The calculus of constructions

T. Coquand, Gérard Huet

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 530

THE CALCULUS OF CONSTRUCTIONS

Thierry COQUAND
Gérard HUET

Mai 1986

The Calculus of Constructions

Thierry Coquand and Gérard Huet

INRIA

Résumé

Nous présentons le Calcul des Constructions, un formalisme d'ordre supérieur approprié au développement de preuves constructives dans un style de déduction naturelle. Chaque preuve est une λ-expression, typée avec les propositions de la logique sous-jacente. En effaçant les types on obtient un λ-terme pur, qui représente l'algorithme associé à la preuve. La mise en forme normale de ce λ-terme correspond à l'élimination des coupures.

Nous pensons que la correspondance de Curry-Howard entre les propositions et les types est un guide conceptuel important pour Informatique. Dans le cas des Constructions, nous obtenons un language fonctionnel de très haut niveau, avec une notion de polymorphisme appropriée au développement d'algorithmes modulaires. La notion de type comprend la notion usuelle de type de donnée, mais autorise aussi bien l'expression de spécifications algorithmiques arbitrairement complexes.

Nous développons la théorie de base d'un Calcul des Constructions, et prouvons un résultat de normalisation forte impliquant la terminaison des calculs, et la cohérence de la logique. Finalement, nous suggérons diverses extensions possibles.

Une version préliminaire de ce papier a été présentée au Colloque International sur les Types de Sophia-Antipolis en Juin 1984.

Abstract

We present the Calculus of Constructions, a higher-order formalism for constructive proofs in natural deduction style. Every proof is a λ-expression, typed with propositions of the underlying logic. By removing types we get a pure λ-expression, expressing its associated algorithm. Computing this λ-expression corresponds roughly to cut-elimination.

It is our thesis that the Curry-Howard correspondance between propositions and types is a powerful paradigm for Computer Science. In the case of Constructions, we obtain the notion of a very high-level functional programming language, with complex polymorphism well-suited for modules specification. The notion of type encompasses the usual notion of data type, but allows as well arbitrarily complex algorithmic specifications.

We develop the basic theory of a Calculus of Constructions, and prove a strong normalization theorem showing that all computations terminate. The logical consistency is obtained as a corollary. Finally, we suggest various extensions to stronger calculi.

A preliminary version of this paper was presented in June 1984 at the International Symposium on Semantics of Data Types in Sophia-Antipolis.

The Calculus of Constructions

Thierry Coquand and Gérard Huet

INRIA

Introduction

The Calculus of Constructions is a higher-order formalism for constructive proofs in natural deduction style. Every proof is a λ-expression, typed with propositions of the underlying logic. By removing types we get a pure λ-expression, expressing its associated algorithm. Computing this λ-expression corresponds roughly to cut-elimination. It is our thesis that (as already advocated by Martin-Löf [36]) the Curry-Howard correspondance between propositions and types is a powerful paradigm for Computer Science. In the case of Constructions, we obtain the notion of a very high-level functional programming language, with complex polymorphism well-suited for modules specification [8]. The notion of type encompasses the usual notion of data type, but allows as well arbitrarily complex algorithmic specifications. We develop the basic theory of a Calculus of Constructions, and prove a strong normalization theorem showing that all computations terminate. Finally, we suggest various extensions to stronger calculi.

1 The abstract syntax of terms

Our term structures are inspired by the Automath formalisms. A term is a λ-expression, where variables are typed with types which are themselves terms of the same nature. Lambda-abstraction is written $(\lambda x : N)M$. The name x is of course completely irrelevant, and belongs only to the concrete syntax of the term. Abstractly, this binding operator is unary. Occurrences of x in the concrete syntax of term M will be replaced by de Bruijn's indexes [4], i.e. integers denoting the reference depth of the occurrence. Thus the string $(\lambda x : M)(\lambda y : N)((x\ y)\ x)$ represents concretely the abstract term $\lambda(M, \lambda(N, (2\ 1)\ 1))$. That is, the integer n denotes the variable bound at the nth binder upward in the term. As usual in combinatory logic we write $(M\ N)$ for the application of term M to term N (for a survey of the λ-calculus with this notation see [4], or [1]).

We have a second binding operator for representing products: $[x : A]M$ where A is a (type) term denoting the domain over which x ranges. The same convention holds for this operator. Thus, the string $[x : A]\,[y : B](x\ y)$ represents concretely the abstract term $[A]\,[B](2\ 1)$.

Our term algebra is completed by a constant $*$, which plays the role of the universe of all types. In Automath languages, $*$ is written *prop* or *type*. There is no analogue (in this version) of the term τ of Automath which represents the common type of both *prop* and *type* (see de Bruijn, [6]). Here, in the spirit of the so-called "Curry-Howard isomorphism", $*$ must be thought of as the type of all types, and the type of all propositions. However, note that there is no circularity here: $*$ is not of type $*$, though we shall see that this system is powerful enough to share many features with systems possessing a type of all types (but still with a normalisation property [16]).

Terms formed solely of products over $*$ are distinguished and called *contexts*. They are the types of logical propositions and proposition schemas, all other terms being called *objects*. We shall thus see that there are two kinds of types: the usual ones (which are certain objects, and more precisely the objects of type $*$) and contexts, which are in a way "types of types", and whose

role is to support full polymorphism.

Because of the binding operators, our term structures do not form a free algebra. We must ensure that in $[x : A]M$ the variable x may occur in M, but not in A. To do this, we now define precisely the set Λ^n of terms legal in a context of depth n.

Definition. We define the set of *terms* as:

$$\Lambda = \Lambda_\iota \cup \Lambda_o$$

where Λ_ι is the set of *objects* (or individual types)

$$\Lambda_\iota = \cup_{n\geq 0}\Lambda_\iota^n$$

and Λ_o is the set of *contexts* (or logical types)

$$\Lambda_o = \cup_{n\geq 0}\Lambda_o^n$$

where the Λ_o^n are the sets generated by the following inductive rules

$$* \in \Lambda_o^n \qquad \textit{universe}$$

$$[x : M]N \in \Lambda_o^n \text{ if } M \in \Lambda^n, N \in \Lambda_o^{n+1} \qquad \textit{quantification}$$

and Λ_ι^n are the sets generated by the inductive rules

$$k \in \Lambda_\iota^n \text{ if } 1 \leq k \leq n \qquad \textit{variables}$$

$$[x : M]N \in \Lambda_\iota^n \text{ if } M \in \Lambda^n, N \in \Lambda_\iota^{n+1} \qquad \textit{product}$$

$$(\lambda x : M)N \in \Lambda_\iota^n \text{ if } M \in \Lambda^n, N \in \Lambda_\iota^{n+1} \qquad \textit{abstraction}$$

$$(M\ N) \in \Lambda_\iota^n \text{ if } M, N \in \Lambda_\iota^n \qquad \textit{application}$$

and

$$\Lambda^n = \Lambda_o^n \cup \Lambda_\iota^n.$$

The *closed terms* are the terms in Λ^0, legal in the empty context $*$. We shall usually view contexts as lists of terms, with quantification the list constructor. A context Γ has an associated length $|\Gamma|$ defined by:

$$|*| = 0$$

$$|[x : M]\Delta| = 1 + |\Delta|.$$

Now if $\Gamma \in \Lambda_o^m$ and $\Delta \in \Lambda_o^n$ with $n = |\Gamma|$, we define the concatenation $\Gamma;\Delta$ as the context in Λ_o^m defined as Δ if $\Gamma = *$ and as $[x : M]\Delta'$ if $\Gamma = [x : M]\Gamma'$ and $\Delta' = \Gamma';\Delta$. Finally, if $\Gamma \in \Lambda_o^0$ and $M \in \Lambda^n$ with $n = |\Gamma|$, we use the notation $\Gamma[x : M]$ for the closed context $\Gamma;[x : M]*$. Thus a closed context Γ of length $n > 0$ may be written as

$$\Gamma = [x_n : \Gamma_n]\,[x_{n-1} : \Gamma_{n-1}]\cdots[x_1 : \Gamma_1]$$

with $\Gamma_i \in \Lambda^{n-i}$. (The justification for this numbering is that in this context the de Bruijn index 1 refers to variable x_1).

