Proof Representation in Type Theory: State of the Art

César Muñoz
INRIA Rocquencourt
B.P. 105
78153 Le Chesnay Cedex, France
E-mail: Cesar.Munoz@inria.fr
Tel: (33) 1 39 63 51 57, Fax: (33) 1 39 63 53 30

February 13, 1996

Abstract

In the frame of intuitionistic logic and type theory, it is well known that there is an isomorphism between types and propositions; the Curry-Howard Isomorphism. However, it is less clear the relation between terms construction and proofs development. The main difficulty arises when we try to represent incomplete proofs as terms describing a state of knowledge where some part of the proof is built, but another part remains to be built. The pieces of proof terms that are unknown are called places-holders. We present a theoretical approach to place-holders in type theory. In this approach place-holders are represented by metavariables and terms are built incrementally by instantiation of metavariables. We show how an appropriate extension to typed λ-calculus with explicit substitutions and explicit typing of metavariables allows to identify terms construction and proofs development activities.

Representación de pruebas en la teoría de tipos: Estado del arte

Resumen

En el marco de la lógica intuicionista y la teoría de tipos es bien conocido que las nociones de tipo y proposición coinciden (este hecho se conoce como Isomorfismo de Curry-Howard). Es menos claro, sin embargo, la relación entre los procesos de construcción de términos y desarrollo de pruebas. La principal dificultad consiste en la representación de pruebas incompletas por medio de términos no totalmente construidos. Este artículo presenta un enfoque teórico de términos incompletos en la teoría de tipos. En este enfoque, las metavariables permiten expresar términos parcialmente construidos, y la instanciación de metavariables es el mecanismo para construir términos incrementalmente. Tal enfoque, aplicado a un λ-cálculo tipado extendido con substituciones explícitas y con reglas explícitas de tipado de metavariables, identifica la actividades de construcción de términos y desarrollo de pruebas.

Preliminaries

We recall a few concepts about intuitionistic logic, typed λ-calculus and the Curry-Howard isomorphism. The reader familiar with these concepts can skip this section. For a more detailed explanation, refer to [Kle52, Fit69, Bar84, GTL89].

First, we consider an intuitionistic minimal logic in which propositional formulas are built from atomic propositions and the imply connector ($\rightarrow$). We use uppercase Latin letters $A, B, \ldots$ to denote formulas and uppercase Greek letters Γ, Δ to denote sets of formulas. We write Γ, A for $\Gamma \cup \{A\}$. As usual, we use parentheses to indicate sub-term structure.

A *judgement* is a pair formed by a set of formulas Γ and a formula A, noted by $\Gamma \vdash_I A$. Informally, a judgement $\Gamma \vdash_I A$ can be read as "A is a logical consequence of the hypotheses Γ". A judgement is *provable* if and only if it is derived by top-down application of the following rules:

$$\frac{}{\Gamma, A \vdash_I A}\,(\text{Axiom}) \qquad \frac{\Gamma, A \vdash_I B}{\Gamma \vdash_I A \rightarrow B}\,(\text{Intro}^{\rightarrow}) \qquad \frac{\Gamma \vdash_I A \rightarrow B \quad \Gamma \vdash_I A}{\Gamma \vdash_I B}\,(\text{Elim}^{\rightarrow})$$

A formula A is a *tautology* if and only if the judgement $\vdash_I A$ is provable. For example, the formula $(A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$ is a tautology since it can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \rightarrow B, B \rightarrow C, A \vdash_I A \rightarrow B}\,(\text{Axiom}) \quad \dfrac{}{A \rightarrow B, B \rightarrow C, A \vdash_I A}\,(\text{Axiom})}{A \rightarrow B, B \rightarrow C, A \vdash_I B}\,(\text{Elim}^{\rightarrow}) \quad \dfrac{}{A \rightarrow B, B \rightarrow C, A \vdash_I B \rightarrow C}\,(\text{Axiom})}{A \rightarrow B, B \rightarrow C, A \vdash_I C}\,(\text{Elim}^{\rightarrow})}{A \rightarrow B, B \rightarrow C \vdash_I A \rightarrow C}\,(\text{Intro}^{\rightarrow})}{A \rightarrow B \vdash_I (B \rightarrow C) \rightarrow (A \rightarrow C)}\,(\text{Intro}^{\rightarrow})}{\vdash_I (A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))}\,(\text{Intro}^{\rightarrow})$$

Now, we consider a simply typed λ-calculus that contains a set of basic types and its functional closure, i.e. A is a type if and only if (1) it is a basic type or (2) it has the form $B \rightarrow C$ where B and C are types. In the following, we are going to use the uppercase Latin letters $A, B, \ldots$ to range over types.

