Towards Certified Meta-Programming with Typed Template-Coq

Abhishek Anand[1], Simon Boulier[2], Cyril Cohen[3], Matthieu Sozeau[4] and Nicolas Tabareau[2]

[1] Cornell University, Ithaca, NY U.S.A.
[2] Gallinette Project-Team, Inria Nantes, France
[3] Université Côte d'Azur, Inria, France
[4] Pi.R2 Project-Team, Inria Paris and IRIF, France

Abstract. Template-Coq[5] is a plugin for Coq, originally implemented by Malecha [17], which provides a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Initially, it was developed for the purpose of writing functions on Coq's AST in Gallina. Recently, it was used in the CertiCoq certified compiler project [4], as its front-end language, to derive parametricity properties [3], and to extract Coq terms to a CBV λ-calculus [13]. However, the syntax lacked semantics, be it typing semantics or operational semantics, which should reflect, as formal specifications in Coq, the semantics of Coq's type theory itself. The tool was also rather bare bones, providing only rudimentary quoting and unquoting commands. We generalize it to handle the entire Calculus of Inductive Constructions (CIC), as implemented by Coq, including the kernel's declaration structures for definitions and inductives, and implement a monad for general manipulation of Coq's logical environment. We demonstrate how this setup allows Coq users to define many kinds of general purpose plugins, whose correctness can be readily proved in the system itself, and that can be run efficiently after extraction. We give a few examples of implemented plugins, including a parametricity translation. We also advocate the use of Template-Coq as a foundation for higher-level tools.

1 Introduction

Meta-programming is the art of writing programs (in a *meta-language*) that produce or manipulate programs (written in an *object language*). In the setting of dependent type theory, the expressivity of the language permits to consider the case were the meta and object languages are actually the same, *accounting for well-typedness*. This idea has been pursued in the work on inductive-recursive (IR) and quotient inductive-inductive types (QIIT) in Agda to reflect a syntactic model of a dependently-typed language within another one [9,2]. These term encodings include type-correcteness internally by considering only well-typed terms of the syntax, i.e. derivations. However, the use of IR or QIITs complicates considerably

[5] `https://template-coq.github.io/template-coq`

the meta-theory of the meta-language which makes it difficult to coincide with the object language represented by an inductive type. More problematically in practice, the concision and encapsulation of the syntactic encoding has the drawback that it is very difficult to use because any function from the syntax can be built only at the price of a proof that it respects typing, conversion or any other features described by the intrinsically typed syntax right away.

Other works have taken advantage of the power of dependent types to do meta-programming in a more progressive manner, by first defining the syntax of terms and types; and then defining out of it the notions of reduction, conversion and typing derivation [11,23] (the introduction of [11] provides a comprehensive review of related work in this area). This can be seen has a type-theoretic version of the functional programming language designs such as Template Haskell [20] or MetaML [22]. This is also the approach taken by Malecha in his thesis [17], where he defined Template-Coq, a plugin which defines a correspondence—using quoting and unquoting functions—between Coq kernel terms and inhabitants of an inductive type representing internally the syntax of the calculus of inductive constructions (CIC), as implemented in Coq. It becomes thus possible to define programs in Coq that manipulate the representation of Coq terms and reify them as functions on Coq terms. Recently, its use was extended for the needs of the CertiCoq certified compiler project [4], which uses it as its front-end language. It was also used by Anand and Morissett [3] to formalize a modified parametricity translation, and to extract Coq terms to a CBV λ-calculus [13]. All of these translations however lacked any means to talk about the semantics of the reified programs, only syntax was provided by Template-Coq. This is an issue for CertiCoq for example where both a non-deterministic small step semantics and a deterministic call-by-value big step semantics for CIC terms had to be defined and preserved by the compiler, without an "official" specification to refer to.

This paper proposes to remedy this situation and provides a formal semantics of Coq's implemented type theory, that can independently be refined and studied. The advantage of having a very concrete untyped description of Coq terms (as opposed to IR or QIITs definitions) together with an explicit type checker is that the extracted type-checking algorithm gives rise to an OCaml program that can directly be used to type-check Coq kernel terms. This opens a way to a concrete solution to bootstrap Coq by implementing the Coq kernel in Coq. However, a complete reification of CIC terms and a definition of the checker are not enough to provide a meta-programming framework in which Coq plugins could be implemented. One needs to get access to Coq logical environments. This is achieved using a monad that reifies Coq general commands, such as lookups and declarations of constants and inductive types.

As far as we know this is the only reflection framework in a dependently-typed language allowing such manipulations of terms and datatypes, thanks to the relatively concise representation of terms and inductive families in CIC. Compared to the MetaCoq project [24], Lean's tactic monad [12], or Agda's reflection framework [23], our ultimate goal is not to interface with Coq's unification

```
Inductive term : Set :=
| tRel       : ℕ → term
| tVar       : ident → term
| tEvar      : ℕ → list term → term
| tSort      : universe → term
| tCast      : term → cast_kind → term → term
| tProd      : name → term → term → term
| tLambda    : name → term → term → term
| tLetIn     : name → term → term → term → term
| tApp       : term → list term → term
| tConst     : kername → universe_instance → term
| tInd       : inductive → universe_instance → term
| tConstruct : inductive → ℕ → universe_instance → term
| tCase      : (inductive ∗ ℕ) → term → term → list (ℕ ∗ term) → term
| tProj      : projection → term → term
| tFix       : mfixpoint term → ℕ → term
| tCoFix     : mfixpoint term → ℕ → term.
```

Fig. 1. Representation of the syntax in Template-Coq

and type-checking algorithms, but to provide a self-hosted, bootstrappable and verifiable implementation of these algorithms. On one hand, this opens the possibility to verify the kernel's implementation, a problem tackled by Barras [6] using set-theoretic models. On the other hand we also advocate for the use of Template-Coq as a foundation on which higher-level tools can be built: meta-programs implementing translations, boilerplate-generating tools, domain-specific proof languages, or even general purpose tactic languages.

