The Implicit Calculus of Constructions

Extending Pure Type Systems with an Intersection Type Binder and Subtyping

Alexandre Miquel

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 revisiting 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 work of Martin-Löf in the 70's, 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-use

[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.

S. Abramsky (Ed.): TLCA 2001, LNCS 2044, pp. 344–359, 2001.

abstraction and application, particularly 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 explicitly instanciate 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: type abstraction (type application) is only a particular case of λ-abstraction (term application). Another reason is that without such type annotations, decidability of type-checking may be lost when the considered PTS is expressive enough. This 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 this proof ? It is simply the identity function $\lambda p\,.\,p$. To understand that point, let us remove type annotations in all λ-abstractions (since they play no role in the process of computation) to obtain :

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

The term above shows that the first three arguments are only used for type-checking purposes, and that only the fourth one is really involved in the computation process.

Many solutions have been proposed to that problem, both on the theoretical and practical sides. Most proof assistants (Coq [3,15], LEGO [14]) 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, which may be either 'explicit' or '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 is able to infer them. Such arguments are reconstructed during the type-checking process

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

and then silently kept into the internal representation of terms, since they might be needed later by the conversion test.

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 following idea: 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 performed on the corresponding reconstructed explicit terms.

To achieve this goal, they propose a restriction of 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 be 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 (this is called 'structural polymorphism'). As a consequence, the authors 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 between this work and the approaches described above is that the implicit use of the dependent product is not determined by some coloring 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: if

the erasing function erases too much information, then it will identify terms which were not originally convertible. The isomorphism between 'explicit' and 'implicit' formalisms is then irremediably lost. In the framework of TAS, this problem arises in the systems of the cube involving dependent types [16].

Towards Implicit Pure Type Systems. The main limitation of the approach of TAS is that it restricts the 'implicit' use of the dependent product to polymorphism. If we want to generalize this approach to all PTS—which are not necessarily impredicative—it seems natural to equip them with an implicit product binder (written $\forall x : T \,.\, U$). Such a syntactic distinction naturally disconnects the kind of dependent product (explicit or implicit) from the stratification. Nevertheless, this approach raises two important issues:

The first one is that the presence of an implicit product binder (which can be used at any level of the hierarchy) induces a deep change of the underlying semantics. In particular, the isomorphism between explicit and implicit formalisms is definitively lost. This is not necessarily a negative aspect: it simply means that in our approach, 'implicit arguments' are now really implicit, in the sense that they can no more be interpreted by some invisible applications or abstractions. (In particular, the domain-theoretical model described in [13] really interprets implicit products as intersections.)

The other point raised by the introduction of an implicit product binder is that the arguments which may become really implicit (without jeopardizing the consistency of the system) have little to do with the arguments that today's algorithms are able to infer (this will be illustrated by our examples in Sect. 5). For that reason, our approach has mostly a theoretical significance, especially to understand the computational meaning of proofs, but the formalism seems to be a bad candidate for being used practically in a real proof-checking environment.

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 Sect. 3 can be generalized to what we could call *Implicit Pure Type Systems*.

2 The Implicit Calculus of Constructions

2.1 Syntax

The Implicit Calculus of Constructions (ICC)—or, shortly, the *implicit calculus*—is a Curry-style variant of the Calculus of Constructions 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 $\forall x : T \,.\, U$. The syntax of *sorts*, *terms* and *contexts* is given in Fig. 1. We follow here the convention of the Calculus of Inductive Constructions [18] by making a distinction between two impredicative sorts : a sort Prop for propositional types, and a sort Set for impredicative data types. However, both impredicative sorts are 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

Terms will be considered up to α-conversion. The set of free variables of a term M is written $FV(M)$, and $M\{x := N\}$ denotes the (external) substitution operation. 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.

The *non-dependent* explicit product $\Pi x : T \,.\, U$ (where $x \notin FV(U)$) is written $T \to U$.[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 write $(x : T) \in \Gamma$. Contexts are ordered by

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

$$FV(\Gamma) = FV(T_1) \cup \cdots \cup FV(T_n)$$

$$\text{and} \qquad \Gamma\{x := N\} = \big[x_1 : T_1\{x := N\}; \ldots ; x_n : T_n\{x := N\}\big],$$

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

[4] There is no equivalent notation for the non-dependent implicit product, whose meaning will be discussed in paragraph 2.3.