Let $\mathcal{V}$ be an infinite set of variables $x, y, \ldots$. The well formed terms of the typed λ-calculus are defined as the smallest set Λ that satisfy: (1) $\mathcal{V} \subseteq \Lambda$, and (2) if M and N are in Λ and A is a type, then both of $\lambda x : A.M$ and $(M\ N)$ are in Λ. A term like $\lambda x : A.M$ is called *abstraction* (informally, it represents a function with a parameter x of type A and the body M) and a term like $(M\ N)$ is called *application*.

The *free variables* set of a λ-term M is denoted $F_v(M)$ and defined inductively over M by $F_v(x) = \{x\}$, $F_v(M\ N) = F_v(M) \cup F_v(N)$ and $F_v(\lambda x : A.M) = F_v(M) - \{x\}$.

In a term $\lambda x : A.M$, all the free occurrences of the variable x in M are *bound* by the outermost λ-constructor. As usual, the names of bound variables are not important, and they can be renamed in order to obtain equivalent terms. In λ-calculus this is expressed by the *α-conversion*: $\lambda x.M =_\alpha \lambda y.(M[x \leftarrow y])$ where y is a fresh variable. The general form $M[x \leftarrow N]$ does not belong to the rules of well formed λ-terms. This is because formally $M[x \leftarrow N]$ is not a λ-term, but a notation to represent "the λ-term in which all the free occurrences of the variable x in the term M have been replaced by N". This form is called *substitution* and it can be used to express the principal rule of the λ-calculus, the *β-reduction*: $(\lambda x.M)N \xrightarrow{\beta} M[x \leftarrow N]$.

A *context* is a set of variable declarations that assign a type to each variable. We use uppercase Greek letters Γ, Δ to denote contexts.

A *typing judgement* is an expression $\Gamma \vdash M : A$ where M is a λ-term, A is a type and Γ is a context. It can be read as "M is a term of type A in Γ".

A judgement is *valid* if and only if it can be derived by top-down application of the following rules:

$$\frac{}{\Gamma, x : A \vdash x : A}\,(\text{Var}) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : A \rightarrow B}\,(\text{Abs}) \qquad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B}\,(\text{Appl})$$

We say that a λ-term M is *well typed* if and only if there exists a type A such that $\vdash M : A$, and we say that a type A is *inhabited* if and only if there exists a λ-term M such that $\vdash M : A$.

If we identify propositions (of the intuitionistic minimal logic) with types (of the typed of λ-calculus), then we can realize that the derivation rules (Axiom), (Intro$^{\rightarrow}$) and (Elim$^{\rightarrow}$) correspond one to one to the typing rules (Var), (Abs) and (Appl). Indeed, typing rules are logical rules decorated by typed λ-terms. This is essentially the *Curry-Howard isomorphism.* The idea to represent theorem proofs by objects was originally developed by Heyting and Kolmogorov in the 1930's. The Heyting and Kolmogorov's semantic proposes to model proofs as functional objects. For instance, a proof of $A \rightarrow B$ is a function f, which maps each proof p of A to a proof $f(p)$ of B. Curry and Howard's works regain the original idea but replace functional objects by formal terms.

