A Complete Proof Synthesis Method for the Cube of Type Systems

Gilles Dowek

A Complete Proof Synthesis Method for the Cube of Type Systems

Gilles Dowek

INRIA * †

Abstract

We present a complete proof synthesis method for the eight type systems of Barendregt's cube extended with η-conversion. Because these systems verify the proofs-as-objects paradigm, the proof synthesis method is a one level process merging unification and resolution. Then we present a variant of this method, which is incomplete but much more efficient. At last we show how to turn this algorithm into a unification algorithm.

Introduction

The Calculus of Constructions is an extension of Church higher order logic in which the terms belong to a λ-calculus with dependent types, polymorphism and type constructors. Because of the richness of the terms structure, proofs can be represented as terms through Heyting semantics and Curry-Howard isomorphism. So a proof of a proposition P is merely a term of type P.

A proof synthesis method for Church higher order logic is given in [20]. In this paper we generalize this method to find terms of a given type in the systems of Barendregt's cube (in particular in the Calculus of Constructions) extended with η-conversion. In some sense, because these type systems are more powerful than Church higher order logic, the proof synthesis problem is more complicated. But we claim that because the proofs-as-objects paradigm simplifies the formalisms, it also simplifies the proof synthesis algorithms. In particular resolution and unification can be merged in a uniform algorithm.

We first present a complete method, then we discuss some efficiency improvements, we present a variant of this method which is incomplete but much more efficient and we show how to turn this algorithm into a unification algorithm.

1 Outline of the Method

1.1 Resolution and Unification

In first order logic or higher order logic, we have two syntactical categories, terms and proofs. Terms are trees (in first order logic) or simply typed λ-terms (in higher order logic) and proof are build, for instance, with natural deduction rules. These rules combine some proofs and terms to form new proofs. For instance the rule $\rightarrow$-elim

$$\frac{A \rightarrow B \quad A}{B}$$

*B.P. 105, 78153 Le Chesnay CEDEX, France. Gilles.Dowek@inria.fr

†This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

combines two proofs (one of $A \to B$ and one of A) to get a proof of B. The rule $\forall$-elim

$$\frac{\forall x : A.B \quad t : A}{B[x \leftarrow t]}$$

combines a proof (of $\forall x : A.B$) and a term (of type A) to get a proof of $B[x \leftarrow t]$. So proofs are heterogeneous trees. For instance, in the proof

$$\frac{\forall f : T \to T.((P\ f) \to (Q\ f)) \quad \dfrac{[x : T]}{[x : T]x : T \to T}}{\dfrac{(P\ [x : T]x) \to (Q\ [x : T]x)}{(Q\ [x : T]x)} \quad (P\ [x : T]x)}$$

the subtree

$$\frac{[x : T]}{[x : T]x : T \to T}$$

is a term derivation.

In first order logic and higher order logic, the *resolution* algorithm searches for proof-trees. When during the search of a proof-tree, a term-tree is needed, the *unification* algorithm is called. In type systems, term-trees and proof-trees belong to a single syntactical category, so our algorithm is a one level process merging resolution and unification.

We shortly present now the unification and resolution algorithms for higher order logic [20] [21] [22], our goal is to show that a single idea on term enumeration underlies both algorithms.

1.2 Higher Order Unification

Higher order unification is based on an algorithm that enumerates all the normal η-long terms of a given type in the simply typed λ-calculus.

Let $P_1 \to ... \to P_n \to P$ (P atomic) be a type. All normal η-long terms of this type begin by n abstractions

$$t = [x_1 : P_1]...[x_n : P_n]t'$$

The term t' must have type P so it is an atomic term. A variable $w : Q_1 \to ... \to Q_p \to Q$ of the context or which is an x_i can be the head of this term, only if $Q = P$. Then this variable must be applied p times in order to get a term of the good type. So

$$t = [x_1 : P_1]...[x_n : P_n](w\ c_1\ ...\ c_p)$$

To mark the dependency of the c_i on the x_j we write

$$t = [x_1 : P_1]...[x_n : P_n](w\ (d_1\ x_1\ ...\ x_n)\ ...\ (d_p\ x_1\ ...\ x_n))$$

To find the d_i we use recursively the same algorithm. So we first take

$$t = [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

then we use recursively our algorithm to instantiate the $h_i : P_1 \to ... \to P_n \to Q_i$.

In unification, flexible-rigid equations are solved by instantiating the head of the flexible term using this method, the choice of the variable w is very restricted by the rigid term. Rigid-rigid equations are simplified. And flexible-flexible equations are solved in a trivial way.

So the method to enumerate terms underlying higher order unification can be summaried

- try all the possible head variables of the normal η-long form of the term,
- generate new variables for the rest of the term,
- generate a typing constraint to enforce well-typedness of the term,
- use recursively this method to fill these variables.

1.3 Higher Order Resolution

Let us present now the resolution method. We just present an incomplete restriction of this method to Horn clauses i.e. propositions on the form $Q_1 \to ... \to Q_n \to Q$, where $Q_1, ..., Q_n, Q$ are atomic propositions (free variables are considered as universally quantified in the head of the clause).

When the goal P is atomic we unify it with Q the head of an hypothesis $Q_1 \to ... \to Q_n \to Q$, we generate subgoals $\sigma Q_1, ..., \sigma Q_n$ (where σ is a unifer of P and Q) and we recursively search proofs of these propositions.

When the goal is also a Horn clause $P_1 \to ... \to P_n \to P$, then the clause form of the negation of this goal introduces P_1, ..., P_n as hypothesis and the atomic goal P.

This restriction is called the *introduction-resolution* algorithm. It can be presented with two rules: the resolution rule between an atomic goal and the head of an hypothesis is just called the *resolution* rule and the introduction of the hypothesis P_1 when we want to prove $P_1 \to P$ is called the *introduction* rule. It is incomplete. In [20] another rule (the *splitting* rule) is added to make it complete.

Remark that this method can also be used for *hereditary Horn clauses* i.e. propositions on the form $Q_1 \to ... \to Q_n \to Q$, where Q is atomic and $Q_1, ..., Q_n$ are hereditary Horn clauses.

1.4 Introduction-Resolution in Type Systems

In type systems, all the propositions can be considered as hereditary Horn clauses where the arrow is generalized to a dependent product, so we can apply the same method.

Furthermore, in type systems, a proof-synthesis method must not only assert that the proposition is provable but also exhibit a proof-term of this proposition.

The introduction-resolution algorithm can be presented that way:

- *Introduction*: To find a proof of a proposition $(x : P_1)P$ in a context Γ, find a proof of P in the context $\Gamma[x : P_1]$. When we have a proof t of P, the term $[x : P_1]t$ is a proof of $(x : P_1)P$ in the context Γ.
- *Resolution*: To find a proof of P in a context Γ where there is a proposition

  $$f : (x_1 : Q_1)...(x_n : Q_n)Q$$

  unify P and Q (consider $x_1, ..., x_n$ as variables in Q) and then if x_i has not been instantiated during the unification process, find a proof of σQ_i (where σ is a unifier of P and Q). For each i, we get (by unification or as a proof of a subgoal) a term c_i of type $Q_i[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}]$. The term $(f\ c_1\ ...\ c_n)$ is a proof of P.

This method is fully described in [18]. Let us give an example. We have an hypothesis

$$f : (x : T)(y : T)(z : T)(u : (R\ x\ y))(v : (R\ y\ z))(R\ x\ z)$$

and we want to prove $(R\ a\ c)$. We unify $(R\ x\ z)$ and $(R\ a\ c)$ this gives the substitution $x \leftarrow a, z \leftarrow c$. Then we have to find terms $b : T$, $t : (R\ a\ b)$ and $u : (R\ b\ c)$. The term $(f\ a\ b\ c\ t\ u)$ is a proof of $(R\ a\ c)$.

This method is in general incomplete. Another drawback is that no general unification algorithm is known for type systems. Algorithms are known only for the simply typed λ-calculus [21] [22] and the $\lambda\Pi$-calculus [10] [11] [28].

An alternative presentation of this method is the following

- *Introduction*: we give the proof $[x : P_1]h$ for the proposition and the variable h is a subgoal to be proved.
- *Resolution*: we give the proof $(f\ h_1\ ...\ h_p)$ for the proposition together with an equation $Q[x_1 \leftarrow h_1, ..., x_p \leftarrow h_p] = P$ (in our example $(R\ h_1\ h_3) = (R\ a\ b)$) to force the term $(f\ h_1\ ...\ h_p)$ to have type P. We solve this unification problem which fills some of the variables h_i (here h_1 and h_3) and then the other variables (here h_2, h_4 and h_5) are subgoals to be proved.

When, to prove a proposition, we first apply the introduction rule n times then the resolution rule, we give the proof $[x_1 : P_1]...[x_n : P_n](f\ h_1\ ...\ h_p)$ i.e. we give f for the head variable of proof. Remark that the dependence of the h_i on the x_j is implicit.

So the method to enumerate terms underlying the introduction-resolution method is

- try all the possible head variables of the normal η-long form of the term,
- generate new variables for the rest of the term,
- generate a typing constraint to enforce well-typedness of the term,
- use recursively this method to fill these variables.

So this method is the same as the method underlying higher order unification.

1.5 A Method to Enumerate the Terms of a Given Type

Now we use this idea to construct a complete proof synthesis method for type systems. To enumerate the terms t that we can substitute to a variable x of type T we imagine the normal η-long form of t

$$t = [x_1 : P_1]...[x_n : P_n](w\ c_1\ ...\ c_p)$$

or

$$t = [x_1 : P_1]...[x_n : P_n](y : A)B$$

and we perform all the elementary substitutions

$$x \leftarrow [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

$$x \leftarrow [x_1 : P_1]...[x_n : P_n](y : (h_1\ x_1\ ...\ x_n))(h_2\ x_1\ ...\ x_n\ y)$$

Then we enumerate all the terms that can be substituted to the variables $h_1, ..., h_p$.

Because we consider terms in η-long form, the number of abstractions n is the number of products in the type T and the types $P_1, ..., P_n$ are the types of the variables bound in these products. So for completeness we have to consider

- all the possible w,
- all the possible p and types for $h_1, ..., h_p$.

When we perform such a substitution, we have to make sure to get a term of type T. This well-typedness constraint is an equation.

In the introduction-resolution algorithm, the solution of this constraint is called the unification step of resolution. In unification in simply typed λ-calculus, types are always ground, so the constraint relates ground terms an we only have to check that they are identical, this is the selection of the head variables step. In unification in the $\lambda\Pi$-calculus [10] [11] [28] this constraint is added to the set of equations, it is the *accounting* equation.

