CoqInE: Translating the Calculus of Inductive Constructions into the λΠ-calculus Modulo

Mathieu Boespflug[1] and Guillaume Burel[2]

[1] McGill University
Montréal, Québec, Canada
mboes@cs.mcgill.ca
[2] ÉNSIIE/Cédric/Inria AE Deducteam
1 square de la résistance, 91025 Évry cedex, France
guillaume.burel@ensiie.fr

Abstract

We show how to translate the Calculus of Inductive Constructions (CIC) as implemented by Coq into the λΠ-calculus modulo, a proposed common backend proof format for heterogeneous proof assistants.

1 Introduction

Verification of large developments will rely increasingly on a combination of heterogeneous provers, each being adapted to a particular aspect of the specification to realize. The question of the validity of the whole proof quickly arises. A way to recover confidence over all parts of the proofs is to translate them into a unique, simple but strong enough, formalism, in which they are all checked. A good candidate for such a formalism is the λΠ-calculus modulo, a simple yet extensible logical framework that conserves good properties of proofs, as presented in a companion paper [3]. A proof checker based on it, called Dedukti, has been developed by the first author. It was shown that many proof systems, if not all, can be encoded into the λΠ-calculus modulo, including HOL [7], pure type systems [4], resolution proofs for first-order logic, etc. In this paper, we present how to translate proofs of the Coq proof assistant into this formalism. This translation has been implemented in a tool called CoqInE[1], making it possible to use Dedukti to check Coq's developments.

The core formalism implemented by Coq, the Calculus of inductive Constructions (CIC), is an immensely expressive type theory. As implemented by Coq, it embeds all other systems of the λ-cube [1], includes local and global definitions, full dependent inductive datatype definitions, guarded fixpoint definitions and pattern matching, large eliminations, as well as a cumulative hierarchy of universes, to name but a few features. None of these features are native to the λΠ-calculus modulo in its bare form, which we call the λΠ-calculus. Encoding specifications and proofs from such a powerful formalism is therefore a good benchmark as to the adaptability and versatility of the λΠ-calculus modulo, and its convenience as a common proof format for all manner of systems.

2 Encoding the Calculus of Inductive Constructions

2.1 The Calculus of Constructions

The calculus of inductive constructions (CIC) is an extension of the calculus of constructions (CC), which is a type system for the λ-calculus with dependent types. [4] provides a sound

[1] Available at https://github.com/gburel/coqine.

and conservative encoding of all pure type systems, including CC, into the λΠ-calculus modulo (see also in [3] the instance of this encoding for System F). The CIC as implemented in Coq is in fact a pure type system with an infinity of sorts, called universes. To keep things simple, we will as a first approximation keep to only three sorts, Prop, Set and Type. Section 2.5 will discuss how to deal with universes.

In this paper, we use Dedukti's syntax for the λΠ-calculus modulo, where `x : A -> B` denotes the dependent product, `x : A => B` denotes the abstraction (*la* Church) and `[ Γ ] l --> r.` is the rule rewriting `l` to `r`, Γ being the typing context of the free variables of `l`. Following [4], we have the following constants for each sort $\mathsf{s} \in \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}\}$:

```
Us : Type.
es : Us -> Type.
```

The first is called a *universe constant*, for which we also have an associated *decoding function* (ε_{s}). For each axiom $\mathsf{s}_1 : \mathsf{s}_2$ in $\{\mathsf{Prop} : \mathsf{Type}, \mathsf{Set} : \mathsf{Type}\}$, [4] introduce a constant ($\dot{\mathsf{s}}$) and a rewrite rule:

```
dots1 : Us2.
[] es2 dots1 --> Us1.
```

For each rule $\langle \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_2 \rangle$ in $\{\langle \mathsf{Prop}, \mathsf{Prop}, \mathsf{Prop} \rangle, \langle \mathsf{Prop}, \mathsf{Set}, \mathsf{Set} \rangle, \langle \mathsf{Prop}, \mathsf{Type}, \mathsf{Type} \rangle, \langle \mathsf{Set}, \mathsf{Prop}, \mathsf{Prop} \rangle, \langle \mathsf{Set}, \mathsf{Set}, \mathsf{Set} \rangle, \langle \mathsf{Set}, \mathsf{Type}, \mathsf{Type} \rangle, \langle \mathsf{Type}, \mathsf{Prop}, \mathsf{Prop} \rangle, \langle \mathsf{Type}, \mathsf{Set}, \mathsf{Set} \rangle, \langle \mathsf{Type}, \mathsf{Type}, \mathsf{Type} \rangle\}$ we have a constant encoding the dependent product ($\dot{\Pi}_{\langle \mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_2 \rangle}$) together with a rewrite rule:

```
dotpis1s2 : x : Us1 -> y : (es1 x -> Us2) -> Us2).
[x : Us1, y : es1 x -> Us2]  es2 (dotpis1s2 x y) --> w : es1 x -> es2 (y w).
```