2.2 Reduction Rules

As for the untyped λ-calculus, we will use the notions of β and η-reduction. (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.) For each reduction rule $R \in \{\beta;\ \eta;\ \beta\eta\}$, we define

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

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

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.

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, which will be useful for proving the $\beta\eta$-subject reduction property :

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

The typing rules of the implicit calculus are parametrized by a set $\mathbf{Axiom} \subset \mathcal{S}^2$ for typing sorts, a set $\mathbf{Rule} \subset \mathcal{S}^3$ for typing both explicit and implicit products, and a cumulative ordering $s_1 \leq s_2$ between sorts, which are summarized in Fig. 2. Typing rules of the implicit calculus involve two judgments:

- $\Gamma \vdash$, which means: "the context Γ is well-formed";
- $\Gamma \vdash M : T$, which means: "under the context Γ, the term M has type T".

Validity of those judgments is defined by mutual induction using rules of Fig. 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 as the rule for the explicit product. Moreover, the convertibility rule (Conv) now identifies types up to $\beta\eta$-convertibility.

The rules (Gen) and (Inst) are 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

Axioms, product formation rules and cumulative ordering

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

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

$$\mathsf{Prop} \leq \mathsf{Prop}; \quad \mathsf{Set} \leq \mathsf{Set}; \quad \mathsf{Prop} \leq \mathsf{Type}_i; \quad \mathsf{Set} \leq \mathsf{Type}_i; \quad \mathsf{Type}_i \leq \mathsf{Type}_j \text{ if } i \leq j$$

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

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

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

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

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

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

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

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

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

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

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

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 Curry-style system F, which is included in ICC. This rule is desirable here, since it gives smoother properties to the subtyping relation, such as the contravariant/covariant subtyping rules in products.[5]

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 products. 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.

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).

Step 1: Preliminary Results. We first prove the following three lemmas by an immediate induction on the structure of derivations :

Lemma 3 (Well-formed contexts). — *Let Γ be a context.*

1. *If Γ is well-formed, then each prefix of Γ is also well-formed.*
2. *If $\Gamma \vdash M : T$, then Γ is well-formed.*

[5] See lemma 14 in paragraph 3.2.

Lemma 4 (Weakening). — *Let Γ and Γ' be two contexts such that $\Gamma \subset \Gamma'$. If $\Gamma \vdash M : T$ and Γ' is well-formed, then $\Gamma' \vdash M : T$.*

Lemma 5 (Substitutivity). — *If $\Gamma_1 \vdash M_0 : T_0$ and $\Gamma_1; [x_0 : T_0]; \Gamma_2 \vdash M : T$, then*

$$\Gamma_1; (\Gamma_2\{x_0 := M_0\}) \vdash M\{x_0 := M_0\} : T\{x_0 := M_0\}.$$

Step 2: The η-Subject Reduction Property. We now need to show that rule (Ext) can only be used at some places in a derivation. For that, we have to introduce the notion of *stable form*. A term M is said to be

1. a *sort form* if $M \cong_{\beta\eta} \forall \Delta \,.\, s$ for some context Δ and some sort s.
2. a *product form* if $M \cong_{\beta\eta} \forall \Delta \,.\, \Pi x : T \,.\, U$ for some context Δ and some terms T and U;
3. a *stable form* if M is either a sort form or a product form.

The terminology of 'stable form' comes from the fact that stable forms are preserved at the right-hand side of judgments by subtyping rules such as (Inst), (Gen), (Ext), (Str), (Conv) or (Cum).

Lemma 6 (Stable forms). — *If $\Gamma \vdash M : T$, then*

1. *if M is a sort, an explicit or an implicit product, then T is a sort form;*
2. *if M is a λ-abstraction, then T is a product form*

Using this lemma, we prove the inversion lemma for explicit and implicit products, which is necessary to establish the η-subject reduction property.