Here we keep this equation in the context. Equations are constraints that the forthcoming substitutions must verify. Since we do not have a special algorithm to solve these equations but use the standard method to fill the variables and use the equations as constraints on the substitutions, this method is a one level process merging resolution and unification.

1.6 Number of Applications

In the introduction-resolution algorithm when we want to prove an atomic proposition P and we resolve it with a variable $w : (y_1 : Q_1)...(y_q : Q_q)Q$ (Q atomic) we only consider proofs of the form $(w\ h_1\ ...\ h_q)$ and the constraint is that the type of this term must be equal to P i.e. $Q[y_1 \leftarrow h_1, ..., y_q \leftarrow h_q] = P$.

But in polymorphic type systems the term $(w\ c_1\ ...\ c_q)$ can still have a type which is a product and we can go on applying variables. This is why the introduction-resolution method is incomplete and why a splitting rule is needed in [20].

Here we consider an infinite number of possibilities for the number of applications and an infinitely branching search tree, we use interleaving to enumerate its nodes. In an incomplete but more efficient method we consider a restriction which is more or less similar to the introduction-resolution method.

1.7 Variables in the Proposition to be Proved

In some type systems, the variables h_i may have occurrences in the type of h_j for $j > i$. We could first instantiate the variable h_i and then instantiate the variable h_j when it has a ground type, but it is well known that it is more efficient to solve the variable h_j first. For instance to enumerate the terms of the type $T = \exists x : Nat.(P\ x)$ we must not enumerate all the terms n of type Nat and for each n enumerate the terms of type $(P\ n)$, but enumerate first the terms of type $(P\ x)$.

Thus in polymorphic type systems when we have a variable in the proposition to be proved, it is not always possible to know the number and the types of abstractions: if the proposition to be proved is $T = (x_1 : P_1)...(x_n : P_n)P$ (P atomic) and the head of P is a variable that can be substituted then a substitution may increase the number of products in T and therefore the number of abstractions in t. So we have to delay the instantiation of $x : T$ until we have instantiated T in a term $(x_1 : P_1)...(x_n : P_n)P$ where the head of P cannot be substituted.

1.8 η-conversion

In this paper we consider pure type systems extended with η-conversion. The methods developed here should also be applicable for pure type systems without η-conversion but, as in higher order unification [21] [22], we would need to consider more elementary substitutions.

2 Pure Type Systems Extended with η-conversion

2.1 Definition

A pure type system extended with η-conversion [1] is a λ-calculus given by a set S (the elements of S are called *sorts*), a subset Ax of $S \times S$ and a subset R of $S \times S \times S$.

Definition 1 Functional Type System
A type system is said to be *functional* if

$$< s, s' > \in Ax \text{ and } < s, s'' > \in Ax \text{ implies } s' = s''$$

$$< s, s', s'' > \in R \text{ and } < s, s', s''' > \in R \text{ implies } s'' = s'''$$

In this paper we consider systems such that $S = \{Prop, Type\}$, $Ax = \{< Prop, Type >\}$, if $< s, s', s'' > \in R$ then $s' = s''$ and $< Prop, Prop, Prop > \in R$. These systems are the eight systems of Barendregt's cube. All these systems are functional. Examples are the simply typed λ-calculus, the $\lambda\Pi$-calculus [19] [7] [16], the system F, the system Fω [15] and the Calculus of Constructions [2] [5].

Definition 2 Syntax

$$T \ ::= \ s \mid x \mid (T \ T) \mid [x : T]T \mid (x : T)T$$

In this paper we ignore variable renaming problems. A rigorous presentation would use de Bruijn indices [6]. The terms s are sorts, the terms x are called variables, the terms $(T \ T')$ applications, the terms $[x : T]T'$ λ-abstractions and the terms $(x : T)T'$ products. The notation $T \to T'$ is used for $(x : T)T'$ when x has no free occurrence in T'.

Let t and t' be terms and x a variable. We write $t[x \leftarrow t']$ for the term obtained by substituting t' for x in t. We write $t \equiv t'$ when t and t' are $\beta\eta$-equivalent.

Definition 3 Context
A *context* Γ is a list of pairs $< x, T >$ (written $x : T$) where x is a variable and T a term. The term T is called the *type* of x in Γ.

We write $[x_1 : T_1; ...; x_n : T_n]$ for the context with elements $x_1 : T_1, ..., x_n : T_n$ and $\Gamma_1\Gamma_2$ for the concatenation of the contexts Γ_1 and Γ_2.

Definition 4 Typing Rules
We define inductively two judgements: Γ *is well-formed* and t *has type* T *in* Γ ($\Gamma \vdash t : T$) where Γ is a context and t and T are terms.
Empty:

$$\frac{}{[\,] \text{ well-formed}}$$

Declaration:

$$\frac{\Gamma \vdash T : s}{\Gamma[x:T] \text{ well-formed}}\, s \in S$$

Sort:

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash s : s'} < s, s' > \in Ax$$

Variable:

$$\frac{\Gamma \text{ well-formed} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

Product:

$$\frac{\Gamma \vdash T : s \quad \Gamma[x:T] \vdash T' : s'}{\Gamma \vdash (x:T)T' : s''} < s, s', s'' > \in R$$

Abstraction:

$$\frac{\Gamma \vdash (x:T)T' : s \quad \Gamma[x:T] \vdash t : T'}{\Gamma \vdash [x:T]t : (x:T)T'}\, s \in S$$

Application:

$$\frac{\Gamma \vdash t : (x:T)T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

Conversion:

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'}\, s \in S$$

Definition 5 Well-typed Term

A term t is said to be *well-typed* in a context Γ if there exists a term T such that $\Gamma \vdash t : T$.

Definition 6 Type

A term T is said to be a *type* in the context Γ if it is a sort or if there exists a sort s such that $\Gamma \vdash T : s$.

Proposition 1 If $\Gamma \vdash t : T$ then T is a type.
Proof By induction on the length of the derivation of $\Gamma \vdash t : T$.

Proposition 2 The $\beta\eta$-reduction is strongly normalizable and confluent on well-typed terms. Thus a well-typed term has a unique $\beta\eta$-normal form. Two well-typed terms are equivalent if they have the same $\beta\eta$-normal form.
Proof Proofs of strong normalization and confluence for β-reduction are given in [2] [13]. As remarqued in [12] the strong normalization proof of [13] can be adapted to $\beta\eta$-reduction and does not need confluence. Then confluence proofs for $\beta\eta$-reduction are given in [12] [30] [4].

Proposition 3 In a functional type system, in which reduction is confluent, a term t well-typed in a context Γ has a unique type modulo $\beta\eta$-equivalence.
Proof By induction over the structure of t.

Definition 7 Atomic Term

A term t is said to be *atomic* if it has the form $(u\ c_1\ ...\ c_n)$ where u is a variable or a sort. The symbol u is called the *head* of the term t.

Proposition 4 A normal well-typed term t is either an abstraction, a product or an atomic term.
Proof If the term t is neither an abstraction nor a product then it can be written in a unique way

$$t = (u\ c_1\ ...\ c_n)$$

where u is not an application. The term u is not a product (if $n \neq 0$ because a product is of type s for some sort s and therefore cannot be applied and if $n = 0$ because t is not a product). It is not an abstraction (if $n \neq 0$ because t is in normal form and if $n = 0$ because t is not an abstraction). It is therefore a variable or a sort.

Proposition 5 Let T be a well-typed normal type, the term T can be written in a unique way $T = (x_1 : P_1)...(x_n : P_n)P$ with P atomic.
Proof By induction over the structure of T.

Definition 8 η-long form

Let Γ be a context and t be a $\beta\eta$-normal term well-typed in Γ and T the $\beta\eta$-normal form of its type. The *η-long form* of the term t is defined as

- If $t = [x : U]u$ then the η-long form of t is $[x : U']u'$ where U' is the η-long form of U in Γ and u' the η-long form of u in $\Gamma[x : U]$.
- If $t = (x : U)V$ then the η-long form of t is $(x : U')V'$ where U' is the η-long form of U in Γ and V' the η-long form of V in $\Gamma[x : U]$.
- If $t = (w\ c_1\ ...\ c_p)$ then we let $T = (x_1 : P_1)...(x_n : P_n)P$ (P atomic). The η-long form of t is $[x_1 : P'_1]...[x_n : P'_n](w\ c'_1\ ...\ c'_p\ x'_1\ ...\ x'_n)$ where c'_i is the η-long form of c_i in Γ, P'_i the η-long form of P_i in $\Gamma[x_1 : P_1; ...; x_{i-1} : P_{i-1}]$ and x'_i the η-long form of x_i in $\Gamma[x_1 : P_1; ...; x_i : P_i]$.

The well-foundedness of this definition is proved in [8] [9].

Definition 9 normal η-long form

Let t be a term well-typed in a context Γ, the *normal η-long form* of t is the η-long form of its $\beta\eta$-normal form.

Definition 10 Subterm

We consider well-typed normal η-long terms labeled with the contexts in which they are well-typed: t_Γ. Let t_Γ such a term, we define by induction over the structure of t_Γ the set $Sub(t_\Gamma)$ of *strict subterms* of t_Γ

- if t_Γ is a sort or a variable then $Sub(t_\Gamma) = \{\}$,
- if t_Γ is an application, $t = (u\ v)$, then $Sub(t_\Gamma) = \{u_\Gamma, v_\Gamma\} \cup Sub(u_\Gamma) \cup Sub(v_\Gamma)$,
- if t_Γ is an abstraction, $t = [x : P]u$, then $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$,
- if t_Γ is a product, $t = (x : P)u$, then $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$.

2.2 The $Meta$ Type System

Let $\mathcal{T}$ be a type system of the cube. In order to express the proof synthesis method, we have to be allowed to declare a variable that stands for any well-typed term of $\mathcal{T}$. For instance a variable that stands for $Prop$ i.e. a variable of type $Type$ and this is not possible in $\mathcal{T}$ because the term $Type$ is not well-typed.

We want also to be allowed to express a term $t : T'$ which is well-typed in $\Gamma[x : T]$ as $t = (f\ x)$ with f well-formed in Γ. We want actually to be allowed to define $f = [x : T]t$ whatever the terms T and t may be. In general this cannot be done in $\mathcal{T}$, because the type $(x : T)T'$ may be not well-typed. So we are going to embed our type system $\mathcal{T}$ in another type system: the $Meta$ type system.