[4] also defines two translations: $|t|$ translates t viewed as a term, and $||t||$ translates t viewed as a type, with $||A|| = \mathsf{es}\ |A|$ for terms A typed by a sort s.

2.2 Inductive Types

CIC extends CC such that it becomes possible to define inductive types. In our translation, each time an inductive type is defined, we add new declarations and rewrite rules corresponding to the type. To give an intuition, let us first present how a concrete example, namely the vector indexed type, is translated.

Indexed vectors are defined in Coq by:

```
Inductive vector (A : Type) : nat -> Type :=
  Vnil : vector A O
| Vcons : forall (a : A) (n : nat), vector A n -> vector A (S n).
```

where `nat` is itself an inductive type with constructors `O` and `S`. The translation consists first in declaring two corresponding types in λΠ-calculus modulo:

```
vector : A : Utype -> (eset nat) -> Utype.
pre__vector : A : Utype -> (eset nat) -> Utype.
```

The `pre__vector` will be used to tag constructors. This will be useful for fixpoints (see next section). First, we give constants for each constructor, building a `pre__vector`:

```
Vnil : A : Utype -> etype (pre__vector A (nat__constr O)).
Vcons : A : Utype -> a : etype A -> n : eset nat ->
  etype (vector A n) -> etype (pre__vector A (nat__constr (S n))).
```

As we can see above in the case of the constructors of `nat`, constructors are guarded by a constant `vector__constr` allowing to pass from `pre__vector`s to `vector`s:

```
vector__constr : A : Utype -> n : eset nat ->
   etype (pre__vector A n) -> etype (vector A n).
```

Thus, the translation as a term of a constructor applied to its arguments will use this guard:

$$|\texttt{Vnil}\ A| = \texttt{vector\_\_constr}\ |A|\ (\texttt{nat\_\_constr O})\ (\texttt{Vnil}\ |A|)$$
$$|\texttt{Vcons}\ A\ a\ n\ v| = \texttt{vector\_\_constr}\ |A|\ (\texttt{nat\_\_constr S}\ (|n|))\ (\texttt{Vcons}\ |A|\ |a|\ |n|\ |v|).$$

If a constructor is only partially applied, we use η-expand it during translation. For the sake of readability, we now hide `__constr` guards using italics : *O* represents `nat__constr O`, *Vnil* A denotes `vector__constr` A *O* `(Vnil` A`)`, etc.

To translate pattern matching, we also add a case constant:

```
vector__case : A : Utype ->
  P : (n : eset nat -> vector A n -> Utype) ->
  fVnil : etype (P O (Vnil A)) ->
  fVcons : ( a : A -> n : eset nat -> v : etype (vector A n) ->
              P (S n) (Vcons A a n v) ) ->
  n : eset nat ->
  m : etype (vector A n) ->
    etype (P n m).
```

together with two rewrite rules, one for each constructor:

```
[A : Utype, P : ...] vector__case A P fVnil fVcons O (Vnil A) --> fVnil.
[...] vector__case A P fVnil fVcons (S n) (Vcons A a n v) --> fVcons a n v.
```

We now turn to the general case. Following the notations of Coq's manual [8], an inductive definition is given by $\mathsf{Ind}()[p](\Gamma_I := \Gamma_c)$ where p is the number of inductive parameters (as opposed to real arguments), Γ_I is the context of the definitions of (mutual) inductive types and Γ_C the context of constructors. For each inductive type definition $I : t$ we first add two declarations

```
I : ||t||.
pre__I : ||t||.
```

When the type t of the inductive type is of the form $\Pi x_1 : t_1.\ \ldots \Pi x_n : t_n.\ \mathtt{s}$, we also define a guarding constant:

```
I__constr : x1:||t1|| -> ··· -> xn:||tn|| -> es (pre__I x1 ··· xn) -> es (I x1 ··· xn).
```

