The Implicit Calculus of Constructions as a Programming Language with Dependent Types

Bruno Barras and Bruno Bernardo

INRIA Futurs and Ecole polytechnique, France
{Bruno.Barras,Bruno.Bernardo}@lix.polytechnique.fr

Abstract. In this paper, we show how Miquel's Implicit Calculus of Constructions (ICC) can be used as a programming language featuring dependent types. Since this system has an undecidable type-checking, we introduce a more verbose variant, called ICC* which fixes this issue. Datatypes and program specifications are enriched with logical assertions (such as preconditions, postconditions, invariants) and programs are decorated with proofs of those assertions. The point of using ICC* rather than the Calculus of Constructions (the core formalism of the Coq proof assistant) is that all of the static information (types and proof objects) is transparent, in the sense that it does not affect the computational behavior. This is concretized by a built-in extraction procedure that removes this static information. We also illustrate the main features of ICC* on classical examples of dependently typed programs.

1 Introduction

In software verification, typing disciplines have shown to be a decisive step towards safer programs. The success of strongly typed functional languages of the ML family is an evidence of that claim. Still, in those systems, typing is not expressive enough to address problems such as array bound checks.

Such issue can be alleviated by using dependent types. Dependent ML [19] is an extension of SML implementing a restricted form of dependent types. The idea is to annotate datatype specifications and program types with expressions in a given constraint domain. Type-checking generate constraints whose satisfiability is checked automatically. But it is limited to decidable constraint domains since the programmer is not allowed to help the type-checker by providing the proof of the satifiability of constraints. The main point of having restricted dependent types is that it applies to programming languages with non-pure features (side-effects, input/output,...).

The system ATS [4] is an evolution of DML that integrates theorem proving in the LF style in case the automatic solver fails. It lets the programmer prove simple invariants (there is very little support for proof construction). Proofs systems (let us name a small number of them: Epigram [8], Agda [13], NuPRL [5] and Coq [17]) provide better tools for proof automation, but in most of them, the distinction between statics (logical and typing arguments) and dynamics (actual code) raises problems.

To illustrate this claim, let us recall one typical example in programming with dependent types: vectors. In order to statically check that programs never access a vector out of its bounds, its type is decorated with an integer representing its length. This can lead

R. Amadio (Ed.): FOSSACS 2008, LNCS 4962, pp. 365–379, 2008.

to more efficient programs since no runtime check is necessary. It is also safer since it is known statically that such program never raises an exception nor returns dummy values.

In the Calculus of Constructions (CC, the core formalism of Coq), one defines a type `vect` parameterized by a natural number and two constructors `nil` and `cons`. `vect` n is the type of vectors of length n (A is the type of the elements).

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

For instance the list $[x_1; x_2; x_3]$ is represented as $(\texttt{cons}\, 2\, x_1\, (\texttt{cons}\, 1\, x_2\, (\texttt{cons}\, 0\, x_3\, \mathit{nil})))$. In fact, the first argument of `cons` is not intended to be part of the data structure: it is used only for type-checking purposes. The safe access function can be specified as

$$\texttt{get} : \Pi(n{:}\texttt{nat})\,(i{:}\texttt{nat}).\, \texttt{vect}\, n \to (i < n) \to A,$$

That is, `get` is a function taking as argument a vector size n, the accessed index i, a vector of the specified size and a proof that i is within the bounds of the vector. Here again, we have two arguments (n and the proof) that do not participate in computing the result, but merely help type-checking.

We can see that programming in the Calculus of Constructions is quite coarse: programs have arguments that are indeed only static information (type decorations, proof objects, dependencies). There exists a procedure called extraction (described in [6]) that produces source code for a number of functional languages from intuitionistic proofs. It tries to remove this static information. The decision of keeping or removing an argument is made by the user when he defines a new type. For this purpose, Coq considers two sorts (the types of types) Prop and Set that are almost identical regarding typing, but types of Prop are intended to be logical propositions (like $i < n$), while types of Set are the actual datatypes (like `nat` and `vect`). In the example, the proof argument of `get` would be erased, but the length n would not. This issue could be alleviated by doing a dead-code analysis [15], but it would not allow the user to specify *a priori* arguments that shall not be used in the algorithmic part of the proof. Another drawback of the extraction approach is that it is external to the system. This means that within the logic, the programmer deals with the fully decorated term. In situations where datatypes carry proofs to guarantee invariants, two datastructures may be equal, but containing different proofs. Since there is no proof-irrelevance, such objects cannot be proven equal in spite of having the same runtime counterparts.

The Implicit Calculus of Constructions (ICC, see [10] and [11]) offers a more satisfying alternative to the distinction between Prop and Set. It is a Curry-style presentation of the Calculus of Constructions.[1] It features a so-called implicit product that corresponds to an intersection type rather than a function type. The main drawback of this system is the undecidability of type-checking. This is mainly because terms do not carry the arbitrarily complex proofs. This means that programs do not carry enough information to recheck them, which is a problem from an implementation point of view. Proving

[1] Type Assignment Systems are also type systems in Curry style, but the usage of the polymorphic quantification is too restrictive for our purposes.

a skeptical third party that a program is correctly typed requires communicating the full derivation.

The main idea of this paper is to introduce a more verbose variant of ICC such that typing is decidable. This is made by decorating terms with the implicit information. The challenge is to ensure that this information does not get in the way, despite the fact that it must be maintained consistent when evaluating a term.