Definition 11 $Meta$

$$Meta =< S', Ax', R' >$$

$$S' = \{Prop, Type, Extern\}$$

$$Ax' = \{< Prop, Type >, < Type, Extern >\}$$

$$R' = \{< Prop, Prop, Prop >, < Prop, Type, Type >, < Type, Prop, Prop >,$$
$$< Type, Type, Type >, < Prop, Extern, Extern >, < Type, Extern, Extern >\}$$

The strong normalization and confluence of $\beta\eta$-reduction for this system are not yet fully proved. But since all the terms typable in the $Meta$ type system are typable in the Calculus of Constructions with Universes [3] [25] (identifying the sorts $Extern$ and $Type_1$), the β-reduction is strongly normalizable and confluent [3] [25]. It is conjectured in [12] that in a pure type system in which the β-reduction is strongly normalizable the $\beta\eta$-reduction also is. Then assuming strong normalization, confluence is proved in [12] [30] [4].

Proposition 6 Because the $Meta$ type system is functional and $\beta\eta$-reduction is confluent, if a term t is well-typed in a context Γ then it has a unique type modulo $\beta\eta$-equivalence.

3 Constrained Quantified Contexts and Substitution

3.1 Constrained Quantified Contexts

Definition 12 Constrained Quantified Contexts

A *quantified declaration* is a triple $< Q, x, T >$ (written $Qx : T$) where Q is a quantifier ($\forall$ or $\exists$), x a variable and T is a term. A *constraint* is a pair of terms $< a, b >$ (written $a = b$). A *constrained quantified context* is a list of quantified declarations and constraints. If Γ contains the declaration $\forall x : T$ then the variable x is said to be *universal* in Γ. If it contains the declaration $\exists x : T$ then x is said to be *existential* in Γ. These constrained quantified contexts are generalizations of Miller's *mixed prefixes* [26] which are lists of quantified declarations.

Usual contexts are identified with constrained quantified contexts with only universal variables.

Definition 13 Equivalence Modulo Constraints

Let Γ be a constrained quantified context, we define the relation between terms $\equiv_\Gamma$ as the smallest equivalence relation compatible with terms structure such that

- if $t \equiv t'$ then $t \equiv_\Gamma t'$,
- if $(a = b) \in \Gamma$ then $a \equiv_\Gamma b$.

Definition 14 Typing Rules

First we modify the rules to deal with the new syntax.
The *declaration* rule is modified in

$$\frac{\Gamma \vdash T : s}{\Gamma[Qx : T] \text{ well-formed}}\, s \in S$$

the *variable* rule is modified in

$$\frac{\Gamma \text{ well-formed } \ Qx : T \in \Gamma}{\Gamma \vdash x : T}$$

the *product* rule is modified in

$$\frac{\Gamma \vdash T : s \ \ \Gamma[Qx : T] \vdash T' : s'}{\Gamma \vdash (x : T)T' : s''} < s, s', s'' > \in R$$

the *abstraction* rule is modified in

$$\frac{\Gamma \vdash (x : T)T' : s \ \ \Gamma[Qx : T] \vdash t : T'}{\Gamma \vdash [x : T]t : (x : T)T'}\, s \in S$$

and we add a *constraint* rule

$$\frac{\Gamma \vdash a : T \ \ \Gamma \vdash b : T}{\Gamma[a = b] \text{ well-formed}}$$

Then we extend the system by replacing the *conversion* rule by

$$\frac{\Gamma \vdash T : s \ \ \Gamma \vdash T' : s \ \ \Gamma \vdash t : T \ \ T \equiv_\Gamma T'}{\Gamma \vdash t : T'}\, s \in S$$

This defines two new judgements: Γ is *well-formed using the constraints* and t *has type* T *in* Γ *using the constraints.*

Remark that a term may be well-typed in Γ using the constraints and still be not normalizable.

Definition 15 Well-typed Without Using the Constraints

Let Γ be a context and t and T be two terms. The term t is said to be *of type* T *in* Γ *without using the constraints* if there exists Δ subcontext of Γ (i.e. obtained by removing some items of Γ) such that Δ has no constraints, is a well-formed context and $\Delta \vdash t : T$.

Proposition 7 If a term is well-typed in a context without using the constraints then it is strongly normalizable.

Definition 16 Normal Form of a Context

Let Γ be a well-formed context, the *normal form* of Γ is obtained by normalizing (in normal η-long form) all the types of variables that are well-typed without using the constraints and all the constraints which terms are well-typed without using the constraints.

Proposition 8 Let Γ be a well-formed context and Γ' be its normal form. The context Γ' is well-formed and if $\Gamma \vdash t : T$ then $\Gamma' \vdash t : T$.
Proof By induction on the length of the derivation of Γ *well-formed* and $\Gamma \vdash t : T$.

Definition 17 Ground Term

A term t is said to be *ground* in a context Γ if it has no occurrence of an existential variable.

Definition 18 Rigid and Flexible Terms

A normal η-long term well-typed without the constraints which is an abstraction, a product or an atomic term $(w\ c_1\ ...\ c_n)$ with w universal variable or sort is said to be *rigid*. A normal η-long term well-typed without the constraints which is atomic $(w\ c_1\ ...\ c_n)$ with w existential variable is said to be *flexible*.

Definition 19 Success and Failure Contexts

A normal well-formed context Γ is said to be *a success context* if it has only universal variables and constraints relating identical terms. It is said to be *a failure context* if it contains a constraint relating two normal η-long ground terms which are not identical.

Proposition 9 Let Γ be a normal context which is neither a success context nor a failure context. Then there exists an existential variable $x : T$ whose type T is well-typed without using the constraints and has the form $(x_1 : P_1)...(x_n : P_n)P$ with the term P atomic and rigid in the context $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.
Proof Let us consider the leftmost item which is neither a universal variable nor a normal η-long constraint relating identical terms (such an item exists since the context is not a success context).

If this item is a constraint $(a = b)$, we let $\Gamma = \Delta[a = b]\Delta'$, the terms a and b are well-typed in Δ so, since there are neither existential variables nor (non trivial) constraints in Δ, they are ground, well-typed without the constraints and normal. These terms are different and the context Γ is a failure context. So if Γ is not a failure context then this item is an existential variable $\exists x : T$ and T is well-typed without the constraints and ground.

3.2 Substitution

Definition 20 Existential Context

A context (well-formed or not) is said to be *existential* if it contains only declarations of existential variables and constraints but no declarations of universal variables.

Definition 21 Substitution

A finite set σ of triples $< x, \gamma, t >$ where x is a variable, γ an existential context and t a term is said to be a substitution if for each variable x there is at most one triple of the form $< x, \gamma, t >$ in σ. When σ contains only one triple $< x, \gamma, t >$ we write it $\sigma = x \leftarrow \gamma, t$.

Definition 22 Variable Bound by a Substitution

Let x be a variable and σ be a substitution. If the substitution σ contains a triple $< x, \gamma, t >$ then x is said to be *bound* by σ. The context γ is said to be *the context associated to x in σ*.

Definition 23 Substitution Applied to a Term

Let x be a variable and σ a substitution. If there is a triple $< x, \gamma, t >$ in σ then we let $\sigma x = t$ else we let $\sigma x = x$. This definition extends straightforwardly to terms by

- $\sigma s = s$,
- $\sigma(t\ u) = (\sigma t\ \sigma u)$,
- $\sigma[x : T]u = [x : \sigma T]\sigma u$,
- $\sigma(x : T)u = (x : \sigma T)\sigma u$.

Definition 24 Substitution Applied to a Context
Let Γ be a context and σ a substitution, the context $\sigma\Gamma$ is defined inductively by

- $\sigma[\,] = [\,]$
- $\sigma(\Delta[Qx : T]) = (\sigma\Delta)\gamma$ where γ is the context associated to x by σ if the variable x is bound by σ and $\gamma = [Qx : \sigma T]$ otherwise.
- $\sigma(\Delta[a = b]) = (\sigma\Delta)[\sigma a = \sigma b]$

Definition 25 Substitution Well-typed in a Context
The substitution σ is said to be *well-typed* in a context Γ if and only if the context $\sigma\Gamma$ is well-formed, no universal variable of Γ is bound by σ and for each existential variable x of type T in Γ and bound by σ, we have $(\sigma\Delta)\gamma \vdash t : \sigma T$ where Δ is the unique context such that $\Gamma = \Delta[\exists x : T]\Delta'$ and $< x, \gamma, t >$ the unique triple of σ binding the variable x.

Proposition 10 Let Γ be a context, σ a substitution and T and T' terms such that $T \equiv_\Gamma T'$. Then $\sigma T \equiv_{\sigma\Gamma} \sigma T'$.
Proof By induction on the length of the derivation of $T \equiv_\Gamma T'$.

Proposition 11 Let σ be a substitution, t and u two terms and x a variable which is not bound by σ and which is not free in any σy for $y \neq x$. Then $\sigma(t[x \leftarrow u]) = (\sigma t)[x \leftarrow \sigma u]$.
Proof By induction over the structure of t.

Proposition 12 Let Γ be a context, σ a substitution well-typed in Γ and x a variable declared of type T in Γ. Then $\sigma\Gamma \vdash \sigma x : \sigma T$.
Proof Since the substitution σ is well-typed in Γ, the context $\sigma\Gamma$ is well-formed. If x not bound by Γ then $\sigma x = x$ and $Qx : \sigma T \in \sigma\Gamma$ so $\sigma\Gamma \vdash \sigma x : \sigma T$. If x is bound by σ then there exists a unique triple $< x, \gamma, t >$ in σ. Let us write $\Gamma = \Delta[\exists x : T]\Delta'$. We have $\sigma x = t$. Since σ is well-typed in Γ we have $(\sigma\Delta)\gamma \vdash t : \sigma T$ and $(\sigma\Delta)\gamma$ is a prefix of $\sigma\Gamma$. So $\sigma\Gamma \vdash \sigma x : \sigma T$.

Proposition 13 Let Γ be a context, σ a substitution well-typed in Γ, t and T two terms such that $\Gamma \vdash t : T$. We have $\sigma\Gamma \vdash \sigma t : \sigma T$.
Proof By induction on the length of the derivation of $\Gamma \vdash t : T$.

- If the last rule is the *sort* rule then we use the fact that $\sigma\Gamma$ is well-formed.
- If the last rule is the *variable* rule then we use the fact that $\sigma\Gamma$ is well-formed and $\sigma\Gamma \vdash \sigma x : \sigma T$.
- If the last rule is the *product* rule then by induction hypothesis we have $\sigma\Gamma \vdash \sigma T : s$ and $(\sigma\Gamma)[\forall x : \sigma T] \vdash \sigma T' : s'$, so $\sigma\Gamma \vdash \sigma(x : T)T' : s''$.