Lemma 7 (Inversion of products). — *If $\Gamma \vdash Bx : T \,.\, U : R$ (where B is one of Π or $\forall$), then there exists a context Δ and four sorts s_1, s_2, s_3, s such that*

1. $R \cong_{\beta\eta} \forall \Delta \,.\, s$;
2. $\Gamma; \Delta \vdash T : s_1$;
3. $\Gamma; \Delta; [x : T] \vdash U : s_2$;
4. $(s_1, s_2, s_3) \in \mathbf{Rule}$;
5. $s_3 \leq s$.

Lemma 8 (Type of types). — *If $\Gamma \vdash M : T$, there there exists a sort s such that $\Gamma \vdash T : s$.*

Lemma 9 (Context conversion). — *Let Γ and Γ' be contexts such that $\Gamma \cong_{\beta\eta} \Gamma'$. If $\Gamma \vdash M : T$ and if Γ' is well-formed, then $\Gamma' \vdash M : T$.*

Proposition 2 (η-subject reduction). — *If $\Gamma \vdash M : T$ and $M \to_\eta M'$, then $\Gamma \vdash M' : T$.*

Step 3: η-direct Derivations. We now need to isolate rule (Ext). For that, we say that a derivation of $\Gamma \vdash M : T$ is *η-direct* if one of the following conditions is satisfied :

- the last rule is (Var) or (Sort);
- the last rule is (ExpProd), (ImpProd) or (App), and the derivation of both premises are η-direct;
- the last rule is (Lam) and the derivation of the first premise is η-direct;
- the last rule is (Gen), (Inst), (Conv), (Cum) or (Str) and the derivation of the first premise is η-direct.

Intuitively, an η-direct derivation of a judgement $\Gamma \vdash M : T$ is a derivation in which the rule (Ext) can not appear in the parts of the derivation corresponding to the destructuration of the term M.

In the following, we will write $\Gamma \vdash_d M : T$ when a judgment $\Gamma \vdash M : T$ has an η-direct derivation. This notion has good closure properties: lemmas 4, 5, 6, 7, 8 and 9 still hold even if we replace $\Gamma \vdash M : T$ by $\Gamma \vdash_d M : T$ everywhere. (However, the η-subject reduction property does not hold when considering η-direct derivations only.)

Lemma 10 (η-direct inversion of abstraction). — *If $\Gamma \vdash_d \lambda x \,.\, M : R$, then there exists a context Δ and two terms T, U such that :*

1. $R \cong_{\beta\eta} \forall \Delta \,.\, \Pi x : T \,.\, U$;
2. $\Gamma; \Delta; [x : T] \vdash_d M : U$.

Lemma 11 (η-direct β-subject reduction). — *If $\Gamma \vdash_d M : T$ and $M \to_\beta M'$, then $\Gamma \vdash_d M' : T$.*

Step 4: β-Subject Reduction. Before concluding, we need to show that any derivation of $\Gamma \vdash M : T$ can be transformed into an η-direct derivation provided we make some η-expansions in the term M.

Lemma 12 (η-direct expansion). — *If $\Gamma \vdash M : T$, then there exists a term M_0 such that $M_0 \twoheadrightarrow_\eta M$ and $\Gamma \vdash_d M_0 : T$.*

The β-subject reduction property is then an immediate consequence of lemmas 1, 11 and 12.

Proposition 3 (β-subject reduction). — *If $\Gamma \vdash M : T$ and $M \twoheadrightarrow_\beta M'$, then $\Gamma \vdash M' : T$.*

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 13 (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 a contravariant way for the domain part, and in a covariant way for the codomain part:

Lemma 14 (Subtyping in products). — *The following rules are admissible:*

$$\frac{\Gamma \vdash T' \leqslant T \quad \Gamma; [x : T'] \vdash U \leqslant U'}{\Gamma \vdash \Pi x : T \,.\, U \leqslant \Pi x : T' \,.\, U'} \qquad \frac{\Gamma \vdash T' \leqslant T \quad \Gamma; [x : T'] \vdash U \leqslant U'}{\Gamma \vdash \forall x : T \,.\, U \leqslant \forall x : T' \,.\, U'}$$