The paper is organized as follows: we define a new calculus (ICC*), and show its main metatheoretical properties. Consistency is established thanks to a sound and complete translation towards a subset of ICC called ICC$^-$. We also introduce a reduction on decorated terms that enjoys subject-reduction, meaning that it maintains decorations consistent. Then we revisit some classical examples and show that a significative part of the expressiveness of ICC is captured. We also discuss several extensions that would make the system more akin to be a pratical programming language.

2 A Decidable Implicit Calculus of Constructions

2.1 Syntax

Its syntax (see Figure 1) is the same that in the standard Calculus of Constructions in Church style, except that we duplicate each operation (product, abstraction and application) into an explicit one and an implicit one. As often in Type Theory, terms and types belong to the same syntactic class, and special constants called sorts represent the types of types.

Sorts $(\mathsf{Type}_i)_{i\in\mathbb{N}}$ denote the usual predicative universe hierarchy of the Extended Calculus of Constructions [7]. There is only one impredicative sort, Prop, because the distinction between propositional types and data types will be made by defining a term as being explicit (data types) or implicit (propositional types) instead of giving it a type of type Set or Prop.

Sorts	$s ::= \mathsf{Prop} \mid \mathsf{Type}_i \ (i \in \mathbb{N})$
Terms	$M ::= x \mid s$
	$\mid \Pi(x:M_1).\, M_2 \mid \lambda(x:M_1).\, M_2 \mid M_1\, M_2 \ \ (explicit)$
	$\mid \Pi[x:M_1].\, M_2 \mid \lambda[x:M_1].\, M_2 \mid M_1[M_2] \ \ (implicit)$
Contexts	$\Gamma ::= [] \mid \Gamma; x : M$

Fig. 1. Syntax of ICC*

As usual we consider terms up to α-conversion. The set of free variables of term t is written $\mathsf{FV}(t)$. Arrow types are explicit non-dependent products (if $x \notin \mathsf{FV}(U)$, we write $T \to U$ for $\Pi(x:T).\, U$). Substitution of the free occurrences of variable x by N in term M is noted $M\{x/N\}$. We write $\mathsf{DV}(\Gamma)$ the set of variables x that are declared in Γ, *i.e.* such that $(x : T) \in \Gamma$ for some term T.

2.2 Extraction

We define inductively (see Fig.2) an *extraction* function $M \mapsto M^*$ that associates a term of ICC to every term of our calculus. This function removes the static information: domains of abstractions, implicit arguments and implicit abstractions. Beware that extraction does not preserve α-conversion for any term. So, many properties of extraction will hold only for well-typed terms[2].

ICC Terms $M ::= x \mid s \mid \Pi(x{:}M_1).\, M_2 \mid \forall(x{:}M_1).M_2 \mid \lambda x.M \mid M_1\, M_2$
(ICC has the same set of sorts $\mathcal{S}$ as ICC)*

Extraction

$$\begin{array}{rlrl} s^* &= s & x^* &= x \\ (\Pi(x{:}T).\, U)^* &= \Pi(x{:}T^*).\, U^* & (\Pi[x{:}T].\, U)^* &= \forall(x{:}T^*).U^* \\ (\lambda(x{:}T).\, U)^* &= \lambda x.U^* & (\lambda[x{:}T].\, U)^* &= U^* \\ (MN)^* &= M^*N^* & (M[N])^* &= M^* \end{array}$$

Fig. 2. ICC terms and Extraction

2.3 Typing Rules

For any relation R in ICC or in ICC*, we will use the symbols $\rhd_R$, $\rightarrow_R$, $\rightarrow_R^*$, $\rightarrow_R^+$ and $\cong_R$ to denote the relation itself, its congruence closure, the reflexive and transitive closure of $\rightarrow_R$, the transitive closure of $\rightarrow_R$ and the reflexive, symmetric and transitive closure of $\rightarrow_R$. $\rightarrow_R^h$ will denote the head reduction of $\rhd_R$ - reduction occurs in the left subterm of applications (implicit or explicit applications in the case of decorated terms).

As in the traditional presentation of Pure Type Systems [1], we define two sets **Axiom** $\subset \mathcal{S}^2$ and **Rule** $\subset \mathcal{S}^3$ by

$$\begin{aligned} \textbf{Axiom} &= \{(\mathsf{Prop}, \mathsf{Type}_0);\ (\mathsf{Type}_i, \mathsf{Type}_{i+1}) \mid i \in \mathbb{N}\} \\ \textbf{Rule} &= \{(\mathsf{Prop}, s, s);\ (s, \mathsf{Prop}, \mathsf{Prop}) \mid s \in \mathcal{S}\} \\ &\quad \cup \{(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{max(i,j)}) \mid i, j, \in \mathbb{N}\} \end{aligned}$$

We will also consider these two judgements:

- the judgement $\Gamma \vdash$ that means "the context Γ is well-formed"
- the judgement $\Gamma \vdash M : T$ that means "under the context Γ, the term M has type T". By convention, we will implicitly α-convert M in order that $\mathsf{DV}(\Gamma)$ and the set of bound variables of M are disjoint.