- If the last rule is the *abstraction* rule then by induction hypothesis we have $\sigma\Gamma \vdash \sigma(x:T)T' : s$ and $(\sigma\Gamma)[\forall x : \sigma T] \vdash \sigma t : \sigma T'$, so $\sigma\Gamma \vdash \sigma[x:T]t : \sigma(x:T)T'$.
- If the last rule is the *application* rule then by induction hypothesis we have $\sigma\Gamma \vdash \sigma t : \sigma(x:T)T'$ and $\sigma\Gamma \vdash \sigma t' : \sigma T$, so we get $\sigma\Gamma \vdash \sigma(t\ t') : (\sigma T')[x \leftarrow \sigma t']$ and since x is not bound by σ and is not free in any σy for $y \neq x$, we have $\sigma\Gamma \vdash \sigma(t\ t') : \sigma(T'[x \leftarrow t'])$.
- If the last rule is the *conversion* rule then by induction hypothesis we have $\sigma\Gamma \vdash \sigma T : s$, $\sigma\Gamma \vdash \sigma T' : s$ and since $T \equiv_\Gamma T'$ we have $\sigma T \equiv_{\sigma\Gamma} \sigma T'$, so $\sigma\Gamma \vdash \sigma t : \sigma T'$.

Definition 26 Composition of Substitutions
Let σ and τ two substitutions. The substitution $\tau \circ \sigma$ is defined as

$$(\tau \circ \sigma) = \{< x, \tau\gamma, \tau t > \ | \ < x, \gamma, t > \in \sigma\} \cup \{< x, \gamma, t > \ | \ < x, \gamma, t > \in \tau \text{ and } x \text{ not bound by } \sigma\}$$

Proposition 14 Let σ and τ two substitutions and t be a term, we have $(\tau \circ \sigma)t = \tau\sigma t$.
Proof By induction over the structure of t.

Proposition 15 Let σ and τ two substitutions and Γ be a context, we have $(\tau \circ \sigma)\Gamma = \tau\sigma\Gamma$.
Proof By induction over the length of Γ.

Proposition 16 Let Γ be a context and σ and τ two substitutions, such that σ is well-typed in Γ and τ is well-typed in $\sigma\Gamma$ then $\tau \circ \sigma$ is well-typed in Γ.
Proof We have $(\tau \circ \sigma)\Gamma = \tau\sigma\Gamma$, so the context $(\tau \circ \sigma)\Gamma$ is well-formed. No universal variable of Γ is bound by $\tau \circ \sigma$. If $\Gamma = \Delta[\exists x : T]\Delta'$ then $\sigma(\Delta[\exists x : T]) \vdash \sigma x : \sigma T$ so using the previous proposition $\tau\sigma(\Delta[\exists x : T]) \vdash \tau\sigma x : \tau\sigma T$, i.e. $(\tau \circ \sigma)(\Delta[\exists x : T]) \vdash (\tau \circ \sigma)x : (\tau \circ \sigma)T$.

4 A Complete Method

Informal Introduction

When we search a proof of a proposition T in a context Γ we search a substitution σ well-typed in $\Gamma[\exists x : T]$ such that $\sigma(\Gamma[\exists x : T])$ is a success context. When we have a context which contains several existential variables we choose such a variable x of type $T = (x_1 : P_1)...(x_n : P_n)P$ with P atomic rigid in $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ and we perform an elementary substitution instantiating this variable. In the general case the elementary substitutions have the form

$$x \leftarrow [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

Let us write $(y_1 : Q_1)(y_2 : (Q_2\ y_1))...(y_q : (Q_q\ y_1\ ...\ y_{q-1}))(Q\ y_1\ ...\ y_q)$ the type of w. For the substitutions such that $p = q$ we only need to declare the new variables $h_1, ..., h_q$ used in the substitution and the constraint expressing the well-typedness of the substitution

$$(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P$$

(where $(\vec{x} : \vec{P})t$ is an abbreviation for $(x_1 : P_1)...(x_n : P_n)t$). For the substitutions such that $p = q + r$, $r \geq 1$, we first declare the variables $h_1, ..., h_q$, then we need the type of the term

$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$ to be a product, we introduce therefore two existential variables H_1 and K_1 and a constraint

$$(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

Then we can introduce the variable h_{q+1} with the type $(\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)$. If $r = 1$ we just need a last constraint expressing the well-typedness of the substitution

$$(\vec{x} : \vec{P})(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P$$

and in the general case we introduce in the same way $4r$ items and then a well-typedness constraint.

We also have to include other elementary substitutions in which the variable x is substituted by a term of the form $[x_1 : P_1]...[x_n : P_n]u$ where u is a sort or a product.

Definition 27 Elementary Substitutions

Let Γ be a normal context well-formed in the *Meta* type system such that the type of the type of universal variables is *Prop* or *Type* but not *Extern* and which is neither a success nor a failure context. We choose in Γ an existential variable x whose type is normal η-long and has the form $T = (x_1 : P_1)...(x_n : P_n)P$ with P atomic rigid in $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ (such a variable exists because the context in neither a success nor a failure context) and we construct the set of substitutions $\Sigma(\Gamma)$ in the following way.

Notation If t is a term, $(\vec{x} : \vec{P})t$ is an abbreviation for $(x_1 : P_1)...(x_n : P_n)t$.

- For every w which is a universal variable declared in the left of x or which is an x_i

  $$\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n] \vdash w : (y_1 : Q'_1)...(y_q : Q'_q)Q'\ (Q' \text{ atomic})$$

  We let

  $$Q_1 = Q'_1$$
  $$Q_2 = [y_1 : Q'_1]Q'_2$$
  $$...$$
  $$Q_q = [y_1 : Q'_1]...[y_{q-1} : Q'_{q-1}]Q'_q$$
  $$Q = [y_1 : Q'_1]...[y_q : Q'_q]Q'$$

  We have

  $$w : (y_1 : Q_1)(y_2 : (Q_2\ y_1))...(y_q : (Q_q\ y_1\ ...\ y_{q-1}))(Q\ y_1\ ...\ y_q)$$

  For every $r \geq 0$ we consider the following *substitutions with r-splitting*. Let s be the sort type of Q' and s' the sort type of P in $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$. For every sequence $s_1, s'_1, ..., s_i, s'_i, ..., s_r, s'_r$ such that $< s_1, s'_1, s >\in R$, ..., $< s_i, s'_i, s'_{i-1} >\in R$ and $s'_r = s'$, we consider the substitution

  $$x \leftarrow \gamma, t$$

  where

  $$t = [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n))$$
  $$\gamma = \varphi\chi_1...\chi_r\psi$$

with

$$\varphi = [\exists h_1 : (\vec{x} : \vec{P})Q_1;$$
$$\exists h_2 : (\vec{x} : \vec{P})(Q_2\ (h_1\ x_1\ ...\ x_n));$$
$$...;$$
$$\exists h_q : (\vec{x} : \vec{P})(Q_q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q-1}\ x_1\ ...\ x_n))]$$
$$\chi_1 = [\exists H_1 : (\vec{x} : \vec{P})s_1;$$
$$\exists K_1 : (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))s_1';$$
$$(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z);$$
$$\exists h_{q+1} : (\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)]$$

for all i, $1 < i \leq r$

$$\chi_i = [\exists H_i : (\vec{x} : \vec{P})s_i;$$
$$\exists K_i : (\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))s_i';$$
$$(\vec{x} : \vec{P})(K_{i-1}\ x_1\ ...\ x_n\ (h_{q+i-1}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))(K_i\ x_1\ ...\ x_n\ z);$$
$$\exists h_{q+i} : (\vec{x} : \vec{P})(H_i\ x_1\ ...\ x_n)]$$

and if $r = 0$

$$\psi = [(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P]$$

otherwise

$$\psi = [(\vec{x} : \vec{P})(K_r\ x_1\ ...\ x_n\ (h_{q+r}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P]$$

- If P is a sort then for every sort s, such that $< s, P > \in Ax$, we consider also the substitution

$$x \leftarrow [\ ], [x_1 : P_1]...[x_n : P_n]s$$

and for every pair of sorts $< s, s' >$ such that $< s, s', P > \in R$ we also consider the substitution

$$x \leftarrow \gamma, t$$

where

$$t = [x_1 : P_1]...[x_n : P_n](y : (h\ x_1\ ...\ x_n))(k\ x_1\ ...\ x_n\ y)$$
$$\gamma = [\exists h : (\vec{x} : \vec{P})s\ ;\ \exists k : (\vec{x} : \vec{P})(y : (h\ x_1\ ...\ x_n))s']$$

The set $\Sigma(\Gamma)$ is the set which contains all the substitutions considered above.

Definition 28 Derivation

A *derivation* of a context Γ is a list of substitutions $[\sigma_1; ...; \sigma_m]$ such that $\sigma_i \in \Sigma(\sigma_{i-1}\sigma_{i-2}...\sigma_1\Gamma)$ and $\sigma_m\sigma_{m-1}...\sigma_1\Gamma$ is a success context.

Definition 29 Search tree

Let Γ be a context, we build a tree, called *the search tree* of Γ. Nodes are labeled by contexts and edges by elementary substitutions. The root is labeled by Γ. Nodes labeled by success and failure contexts are leaves. From a node labeled by a context Δ which is neither a success context nor failure context, for each σ of $\Sigma(\Delta)$ we grow an edge labeled σ to a new node labeled by $\sigma\Delta$.

Success nodes are in bijection with the derivations of Γ. A semi-algorithm of proof synthesis is to enumerate the nodes of the tree in order to find a success node. Since the number of sons of a node may be infinite we have, in order to get an finitely branching tree, to delay the exploration of node with r-splitting for r generations.

