q\ p \to p\ C \\
lem_2 & : & \neg\ A & = & \lambda h.h\ B\ lem_1\ (\lambda p.h\ (\lambda y.p\ (\tau\ (\sigma\ y)))) \\
lem_3 & : & A & = & \lambda p \lambda h.h\ C\ (\lambda x.h\ (\tau\ (\sigma\ x))) \\
loop & : & \perp & = & lem_2\ lem_3
\end{array}
$$

Girard and Reynolds

We can try to understand the proof by instantiations/β, ι-reduction

The proof reduces to a similar term but with bigger types

Intuitively, the proof becomes more and more complex when we try to understand it!

Meyer-Reynolds 1986: we can use this term and encode a quasi-fixed-point combinator and encode any general recursive function as a term $N \rightarrow N$

Consistency

This paradox is not so surprising since it is intuitive that system F could not have a set theoretic semantics

(Still for a while Reynolds thought it had one)

Girard found then that it can apply to type : type since we can translate such a system in this extension of system F

This was then *really* surprising since Martin-Löf had a consistency proof!

Consistency

With respect to the calculus I was studying this meant that we cannot add $\mathsf{type} : \mathsf{type}_1$ with type_1 being impredicative

$$\frac{\Gamma, x : A \vdash B : \mathsf{type}}{\Gamma \vdash [x : A]B : \mathsf{type}} \qquad \frac{\Gamma, x : A \vdash B : \mathsf{type}_1}{\Gamma \vdash [x : A]B : \mathsf{type}_1}$$

In order to have a system (hopefully) consistent, we need the level type_1 to be predicative

$$\frac{\Gamma \vdash A : \mathsf{type}_1 \qquad \Gamma, x : A \vdash B : \mathsf{type}_1}{\Gamma \vdash [x : A]B : \mathsf{type}_1}$$

Coherence et expressivite

We then get a system with $\mathsf{type}_i : \mathsf{type}_{i+1}$ and the law

$$\frac{\Gamma \vdash A : \mathsf{type}_i \qquad \Gamma, x : A \vdash B : \mathsf{type}_j}{\Gamma \vdash [x : A]B : \mathsf{type}_{max(i,j)}}$$

and it is natural to use prop for the base impredicative sort type

$$\frac{\Gamma, x : A \vdash B : \mathsf{prop}}{\Gamma \vdash [x : A]B : \mathsf{prop}}$$

Th. C. *An analysis of Girard's paradox*, LICS 1986

Consistency

How does this system compare with set theory?

Conjecture (86): this system should be stronger than Zermelo set theory

Expressiveness

Encode $\mathsf{nat} : \mathsf{type}_2$ as $[A : \mathsf{type}_1]A \rightarrow (A \rightarrow A) \rightarrow A$

$0 : \mathsf{nat}$ and $\mathsf{S} : \mathsf{nat} \rightarrow \mathsf{nat}$

The infinity axiom is provable!!

Quite surprising: Russell thought that this should not be provable

$[x : \mathsf{nat}]\neg\mathsf{Eq}\ \mathsf{nat}\ 0\ (\mathsf{S}\ x)$

$[x : \mathsf{nat}][y : \mathsf{nat}]\mathsf{Eq}\ \mathsf{nat}\ (\mathsf{S}\ x)\ (\mathsf{S}\ y) \rightarrow \mathsf{Eq}\ \mathsf{nat}\ x\ y$

Expressiveness

A. Miquel PhD thesis 2001

We can encode *pointed graphs* as binary relations

Since co-induction is definable, we can define bissimulation

A *set* can then be encoded as a pointed graph up to bissimulation

This is a model of Aczel *non well-founded* set theory

All the axioms of Zermelo set theory are then satisfied

There is even a double negation interpretation to get classical logic

Expressiveness

Problems with data type representations

-relativisation $C : N \rightarrow \mathsf{prop}$ to get induction principle

-representation of primitive recursion

$f\ 0 = a \qquad f\ (\mathsf{S}\ n) = g(n, f\ n)$

possible, but not natural

-equality is definable but we don't get the dependent elimination rule

Expressiveness

All this suggests to add data types like in Martin-Löf type theory with computation rules

Inductive Definition in Type Theory
N. Mendler, PhD thesis, 1987

Inductively defined types
Th. C., Ch. Paulin-Mohring, COLOG-88

Expressiveness

If we allow inductive definitions Girard's paradox becomes very simple: the type U with a constructor of type

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

has an element $\mathrm{sup}\ \ U\ \ ([x : U]x)$ which is both well-founded and has itself has a subtree

Cf. The Paradox of Trees in Type Theory, T.C. 1992

With a predicative system of universes and data types we get a system weaker than Zermelo-Fraenkel (work of M. Rathjen)

Expressiveness

AUTOMATH : notion of *context* and *proof-checking* reduced to *type-checking*

Martin-Löf : data types, and correspondance between the notion of *constructors* and *introduction rules* (non decidable typing, 79-86)

Predicativity/Impredicativity

Spent a lot of times 88-90 on implementation of inductive families

Definitively less elegant than the impredicative encoding

On the other hand, this encoding is not expressive enough

This work suggested the use of pattern-matching notation 92 and seeing type theory as a total fragment of a programming language with dependent type and definitions by pattern-matching

Still some not so elegant syntactical restriction

Very relevant work of Jesper Cockx

Predicativity/Impredicativity

The work by A. Stump (2018) on *Cedille* tries to get a good encoding of data types

Use another encoding than Church encoding of natural numbers where we have a nice representation of primitive recursion

Also used in Daniel Fridlender's work A Proof-Irrelevant Model of Martin-Löf's Logical Framework (2002)

Predicativity/Impredicativity

Is the impredicative type prop crucial?

In practice, most proofs in mathematics do not use impredicativity in an essential way

However in practice, all notions become dependent on the universe levels

So it has been very convenient (but not essential) to have such a type, e.g. in Gonthier's proof of the 4 color theorem

Same question for programming: is the use of Fω essential for Haskell?

Expressiveness

With recent works on univalence, and for category theory, we *need* to deal with varying universes in any case

Girard's paradox shows that this is necessary

The recent (2010?) universe level system of Agda seems to be a really good solution

Cf. Martin Escardo's
Introduction to Univalent Foundations of Mathematics with Agda

According to Voevodsky, this question of dealing with universe is essential but was systematically "hidden" in recent works in mathematics (Grothendieck always tried to be very precise about it)

Expressiveness

Voevodsky has suggested to add a “resizing” rule

This could be added to cubical Agda

But the normalisation property is completely open and seems to be an interesting problem