Definition 1 (Typing judgements). *They are defined in Fig. 3.*

They are very similar to the rules of a standard Calculus of Constructions where product, abstraction and application are duplicated into explicit and implicit ones. There are though two important differences:

[2] For instance, $(\lambda[x{:}T].\, x)^*$ depends on the name of the binder.

$$\frac{}{[] \vdash}\,(\textsc{Wf-E}) \qquad \frac{\Gamma \vdash T : s \quad x \notin \mathsf{DV}(\Gamma)}{\Gamma; x : T \vdash}\,(\textsc{Wf-S})$$

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

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

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

$$\frac{\Gamma; x : T \vdash M : U \quad \Gamma \vdash \Pi(x{:}T).\, U : s}{\Gamma \vdash \lambda(x{:}T).\, M : \Pi(x{:}T).\, U}\,(\textsc{E-Lam})$$

$$\frac{\Gamma; x : T \vdash M : U \quad \Gamma \vdash \Pi[x{:}T].\, U : s \quad x \notin \mathsf{FV}(M^*)}{\Gamma \vdash \lambda[x{:}T].\, M : \Pi[x{:}T].\, U}\,(\textbf{I-Lam})$$

$$\frac{\Gamma \vdash M : \Pi(x{:}T).\, U \quad \Gamma \vdash N : T}{\Gamma \vdash M\, N : U\{x/N\}}\,(\textsc{E-App}) \qquad \frac{\Gamma \vdash M : \Pi[x{:}T].\, U \quad \Gamma \vdash N : T}{\Gamma \vdash M[N] : U\{x/N\}}\,(\textsc{I-App})$$

$$\frac{\Gamma \vdash M : T \quad \Gamma \vdash T' : s \quad T^* \cong_{\beta\eta} T'^*}{\Gamma \vdash M : T'}\,(\textsc{Conv})$$

Fig. 3. Typing rules of ICC*

- in the **(I-Lam)** rule, we add the condition $x \notin \mathsf{FV}(M^*)$ (variables introduced by an implicit abstraction cannot appear in the extraction of the body), so x is not used during the computation. This excludes meaningless terms like $\lambda[x{:}T].\, x$.
- In the **(Conv)** rule we replace the usual conversion by the conversion of extracted terms. This makes it clear that the denotation of objects do not depend on implicit information.

This last modification completely changes the semantics of the formalism. Despite of being apparently very close to the Calculus of Constructions, it is in fact semantically much closer to ICC (usual models of CC do not validate the (Conv) rule).

Before developing the metatheory of our system, we shall make the subset of ICC we are targeting more precise.

Definition 2 (Typing rules of ICC$^-$). *See Fig.4.*

In comparison to ICC as presented in [10], we made the following restrictions:

- we removed the rules related to subtyping (rules Cum and Ext), to make things simpler, but we consider extending our system with these rules (or equivalent ones);
- we also removed the context strengthening rule (Str) for quite different reasons. In our formalism, non-dependent implicit products are intended to encode preconditions of a program: a proof of $\Pi[\_ : P].\, Q$ yields a program with specification

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

$$\frac{\Gamma \vdash_{\text{ICC}^-} \quad (s_1, s_2) \in \textbf{Axiom}}{\Gamma \vdash_{\text{ICC}^-} s_1 : s_2} (\textsc{Sort}) \qquad \frac{\Gamma \vdash_{\text{ICC}^-} \quad (x : T) \in \Gamma}{\Gamma \vdash_{\text{ICC}^-} x : T} (\textsc{Var})$$

$$\frac{\Gamma \vdash_{\text{ICC}^-} T : s_1 \quad \Gamma; x{:}T \vdash_{\text{ICC}^-} U : s_2 \quad (s_1, s_2, s_3) \in \textbf{Rule}}{\Gamma \vdash_{\text{ICC}^-} \Pi(x{:}T).U : s_3 \qquad \Gamma \vdash_{\text{ICC}^-} \forall(x{:}T).U : s_3} (\textsc{ExpProd})\&(\textsc{ImpProd})$$

$$\frac{\Gamma; x{:}T \vdash_{\text{ICC}^-} M : U \quad \Gamma \vdash_{\text{ICC}^-} \Pi(x{:}T).U : s}{\Gamma \vdash_{\text{ICC}^-} \lambda x.M : \Pi(x{:}T).U} (\textsc{Lam})$$

$$\frac{\Gamma \vdash_{\text{ICC}^-} M : \Pi(x{:}T).U \quad \Gamma \vdash_{\text{ICC}^-} N : T}{\Gamma \vdash_{\text{ICC}^-} M\,N : U\{x/N\}} (\textsc{App})$$

$$\frac{\Gamma; x{:}T \vdash_{\text{ICC}^-} M : U \quad \Gamma \vdash_{\text{ICC}^-} \forall(x{:}T).U : s \quad x \notin \mathsf{FV}(M)}{\Gamma \vdash_{\text{ICC}^-} M : \forall(x{:}T).U} (\textsc{Gen})$$

$$\frac{\Gamma \vdash_{\text{ICC}^-} M : \forall(x{:}T).U \quad \Gamma \vdash_{\text{ICC}^-} N : T}{\Gamma \vdash_{\text{ICC}^-} M : U\{x/N\}} (\textsc{Inst})$$

$$\frac{\Gamma \vdash_{\text{ICC}^-} M : T \quad \Gamma \vdash_{\text{ICC}^-} T' : s \quad T \cong_{\beta\eta} T'}{\Gamma \vdash_{\text{ICC}^-} M : T'} (\textsc{Conv})$$

Fig. 4. Typing rules of ICC^-

Q provided you can produce a proof of P; but the strengthening rule makes this type equivalent to Q. So this rule would not require a proof of P prior to using a program with specification Q.

3 Metatheory

Unlike ICC, the basic metatheory can be proven just like for PTSs (see for instance [1]). This is due to the fact that it relies heavily on the form of the typing rules, but very little on the nature of conversion. We first prove inversion lemmas. They allow us to characterize the type R of judgment $\Gamma \vdash M : R$ according to the nature of the term M. Other important properties are substitutivity and context conversion (if $\Gamma \vdash M : T$, $\Delta \vdash$ and $\Gamma^* \cong_{\beta\eta} \Delta^*$ hold, then we have $\Delta \vdash M : T$).