For instance, if we decorate the above tree derivation of the tautology

$$(A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$$

then we obtain the valid typing judgement

$$\vdash \lambda x : A \rightarrow B.\lambda y : B \rightarrow C.\lambda z : A.(y\ (x\ z)) : (A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C))$$

In this frame, we have the following nice properties:

1. The set of tautologies formulas is exactly the set of inhabited types, i.e. $\vdash_I A$ if and only if there exists a λ-term M such that $\vdash M : A$. In the following we are going to confuse the words *types* and *propositions*, and if $\vdash M : A$ is a valid judgement, then we say that M is a proof object of A.
2. The β-reduction is *strongly normalising* on well typed λ-terms.
3. The β-reduction is *confluent* on well typed λ-terms.
4. Typing rules and β-reduction commute, i.e. if $\Gamma \vdash M : A$ and $M \xrightarrow{\beta^*} M'$ then $\Gamma \vdash M' : A$.
5. A term M in β-normal of type A, is a cut free proof of A, i.e. a proof that does not use lemmas.

The Curry-Howard isomorphism is extended to intuitionistic first order and higher-order logics and it is widely studied in proof theory. It is at the base of mathematics formalisation where proofs are just mathematical objects. Such languages are the fundament of automatic system for proof construction, program verification and program synthesis.

1 Introduction

It is clear that in an intuitionistic logic we can identify the concepts of *types* and *propositions*. However, it is less clear the relationship between terms construction and proofs development.

Generally, proofs are developed in a bottom-up way. For example, if we want to prove a proposition of the form $A \rightarrow B$, then we use the rule (Intro$^{\rightarrow}$) and then we try to prove B under the assumption A. In contrast, terms construction is essentially a top-down process. For example, if we know that M is a λ-term of type B when we assume that x is of type A, using the rule (Appl) we obtain the term $\lambda x : A.M$ of type $A \rightarrow B$.

What happens if we build λ-terms in a bottom-up way? For instance, assume that we want to prove $A \rightarrow B$ in an empty context and we do a bottom-up application of the rule (Var), then we would say that the term $(\lambda x : A.\Box)$ is an incomplete proof of $A \rightarrow B$, where $\Box$ represents a term of type B under the assumption that $x : A$ is declared in the context. The symbol $\Box$ is usually called a *place-holder* and the term $(\lambda x : A.\Box)$ is an *incomplete object* proof, since it represents a hypothetical future proof. In this way, it is possible to build a proof incrementally.

But, what is the status of a term like $(\lambda x : A.\Box)$ in λ-calculus? How can we represent incomplete proofs in the kernel of a proof assistant system based on type theory? In certain proof systems, proof terms and incomplete proofs are represented in different ways. For example, in the system Coq ([CCF+95]) proof terms

are represented in the Calculus of Inductive Construction (an extension of the simply typed λ-calculus) and incomplete proofs are represented in an *ad hoc* structure called *tactic trees*. At the end of the proving process, if a tactic tree represents a complete proof, it is translated to a proof term. An advantage of this approach is the flexibility that the user has to program his own tactics. In general, these *ad hoc* structures can be as complex as the system programmer wants.

Another point of view, taken for example in the system Alf ([AGNvS94]), is that incomplete proof objects are in some sense "incomplete proof terms", and then incomplete proof objects must be represented as λ-terms also, i.e. in the above example $(\lambda x : A.\Box)$ must be considered like a λ-term. An advantage of this approach is that a unique proof representation is simpler and clearer; specially if we want to prove the correction of the system because there is only one data structure to verify.

In this paper, we consider the second possibility, i.e. the representation of partial proofs in λ-calculus. In particular, we want to answer the question about the status of a term like $\Box$ inside the term $(\lambda x : A.\Box)$. *A priori*, a place-holder can be seen as a lemma (variable) that we assume during term construction, but that must be proved (instantiated) in order to have a complete proof. But is it sufficient to consider a classical typed λ-calculus to represent place-holders and to explain their refinement mechanism? We detail all of these considerations in Section 2. In Section 3 we introduce a new sort of variables that are called Metavariables. Incomplete proof terms are just terms with metavariables, and term refinement can be implemented by instantiation of metavariables. In Section 4 we discuss the metavariable approach for first order and higher-order type theories. In Section 5 we compare our work with related works. Finally, the last section provides the conclusion of this work and our future research direction.

2 Place-holders, Variables and Explicit Substitutions

In the classical λ-calculus there is a notion of variable and a mechanism to substitute it, then, why not to use them to represent the place-holders and their refinement mechanism? We are going to study this possibility with some examples.

Assume that we have an hypothesis h of the proposition A and we want to find a proof term of $B \to A$. We propose the incomplete proof $(\lambda x : B.\Box)$. If we substitute $\Box$ for h we obtain the term $(\lambda x : B.h)$ which does not contain the place-holder.

Now, assume that we want to prove $A \to A$. Again, we propose the incomplete proof $(\lambda x : A.\Box)$. We would like to substitute $\Box$ for x in order to obtain something like $(\lambda x : A.x)$, but x is a bound variable in $(\lambda x : A.\Box)$, and then it is not possible to do this substitution. We recall that the substitution operation in λ-calculus takes care of renaming bound variables.

One solution to the above problem, taken from the Higher-Order Unification tradition ([Hue75],[Ell89]), is the functional handle of scope. With this technique, the information that a variable can indeed occur in the substitution needs to be functionally handled by the variable $\Box$. Thus we do not consider the term $(\lambda x : A.\Box)$, but the term $(\lambda x : A.(\Box\ x))$ where $\Box$ is of type $A \to A$. Now, to complete the proof we can substitute $\Box$ for "any term of type $\overline{A \to A}$". But we are again in the initial state: we must find a term of type $A \to A$. Therefore, the only possible refinement is a complete proof term!

If we use the standard λ-calculus to represent incomplete proofs, and its substitution mechanism to refine $\Box$-terms, then we cannot build a proof incrementally. In particular, it is not possible to simulate the `Intro` tactic that exists in many proof assistant systems (the `Intro` tactic corresponds to the application bottom-up of the typing rule (Abs)).

Actually, to build a proof of $A \to A$ from the incomplete object $(\lambda x : A.\Box)$, we do not want to substitute the term $\Box$ for x, but "refine" the place-holder $\Box$ with the term x. Then we can imagine a λ-calculus with another operation that we call *instantiation*. Instantiation replaces a variable by a term without renaming bound variables. A serious problem is that instantiation and β-reduction do not commute. For instance the term $((\lambda x : A.\Box)y)\{\Box \mapsto x\}$ (where $\{\Box \mapsto x\}$ is a notation for the instantiation of $\Box$ with x) is β-equivalent to $\Box\{\Box \mapsto x\}$ and is equivalent modulo instantiation to x, but $((\lambda x : A.\Box)y)\{\Box \mapsto x\}$ is also equivalent modulo instantiation to $((\lambda x : A.x)y)$ and this term is β-reduced to y! The problem is that we cannot apply

β-reduction under incomplete object proofs. But, how can we explain this restriction in λ-calculus if the β-reduction is expressed by means of an external substitution operation that cannot be delayed?

Explicit substitutions calculi support a mechanism of lazy evaluation for substitutions. In an explicit substitutions calculus the β-reduction is implemented by means of an internal substitution operator. There are many different presentations for these systems ([ACCL91], [CHL95], [Les94], [Río93], [Blo95], [KR95], [Muñ95], [Ros92]...), each one with different meta-theoretical properties. For the discussion that follows we consider a very general explicit substitutions calculus with names, without any special meta-theoretical property. However, in order to prove certain typing properties like decidability or uniqueness, it is necessary to provide an explicit substitutions calculus with meta-theoretical properties like confluence or weak normalisation on well typed terms.

We write $M[x := N]$ the explicit substitution of the variable x by the term N to the term M. In such a calculus the β-redexes are reduced by the rule (Beta) $(\lambda x : A.M)N \rightarrow M[x := N]$. Here, $M[x := N]$ is not a notation for an external substitution operation, but a real and well formed syntactic term. Thus, in an explicit substitution calculus it is necessary to have an explicit substitutions reduction system (usually called δ) that contains rules like (Var_1) $x[x := N] \rightarrow N$ and (Var_2) $y[x := N] \rightarrow N$ if $x \neq y$.

However, an explicit substitutions calculus is not sufficient to express incremental construction of proofs by means of refinement of incomplete proofs. It is necessary to provide a mechanism to delay the effective application of a substitution to a place-holder, and thus for example the term $\Box[x := N]$ must not be reduced by (Var_1), nor by (Var_2) and it is if $\Box$ is a variable.

3 Place-holders as Metavariables

Actually, variables and place-holders play very different roles in proof objects. The former denote calculus objects of the theory, for example the parameters in a λ-abstraction. The latter represent unknown pieces of proof which will be found in order to build a complete proof object.

If we consider the λ-calculus defined as an algebra on a set $\mathcal{X}$ of variables and a set of operators containing a set of constants $\mathcal{V}$ and a set of unary abstractors indexed on $\mathcal{V}$ and the application, then we have two sort of variables:

- Those of $\mathcal{V}$ (noted by the lowercase letters $x, y, \ldots$) that correspond to the variables of the calculus, for example x in the term $\lambda x : A.x$, and
- Those of $\mathcal{X}$ (noted by the uppercase letters $X, Y, \ldots$) that correspond to arbitrary terms of the algebra and are usually called *Metavariables*

Proof terms are built incrementally by refinement steps. Refinement is no more than instantiation of metavariables. In order to have a consistent system, we demand the following invariant property over refinements:

Definition 1 *A refinement operation commutes with typing if and only if for any terms M and M', context Γ and type A, if M is refined into M' and $\Gamma \vdash M : A$ then $\Gamma \vdash M' : A$*

Unfortunately, instantiation of metavariables and typing do not commute. For example, let Γ be the context declaration $x : A, z : (B \rightarrow A) \rightarrow C$ of the incomplete term $(z\ (\lambda x : B.Y))$ of type C, where Y is a metavariable of type A. If we instantiate Y with the variable x of type A declared in Γ, then we obtain an ill-typed term since $(\lambda x : B.x)$ has type $B \rightarrow B$ and not $B \rightarrow A$. However, if we consider that the context declaration of Y is $\Gamma, x : B$, then we cannot instantiate Y with x because they have different types.

Therefore, we can say that the context declaration of a metavariable is the context where it is used, but what happens when we use the same metavariable in different contexts? A possible answer, given by [DHK95] and also implemented in [Mag95], is to associate to each metavariable X a *unique* type A_X, a *unique* context Γ_X, and the implicit typing rule:

$$\frac{}{\Gamma_X \vdash X : A_X}\,(\text{Meta}_X)$$

Thus, all the occurrences of a metavariable must be typed in the same context, with the same type, and they are uniquely determined by their position in the term. For instance, the judgement $w : A \vdash ((\lambda x : A \to B \to C.x\ Y)Z) : C$ is valid if and only if $T_Y = A$, $\Gamma_Y = w : A, x : A \to B \to C$, $T_Z = B$ and $\Gamma_Z = w : A$. Therefore, we have the following typing rules:

$$\frac{}{w : A, x : (A \to B \to C) \vdash Y : A}\,(\text{Meta}_Y) \qquad \frac{}{w : A \vdash Z : B}\,(\text{Meta}_Z)$$

In contrast, the term $(\lambda x : A.X)X$ is not well typed in any context, since if we assume that this term is well typed in a context Γ, then X would be typed in two different contexts: Γ and $\Gamma, x : A$.

Due to the fact that all the occurrences of a metavariable X refer to the same metavariable X, the refinement operation is a global operation simply defined as:

Definition 2 *Let X be a metavariable, T a term such that $\Gamma_X \vdash T : T_x$ and M a term possibly containing X. The refinement of X with T transforms the term M into $M\{X \mapsto T\}$.*

An additional point is that the context of a metavariable is not invariant under β-reductions of λ-calculus. For instance, if we take the term $(\lambda x : A.(\lambda z : A.x))Y$ in a context Γ, where Y is a metavariable of type A, and we do a step of β-reduction, we get the term $(\lambda z : A.Y)$ where the metavariable Y is used in a context $\Gamma, z : A$. One possible solution, implemented in [Mag95], is to avoid the introduction of a substitutions inside an λ-abstraction and thus the term $(\lambda x : A.(\lambda z : A.x))Y$ reduces to $(\lambda z : A.x)[x := Y]$ which is a normal form.

Another solution is to use an explicit substitutions calculus with a De Bruijn's index notation of variables, for example the system λ_σ ([ACCL91]). In the de Bruijn's index notation, the name of bound variables are dropped (because they are irrelevant) and each occurrence of a variable in a term is replaced by a natural number (its de Bruijn's index). The index $\underline{n}$ is captured by the n-th λ-constructor surrounding it. For instance, the term $(\lambda x : B \to A.(\lambda y : B.x\ y))$ becomes $(\lambda B \to A.(\lambda B.\underline{2}\ \underline{1}))$. The renaming of bound variables is handled by the *shift* substitution ($\uparrow$) which increments by one all the indices. For example, if we take the term $(\lambda x : A.(\lambda z : A.x))Y$ in a context Γ where Y is a metavariable of type A, then we can represent it by $(\lambda A.(\lambda A.\underline{2}))Y$ in de Bruijn's index notation; this term is reduced in λ_σ-calculus to $(\lambda A.Y[\uparrow])$ which is well typed in the context Γ. In this calculus we have that β-reduction preserves the context declaration of a metavariable.

For the simply typed λ_σ-calculus the following commutation property holds ([DHK95]):

Proposition 1 *Typing and refinement of metavariables commute.*

4 What about first order and higher-order type theories?

The Curry-Howard isomorphism is achieved in intuitionistic higher-order logic by introducing Π and Σ types (which corresponds respectively to universal and existential quantifiers).

In an intuitionistic first order logic, derivation rules that correspond to the quantifier $\forall$ are:

$$\frac{\Gamma \vdash_I A}{\Gamma \vdash_I \forall x A}\,(\text{Intro}^\forall) \qquad \frac{\Gamma \vdash_I \forall x A}{\Gamma \vdash_I A[x \leftarrow t]}\,(\text{Elim}^\forall)$$

In $(\text{Intro}^\forall)$ we assume that x is not free in Γ and in $(\text{Elim}^\forall)$ we assume that t is an object.

In type theory there is no distinction between objects (as t or x in the rule $(\text{Elim}^\forall)$) and propositions (as A). But all the terms are typed even the propositions. In order to have a consistent system, there is an

infinite type hierarchy that says $A, B, \ldots : Type_0, Type_0 : Type_1, \ldots, Type_i : Type_{(i+1)}, \ldots$. And we add the property: if $x : Type_i$ and $i < j$ then $x : Type_j$.

The typing rules that correspond to (Intro$^\forall$) and (Elim$^\forall$) are:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (\forall x : A.B)}\ (\text{Pi}) \qquad \frac{\Gamma \vdash M : (\forall x : A.B) \quad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B[x \leftarrow N]}\ (\text{ApplPi})$$

The type $(\forall x : A.B)$ is usually written $(\Pi x : A.B)$.

We remark that the rules (Pi) and (ApplPi) are a generalisation of (Abs) and (Appl) where the type $A \to B$ is just a notation for $(\Pi x : A.B)$ where x does not occur in B.

Assume the induction axiom over naturals:
$nat\_ind : (\Pi p : nat \to Type.(p\ 0) \to (\Pi x : nat.((p\ x) \to (p\ (succ\ x)))) \to (\Pi n : nat.p\ n))$ in a context Γ where $0 : nat$ and $succ : nat \to nat$. If Z, Z_0, Z_s and W are metavariables, then the term $(nat\_ind\ Z\ Z_0\ (\lambda_{x:nat}.\lambda_{y:(Z\ x)}.Z_s)\ W)$ is well typed in Γ if

1. $\Gamma \vdash W : nat$,
2. $\Gamma \vdash Z : nat \to Type$,
3. $\Gamma \vdash Z_0 : (Z\ 0)$ and
4. $\Gamma, x : nat, y : (Z\ x) \vdash Z_s : (Z\ (succ\ y))$.

Typing judgements (1) and (2) are similar to those of the simply typed theory. However, in judgement (3) we have a type that contains a metavariable, and in judgement (4) we have that the context depends on metavariables also.

If we have implicit typing rules for metavariables (like (Meta_X) in the simply typed theory), then how can we detect and avoid circular typing judgements like:

- $\Gamma_X \vdash X : (w\ X)$,
- $\Gamma_X \vdash X : (w\ Y)$ and $\Gamma_Y \vdash Y : (z\ X)$, or
- $\Gamma_X, y : (w\ X) \vdash X : (z\ y)$?

And, how can we stand the commutation property?

We propose a new declaration scope for metavariables that we call *signature*. A signature is a list of metavariable declarations $\Gamma_{X_1} \vdash X_1 : T_{X_1}, \ldots, \Gamma_{X_n} \vdash X_n : T_{X_n}$ that we denote usually by a Σ symbol where Γ_{X_i} (for each $1 \leq i \leq n$) is the unique context declaration of the metavariable X_i and T_{X_i} is its type.

A (higher-order) typing judgement has the form $\Sigma \triangleright \Gamma \vdash M : A$, where Σ is a signature, Γ is a context, M is a term and A is a type. Intuitively it means that "the judgement $\Gamma \vdash M : A$ is valid, under the signature Σ".

Circular typing judgements may be avoided by rules of well formation of signatures, just like in the case of variable declaration. The refinement operation can be defined as ([Muñ96]):

Definition 3 *Let be a metavariable X, a signature Σ, a context Γ, a type A and the terms M and T such that $\Sigma \triangleright \Gamma_X \vdash T : A_X$ and $\Sigma, (\Gamma_X \vdash X : A_X), \Sigma' \triangleright \Gamma \vdash M : A$. By refining the metavariable X with the term T we obtain the judgement $(\Sigma, \Sigma' \triangleright \Gamma \vdash M : A)\{X \mapsto T\}$*

And the commutation property that we can prove is:

Proposition 2 *Typing judgements obtained by refinement of valid typing judgements are valid too.*

5 Related Works

Metavariables are widely used in very different frameworks. For example, [FD94] presents a proof system, implemented in λProlog, ([Mil91]), that "...supports the construction and manipulation of tree-structured proof that can contain metavariables ..." and [BBKM93] show that the activities of program verification and program synthesis can be unified if metavariables are admitted into proofs. This work is illustrated with an example developed in the Isabelle system ([Pau90]). In both cases, the metavariables are inherited from the implementation languages, that are based in a classical higher-order logical framework.

[Cle92] presents a natural deduction calculus with metavariables in order to propose a proof refinement mechanism. [Mag95] uses metavariables to represent place-holders of incomplete proofs in a proof system based on a constructive type theory. In [DHK95] the metavariables are used as unification variables in a higher-order unification system based on explicit substitutions. In [Bur94] there is a very clear presentation to understand the construction of terms via interactive refinement of proofs, but there are no an explicit usage of places-holders.

In our case, we propose a theoretical support for metavariables, i.e. explicit typing of metavariables (this is the main difference with respect to the approach of [Mag95]) and we are placed in a constructive (intuitionistic) higher-order logical framework (this is the main difference with works in [FD94] and [BBKM93]). We use metavariables to represent incomplete proof terms (in the sense of the Curry-Howard isomorphism) and not unknown propositions as in [Cle92]. We are interested in certain meta-theoretical properties of the proposed calculus as: subject reduction, confluence, strong normalisation, in order to build a theoretically clear system that can be bootstrapped.

6 Conclusions

Place-holders are unknown pieces of terms that allow to represent hypothetical future proofs. A proving process consist in a sequence of refinement steps that replace place-holders in order to build a complete proof.

In an intuitionistic framework, there are at least two families of representations for incomplete proofs: Those that uses an ad hoc data structure, absolutely independent of λ-terms, and those that use the typed λ-calculus. The system Coq ([CCF+95]) is an example of the first family and the system ALF ([AGNvS94]) is an example of the latter one. We remark that both uses λ-calculus to represent (complete) proofs!

A classical λ-calculus gives a very limited mechanism to build proofs. In order to have a flexible and powerful system to represent incomplete proofs and to refine them, we must use an extended typed λ-calculus with explicit substitutions and metavariables. An important invariant over refinements step of incomplete terms is the commutation property with typing rules. Thus we avoid to have inconsistent refinement steps.

For first order and higher-order types theories we propose explicit typing of metavariables. We then introduce the notion of signature that corresponds to a metavariable context declaration; we define for these calculus the refinement operation and the commutation property.

Place-holders and metavariables are widely studied in very different frameworks. Our major contribution to this study is to propose a theoretical handling of this feature in a constructive logic in order to have an isomorphism not only between types and propositions but also between their constructions process.

Direction for future works

We are interested in studying a higher-order constructive calculus with inductive types, explicit substitutions and explicit typing of metavariables, in order to develop a prover system with a flexible mechanism for programming tactics, but also auto-verified and possibly auto-generated, i.e. a boot-strapped system.

Acknowledgement

The author is grateful to Gilles Dowek for many helpful discussions and valuable comments on previous versions of this work. We would also like to thank Cristina Cornes and Cristina Cifuentes by a careful reading of this paper, and in general all the members of the Coq project for fruitful discussions.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitution. *Journal of Functional Programming*, 1(4):375–416, 1991.

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994.

[Bar84] H. P. Barendregt. *The Lambda Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. Elsevier Science Publishers B.V. (North-Holland), 1984.

[BBKM93] D. A. Basin, A. Bundy, I. Kraan, and S. Matthews. A framework for program development based on schematic proof. In *7th Intern. Workshop on Software Specification and Design* (Redondo Beach, CA), pages 162–171, Los Alamitos, CA, 1993. IEEE Computer Society Press. Also available as Technical Report MPI-I-93-231.

[Blo95] R. Bloo. Preservation of strong normalisation for explicit substitution. Technical Report 95-08, Eindhoven University of Technology, Department of Computing Science, March 1995.

[Bur94] R. Burstall. Terms, proofs, and refinement (extended abstract). In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 2–7, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[CCF+95] C. Cornes, J. Courant, J.-C. Filliâtre, G. Huet, P. Manoury, C. Paulin-Mohring, C. Muñoz, C. Murthy, C. Parent, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual Version 5.10. Technical Report RT-0177, INRIA, July 1995.

[CHL95] P.-L. Curien, T. Hardin, and J.-J Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 1995. To appear. Also as 1992 INRIA report 1617.

[Cle92] T. Clement. Using metavariables in natural deduction proofs. In *Proceedings of the 5th Refinement Workshop*, pages 255–271. Springer-Verlag, 1992.

[DHK95] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions (extended abstract). In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 366–374, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[Ell89] Conal Elliott. Higher-order unification with dependent types. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 121–136, Chapel Hill, North Carolina, April 1989. Springer-Verlag LNCS 355.

[FD94] A. Felty and D.Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In Alan Bundy, editor, *Proceedings, 12th International Conference on Automated Deduction*, pages 605–619, Nancy, France, June 1994. Springer-Verlag LNAI 596.

[Fit69] M. C. Fitting. *Intuitionistic Logic Model Theory and Forcing.* North-Holland Publishing Co. Amsterdam, 1969.

[GTL89] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proof and Types.* Cambridge University Press, 1989.

[Hue75] G. Huet. A unification algorithm for typed lambda calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[Kle52] S. C. Kleene. *Introductions to Meta-mathematics.* North-Holland Publishing Co. Amsterdam, 1952.

[KR95] F. Kamareddine and A. Rios. The confluence of the λs_e-calculus via generalized interpretation method. Personal communication, 1995.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$ a journey through calculi of explicit substitutions. In *Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–69, January 1994.

[Mag95] Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Muñ95] C. Muñoz. Confluence and preservation of strong normalisation in an explicit substitutions calculus. Technical Report RR-2762, Unité de recherche INRIA-Rocquencourt, Decembre 1995.

[Muñ96] C. Muñoz. Dependent types with explicit substitutions and explicit typing of meta-variables. In preparation, 1996.

[NG94] R.G. Nickson and L.J. Groves. Metavariables and conditional refinements in the refinement calculus. In D. Till and R. G. F. Shaw, editors, *Proceedings of the 6th Refinement Workshop.* London, Springer-Verlag, 5–7 January 1994.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Río93] A. Ríos. *Contributions à l'étude de λ-calculs avec des substitutions explicites.* PhD thesis, U. Paris VII, 1993.

[Ros92] Kristoffer H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Rémy, editors, *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, number 656 in Lecture Notes in Computer Science, pages 36–50, Pont-a-Mousson, France, July 1992. Springer-Verlag.

[Tal93] Carolyn L. Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112, 1993.