For each constructor $c : \Pi x_1 : t_1.\ \ldots \Pi x_p : t_p.\ \Pi y_1 : s_1.\ \ldots \Pi y_l : s_l.\ I x_1 \ldots x_p\ u_1 \ldots u_{n-p}$ we declare a constant:

```
c : x1 : ||t1|| -> ··· -> xp : ||tp|| -> y1 : ||s1|| -> ··· -> yl : ||sl|| ->
      es (pre__I x1 ··· xp |u1| ··· |un−p|).
```

The translation as a term of a constructor applied to its arguments will therefore be:

$$|c\ p_1 \ldots p_p\ r_1 \ldots r_l| = \texttt{I\_\_constr}\ |p_1| \ldots |p_p|\ |v_1| \ldots |v_{n-p}|\ (\texttt{c}\ |p_1| \ldots |p_p|\ |r_1| \ldots |r_l|)$$

where $v_i = u_i\{p_1/x_1, \ldots, p_p/x_p, r_1/y_1, \ldots, r_l/y_l\}$.

To translate pattern matching, we also add a case constant:

```
I__case : x1 : ||t_1|| -> ··· -> xp : ||t_1|| ->
  P : (xp+1 : ||t_{p+1}|| -> ··· -> xn : ||t_n|| -> I x1 ··· xn -> UType) ->
    ... ->
  fc : (y1:||s_1|| -> ··· -> yl:||s_l|| -> P |u_1| ··· |u_{n-p}| |c x1 ··· xp y1 ··· yl|) ->
    ... ->
  xp+1 : ||t_{p+1}|| ->  ··· -> xn : ||t_n|| ->
  m : I x1 ··· xn ->
  etype (P xp+1 ··· xn m).
```

where `P` is the return type, and we have `fc` for each constructor, representing the corresponding branch of the matching. We then add a rewrite rule for each constructor:

```
[...] I__case x1 ... xp P ... fc ... |u_1| ... |u_{n-p}|
        (I__constr x1 ... xp |u_1| ... |u_{n-p}| (c x1 ... xp y1 ... yl)
  --> fc y1 ... yl.
```

2.3 Fixed points

For each fixed point in a term, we add a new constant with an associated rewrite rule. Since rewrite rules are defined globally, this constant must take as parameters the context of the fixed point, that is, all bounded varables above the position of the fixed point.

Given a fixed point $\mathsf{Fix} f_i\{f_1/k_1 : A_1 := t_1 \quad \ldots \quad f_n/k_n : A_n := t_n\}$ in a context $x_1 : B_1$, $\ldots$, $x_m : B_m$, assuming A_i is $\Pi y_i^1 : A_i^1. \ldots \Pi y_i^{k_i} : A_i^{k_i}. A_i'$ where $A_i^{k_i}$ is an inductive type $I_i w_i^1 \ldots w_i^{l_i}$ of sort `s`, we obtain the following declarations:

```
f1 : x1 : ||B_1|| -> ... -> xm : ||B_m|| -> ||A_1||.
[x1 : ||B_1||, ..., xm : ||B_m||, y11 : ||A_1^1||, ..., yk1-1 : ||A_1^{k_1-1}||,
 z1 : es (pre__I1 |w_1^1| ... |w_1^{l_1}|)]
   f1 x1 ... xm y11 ... y1k1-1 (I1__constr |w_1^1| ... |w_1^{l_1}| z1)
     --> |t_1| y11 ... y1k1-1 (I1__constr |w_1^1| ... |w_1^{l_1}| z1).
  ⋮
fn : x1 : ||B_1|| -> ... -> xm : ||B_m|| -> ||A_n||.
[x1 : ||B_1||, ..., xm : ||B_m||, yn1 : ||A_n^1||, ..., ynkn-1 : ||A_n^{k_n-1}||,
 zn : es (pre__In |w_n^1| ... |w_n^{l_n}|)]
   fn x1 ... xm yn1 ... ynkn-1 (In__constr |w_n^1| ... |w_n^{l_n}| zn)
     --> |t_n| yn1 ... ynkn-1 (In__constr |w_n^1| ... |w_n^{l_n}| zn).
```

The `Ii__constr` prevents the reduction of the fixed point unless a constructor is present as its k_ith parameter, which is exactly its semantics. Then, $\mathsf{Fix} f_i\{f_1/k_1 : A_1 := t_1 \ldots f_n/k_n : A_n := t_n\}$ is translated as `fi x1` $\ldots$ `xm` where the `xi` are the actual variables of the context of the fixed point. Because of dependent types, it is not possible to have a rewrite rule for the unfolding of the fixed point for each constructors of the inductive type. Indeed, the arguments w_i^j of the inductive type I_i can depend on y_i^k, and this can be incompatible with the type of the constructors. For instance, if we had a fixed point $\mathsf{Fix} f\{f : \Pi A : \mathsf{Set}.\ \Pi x\ y : A.\ \Pi p : x =_A y.\ P := t\}$, the following rewrite rule would be ill-typed:

```
f : A : Uset -> x : eset A -> y : eset A -> p : eprop (eq A x y) -> ||P||.
[A:Uset, x:eset A, y:eset A] f A x y |eq_refl A x|
--> |t| A x y |eq_refl A x|.
```

because $|eq\_refl\ A\ x|$ has type `eprop (eq A x x)` and not `eprop (eq A x y)`. This is the reason why we choose to translate constructors with the guarding constant `I__constr`.

Another solution would have been to duplicate the argument on which structural induction is based, one copy serving to match a constructor, the other being used to typecheck the body of the fixed point. Concretely, on the example above, this would give:

```
f : A : Uset -> x : eset A -> y : eset A -> p : eprop (eq A x y) -> ||P||.
[...] f A x y p -> f' A x y p A x y p.
f' : A : Uset -> x : eset A -> y : eset A -> p : eprop (eq A x y) ->
 A' : Uset -> x' : eset A' -> y' : eset A' -> p' : eprop (eq A' x' y') -> ||P||.
[...] f' A x y p A' x' x' (eq_refl A' x') --> |t|.
```

We plan to implement this second solution in a future version of CoqInE, since it avoids doubling the size of ground terms (*i.e.* terms build using only constructors).

Note that Dedukti does not check that the rewrite system terminates, in particular the rules corresponding to the fixed points. Indeed, the philosophy behind Dedukti is to have different tools for each purpose. The termination of the system corresponding to a fixed point should, however, be checkable using a tool based on size-based termination [2].

2.4 Modules

Coq features a powerful system of modules, with functors and refined structures. However, Dedukti understands only a notion of namespaces, lacking anything close to a full fledged module system.

To be able to translate functors, our choice is to parameterize each definition of a structure by all the parameters of the argument module. Then, for functor application, we need to apply all arguments of the applied module in each definitions. Concretely, if we have the following definitions in Coq:

```
Module Type T.                  Module M : T.
Parameter a : Type.             Definition a := Prop.
Parameter b : a.                Definition b : Prop := forall p : Prop, p.
End T.                          End M.

Module N (O : T).               Module P := N(M).
Definition x := O.b.
End N.
```

we obtain the following modules in Dedukti:

In module `M.dk`*:*

```
a : Utype.
[] a -->  dotprop.
b : Uprop.
[] b -->  dotpitp dotprop (p : etype dotprop => p).
```

In module `N.dk`*:*

```
x : fp_a : Utype -> fp_b : etype fp_a -> etype fp_a.
[fp_a : Utype, fp_b : etype fp_a] x fp_a fp_b --> fp_b.
```

In module `P.dk`*:*

```
x : etype M.a.
[] x -->  N.x M.a M.b.
```

2.5 Universes

Since in CIC there are in fact an infinity of universes, we cannot declare constants `Us`, `es`, etc. for each of them. A first solution would be to restrict oneself to the universes actually used in the module we are translating. This number is finite and typically very small [2]. However, this solution is not modular, since there are definitions that are polymorphic with respect to the universes. Typically, the `list` type takes as first parameter an argument of type `Type` for all universes, so that it can be seen as a family of inductive types. If a module uses `list` on a higher universe than those that were used when `list` was defined, we would have to duplicate its definition.

A better solution would be to have sorts parameterized by universe levels. Although technically possible, it should be proved that this provides a sound and conservative encoding of CIC. Currently, we bypass this issue by translating all universes by `Type`, and having `dottype : Utype`. Of course, this is not sound, since Girard's paradox can be reproduced.

Another issue concerns subtyping of sorts. Indeed, in CIC, if $t : Type(i)$ then $t : Type(j)$ for all $j \geq i$. Rewrite rules could be used to encode subtyping, for instance `Utype i --> Utype j`, but only if they are applied at positive positions. Since subtyping is contravariant, at negative positions, the dual rule `Utype j --> Utype i` should be used. Nevertheless, such a mechanism of *polarized* rewrite rules [5, 6] is currently not present in Dedukti.

