An Analysis of Girard's Paradox

Thierry Coquand

CMU and INRIA

Introduction

The purpose of what follows is to present the paradox of Girard, sligthly modified, and to use it for gaining an understanding of the connection between type theory and set theory, and also to analyse the Curry-Howard analogy between propositions and types.

But the main point of this article is that Girard's paradox has been mechanically checked in an implementation of the typed calculus in the construction's style. This checking illustrates well the use of a type-checker as a proof-checker system. A study of the implemented paradox shows in particular that some parts in the derivation illustrate in a concrete way the traduction of Topos Theory in Type Theory. This article claims thus to be a comment of Girard's result in a resolutely computer science spirit.

1 Some motivations

We are going to study some extensions of the higher-order type system and the construction calculus. The first questions to answer are: what is the purpose of considering such systems ? Why is the usual typed calculus not sufficient ?

The point is that the usual type structure with only the arrow and the product does not capture certain uniformities. By example, if we want to write the algorithm "quicksort", we have in mind something "parametric" in the data of elements to be sorted and the binary ordering predicate on it. We want thus to write a general algorithm, and to specialize it on given data. This is the notion of polymorphism implemented in the language ML (cf. [18]). Note that this notion of parametric type is restricted: we cannot pass our procedure quicksort as a parameter of another program, i.e. we cannot use quicksort as an argument, but only as something expecting arguments (here a type and a binary relation on it). Note also that the type of a parametric algorithm such as quicksort is not a type like int, $int \to bool$,... but it is an object of another sort. There is no circularity in such a type system.

If we are looking for a real motivation of the full polymorphism (where we can use a type of a parametric algorithm as a real type), we must consider the analogy between propositions and types . This is known as the "Curry-Howard isomorphism", actually used for the first time in the Automath project [4]. We shall see that the term "isomorphism" is perhaps misleading.

This analogy is the basis of the implemented calculus of constructions (as in Automath): the notion of arrow between types corresponds to the notion of implication, the notion of "cut-elimination" corresponds to the β-reduction, the modus-ponens rule corresponds to the application of a program to an argument... Note also that some of these operations have a corresponding one in set theory: exponentiations, set-theoretic application,...

In the world of proofs and propositions, and for building real proofs (in "real" mathematic), we need quantifications on predicates. We need to consider the set of propositions closed by such an

operation. This fact is known as the "non-predicativity" of mathematics and it has been noticed for the first time by Russel (this is the essence of the "reducibility axiom" [22]).

The non-predicative facility is not in the usual version of Automath (though it appears in an unpublished paper of de Bruijn [5]). The motivation for such a restriction appears to be an overestimation of the Curry-Howard analogy: if we want a complete analogy between set operations and proof operations (as this is the case in Automath), we have thus to forbid the non-predicativity for propositions, because such non-predicativity is dubious for sets (and actually, Girard's paradox shows that it leads to inconsistency). However, the non-predicativity for propositions appears to be essential for the development of mathematics (especially if one likes "abstract" and "set-theoretical" proofs).

This remark can perhaps be applied to other systems of proof-checker, where the non-predicativity is avoided at the two levels: for the sets and for the propositions. One purpose of this work is to show that this restriction has sound foundations for the the level of sets, but there seems to be no reasons to restrict a proof-checker as being predicative, since the non-predicative calculus (which is the usual one in mathematics) is as easy to implement, has nice properties of termination [11], and offers a greater power both of concision and of expressivity. These are the principal motivations (and it appears so to be purely empirical) for the use of the complete polymorphism (at least for propositions) in the construction calculus.

The higher-order logic of Girard (which can be viewed as a system of notation for the proofs in the higher-order predicate calculus) is completely implemented in the construction calculus, by using the nice notations of Automath. Experimentations with this system has shown however that this system does not capture enough uniformity. This means that the usual way of presenting higher-order logic (as [6], or as in topos, if one wants intuitionnistic logic) is not sufficient for a real development of mathematics. This fact is already noticed in the Principia, and this has to be connected with the fact that this is the last work where it is attempted to construct in a completely "symbolic" way the foundation of mathematics.