We shall generally use meta-variables Γ, Δ for contexts and A, B, M, N, P, Q for terms in general (i.e. objects and contexts). However, Γ_i will denote the i-th element of context Γ as defined above, and this may be an object or a context.

Object terms will serve to denote logical propositions, as well as individual and functional terms. The latter may be thought of as *proofs* of their types, according to the Curry-Howard isomorphism [26]. In particular, we shall interpret $(\lambda x : M)N$ as a proof of $[x : M]P$ when N is a proof of P under hypothesis M. Hence $[x : M]N$ can be thought of as the universal quantification $(\forall x : M)N$ as in the polymorphic λ-calculus [24,40]. But it can represent also the product $(\Pi x : M)N$. We adopt a uniform notation for the abstraction over what is usually viewed as "proofs" (corresponding to an introduction rule of natural deduction), and the abstraction over what is usually viewed as "terms" (corresponding to functional abstraction). In the same way, we adopt a uniform notation for application, denoting both modus ponens and functional application. More explicitly, we consider that the λ-notation, which was originally used for building terms of higher-order logic [10], is also very well-suited as a notation for proofs, and our formalism tries to reflect this fact. Note that our class of terms is almost identical to the one considered in Martin-Löf's original theory of types [32], but our syntactic treatment is closer to that of the Automath languages, and particularly to Jutting's [29].

Formally we need a relocation operation, which replaces every free variable k of a term M in Λ^n by $k+1$, obtaining a term M^+ in Λ^{n+1}. Formally, M^+ is defined as $\xi_0(M)$ where ξ is defined recursively by:

$$\begin{aligned}
\xi_i(*) &= * \\
\xi_i([x : M]N) &= [x : \xi_i(M)]\,\xi_{i+1}(N) \\
\xi_i(k) &= k \text{ if } k < i \text{ and } k+1 \text{ otherwise} \\
\xi_i((\lambda x : M)N) &= (\lambda x : \xi_i(M))\xi_{i+1}(N) \\
\xi_i((M\ N)) &= (\xi_i(M)\ \xi_i(N)).
\end{aligned}$$

Before giving the type inference system, we need one more notation. Assume that Γ is a context of length n: $\Gamma = [\Gamma_n]\,[\Gamma_{n-1}]\ldots[\Gamma_1]$. For any variable k bound in Γ, i.e. $1 \le k \le n$, we define Γ/k as $add(k, \Gamma_k)$ where we define recursively the operation add by $add(0, M) = M$ and $add(i+1, M) = (add(i, M))^+$. Basically, Γ/k is Γ_k "seen" in context Γ.

A general comment on our notations: we use de Bruijn indexes for a rigorous and complete presentation of our rules, and we prefer them to the usual solution of Curry with renaming of bound variables. But we never need them in the "concrete" syntax used in the presentation of the rules of construction, so these can be read in the usual way. For simplicity of notations, we shall still write $\lambda x \cdot N$ for $(\lambda x : M)N$ if the type M of x is determined from the context.

2 A first attempt at a construction calculus

We are now ready to define the calculus. It is an inductive definition of two relations. The first one, $\Gamma \vdash \Delta$, is a binary relation between contexts. The second one, $\Gamma \vdash M : P$, is a ternary relation between a context and two terms. We ensure that, when Γ is a context of length n, if $\Gamma \vdash \Delta$ we have $\Delta \in \Lambda_o^n$, and if $\Gamma \vdash M : P$, we have $M \in \Lambda_t^n$ and $P \in \Lambda^n$.

Intuitively, $\Gamma \vdash \Delta$ means that the context Δ is valid in the (valid) context Γ, and $\Gamma \vdash M : P$ means that, in the valid context Γ, M is a well-typed term of type P. Here contexts are terms which are types, but are themselves non typable. For instance, $*$ is not of type $*$, and actually, $*$ has no type at all. In the terminology of Martin-Löf, we have thus two kinds of "judgements". One is the judgement that a context is valid, and the other that a term is the type of another term. In particular, $\Gamma \vdash *$ says that the context Γ is valid. We may think of $\Gamma \vdash M : \Delta$ as saying that in the context Γ, M is a well-formed proposition schema over a list of parameters declared by Δ. Finally,

we may think of $\Gamma \vdash M : P$, where P is an object, as saying that, in the context of hypothesis Γ, M is a well-formed proof of proposition P. This terminology is consistent, since we shall see that if $\Gamma \vdash M : P$, where P is a term, then $\Gamma \vdash P : *$. We abbreviate $* \vdash \Delta$ as $\vdash \Delta$ and $* \vdash M : P$ as $\vdash M : P$.

2.1 The inference system of constructions

Under this intuitive interpretation, all the following rules appear natural. We first give the rules for construction of valid contexts:

$$\frac{}{\vdash *}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[x : \Delta] \vdash *}$$

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

Next we give rules for product formation:

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

$$\frac{\Gamma[x : M_1] \vdash M_2 : *}{\Gamma \vdash [x : M_1]M_2 : *}$$

Finally we give rules corresponding to the variables, abstraction and application:

$$\frac{\Gamma \vdash *}{\Gamma \vdash l : \Gamma/l} \quad (l \leq |\Gamma|) \qquad \textit{variable}$$

$$\frac{\Gamma[x : M_1] \vdash M_2 : P}{\Gamma \vdash (\lambda x : M_1)M_2 : [x : M_1]P} \qquad \textit{abstraction}$$

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

In the last rule $[N/x]Q$ denotes the term obtained by substituting the term N for the index 1 in the term Q. This operation may be formally defined without difficulty [4].

2.2 Discussion

Some remarks are called for, since the extreme conciseness of this formalism may be an obstacle for its understanding. What is the purpose of such a system?

The main point is that these simple rules give us a complete presentation of both the higher-order logic and the higher-order functional system of J.Y. Girard. This system prodives a very concise notation for proofs in higher-logic presented in a natural deduction framework, while at the same time it can also be viewed as a description of a type-checker for a programming language with a powerful class of types. Note that we have in this system a uniform presentation for terms, proofs and types.

It is important to note that this kind of system contains a "built-in" notion of realisability, since the notations for proofs are precisely the λ-terms. Since it is possible [31] to give a translation of Topos Theory into Church's calculus, we obtain in this way a notion of realisability for Topos Theory. But our formalism is more general [16].

Syntactically, there are two kinds of types: the contexts, and the terms of type $*$, which we shall call propositions. The rules of type formation express that these two classes are closed under product formation. One can see the types which are terms of type $*$ as the expression of the proposition-as-types principle: in this formalism, we identify a proposition (terms of type $*$), with its associated type of proofs. Here, it is useful to think about Martin-Löf's system ([34]) with one universe: the term U_0 of Martin-Löf's system, which represents the type of all small types, is the analogue of our $*$. But what is new here, is that the set of all "small" types is closed by products over all types. For instance, in Martin-Löf's system, $[A : U_0]A$ is of type U_1 and not of type U_0. Here, $[A : *]A$ is of type $*$. Thus, although our calculus appears to be close to the one of Martin-Löf, we encompass the non-predicative calculus of the Principia [48] with the axiom of reducibility, as well as the second-order calculus of Girard-Reynolds [23,40].

Let us make this point more precise. We adopt the following abbreviations for the arrow: if x does not appear free in M then we write $M \to N$ for $[x : M]N$. Also we use the abbreviation $\forall A \cdot B$ for $[A : *]B$. It is then straightforward to translate the system of Girard-Reynolds in our notation. For example, the generic identity, written $(\Lambda\alpha) \cdot (\lambda x : \alpha) \cdot x$ by Reynolds, becomes here $\lambda\alpha \cdot \lambda x \cdot x$. The type of this term, which is written $(\Delta\alpha)(\alpha \to \alpha)$ by Reynolds, is here $[\alpha : *][x : \alpha]\alpha$, or, with the previous abbreviations, $\forall\alpha \cdot \alpha \to \alpha$, that is, we infer $\vdash \lambda\alpha \cdot \lambda x \cdot x : [\alpha : *][x : \alpha]\alpha$.