3 Implementation details

CoqInE takes a Coq compiled file (`.vo`) and translates it into a Dedukti input file (`.dk`). Concretely speaking, a Coq compiled file contains a copy of the Coq kernel's in-memory representation of a module marshalled to disk. Since the marshalling facility is provided by the OCaml compiler, the format of the content in a `.vo` file is dependent both on the version of the compiler that compiled Coq, and also the version of Coq itself. Reading a Coq compiled file is a simple matter of unmarshalling the data, as implemented in the OCaml standard library.

The advantage of this scheme is that it avoids having to write a parser to read Coq proofs. Moreover, this ensures that the translation is performed on exactly the same terms that are used in Coq: on the contrary, an export of Coq's terms into another format could be unsound. And last but not least, it makes it easy to reuse part of Coq's code, because the data structures used to represent a Coq module are identical. In fact, CoqInE started as a fork of Chicken, a stripped down version of the Coq kernel. However, this has to following drawbacks:

- the same version of OCaml as the one used to compile Coq has to be used to compile CoqInE;
- The close coupling between the data structures of CoqInE and those of the Coq implementation means that CoqInE is tributary to Coq's implementation: core data structures change frequently between Coq releases (as happened between versions 8.2 and 8.3 for instance) and these changes need to be ported to CoqInE's codebase every time.

A better scheme would be to harness an export facility to a generic and stable format. Coq can already export proof snippets in XML. Unfortunately, however, this facility does not currently scale up to all the content of a full module.

[2] For instance, it is possible to type all of the Coq 8.2 standard library using only 4 levels of universes.

4 Results and Future Work

CoqInE is able to translate 256 modules of the standard library of Coq out of 304 (84%). The main features that are missing to translate the whole library are the following: coinductive types are not supported for the moment. A mechanism for handling lazy terms would be needed to get them. We cannot rely on abstracting terms to avoid their reduction, because Dedukti reduces terms under abstractions when it checks their convertibility. Also, translation of modules is not complete, in particular we do not translate refined structures (`S with p := p'`).

We could not check all of these translations using Dedukti. Indeed, although the translation is linear, some of them are too big. For instance, the `Coq.Lists.List` module, whose `.vo` file has length 260KB, is translated into a Dedukti file of length 1.2MB, which is in turn translated into a Haskell file of size 12MB. Of course, GHC is not able to run or compile this file. A simple solution to overcome this issue would be to cut big modules into smaller files. A more stable solution consists in making Dedukti no longer produce Haskell code, but code that can be compiled just-in-time. Dedukti is currently rewritten to allow this.

Another limitation of Dedukti is the absence of higher-order matching. Higher-order patterns can appear for instance in the rules for `I__case` when the real arguments of the inductive type I are higher order. For instance, if we define the inductive type

```
Inductive I : (Prop -> Prop) -> Prop := c : I (fun t => t).
```

we obtain the following rewrite rule:

```
[...] I__case P fc (t : Uprop => t) (I__constr (t : Uprop => t) c) --> fc.
```

which contains higher-order patterns. However, these higher-order patterns are not useful for the matching, since the important pattern is the constructor `c`. Therefore, the possibility to declare that a pattern should be used for the typing of the rule but not for the matching has been added recently to Dedukti. Nevertheless, these so-called dot patterns (see [3]) are currently not used in CoqInE yet.

As noted in Section 2.5, one last remaining point is the encoding of universes.

References

[1] Henk Barendregt. *Handbook of logic in computer science*, volume 2, chapter Lambda calculi with types, pages 117–309. Oxford University Press, 1992.

[2] Gilles Barthe, Maria João Frade, E. Giménez, Luis Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Math. Struct. Comp. Sci.*, 14(1):97–141, 2004.

[3] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *PxTP*, 2012.

[4] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

[5] Gilles Dowek. What is a theory? In Helmut Alt and Afonso Ferreira, editors, *STACS*, volume 2285 of *LNCS*, pages 50–64. Springer, 2002.

[6] Gilles Dowek. Polarized resolution modulo. In Cristian S. Calude and Vladimiro Sassone, editors, *IFIP TCS*, volume 323 of *IFIP AICT*, pages 182–196. Springer, 2010.

[7] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. HOL-λσ an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):1–25, 2001.

[8] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 2010. Version 8.3, available at `http://coq.inria.fr/refman/index.html`.