Here is an example of what is lacking. Let *Prop* be the type of the propositions (i.e. * in the system of [9], Kind in the system of [13], () in [11], and o in [6]). Suppose we want to state and prove the Knaster-Tarski theorem about fixed-points of increasing functions over a complete order.

What we want is to present a general version of this theorem, in such a way that it can be applied in different "concrete" given complete orders (by example, for the set of subsets of a given set, with the inclusion for the order, or the sets of relations over a given set, with inclusion). In the construction calculus, the sets of predicates (resp. relation) on a given "type" A is represented by $A \to Prop$ (resp. $A \to A \to Prop$). If we prove the theorem of Tarski for A, we want then be able to replace A by something of the form $B \to Prop$, or $B \to B \to Prop$. It appears thus that the range of the variable A must be what is called a "context" in the construction calculus (the motivation of this name was that, in some way, such object, as $B \to Prop$, $B \to B \to Prop$ are the internalisation of the notion of type environment), and what is called kinds in [17], in [19] and in [13]. This shows that it is perhaps misleading to take A of type *Prop*, as it is suggested by the Curry-Howard analogy between types and propositions, and as it was done in first attempts with the constructions [9].

We need thus a notion of "context" variable, or "Type" variable, to state general theorems and to be able to instanciate them in concrete situations. It is worth to notice that what we need is exactly what is called "typical ambiguity" in the Principia [22]. This is the possibility of writing and proving general statements uniformly in some type. We present the general calculus thus obtained at the end of this paper, which is basically higher-order logic, with a (weak) notion of "type" variable. This seems to be the exact analogue of the polymorphism of ML (this remark is

made precise later).

In summary, attempts of writing mathematical proofs in the constructions system show that the natural "level" of the objects which represent "sets" is the level of contexts (or Kinds in [13] or "ordres" in [11] or types in [6]) and that we need an extension of the usual higher-order calculus by the introduction of "context" variables. These considerations raise the question: is it possible to add the general polymorphism at the level of the types, in the same way that one can try to extend the usual language by the addition of the second-order types. The motivations are very strong: we know that such an extension is possible at the level of propositions, and this seems to be only the "internalisation" of the notion of "type" variable in the system.

The paradox of Girard shows that such an extension is not possible, as far as we want to preserve the termination property and the logical expressiblity of the language.

We must now make precise the terminology, as similar objects are named in very different ways in the litterature. The * of [9] is renamed in *Prop* and the type of "contexts" is denoted by *Type*. So, our *Prop* is what corresponds to the "*" of [9], to the Type of [17] and [13], to the () of [11], and to the *o* of [6]. Our *Type* is what corresponds to the "context" of [9], to the Kind of [13], the "orders" of [11] and the usual notion of types in [6]. This notation seems consistent: the Type of [13] has to be renamed in *Prop*, so long it is not clear if the types of a programming language corresponds really to the propositions of higher-order logic, or to the "types" of this logic (we have to choose the "level" of the programs).

The presentation of rules differs from our preceding presentations in one point: the use of an explicit constant for the product. The motivation for the use of an uniform notation $[x : A]B$ for the (typed) abstraction and for the product was the hope that such an uniformity will be reflected in the implementation. Experimentation has shown that this seems not to be true, and it is clearer to distinguish in the notation (and in the abstract syntax) the product and the abstraction. It is then not necessary to keep all typed information for the abstraction, and the abstraction becomes thus an unary operator, as in [16].