2.3 The need for conversion rules

This formalism is entirely self-contained, since even its linguistic aspect (the traditional notion of well-formed formulæ) is axiomatised within it. The λ-abstraction is used in particular for forming propositional schemas, which could in turn become arguments of such schemas for higher-order reasoning. We need now to complement this basic formalism with rules allowing for instantiation of these schemas, that is conversion rules.

Indeed, problems appear when one tries to translate the third-order (and higher) aspect of Girard's system. Note that these higher-order terms did not appear in Reynolds' system, but are present in the work of McCracken [37]. For instance, if we try to internalize our abbreviation of the arrow as a definition, we define $\to$ as the term $\lambda A \cdot \lambda B \cdot [x : A]B$ of type $\forall A \cdot \forall B \cdot *$. But this is not sufficient: if t is of type $((\to\ A)\ B)$ and u of type A, then our rules do not allow the application of t to u. What is lacking here is a rule of type conversion: $((\to\ A)\ B)$ is β-convertible to $[x : A]B$, and so, one wants to say that t, of type $((\to\ A)\ B)$, is also of type $[x : A]B$.

We are thus going to extend our previous calculus with conversion rules. In the terminology of Martin-Löf, we introduce another kind of judgement $M \cong N$, whose intuitive meaning is that the terms M and N denote the same object. There are many possibilities for doing this, and the one presented here deals only with conversion rules at the level of types (and not at all levels, as in [15]). This presentation has the advantage that the proof of decidability of this inference system is easier.

3 The calculus with conversion rules

We thus add to the previous calculus another kind of sequent, of the form $\Gamma \vdash M \cong N$, where B is a context and M and N are terms. We try to formalise the notion of "logical" conversion between types.

3.1 The conversion rules

Definition. $\cong$ is the smallest congruence over propositions and contexts containing β-conversion, i.e. the relation defined by the following inductive rules:

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta \cong \Delta}$$

$$\frac{\Gamma \vdash M : \Delta}{\Gamma \vdash M \cong M} \quad (*)$$

$$\frac{\Gamma \vdash M \cong N}{\Gamma \vdash N \cong M}$$

$$\frac{\Gamma \vdash M \cong N \quad \Gamma \vdash N \cong P}{\Gamma \vdash M \cong P}$$

$$\frac{\Gamma \vdash P_1 \cong P_2 \quad \Gamma[x : P_1] \vdash M_1 \cong M_2}{\Gamma \vdash [x : P_1]M_1 \cong [x : P_2]M_2}$$

$$\frac{\Gamma \vdash P_1 \cong P_2 \quad \Gamma[x : P_1] \vdash M_1 \cong M_2 \quad \Gamma[x : P_1] \vdash M_1 : \Delta_1}{\Gamma \vdash (\lambda x : P_1)M_1 \cong (\lambda x : P_2)M_2} \quad (*)$$

$$\frac{\Gamma \vdash (M\ N) : \Delta \quad \Gamma \vdash M \cong M_1}{\Gamma \vdash (M\ N) \cong (M_1\ N)} \quad (*)$$

$$\frac{\Gamma \vdash (M\ N) : \Delta \quad \Gamma \vdash N \cong N_1}{\Gamma \vdash (M\ N) \cong (M\ N_1)} \quad (*)$$

$$\frac{\Gamma[x : P] \vdash M : \Delta \quad \Gamma \vdash N : P}{\Gamma \vdash ((\lambda x : P)M\ N) \cong [N/x]M} \quad (*)$$

the last rule being the most important (note that we restrict ourselves to the reduction of "logical" redexes), and we finally add a rule of type conversion:

$$\frac{\Gamma \vdash M : P \quad \Gamma \vdash P \cong Q}{\Gamma \vdash M : Q}.$$

With these rules of conversion, we obtain what may be called the "restricted" system of constructions (by opposition to the system of [15], where the conversion is allowed at all the levels). One advantage of this system is the fact that the decidability of all "judgements" can be proved independently of the normalisation theorem.

Definition. The *restricted* calculus of constructions is the typed system defined by the rules of the previous section and the previous rules of conversion. The (full) calculus of constructions is the typed system defined by the restricted calculus where the starred rules are replaced by the following rules:

$$\frac{\Gamma \vdash M : N}{\Gamma \vdash M \cong M}$$

$$\frac{\Gamma \vdash P_1 \cong P_2 \quad \Gamma[x : P_1] \vdash M_1 \cong M_2 \quad \Gamma[x : P_1] \vdash M_1 : N_1}{\Gamma \vdash (\lambda x : P_1)M_1 \cong (\lambda x : P_2)M_2}$$

$$\frac{\Gamma \vdash (M\ N) : P \quad \Gamma \vdash M \cong M_1 \quad \Gamma \vdash N \cong N_1}{\Gamma \vdash (M\ N) \cong (M_1\ N_1)}$$

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

where we allow β-reduction over arbitrary λ-terms (and this is the system presented in [15]).

Example. We want to define the intersection of a class of classes on a given type A_0. A natural attempt is to take (where we omit types for legibility):

$$inter = \lambda\alpha.\ \lambda x.[P : A_0 \to *](\alpha\ P) \to (P\ x)$$

so that $(inter\ \alpha)$ is of type $A_0 \to *$ whenever α is of type $(A_0 \to *) \to *$. Assume that α_0 is a given term of type $(A_0 \to *) \to *$, P_0 is a given term of type $A_0 \to *$ and we have a proof p_0 of type $(\alpha_0\ P_0)$. We shall build a proof of the inclusion of $(inter\ \alpha_0)$ in P_0. Let $x : A_0$ and $h : (inter\ \alpha_0\ x)$. We have to build with p_0, x, h, P_0, α_0 a term of type $(P_0\ x)$.

Intuitively, h which is of type $(inter\ \alpha_0\ x)$ is also of type $[P : A_0 \to *](\alpha_0\ P) \to (P\ x)$ (by "definition" of *inter*), so that the answer must be the term $(h\ P_0\ p_0)$. We have applied here a conversion rule. Let *subset* be the term $\lambda P \cdot \lambda Q \cdot [x : A_0]P(x) \to Q(x)$ of type $[P : A_0 \to *][Q : A_0 \to *]*$. We can infer that the term $\lambda p_0 \cdot \lambda x \cdot \lambda h.(h\ P_0\ p_0)$ is of type $(\alpha_0\ P_0) \to (subset\ (inter\ \alpha_0)\ P_0)$. This example shows that the rules of type conversion are absolutely needed as soon as one wants to develop mathematical proofs (note that this example can be developed in the restricted calculus as well as in the full calculus). The need for conversion rules is equally emphasized in [35] and [43].

3.2 A few properties of this calculus

In the following statements, the meta-variable E denotes an arbitrary judgement (which may be of the form Δ, $M : P$ or $M \cong N$). All these lemmas are valid for the restricted calculus as well as for the full calculus. First, we need some lemmas which are provable by induction on derivations.

Lemma 1. If $\Gamma \vdash E$, then $\Gamma \vdash *$, and more precisely, every derivation of $\Gamma \vdash E$ contains a subderivation of $\Delta \vdash *$ for all Δ a prefix of Γ.

Lemma 2. If $\Gamma[x : P]\Delta \vdash E$ and $\Gamma \vdash M : P$, then $\Gamma[M/x]\Delta \vdash [M/x]E$.

Lemma 3. If $\Gamma \vdash M : P$ and $\Gamma \vdash M \cong N$, then $\Gamma \vdash N : P$.

Lemma 4. If $\Gamma \vdash M : N$, and N is an object, then $\Gamma \vdash N : *$, and if $\Gamma \vdash M : \Delta$, then $\Gamma \vdash \Delta$.

Thus, the only types are propositions and contexts, and the type of a valid term is a valid term. Finally, we may show that types are unique, up to conversion:

Lemma 5. If $\Gamma \vdash M : N_1$ and $\Gamma \vdash M : N_2$, then $\Gamma \vdash N_1 \cong N_2$.

All the proofs are straightforward and given in full in [15].