3.1 Preservation of the Theory and Consistency

In this section, we prove that ICC^* is consistent and that it is equivalent to ICC^- (any derivation in ICC^* has a counterpart in ICC^- and vice versa). First, we prove by mutual structural induction that any derivation of ICC^* can be mapped into a derivation in ICC^-:

Proposition 1 (Soundness of extraction)

$$\begin{array}{ll} (i) & \Gamma \vdash \quad \Rightarrow \quad \Gamma^* \vdash_{ICC^-} \\ (ii) & \Gamma \vdash M : T \Rightarrow \Gamma^* \vdash_{ICC^-} M^* : T^* \end{array}$$

Consistency of ICC* is an easy consequence of consistency of ICC$^-$ [11] and of Proposition 1.

Proposition 2 (Consistency). *There is no proof of the absurd proposition. There exists no term M such that the following judgment is derivable:*

$$[] \vdash M : \Pi(A{:}\mathsf{Prop}).\, A.$$

If we cannot prove the completeness results for the full ICC (subtyping and strengthening are not derivable), we can still prove it for the restricted system ICC$^-$ using mutual structural induction and context conversion.

Proposition 3 (Completeness of extraction)

1. *For any judgement $\Gamma \vdash_{ICC^-}$ there exists a context Δ in ICC* such that $\Delta^* = \Gamma \ \wedge \ \Delta \vdash$.*
2. *For any judgement $\Gamma \vdash_{ICC^-} M : T$ there exists Δ, N and U such that $\Delta \vdash N : U \ \wedge \ \Delta^* = \Gamma \ \wedge \ N^* = M \ \wedge \ U^* = T$*

Note that since we have completeness for a subset of ICC rules and not for ICC itself, we cannot deduce properties of ICC* from properties of ICC.

3.2 Decidability of Type Inference

As usual, decidability of type-checking requires the decidability of type inference. In our case, we can consider two kinds of type inference: inferring a decorated term or a term of ICC$^-$. The latter is enough for decidability of type-checking, but for implementation purposes, it is desirable to have the former. We want then to prove the following:

Proposition 4 (Decidability of type inference). *There exists a sound and complete algorithm that receives as an input a well-founded context Γ and a term M and returns a decorated term T such that $\Gamma \vdash M : T$ if there exists one and* `false` *otherwise.*

Proof. The cases of variable and sort are trivial. For products and abstractions we can conclude easily because ICC is strongly normalizing (see [11]) and because we can infer a sort s from $s^*(= s)$.

For applications it is much trickier because we need precise information to infer the type. If we consider *e.g.* an implicit application $M[N]$, infering the type of M and N is not enough. If we know that M has type T', we have to reach, if it exists, to the *annotated* term U such that $T'^* \cong_{\beta\eta} \forall(x : T).U^*$. The problem is that we only have access to the extracted term U^*.

In order to solve this we need to introduce some reduction rules to ICC* terms. The key point is that we only need to do head reduction : if T' is reduced to a product, this product can let us infer the type of the application. Since for well-typed terms, we

cannot have $(\lambda(x:T).\,Mx) \rhd_\eta M$ with M being a product, η-reduction rules are not useful and thus we only introduce β-reduction rules.

These rules and two basic properties are presented in Fig.5. Note that these rules are not necessary for the definition of our calculus. We only need them to infer types.

We also have a completeness result saying that, given M a well-typed term of ICC* and N' a term of ICC such that $M^* \rightarrow_\beta N'$, there exists a term N of ICC* such that $N^* = N'$ and $M \rightarrow^+_{\beta_{ie}} N$. It is proved as in Lemma 3.4 of [9] except that we use the fact that every well-typed term has a β_i-weak head normal form (WHNF) - instead of a β_i-normal form in [9].

The last result that we need is the existence of a β_{ie}-WHNF for every well-typed term of ICC*. This is a consequence of the soundness and completeness of β_{ie}-reduction and of the existence of a β_i-WHNF for every well-typed term of ICC*.

We can now infer the type of an application MN or $M[N]$. We compute the β_{ie}-WHNF of the type of M. If it is a product then M is typed by this product (soundness and subject reduction of β_{ie}) and we can infer the type of the application. If not, completeness informs us that the application is not well-typed.

Definitions	$(\lambda(x:T).\,M)\,N$	$\rhd_{\beta_e}$	$M\{x/N\}$	**(explicit β-reduction)**
	$(\lambda[x:T].\,M)\,[N]$	$\rhd_{\beta_i}$	$M\{x/N\}$	**(implicit β-reduction)**
		$\rhd_{\beta_{ie}}$ $=$	$\rhd_{\beta_e} \cup \rhd_{\beta_i}$	**(β_{ie}-reduction)**
Properties	$(\Gamma \vdash M : T) \wedge (M \rightarrow_{\beta_{ie}} N)$	$\Rightarrow$	$M^* \rightarrow^*_\beta N^*$	**(Soundness)**
	$(\Gamma \vdash M : T) \wedge (M \rightarrow^*_{\beta_{ie}} M')$	$\Rightarrow$	$\Gamma \vdash M' : T$	**(Subject Reduction)**

Fig. 5. β_{ie}-reduction in ICC*

4 Implementation and Inductive Types

Our long term goal is to design a formalism dedicated to software verification. To experiment on examples, we have developped a clone of Coq that implements ICC*.[3] Despite the fact that the formalism is quite different from the one implemented by Coq (the underlying models are completely different), the cost of its implementation was amazingly low: the type of terms has been slightly changed (to duplicate product, abstraction and application into implicit/explicit pairs). But then the code is adapted straightforwardly since the type-checking algorithm is modular w.r.t. conversion.