The subtyping rule for explicit products would not hold without the rule (Ext). This is the main motivation for introducing the rule (Ext), which has been proven equivalent to the subtyping rule for explicit products 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 15 (Product commutations). — *The following rules are admissible:*

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

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

(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 Consistency Results

In the implicit calculus, there are two propositions for representing the falsity: the *explicit falsity* $\Pi A : \mathsf{Prop} \,.\, A$ and the *implicit falsity* $\forall A : \mathsf{Prop} \,.\, A$. However, both falsities are provably equivalent:

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

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.

The main consistency result of the Implicit Calculus of Constructions is a consequence of the following lemma :

Lemma 16 (Stable forms). — *In the empty context, the type of a term which has a weak head normal form is a stable form.*

Since the implicit falsity is not a stable form, we have :

Proposition 4. — *If the Implicit Calculus of Constructions is strongly normalizing, then it is logically consistent.*

4 Semantics and Strong Normalization

Building a model of the Implicit Calculus of Constructions is a fascinating challenge, especially because 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*. For instance, the identity $\lambda x \,.\, x$ has several types such as $\forall A : \mathsf{Prop} \,.\, A \to A$ or $\forall A : \mathsf{Type}_i \,.\, A \to A$ $(i > 0)$ which are not defined at the same level of the universe hierarchy.

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

More recently, we have transformed this model into a strong normalization model, using the ideas of [1] by incorporating reducibility information into the denotation of types. This normalization model now interprets the full calculus—including the strengthening rule—thus proving the following result :[6]

Theorem 1 (Strong normalization). — *Every well-typed term of the Implicit Calculus of Constructions is strongly normalizing.*

Corollary 1. — *The Implicit Calculus of Constructions is logically consistent.*

[6] The manuscript of the strong normalization proof is available on the author's web page at `http://pauillac.inria.fr/~miquel`.

5 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:

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

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 instanciating 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

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

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

$$\mathsf{nil} \quad : \quad \forall A : \mathsf{Set} \,.\, \mathsf{vect}\ A\ 0$$

$$\mathsf{cons} \quad : \quad \forall A : \mathsf{Set} \,.\, \forall n : \mathsf{nat} \,.\, A \to \mathsf{vect}\ A\ n \to \mathsf{vect}\ A\ (\mathsf{S}\ n)$$

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

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

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 study the case of Leibniz equality. In the implicit calculus, the natural impredicative encoding of equality is the following:

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

The reflexivity of equality is simply proven by the identity function

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

whereas the proof of transitivity is given by the composition operator

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

A Remark about Implicit Positions. In the example above, the type parameter A is an implicit argument of the reflexivity and transitivity proofs, but it is an explicit argument of the equality predicate, although it can be easily infered in that context. On the contrary, the implicit calculus allows the use of implicit elimination predicates (see for instance the encoding of vectors), although the inference of such predicates require complex techniques based on higher-order unification in practice. Those examples show that the arguments that can be automatically infered and the arguments that can be dropped out of the syntax without harm for the consistency are generally not the same.

6 Future Work

Undecidability of Type-Checking. 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 far more typing information than in the TAS. For that reason, the implicit calculus is not suitable for being used in a proof assistant system. Nevertheless, it could be fruitful to study *ad hoc* restrictions of the implicit calculus, in which decidability of type-checking is preserved.

Extending this Approach to All PTS. The approach described here can be easily extended to all Pure Type Systems. Within the more general framework of *Implicit Pure Type Systems*, it is possible to have different formation rules for explicit and implicit products (by introducing two sets $\mathbf{Rule}^{\Pi}, \mathbf{Rule}^{\forall} \subset \mathcal{S}^3$ instead of the single set **Rule** of the Implicit Calculus of Constructions). In that framework, most of the results exposed in Sect. 3 still hold (including the $\beta\eta$-subject reduction property), since their proofs do not rely on the assumption that 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. Honsell, 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.