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

Bruno Barras, Bruno Bernardo

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

Bruno Barras Bruno Bernardo
INRIA Futurs and LIX - LogiCal Project, Ecole polytechnique, France

E-mail: {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

Software verification is an ubiquitous problem. Typing discipline has 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 (avoiding access out of array bounds which would lead to runtime errors).

Such an issue can be delt with by using dependent types. Dependent ML [17] 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, etc.).

Much further in the direction of unrestricted dependent types, we find type systems based on the famous Curry-Howard correspondance that expresses the tight relation between programming and proving. In Martin Löf's Type Theory, types can depend not only on a restricted domain, but on arbitrary objects of the theory. Many actively developped proof systems are based on Type Theory, generally extended with primitive inductive datatypes, and other features sometimes borrowed from programming languages. We list only a small number of these: Epigram, Alf, NuPRL and Coq [15]. Ideas in this article apply directly to the latter, but we expect possible fallouts for other systems.

Paulin and Werner [13] show how Coq can be seen as a software verification tool. One typical example in programming with dependent types is that of 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 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`. $\texttt{vect}\ n$ is the type of vectors of length n (A is the type of the elements).

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

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 verbose: programs have arguments that are indeed only static information (type decorations, proof objects, dependencies). There exists a procedure called extraction (described in [4]) 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 similar 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.

The extraction approach of Coq has two main drawbacks. Firstly, since it is based on the Prop/Set dichotomy, it is not able to remove dependency arguments that belong to the Set sort (arguments n of `cons` and `get`). This could be improved by doing a dead-code analysis, but it would not allow the user to specify *a priori* arguments that shall not be used in the algorithmic part of the proof. Secondly, extraction is a tool 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 (see section 4.4 for a typical example), two datastructures may be equal, but containing different proofs. They could not be proven equal.

The Implicit Calculus of Constructions (ICC, see [9] and [10]) offers a more satifying alternative to the distinction between Prop and Set. It is [1] a Curry-style presentation of the Calculus of Constructions. It features an implicit product that correspond to an intersection type. It allows the quantification over a domain without introducing extra arguments. 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 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 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.

Definition 1 (Sorts) *The set of sorts is*
$\mathcal{S} ::= \mathsf{Prop} \mid \mathsf{Type}_i \quad (i \in \mathbb{N})$

Sorts $(\mathsf{Type}_i)_{i\in\mathbb{N}}$ denote the usual predicative universe hierarchy of the Extended Calculus of Constructions [5]. There is only one impredicative sort, Prop, because the distinction between propositional types and data types will be made when we define a term as being explicit (data types) or implicit (propositional types).

Definition 2 (Terms) *The syntax of terms is:*

$$\begin{aligned} M ::= {} & x \mid s \quad (s \in \mathcal{S}) \\ & \mid \Pi(x{:}M_1).\,M_2 \mid \lambda(x{:}M_1).\,M_2 \mid M_1\,M_2 \\ & \mid \Pi\{x{:}M_1\}.\,M_2 \mid \lambda\{x{:}M_1\}.\,M_2 \mid M_1\{M_2\} \end{aligned}$$

The second line gathers explicit constructions while the third one gathers implicit constructions. 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 ($T \to U$ stands for $\Pi(x:T).\,U$ when $x \notin \mathsf{FV}(U)$). Substitution of the free occurrences of variable x by N in term M is noted $M[x/N]$.

Definition 3 (Contexts) *Contexts are mappings from variables to types:*

$$\Gamma ::= [] \mid \Gamma; x : M$$

[1] *avant :* can be seen as

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 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.