Definition. Let Γ be a context such that $\Gamma \vdash *$, then a Γ-*proposition*, or Γ-type, is a term M such that $\Gamma \vdash M : *$. A Γ-*context* is a context Δ such that $\Gamma \vdash \Delta$. Finally, a Γ-*proof*, or Γ-*functional*, is a term M such that there exists a Γ-*type* P such that $\Gamma \vdash M : P$

For the **restricted** calculus, we can state directly:

Proposition. The relation $\Gamma \vdash E$ between contexts and judgements is decidable in the restricted calculus of constructions.

The proof in all details is rather long, but the main idea is simple, and its development straightforward. One defines first the notion of *reduction* $\triangleright$ associated to our notion of conversion as:

$$\frac{\Gamma \vdash M \ \triangleright\ N \quad \Gamma \vdash N \ \triangleright\ P}{\Gamma \vdash M \ \triangleright\ P}$$

$$\frac{\Gamma[x:P] \vdash M \ \triangleright \ N}{\Gamma \vdash [x:P]M \ \triangleright \ [x:P]N}$$

$$\frac{\Gamma[x:P] \vdash M \ \triangleright \ N \quad \Gamma[x:P] \vdash M:\Delta}{\Gamma \vdash (\lambda x:P)M \ \triangleright \ (\lambda x:P)N}$$

$$\frac{\Gamma[x:P] \vdash M:* \quad \Gamma \vdash P \triangleright Q}{\Gamma \vdash [x:P]M \ \triangleright \ [x:Q]M}$$

$$\frac{\Gamma[x:P] \vdash M:\Delta \quad \Gamma \vdash P \triangleright Q}{\Gamma \vdash (\lambda x:P)M \ \triangleright \ (\lambda x:Q)M}$$

$$\frac{\Gamma[x:P] \vdash \Delta \quad \Gamma \vdash P \triangleright Q}{\Gamma \vdash [x:P]\Delta \ \triangleright \ [x:Q]\Delta}$$

$$\frac{\Gamma \vdash (M\ N):\Delta \quad \Gamma \vdash M \ \triangleright \ M_1}{\Gamma \vdash (M\ N) \ \triangleright \ (M_1\ N)}$$

$$\frac{\Gamma \vdash (M\ N):\Delta \quad \Gamma \vdash N \ \triangleright \ N_1}{\Gamma \vdash (M\ N) \ \triangleright \ (M\ N_1)}$$

$$\frac{\Gamma[x:P] \vdash M:\Delta \quad \Gamma \vdash N:P}{\Gamma \vdash ((\lambda x:P)M\ N) \ \triangleright \ [N/x]M}.$$

Then the usual argument of normalisation for the (simply) typed λ-calculus applies, with the notion of *complexity* of a term defined as follows.

Definition. The logical *rank* $\delta(M)$ of a term M is defined by the inductive rules:

1. $\delta(M) = 0$, if M is an object
2. $\delta(*) = 1$
3. $\delta([x:M]\Gamma) = max(\delta(M)+1, \delta(\Gamma))$.

Lemma 6. If $\Gamma \vdash M \cong N$, then $\delta(M) = \delta(N)$.

This lemma shows that all the types of a constructed object have the same rank, and it allows the definition:

Definition. Let $\Gamma \vdash ((\lambda x:P)M\ N):\Delta$ be a constructed (logical) redex. We define the *complexity* of the redex, as the rank $\delta(P)$. The complexity of a construction $\Gamma \vdash M:N$ is then defined as the multiset of complexities of all its logical redexes.

Note then that this complexity decreases by innermost reduction, whence the existence of a normal form, and the decidability of the conversion relation. The normalisation property of $\triangleright$ entails the decidability of $\Gamma \vdash E$.

Theorem. Given Γ and M, it is decidable whether or not there exists a term N such that $\Gamma \vdash M:N$. Furthermore, if the answer is positive, we can compute effectively such an N.

The proof is an induction on the sum of the length of M and the length of Γ, as in [32].

The reduction rules above correspond to the notion of instantiation for predicate variables (see [47] for a more traditional presentation). Strong normalisation also holds, and this is also provable analogously to the simply typed λ-calculus (for example, see [44]).

For the full calculus, the decidability property still holds, but its proof is harder, since we need the normalisation property for all constructed terms (since arbitrary proofs can appear in the types, see [15]).

4 Stripping

We shall now show how to extract from a given proof (i.e. a given functional) its associated pure (non-typed) λ-term which represents in some way its computational contents. All this is a generalisation of the realisability concept [30], but we use λ-terms instead of Gödel's codes for recursive functions. This can be done for the full calculus as well as for the restricted calculus.

First we develop the syntactic theory of ordinary λ-calculus in a way which is consistent with our notations.

4.1 Untyped λ-calculus

We define the set λ^n of λ-expressions generated by n free variables by the following inductive rules:

$$k \in \lambda^n \text{ if } 1 \leq k \leq n \qquad \textit{variables}$$

$$\lambda x \cdot N \in \lambda^n \text{ if } N \in \lambda^{n+1} \qquad \textit{abstraction}$$

$$(M\ N) \in \lambda^n \text{ if } M, N \in \lambda^n \qquad \textit{application}$$

As before, the name x associated with the abstraction operation is a pure dummy which is not part of the abstract structure.

4.2 The context contraction map

Let $\Gamma \vdash *$ be a well-formed context. We shall distinguish in Γ the quantifications over contexts from the quantifications over objects, since only the latter will be considered free variables of stripped formulas. The quantifications over contexts are used solely at compile-time, for polymorphism type-checking.

Definition. The number of parameters, or *arity*, α_Γ and the canonical injection $j_\Gamma : \alpha_\Gamma \to |\Gamma|$ of a context Γ are determined by the following inductive rules (confusing n with $\{1, ..., n\}$):

$$\alpha_* = 0 \quad j_* = Id_0.$$

If $\Gamma = \Delta[x : M]$, then if M is a context, we take

$$\alpha_\Gamma = \alpha_\Delta, \quad j_\Gamma(k) = j_\Delta(k) + 1,$$

and if M is an object, we take

$$\alpha_\Gamma = \alpha_\Delta + 1, \quad j_\Gamma(1) = 1, \quad j_\Gamma(k+1) = j_\Delta(k) + 1.$$

4.3 Untyping

Definition. if $\Gamma \vdash M : N$, and N is an object, we define the *stripped* algorithm $\nu_\Gamma(M) \in \lambda^{\alpha_\Gamma}$ by induction on M:

1. If $M = k$, we take $\nu_\Gamma(M) = j_\Gamma^{-1}(k)$.
2. If $M = (M_1\ M_2)$, we know that $\Gamma \vdash M_1 : P_1$ and $\Gamma \vdash M_2 : P_2$. If P_2 is an object, we take $\nu_\Gamma(M) = (\nu_\Gamma(M_1)\ \nu_\Gamma(M_2))$, and if P_2 is a context, we take $\nu_\Gamma(M) = \nu_\Gamma(M_1)$ (we simply forget all type information, which is now viewed as a comment in the algorithm).

3. If $M = (\lambda x : P)N$, we know that $\Delta \vdash N : Q$, with $\Delta = \Gamma[x : P]$, and Q an object. Now if P is an object, we take $\nu_\Gamma(M) = \lambda x.\nu_\Delta(N)$, and if P is a context we take $\nu_\Gamma(M) = \nu_\Delta(N)$.

We shall usually write $\nu(M)$ instead of $\nu_\Gamma(M)$ when the context Γ is clear.

This λ-term $\nu(M)$ may be thought of as the computational contents of the proof M. The intuitive meaning of the previous translation rules is then that the propositions are comments of programs, and that those programs behave in a uniform way with respect to these comments.

5 An interpretation of constructions

We first need some notations: let I be the set of all closed λ-terms, built on a special constant named Ω.

Definition. We say that a subset A of I is *saturated* if, and only if,

1. $\Omega \in A$,
2. if $b_1, ..., b_n$ are strongly normalisable, then $(\Omega\ b_1 ... b_n) \in A$,
3. $a \in A$ implies a strongly normalisable,
4. if b is strongly normalisable, then

$$([b/x]a\ b_1 ...\ b_n) \in A \Rightarrow (\lambda x.a\ b\ b_1 ...\ b_n) \in A.$$

Now, let $\mathcal{U}$ be the set of all saturated subsets of I.