Our prototype inherits inductive types from Coq, but this is considered an experimental feature. We proceeded by analogy with their impredicative encoding. Of course, inductive types are more expressive: strong and dependent elimination schemes cannot be derived with the impredicative encodings. The most challenging task part of justifying our implementation of inductive types is probably to introduce an implicit sigma type $\Sigma[x:A].\,B_x$ that should be interpreted as a union of the B_x family of types. It is not clear how to define union of types in Miquel's model.

[3] This implementation is available as a *Darcs* repository at
`http://www.lix.polytechnique.fr/Labo/Bruno.Barras/coq-implicit`

In the following examples, we will rather use impredicative encodings and axioms that correspond to derivable properties of inductive types. Among those are:

- the strong elimination of absurdity $\texttt{False\_elim} : \Pi[P:\mathsf{Type}].\,[\texttt{False}] \to P$, [4] which is used to mark parts of a program as dead code,
- the strong elimination of equality proofs $\texttt{eq\_elim} : \Pi[A]\,[x]\,[y]\,[P:A \to \mathsf{Type}].\,[x = y] \to P\,x \to P\,y$, which is used to "cast" an expression of type $P\,x$ with type $P\,y$ whenever we can prove $x = y$,
- and a strong elimination of accessibility proofs that would allow building recursive functions whose termination is justified by a logical (implicit) proof of well-foundedness, but this is not used in this work.

The former is easy to validate since the denotation of `False` is empty, $[\texttt{False}] \to P$ is the full interpretation domain. `eq_elim` can be interpreted by the denotation of $\lambda x.\,x$: if x and y are equal (in A), then a predicate (of domain A) cannot distinguish them, so $P\,x$ and $P\,y$ are the same type, and finally the identity is actually a mapping from $P\,x$ to $P\,y$.[5]

In the implementation, the above properties result from two principles: (dependent) pattern-matching on the proof object (which would require this proof to be present at runtime), and the following axiom:

$$\texttt{impl\_PI} : \Pi[P:\mathsf{Prop}].\,[P] \to P.$$

Informally it says that there exists an object that belongs to any provable proposition, which holds in the model. This is weaker than the usual axiom of proof irrelevence which says that there exists only one proof object. Thanks to that axiom, logical arguments of programs can always be made implicit (as for `False_elim` and `eq_elim`) and thus never compared. Note that this was an important motivation for considering a proof irrelevant Calculus of Constructions [18].

One might want not just axiom `eq_elim`, but also a reduction associated to it (witnessing the fact that it can be interpreted by the identity). Unfortunately we see no way to do this since this axiom should decide if reduction is possible without any information about x, y or the proof of $x = y$ (they are all implicit objects). Nonetheless, it is possible to add a variant of Streicher's K axiom (justified by the model):

$$\Pi[A]\,[x]\,[P]\,[e:x = x][a:P\,x].\,\texttt{eq\_elim}\,[A]\,[x]\,[x]\,[P]\,[e]\,a = a.$$

5 Examples

In this section, we develop several examples that illustrate the features of our formalism. We first show how preconditions and postconditions can be encoded in ICC* by defining a simple division algorithm. Then, we define concatenation and head of vectors, the latter illustrate how to deal with absurd cases. Such examples are direct translations

[4] We do not to write type of variables when it can be easily inferred from the context, and notation $[A] \to B$ stands for $\Pi[\_:A].\,B$.

[5] Thanks to Alexandre Miquel for pointing out this fact.

of what could already be done in the Calculus of Constructions, but our claim is that ICC provides a better framework. Next examples show features that depart from CC: PVS' predicate subtyping can be encoded faithfully, and under some circumstances, datastructures carrying dependencies do not have to be copied.

5.1 Euclidean Division

This example illustrates how to encode programs with preconditions and postconditions. Euclidean division can be expressed as a primitive recursive function, following this informal algorithm:

$$\begin{array}{l}\mathtt{div}\ a\ b := \mathtt{if}\ a = 0\ \mathtt{then}\ (0,0)\ \mathtt{else}\\ \qquad\qquad \mathtt{let}\ (q,r) := \mathtt{div}\ (a-1)\ b \mathtt{in}\\ \qquad\qquad \mathtt{if}\ r = b-1\ \mathtt{then}\ (q+1,0)\ \mathtt{else}\ (q,r+1)\end{array}$$

$$\begin{array}{rl}\mathtt{nat\_elim}: & \Pi(n{:}\mathtt{nat})\,[P{:}\mathtt{nat}\to\mathsf{Prop}].\,P\,0\to(\Pi k.\,P\,k\to P\,(S\,k))\to P\,n\\ \mathtt{eq\_nat\_elim}: & \Pi m\,n\,[P{:}\mathsf{Prop}].\,(\Pi[H{:}m=n].\,P)\to(\Pi[H{:}m\neq n].\,P)\to P\\ \mathtt{diveucl}: & \mathtt{nat}\to\mathtt{nat}\to\mathsf{Prop}\\ \mathtt{div\_intro}: & \Pi[a]\,[b]\,q\,r\,[H{:}a=bq+r\wedge r<b].\,\mathtt{diveucl}\,a\,b\\ \mathtt{div\_elim}: & \Pi[a]\,[b]\,[P{:}\mathsf{Prop}].\,\mathtt{diveucl}\,a\,b\to(\Pi q\,r\,[H{:}a=bq+r\wedge r<b].\,P)\to P\end{array}$$