Let us first recall the syntax of terms of ICC (see [10] or [9])

Definition 4 (ICC terms)

$$M ::= x \mid s \quad (s \in \mathcal{S}) \\ \mid \Pi(x\!:\!M_1).\, M_2 \mid \forall(x\!:\!M_1).M_2 \mid \lambda x.M \mid M_1\, M_2$$

We write $\rhd_\beta$ for the one step β-reduction on ICC terms. As a general rule, we write $\rightarrow_R^h$ the head reduction of $\rhd_R$ (for any relation R, e.g. β), so that reduction occurs in the left subterm of applications (implicit or explicit applications in the case of decorated terms). $\rightarrow_R$ denotes the contextual closure, $\rightarrow_R^*$ the reflexive transitive closure of $\rightarrow_R$, $\rightarrow_R^+$ the transitive closure of $\rightarrow_R$,and $\cong_R$ its reflexive, symmetric, transitive closure.

Definition 5 (Extraction)

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

Beware that extraction does not preserve α-conversion for any term. So, many properties of extraction will hold only for well-typed terms[2].

2.3 Typing rules

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

$$\begin{array}{rcl} \textbf{Axiom} & = & \{(\mathsf{Prop}, \mathsf{Type_0});\ (\mathsf{Type_i}, \mathsf{Type_{i+1}}) \mid i \in \mathbb{N}\} \\ \textbf{Rule} & = & \{(\mathsf{Prop}, s, s);\ (s, \mathsf{Prop}, \mathsf{Prop}) \mid s \in \mathcal{S}\} \\ & & \cup\{(\mathsf{Type_i}, \mathsf{Type_j}, \mathsf{Type}_{max(i,j)}) \mid i, j, \in \mathbb{N}\} \end{array}$$

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 6 (Typing judgements) *They are defined in figure 1.*

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:

- 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 terms like $\lambda\{x\!:\!T\}.\, x$.
- In the **(CONV)** rule we replace the usual conversion (it would be $\cong_{\beta_{ie}}$) by the conversion of extracted terms.

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 developping the metatheory of our system, we shall make some precisions on the subset of ICC we are targetting.

Definition 7 (Typing rules of ICC$^-$) *See figure 2.*

In comparison to ICC as presented in [9], we made the following restrictions:

- we removed η-reduction and 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 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 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 [2]). This is due to the fact that it relies heavily on the form of the typing rules, but very little

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

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

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

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

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

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

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

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

Figure 1. Typing rules of ICC*

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.

Lemma 1 (Substitutivity) *The following rule is derivable:*

$$\frac{\Gamma_1 \vdash M_0 : T_0 \qquad \Gamma_1; x_0 : T_0; \Gamma_2 \vdash M : T}{\Gamma_1; (\Gamma_2[x_0/M_0]) \vdash M[x_0/M_0] : T[x_0/M_0]}$$

Lemma 2 (Context conversion) *Typing is preserved by well-typed conversion in the context:*

$$\Gamma \vdash M : T \ \wedge\ \Gamma^* \cong_\beta \Delta^* \ \wedge\ \Delta \vdash \ \Rightarrow\ \Delta \vdash M : T$$

3.1 Preservation of the theory

In this section, we prove that ICC* is equivalent to ICC$^-$. That is, any derivation in ICC* has a counterpart in ICC$^-$ and vice versa. First, it is easy to show that any derivation of ICC* can be mapped into a derivation in ICC$^-$:

Proposition 1 (Soundness of extraction)

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

Proof. By mutual structural induction on the derivations of $\Gamma \vdash$ and $\Gamma \vdash M : T$. To each rule of ICC* correspond an equivalent rule of ICC$^-$. For rules (E-App) and (I-App), we use the substitutivity of extraction ($(M[x/N])^* = M^*[x/M'^*]$) which holds for well-typed terms.

Obviously, the completeness results regarding typing does not hold for the full ICC system, since the removed rules (subtyping and strengthening) are clearly not derivable. But we can show the completeness w.r.t. ICC$^-$.

Proposition 2 (Completeness of extraction) *For any derivable judgement $\Gamma \vdash_{\mathrm{ICC}^-} M : T$, there exists Δ, N and U such that*

$$\Delta \vdash N : U \ \wedge\ \Delta^* = \Gamma \ \wedge\ N^* = M \ \wedge\ U^* = T$$

Proof. This property relies mostly on the context conversion result. The proof is a bit tedious but is not difficult.