5 Examples

5.1 A First Order Example

Let $\Delta = [\forall T : Prop; \forall R : T \to T \to Prop; \forall Eq : T \to T \to Prop;$
$\forall Antisym : (x : T)(y : T)((R\ x\ y) \to (R\ y\ x) \to (Eq\ x\ y)); \forall a : T; \forall b : T; \forall u : (R\ a\ b); \forall v : (R\ b\ a)]$
and $\Gamma = \Delta[\exists x : (Eq\ a\ b)]$.
For x, we perform the substitution with head $Antisym$ and 0-splitting

$$x \leftarrow (Antisym\ h_1\ h_2\ h_3\ h_4)$$

We get the context $\Delta[\exists h_1 : T; \exists h_2 : T; \exists h_3 : (R\ h_1\ h_2); \exists h_4 : (R\ h_2\ h_1); (Eq\ h_1\ h_2) = (Eq\ a\ b)]$.
For h_1, we perform the substitution with head a and 0-splitting

$$h_1 \leftarrow a$$

For h_2, we perform the substitution with head b and 0-splitting

$$h_2 \leftarrow b$$

We get the context $\Delta[\exists h_3 : (R\ a\ b); \exists h_4 : (R\ b\ a)]$ (we do not write the trivial constraints).
For h_3, we perform the substitution with head u and 0-splitting

$$h_3 \leftarrow u$$

We get the context $\Delta[\exists h_4 : (R\ b\ a)]$.
For h_4, we perform the substitution with head v and 0-splitting

$$h_4 \leftarrow v$$

And we get the success context Δ.
Let θ be the composition of these substitutions, $\theta x = (Antisym\ a\ b\ u\ v)$.

As we will see in the following, in this example, at each step, all the substitutions but the one considered leed to obviously hopeless contexts.

5.2 An Example with Splitting

Let $\Delta = [\forall A : Prop; \forall B : Prop; \forall I : Prop \to Prop; \forall u : (P : Prop)((I\ P) \to P);$
$\forall v : (I\ (A \to B)); \forall w : A]$
and $\Gamma = \Delta[\exists p : B]$.
For p, we perform the substitution with head u and 1-splitting

$$p \leftarrow (u\ h_1\ h_2\ h_3)$$

with the new variables: $h_1 : Prop$, $h_2 : (I\ h_1)$, $H : Prop$, $K : H \to Prop$ and $h_3 : H$.
We get the context
$\Delta[\exists h_1 : Prop; \exists h_2 : (I\ h_1); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H; h_1 = (x : H)(K\ x); (K\ h_3) = B]$.
For h_1, we perform a product substitution

$$h_1 \leftarrow (x : h')(k'\ x)$$

We get the context
$\Delta[\exists h' : Prop; \exists k' : h' \to Prop; \exists h_2 : (I\ ((x : h')(k'\ x))); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H;$
$(x : H)(K\ x) = (x : h')(k'\ x); (K\ h_3) = B]$.
For h_2 we perform the substitution with head v and 0-splitting

$$h_2 \leftarrow v$$

We get the context
$\Delta[\exists h' : Prop; \exists k' : h' \to Prop; (I\ ((x : h')(k'\ x))) = (I\ (A \to B)); \exists H : Prop; \exists K : H \to Prop;$
$\exists h_3 : H; (x : H)(K\ x) = (x : h')(k'\ x); (K\ h_3) = B]$.
For h' we perform the substitution with head A and 0-splitting

$$h' \leftarrow A$$

$\Delta[\exists k' : A \to Prop; (I\ ((x : A)(k'\ x))) = (I\ (A \to B)); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H;$
$(x : H)(K\ x) = (x : A)(k'\ x); (K\ h_3) = B]$.
For k' we perform the substitution with head B and 0-splitting

$$k' \leftarrow [x : A]B$$

We get the context $\Delta[\exists H : Prop; \exists K : H \to Prop; \exists h_3 : H; (x : H)(K\ x) = (A \to B); (K\ h_3) = B]$.
For H we perform the substitution with head A and 0-splitting

$$H \leftarrow A$$

We get the context $\Delta[\exists K : A \to Prop; \exists h_3 : A; (x : A)(K\ x) = (A \to B); (K\ h_3) = B]$.
For K we perform the substitution with head B and 0-splitting

$$K \leftarrow [x : A]B$$

We get the context $\Delta[\exists h_3 : A]$.
For h_3 we perform the substitution with head w and 0-splitting

$$h_3 \leftarrow w$$

And we get the success context Δ.
Let θ be the composition of all these substitutions, $\theta p = (u\ (A \to B)\ v\ w)$.

6 Properties

6.1 Well-typedness

In this section Γ is a normal context well-formed in the $Meta$ type system such that the type the type of universal variables is $Prop$ or $Type$ but not $Extern$ and which is neither a success nor a failure context and $\sigma = x \leftarrow \gamma, t$ is a substitution of $\Sigma(\Gamma)$. We write $\Gamma = \Delta[\exists x : T]\Delta'$. We prove that the substitution σ is well-typed in the context Γ.

Proposition 17 The context $\Delta\gamma$ is well-formed in the $Meta$ type system.
Proof By induction on the length of γ, we check that the types of the existential variables and the constraints of γ are well-typed in the $Meta$ type system.

Since $(x_1 : P_1)...(x_n : P_n)P$ is well-typed in the $Meta$ type system, the terms P_i are well-typed in the $Meta$ type system of type $Prop$ or $Type$ but not $Extern$. The term $(y_1 : Q'_1)...(y_q : Q'_q)Q'$ is either the type of a universal variable or one of the P_i, its type is therefore $Prop$ or $Type$ but not $Extern$. So the types of the Q'_i and of Q' are $Prop$ or $Type$ but not $Extern$. So the terms Q_i and Q are well-typed in the $Meta$ type system. The terms $(H_i\ x_1\ ...\ x_n)$ and $(h\ x_1\ ...\ x_n)$ are well-typed in the $Meta$ type system and have the type s with $< s, s', s'' >\in R$ for some s' and s''. So s is equal to $Prop$ or $Type$, but not to $Extern$.

So the terms $(\vec{x} : \vec{P})(Q_i\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{i-1}\ x_1\ ...\ x_n))$, $(\vec{x} : \vec{P})s_i$, $(\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))s'_i$, $(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$, $(\vec{x} : \vec{P})(K_{i-1}\ x_1\ ...\ x_n\ (h_{q+i-1}\ x_1\ ...\ x_n))$, $(\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))(K_i\ x_1\ ...\ x_n\ z)$, $(\vec{x} : \vec{P})(H_i\ x_1\ ...\ x_n)$, $(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$, $(\vec{x} : \vec{P})(K_r\ x_1\ ...\ x_n\ (h_{q+r}\ x_1\ ...\ x_n))$, $(\vec{x} : \vec{P})P$, $(\vec{x} : \vec{P})s$ and $(\vec{x} : \vec{P})(y : (h\ x_1\ ...\ x_n))s'$ are well-typed in the $Meta$ type system and have the type $Prop$, $Type$ or $Extern$.

Proposition 18 In the $Meta$ type system we have $\Delta\gamma \vdash t : T$.
Proof For the substitution with 0-splitting we have

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$$

then using the constraint ψ

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})P$$

For the substitution with r-splitting ($r \geq 1$), we prove by induction on i that in $\Delta\gamma$, for all $i \geq 1$

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_i\ x_1\ ...\ x_n\ (h_{q+i}\ x_1\ ...\ x_n))$$

For $i = 1$ we have

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$$

so using the constraint of χ_1

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

in the context $\Delta\gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ we have

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

and since

$$h_{q+1} : (\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)$$

we deduce

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n)) : (K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$$

so in the context $\Delta\gamma$

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$$

Then if we assume this for i

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_i\ x_1\ ...\ x_n\ (h_{q+i}\ x_1\ ...\ x_n))$$

using the constraint of χ_i

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(z : (H_{i+1}\ x_1\ ...\ x_n))(K_{i+1}\ x_1\ ...\ x_n\ z)$$

in the context $\Delta\gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ we have

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (z : (H_{i+1}\ x_1\ ...\ x_n))(K_{i+1}\ x_1\ ...\ x_n\ z)$$

and since

$$h_{q+i+1} : (\vec{x} : \vec{P})(H_{i+1}\ x_1\ ...\ x_n)$$

we deduce

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i+1}\ x_1\ ...\ x_n)) : (K_{i+1}\ x_1\ ...\ x_n\ (h_{q+i+1}\ x_1\ ...\ x_n))$$

so in the context $\Delta\gamma$

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i+1}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_{i+1}\ x_1\ ...\ x_n\ (h_{q+i+1}\ x_1\ ...\ x_n))$$

So we deduce

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_r\ x_1\ ...\ x_n\ (h_{q+r}\ x_1\ ...\ x_n))$$

then using the last constraint (ψ)

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})P$$

In the same way, if $\sigma = x \leftarrow \gamma, t$ with $t = [x_1 : P_1]...[x_n : P_n]u$ where u is a sort or a product then the term t also is well-typed and has also type T in the context $\Delta\gamma$ in the $Meta$ type system.

Proposition 19 The substitution σ is well-typed in the context Γ in the $Meta$ type system.
Proof Let us write $\Delta' = [e_1; ...; e_n]$. We prove by induction on i that the substitution σ is well-typed in $\Delta[\exists x : T][e_1; ...; e_i]$ in the $Meta$ type system.

We have $\sigma(\Delta[\exists x : T]) = \Delta\gamma$ and $\sigma T = T$, so the context $\sigma(\Delta[\exists x : T])$ is well-formed in the $Meta$ type system and we have $\sigma(\Delta[\exists x : T]) \vdash t : \sigma T$. Obviously σ binds no universal variables of $\Delta[\exists x : T]$. So σ is well-typed in $\Delta[\exists x : T]$.

Let us assume now that the substitution σ is well-typed in $\Delta[\exists x : T][e_1; ...; e_i]$. The item e_{i+1} is either the declaration of a variable $Qy : P$ or a constraint $a = b$. In the first case we have

$$\Delta[\exists x : T][e_1; ...; e_i] \vdash P : s$$

for some sort s, and in the second

$$\Delta[\exists x : T][e_1; ...; e_i] \vdash a : U \quad \text{and} \quad \Delta[\exists x : T][e_1; ...; e_i] \vdash b : U$$

for some U. So since the substitution σ is well-typed in $\Delta[\exists x : T][e_1; ...; e_i]$ we have in the first case

$$\sigma(\Delta[\exists x : T][e_1; ...; e_i]) \vdash \sigma P : s$$

and in the second

$$\sigma(\Delta[\exists x : T][e_1; ...; e_i]) \vdash \sigma a : \sigma U \quad \text{and} \quad \sigma(\Delta[\exists x : T][e_1; ...; e_i]) \vdash \sigma b : \sigma U$$

So the context $\sigma(\Delta[\exists x : T][e_1; ...; e_{i+1}]) = \sigma(\Delta[\exists x : T][e_1; ...; e_i])[\sigma e_{i+1}]$ is well-formed. Then since we have $\Delta\gamma \vdash t : T$ and σ obviously binds no universal variables of $\Delta[\exists x : T][e_1; ...; e_{i+1}]$, the substitution σ is well-typed in $\Delta[\exists x : T][e_1; ...; e_{i+1}]$ in the $Meta$ type system.