Definition. If $A \in \mathcal{U}$ and $F \in I \to \mathcal{U}$, then the *dependent product* $\Pi(A, F)$ of A and F is the set $\{t \in I \mid \forall x \in A\ (t\ x) \in F(x)\}$.

Intuitively, the elements of I are the programs and the elements of $\mathcal{U}$ the types. In the previous definition, F is a dependent type.

We may then check that $\mathcal{U}$ has the following closure properties:

Lemma 7. *$\mathcal{U}$ is closed under intersection of non empty families and under dependent product.*

The introduction of the special constant Ω is needed in the proof of these properties.

What follows is a realisability interpretation [30], which is very close to the one defined by Tait [45].

5.1 The functionality of a term

Definition. We define the *functionality* $\varphi(M)$ of a term M as follows. If M is an object, we take $\varphi(M) = I$. For contexts, we take $\varphi(*) = \mathcal{U}$, and $\varphi([x : P]\Gamma) = \varphi(P) \to \varphi(\Gamma)$, the set of all functions from $\varphi(P)$ to $\varphi(\Gamma)$.

This definition holds for the restricted calculus. In the full calculus, we would define $\varphi([x : P]\Gamma)$ as $\varphi(P) \to \varphi(\Gamma)$, if P is a context, and if P is an object, as the set of all functions f from I to $\varphi(\Gamma)$ such that $f(t) = f(u)$ if t and u are β-convertible.

The following lemma is true for both the restricted and the full calculus:

Lemma 8. If $\Gamma \vdash M \cong N$ then $\varphi(M) = \varphi(N)$.

Definition. If $\Gamma \vdash *$ is any valid context, $\Gamma = [x_n : A_n] \dots [x_1 : A_1]$, then the *environment* associated to Γ is the product $\vec{\varphi}(\Gamma) = \varphi(A_n) \times \cdots \times \varphi(A_1)$.

5.2 Interpretation of objects

Let $\Gamma \vdash M : N$ be a derived sequent. We shall interpret it as a function $\rho_\Gamma(M) : \vec{\varphi}(\Gamma) \to \varphi(N)$.

There are two cases, according to whether N is a context or not.

When N is not a context, let us consider the pure λ-term $\nu_\Gamma(M)$. It has α_Γ free variables, and may thus be interpreted as a function $\overline{\nu_\Gamma(M)} : I^{\alpha_\Gamma} \to I$, which simply substitutes its actual arguments to the corresponding free variables. Furthermore, to the previously defined type forgetting operation j_Γ corresponds the projection $\pi_\Gamma : \vec{\varphi}(\Gamma) \to I^{\alpha_\Gamma}$. We then define $\rho_\Gamma(M)$ as $\overline{\nu_\Gamma(M)} \circ \pi_\Gamma$.

When N is a context, we define $\rho_\Gamma(M)$ by induction on the derivation of the sequent $\Gamma \vdash M : N$ as follows.

- **product formation:** $\Gamma \vdash [x : M_1]M_2 : *$ results from $\Gamma[x : M_1] \vdash M_2 : *$. Let $\Delta = \Gamma[x : M_1]$. We have two subcases according to whether M_1 is an object or not:
  subcase 1: M_1 is an object (and we have $\Gamma \vdash M_1 : *$), then by induction we can compute $f = \rho_\Gamma(M_1)$ and $g = \rho_\Delta(M_2)$. We then have $f : \vec{\varphi}(\Gamma) \to \mathcal{U}$ and $g : \vec{\varphi}(\Gamma) \times I \to \mathcal{U}$ and we define $\rho_\Gamma([x : M_1]M_2)$ as the function from $\vec{\varphi}(\Gamma)$ mapping a to $\Pi(f(a), g(a))$.
  subcase 2: M_1 is a context, then by induction we can compute $f = \rho_\Delta(M_2)$, so that $f : \vec{\varphi}(\Gamma) \times \varphi(M_1) \to \mathcal{U}$. We then define $\rho_\Gamma([x : M_1]M_2)$ as the function from $\vec{\varphi}(\Gamma)$ mapping a to $\cap\{f(a, x) \mid x \in \varphi(M_1)\}$.

- **variable:** we have $\Gamma \vdash l : \Gamma/l$, with $l \leq |\Gamma|$. Then, $\rho_\Gamma(M)$ is simply the projection mapping $(x_n, ..., x_1)$ to x_l.

- **abstraction:** $\Gamma \vdash (\lambda x : M_1)M_2 : [x : M_1]P$ results from $\Gamma[x : M_1] \vdash M_2 : P$ by abstraction. By induction, we can compute $f = \rho_\Delta(M_2)$ (where $\Delta = \Gamma[x : M_1]$), which is a function from $\vec{\varphi}(\Gamma) \times \varphi(M_1)$ to $\varphi(P)$. We then define $\rho_\Gamma((\lambda x : M_1)M_2)$ as the application from $\vec{\varphi}(\Gamma)$ to $\varphi(M_1) \to \varphi(P)$ mapping a to the function mapping x to $f(a, x)$.

- **application:** $\Gamma \vdash (M\ N) : [N/x]Q$ results from $\Gamma \vdash M : [x : P]Q$ and $\Gamma \vdash N : P$. By induction, we have defined $\rho_\Gamma(M) : \vec{\varphi}(\Gamma) \to (\varphi(P) \to \varphi(Q))$ and $\rho_\Gamma(N) : \vec{\varphi}(\Gamma) \to \varphi(P)$. We then define $\rho_\Gamma((M\ N))$ as the application from $\vec{\varphi}(\Gamma)$ to $\varphi(Q)$ mapping x to $\rho_\Gamma(M)(x, \rho_\Gamma(N, x))$.

We do not take the conversion rules into account, and this is justified by lemma 8.

Example. The sequent $\vdash [A : *][x : A]A : *$ is interpreted as $\{t \in I \mid \forall A \in \mathcal{U}\ \forall x \in A\ (t\ x) \in A\}$ and the sequent $\vdash \lambda A.\lambda x.x : [A : *][x : A]A$ is interpreted as the untyped λ-term $\lambda x.x$.

Lemma 9. Let M and N be objects such that $\Gamma \vdash M \cong N$. We have $\rho_\Gamma(M) = \rho_\Gamma(N)$.

This lemma holds for the restricted calculus. Similarly, in the full calculus, if $\Gamma \vdash M \cong N$, either M and N are both proofs, in which case $\rho_\Gamma(M)$ and $\rho_\Gamma(N)$ are two β-convertible λ-terms, or else M and N are both propositions (or proposition schemas), in which case $\rho_\Gamma(M) = \rho_\Gamma(N)$.

5.3 Interpretation of contexts

To each context $\Gamma \vdash *$, we shall associate an inclusion $D(\Gamma) \hookrightarrow \vec{\varphi}(\Gamma)$ by induction on the formation of $\Gamma \vdash *$:

- **case 1** : $\vdash *$, we take $D(\Gamma) = \vec{\varphi}(\Gamma) = 1$.
- **case 2** : $\Gamma[x : M] \vdash *$, we have $\Gamma \vdash M : *$, and so by the previous part, we have already defined $\rho_\Gamma(M)$. By induction, we have already an inclusion $D(\Gamma) \hookrightarrow \vec{\varphi}(\Gamma)$. We take $D(\Gamma[x : M]) = \{(a, x) \mid a \in D(\Gamma) \ \wedge \ x \in \rho_\Gamma(M)(a)\}$.
- **case 3** : $\Gamma[x : \Delta] \vdash *$, we have by induction an inclusion $D(\Gamma) \hookrightarrow \vec{\varphi}(\Gamma)$ and we take $D(\Gamma[x : \Delta]) = \{(a, x) \mid a \in D(\Gamma) \ \wedge \ x \in \varphi(\Delta)\} = D(\Gamma) \times \varphi(\Delta)$.

Example : $[A : *][x : A] \vdash *$ is interpreted as $\{(A, x) \in \mathcal{U} \times I \mid x \in A\}$.