3.2 Consistency

Consistency of ICC* is an easy consequence of consistency of ICC$^-$ [10] and of Lemma 3 (Soundness of extraction).

Proposition 3 (Consistency) *There is no proof of the absurd proposition. There exists no term M such that the fol-*

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

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

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

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

$$\frac{\Gamma; x : T \vdash_{\mathrm{ICC}^-} M : U \quad \Gamma \vdash_{\mathrm{ICC}^-} \Pi(x{:}T).\, U : s}{\Gamma \vdash_{\mathrm{ICC}^-} \lambda x.M : \Pi(x{:}T).\, U} (\textsc{Lam}) \quad \frac{\Gamma \vdash_{\mathrm{ICC}^-} M : \Pi(x{:}T).\, U \quad \Gamma \vdash_{\mathrm{ICC}^-} N : T}{\Gamma \vdash_{\mathrm{ICC}^-} M\, N : U[x/N]} (\textsc{App})$$

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

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

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

Figure 2. Typing rules of ICC$^-$

lowing judgment is derivable:

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

3.3 Reduction rules for decorated terms

We consider two kinds of β-reduction, whether related to explicit or implicit terms.

It is important to note that these reduction rules are not necessary to the definition of our calculus. However, they play an important role for the decidability of type-checking since they provide a way to find a decorated and well-typed term in weak head normal form.

Definition 8 (β-reduction) *The explicit β-reduction or β_e-reduction, and the implicit β-reduction or β_i-reduction, are defined by*

$$\begin{array}{rcl} (\lambda(x\!:\!T).\, M)\, N & \rhd_{\beta_e} & M[x/N] \\ (\lambda\{x\!:\!T\}.\, M)\, \{N\} & \rhd_{\beta_i} & M[x/N] \end{array}$$

We define then the β-reduction by $\rhd_{\beta_{ie}} = \rhd_{\beta_e} \cup \rhd_{\beta_i}$.

We have the following result:

Lemma 3 (Soundness of the β_{ie}-reduction) *We have for well-typed terms:*

$$M \to_{\beta_{ie}} N \Rightarrow M^* \to_{\beta}^* N^*$$

Note that β_i-reductions are not translated: if $M \to_{\beta_i} N$, then $M^* = N^*$.

Proposition 4 (Subject reduction) *If $\Gamma \vdash M : T$ and $M \to_{\beta_{ie}}^* M'$ then we have $\Gamma \vdash M' : T$.*

The proof is quite long although not difficult. It follows the same pattern as in the standard Calculus of Constructions.

3.4 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 second one is enough for decidability of type-checking, but from implementation purposes, it is desirable to have the former. This raises the problem of inferring the type of an application. For this, we compute the weak head normal form (w.r.t. β_{ie}) of type of the function to check if it reduces to a product. If so, using subject reduction and soundness of β_{ie} we prove that the function is typed by the computed product. If not, we need a completeness result saying that if a type is equivalent to a product, then it β_{ie} reduces to a product.

We need some preliminaries in order to introduce a type inference algorithm.

Lemma 4 (Normalization of $\to_{\beta_i}^h$) *Let M be a well-typed term of ICC*. There exists a term N in β_i weak head normal form (WHNF) such that $M \to_{\beta_i}^* N$.*

Proof. Intuitively, this is because β_i substitutes variables introduced by an implicit abstraction, and such variables are not allowed to appear in the head of the term.

Lemma 5 (Completeness of β_{ie}-reduction) *Let M be a well-typed term of ICC* and N' be a term of ICC such that $M^* \to_{\beta} N'$. There exists a term N of ICC* such that $N^* = N'$ and $M \to_{\beta_{ie}}^+ N$.*

Proof. We proceed as in Lemma 3.4 of [8] except that we only need the fact that $\to_{\beta_i}^h$ terminates.

Proposition 5 (Existence of a β_{ie}-WHNF) *For every well-typed term M of ICC* there exists a term N in β_{ie}-WHNF such that $M \to_{\beta_{ie}}^* N$.*

Proof. Let N' be the normal form of M^*. (ICC is strongly normalizing cf [10]). By lemma 5, we have a term M_0 such that $M \to_{\beta_{ie}}^* M_0$ with $M_0^* = N'$. Then, by lemma 4, there is a term N in β_i-WHNF such that $M_0 \to_{\beta_i}^* N$ and $N^* = N'$ (lemma 3). N is also in β_e-WHNF otherwise its extracted term would not be in normal form. So N is a reduct of M in WHNF.

Proposition 6 (Decidability of type inference) *There exists a sound, complete and terminating type inference algorithm.*

This algorithm receives as an input a well-founded context Γ and a term M, and returns either `false` or a decorated term T such that $\Gamma \vdash M : T$.

Proof. The cases of variable and sort are trivial. Cases of abstractions and products are not very tough either because we can decide in ICC whether we have or not $T^* \cong_{\beta} s$. We already explained how to deal with the application cases.

4 Examples

Our long term goal is to design a formalism dedicated to software verification. In this section, we develop several examples that illustrate the features of our formalism such as programming with dependent types, datatype invariants, pre/post-conditions. Short of having datatypes as in the Calculus of Inductive Constructions, we use impredicative encodings.

In the following examples, when the type of variables is obvious, we shall not write them (as in $\lambda x\, \{y\}.\, \Pi z.\, M$).

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

$$
\begin{array}{rcl}
\texttt{append} & : & \Pi\{n_1\}\;\{n_2\}.\, \texttt{vect}\;n_1 \to \texttt{vect}\;n_2 \to \texttt{vect}\;(n_1 + n_2) \\
& := & \lambda\{n_1\}\;\{n_2\}\;v_1\;v_2.\, v_1\;\{\lambda n'.\, \texttt{vect}\;(n' + n_2)\}\;v_2\;(\lambda\{n'\}\;x\;v'.\, \texttt{cons}\;\{n' + n_2\}\;x\;v')
\end{array}
$$

Figure 3. Vectors

4.1 Vectors

Miquel showed how lists and vectors can be encoded in ICC [9]. We adapt his example to ICC* (see figure 3). 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.

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

$$
\begin{array}{rcl}
\texttt{nil}^* & = & \lambda f\; g.\, f \\
\texttt{cons}^* & = & \lambda x\; v\; f\; g.\, g\;x\;(v\;f\;g)
\end{array}
$$

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 $\lambda\{n\}\,(v : \texttt{vect}\;n).\, n$. 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.

4.2 Equality

As already shown by Miquel, Leibniz equality can be defined by impredicative encodings (see figure 4) in ICC, and adapting it to ICC* is straightforward. Let us remark that x is not needed to produce a proof of $x = x$. The elimination rule `eq_ind` allows to derive the usual theory of equality such as symmetry and transitivity, etc.

In the example of `append` above, we avoided many troubles because addition was defined a particular way. If it had been defined for instance by induction on the second argument, then types $\texttt{vect}\;n_2$ and $\texttt{vect}\;(0 + n_2)$ would not be convertible. However, it is possible to prove them equal, we should be able to use such a proof to "change" one type for the other. Definition `eq_ind` allows that, but it raises the issue that the equality proof has to be explicit. We would like to avoid this situation because it makes the computational behavior of the extracted term depend on logical proofs.

In order to fix this, we can think of having a stronger elimination principle `eq_ind2` where the equality proof is also implicit. In ICC, such axiom is problematic since it is equivalent to $\forall x\, y, x = y$, which is not inconsistent *per se*, but it is incompatible with most useful extensions, e.g. assuming $0 \neq 1$. Anyway, in the restricted calculus (ICC$^-$), this axiom is validated by the model of ICC$^-$ in Miquel's thesis. Furthermore, since the denotation of the type of `eq_ind2` contains the identity $\lambda x.\, x$, it is also valid to add axiom `eq_ind2_elim`.[3] There is, to our knowledge, no easy way to turn this equality into a reduction rule, since x, y and p are all implicit arguments, without losing the Church-Rosser property.

4.3 Predicate subtyping a la PVS

One key feature of PVS [12] is its allows 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 had 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

[3] Thanks to Alexandre Miquel for pointing out these two facts.

$$
\begin{array}{rcl}
\texttt{eq} & := & \lambda A\, x\, y.\, \Pi\{P : A \to \mathsf{Prop}\}.\, P\, x \to P\, y \\
\texttt{refl} & : & \Pi\{A\}\, \{x\}.\, \texttt{eq}\, A\, x\, x \\
 & := & \lambda\{A\}\, \{x\}\, \{P\}\, (H : P\, x).\, H : \\
\texttt{eq\_ind} & : & \Pi\{A\}\, \{x\}\, \{y\}\, (p : \texttt{eq}\, A\, x\, y)\, \{P : A \to \mathsf{Prop}\}.\, P\, x \to P\, y \\
 & := & \lambda(p : \texttt{eq}\, A\, x\, y)\, p\, \{P\}\, H.\, p\, \{P\}\, H \\
\texttt{eq\_ind2} & : & \Pi\{A\}\, \{x\}\, \{y\}\, \{p : \texttt{eq}\, A\, x\, y\}\, \{P : A \to \mathsf{Prop}\}.\, P\, x \to P\, y \\
\texttt{eq\_ind2\_elim} & : & \Pi\{A\}\, \{x\}\, \{p\}\, \{P\}\, a.\, \texttt{eq}\, (P x)\, (\texttt{eq\_ind2}\, \{A\}\, \{x\}\, \{x\}\, \{p\}\, \{P\}\, a)\, a
\end{array}
$$

Figure 4. Equality

distinct proofs[4] (let us call them π_1 and π_2) of the fact that, say, 4 is even. Then $(4, \pi_1)$ and $(4, \pi_2)$ and 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.[5] Let us define `Even` as:

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

The coercion can then be defined as the first projection:

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

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

4.4 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 $\Pi\{P : \mathsf{Prop}\}.\, P \to (A \to P \to P) \to P$):

$$\lambda(v : \texttt{vect}\ n)\, \{P\}\, f\, g.\, v\, \{\lambda\_.\, P\}\, f\, \lambda\{n\}\, x\, v.\, g x v$$

has type $\texttt{vect}\ n \to \texttt{list}$. 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 variables (`var` denotes the type used to represent variables):

$$
\begin{array}{rcl}
\texttt{term} & := & \lambda(s : \texttt{var} \to \mathsf{Prop}).\, \Pi\{P : \mathsf{Prop}\}. \\
 & & (\Pi(x : \texttt{var}).\, \{s\, x\} \to P) \to \\
 & & (P \to P \to P) \to P
\end{array}
$$

It corresponds to a type with two constructors: one of type $\Pi\{s : \texttt{var} \to \mathsf{Prop}\}\, (x : \texttt{var}).\, \{s\, x\} \to \texttt{term}\, s$ to build variables, and the second of type $\Pi\{s : \texttt{var} \to \mathsf{Prop}\}.\, \texttt{term}\, s \to \texttt{term}\, s \to \texttt{term}\, s$ to build applicative terms.

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 $\texttt{term}\, s$ can be seen as a term of $\texttt{term}\, s'$. This can be done by the following function that recursively goes through the term to update the proofs:

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

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

This is no coincidence since in ICC, vectors are a subtype of lists and $\texttt{term}\, s$ is a subtype of $\texttt{term}\, s'$, and Miquel proved that if M has type T in ICC, there is a derivation that there exists a term M' that (1) has type M without using rule (EXT) (thus there exists a term of ICC* that extracts to M') and (2) M' η-reduces to M.

We conclude that adding η-reduction to our system would be of great interest. It would be very 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.

4.5 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:

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

One would like to specifiy that if b is not 0 (precondition), then $\texttt{div}\, a\, b$ 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 instance of the predicate subtyping scheme, section 4.3). See figure 5 for the types of

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

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

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

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

where:

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

Figure 5. Euclidean division

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

$$\Pi a\,b\,\{H{:}b\neq 0\}.\,\texttt{diveucl}\,a\,b.$$

The program can then be written without difficulty (figure 5) 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, (`div` 17 5 $\{P_3\}$) is convertible to (`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 Ongoing and 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.

5.1 Heterogeneous equality

As often in type theory, equality is among the most difficult features to design. Leibniz equality allows to compare only objects of the same type, which is a bit restrictive since in our system, comparing objects of different types is not a concern: only the behavior of their extracted form matters. It would be nice to see how an heterogeneous equality can be integrated to ICC*, for instance to have elimination like `eq_ind2_elim`, but not restricted to reflexivity proofs.

McBride [6] has extensively studied an heterogeneous equality (he used to call John Major Equality), and he implemented it in Epigram [7].

5.2 Inductive Types

Most of the ideas extend straightforwardly to inductive types, by analogy with their impredicative encoding. Of course, inductive types are more expressive: injectivity and discrimination of constructors, dependent elimination schemes cannot be derived with the impredicative encodings. However, we believe that these principles do not conflict with the implicit arguments.

5.3 Implementation

We are implementing this system as an alternative version of Coq. Despite the fact that the formalism is quite different from the one implemented by Coq (the models are completely different), the cost of its implementation seems amazingly low: the type of terms has to be 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 almost the same.

6 Related works

Epigram

McBride and Alternkirch have designed the Observational Type Theory [1], and this theory is implemented in Epigram 2. Some efforts have been made to make the theory less verbose. In [3], it is shown how vector length can be removed from constructors. However, 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.

Proof-irrelevant Calculus of Constructions

Werner [16] 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 [11]. On the other hand, this approach does not address the problem of other extra arguments (types, domains of abstractions, dependencies belonging to Set).

Subset Types

In [14], Sozeau introduces a feature similar to the predicate subtyping of PVS. Like in this article, it is encoded thanks to a dependent pair made of an object and a proof that this object satisfies the predicate. In his method, he shows that proofs do not get in the way by forbiding writing programs that depend on the second part of the pair. The latter can be used only to build a proof appearing as a second part of another pair. However, it is not clear that his method would help hiding the proofs in the example of terms carrying their set of variables.

Anyway, he added to Coq a feature that allows a phase distinction between programming and proving properties, like PVS' TCC. Our approach could take advantage of it.

7 Conclusion

We have shown how a restriction of the Implicit Calculus of Constructions can be turned into a implementable system, and we developped 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. Now, it remains to see if we can capture a bit more of the expressiveness of ICC while keeping a decidable typechecking.

References

[1] T. Altenkirch and C. McBride. Towards observational type theory. Manuscript, available online, February 2006.

[2] H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

[3] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, Torino, 2003*, volume 3085 of *LNCS*, pages 115–129. Springer-Verlag, 2004.

[4] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

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

[6] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[7] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[8] A. Miquel. Arguments implicites dans le calcul des constructions: étude d'un formalisme à la curry. Master's thesis, University Paris 7, 1998.

[9] A. Miquel. The implicit calculus of constructions. extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the fifth International Conference on Typed Lambda Calculi and Applications (TLCA'01)*, Krakow (Poland), 2001.

[10] A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. PhD thesis, Université Paris 7, Dec. 2001.

[11] A. Miquel and B. Werner. The not so simple proof-irrelevent model of CC. In *TYPES*, 2002.

[12] S. Owre and N. Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Menlo Park, CA, 1997.

[13] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[14] M. Sozeau. Subset coercions in Coq. In *post-workshop proceedings of TYPES'2006*, 2007.

[15] 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`.

[16] B. Werner. On the strength of proof-irrelevant type theories. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 604–618. Springer, 2006.

[17] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.