6.2 Soundness

Definition 30 : Solution to a Context
Let Γ be a well-formed context, the substitution θ is said to be a *solution* to Γ if

- the substitution θ is well-typed in Γ,
- the context $\theta\Gamma$ is a success context.

Definition 31 Normal Solution to a Context
Let Γ be a well-formed context, the substitution θ, solution to Γ, is said to be a *normal solution* to Γ if

- the substitution θ binds exactly the existential variables of Γ,
- for each existential variable x of Γ, the context associated to x by θ is empty and σx is normal η-long in the context $\theta\Delta$ where Δ is the unique context such that $\Gamma = \Delta[\exists x : T]\Delta'$.

Definition 32 Normal Form of a Solution
Let Γ be a well-formed context and θ a solution to Γ. We define θ' the *normal form of* θ *in* Γ by induction over the length of Γ.

- If $\Gamma = [\,]$ then we let $\theta' = \{\}$.
- If $\Gamma = \Delta[\forall x : T]$ or $\Gamma = \Delta[a = b]$ then θ is a solution to Δ, we let θ' be the normal form of θ in Δ.
- If $\Gamma = \Delta[\exists x : T]$ then θ is a solution to Δ. We let θ_1 be the normal form of θ in Δ. We have $\theta\Gamma \vdash \theta x : \theta T$, let t be the normal η-long form of θx in $\theta\Gamma$. We let $\theta' = \theta_1 \cup \{< x, [\,], t >\}$.

Proposition 20 Let Γ be a context and Γ' be a context obtained by removing some constraints relating identical terms. If Γ is well-formed then Γ' is well-formed. If $\Gamma \vdash t : T$ then $\Gamma' \vdash t : T$.
Proof By induction on the length of the derivation of Γ *well-formed* or $\Gamma \vdash t : T$.

Proposition 21 Let Δ and γ be two contexts such that $\Delta\gamma$ is a success context and γ is an existential context. Then γ is a list of constraints relating terms well-typed without using the constraints and whose normal forms are identical.
Proof By induction on the length of γ.

Proposition 22 Let Γ be a well-formed context and θ a solution to Γ. Let θ' be the normal form of θ. Then θ' is a normal solution to θ.
Proof By induction on the length of Γ.

Lemma 1 Let Γ be a context, if there exists a derivation $[\sigma_1; ...; \sigma_m]$ of Γ, then there exists a normal solution to Γ.
Proof The substitution $\sigma_m \circ ... \circ \sigma_1$, is a solution to Γ. Let θ be its normal form.

The substitution θ called the substitution *denoted* by the derivation $[\sigma_1; ...; \sigma_m]$.

Now we prove that the substitution θ which is well-typed in the $Meta$ type system is also well-typed in the original system $\mathcal{T}$.

Proposition 23 Let Γ be a context well-formed in a type system $\mathcal{T}$ and T be either a term well-typed in Γ in the system $\mathcal{T}$ or the symbol $Type$. Let t be a normal η-long term such that $\Gamma \vdash t : T$ in the $Meta$ type system such that for every subterm of t which is a product $(x : U)U'$ if we let s be the type of U, s' be the type of U' and s'' be the type of $(x : U)U'$, we have $< s, s', s'' > \in R$ (the set of rules of the system $\mathcal{T}$) and for every subterm of t which is a sort s we have $s = Prop$ (and not $Type$), then we have $\Gamma \vdash t : T$ in the system $\mathcal{T}$.
Proof By induction over the structure of t.

- If $t = [x : U]u'$ then $T = (x : U)U'$ is well-typed in the type system $\mathcal{T}$ so U' is well-typed in $\Gamma[\forall x : U]$ in the type system $\mathcal{T}$ and by induction hypothesis $\Gamma[\forall x : U] \vdash u' : U'$ in the system $\mathcal{T}$. So $\Gamma \vdash t : T$ in the system $\mathcal{T}$.

- If $t = (x : U)U'$ then by induction hypothesis U and U' are well-typed in the system $\mathcal{T}$ and since the rule $< s, s', s'' >$ is a rule of the system $\mathcal{T}$, we have $\Gamma \vdash t : T$ in the system $\mathcal{T}$.

- If $t = (x\ c_1\ ...\ c_n)$ with x variable declared in Γ then x is well-typed in Γ in the system $\mathcal{T}$ and we prove by induction on i that the type of c_i is well-typed in the system $\mathcal{T}$ and c_i is well-typed in the system $\mathcal{T}$ and we conclude that $\Gamma \vdash t : T$ in the system $\mathcal{T}$.

- If t is a sort then by hypothesis $t = Prop$, it is well-typed in the system $\mathcal{T}$.

Proposition 24 Let Γ be a well-formed context in the system $\mathcal{T}$, $[\sigma_1; ...; \sigma_m]$ be a derivation of Γ and θ the substitution denoted by this derivation. Let x be an existential variable of Γ and $t = \theta x$. For every subterm of t which is a product $(x : U)U'$ if we let s be the type of U, s' be the type of U' and s'' be the type of $(x : U)U'$, we have $< s, s', s'' > \in R$ (the set of rules of the system $\mathcal{T}$) and for every subterm of t which is a sort s we have $s = Prop$ (and not $Type$),
Proof By induction on m.

Proposition 25 Let Γ be a context well-formed in the system $\mathcal{T}$, $[\sigma_1; ...; \sigma_m]$ a derivation of Γ and θ the substitution denoted by this derivation. The substitution θ is well-typed in Γ in the system $\mathcal{T}$.
Proof By induction on the length of Γ.

Theorem 1 Soundness

Let Γ be a (non constrained, non quantified) context and P a well-typed type in Γ. Let $\Gamma' = \Gamma[\exists x : P]$. If there exists a derivation of Γ' then there exists a proof t of P in Γ in $\mathcal{T}$.

Proof Let θ be the substitution denoted by the derivation of Γ'. The substitution θ is well-typed in Γ' in the system $\mathcal{T}$ and is a normal solution to Γ'. Let $t = \theta x$. Since θ is normal we have $\theta\Gamma' = \theta\Gamma$. We have $\theta\Gamma' = \theta\Gamma$, $\theta\Gamma = \Gamma$, $\theta P = P$ and $\theta\Gamma' \vdash \theta x : \theta P$ so $\Gamma \vdash t : P$.

The proof t is called the proof *denoted* by the derivation.

6.3 Completeness

Definition 33 The Relation $<$

Let $<$ be the smallest transitive relation defined on normal η-long terms such that

- if t_Γ is a strict subterm of t'_Δ then $t_\Gamma < t'_\Delta$,
- if T_Γ is the normal η-long form of the type of t_Γ in Γ and t is not a sort then $T_\Gamma < t_\Gamma$.

As proved in [8] [9], this relation is well-founded, which means that a function that recurses both on a strict subterm and on the type of its argument is total.

Definition 34 Size of a Term

Let Γ be a context and t a term well-typed in Γ. Let T be the normal η-long form of the type of t in Γ. We define by induction over $<$, the *size* of t_Γ ($|t_\Gamma|$)

- If t is a sort then $|t_\Gamma| = 1$.
- If t is a variable then $|t_\Gamma| = |T_\Gamma|$.
- If $t = (u\ v)$ then $|t_\Gamma| = |u_\Gamma| + |v_\Gamma| + |T_\Gamma|$.
- If $t = [x : U]u$ then $|t_\Gamma| = |u_{\Gamma[x:U]}|$.
- If $t = (x : U)V$ then $|t_\Gamma| = |U_\Gamma| + |V_{\Gamma[x:U]}| + |T_\Gamma|$.

Definition 35 Size of a Substitution

The *size* of a substitution $\theta = \{< x_i, \gamma_i, t_i >\}$ is the sum of the sizes of the t_i.

Proposition 26 If Γ is a failure context, then for every substitution σ, $\sigma\Gamma$ is also a failure context and therefore is not a success context.

Proof Let $a = b$ be a constraint of Γ relating two ground different terms, then $(\sigma a = \sigma b)$ is $(a = b)$ which is a constraint relating two well-typed ground different terms.

Lemma 2 Let Γ be a well-formed context and θ a normal solution to Γ, then there exists a derivation of Γ which denotes θ.

Proof By induction on the size of θ.

The context Γ is not a failure context because there exists a substitution θ such that $\theta\Gamma$ is a success context. If it is a success context then [ ] is a derivation of Γ. Otherwise let us chose an existential variable x with a normal η-long type $(x_1 : P_1)...(x_n : P_n)P$ such that P is atomic rigid. Let $t = \theta x$. Since θ is well-typed in Γ, $\theta\Gamma \vdash t : (x_1 : \theta P_1)...(x_n : \theta P_n)\theta P$. Since the term P is atomic rigid, the term θP is atomic. Then $t = [x_1 : \theta P_1]...[x_n : \theta P_n]u$ with u atomic or a product (η-long form).

- If $u = (w\ u_1\ ...\ u_p)$ then let q the number of products of the type of w and $r = p - q$. Since the type of u is atomic we have $p \geq q$, i.e. $r \geq 0$. We let

$$\sigma = \{< x, \gamma, [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n)) >\}$$

with γ defined in the algorithm. Then we build the terms to be substituted to the variables h_i $(1 \leq i \leq p)$ and H_i and K_i $(1 \leq i \leq r)$. We let

$$u'_i = [x_1 : P_1]...[x_n : P_n]u_i$$

And for all $1 \leq i \leq r$, $(w\ u_1\ ...\ u_{q+i-1})$ has a type which is a product, let $(y : U_i)V_i$ be this product. We let

$$U'_i = [x_1 : P_1]...[x_n : P_n]U_i$$
$$V'_i = [x_1 : P_1]...[x_n : P_n][y : U_i]V_i$$
$$\theta' = \theta - \{< x, [\,], \theta x >\} \cup \{< h_i, [\,], u'_i >\} \cup \{< H_i, [\,], U'_i >, < K_i, [\,], V'_i >\}$$

We have $\theta = \theta' \circ \sigma$. Let us prove that the substitution θ' is smaller than θ. We have

$$\theta x = [x_1 : \theta P_1]...[x_n : \theta P_n](w\ u_1\ ...\ u_p)$$

Let T_i be the type of $(w\ u_1\ ...\ u_i)$.
$|\theta x| = |(w\ u_1\ ...\ u_p)| = |T_0| + |u_1| + |T_1| + ... + |u_p| + |T_p|$
$> |T_q| + ... + |T_{p-1}| + |u_1| + ... + |u_p| = |(y : U_1)V_1| + ... + |(y : U_r)V_r| + |u_1| + ... + |u_p|$
$> |U_1| + ... + |U_r| + ... + |V_1| + ... + |V_r| + |u_1| + ... + |u_p|$
$= |U'_1| + ... + |U'_r| + |V'_1| + ... + |V'_r| + |u'_1| + ... + |u'_p|$.
So $|\theta| > |\theta'|$.