5.4 Consistency

We can now state the principal theorem, whose proof is a straightforward (but somewhat tedious) structural induction and which holds in the restricted and in the full calculus.

Theorem : If $\Gamma \vdash M : P$, and $\Gamma \vdash P : *$ then for all x in $D(\Gamma)$, the pure λ-term $\rho_\Gamma(M, x)$ is an element of the saturated set $\rho_\Gamma(P, x)$.

Example : We have $[A : *][x : A] \vdash x : A$, then, with $\Gamma = [A : *][x : A]$, we have $D(\Gamma) = \{(A, x) \in \mathcal{U} \times I \mid x \in A\}$ and $[A : *][x : A] \vdash x : A$ is interpreted as $f : \mathcal{U} \times I \to I$ mapping (A, x) to x. Similarly $[A : *][x : A] \vdash A : *$ is interpreted as $g : \mathcal{U} \times I \to \mathcal{U}$ mapping (A, x) to A. We see that $f(A, x) \in g(A, x)$ if $(A, x) \in D(\Gamma)$.

Corollary 1. If $\vdash M : N$ and N is an object, then $\rho(M)$ is a strongly normalisable pure λ-term (where ρ is an abbreviation for ρ_*).

It is sufficient to note that $\rho(M)$ is an element of $\rho(N)$, by the previous theorem, and $\rho(N)$ belongs to $\mathcal{U}$ by construction. By the definition of $\mathcal{U}$, we see that $\rho(M)$ is strongly normalisable.

Definition. A proposition $\vdash P : *$ is *inhabited* if, and only if there is a term M such that $\vdash M : P$.

Corollary 2. The Calculus of Constructions is consistent, in the sense that there exists a proposition which is not inhabited.

The intuitive meaning of this statement is that the calculus does not prove all its well-formed propositions. Indeed, consider the term $N = [A : *]A$. We have $\vdash N : *$, and the special constant Ω appears in all the terms of $\rho(N)$, which is the set consisting of all strongly normalizable terms normalizing to Ω or to a term of the form $(\Omega\ b_1\ \ldots\ b_n)$. But if $\vdash M : N$ then Ω does not appear in the term $\rho(M)$, hence the corollary.

The realisability interpretation we have presented is syntactic in nature. However, it is consistent with the set-theoretical intuition of interpreting $M : P$ as $M \in P$. Still, the functional spaces $M \to N$ are not interpreted as the full function space, but only as sets of definable algorithms, closed under the operations corresponding to the syntactic operators. We know from Reynolds' work that a complete set-theoretic semantics cannot exist [41].

Other interpretations of the calculus are possible. For instance, the Boolean interpretation, where each proposition is mapped to 0 or $1 = \{0\}$, and the proofs are mapped to 0, is simpler and

suffices for proving the consistency. In some sense, this is the "proof-irrelevance" interpretation of classical logic.

It is also possible to interpret the calculus in domains such as $P\omega$, where each object (proposition or proof) is mapped to an element of $P\omega$, in such a way that propositions become closures [37]. However, such models also provide an interpretation for logically inconsistent systems (with *Type*: *Type*) [7,2]. Thus, such interpretations fail to capture the essential feature of the calculus.

5.5 Extracting programs from proofs

Every proof construction $\Delta \vdash M : P$ corresponds to an algorithm $\nu_\Delta(M)$. Intuitively, this algorithm obeys proposition P considered as its specification, under the hypothesis on its α_Δ inputs described by Δ. This algorithm, a pure λ-expression in λ^{α_Δ}, always terminates *for well-typed values of its inputs*. This is the main limitation of our calculus as far as its programming language character goes. However, almost all partial recursive functions are definable in the calculus. For instance, all total recursive functions which are provably total in higher order arithmetic are definable, as shown in Girard [23]. They correspond to the stripped proofs of the proposition $nat \to nat$, for the appropriate type $nat = \forall A(A \to A) \to (A \to A)$.

As another example, we may consider the partial recursive function defined as:

$$f(n) = \mathit{if}\ n = 0\ \mathit{or}\ 1\ \mathit{then}\ 0\ \mathit{else}\ \mathit{if}\ \mathit{even}(n)\ \mathit{then}\ f(n/2)\ \mathit{else}\ f(3\ n+1).$$

This function is easily definable in our calculus, as a proof of $[n : nat](D\ n) \to nat$, with the domain D defined as the proper smallest predicate preserving termination of f, that is $(D\ n)$ is:

$$[P : nat \to *](P\ 0) \to (P\ 1) \to ([u : nat](P\ u) \to (P\ 2u)) \to ([u : nat](P\ 3u{+}2) \to (P\ 2u{+}1)) \to (P\ n).$$

Note that here nothing tells us that f is total on non-negative integers. If some day a proof of that fact is known, we shall get f as an algorithm in $nat \to nat$ by feeding it this proof as the $(D\ n)$ argument. This example is especially simple, since the domain argument is redundant for the computation. For more complicated examples, the domain argument may be needed, since its proof may describe the recursion structure.

Of course the above discussion on recursion extends to inductive definitions on any data type. Note that non-predicativity is needed here for the definition of such inductive predicates. By contrast, Constable and Mendler [14] must extend the basic PRL system with recursive types.

We may thus consider our calculus as a general formalism in which to develop programs consistently with their specifications. Our logic is strong enough to articulate arbitrarily complex algorithmic specifications, as well as the more mundane standard data-types found in usual programming languages [18].

6 Variations on the basic calculus

6.1 A system with normal types

It is important to clearly distinguish between the presentation of the construction calculus for a metamathematical study and its presentation for an implementation and the development of proofs and programs in this calculus. The presentation we have chosen here is the best suited for the proofs of the mathematical properties of the calculus of constructions. But once we have these properties, it is possible to derive other presentations of the system. For example, since we have the

normalisation property [15], it is possible to present the full calculus in the following way, where $\mathcal{N}(M)$ denotes the normal form of the term M:

$$\frac{}{\vdash *}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma[x : \mathcal{N}(\Delta)] \vdash *}$$

$$\frac{\Gamma \vdash M : *}{\Gamma[x : \mathcal{N}(M)] \vdash *}$$

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

$$\frac{\Gamma[x : M_1] \vdash M_2 : *}{\Gamma \vdash [x : M_1].M_2 : *}$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash l : \Gamma/l} \quad (l \leq |\Gamma|) \qquad \textit{variable}$$

$$\frac{\Gamma[x : M_1] \vdash M_2 : P}{\Gamma \vdash (\lambda x : M_1)M_2 : [x : M_1]P} \qquad \textit{abstraction}$$

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

This presentation, by putting systematically types in normal form, avoids the conversion rules and thus seems a bit simpler (and it is the one used in [19]). But this system does not seem to be well suited to a metamathematical study.

6.2 Confusing abstraction with product *à la Automath*

The calculus has three levels and two binding operators. At the level of contexts, only the product binding is allowed. At the level of proofs, the only binding is the λ-abstraction. The two bindings may appear together only at the level of propositions, but in a special order: a sequence of abstractions followed by a sequence of quantifications. Thus we could confuse the two bindings, replacing

$$(\lambda x_1 : P_1) \cdots (\lambda x_k : P_k)[x_{k+1} : P_{k+1}] \cdots [x_n : P_n]P$$

by

$$[x_1 : P_1] \cdots [x_k : P_k] \bullet [x_{k+1} : P_{k+1}] \cdots [x_n : P_n]P,$$

where $\bullet$ separates the abstractions from the quantifications. Finally, it may be useful to confuse the propositional schema $[x : A] \bullet P$ with its universal closure $\bullet[x : A]P$ by using an ambiguous notation without $\bullet$. This has some notational advantages, and the implementation described in [18] used this facility. This allows in particular the denotation by one term of several concepts: a propositional schema with free variables and its universal closures. This can be seen as a facility for overloading the meaning of the types in the calculus. We refer to the Automath literature for this question, especially [21], where such facility is called "type-inclusion".