$$\begin{array}{l}\mathtt{div} := \lambda(a\,b{:}\mathtt{nat})\,[H{:}b<>0].\\ \quad \mathtt{nat\_elim}\,a\,[\lambda a.\,\mathtt{diveucl}\,a\,b]\\ \quad (\mathtt{div\_intro}\,[0]\,[b]\,0\,0\,[\pi_1\,H])\\ \quad (\lambda k\,(div_k{:}\mathtt{diveucl}\,k\,b).\\ \quad\ \mathtt{div\_elim}\,[k]\,[b]\,div_k\,[\mathtt{diveucl}\,(S\,k)\,b]\\ \quad\ (\lambda q\,r\,[H_0].\\ \quad\ \mathtt{eq\_nat\_elim}\,r\,(b-1)\,[\mathtt{diveucl}\,(S\,k)\,b]\\ \quad\ (\lambda[H_1].\,\mathtt{div\_intro}\,[S\,k]\,[b]\,(S\,q)\,0\,[\pi_2\,H_0\,H_1])\\ \quad\ (\lambda[H_1].\,\mathtt{div\_intro}\,[S\,k]\,[b]\,q\,(S\,r)\,[\pi_3\,H_0\,H_1])))\end{array}$$

where:

$$\begin{array}{rl}\pi_1: & b\neq 0\to 0=b.0+0\wedge 0<b\\ \pi_2: & k=bq+r\wedge r<b\to r=b-1\to Sk=b(q+1)+0\wedge 0<b\\ \pi_3: & k=bq+r\wedge r<b\to r\neq b-1\to Sk=bq+(Sr)\wedge Sr<b\end{array}$$

Fig. 6. Euclidean division

One would like to specifiy that if b is not 0 (precondition), then $\mathtt{div}\,a\,b$ returns a pair (q,r) such that $a = bq + r \wedge r < b$ (α) (postcondition). To express the result, we define a type `diveucl`, parameterized by a and b that encodes pairs (q,r) that satisfy (α). More precisely, it is a triple made of 2 explicit components of type `nat` and an implicit proof of (α) (this is an instance of the predicate subtyping scheme, section 5.3). See Fig.6 for the types of the introduction and elimination rules. We can see that the introduction rule has only 2 explicit arguments, so it will behave (w.r.t. conversion) as a pair of numbers. Then, the division program can be specified by type

$$\Pi a\,b\,[H{:}b\neq 0].\,\mathtt{diveucl}\,a\,b.$$

The program can then be written without difficulty (Fig.6) by adapting the proof made in the Calculus of Constructions, assuming `nat_elim` to define programs by recursion on a natural number and `eq_nat_elim` to decide if two numbers are equal.

The point of using ICC* is that `div` actually behaves as the informal algorithm. Within CC, `div` is a function of arity 3 returning a triple. So if we have two distinct proofs P_1 and P_2 of $b \neq 0$, $(\texttt{div}\, a\, b\, P_1)$ and $(\texttt{div}\, a\, b\, P_2)$ reduce to triples $(q, r, f(P_1))$ and $(q, r, f(P2))$ respectively, for some f. The two proofs $f(P_1)$ and $f(P_2)$ of the postcondition (α) have no particular reason to be equal.

Not only this is solved by ICC*, but furthermore, $(\texttt{div}\ 17\ 5\ [P_3])$ is convertible to $(\texttt{div}\ 11\ 3\ [P_4])$ since the quotient and remainder of both divisions are equal. This is not the case in CC (even assuming proof-irrelevance) since `div_intro` also depends on the inputs a and b.

5.2 Vectors

Miquel showed how lists and vectors can be encoded in ICC [10]. We adapt his example to ICC* (see Fig.7). It consists in merging definitions of vectors in ICC and CC: the definitions of CC have to be decorated with implicit/explicit flags as in ICC. Note that `vect` and P are explicit functions since we want to distinguish vectors of different lengths. But P itself is implicit since it is only used to type the other two arguments, which correspond to constructors.

$$
\begin{aligned}
\texttt{vect} &:= \lambda m.\, \Pi[P{:}\texttt{nat} \to \mathsf{Prop}].\, P\ 0 \to (\Pi[n].\, A \to P\ n \to P\ (S\ n)) \to P\ m \\
\texttt{nil} &:= \lambda[P]\ f\ g.\, f \\
\texttt{cons}\ [n]\ x\ v &:= \lambda[P]\ f\ g.\, g\ [n]\ x\ (v\ [P]\ f\ g)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{append}: &\quad \Pi[n_1]\,[n_2].\, \texttt{vect}\ n_1 \to \texttt{vect}\ n_2 \to \texttt{vect}\ (n_1 + n_2) \\
&:= \lambda[n_1]\,[n_2]\, v_1\, v_2.\, v_1\ [\lambda n'.\, \texttt{vect}\ (n' + n_2)]\ v_2\ (\lambda[n']\, x\, v'.\, \texttt{cons}\ [n' + n_2]\, x\, v') \\
\texttt{head}: &\quad \Pi[n].\, \texttt{vect}\ (\texttt{S}\ n) \to A \\
&:= \lambda[n]\, v.\, v\ [\lambda k.\, [k = \texttt{S}\ n] \to A] \\
&\qquad (\lambda[H{:}0 = \texttt{S}\ n].\, \texttt{False\_elim}\ [A]\ [\texttt{discr}\ n\ H]) \\
&\qquad (\lambda[k]\, x\, y\, [\_{:}\texttt{S}\ k = \texttt{S}\ n].\, x) \\
&\qquad [\texttt{refl}\ [\texttt{S}\ n]]
\end{aligned}
$$

Fig. 7. Vectors

The reader can check that by extraction towards ICC, these definitions becomes strictly those for the untyped λ-calculus:

$$\texttt{nil}^* = \lambda f\ g.\, f \qquad \texttt{cons}^* = \lambda x\, v\, f\, g.\, g\ x\ (v\ f\ g)$$

Here, $\texttt{cons}^*$ has arity 2 (if we consider it returns vectors), and there is no extra argument P.