2 The rules of typing

We suppose known the basic notions of λ-calculus , such as β-reduction. The terms we use have the following structure

1. constants: *Prop*, *Type*
2. identifiers, and numbers (de Bruijn indexes)
3. abstraction $\lambda(N)$ and product $\Pi(M, N)$, where M and N are terms
4. application $(M\ N)$ where M and N are terms

We adopt the following notation: if M is a term and x an identifier, then $\lambda x.M$ denotes the following term: first, replace each occurrence of x in M by the appropriate number (as in [3]), and then make the abstraction. So, the identifier x does not appear in $\lambda x.M$. By example, if M is $(x\ \lambda y.(y\ x\ z))$ (which denotes $(x\ \lambda(1\ x\ z))$) then $\lambda x.M$ will be $\lambda(1\ \lambda y.(y\ 2\ z))$ (which denotes $\lambda(1\ \lambda(1\ 2\ z))$).

With this convention, all problems about bound variables disappear [3]. There are two kinds of variables: identifiers for the "free" variables, and numbers for the "bounds" variables. The same convention holds for product, and thus $(\Pi x : A)(x\ y)$ denotes $\Pi(A, (1\ y))$.

The notion of reduction is the usual one (cf. [1]), and we consider a product as irreductible for the reduction (we can also admit the reduction inside the product, but we choose the notion of reduction as effectively implemented). This relation is noted red. We note conv the smallest congruence on terms (for abstraction, product and application) which contains the reduction.

We now present rules of typing of the extension of the construction calculus where we allow "context" variables and general polymorphism on them This calculus will be named in the sequel the "general polymorphic calculus". We present these rules in the way they are effectively implemented in the language ML, and then discuss their "intuitive" meaning.

The rules construct some typing sequents, i.e. sequents of the form $B \vdash M : P$, where B is a list of typings $x : A$, where x is an identifier and M, P are terms. This relation can be read as "M is a valid term of type P in the type assignment B".

2.1 Assignments

1. The empty assignment is valid
2. If B is valid, and $B \vdash M : Prop$, (or $B \vdash M : Type$) then $B, x : M$ is valid, for any identifier x which does not occur in B

2.2 Type Inference Rule

1. If B is valid, then $B \vdash Prop : Type$
2. If B is valid, and x is an identifier which occurs in B with the type M, then $B \vdash x : M$
3. If $B, x : M \vdash N : P$, then $B \vdash \lambda x.N : (\Pi x : M)P$
4. If $B, x : M \vdash N : Prop$ (resp. $Type$), then $B \vdash (\Pi x : M)N : Prop$ (resp. $Type$)
5. If $B \vdash M : P$, $B \vdash N : A$, and P red $(\Pi x : R)S$, with R conv A, then $B \vdash (MN) : [N/1]S$

We can, as in Automath-like languages, define the degree of a term (cf. [4]) with $degree(Type) = 0$. We have then

Definition . We shall say that M is a $Type$ if, and only if, $degree(M) = 1$, and that M is a $Prop$ if, and only if, $B \vdash M : N$ with M conv $Prop$

Proposition 1. if $B \vdash M : N$, then $degree(M) = degree(N) + 1$, and $degree(M)$ is 1 , 2 *or* 3

Proposition 2 if $B \vdash M : N$, and $degree(M) = 3$ (resp. 2) then we have $B \vdash N : Prop$ (resp. $B \vdash N : Type$)

This presentation is very compact, which is convenient for the implementation, but not very "intuitive". It is worth it to compare this presentation of a typed calculus, which contains higher-order logic, with more standard presentations of higher-logic, such as [23]. The main differences with the calculus of Church (apart the fact that we have a notion of type variable) are first, the fact that this calculus is intuitionnistic and secondly, the fact that we are manipulating proofs and not only propositions, so that we have a λ-calculus at two differents levels. Apart from these (essential) differences, this system appears as a direct generalisation of Church's system, with the addition of type variables and a "double" quantification on these variables.