7 Towards a reasonable user interface

7.1 Introducing constants

The first step toward providing a usable system consists in defining combinators which abbreviate definitions. These constants are given, in a context Γ, with a definition which is an object term M, and a unique name. We check before entering the constant in the theory that $\Gamma \vdash M : \Delta$ (propositional constant) or $\Gamma \vdash M : N$ (proof constant). Later on the type checker retrieves the type of each constant by looking it up in the theory tables. This permits the saving of space (by sharing commonly used constructions) and time (by not re-checking similar constructions). These constant definitions can be internalized in the language by the "let" construct, where $let\ x = M_1\ in\ M_2$ abbreviates $((\lambda x : P)M_2\ M_1)$ (where P is the type of M_1). We can thus get "local" constants at any context depth.

No extension of the theory is required to explain the calculus with constants. The only problem is to implement an absolute naming scheme, orthogonal to de Bruijn's indexes considered so far, while preserving a notion of static scope. This problem is the logical analogue of the problem of linking separately compiled modules in a programming language. We do not comment further on this issue, but we remark that from a practical point of view this facility is crucial, since it would be impossible to effectively realize any significant proof without constants. Adding constants is here the analogue of going from single-line Automath to full Automath books.

7.2 Synthesis of implicit arguments

The next step in providing the user with a realistic system in which to develop proofs is to reduce his burden of polymorphic instantiation. Many propositional arguments are redundant, since they may be inferred automatically as sub-components of types of further arguments. Thus a certain amount of type synthesis is possible without any non-deterministic search. Let us give a trivial example. In the following discussion, we shall confuse abstractions with products.

If one wants to define composition (i.e. the cut rule of propositional logic) in the basic calculus, we have to define the constant:

$$Comp \leftarrow [A : *][B : *][C : *][f : A \to B][g : B \to C][x : A](g\ (f\ x)).$$

This is very cumbersome, and if one assumes that *Comp* is always used with all arguments up to g there is a lot of redundancy, since the actual arguments corresponding to A,B and C are necessary parts of the types of the actual arguments corresponding to f and g. The crucial observation is that certain parts of the terms will always have residuals in every reduction of every substitution instance of the term. This determines in the normal forms of types rigid skeletons in which one may access sub-components by pattern-matching. For instance, in $A \to B$, i.e. $[u : A]B$, we can use the whole term as a pattern in the free variables A and B. This method relies on the variant explained above of keeping types in normal form.

The notion of rigid skeleton was defined in [27] in the context of a unification algorithm for typed λ-calculus. Let us recall this notion. Let

$$M = [u_1 : P_1] \cdots [u_n : P_n](x\ N_1\ \cdots\ N_p)$$

be a term in normal form (in this discussion, λ-abstractions are treated in the same way as products). Let V be a set of variables. We call *rigid occurrence* of M relatively to V any member of the following set of positions in M. First, we take the rigid occurrences in P_i relative to $V \cup \{u_1, ..., u_{i-1}\}$,

for $i = 1, ..., n$. Then, if $p = 0$, the occurrence of the head variable x, and if $p > 0$, and when $x \in W = V \cup \{u_1, ..., u_n\}$, the rigid occurrences in N_j relative to W, for $j = 1, ..., p$. Now let z be any variable. We say that M *determines* z iff z appears in M at a rigid occurrence relative to $\emptyset$.

We are now able to explain how to declare combinators of the calculus given with an arity of *explicit* arguments, whose types determine automatically *implicit* arguments which will be automatically synthesized. In our example above, we would write:

$$Comp\{A : *\}\{B : *\}\{C : *\}[f : A \to \Gamma]\,[g : \Gamma \to C] \leftarrow [x : A](g\ (f\ x)),$$

where the curly brackets indicate the implicit arguments. Now the combinator *Comp* may be invoked with only its explicit actual arguments, like in $Comp(F, G)$. In the general situation, a declaration of a combinator with arguments $u_i : P_i$ will be legal iff for every implicit i there exists an argument $j > i$ such that P_j determines u_i. It is not mandatory that j be itself explicit, since the synthesis of implicit arguments may be iterated (from right to left).

Remark. It is possible to generalize this method, by computing recursively whether some functional argument determines some of its parameters. For instance, consider:

$$C \leftarrow [P : A \to *]\,[x : A]\,[h : (P\ x)] \cdots$$

The occurrence of x in $(P\ x)$ is *not* rigid. However, if the actual first argument P_0 of a given application $(C\ P_0\ x_0\ h_0)$ is of the form $[u : A]\,M$ such that M has a rigid occurrence of u, then x_0 may be synthetized from the type of h_0; i.e. rigidity may be inherited. However, it is not yet clear how to specify such iterated synthesis in a clearly understandable way, since the notion of implicit argument is not bound to the definition of combinator C anymore, but rather varies dynamically with every use of C. A possibly useful restriction would be to impose in the definition of C that certain arguments ought to determine certain of their own parameters, using a syntax such as:

$$C\,[P : \{u : A\} *]\{x : A\}[h : (P\ x)] \leftarrow \cdots$$

This is in a way a natural extension of restrictions of λ-calculus expressibility at the proposition level, such as Church's use of λI-calculus.

Note that the synthesis of implicit arguments corresponds exactly to the mathematical practice. For instance, in category theory, one writes Id_A, but $f \circ g$ is not annotated with objects, since the arrows f and g determine the proper composition from their domains and co-domains.

Finally, we stress that a certain sophistication in concrete syntax, i.e. in the way new notations may be associated to concepts by the user in the course of the development of a theory, is crucial if one wants to mechanize mathematical concepts beyond the attempts of Frege, the Principia and even Automath. Hopefully modern computer technology will help, and dynamically extendable parsers and complex window managers seem to be necessary components of user interfaces to programming and proving environments [13]. Let us just mention one proposal [19] for concrete syntax definition of combinators given with arities, which fits nicely with the above algorithm for synthesis of implicit arguments.

7.3 Concrete syntax

Since we now accept combinators with arities, we might as well endow them with concrete syntax. A straightforward device for declaring arbitrary mixfix notation is to allow the declaration of combinators by *patterns*:

$$pattern \leftarrow term,$$

where *pattern* is an arbitrary sequence of concrete strings, implicit argument declarations $\{x : M\}$, and explicit argument declarations $[x : M]$. Standard methods such as precedence declarations may complement this basic mechanism to resolve ambiguities. For instance, we would now allow the declaration:

$$\{A : *\}\{B : *\}\{C : *\}[f : A \to \Gamma] \circ [g : \Gamma \to C] \leftarrow [x : A](g\ (f\ x)),$$

and be able to write in the usual manner $F \circ G$. Examples of development of mathematical notions along those lines are presented in [19].

More ambitiously, we may imagine incorporating progressively theorem-proving capabilities to what is initially an interactive proof-checker. We may synthesize whole constructions by systematic search of possible combinations of given sets of combinators. Such tacticals may be programmed in the meta language of the system, in the tradition of LCF [25] or Pearl [13]. This will offer a powerful help to the mathematician, who will be able to concentrate on the global proof strategy, i.e. on the proper ordering of lemmas, without losing time over the combinatory headaches of the technical proofs.

8 Possible extensions

The first extension is to add operators with special rules of conversion and reduction. For instance, we can add pairing, disjoint sums, integers, booleans as primitives. As an example, let us add the special constants $int : *$, with $0 : int$, $S : int \to int$ and

$$rec : [P : int \to *](P\ 0) \to ([u : int](P\ u) \to (P\ (S\ u))) \to [n : int](P\ n),$$

with the conversion rules:

$$\frac{\Gamma \vdash P : int \to *\quad \Gamma \vdash a : (P\ 0)\quad \Gamma \vdash f : [u : int](P\ u) \to (P\ (S\ u))}{\Gamma \vdash (rec\ P\ a\ f\ 0)\ \cong\ a}$$

$$\frac{\Gamma \vdash P : int \to *\quad \Gamma \vdash a : (P\ 0)\quad \Gamma \vdash f : [u : int](P\ u) \to (P\ (S\ u))\quad \Gamma \vdash n : int}{\Gamma \vdash (rec\ P\ a\ f\ (S\ n))\ \cong\ (f\ n\ (rec\ P\ a\ f\ n))}.$$

The normalisation proof of [24] still extends to this calculus. It is even possible to add a fixpoint operator (only to the restricted calculus if one still wants the normalisation property for the type-checking). This calculus appears then as a direct generalisation of high-level functional languages, such as Ponder [22].