- If $u = (x : U)V$ then we let

$$\sigma = \{< x, \gamma, [x_1 : P_1]...[x_n : P_n](y : (h\ x_1\ ...\ x_n))(k\ x_1\ ...\ x_n\ y) >\}$$

with γ defined in the algorithm. Then we build the terms to be substituted to the variables h and k. We let

$$U' = [x_1 : P_1]...[x_n : P_n]U$$
$$V' = [x_1 : P_1]...[x_n : P_n][y : U]V$$
$$\theta' = \theta - \{< x, [\,], \theta x >\} \cup \{< h, [\,], U' >, < k, [\,], V' >\}$$

We have $\theta = \theta' \circ \sigma$. Let us prove that the substitution θ' is smaller than θ. We have

$$\theta x = [x_1 : \theta P_1]...[x_n : \theta P_n](y : U)V$$

$|\theta x| = |(y : U)V| > |U| + |V| = |U'| + |V'|$.
So $|\theta| > |\theta'|$.

In both cases, by induction hypothesis, there exists a derivation D of $\sigma\Gamma$ that denotes θ' and $[\sigma]D$ is a derivation of Γ that denotes θ.

Theorem 2 Completeness

Let Γ be a (non constrained, non quantified) context and P a type well-typed in Γ such that there exists a term t such that $\Gamma \vdash t : P$ then there exists a derivation of $\Gamma' = \Gamma[\exists x : P]$ which denotes the normal η-long form of the proof t.

Proof Let t' be the normal η-long form of t, and $\theta = \{< x, [\,], t' >\}$. The substitution θ is a normal solution to Γ', so there exists a derivation of Γ' that denotes θ.

7 Efficiency Improvements

To improve the efficiency of this method we have to recognize nodes that cannot lead to success nodes and prune the search tree.

7.1 Incremental Partial Checking of Constraints

First we must perform incremental partial checking of the constraints. So we define a function of simplification, which is very similar to the SIMPL function from [21] [22]. In order to define this function we have to modify a little the syntax of constraints: a constraint is now a triple $< \delta, a, b >$ where δ is a context with universal variables only. If $\Gamma = \Delta[< \delta, a, b >]\Delta'$ then the terms a and b must be well-typed and have the same type in $\Delta\delta$.

Definition 36 Simplification

We consider only constraints well-typed without the constraints. While there are rigid-rigid constraints in Γ and the simplification has not failed we iterate the process of replacing Γ by Γ'.

Let $\delta, a = b$ be a rigid-rigid constraint of Γ. $\Gamma = \Delta[\delta, a = b]\Delta'$. We define Γ' as

- if a and b are both abstractions $a = [x : T]a'$, $b = [x : U]b'$, since the terms a and b have the same type in $\Delta\delta$ we have $T = U$, we let $\Gamma' = \Delta[\delta[\forall x : T], a' = b']\Delta'$,
- if a and b are both products $a = (x : T)a'$, $b = (x : U)b'$ and T and U have the same type, we let $\Gamma' = \Delta[\delta, T = U][\delta[\forall x : T], a' = b']\Delta'$,
- if a and b are both atomic with the same head variable and they have the same number of arguments, $a = (w\ u_1\ ...\ u_p)$, $b = (w\ t_1\ ...\ t_p)$, we let $\Gamma' = \Delta[\delta, u_1 = t_1]...[\delta, u_p = t_p]\Delta'$,
- in the other cases the simplification fails.

If the simplification fails then the branch is hopeless, it can be pruned.

7.2 Using the Flexible-Rigid Constraints

When we have a well-typed flexible-rigid constraint, and we can solve the head variable of the flexible term (i.e. when the head of its type is rigid) we must solve it first, the only candidates for the head variable in an elementary substitution are the bound variables and the head of the rigid term [21] [22]. The other variables lead to unsolvable rigid-rigid constraints.

7.3 Using the Flexible-Flexible Constraints

When we have a well-typed flexible-flexible constraint and we can solve one of the head variables, we must solve it first, the constraint does not help to restrict the number of substitutions, but it may become a flexible-rigid constraint the next step. Moreover solving constraints helps to unfreeze types of existential variables and constraints that are not well-typed without the constraints.

7.4 Avoiding Splitting

The term $(Q\ (h_1\ x_1\ ...\ x_n)...(h_q\ x_1\ ...\ x_n))$ in the definition of the method is atomic, if it is rigid then all the solutions with r-splitting $r \neq 0$ lead to hopless constraints and therefore they can be avoided. In particular in the $\lambda\Pi$-calculus, we never consider existential variables for types or predicates and so splitting can always be avoided.

7.5 Solving some Decidable Unification Problems

When we have a constraint well-typed without the constraints which is either a first order unification problem [29], an argument-restricted unification problem [26] [27], a second order matching problem [22] [24] [8] [9], a second-order-argument-restricted matching problem [8] [9] then we can solve it using always terminating algorithms and apply the substitutions obtained in this way. In a system with such heuristics, the elimination of a first-order universal quantifier is a one-step operation and the elimination of a higher-order universal quantifier is a more complex operation that needs several steps and can lead to backtracking.

7.6 Priority to the Rightmost Variable

When we have two existential variables x and y such that y is declared on the right of x and x has an occurrence in the type of y, if both variables may be instantiated then we have to begin with the rightmost. For instance if we have an axiom $u : (P\ n)$ and we search an integer x such that $(P\ x)$. We have two existential variables: $x : Nat$ and $y : (P\ x)$. If we instantiate y by u then x will be automatically instantiated by n. But if we begin by instantiating x, we will instantiate x by 0, 1, 2, etc. and fail to prove $(P\ 0)$, $(P\ 1)$, $(P\ 2)$, etc. before we reach $x = n$.

7.7 Normalizing some ill-typed terms

An important improvement of the method would be to recognize some normalizable ill-typed terms and then be able to use the constraints more efficiently. For instance it is possible to prove that if t is a normal η-long term and σ an elementary substitution $x \leftarrow [x_1 : P_1]...[x_n : P_n]u$ where u is a product or an atomic term with a head variable which is a universal variable or a sort (but not one of the x_i), then σt is normalizable.

8 Extension to The Calculus of Constructions With Universes

Definition 37 The Calculus of Constructions With Universes and Without Cumulativity

The *Calculus of Constructions With Universes and Without Cumulativity* is a pure type system which has an infinite number of sorts : $Prop, Type_0(= Type), Type_1, Type_2, Type_3, ...$, the axioms $Prop : Type_0$ and $Type_i : Type_{i+1}$ and the rules

$$< Prop, Prop, Prop >, < Type_i, Prop, Prop >,$$

$$< Prop, Type_i, Type_i >, < Type_i, Type_j, Type_{max\{i,j\}} >$$

The strong normalization and confluence of $\beta\eta$-reduction for this system are not yet fully proved. But since all the terms typable in this system are typable in the Calculus of Constructions with

Universes [3] [25], the β-reduction is strongly normalizable and confluent [3] [25]. It is conjectured in [12] that in a pure type system in which the β-reduction is strongly normalizable the $\beta\eta$-reduction also is. Then assuming strong normalization, confluence is proved in [12] [30] [4].

In an extension of the method presented here to the Calculus of Constructions With Universes and Without Cumulativity we do not need a $Meta$ type system any more because all the terms we consider in the algorithm in this system are well-typed in this system. More generaly the forumulation of this method for an arbitrary normalizable pure type system $\mathcal{T}$ seems to requires only the definition of a $Meta$ type system for $\mathcal{T}$.

In a system which has a large number of sorts, there may be a large number of substitutions of the form $\sigma = x \leftarrow \gamma, t$ with $t = [x_1 : P_1]...[x_n : P_n]u$ where u is a sort or a product. For instance, in the Calculus of Constructions With Universes and Without Cumulativity, we can instantiate a variable $x : Prop$ by an infinite number of terms $(y : h)(k\ y)$ with $h : Type(i)$ and $k : (y : h)Prop$. In order to avoid more inefficiencies it seems possible to use sort variables and constraints on these variables as in the method of floating universes [17] [23].

9 An Incomplete but more Efficient Method

In the example in section 5.2, in order to get the proof $(u\ (A \to B)\ v\ w) : B$ we have considered the elementary substitution with 1-splitting

$$p \leftarrow (u\ h_1\ h_2\ h_3)$$

with three applications although the type of u begins by only two products.

Because we have to consider all these possible degrees of splitting, the method is quite inefficient, we get a much more efficient one if we restrict the rule to elementary substitutions with 0-splitting and a number of abstractions ranging from 0 to the number of products of the type of the existential variable we substitute. Let us call this algorithm *algorithm with weak splitting* (by opposition the previous one can be called *algorithm with strong splitting*). The search tree of the weak splitting algorithm is finitely branching.

This algorithm with weak splitting cannot synthesize a proof of B, but it can synthesize a proof of $A \to B$. Indeed with the variable $p' : A \to B$ we perform the substitution

$$p' \leftarrow (u\ h_1\ h_2)$$

with the types $h_1 : Prop$, $h_2 : (I\ h_1)$ (remark that we have no abstraction although the type of p' begins by a product). The constraint $h_1 = A \to B$ suggests the substitution

$$h_1 \leftarrow (A \to B)$$

then we have an existential variable $h_2 : (I\ (A \to B))$ we perform the substitution

$$h_2 \leftarrow v$$

The synthesized proof is $(u\ (A \to B)\ v) : (A \to B)$. Remark that this term is not in η-long form, this form is $[w : A](u\ (A \to B)\ v\ w) : (A \to B)$.