Note that we can define the arrow by $M \rightarrow N = \Pi(M, N)$ if M, N are both Types, and $M \Rightarrow N = \Pi(M, N)$ if M, N are both Props, in the same type assignment. We have thus two

notions of arrows. The notations are justified by the considerations of the first section: we must view Types as sets (and the $\rightarrow$ is viewed as exponentiation), and Prop as the type of propositions (and the $\Rightarrow$ is viewed as implication). These are only analogies, which can be made precise with the connection with Topos Theory (as done in [12]). As usual in that kind of system, the absurd proposition can be coded as $(\Pi p : Prop)p$.

We have two sorts of quantifications on a variable of type Type. A quantification at the same level $(\Pi x : Type)N : Type$, if $N : Type$ (with the hypothesis $x : Type$) and a quantification on the level of $Prop$,$(\Pi x : Type)N : Prop$, if $N : Prop$ (with the hypothesis $x : Type$). We shall see that the core of the paradox lies in this double quantification. Notice that the second quantification has a "set-theoretic" interpretation: it's only the formation of an universal statement on "set", but the first quantification has no such meaning (it has to be a product over all sets).

If we want to have the usual higher-order type system, all we have to do is to restrict the formation of type assignments by forbidding the introduction of type variables. We obtain thus the calculus of constructions [7].

Note finally that the fact that type assignments are sequents and not sets, allows ourselves to present a system without "typable conditions" on identifiers.

We shall now show how to prove the following

Theorem (Girard's paradox). There exists a term M such that $\vdash M : (\Pi p : Prop)p$, and no term satisfying this condition is normalisable.

3 The paradox

What follows is a comment of the "type-checked" version of Girard's paradox. The intuitive idea is the following: the set of all well-founded transitive relations cannot exist. This is thus analogous to the usual Burali-Forti paradox, but there is a supplementary trick to show that it is possible to build a type of all well-founded relations in the general polymorphic typed calculus.

The paradox is described also in [15], but Martin-Löf uses the sum and the integer as primitives. As we do not have the exact traduction of the sum and the integer in the "pure" system with only arrow and product as constructors (cf. [9]), it would be very heavy, if one wants an effective version of the paradox (i.e. type-checked) to use such a derivation.

Instead, we shall use one intermediate between [11] and [15].

We want first to emphasize one important point for the coding of mathematics in type theory.

3.1 How to represent sets in type theory

If we want to represent the notion of relation in type theory, the most natural attempt is to take a parameter $A : Type$, with a parameter $R : A \rightarrow A \rightarrow Prop$. This approach does not work when we try to built the notion of "segment" for the relation R and the object $x : A$, which must represent the set of all y such that $R(y, x)$. This segment has to be a type, with a binary relation on it, which captures the usual notion of segment. But we do not have enough type formations to construct the type corresponding to this notion (at least, not in a direct way, we can actually "code" the notion of sums, cf. [15], but the terms we could consider by this method would be more complex than the one obtained by the method described below). This difficulty appears very naturally when we try to construct the Burali-Forti paradox in the general polymorphic calculus.

Fortunately, there is a way to turn around this difficulty (and this is the way that is build the notion of sets in [22]). We code the notion of set as a pair of a type $A : Type$ and a predicate on it $P : A \rightarrow Prop$. Then, we represent the notion of relation as a type A together with a

predicate $P : A \to Prop$ and a relation $R : A \to A \to Prop$ on it, and for representing the segment determined by the element x of type A, all we have to do is to restrict the predicate P with predicate $\lambda y.R(y,x)$. This is the way chosen for the construction of the paradox.

This predicate can be looked as an "existence" predicate on the type A, but we see that there is very pragmatical reasons to its introduction. Another example is if one wants to state something about a class of predicates on a type. In general, the statement is only true for extentionnal classes, and so one has to relativize the assertion with the extentionality predicate on the types of predicates of predicates.

Actually, the best way for coding the notion of sets is the following: a pair of a type $A : Type$ and a binary relation $E : A \to A \to Prop$, with a proof p that this relation is symmetric, and transitive (this notion contains the previous one: take for the predicate P the term $\lambda\, x.E(x,x)$, which can be thought of as an "existence" predicate on the type A). We did not take this representation in our type-checked version, as the previous one is sufficient for the derivation of the paradox, and produces a smaller λ-term for this derivation. The derivation of [11] uses this translation of set in type theory. The first reason that the second translation is better is that it allows to state general statements, prove them in all generality, and then instantiate them in "concrete" situation, such as the type of predicate on a given type, with the extentional equality. The second reason is that this is exactly the way of representing topos theory in type theory [12]. We can see that this representation is very natural, as soon as we try to develop mathematics in type theory.

For example, the definition of the intentional equality on a type A follows the idea of Leibnitz: Two elements x and y of type A are *intentionally equal* if, and only if, there is a proof of the "proposition" $(\Pi P : A \to Prop)P(x) \Rightarrow P(y)$. It is then possible to prove that this relation is reflexive, symmetric and transitive (cf. [9] and [11]).

For simplification, we shall use in the sequel the usual mathematical terminology, instead of the effective coding in the type theory, as far as there is no difficulty in the translation.

In our effective derivation, we have to use a "purely logical" definition for the notion of well-foundation. First, we define the notion of eternal predicate P for a relation $R : P$ is eternal if, and only if, for all x such that $P(x)$, there is a y such that $P(y)$ and $R(y,x)$. Then R is well-founded if, and only if, all eternal predicates are void. It seems to be a good point for the use of typed system to mechanize mathematical inferences that we have to choose rather "abstract" definition of concepts for doing an explicit derivation, as it is this kind of definition which fits best in type theory.

3.2 The core of Girard's paradox

Once we understand how to code the usual notion of set theory in typed system, the writing of the preliminary for the "Burali-Forti" paradox is straigthforward (and all these derivations could be done in higher-order logic). Now, we have to understand why the introduction of Type variables opens the door to the paradox.

First, to derive the paradox, it is useful to remark that all we have to do is to construct a "universal notation system for relations" in the following meaning.

Definition. A *universal system of notation for relations* is a type $A : Type$ together with a term $i : (\Pi B : Type)(B \to B \to Prop) \to A$, such that if $i(B,R)$ and $i(C,S)$ are intentionally equal, then (B,R) and (C,S) are isomorphic.

It is then known how to derive a contradiction, once we have such terms, and the important thing is how to construct them.

One possiblity for the construction of such a type, perhaps the simplest one, is to take for A the type $((\Pi B : Type)((B \to B \to Prop) \to Prop)) \to Prop$, and for i the term $\lambda B.\lambda R.\lambda x.x(B,R)$ (this seems to be a little simpler than the one of [11]). This can be motivated by the usual way of embedding a type A in his type of class $(A \to Prop) \to Prop$. This is a general method of construction in the Principia [22]. For exemple, we can view pairs over the types A and B as elements of type $(A \to B \to Prop) \to Prop$. Here, this method solves our problem of the definition of an universal system of notation for relations.

Suppose now that $i(B,R)$ and $i(C,S)$ are intentionally equal as defined in the previous part, i.e. that we have a proof of $(\Pi P : A \to Prop)P(i(B,R)) \Rightarrow P(i(C,S))$, then we have to show that (B,R) and (C,S) are isomorphic. Let $Isom$ be the relation which expresses the isomorphism between two relations (whose exact construction is not relevant here). Then we can instantiate the given proof of equality between $i(B,R)$ and $i(C,S)$ on the predicate $Q = \lambda x.x(F)$, where F is the term $\lambda D.\lambda T.Isom(B,R,D,T)$. Then, $Q(i(D,T))$ is convertible to $Isom(B,R,D,T)$, and as the isomorphism is a reflexive relation, we obtain in this way a proof of $Isom(B,R,C,S)$. All the derivation is completely described in the appendix.

This shows that the "syntactic" explanation of the paradox is the double quantification on the Types. One has a clear semantics: the quantification $(\Pi x : Type)N$, if $N : Prop$ (it corresponds only to an universal statement on all sets), but the second one $(\Pi x : Type)N$, with $N : Type$, has no corresponding one in set theory (it should be a product over all sets).

The reason why all terms of type $(\Pi p : Prop)p$ are not normalisable is a straigthforward combinatorial reasoning (cf. [14]). The "surprising" fact is that this λ-term "explodes" by β-reduction. If we look at the process of reduction as the process of "understanding" one proof we can say that the "proof" of Girard's paradox becomes more and more complex when one try to understand it !

It is worth it to note that we use only a tiny part of the general polymorphism on types. The paradox appears as soon as we allow ourselves to consider the "polymorphic" types (as in ML, with one universal quantification) as real types. Note also this derivation seems very close in spirit to the derivation of [20] for a construction of a contradiction for a system of Quine, that uses a generalisation of the notion of "systematic ambiguity" of the Principia.

4 Applications of the previous construction

4.1 "Type is of type Type" is inconsistent

The first application is the inconsistency of the first calculus of Martin-Löf [14]. This can be seen as the generalisation of the calculus of Church (but, in an intuitionnistic framework), where we add that there is a type $Type$ of all types.

The precise formulation of this system is simply the one given in the second section, and replacing everywhere $Prop$ by $Type$. All the previous derivation of Girard's paradox can then be translated in this system, and we obtain thus a non-normalisable term in this system. This is the way Girard derives a paradox in the 1971 system of Martin-Löf.

There have been proposed some programming or specification languages which contains the idea of a type $Type$ of all types, together with the fact that this type is also of type $Type$ [2]. Though the termination property is not the primary concern of the computer programmer, it seems very important to study the "computationnal" relevance of Girard's Paradox (what seems to be lost forever is the possibility of doing proofs about programs in such systems).

4.2 CAT is not a CAT

It appears it is possible to apply the ideas used in the typed-checked paradox to derive a contradiction from the fact that there is a category of all categories.

Definition. Let C be a category, the order $R(C)$ is defined on $obj(C)$ by $R(C)(a,b)$ if, and only if, $C(a,b)$ is non void and $C(b,a)$ is void; then we shall say that C is *well-founded* if, and only if, $R(C)$ is a well-founded ordering.

Definition. Let C and D be two categories, then a functor F between C and D is said to be *dominated* if, and only if, there is an object y of D such that, for all object x of C, we have $R(C)(F(x),y)$.

Proposition. The "category" which has for objects all well-founded categories and for morphisms the identities and the dominated functors does not exist.

The reason is that we can reproduce the typed-checked example if such a category exists. This shows that it is not possible to add a special type "Cat" to the theory of types of Church which captures the idea of the category of all categories.

Conclusion

In view of this paradox, and of the considerations of the first section, it is natural to consider the restricted system where we do not allow the formation of product $\Pi(M,N)$ when M is *Type*. The system is exactly the system described in the second section, with the restricted product formation for the rule 2.2.4. It seems that we obtain by this way a complete formalisation of the system used in the Principia (but in an intuitionnistic framework) with the notion of "typical" or "systematic" ambiguity. This system can so be thought of as the language ML, where we add a special type *Prop*, which is the type of the specification of the programs. A more detailed study of this system will appear in [8].

Experiments with this system show that this is a good generalisation of the calculus of construction. We can now, for example, define "abstractly" the equivalence relation related to a preordering, and prove that this is an equivalence relation. The important point is that we can also instantiate this "abstract" situation on a "concrete" given preordering, by example the inclusion, and specialize both the definition and the proof, obtaining thus the concept of extentional equality, and the proof that this is an equivalence relation.

It seems thus that the difficulty raised by Girard's Paradox can be in practice solved in a consistent way.