Concatenation of two vectors can be expressed easily if we assume that addition satisfies $0 + n \cong_\beta n$ and $(S\, n) + m \cong_\beta S\, (n + m)$, which is the case for the usual impredicative encoding of natural numbers.

In the Calculus of Constructions, computing the length of a vector is useless since a vector always comes along with its length (either as an extra argument or because it

is fixed). In ICC*, when we decide to make the length argument implicit, it cannot be used in the explicit part of the program. For instance, it is illegal to define the length of a vector as $\lambda[n]\,(v{:}\mathtt{vect}\,n).\,n$. It has to be defined as recursive function that examines the full vector.

We hope we made clear that in many situations, the Implicit Calculus of Constructions allows to have the safety of a dependently typed λ-calculus, but all the typing information (here P and the vector size) does not get in the way, since objects are compared modulo extraction.

5.3 Predicate Subtyping a la PVS

One key feature of PVS [14] is predicate subtyping, which corresponds to the comprehension axiom in set theory. For instance, one can define the type of even number as a subtype of the natural numbers satisfying the appropriate predicate. When a number is claimed to have this type, it has to be proven it is actually even, and a type-checking condition (TCC) is generated.

In the Calculus of Constructions, this can be encoded as a dependent pair formed by a natural number and a proof object that it is actually even. The coercion from even numbers to numbers is simply the first projection. This encoding is not faithfull since it might happen that there exists two distinct proofs[6] (let us call them π_1 and π_2) of the fact that, say, 4 is even. Then $(4,\pi_1)$ and $(4,\pi_2)$ are distinct inhabitants of the type of even numbers while they represent the same number. In ICC*, this issue can be avoided by making the second component of the pair implicit.[7] Let us define `Even` as:

$$\Pi[P{:}\mathsf{Prop}].\,(\Pi(n{:}\mathtt{nat})\,[H{:}\mathtt{even}\,n].\,P) \to P$$

Coercion can then be defined as the first projection:

$$\lambda(x{:}\mathtt{Even}).\,x\,[\mathtt{nat}]\,(\lambda n\,[H].\,n) : \mathtt{Even} \to \mathtt{nat}$$

and equalities such as $(4,\pi_1) = (4,\pi_2)$ are proven by reflexivity.

These facts generalize to any predicate over any type since no particular property of natural numbers has been used here.

In [16], Sozeau introduces a feature of Coq similar to the predicate subtyping of PVS, including the possibility to prove claimed invariants by generating proof obligations. Subset types are coded by a pair of an object and a proof of the claimed invariants (e.g. being even). Proof obligations are metavariables that the user must instantiate. This method relies on the fact that programs shall never try to access the proof object, to avoid the issue above mentioned. Combining ICC and his method seems to solve the problem.

5.4 Subtyping Coercions

In ICC, vectors are subtypes of lists. Here, we have no subtyping but we can easily write a coercion from vectors to lists (defined as $\Pi[P{:}\mathsf{Prop}].\,P \to (A \to P \to P) \to P$):

[6] Proof-irrelevance is not provable in CC, nor in Coq.

[7] Note that this does not work in ICC because of the strengthening rule.

$$\lambda(v{:}\mathtt{vect}\ n)\,[P]\,f\,g.\,v\,[\lambda\_.\,P]\,f\,\lambda[n]\,x\,v.\,g\,x\,v \quad : \quad \mathtt{vect}\ n \to \mathtt{list}$$

Remark that the extraction of this coercion is $\lambda v\ f\ g.v\ f\ (\lambda x\,v.g\,x\,v)$, which η-reduces to the identity.

Another illustration of the expressiveness of ICC is the example of terms indexed by the set of free variables (`var` denotes the type used to represent variables):

$$\begin{aligned}\mathtt{term} := \lambda(s{:}\mathtt{var} \to \mathsf{Prop}).\ &\Pi[P{:}\mathsf{Prop}].\\ &(\Pi(x{:}\mathtt{var}).\,[s\,x] \to P) \to\\ &(P \to P \to P) \to P\end{aligned}$$

It corresponds to a type with two constructors:

$$\begin{aligned}&\mathtt{Var} : \Pi[s{:}\mathtt{var} \to \mathsf{Prop}]\,(x{:}\mathtt{var}).\,[s\,x] \to \mathtt{term}\,s\\ &\mathtt{App} : \Pi[s{:}\mathtt{var} \to \mathsf{Prop}].\,\mathtt{term}\,s \to \mathtt{term}\,s \to \mathtt{term}\,s.\end{aligned}$$

If set s is included in set s' (i.e. there is a proof h of $\Pi x.\,s\,x \to s'\,x$), then any term of $\mathtt{term}\,s$ can be seen as a term of $\mathtt{term}\,s'$. This can be done by the following function that recursively goes through the term to update the proofs:

$$\begin{aligned}\mathtt{lift} :\ & \mathtt{term}\,s \to \mathtt{term}\,s'\\ &:= \lambda t\,[P]\,f\,g.\,t\,[P]\,(\lambda x\,[H].\,f\,x\,[h\,x\,H])\,(\lambda t_1\,t_2.\,g\,t_1\,t_2)\end{aligned}$$

We can notice that, as previously, the extraction of this term η-reduces to the identity.

It seems natural to introduce a notion of *coercion*: terms which extraction reduces to the identity. In cases where we manipulate extracted terms (for instance in the conversion test), then coercions can be dropped. On the other hand, for reduction of decorated terms, coercions can be used to update the implicit subterms of its argument. This duality between annotated and extracted terms forms the most striking feature of ICC*. The current implementation does not optimize coercions yet.

6 Related Works

Epigram. In comparison with our system, Epigram focuses on implementing inductive types in a more powerful way, and also provides more natural properties for equality. On the other hand, the theory behind is not so different from usual Type Theories (in the tradition of Martin Löf theories). A number of techniques are developed in order to optimize the evaluation:

- in the case of vectors, the length argument can be removed from the constructor [3]. Unfortunately, only information that is uniquely recoverable can be erased. For instance, in the example of terms with their set of variables, the proof that variables belong to the set cannot be made implicit.
- A notion of compilation stages is introduced and an erasure function removes parts that belong to the most "static" stages [2]. However, this process faces the same problems that the extraction of Coq: the conversion rule applies on the fully decorated term, so the erased parts are still compared.

We emphasize on the fact that ICC (and ICC*) have a very powerful conversion rule where logical information is actually irrelevant.

Proof-irrelevant Calculus of Constructions. Werner [18] introduces a variant of the Calculus of Constructions where objects of the Prop kind can be erased. His idea is very similar to ours since he modifies conversion so that proofs of a given proposition are always convertible. On the one hand, this does not require a complicated model since proof-irrelevance is a valid property in the classical model of the Calculus of Constructions [12]. On the other hand, this approach does not address the problem of other extra arguments (types, domains of abstractions, dependencies belonging to Set).

7 Future Work

We have shown that the Implicit Calculus of Constructions provides a simple yet powerful way to write dependently typed programs and proof-carrying programs where specifications and proofs do not interfere with the computational content. However there are still aspects that are not completely satisfactory.

Inductive Types. The key one is to extend the model of ICC to inductive types. As already mentionned, the difficult part is understanding how to support union types in the setting of coherent spaces.

Once the theory is set up, it is very desirable to have a powerful case-analysis operator as in Epigram or Agda. This can save awkward manipulations of equality proofs as in the vector `head` example.

Other programming paradigms. ATS can deal with a large variety of aspects: side-effects, non-termination, memory allocation, and more. This is definitely a must have if we expect dependent types to reach a larger audience.

8 Conclusion

We have shown how a restriction of the Implicit Calculus of Constructions can be turned into an implementable system, and we developed several typical examples. We shall stress on the fact that some problems seem less difficult to deal with than in most type systems implemented so far. Moreover, from the Coq user point of view, there are only a few changes, and the new features can be learnt quickly.

References

1. Barendregt, H.: Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen. In: Handbook of Logic in Computer Science, vol. II (1991)
2. Brady, E.: Practical Implementation of a Dependently Typed Functional Programming Language. PhD thesis, Durham University (2005)
3. Brady, E., McBride, C., McKinna, J.: Inductive families need not store their indices. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 115–129. Springer, Heidelberg (2004)
4. Chen, C., Xi, H.: Combining Programming with Theorem Proving. In: Proceedings of the tenth ACM SIGPLAN ICFP, Tallinn, Estonia, pp. 66–77 (September 2005)

5. Constable, R.L., Allen, S.F., Bromley, H.M., Cleaveland, W.R., Cremer, J.F., Harper, R.W., Howe, D.J., Knoblock, T.B., Mendler, N.P., Panangaden, P., Sasaki, J.T., Smith, S.F.: Implementing Mathematics with the Nuprl Development System. Prentice-Hall, NJ (1986)
6. Letouzey, P.: Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq. PhD thesis, Université Paris-Sud (July 2004)
7. Luo, Z.: An Extended Calculus of Constructions. PhD thesis, University of Edinburgh (1990)
8. McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming 14(1), 69–111 (2004)
9. Miquel, A.: Arguments implicites dans le calcul des constructions: étude d'un formalisme à la curry. Master's thesis, University Paris 7 (1998)
10. Miquel, A.: The implicit calculus of constructions. Extending pure type systems with an intersection type binder and subtyping. In: Abramsky, S. (ed.) TLCA 2001. LNCS, vol. 2044, Springer, Heidelberg (2001)
11. Miquel, A.: Le Calcul des Constructions implicite: syntaxe et sémantique. PhD thesis, Université Paris 7 (December 2001)
12. Miquel, A., Werner, B.: The not so simple proof-irrelevent model of CC. In: Geuvers, H., Wiedijk, F. (eds.) TYPES 2002. LNCS, vol. 2646, Springer, Heidelberg (2003)
13. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September 2007)
14. Owre, S., Shankar, N.: The formal semantics of PVS. Technical Report SRI-CSL-97-2, Menlo Park, CA (1997)
15. Prost, F.: Interprétation de l'analyse statique en théorie des types. PhD thesis, École Normale Supérieure de Lyon (December 1999)
16. Sozeau, M.: Subset coercions in Coq. In: Altenkirch, T., McBride, C. (eds.) TYPES 2006. LNCS, vol. 4502, Springer, Heidelberg (2007)
17. The Coq development team. The coq proof assistant reference manual v8.0. Technical report, INRIA, France, mars (2004), `http://coq.inria.fr/doc/main.html`
18. Werner, B.: On the strength of proof-irrelevant type theories. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 604–618. Springer, Heidelberg (2006)
19. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages, San Antonio, pp. 214–227 (January 1999)