Another possible extension is suggested by the connection with the calculus of Martin-Löf. We have seen that our $*$ can be thought of as the first universe U_0 of Martin-Löf, but with the property that $[A : U_0]A$, for example, is still of type U_0. It is then natural to try to extend the calculus of constructions with a universe hierarchy U_1,... such that $*$ is of type U_1, which appears to be the type of all contexts, U_1 is of type U_2,... This is possible, but with great care if one wants to preserve the normalisation property. In particular, it results from [24], that the normalisation property is lost for the natural attempt of adding the rules:

$$\frac{\Gamma \vdash *}{\Gamma[x : U_1] \vdash *}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta : U_1}$$

$$\frac{\Gamma[x : M_1] \vdash M_2 : U_1}{\Gamma \vdash [x : M_1]M_2 : U_1}$$

Another possibility is to add the rule $\Gamma \vdash * : *$, for every valid context Γ. Girard [24] shows that this calculus does not have the normalisation property (see also [16]). However, it may be useful to consider this calculus as a type system for programming languages (see [7]), but the Curry-Howard paradigm seems to be lost forever then, since all propositions become provable.

A satisfactory rule for extended product is obtained by replacing the third rule above by:

$$\frac{\Gamma \vdash M_1 : U_1 \quad \Gamma[x : M_1] \vdash M_2 : U_1}{\Gamma \vdash [x : M_1]M_2 : U_1,}$$

and similarly for higher universes. We then add the corresponding conversion rules (see [16] for a complete presentation).

Let us say that an object is *predicative* if it is defined by a quantification over a type which does not contain this object. In this sense, the calculus of construction allows the formation of non predicative notions. For instance, the polymorphic identity is not predicative since it can be instantiated over its own type.

There is a tension between a purely logical language based on the Curry-Howard correspondance, and the power of expression of set theory. It is legitimate to use impredicative quantification inside the logical language, but if we want to complement it with a set-theoretic hierarchy, this latter part must be strictly stratified.

Conclusion

We have proposed a Calculus of Constructions and shown how to use it to derive pure strongly normalisable λ-terms. This calculus blends together earlier proposals of de Bruijn [6], Martin-Löf [32], and Girard [24]. Its syntax is closest to the Automath languages. In some sense, this calculus is the "universal functional system", in the spirit of "Curry's program" [43]. A prototype implementation of the calculus has been implemented at Inria. Numerous examples of mathematical proofs expressed in the Calculus of Constructions, and machine-checked on our implementation, are given in [19]. We hope that this calculus will be useful for future developments of programming environments, where programs will be developed consistently with logical propositions expressing in one unified formalism data types, correctness assertions and inter-modules specifications.

Acknowledgements

We thank J.Y. Girard for his help and guidance for earlier versions of the formalism. We thank D. Scott for his invitation at Carnegie-Mellon University where the final version of this paper was written.

References

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

[2] H. Barendregt and A. Rezus. "Semantics for Classical AUTOMATH and Related Systems." Information and Control **59** (1983) 127–147.

[3] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29-61.

[4] 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.

[5] N.G. de Bruijn. "Some extensions of Automath: the AUT-4 family." Internal Automath memo M10 (Jan. 1974).

[6] 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).

[7] L. Cardelli. "A Polymorphic λ-calculus with Type:Type." Private communication (1986).

[8] L. Cardelli and P. Wegner. "On Understanding Types, Data abstraction, and Polymorphism." Private communication (May 1985).

[9] A. Church. "A set of postulates for the foundation of logic." Annals of Mathematics (1932).

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

[11] A. Church. "The Calculi of Lambda-Conversion." Princeton U. Press, Princeton N.J. (1941).

[12] R.L. Constable et al. "Implementing Mathematics in the NuPrl System." Prentice-Hall (1986).

[13] R.L. Constable and J.L. Bates. "The Nearly Ultimate PRL." Tech. report TR 83-551, Dept. of Computer Science, Cornell University. (Dec. 1983).

[14] R.L. Constable and N.P. Mendler. "Recursive Definitions in Type Theory." In Proc. Logic of Programs, Springer-Verlag Lecture Notes in Computer Science **193** (1985).

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

[16] Th. Coquand. "An Analysis of Girard's Paradox." Logic in Computer Science Conference, Boston (June 1986).

[17] Th. Coquand, G. Huet. "A Theory of Constructions." Preliminary version, presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis (June 84).

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

[19] Th. Coquand, G. Huet. "Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions." Colloque de Logique, Orsay (July. 1985). To appear, North-Holland.

[20] H. B. Curry, R. Feys. "Combinatory Logic Vol. I." North-Holland, Amsterdam (1958).

[21] D. Van Daalen. "The language theory of Automath." Ph. D. Dissertation, Technological Univ. Eindhoven (1980).

[22] J. Fairbairn. "Design and Implementation of a Simple Typed Language Based on the Lambda-calculus." University of Cambridge Computer Lab. Tech. Report 75 (May 1985).

[23] 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. Proceedings of the Second Scandinavian Logic Symposium, Ed. J.E. Fenstad, North Holland (1970) 63–92.

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

[25] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF" Springer-Verlag LNCS **78** (1979).

[26] 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).

[27] G. Huet. "A Unification Algorithm for Typed Lambda Calculus." Theoretical Computer Science, **1.1** (1975) 27–57.

[28] L.S. Jutting. "A translation of Landau's "Grundlagen" in AUTOMATH." Eindhoven University of Technology, Dept of Mathematics (Oct. 1976).

[29] L.S. van Benthem Jutting. "The language theory of Λ_{∞}, a typed λ-calculus where terms are types." Unpublished manuscript (1984).

[30] S.C. Kleene. "Introduction to Meta-mathematics." North Holland (1952).

[31] J. Lambek and P. J. Scott. "Aspects of Higher Order Categorical Logic." Contemporary Mathematics **30** (1984) 145–174.

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

[33] P. Martin-Löf. "About models for intuitionistic type theories and the notion of definitional equality." Paper read at the Orléans Logic Conference (1972).

[34] P. Martin-Löf. "An intuitionistic Theory of Types: predicative part." Logic Colloquium 73, Eds. H. Rose and J. Shepherdson, North-Holland, (1974) 73–118.

[35] P. Martin-Löf. "Constructive Mathematics and Computer Programming." In Logic, Methodology and Philosophy of Science **6** (1980) 153–175, North-Holland.

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

[37] N. McCracken. "An investigation of a programming language with a polymorphic type structure." Ph.D. Dissertation, Syracuse University (1979).

[38] R.P. Nederpelt. "Strong normalization in a typed λ calculus with λ structured types." Ph. D. Thesis, Eindhoven University of Technology (1973).

[39] R.P. Nederpelt. "An approach to theorem proving on the basis of a typed λ-calculus." 5th Conference on Automated Deduction, Les Arcs, France. Springer-Verlag LNCS **87** (1980).

[40] J. C. Reynolds. "Towards a Theory of Type Structure." Programming Symposium, Paris. Springer Verlag LNCS **19** (1974) 408–425.

[41] J. C. Reynolds. "Polymorphism is not set-theoretic." International Symposium on Semantics of Data Types, Sophia-Antipolis (June 1984).

[42] D. Scott. "Constructive validity." Symposium on Automatic Demonstration, Springer-Verlag Lecture Notes in Mathematics, **125** (1970).

[43] J.P. Seldin. "Progress report on generalized functionality." Ann. Math. Logic. **17** (1979).

[44] S. Stenlund. "Combinators λ-terms, and proof theory." Reidel (1972).

[45] W. Tait. "A Realizability Interpretation of the Theory of Species." Logic Colloquium, Ed. R. Parikh, Springer Verlag Lecture Notes **453** (1975).

[46] G. Takeuti. "On a generalized logic calculus." Japan J. Math. **23** (1953).

[47] G. Takeuti. "Proof theory." Studies in Logic **81** Amsterdam (1975).

[48] A. N. Whitehead, B. Russell. "Principia Mathematica" Cambridge University Press (1911).

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique