The Implicit Calculus of Constructions

Extending Pure Type Systems with an Intersection Type Binder and Subtyping

Alexandre Miquel[1]

INRIA Rocquencourt – Projet LogiCal
BP 105, 78 153 Le Chesnay cedex, France
`Alexandre.Miquel@inria.fr`

Abstract. In this paper, we introduce a new type system, the *Implicit Calculus of Constructions*, which is a Curry-style variant of the Calculus of Constructions that we extend by adding an intersection type binder—called the *implicit dependent product*. Unlike the usual approach of Type Assignment Systems, the implicit product can be used at every place in the universe hierarchy. We study syntactical properties of this calculus such as the $\beta\eta$-subject reduction property, and we show that the implicit product induces a rich subtyping relation over the type system in a natural way. We also illustrate the specificities of this calculus by revisitting the impredicative encodings of the Calculus of Constructions, and we show that their translation into the implicit calculus helps to reflect the computational meaning of the underlying terms in a more accurate way.

1 Introduction

In the last two decades, the proofs-as-programs paradigm—the Curry-Howard isomorphism—has been used successfully both for understanding the computational meaning of intuitionistic proofs and for implementing proof-assistant tools based on Type Theory. Since earlier work of Martin-Löf in the 70's—themselves inspired by Russel and Whitehead's *Principia*—a large scale of rich formalisms have been proposed to enhance expressiveness of Type Theory. Among those formalisms, the theory of Pure Type Systems (PTS) [2][1] plays an important role since it attempts to give a unifying framework to what seems to be a 'jungle of formalisms' for the one who enters for the first time into the field of Type Theory. Most modern proof assistants based on the Curry-Howard isomorphism such as Alf [11], Coq [3], LEGO [10] or Nuprl [7] implement a formalism which belongs to this family.[2]

Despite of this, PTS-based formalisms have some practical and theoretical drawbacks, due to the inherent 'verbosity' of their terms, which tends to over-

[1] Formerly called *Generalized Type Systems*.

[2] In fact, this is only true for the core language of those proof-assistants, since they also implement features that go beyond the strict framework of PTS, such as sigma-types, primitive inductive data-type declarations and recursive function definitions.

use abstraction and application, especially for type arguments. This is especially true when compared with ML-style languages.

From a practical point of view, writing polymorphic functional programs may become difficult since the programmer has to instantiate explicitly each polymorphic function with the appropriate type arguments before applying its 'real' arguments.

However, there are good reasons to write those extra-annotations in a PTS. The first reason is that there is in general no syntactic distinction between types and terms, so that type abstraction (type application) is only a particular case of λ-abstraction (application). Another reason is that without such type annotations, decidability of type-checking may be lost provided the considered PTS is expressive enough—which is the case of system F for example [17].

From a more theoretical point of view, the verbosity of PTS-terms also tends to hide the real computational contents of proof-terms behind a lot of 'noise' induced by all those type abstractions and applications. A simple example is given by the Leibniz equality which can be defined impredicatively in the Calculus of Constructions[3] by

$$\begin{aligned} \mathsf{eq} &= \lambda A\!:\!\mathsf{Set}\,.\,\lambda x,y\!:\!A\,.\,\Pi P\!:\!A \to \mathsf{Prop}\,.\,P\ x \to P\ x \\ &: \Pi A\!:\!\mathsf{Set}\,.\,A \to A \to \mathsf{Prop} \end{aligned}$$

Using that definition, we can prove reflexivity of equality by the following term:

$$\lambda A\!:\!\mathsf{Set}\,.\,\lambda x\!:\!A\,.\,\lambda P\!:\!A \to \mathsf{Prop}\,.\,\lambda p\!:\!P\ x\,.\,p \quad : \quad \Pi A\!:\!\mathsf{Set}\,.\,\Pi x\!:\!A\,.\,\mathsf{eq}\ A\ x\ x.$$

What is the computational meaning of that proof ? It is simply the identity function $\lambda p\,.\,p$! To understand that point, let us simply remove type annotations in all λ-abstractions—since they play no role in the real process of computation—and write:

$$\lambda A\,.\,\lambda x\,.\,\lambda P\,.\,\lambda p\,.\,p \quad : \quad \Pi A\!:\!\mathsf{Set}\,.\,\Pi x\!:\!A\,.\,\mathsf{eq}\ A\ x\ x.$$

The term above clearly shows that the first three arguments are only used for type-checking purposes, but that only the fourth one is really involved in the computation process. Also notice that the second 'unuseful' λ-abstraction is not a type constructor abstraction.

Many solutions have been proposed to that problem, both on the theoretical and practical sides. Most proof assistants (Coq [3, 15], LEGO [14]) actually implement some kind of 'implicit arguments' to avoid the user the nuisance of writing redundant applications that the system can automatically infer.

A Common Practical Approach. Generally, implementations dealing with implicit arguments are based on a distinction between two kinds of products, abstractions and applications, the ones being called 'explicit' and the other being

[3] For an explanation about the distinction Prop/Set, see paragraph 2.1.

called ‘implicit’. Although explicit and implicit constructions do not semantically differ, the proof-checking system distinguishes them by allowing the user to omit arguments of implicit applications—the ‘implicit arguments’—provided the system may infer them. Such arguments are reconstructed during the type-checking process and then silently kept into the internal representation of terms, since they may be needed later by the conversion test algorithm.

The major advantage of this method is to keep the semantics of the original calculus—modulo the ‘coloring’ of the syntax—since implicit arguments are only implicit for the user, but not for the system. Nevertheless, the user may sometimes be confused by the fact that the system keeps implicit arguments behind its back, especially when two (dependent) types are printed identically although they are not internally identical, due to hidden implicit arguments.

A Calculus with ‘really implicit’ arguments. In [6], M. Hagyia and Y. Toda have studied the possibility of dropping implicit arguments out of the internal representation of the terms of the bicolored Calculus of Constructions—that is, the Calculus of Constructions with explicit and implicit constructors. Their work is based on the idea that if we ensure uniqueness of the reconstruction of implicit arguments (up to β-conversion), then we can drop implicit arguments out of the internal representation of terms since the β-conversion test on implicit terms (i.e terms where implicit arguments have been erased) will give the same result as if done on the corresponding reconstructed explicit terms.

To achieve this goal, they propose a restriction on the syntax of implicit terms in order to ensure decidability and uniqueness (up to β-conversion) of the reconstruction of implicit arguments. But their restriction actually seems to bee too drastic, since it forbids the use of the implicit abstraction in order to avoid dynamic type-checking during β-reduction [6].

The theoretical approach of Type Assignment Systems On the theoretical side, many Curry-style formalisms have been proposed as ‘implicit’ counterparts of usual Pure Type Systems, such as the Curry-style system F [8]. In [5], P. Giannini et al. proposed an uniform description of Curry-style variants of the systems of the cube, which they call the *Type Assignment Systems* (TAS)—as opposed to (Pure) Type Systems. This work follows the idea that from a purely computational point of view, polymorphic terms of the systems of the cube do not depend on their type arguments (which is called ‘structural polymorphism’). As a consequence, they show that it is possible to define an erasing function from Barendregt’s cube to the cube of TAS, which precisely erases all the type dependencies in proof terms, thus mapping PTS-style proof-terms to ordinary pure λ-terms.

The major difference of that work with the approaches described above is that the implicit use of the dependent product is not determined by some colorization of the syntax, but by the stratification of terms. In other words, a dependent product of TAS is ‘implicit’ if and only if it is formed by the rule of polymorphism and, in all other cases, it is an ‘explicit’ product. Also notice that in the TAS framework, the erasing function does not only erase polymorphic applications, but it also erases polymorphic abstractions and type annotations in proof-term abstractions.

It is interesting to mention that the (theoretical) approach of TAS raises the same problem as the (practical) approach of M. Hagiya and Y. Toda, which is the following: if the erasing function erases too much information, then it will identify terms which were not convertible originally, so that the isomorphism between the 'explicit' formalism and the 'implicit' one is irremediably lost. In the framework of TAS, such a problem arises in the systems of the cube involving dependent types [16].

Towards Implicit Pure Type Systems The main limitation of the approach of Type Assignments Systems is to restrict the 'implicit' use of the dependent product to polymorphism. If we want to generalize this approach to all PTS—which are not necessary impredicative—it seems natural to equip them with an implicit product binder, that will be denoted by $\forall x : T . U$ in the following. By making such a syntactic distinction, we allow the choice of the kind of dependent product (explicit or implicit) to be disconnected from the stratification.

Another interesting feature is that this approach tends to identify terms which would not be considered as convertible in a bicolored calculus. As we will see in section 4, such identifications will help us to understand the computational meaning of terms. However, this feature has a high theoretical cost, since it completely changes the underlying semantics.

In the following, we will concentrate our study to the case of the Implicit Calculus of Constructions. However, our approach is general enough to be extended to all the other PTS. In particular, most syntactic results of section 3 can be generalized to what we could call *Implicit Pure Type Systems*.

2 The Implicit Calculus of Constructions

2.1 Syntax

From a pure syntactical point of view, the Implicit Calculus of Constructions (ICC)—or, shortly, the *implicit calculus*—is a Curry-style variant of the Calculus of Construction with universes—a.k.a. ECC [9]—in which we make a distinction between two forms of dependent products: the *explicit product*, denoted by $\Pi x : T . U$, and the *implicit product*, denoted by $\forall x : T . U$. Formally, a term of the implicit calculus (see figure 1) is either:

- a *variable* x;
- a *sort* s;
- an *explicit product* $\Pi x : T . U$, where T and U are terms;
- an *implicit product* $\forall x : T . U$, where T and U are terms;
- an *abstraction* $\lambda x . M$, where M is a term;
- an *application* $M\ N$, where M and N are terms.

The set of sorts of the implicit calculus is defined by

$$\mathcal{S} = \{\mathsf{Prop};\ \mathsf{Set}\} \cup \{\mathsf{Type}_i;\ i > 0\},$$

where Prop and Set denote the impredicative sorts, and $(\mathsf{Type}_i)_{i>0}$ the usual predicative universe hierarchy of the Extended Calculus of Constructions [9]. Notice that here, we follow the convention of the Calculus of Inductive Constructions [18] by making a distinction between two impredicative sorts, since it is convenient to distinguish a sort for propositional types (Prop) from a sort for impredicative data types (Set)—although both sorts are completely isomorphic for the typing rules.

Sorts	s	$::=$	$\mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type}_i \quad (i > 0)$
Terms	M, N, T, U	$::=$	$x \mid s$
		$\mid$	$\Pi x : T . U \mid \forall x : T . U$
		$\mid$	$\lambda x . M \mid M\ N$
Contexts	Γ, Δ	$::=$	$[] \mid \Gamma; [x : T]$

Fig. 1. Syntax of the Implicit Calculus of Constructions

As usual, we will consider terms up to α-conversion. In the following, we will denote by $FV(M)$ the set of free variables of a term M, and by $M\{x := N\}$ the term build by substituting the term N to each free occurrence of x in the term M. Notice that the product binders $\Pi x : T . U$ and $\forall x : T . U$ bind all the free occurrences of the variable x in U, but none of the occurrences of x in T.

In the following, we will denote by $T \to U$ the *non-dependent* explicit product $\Pi x : T . U$ (with $x \notin FV(U)$). This convention only holds for the non-dependent *explicit* product: there is no corresponding notation for the non-dependent *implicit* product.[4] We will also follow the usual writing conventions of the λ-calculus by associating type arrows to the right, multiple applications to the left, and by factorizing consecutive λ-abstractions.

A *declaration* is an ordered pair denoted by $(x : T)$, where x is a variable and T a term. A *typing context*—or shortly, a *context*—is simply a finite ordered list of declarations denoted by $\Gamma = [x_1 : T_1; \ldots; x_n : T_n]$. Concatenation of contexts Γ and Δ is denoted by $\Gamma; \Delta$. A declaration $(x : T)$ *belongs* to a context Γ if $\Gamma = \Gamma_1; [x : T]; \Gamma_2$ for some contexts Γ_1 and Γ_2, that we note $(x : T) \in \Gamma$. Contexts are ordered by

– the *prefix* ordering, denoted by $\Gamma \sqsubset \Gamma'$, which means that $\Gamma' = \Gamma; \Delta$ for some context Δ;

[4] See paragraphs 2.3 and 3.3 for the discussion about the meaning of the non-dependent implicit product, which is given by the (Str) typing rule.

– the *inclusion* ordering, denoted by $\Gamma \subset \Gamma'$, which means that any declaration belonging to Γ also belongs to Γ'.

If $\Gamma = [x_1 : T_1; \ldots; x_n : T_n]$ is a context, the set of *declared variables* of Γ is the set defined by $DV(\Gamma) = \{x_1; \ldots; x_n\}$. We also extend the notations $FV(M)$ and $M\{x := N\}$ to contexts by setting

$$\begin{array}{rrcl} & FV(\Gamma) & = & FV(T_1) \cup \cdots \cup FV(T_n) \\ \text{and} & \Gamma\{x := N\} & = & \big[x_1 : T_1\{x := N\}; \ldots; x_n : T_n\{x := N\}\big], \end{array}$$

the latter notation making sense only if $x \notin DV(\Gamma)$. Finally, we will shall use the notation $\forall\Delta\,.\,U = \forall x_1 : T_1\,.\,\ldots\forall x_n : T_n\,.\,U$ for any context $\Delta = [x_1 : T_1; \ldots; x_n : T_n]$ and for any term U.

2.2 Reduction rules

As for the untyped λ-calculus, we will use the notions of β and η-reduction, defined by

$$(\lambda x\,.\,M)\ N \rhd_\beta M\{x := N\}$$

$$\lambda x\,.\,(M\ x) \rhd_\eta M \qquad (\text{if } x \notin FV(M)),$$

(The need of the η-reduction rule, which is not assumed in the theory of Pure Type Systems, will be explained in paragraphs 2.3 and 3.2.)

We also denote by $\rhd_{\beta\eta} = \rhd_\beta \cup \rhd_\eta$ the notion of $\beta\eta$-reduction, and for each reduction rule $R \in \{\beta;\ \eta;\ \beta\eta\}$, we define

– the *one-step R-reduction*, denoted $\to_R$, as the contextual closure of $\rhd_R$;
– the *R-reduction*, denoted $\twoheadrightarrow_R$, as the reflexive and transitive closure of $\to_R$;
– the *R-convertibility* equivalence, denoted $\cong_R$, as the reflexive, symmetric and transitive closure of $\to_R$.

Proposition 1 (Church-Rosser). *The β-, η- and $\beta\eta$-reduction are Church-Rosser.*

Notice that in the strict framework of Pure Type Systems, the $\beta\eta$-reduction does not satisfy the Church-Rosser property [4], due to the presence of a type annotation in the λ-abstraction. However, such a problem does not arise in the implicit calculus, since we use a Curry-style λ-abstraction. This point has some importance, since the implicit calculus has a strong requirement on the η-reduction rule as we shall see in paragraph 3.2.

As for the untyped λ-calculus, any sequence of $\beta\eta$-reductions can be decomposed as a sequence of β-reductions followed by a sequence of η-reductions. This is a consequence of the following lemma:

Lemma 1 (η-reduction delaying). — *For any terms M_0, M_1 and M_2 such that $M_0 \twoheadrightarrow_\eta M_1$ and $M_1 \twoheadrightarrow_\beta M_2$, there exists a term M_1' such that $M_0 \twoheadrightarrow_\beta M_1'$ and $M_1' \twoheadrightarrow_\eta M_2$.*

2.3 Typing rules

Before introducing typing rules, we have to define two sets **Axiom** $\subset \mathcal{S}^2$ and **Rule** $\subset \mathcal{S}^3$. The set **Axiom**, defined by

$$\begin{array}{rcl}\textbf{Axiom} & = & \{(\mathsf{Prop}, \mathsf{Type}_1);\ (\mathsf{Set}, \mathsf{Type}_1)\} \quad \cup \\ & & \{(\mathsf{Type}_i, \mathsf{Type}_{i+1});\quad i > 0\},\end{array}$$

is used for typing sorts, whereas the set **Rule**, defined by

$$\begin{array}{rcl}\textbf{Rule} & = & \{(s, \mathsf{Prop}, \mathsf{Prop});\quad s \in \mathcal{S}\} \quad \cup \\ & & \{(s, \mathsf{Set}, \mathsf{Set});\quad s \in \mathcal{S}\} \quad \cup \\ & & \{(\mathsf{Type}_i, \mathsf{Type}_i, \mathsf{Type}_i);\quad i > 0\},\end{array}$$

is used for typing products. Note that the same set is used for typing both explicit products and implicit products.

We also need to define an ordering relation between sorts, which is called the *cumulative order*. This ordering relation, denoted by $s_1 \leq s_2$, is defined by

$$\begin{array}{rl}\mathsf{Prop} \leq \mathsf{Prop}, & \mathsf{Set} \leq \mathsf{Set}, \\ \mathsf{Prop} \leq \mathsf{Type}_i, & \mathsf{Set} \leq \mathsf{Type}_i, \\ \mathsf{Type}_i \leq \mathsf{Type}_j & \text{if } i \leq j\end{array}$$

The typing rules of the implicit calculus involve two judgments:

– a judgment denoted by $\Gamma \vdash$, which says: "the context Γ is well-formed";
– a judgment denoted by $\Gamma \vdash M : T$, which says: "under the context Γ, the term M has type T".

Validity of those judgments is defined by mutual induction using rules of figure 2.

The rules (Var), (Sort), (ExpProd), (ImpProd), (Lam), (App), (Conv) and (Cum) are the usual rules of ECC, except that we have an extra rule for the implicit product—which shares the same premises and side-condition as the rule for the explicit product. Another difference between ECC and the implicit calculus is that in the latter, the convertibility rule (Conv) now identifies types up to $\beta\eta$-convertibility.

The rules (Gen) and (Inst) are respectively the introduction and elimination rules for implicit product types. In contrast to the rules (Lam) and (App), the rules (Gen) and (Inst) have no associated constructors. Remark that the rule (Gen) involves a side-condition ensuring that the variable x whose type has to be generalized does not appear free in the term M.

The purpose of the next rule, called (Ext) for 'extensionality', is to enforce the η-subject reduction property in the implicit calculus. Such a rule cannot be derived from the other rules, for the same reasons that it cannot be derived in the Curry-style system F, which is included in ICC. This rule is desirable here, since it gives smoother properties for the subtyping relation, such as the contravariant/covariant subtyping rules in products.[5]

[5] See lemma 4 in paragraph 3.2.

Rules for well-formed contexts

$$\frac{}{[]\vdash}\ (\text{WF-E}) \qquad \frac{\Gamma\vdash T:s \quad x\notin DV(\Gamma)}{\Gamma;[x:T]\vdash}\ (\text{WF-S})$$

Rules for well-typed terms

$$\frac{\Gamma\vdash \quad (x:T)\in\Gamma}{\Gamma\vdash x:T}\ (\textsc{Var}) \qquad \frac{\Gamma\vdash \quad (s_1,s_1)\in\textbf{Axiom}}{\Gamma\vdash s_1:s_2}\ (\textsc{Sort})$$

$$\frac{\Gamma\vdash T:s_1 \quad \Gamma;[x:T]\vdash U:s_2 \quad (s_1,s_2,s_3)\in\textbf{Rule}}{\Gamma\vdash \Pi x:T\,.\,U:s_3}\ (\textsc{ExpProd})$$

$$\frac{\Gamma\vdash T:s_1 \quad \Gamma;[x:T]\vdash U:s_2 \quad (s_1,s_2,s_3)\in\textbf{Rule}}{\Gamma\vdash \forall x:T\,.\,U:s_3}\ (\textsc{ImpProd})$$

$$\frac{\Gamma;[x:T]\vdash M:U \quad \Gamma\vdash \Pi x{:}T\,.\,U:s}{\Gamma\vdash \lambda x\,.\,M:\Pi x:T\,.\,U}\ (\textsc{Lam}) \qquad \frac{\Gamma\vdash M:\Pi x{:}T\,.\,U \quad \Gamma\vdash N:T}{\Gamma\vdash M\ N:U\{x:=N\}}\ (\textsc{App})$$

$$\frac{\Gamma;[x:T]\vdash M:U \quad \Gamma\vdash \forall x:T\,.\,U:s \quad x\notin FV(M)}{\Gamma\vdash M:\forall x:T\,.\,U}\ (\textsc{Gen})$$

$$\frac{\Gamma\vdash M:\forall x:T\,.\,U \quad \Gamma\vdash N:T}{\Gamma\vdash M:U\{x:=N\}}\ (\textsc{Inst})$$

$$\frac{\Gamma\vdash M:T \quad \Gamma\vdash T':s \quad T\cong_{\beta\eta}T'}{\Gamma\vdash M:T'}\ (\textsc{Conv}) \qquad \frac{\Gamma\vdash T:s_1 \quad s_1\leq s_2}{\Gamma\vdash T:s_2}\ (\textsc{Cum})$$

$$\frac{\Gamma\vdash \lambda x\,.\,(M\ x):T \quad x\notin FV(M)}{\Gamma\vdash M:T}\ (\textsc{Ext})$$

$$\frac{\Gamma;[x:T]\vdash M:U \quad x\notin FV(M)\cup FV(U)}{\Gamma\vdash M:U}\ (\textsc{Str})$$

Fig. 2. Typing rules of the Implicit Calculus of Constructions

The meaning of the non-dependent implicit product. The presence of the last rule—called (Str) for "strengthening"—may be surprising, since the corresponding rule is admissible in the (Extended) Calculus of Constructions, and more generally in all functional PTS [4]. In the implicit calculus, this is not the case, due to the presence of non-dependent implicit product. The main consequence of rule (Str)—an the reason for introducing it—is the following:

Lemma 2 (Non-dependent implicit product). — *Let Γ be a context, and let T and U be terms such that $x \notin FV(U)$ and $\forall x\!:\!T\,.\,U$ is a well-formed type in Γ. Then, for any term M we have the equivalence:*

$$\Gamma \vdash M : \forall x\!:\!T\,.\,U \quad \Leftrightarrow \quad \Gamma \vdash M : U$$

In other words, a non-dependent implicit product $\forall x\!:\!T\,.\,U$ has the very same inhabitants as the type U obtained by removing the 'dummy' quantification $\forall\, x\!:\!T$. Without the rule (Str), this result would hold only if the type T is not empty in the context Γ. In paragraph 3.3, we will discuss about the possible problems that this rule might pose with respect to the logical consistency of the calculus.

3 Typing properties

3.1 Subject reduction

The $\beta\eta$-subject reduction of the implicit calculus is surprisingly hard to prove due to the presence of the rule (Ext) whose premise involves a term structurally larger than the term in the conclusion. For that, we have to use a trick based on lemma 1 in order to isolate the rule (Ext). The proof of $\beta\eta$-subject reduction follows the scheme described below:

Step 1. We prove the η-subject reduction property. This result is quite obvious because of rule (Ext), and since rule (Conv) identifies $\beta\eta$-convertible terms.

Step 2. To isolate the rule (Ext), we define a notion of *η-direct derivation*, which restricts the usual notion of derivation by forbidding the use of rule (Ext) in the parts corresponding to the destructuration of the currently typed term—but not in the parts destructuring the context. The corresponding (restricted) judgment is denoted by $\Gamma \vdash_d M : T$.

Step 3. Using that restriction, we prove inversion lemmas and the β-subject reduction property for η-direct judgments $\Gamma \vdash_d M : T$ only.

Step 4. We show that for any valid judgment $\Gamma \vdash M : T$, there exists an η-expansion M_0 of the term M such that $\Gamma \vdash_d M_0 : T$. Using the fact that η-direct judgments enjoy β-subject reduction together with lemma 1, we can extend the β-subject reduction property to the unrestricted form of judgment $\Gamma \vdash M : T$.

3.2 Subtyping

One of the most interesting aspects of the Implicit Calculus of Constructions is the rich subtyping relation induced by the implicit product. This subtyping relation, which is denoted by $\Gamma \vdash T \leqslant T'$, can be defined directly from the typing judgment as the following 'macro':

$$\Gamma \vdash T \leqslant T' \quad \equiv \quad \Gamma; x : T \vdash x : T' \qquad (x \text{ a fresh variable})$$

Using that definition, we can prove that in a given context, subtyping is a preordering on well-formed types which satisfies the expected (Sub) rule:

Lemma 3 (Subtyping preordering). — *The following rules are admissible:*

$$\frac{\Gamma \vdash T : s}{\Gamma \vdash T \leqslant T} \qquad \frac{\Gamma \vdash T_1 \leqslant T_2 \quad \Gamma \vdash T_2 \leqslant T_3}{\Gamma \vdash T_1 \leqslant T_3} \qquad \frac{\Gamma \vdash M : T \quad \Gamma \vdash T \leqslant T'}{\Gamma \vdash M : T'} \text{(Sub)}$$

Moreover, product formation acts in contravariant way for the domain part, and in a covariant way for the codomain part:

Lemma 4 (Subtyping in products). — *The following rules are admissible:*

$$\frac{\Gamma \vdash T' \leqslant T \quad \Gamma; [x : T'] \vdash U \leqslant U'}{\Gamma \vdash \Pi x : T \,.\, U \leqslant \Pi x : T' \,.\, U'} \text{(SubExpProd)}$$

$$\frac{\Gamma \vdash T' \leqslant T \quad \Gamma; [x : T'] \vdash U \leqslant U'}{\Gamma \vdash \forall x : T \,.\, U \leqslant \forall x : T' \,.\, U'} \text{(SubImpProd)}$$

Notice that the rule (SubExpProd) can not be proven to be admissible without the rule (Ext). In fact, this is the main motivation for introducing the rule (Ext), which has been proven equivalent to the rule (SubExpProd) in [12].

Besides the notion of subtyping, we can also define a notion of *typing equivalence*, denoted by $\Gamma \vdash T \sim T'$, which is simply the symmetric closure of the subtyping judgment $\Gamma \vdash T \leqslant T'$. We can prove the following equivalences:

Lemma 5. *The following rules are admissible:*

$$\frac{\Gamma \vdash \forall x_1 : T_1 \,.\, \forall x_2 : T_2 \,.\, U : s \quad \Gamma \vdash \forall x_2 : T_2 \,.\, \forall x_1 : T_1 \,.\, U : s'}{\Gamma \vdash \forall x_1 : T_1 \,.\, \forall x_2 : T_2 \,.\, U \sim \forall x_2 : T_2 \,.\, \forall x_1 : T_1 \,.\, U}$$

$$\frac{\Gamma \vdash \forall x_1 : T_1 \,.\, \forall x_2 : T_2 \,.\, U : s \quad \Gamma \vdash \forall x_2 : T_2 \,.\, \forall x_1 : T_1 \,.\, U : s'}{\Gamma \vdash \Pi x_1 : T_1 \,.\, \forall x_2 : T_2 \,.\, U \sim \forall x_2 : T_2 \,.\, \Pi x_1 : T_1 \,.\, U}$$

(Notice that the premises imply that there is no mutual dependency in the quantifications of conclusions, *i.e.* $x_1 \notin FV(T_2)$ and $x_2 \notin FV(T_1)$.)

3.3 Partial consistency results

In the implicit calculus, we have two ways for encoding the falsity. The first way—which is the usual way in the Calculus of Constructions—is to represent falsity by the type $\Pi A : \mathsf{Prop}\,.\,A$, which is called the *explicit falsity.* But in the implicit calculus, we can also encode the falsity by $\forall A : \mathsf{Prop}\,.\,A$, which is called the *implicit falsity.* As in the Calculus of Constructions, a proof of the explicit falsity is a function which takes a proposition as an argument, and returns a proof of that proposition. On the contrary, a proof of implicit falsity is a proof of any proposition—since implicit falsity is the intersection of all propositional types.

However, both falsities are provably equivalent:

$$\begin{array}{rcl} \lambda f\,.\,f\ (\forall A : \mathsf{Prop}\,.\,A) & : & (\Pi A : \mathsf{Prop}\,.\,A) \to (\forall A : \mathsf{Prop}\,.\,A) \\ \lambda p, A\,.\,p & : & (\forall A : \mathsf{Prop}\,.\,A) \to (\Pi A : \mathsf{Prop}\,.\,A) \end{array}$$

Notice that the last proof is quite general, since we have

$$\lambda p, x\,.\,p \quad : \quad (\forall x : T\,.\,U) \to (\Pi x : T\,.\,U),$$

which means that an explicit product has at least as much inhabitants as the corresponding implicit product.

Proposition 2. — *In the empty context, a proof of implicit falsity has no weak head normal form.*

As a consequence, we have the following relative consistency result:

Proposition 3. — *If the Implicit Calculus of Constructions is strongly normalizing, then it is logically consistent.*

Unfortunately, the strong normalization of the implicit calculus is still an open problem. Nevertheless, we have proven in [13] that a large fragment of the implicit calculus is logically consistent (see section 5). This fragment is precisely the implicit calculus without the rule (Str)—also called the *restricted implicit calculus.*

One may fear that rule (Str) could jeopardize the logical consistency of the whole calculus. For example, the fact that the strengthening rule allows to remove *any* dummy implicit quantification implies that a proof of $\forall x : \mathsf{False}\,.\,\mathsf{False}$ is also a proof of False (here, False denotes one of the falsities). Nevertheless, we must remember that explicit and implicit products are not symmetric—we have an 'explicit' abstraction in the calculus, but no implicit one—so we can not form an implicit equivalent of the identity function to provide a trivial inhabitant of $\forall x : \mathsf{False}\,.\,\mathsf{False}$.

Many arguments—but still no proof—seem to indicate that the (Str) rule does not jeopardize the consistency of the calculus. The first one is that proposition 3 holds for the whole calculus—but this relies on an open normalization problem. In fact, we can prove that strong normalization of the implicit calculus does not depend of the presence of rule (Str):

Proposition 4. — *If the restricted implicit calculus is strongly normalizing, then the (full) implicit calculus is strongly normalizing too.*

4 Impredicative encodings

In this section we shall illustrate the expressiveness of the Implicit Calculus of Constructions by comparing impredicative encodings of lists and dependent lists (vectors), and by studying their relationships with respect to subtyping.

In the implicit calculus, lists are encoded as follows:

$$\begin{aligned}
\mathsf{list} &: \mathsf{Set} \to \mathsf{Set} := \lambda A \,.\, \forall X : \mathsf{Set} \,.\, X \to (A \to X \to X) \to X \\
\mathsf{nil} &: \forall A : \mathsf{Set} \,.\, \mathsf{list}\ A := \lambda x f \,.\, x \\
\mathsf{cons} &: \forall A : \mathsf{Set} \,.\, A \to \mathsf{list}\ A \to \mathsf{list}\ A := \lambda a l x f \,.\, f\ a\ (l\ x\ f)
\end{aligned}$$

Notice that here, the polymorphic constructors nil and cons are exactly the usual constructors of (untyped) lists in the pure λ-calculus. In fact, this result is not specific to the implicit calculus: this example could have been encoded the same way in the Curry-style equivalent of system $F\omega$ in the cube of TAS, since the implicit quantification was precisely used for impredicative products.

In such a framework, it is not necessary to give an extra argument at each 'cons' operation to build a list:

$$\mathsf{cons\ true\ (cons\ false\ (cons\ true\ nil))} \quad : \quad \mathsf{list\ bool.}$$

Using the traditional encoding of lists in the Calculus of Constructions, the same list would have been written

$$\mathsf{cons\ bool\ true\ (cons\ bool\ false\ (cons\ bool\ true\ (nil\ bool)))} \quad : \quad \mathsf{list\ bool}$$

by explicitly instantiating the type of constructors at each construction step.

In the implicit calculus anyway, the constructor of lists has the good covariance property with respect to the subtyping relation:

Proposition 5 (Covariance of the type of lists). — *For all context Γ and for all terms A and B of type* Set *in Γ we have:*

$$\Gamma \vdash A \leqslant B \quad \Rightarrow \quad \Gamma \vdash \mathsf{list}\ A \leqslant \mathsf{list}\ B.$$

In fact, the situation becomes far more interesting if we consider the type of dependent lists—that we call *vectors*. The type of vectors is like the type of lists, except that it also depends on the size of the list. In the implicit calculus, the type of vectors can be encoded as follows

$$\begin{aligned}
\mathsf{vect} \quad &: \quad \mathsf{Set} \to \mathsf{nat} \to \mathsf{Set} \\
&:= \quad \lambda A n \,.\, \forall P : \mathsf{nat} \to \mathsf{Set} \,.\, P\ \mathsf{0} \to (\forall p : \mathsf{nat} \,.\, A \to P\ p \to P\ (\mathsf{S}\ p)) \to P\ n,
\end{aligned}$$

where nat, 0 and S are defined according to the usual encoding of Church integers in Curry-style system F.

Notice that this encoding can not be done in the cube of TAS since the innermost quantification $\forall p : \mathsf{nat}$ introduces an *implicitly dependent type.* Now, the good news come from the fact that we do not need to define a new nil and a new cons for vectors. Indeed, it is straightforward to check that the nil and cons that we defined for building lists have also the following types:

$$\begin{aligned}\mathsf{nil} &\quad:\quad \forall A{:}\mathsf{Set}\,.\,\mathsf{vect}\ A\ 0\\ \mathsf{cons} &\quad:\quad \forall A{:}\mathsf{Set}\,.\,\forall n{:}\mathsf{nat}\,.\,A \to \mathsf{vect}\ A\ n \to \mathsf{vect}\ A\ (\mathsf{S}\ n)\end{aligned}$$

In other words, lists and (fixed-length) vectors share the very same constructors, so we can take back the list of booleans above and assign to it the following more accurate type:

$$\mathsf{cons\ true\ (cons\ false\ (cons\ true\ nil))} \quad:\quad \mathsf{vect\ bool\ (S\ (S\ (S\ 0)))}.$$

In the Calculus of Constructions, such a sharing of constructors is not possible between lists and dependent lists, so we have to define a new pair of constructors nil' and cons' to write the term

$$\begin{array}{l}\mathsf{cons}'\ \mathsf{bool\ (S\ (S\ 0))\ true}\\ \quad(\mathsf{cons}'\ \mathsf{bool\ (S\ 0)\ false}\\ \qquad(\mathsf{cons}'\ \mathsf{bool\ 0\ true}\ (\mathsf{nil}'\ \mathsf{bool})))\quad:\quad \mathsf{vect\ bool\ (S\ (S\ (S\ 0)))}.\end{array}$$

whose real computational contents is completely hidden by the type and size arguments given to the constructors nil' and cons'.

In the implicit calculus, we can even derive that the type of vectors (of a given size) is a subtype of the type of lists:

Proposition 6. — *For all context Γ and for all terms A and n such that $\Gamma \vdash A : \mathsf{Set}$ and $\Gamma \vdash n : \mathsf{nat}$, one can derive the subtyping judgment:*

$$\Gamma \vdash \mathsf{vect}\ A\ n \leqslant \mathsf{list}\ A.$$

To give another illustration of the expressive power of the Implicit Calculus of Constructions, let us go back to the world of propositions by studying the case of Leibniz equality. In the implicit calculus, the natural impredicative encoding of equality is the following[6]:

$$\mathsf{eq}\quad:\quad \Pi A{:}\mathsf{Set}\,.\,A \to A \to \mathsf{Prop}\quad:=\quad \lambda A\,.\,\forall P{:}A \to \mathsf{Prop}\,.\,P\ x \to P\ x.$$

Using that encoding, the reflexivity of equality is simply proven by the identity function

$$\lambda p\,.\,p\quad:\quad \forall A{:}\mathsf{Set}\,.\,\mathsf{eq}\ A\ x\ x$$

whereas the proof of transitivity is given by the composition operator

$$\lambda fgp\,.\,g\ (f\ p)\quad:\quad \forall A{:}\mathsf{Set}\,.\,\forall x,y,z{:}A\,.\,\mathsf{eq}\ A\ x\ y \to \mathsf{eq}\ A\ y\ z \to \mathsf{eq}\ A\ x\ z.$$

[6] Notice that here, the type variable A is explicitely used by the definition of equality so that the corresponding dependency must be explicit in the type $\Pi A{:}\mathsf{Set}\,.\,A \to A \to \mathsf{Prop}$. In proof-checking systems, such a type argument can be easily inferred and is usually treated as implicit. This example shows that the typing information that we can really drop out of the syntax has little to do with the typing information that can be automatically inferred in practice.

5 Semantics

Building a model of the Implicit Calculus of Constructions is a fascinating challenge, especially because of its rich subtyping relation. The main difficulty is caused by the interpretation of the Curry-style λ-abstraction which imply the traditional typing ambiguity, but also a *stratification ambiguity*, since the identity $\lambda x\,.\,x$ has type $\forall A : \mathsf{Prop}\,.\,A \to A$, but also the types $\forall A : \mathsf{Type}_i\,.\,A \to A$ for all $i > 0$.

In [13], we have proposed a model of the restricted implicit calculus—that is the implicit calculus without the rule (Str). This model is based on a untyped domain theoretical interpretation of terms in a large coherence space containing types as 'ground values'. The corresponding interpretation has nice properties, since it allows to interpret all terms—even ill-typed ones—independently of their possible types. Using that interpretation, we proved the logical consistency of the restricted implicit calculus.

6 Future work

Strong normalization The next topic we will focus on is to prove that the implicit calculus is strongly normalizing. It is reasonable to think that the ideas introduced in [13] will be helpful for building a strong normalization model of the implicit calculus. Technically, such a construction could be achieved by adding normalization information into the actual model using Altenkirch's Λ-sets[1]. But for achieving this goal, we first need to modify the model in such a way that all types becomes inhabited, since we want to interpret terms in all contexts.

Decidability of type-checking The decidability of type-checking in the implicit calculus is still an open problem. However, we strongly conjecture that type-checking is undecidable, at least because it contains the Curry-style system F. In fact, the inclusion of Curry-style system F into the implicit calculus seems to be only a minor point, since the implicit product allows to hide in the implicit calculus far more typing information than in Type Assignment Systems. For that reason, the implicit calculus seems to be a poor candidate for being used in a proof assistant system. Nevertheless, it could be fruitful to study *ad hoc* restrictions of the implicit calculus preserving decidability of type-checking, in order to test them in real proof-checking environments.

Extending this approach to all PTS The approach described in this paper seems to be well-suited for being extended to all Pure Type Systems. Within the more general framework of *Implicit Pure Type Systems*, it is possible to increase generality by making a distinction between two sets $\mathbf{Rule}^{\Pi}$ and $\mathbf{Rule}^{\forall}$ for typing explicit and implicit products respectively, since the proofs of most of the results exposed in section 3 (including the $\beta\eta$-subject reduction property) do not rely on the fact that in the implicit calculus, both explicit and implicit products share the same formation rules.

References

1. T. Altenkirch. *Constructions, Inductive types and Strong Normalization.* PhD thesis, University of Edinburgh, 1993.
2. Henk Barendregt. Introduction to generalized type systems. Technical Report 90-8, University of Nijmegen, Department of Informatics, May 1990.
3. B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.
4. J. H. Geuvers and M. J. Nederhof. A modular proof of strong normalization for the calculus of constructions. In *Journal of Functional Programming*, volume 1,2(1991), pages 155–189, 1991.
5. P. Giannini, F. Honsel, and S. Ronchi della Rocca. Type inference: some results, some problems. In *Fundamenta Informaticæ*, volume 19(1,2), pages 87–126, 1993.
6. M. Hagiya and Y. Toda. On implicit arguments. Technical Report 95-1, Department of Information Science, Faculty of Science, University of Tokyo, 1995.
7. Paul B. Jackson. The Nuprl proof development system, version 4.1 reference manual and user's guide. Technical report, Cornell University, 1994.
8. D. Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pages 88–98, 1983.
9. Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.
10. Zhaohui Luo and Randy Pollack. Lego proof development system: User's manual. Technical Report 92-228, LFCS, 1992.
11. Lena Magnusson. Introduction to ALF — an interactive proof editor. In Uffe H. Engberg, Kim G. Larsen, and Peter D. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory* (Aarhus, Denmark, 17–19 October, 1994), number NS-94-6 in Notes Series, page 269, Department of Computer Science, University of Aarhus, December 1994. BRICS. vi+483.
12. Alexandre Miquel. Arguments implicites dans le calcul des constructions: étude d'un formalisme à la Curry. Master's thesis, Université Denis-Diderot Paris 7, octobre 1998.
13. Alexandre Miquel. A model for impredicative type systems with universes, intersection types and subtyping. In *Proceedings of the 15 th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*, 2000.
14. R. Pollack. Implicit syntax. In Gérard Huet and Gordon Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks (Antibes)*, may 1990.
15. A. Saïbi. *Algèbre Constructive en Théorie des Types, Outils génériques pour la modélisation et la démonstration, Application à la théorie des Catégories.* PhD thesis, Université Paris VI, 1998.
16. S. van Bakel, L. Liquori, R. Ronchi della Rocca, and P. Urzyczyn. Comparing Cubes. In A. Nerode and Yu. V. Matiyasevich, editors, *Proceedings of LFCS '94. Third International Symposium on Logical Foundations of Computer Science,* St. Petersburg, Russia, volume 813 of *Lecture Notes in Computer Science*, pages 353–365. Springer-Verlag, 1994.
17. J. B. Wells. Typability and type checking in system F are equivalent and undecidable. In *Annals of Pure and Applied Logic*, volume 98(1-3), pages 111–156, 1999.
18. B. Werner. *Une théorie des Constructions Inductives.* PhD thesis, Université Paris VII, 1994.