In both proofs considered above (of B and $A \to B$) the problem is to remark that the variable u of type $(P : Prop)((I\ P) \to P)$ must be applied to the proposition $(A \to B)$. With the strong

splitting algorithm this is found in both cases, but the price to pay is inefficiency, with the weak splitting this is found when the type of the existential variable is $A \to B$ but not when it is B. Roughly speaking, this means that when we must use a proposition with a quantifier on a proposition or a predicate (for instance an induction axiom) the weak splitting algorithm finds the proposition to be used only when it *can be seen* in the type of the existential variable, and not when the proof requires an *induction loading* [18] (as here from B to $A \to B$).

This weak splitting algorithm is quite similar to the introduction-resolution method [18] for some unification algorithm. Actually some proofs that cannot be synthesized by introduction-resolution can be synthesized with weak splitting. For instance in a context that contains the universal variables $u : (P : Prop)(P \to A)$ and $v : (B \to C)$ the proof $(u\ (B \to C)\ [x : B](v\ x)) : A$ is synthesized by the weak-splitting algorithm and not by the introduction-resolution method. To get exactly the introduction-resolution method we would have to forbid completely product substitutions.

This method with weak splitting is incomplete but its transitive closure is complete [9]. This means that even if it cannot be synthesized, each proof can be broken up in smaller proofs that can be synthesized.

10 Unification

10.1 Ground Unification

The unification problem is to decide if given a context Γ (with universal and existential variables but no constraints) and two terms a and b well-typed in Γ with the same type, there exists a substitution θ well-typed in Γ, such that for every variable x bound by θ, the context associated to x by θ contains no constraints and such that $\theta a = \theta b$. In first order unification and higher order unification we do not require θ to fill all the existential variables of Γ, moreover the substitution θ may introduce new existential variables. This tolerance is due to the fact that in first order and higher order logic, types are all supposed to be inhabited, so from any unifier we can deduce a ground unifier by filling the unfilled variables with arbitrary terms. So unifiability is equivalent to ground unifiability. This is not the case any more in type systems where we have empty types. In unification in simply typed λ-calculus under a quantified context [26] types may also be empty and unifiability is not equivalent to ground unifiability. The notion of unifiability considered in [26] is the one of ground unifiability.

A semi-algorithm for deciding ground unifiability is to apply the method developed in this paper to the context $\Gamma[a = b]$. Moreover this algorithm enumerates all the ground unifiers.

In contrast with what happens in simply typed λ-calculus, we have to solve flexible-flexible equations. Indeed these equations may have no solution since the head variables of the terms may have empty types. Moreover it is proved in [26] that the problem of deciding the existence of solutions for flexible-flexible unification problems under a quantified context is undecidable in simply typed λ-calculus. This proof generalizes easily to type systems.

10.2 Toward Open Unification

The utility of an open unification algorithm is not obvious, since in a proof-search algorithm (or more generally in an algorithm that uses unification) existential variables are introduced to be

filled-in and not to remain forever.

If we still need such an algorithm we may remark that flexible-flexible equations have to be solved since, in contrast with what happens in simply typed λ-calculus, they may have no solution. Consider

$$[\forall A : Prop; \forall B : Prop; \forall u : A; \exists x : (P : Prop)P]$$

$$a = (x\ (A \to B)\ u) \qquad b = (x\ B)$$

The equation $a = b$ is flexible-flexible, but there is no solution to this problem. Indeed, let us suppose there is one and consider $t = \theta x$. We have $t : (P : Prop)P$ so $t = [P : Prop]t'$ with $t' : P$. So t' is neither an abstraction nor a product. It is an atomic term $(f\ c_1\ ...\ c_n)$. If f were the variable P we would have $n = 0$ and thus $t' = P$ which is impossible for type reasons, so $f \neq P$. Since $(t'[P \leftarrow (A \to B)]\ u) = t'[P \leftarrow B]$ we have

$$(f\ c_1[P \leftarrow (A \to B)]\ ...\ c_n[P \leftarrow (A \to B)]\ u) = (f\ c_1[P \leftarrow B]\ ...\ c_n[P \leftarrow B])$$

So $(f\ c_1[P \leftarrow (A \to B)]) = f$ which is impossible since a variable cannot be equal to an application.

Searching for solutions of flexible-flexible equations seems to be rather difficult, since the head variable of an elementary substitution may now be any variable of Γ and also any new variable. We would have to design a proof search algorithm similar to the one described in this paper but that enumerates all the proofs of a proposition, under all the possible sets of axioms.

It may be possible to recognize flexible-flexible equations that have solutions and those that do not have any and keep the former unsolved until the end of the process as we do in unification in simply typed λ-calculus. In this case the normalization of terms well-typed using such constraints is not obvious, we may have to perform one normalization step as an elementary operation of the algorithm. Also some product substitutions have to be performed when the types of the variables to be instantiated in flexible-rigid and flexible-flexible equations has an existential head variable.

In contrast with ground unifiability, open unifiability seems a quite difficult problem with small interest.

Conclusion

In this paper we have described an semi-decision procedure for type systems. In fact, since proof checking is decidable in these systems, the existence of such a procedure is obvious: we just need to enumerate all the lists of characters until we get a proof of the proposition to be proved. Of course this method (which is sometime used to prove the semi-decidability of predicate calculus) is of no practical interest. We can let it be more realist in enumerating not all the lists of characters but only the normal λ-terms (or proof-trees), this method is comparable to the methods of the early sixties based on the search of a counter Herbrand model.

Resolution excludes more hopeless tentatives by remarking that in a premise $\forall x.(P\ x)$, we need to instantiate x by t only if $(P\ t)$ is an instance of the proposition to be proved or an hypothesis of another premise. This method formulated by Robinson [29] for first order logic and by Huet [20] for higher order logic is generalized here to type systems. The notions of proof-term that appear in these systems simplify the method and makes more explicit the idea of blind enumeration of proof-terms regulated by a failure anticipation mechanism exploiting type constraints.

Acknowledgements

The author thanks Gérard Huet who has supervised this work and Amy Felty, Herman Geuvers, Christine Paulin and the anonymous referees for many helpful comments and criticisms.

References

[1] H. Barendregt, Introduction to Generalized Type Systems, *Journal of Functional Programming*, 1, 2, 1991, pp. 125-154.

[2] Th. Coquand, Une Théorie des Constructions, *Thèse de troisième cycle*, Université Paris VII, 1985.

[3] Th. Coquand, An analysis of Girard's paradox, *Proceedings of Logic in Computer Science*, 1986, pp. 227-236.

[4] Th. Coquand, An Algorithm for Testing Conversion in Type Theory, *Logical Frameworks I*, G. Huet and G. Plotkin (Eds.), Cambridge University Press, 1991.

[5] T. Coquand, G. Huet, The Calculus of Constructions, *Information and Computation*, 76, 1988, pp. 95-120.

[6] N.G. de Bruijn, Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem, *Indagationes Mathematicae*, 34, 5, 1972, pp. 381-392.

[7] N.G. de Bruijn, A Survey of the Project Automath, *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.R. Hindley, J.P. Seldin (Eds.), Academic Press, 1980.

[8] G. Dowek, A Second Order Pattern Matching Algorithm in the Cube of Typed λ-Calculi, *Proceedings of Mathematical Foundation of Computer Science*, Lecture Notes in Computer Science, 520, 1991, pp. 151-160. Rapport de Recherche 1585, INRIA, 1992.

[9] G. Dowek, *Démonstration Automatique dans le Calcul des Constructions*, *Thèse*, Université de Paris VII, 1991.

[10] C. M. Elliott, Higher-order Unification with Dependent Function Types, *Proceedings of the 3^{rd} International Conference on Rewriting Techniques and Applications*, N. Dershowitz (Ed.), Lecture Notes in Computer Science, 355, Springer-Verlag, 1989, pp. 121-136.

[11] C. M. Elliott, Extensions and Applications of Higher-order Unification, *PhD Thesis*, Carnegie Mellon University, Pittsburgh, 1990.

[12] H. Geuvers, The Church-Rosser Property for $\beta\eta$-reduction in Typed Lambda Calculi, *Proceedings of Logic in Computer Science*, 1992.

[13] H. Geuvers, M.J. Nederhof, A Modular Proof of Strong Normalization for the Calculus of Constructions, *Catholic University Nijmegen*, 1990.

[14] J. Gallier, On Girard's Candidats de Réductibilité, *Logic and Computer Science*, P. Odifreddi (Ed.), Academic Press, London, 1990, pp. 123-203.

[15] J.Y. Girard, Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur, *Thèse de Doctorat d'État*, Université de Paris VII, 1972.

[16] R. Harper, F. Honsell, G. Plotkin, A Framework for Defining Logics, *Proceedings of Logic in Computer Science*, 1987, pp. 194-204.

[17] R. Harper, R. Pollack, Type Checking, Universe Polymorphism, and Typical Ambiguity in the Calculus of Constructions, *CCIPL, TAPSOFT'89*, Barcelona, 1989.

[18] L. Helmink, Resolution and Type Theory, *Proceedings of the ESOP Conference*, Copenhagen, Lecture Notes in Computer Science, 432, Springer-Verlag, 1990.

[19] W.A. Howard, The Formulæ-as-type Notion of Construction, 1969, *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.R. Hindley, J.P. Seldin (Eds.), Academic Press, 1980.

[20] G. Huet, Constrained Resolution A Complete Method for Higher Order Logic, *PhD Thesis*, Case Western Reserve University, 1972.

[21] G. Huet, A Unification Algorithm for Typed λ-calculus, *Theoretical Computer Science*, 1, 1975, pp. 27-57.

[22] G. Huet, Résolution d'Équations dans les Langages d'Ordre 1,2, ..., ω, *Thèse de Doctorat d'État*, Université de Paris VII, 1976.

[23] G. Huet, Adding Type:Type to the Calculus of Constructions, Unpublished Note, 1988.

[24] G. Huet, B. Lang, Proving and Applying Program Transformations Expressed with Second Order Patterns, *Acta Informatica*, 11, 1978, pp. 31-55.

[25] Z. Luo, An Extended Calculus of Constructions, *PhD Thesis*, University of Edinburgh, 1990.

[26] D. A. Miller, Unification Under a Mixed Prefix, To appear in *Journal of Symbolic Computation*.

[27] F. Pfenning, Unification and anti-Unification in the Calculus of Constructions, *Proceedings of Logic in Computer Science*, 1991, pp. 74-85.

[28] D. Pym, Proof, Search and Computation in General Logic, *PhD thesis*, University of Edinburgh, 1990.

[29] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the Association for Computing Machinery*, 12, 1, 1965, pp. 23-41.

[30] A. Salvesen, The Church-Rosser Theorem for Pure Type Systems with $\beta\eta$-reduction, Manuscript, University of Edinburgh, 1991.