Extending the Calculus of Constructions with Type:Type

Gérard Huet

INRIA

[UNPUBLISHED DRAFT - 1988]

Introduction

The Calculus of Constructions is a higher-order formalism for writing constructive proofs in a natural deduction style, inspired from work of de Bruijn [1, 2], Girard [14] and Martin-Löf [20]. The calculus and its syntactic theory were presented in Coquand's thesis [7], and an implementation by the author was used to mechanically verify a substantial number of proofs demonstrating the power of expression of the formalism [10]. The Calculus of Constructions is proposed as a foundation for the design of programming environments where programs are developed consistently with formal specifications.

1 A summary of the Type:Type and predicativity controversy

Martin-Löf's original 1971 theory of types [20] was a very elegant formal system postulating that the type of all types (let us call it $Type$) was itself an object of type $Type$, that is the judgement $Type : Type$ was given as an axiom. Such a type inference system, however, is too permissive to be used as a logic. Specifically, Girard showed [14] that Burali-Forti's paradox could be encoded in the system, leading to logical inconsistency. The argument was later simplified and mechanically type-checked by Coquand [8]. We remark that this logical inconsistency may not preclude the practical use of such a system for type inference in a programming language. Indeed, Pebble[3] and Quest[4] are programming languages admitting $Type : Type$. There, inconsistency is not any worse than a non-terminating recursion.

Coquand's analysis of Girard's paradox showed that the problematic feature of the system with $Type : Type$ was due to its interference with the impredicative product formation rule:

$$\frac{\Gamma, x : M \vdash N : Type}{\Gamma \vdash (\Pi x : M)N : Type}$$

since the product over all types $(\Pi t : Type)T$ for any type T cannot be assumed without paradox to be of the same "size" (or better, of the same complexity) as an ordinary type. Indeed, the paradox appears as soon as one is allowed to abstract variables of type $Type$ (see section 9 of [8]). On the other hand, it is perfectly reasonable to allow the rule:

$$\frac{\Gamma, x : M \vdash N : Prop}{\Gamma \vdash (\Pi x : M)N : Prop}$$

where $Prop$ is the type of all propositions, since this is nothing more than universal quantification in higher-order logic. That is, we may be "non-predicative" for propositions, but must stay "predicative" for types. Thus the constant "*" from the original Calculus of Constructions (OCC) [7] has to be interpreted as $Prop$, not as $Type$.

It is possible to encode many mathematical constructions in OCC, as shown in computer experiments with an implementation of the calculus [10, 12]. However, it is awkward to develop for instance first-order reasoning over domain D in an environment $[D : *][x : D]$, since there is no reason to confuse the element x with a hypothetical proof of D seen as a proposition variable. In other words, the very useful paradigm of "propositions as types" should not be perverted into the reverse and misleading principle of "types as propositions" (which M. Beeson ridicules as an intellectual version of Peter's Principle).

In order to turn the calculus into a genuine type theory, the original notion of "context" has to be abstracted in the notion of $Type$, with $Prop : Type$. This extension can then be considered a genuine extension of Church's theory of types[6], except that the basic logic is intuitionistic rather than classical, expressed as a natural deduction system, with an explicit notation for proofs, and with dependent product types. Such a extended system (let us call it ECC) has been studied by Th. Coquand, who proved a normalization theorem expressing its logical consistency [9].

This very elegant formal system is however limited in its expressive power, in that quantification over $Type$ is not possible. Such type quantification is needed in order to have uniform proofs (this is the notion of "typical ambiguity" from the Principia). This lead Coquand to formulate a generalization of ECC with a predicative hierarchy of types, in the manner of Grothendieck universes, or of the predicative theory of types of Martin-Löf [22]. Here is Coquand's generalized system, as it appeared in [8].

2 Coquand's Generalized Calculus of Constructions : GCC

2.1 Terms

1. $Type(i)$, for i non-negative integer, and $Prop$ are terms
2. an identifier x is a term (free variables)
3. a non-negative integer k is a term (bound variables)
4. if M and N are terms, then $(M\ N)$ is a term (application)
5. if M is a term, then $[]M$ is a term (abstraction)
6. if M and N are terms, then $\Pi(M, N)$ is a term (product).

A word has to be said about variables. We assume that free variables are denoted by identifiers, whereas bound variables are denoted by their binding depth, or de Bruijn's index. The term $[]M$ denotes functional abstraction. We shall generally use the more usual concrete notation $\lambda x \cdot N$, which is an abbreviation for $[]M$, where M is N with every occurrence of identifier x replaced by the proper index. Similarly we define abstractly the substitution operation $\{P\}M$, which substitutes P with proper shifts for index 0 in M, and we write concretely instead $[P/x]N$. Thus β-reduction may be written abstractly as $([]M\ N) \rhd \{N\}M$, or concretely as $(\lambda x \cdot M\ N) \rhd [N/x]M$. In the following, we denote by $\equiv$ the relation of $\lambda\beta$-conversion between terms. The operator Π is product formation. It is binding in its second argument, and we shall use below the concrete notation $(x : M)N$ as an abbreviation for $\Pi(M, \lambda x \cdot N)$.

2.2 Environments

Environments are ordered lists of *bindings* of the form $x : M$, where x is a variable and M is a term. We assume without loss of generality that every variable occurs at most once in a given environment Γ, and we use the notation Γ_x for the corresponding M. We may thinf of Γ_x as the "type" of x in environment Γ. Not every environment is valid. The following rules define the valid environments.

$$\text{the empty environment is valid}$$

$$\frac{\Gamma \text{ is valid} \quad \Gamma \vdash M : \mathit{Prop} \quad x \text{ is not bound in } \Gamma}{\Gamma, x : M \text{ is valid}}$$

$$\frac{\Gamma \text{ is valid} \quad \Gamma \vdash M : \mathit{Type}(i) \quad x \text{ is not bound in } \Gamma}{\Gamma, x : M \text{ is valid}}$$

These rules are defined mutually recursively with the following type inference rules, which define the judgements $\Gamma \vdash M : N$, to be read "the term M is of type N in environment Γ".

2.3 Type Inference Rules

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash \mathit{Prop} : \mathit{Type}(0)}$$

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash \mathit{Type}(i) : \mathit{Type}(i+1)} \qquad (*)$$

$$\frac{\Gamma \vdash M : \mathit{Type}(i)}{\Gamma \vdash M : \mathit{Type}(i+1)} \qquad (\mathit{coerce})$$

$$\frac{\Gamma \text{ is valid} \quad x \text{ is bound in } \Gamma}{\Gamma \vdash x : \Gamma_x}$$

$$\frac{\Gamma, x : M \vdash N : P}{\Gamma \vdash \lambda x.N : (x : M)P}$$

$$\frac{\Gamma, x : M \vdash N : \mathit{Prop}}{\Gamma \vdash (x : M)N : \mathit{Prop}}$$

$$\frac{\Gamma \vdash M : \mathit{Type}(j) \quad \Gamma, x : M \vdash N : \mathit{Type}(i)}{\Gamma \vdash (x : M)N : \mathit{Type}(\mathit{max}(i,j))}$$

$$\frac{\Gamma \vdash M : \mathit{Prop} \quad \Gamma, x : M \vdash N : \mathit{Type}(i)}{\Gamma \vdash (x : M)N : \mathit{Type}(i)} \qquad (**)$$

$$\frac{\Gamma \vdash M : (x : Q)P \quad \Gamma \vdash N : R \quad Q \equiv R}{\Gamma \vdash (M\ N) : \mathtt{[}N/x\mathtt{]}P}$$

The only serious departure from [8] is the addition of rule $(*)$, which was inadvertently ommitted, and of rule $(**)$, which is needed to prove the following lemma.

Lemma. If $\Gamma \vdash M : N$ is derivable, then either $\Gamma \vdash N : \mathit{Prop}$ is derivable, in which case we say that M is a *proof* of *proposition* N in environment Γ, or else $\Gamma \vdash N : \mathit{Type}(i)$ is derivable for some $i \geq 0$, in which case we say that M is a *realization* of *specification* N in environment Γ.

This lemma shows that there are two distinct kinds of types in the system, in the sense of terms appearing to the right of a colon in a derivable sequent.

2.4 A digression on types, specifications and propositions

We say that term T is a *type* (in a given environment) if it is either a specification or a proposition. We remark that the rules for environment formation are that variables may be bound only to types, not to arbitrary terms. Since these are the two kinds of bindings, we shall speak of the constants $Prop$ and $Type(i)$ of the system as the *kinds*, following the MacQueen-Sethi terminology[19]. Specifications are the natural generalization of the notion of types in the sense of Church's theory of types. They are more general in that the product formation operator is *dependent*, like in Martin-Löf's theory of types [22]. When x does not occur in N, the specification $(x : M)N$ may be abbreviated in the more traditional $M \to N$. For instance, the specification of a predicate over type T would be $T \to Prop$. Similarly, when P is a proposition and Q is a proposition in which x does not occur, we may abbreviate $(x : P)Q$ in $P \Rightarrow Q$. Also, we use $\forall x : M \cdot P$ for $(x : M)P$ when M is a specification and P is a proposition. When P is a proposition and M is a specification, the specification $(x : P)M$ has realizations depending on the proof of P. It is not usual to consider such types in ordinary logic. However, they are needed to formalize constructive mathematics in Bishop's sense, where evidence of properties is taken as computationally meaningful. Here evidence (of properties) is internalized as proofs (of propositions). This is in contrast to the formalism LF (logical framework) developed at the University of Edinburgh [15], where judgements (as opposed to propositions) are types. We refer to [23] for a philosophical discussion of the issues involved.

Remark that the only specifications P which are typable of type $Type(0)$ in the empty environment are (convertible to) the *contexts*, terms of the form:

$$(x_1 : M_1)(x_1 : M_1)...(x_1 : M_1)Prop.$$

The types of the system are more general than just specifications, since we use the paradigm of propositions as types [16]. More precisely, the formulation of the logical part of the system in natural deduction style allows the use of λ-abstraction for the dual purpose of building functional realizations as well as building proofs under hypotheses.

The inference system is completed by type equality rules, as follows.

2.5 Type Equality Rules

$$\frac{\Gamma \vdash M : N \quad \Gamma \vdash P : Prop \quad N \equiv P}{\Gamma \vdash M : P}$$

$$\frac{\Gamma \vdash M : N \quad \Gamma \vdash P : Type(i) \quad N \equiv P}{\Gamma \vdash M : P}.$$

Note that we allow λ-conversion only for types, not for other terms.

Remark 1. It might seem that the previous lemma allows to simplify the two rules in one simpler rule:

$$\frac{\Gamma \vdash M : N \quad N \equiv P}{\Gamma \vdash M : P}.$$

However, we are careful to specify that P must be itself well-typed, since otherwise we might introduce non-typable terms as types of other terms. Indeed, we need this restriction in order to preserve the validity of the lemma above.

Remark 2. The types equality rules allow us to replace the rule of application by the simpler :

$$\frac{\Gamma \vdash M : (x : Q)P \quad \Gamma \vdash N : Q}{\Gamma \vdash (M\ N) : [N/x]P}$$

Indeed, this is the way it was formulated originally [8]. However, our formulation is more consistent from the point of view of the meaning of the meta-variables in the rules, since several occurrences of the same meta-variable should mean that the corresponding term or environment is *shared*, and this is not the case for Q above. Furthermore, our formulation helps in the following section.

Conjecture. With the new formulation of the application rule, the following rule is sound and sufficient :

$$\frac{\Gamma \vdash M : N \quad N \rhd P}{\Gamma \vdash M : P},$$

where $\rhd$ is λ-reduction. The soundness should come from the fact that reduction (as opposed to conversion) preserves typeability. The completeness should come from the Church Rosser property, and say something like $\Gamma \vdash M : N$ in the old system implies that $\Gamma' \vdash M' : N'$ in the new one, where $N \rhd N'$, $M \rhd M'$ in type subterms, and $\Gamma \rhd \Gamma'$.

The system GCC is quite powerful. It extends strictly Girard's higher order system F^ω. It permits to formalize completely the Principia's, including the so-called "typical ambiguity" feature. However, it is not very convenient to use, since we have to explicitly manipulate the universe hierarchy. Furthermore, there is no unicity of types (even modulo lambda-conversion), because of rule (*coerce*). It is the purpose of this paper to show how to solve this difficulty.

3 An Inconsistent Calculus of Constructions : ICC

The idea is quite simple: we just erase the integer arguments to the $Type$ operator.

3.1 Terms

1. $Type$ and $Prop$ are terms
2. an identifier x is a term (free variables)
3. a non-negative integer k is a term (bound variables)
4. if M and N are terms, then $(M\ N)$ is a term (application)
5. if M is a term, then $[]M$ is a term (abstraction)
6. if M and N are terms, then $\Pi(M, N)$ is a term (product).

3.2 Environments

$$\text{the empty environment is valid}$$

$$\frac{\Gamma \text{ is valid} \quad \Gamma \vdash M : Prop \quad x \text{ is not bound in } \Gamma}{\Gamma, x : M \text{ is valid}}$$

$$\frac{\Gamma \text{ is valid} \quad \Gamma \vdash M : Type \quad x \text{ is not bound in } \Gamma}{\Gamma, x : M \text{ is valid}}$$

3.3 Type Inference Rules

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash Prop : Type} \quad (TI1)$$

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash Type : Type} \quad (TI2)$$

$$\frac{\Gamma \text{ is valid} \quad x \text{ is bound in } \Gamma}{\Gamma \vdash x : \Gamma_x} \quad (TI3)$$

$$\frac{\Gamma, x : M \vdash N : P}{\Gamma \vdash \lambda x.N : (x : M)P} \quad (TI4)$$

$$\frac{\Gamma, x : M \vdash N : Prop}{\Gamma \vdash (x : M)N : Prop} \quad (TI5)$$

$$\frac{\Gamma \vdash M : Type \quad \Gamma, x : M \vdash N : Type}{\Gamma \vdash (x : M)N : Type} \quad (TI6)$$

$$\frac{\Gamma \vdash M : Prop \quad \Gamma, x : M \vdash N : Type}{\Gamma \vdash (x : M)N : Type} \quad (TI7)$$

$$\frac{\Gamma \vdash M : (x : Q)P \quad \Gamma \vdash N : R \quad Q \equiv R}{\Gamma \vdash (M\ N) : [N/x]P} \quad (TI8)$$

3.4 Type Equality Rule

$$\frac{\Gamma \vdash M : N \quad N \rhd P}{\Gamma \vdash M : P} \quad (TE)$$

The reader will notice the nice symmetry between the two "kinds" $Prop$ and $Type$. He will also notice that this system, when used without caution, will lead to inconsistency, along the lines of Girard-Coquand [8]. Let us now see how to use it with caution.

4 Type constraints

Consider a derivation tree of some judgement $\Gamma \vdash M : T$ in the system ICC above. We shall associate with it the following information. First we have a finite set of formal variables $\{x_1, ..., x_n\}$. We think of variable x_i as ranging over integers. Next we have a set of *constraints* on the formal variables, which are inequalities in terms of a partial ordering $<$. More precisely, we have constraints of the form $x_i < x_j$, of the form $x_i \leq x_j$, and of the form $x_i = x_j$. Think of these constraints $\{C_1, ..., C_p\}$ as expressing the arithmetic formula

$$\exists x_1 ... \exists x_n \cdot C_1 \wedge ... \wedge C_p.$$

Finally we associate with every occurrence of the constant $Type$ in the judgement $\Gamma \vdash M : T$ one of these formal variables. Several occurrences may be mapped on the same variable. In the following, we indicate $Type_i$ to show that the corresponding occurrence of $Type$ is mapped to the formal variable x_i.

We now show by induction on the derivation of the judgement how to maintain and update this information. First, it is understood that the set of formal variables always increases along a derivation, and that "residual" occurrences of $Type$ keep their mapping $Type_i$. By residual we mean the standard notion: occurrences inside the formulas matching the meta-variables M, N, P, Q, R, Γ

in the premisses of rules have their residuals in the corresponding occurrences in the conclusion of the rule. Also the residuals of N in $[N/x]M$ are the substituted occurrences, as usual. Finally, the residuals along λ-reductions are defined in the usual way.

Now, we consider each inference rule in turn. In this analysis, we assume that the rules are used in proof-checking fashion. We shall explain later how the consistency check may be computed in proof-synthesis mode.

First, the rules for environment formation just preserve constraints by residuals. Next are type inference rules. If rule *(TI1)* is used, we increment n by one, we add a new formal variable x_n, we derive $\Gamma \vdash Prop : Type_n$, and we add the constraint $0 \leq x_n$.

If rule *(TI2)* is used, we increment n by two, we add two new formal variable x_{n-1} and x_n, we derive $\Gamma \vdash Type_{n-1} : Type_n$, and we add the constraint $x_{n-1} < x_n$.

If rule *(TI3)* is used, we preserve the constraints by residuals, except in case x is bound to $Type_i$ in context Γ, in which case we increment n by one, we add a new formal variable x_n, we derive $\Gamma \vdash x : Type_n$, and we add the constraint $x_i \leq x_n$.

If one of the rules *(TI4)* or *(TI5)* is used, we just preserve constraints by residuals.

If rule *(TI6)* is used, we assume that the derivation of the first premiss $\Gamma \vdash M : Type_i$ gave a set of constraints which was used (with proper residuals in M and Γ) to derive the second premiss $\Gamma, x : M \vdash N : Type_j$ with the current set of constraints. We then increment n by one, we add a new formal variable x_n, we derive $\Gamma \vdash (x : M)N : Type_n$ and we add the two constraints $x_i \leq x_n$ and $x_j \leq x_n$.

If rule *(TI7)* is used, we assume that the derivation of the first premiss $\Gamma \vdash M : Prop$ gave a set of constraints which was used (with proper residuals in M and Γ) to derive the second premiss $\Gamma, x : M \vdash N : Type_i$ with the current set of constraints. We then increment n by one, we add a new formal variable x_n, we derive $\Gamma \vdash (x : M)N : Type_n$ and we add the constraint $x_i \leq x_n$.

If rule *(TI8)* is used, we assume that the derivation of the first premiss $\Gamma \vdash M : (x : Q)P$ gave a set of constraints which was used (with proper residuals in Γ) to derive the second premiss $\Gamma \vdash N : R$ with the current set of constraints. We then assume (without loss of generality by the Church-Rosser property) that the test $Q \equiv R$ is effected by reducing Q and R to identical forms. In these identical forms, every corresponding occurrence of say $Type_i$ in Q and $Type_j$ in R will generate a constraint $x_i = x_j$. Finally, we derive $\Gamma \vdash (M\ N) : [N/x]P$ with proper residuals. There is a special case when P is $Type_i$, in which case we increment n by one, we add a new formal variable x_n, we derive $\Gamma \vdash (M\ N) : Type_n$, and we add the constraint $x_i \leq x_n$.

Finally, if the rule *(TE)* is used, we keep the constraints by residual, except when P is $Type_i$, in which case we increment n by one, we add a new formal variable x_n, we derive $\Gamma \vdash M : Type_n$, and we add the constraint $x_i \leq x_n$.

[Note. Maybe this could be simplified a little bit. For instance, in TI7, do we need to consider a new variable, or could we consider the occurrence of Type in the conclusion to be a residual of the one in the hypothesis?]

5 Maintaining consistency

5.1 A consistency check

At every step, we may check in linear time that the set of constraints is satisfiable. This test amounts to showing that the set of constraints defines a partial ordering which may be extended to a linear ordering. It is well known how to do this using topological sort (see Knuth [18] section 2.2.3).

An extended version of this paper will discuss how to maintain this check incrementally.

5.2 Mutual consistency

Theorem. If $\Gamma \vdash M : N$ in ICC with satisfiable constraints, then $\Gamma \vdash M : N$ in GCC. (Or maybe with primes to take care of different equality rules) Proof: use the linear ordering given by the constraints satisfiability test to generate actual indexes to the $Type$ occurrences. Use the (*coerce*) rule when needed. Also the rule for application may need some type equality. Conversely, if $\Gamma \vdash M : N$ in GCC, then $\Gamma \vdash M : N$ in ICC (modulo type equality again). Proof: just delete all applications of coerce.

[PB: can the consistency of GCC be established by an extension of the methods of [9]? Thierry says this poses no conceptual problem.]

5.3 Exemples

With $U = (T : Type)(x : T)T$ and $\Gamma = [u : U]$, show that the derivation of $\Gamma \vdash ((u\ U)\ u)$ leads to an unsatisfiable constraint.

5.4 Discussion

What we have done is actually very simple: we have replaced the meta-variables denoting integers in expressions such as $Type(i)$ by formal variables. We treat these variables not by standard pattern-matching, but rather we keep them implicit, along with a network of constraints. This is consistent with the view of systems manipulating sets of inference rules as type constraints on derivation operators. Constraints are more general than existential equalities (solved as usual by unification), but involve inequalities over a linear ordering. It would be interesting to develop a general theory of such constraint processing in the framework of meta-logical systems such as G. Kahn's "Natural Semantics"[17].

Remark that the important point is that we shall be able to use proofs at various type levels. That is, the proof of a theorem is indexed with constraints pertaining to the formal variables it introduces. Different instances of use of the theorem will correspond to distinct copies of these variables, permitting use of the theorem at various levels simultaneously. In other words, different occurrences of constants are treated as different copies, not as residuals as in the case of variables. [This point has to be developed more fully. More generally, a mecanism for naming constants has to be described.]

6 A simplified system : SCC

We now exploit the (apparent) symmetry between the two kinds $Prop$ and $Type$ in order to simplify the inference rules, as seen by the user of the system. That is, we just factor together rules that are (apparently) symmetric with respect to the kinds.

6.1 Kinds

$Type$ and $Prop$ are kinds. We use the meta variable K to denote one of these two constants.

6.2 Terms

1. kinds are terms
2. an identifier x is a term (free variables)
3. a non-negative integer k is a term (bound variables)
4. if M and N are terms, then $(M\ N)$ is a term (application)
5. if M is a term, then $[]M$ is a term (abstraction)
6. if M and N are terms, then $\Pi(M, N)$ is a term (product).

6.3 Environments

$$\frac{\text{the empty environment is valid}}{\Gamma \text{ is valid} \quad \Gamma \vdash M : K \quad x \text{ is not bound in } \Gamma \over \Gamma, x : M \text{ is valid}}$$

6.4 Type Inference Rules

$$\frac{\Gamma \text{ is valid}}{\Gamma \vdash K : Type} \tag{$TI1$}$$

$$\frac{\Gamma \text{ is valid} \quad x \text{ is bound in } \Gamma}{\Gamma \vdash x : \Gamma_x} \tag{$TI2$}$$

$$\frac{\Gamma, x : M \vdash N : P}{\Gamma \vdash \lambda x.N : (x : M)P} \tag{$TI3$}$$

$$\frac{\Gamma, x : M \vdash N : K}{\Gamma \vdash (x : M)N : K} \tag{$TI4$}$$

$$\frac{\Gamma \vdash M : (x : Q)P \quad \Gamma \vdash N : R \quad Q \equiv R}{\Gamma \vdash (M\ N) : [N/x]P} \tag{$TI5$}$$

6.5 Type Equality Rule

$$\frac{\Gamma \vdash M : N \quad N \rhd P}{\Gamma \vdash M : P} \tag{TE}$$

Remark how this system is close both from Coquand's ECC [9], and from Martin-Löf's original system [20]. More precisely, it adds $Type : Type$ to the first, and the kind $Prop$ to the second.

Of course, the simplicity is apparent, since the symmetry exploited between the two kinds is not true of the constraints mechanism. The point is that most of the time the user should not have to worry about the levels of types. We have provided a global mechanism for checking the consistency of his derivations, as opposed to encoding this information explicitly in the inference rules. The casual user may think of $Type : Type$ exactly in the same way as he is used to think of set theory: in a naive way.

Of course, the problem is pushed to the situation when the user attempts a dangerous construction, which violates the constraints. That is, we shall have to find ways to tell him *why* is derivation is incorrect, and it may not be very easy to do so, because the problem is rather global, it is not simply to say that some inference step is invalid. More experimentation is needed before proposing practical solutions to this crucial problem of explaining the user what goes wrong in his uses of types.

6.6 A general remark on the various systems above

In all the inference systems considered above, the point of view was to describe *derivation operators*, that is operators typed with judgements such as our sequents $\Gamma \vdash M : N$. This point of view is sufficient to explain how to type-check the derivation of such a sequent. Thus we formulate λ-abstraction as untyped, since from $\Gamma \vdash \lambda x.N : (x : M)P$ we may extract the type M of x. If the point of view is rather to describe a proof-checker for terms, we have to regard a sequent $\Gamma \vdash M : N$ as the specification of an algorithm which, givem Γ and M, computes N. If we want to emphasize this point of view, we have to change our term structure so that abstraction is now a binary operator $\Lambda(M, N)$ binding in its second argument and then (using Automath's concrete syntax $[x : M]N$) the abstraction inference rule should have as conclusion: $\Gamma \vdash [x : M]N : (x : M)P$.

This minor difference is of no consequence for the meta-mathematical properties of the systems concerned, but it has occasionnally caused difficulties to readers. We do not feel that this issue is very important, since what we are interested in practice is in still another point of view: to regard a sequent $\Gamma \vdash M : N$ as a non-deterministic specification for a potential proof (resp. realization) M of a conjecture (resp. specification) N. In this view, N is given, M is computed non-deterministically, and Γ is originally given, and dynamically augmented with a computed suffix.

[PB. How do the constraints mix with this proof synthesis point of view?]

References

[1] N.G. de Bruijn. "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[2] N.G. de Bruijn. "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[3] R. Burstall and B. Lampson. "A Kernel Language for Modules and Abstract Data Types." International Symposium on Semantics of Data Types, Sophia-Antipolis. Proceedings Eds. G. Kahn, D. B. MacQueen and G. Plotkin, Springer-Verlag LNCS 173, 1–50.

[4] L. Cardelli. "A Polymorphic λ-calculus with Type:Type." DEC SRC Report 10 (May 1986).

[5] L. Cardelli and P. Wegner. "On Understanding Types, Data abstraction, and Polymorphism." ACM Computing Surveys **17** (Dec. 1985) 471–522.

[6] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic **5,1** (1940) 56–68.

[7] Th. Coquand. "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[8] Th. Coquand. "An analysis of Girard's paradox." First IEEE Symposium on Logic in Computer Science, Boston (June 1986), 227–236.

[9] Th. Coquand. "Metamathematical Investigations of a Calculus of Constructions." Submitted to Theoretical Computer Science.

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

[11] Th. Coquand and G. Huet. "The Calculus of Constructions." To appear, Information and Control.

[12] Th. Coquand and G. Huet. "Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions." Logic Colloquium, Orsay (July 85). To appear, North-Holland.

[13] A. Demers and J. Donahue. "Datatypes, parameters and type checking." 7th ACM Symposium on Principles of Programming Languages, Las Vegas (1980), 12–23.

[14] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[15] R. Harper, F. Honsell and G. Plotkin. "The Edinburgh Logical Framework." Private communication (Oct. 1986).

[16] W. A. Howard. "The formulæ-as-types notion of construction." Unpublished manuscript (1969). Reprinted in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds J. P. Seldin and J. R. Hindley, Academic Press (1980).

[17] "Natural Semantics." Private communication (Dec. 1986).

[18] D. Knuth. "The Art of Programming", Vol 1, Addison-Wesley (1968).

[19] D. B. MacQueen, R. Sethi. "A semantic model of types for applicative languages." ACM Symposium on Lisp and Functional Programming (Aug. 1982).

[20] P. Martin-Löf. "A Theory of Types." Report 71-3, Dept. of Mathematics, University of Stockholm, Feb. 1971, revised (Oct. 1971).

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

[22] P. Martin-Löf. "Intuitionistic Type Theory." Studies in Proof Theory, Bibliopolis (1984).

[23] P. Martin-Löf. "Truth of a proposition, evidence of a judgement, validity of a proof." Transcript of talk at the workshop "Theories of Meaning", Centro Fiorentino di Storia e Filosofia della Scienza, Villa di Mondeggi, Florence (June 1985).

[24] A. R. Meyer and M. B. Reinhold. "Type is Not a Type: Preliminary Report." Proceedings, Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida (Jan. 1986) 287–295.

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