All these considerations raise the following problem about the Curry-Howard isomorphism between propositions and types: is this really an isomorphism ? It seems that there is a problem of "levels". We have the choice a priori for the level of programs, as term of degree 3 or of degree 2. If we choose the degree 3, then we have the general polymorphism for programs (it is shown in [7] that we can add the dependent product in a consistent way). If we choose the degree 2, then we lose definitely the general polymorphism, but we have a clear set-theoretic semantics for the programs and a clear way for the development of proofs about programs in the construction system.

Girard's paradox seems to have some connections with the result of Reynolds [21] that there is no set-theoretical models. Actually, this can be used to show that there is no extensionnal models with a polymorphic notion of equality (cf. [11]), but it does not seem that it entails directly the

Reynolds's theorem, as his definition of what is a set-theoretic model has very weak conditions. It is likely that the derivation of Reynolds can produce a non-normalisable term in the general polymorphic calculus, but not shorter that the typed-checked one. More generally, this raises the following questions: is it possible to derive another kind of paradox in the general polymorphic calculus (for example, Russel's paradox) ? If possible, could the different ideas behind these paradoxes be characterized by the behavior of the corresponding λ-terms by reduction ?

The derivation of a paradox in the general polymorphic system can be seen as the syntactical counterpart of the fact that there is no set-theoretical model of the second-order typed calculus. This is morever perhaps an explanation of this non-existence: the typed systems showed in this paper are formal systems whose syntax is as complex as what is usually regarded as semantics (i.e. set-theory).

References

[1] H. Barendregt "The lambda -Calculus: Its Syntax and Semantics." North-Holland (1980).

[2] R. Burstall and B.Lampson "A kernel language for abstract data types and modules." Lecture Notes in Computer Science 173, Springer-Verlag, 1984

[3] N.G. de Bruijn "lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." Indag. Math. **34,5** (1972), 381–392.

[4] N.G. de Bruijn "A survey of the project Automath." in Curry Volume, Acc. Press (1980)

[5] N.G. de Bruijn "Some extensions of Automath: the Aut-4 family" unpublished paper (1974)

[6] A. Church "A formulation of the simple theory of types" Journal of Symbolic Logic (1940), 56–68

[7] Th. Coquand "Une Theorie des Constructions" these de 3eme cycle, Paris VII (1985)

[8] Th. Coquand "Modeles de la theorie des constructions" en preparation (1986)

[9] Th. Coquand, G. Huet "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[10] Th. Coquand, G. Huet "Concepts Mathematiques et Informatiques formalises dans le Calcul des Constructions" Papier presente au Colloque de Logique d'Orsay (1985)

[11] J.Y. Girard "Interpretation fonctionnelle et elimination des coupures de l'arithmetique d'ordre superieur" These d'Etat, Paris VII (1972)

[12] J. Lambek "From types to sets" Advances in mathematics 35 (1980)

[13] D.B. MacQueen and R. Sethi "A higher-order polymorphic type system for applicative languages" Symposium on Lisp and Functionnal Programming, 243–252 (1982)

[14] P. Martin-Löf "A Theory of Types" unpublished (1971)

[15] P. Martin-Löf "An intuitionistic theory of types: predicative part" Logic Colloquium, North-Holland (1975)

[16] P. Martin-Löf "intuitionistic Type Theory" Bibliopolis, (1980)

[17] A.R. Meyer and J.C. Mitchell "Second-order Logical Relations" extended abstract (1985)

[18] R. Milner "A theory of type polymorphism in programming" JCSS 17(3), 348-375 (1978)

[19] J.C. Mitchell "Lambda Calculus Models of Typed Programming Languages" Ph. D. thesis, M.I.T. (1984)

[20] J.B. Rosser "The Burali-Forti paradox" Journal of Symbolic Logic 7, 1–17 (1942)

[21] J.C. Reynolds "Polymorphism is not Set-Theoretic" Lecture Notes in Computer Science 173, Springer-Verlag, 1984

[22] B. Russel and A.N. Whitehead "Principia Mathematica" Volume 1,2,3 Cambridge University Press (1912)

[23] G. Takeuti "Proof Theory" North-Holland, part II (1975)