Plan of the paper. In Section 2, we present the complete reification of Coq terms, covering the entire CIC and define in Section 3 the type-checking algorithm of Coq reified terms in Coq. In Section 4, we show the definition of a monad for general manipulation of Coq's logical environment and use it to define plugins for various translations from Coq to Coq (Section 5). Finally, we discuss related and future work in Section 6.

2 Reification of Coq Terms

Reification of syntax. The central piece of Template-Coq is the inductive type `term` which represents the syntax of Coq terms, as defined in Fig. 1. This inductive follows directly the `constr` datatype of Coq terms in the OCaml code of Coq, except for the use of OCaml's native arrays and strings; an upcoming extension of Coq [5] with such features should solve this mismatch.

Constructor `tRel` represents variables bound by abstractions (introduced by `tLambda`), dependent products (introduced by `tProd`) and local definitions (introduced by `tLetIn`), the natural number is a De Bruijn index. The `name` is a printing annotation.

Sorts are represented with `tSort`, which takes a `universe` as argument. A universe is the supremum of a (non-empty) list of level expressions, and a level is either `Prop`, `Set`, a global level or a De Bruijn polymorphic level variable.

```
Inductive level ≔ lProp | lSet | Level (_ : string) | Var (_ : ℕ).
Definition universe ≔ list (level ∗ bool). (* level+1 if true *)
```

The application (introduced by `tApp`) is n-ary. The `tConst`, `tInd` and `tConstruct` constructors represent references to constants (definitions or axioms), inductives, or constructors of an inductive type. The `universe_instances` are non-empty only for polymorphic constants. Finally, `tCase` represents pattern-matchings, `tProj` primitive projections `tFix` fixpoints and `tCoFix` cofixpoints.

Quoting and unquoting of terms. Template-Coq provides a lifting from concrete syntax to reified syntax (quoting) and the converse (unquoting). It can reify and reflect all kernel Coq terms.

The command `Quote Definition` reifies the syntax of a term. For instance,

```
Quote Definition f ≔ (fun x ⇒ x + 0).
```

generates the term `f` defined as

```
f = tApp (tLambda (nNamed "x") (tInd (mkInd "Coq.Init.Datatypes.nat" 0) [ ]) (
    tApp (tConst "Coq.Init.Nat.add" [ ]) [tRel 0; tRel 0])) [tConstruct (mkInd "
    Coq.Init.Datatypes.nat" 0) 0 [ ]] : term
```

On the converse, the command `Make Definition` constructs a term from its syntax. This example below defines `zero` to be 0 of type ℕ.

```
Make Definition zero ≔ (tConstruct (mkInd "Coq.Init.Datatypes.nat" 0) 0 [ ]).
```

where `mkInd n k` is the $\mathtt{k}^{\mathrm{th}}$ inductive of the mutual block of the name `n`.

Reification of environment. In Coq, the meaning of a term is relative to an environment, which must be reified as well. Environments consist in three part: (i) a graph of universes (ii) declarations of definitions, axioms and inductives (iii) a local context registering types of De Bruijn indexes.

As we have seen in the syntax of terms, universe levels are not explicits in Coq. Instead, level variables are introduced and constraints between them are registered in a *graph of universes'*. This is the way typical ambiguity is implemented in Coq. A constraint is given by two levels and a `constraint_type` (`Lt`, `Le` or `Eq`):

```
Definition univ_constraint ≔ Level ∗ constraint_type ∗ Level.
```

Then the graph is given by a set of level variables and of constraints.

```
Definition uGraph.t ≔ LevelSet.t ∗ constraints.
```

Functions to query the graph are provided, for the moment they rely on a naive implementation of Bellman-Ford algorithm. `check_leq u1 u2` checks if the graph enforce `u1` ≤ `u2` and `no_universe_inconsistency` checks that the graph has no negative cycle.

Constant and inductive declarations are grouped together, properly ordered according to dependencies, in a global context (`global_ctx`), which is a list of global declarations (`global_decl`).

```
Inductive global_decl :=
| ConstantDecl : ident → constant_decl → global_decl
| InductiveDecl : ident → minductive_decl → global_decl.
```

Definitions and axioms just associate a name to a universe context, and two terms for the optional body and type. Inductives are more involved:

```
 (* Declaration of one inductive type *)
Record inductive_body := { ind_name : ident;
    ind_type : term; (* closed arity *)
    ind_kelim : list sort_family; (* allowed elimination sorts *)
    (* names, types, number of arguments of constructors *)
    ind_ctors : list (ident * term * nat);
    ind_projs : list (ident * term) (* names and types of projections *) }.

(* Declaration of a block of mutual inductive types *)
Record minductive_decl :=
 { ind_npars : nat; (* number of parameters *)
   ind_bodies : list inductive_body; (* inductives of the mutual block *)
   ind_universes : universe_context (* universe constraints *) }.
```

In Coq internals, there are in fact two ways of representing a declaration: either as a "declaration" or as an "entry". The kernel takes entries as input, type-check them and elaborate them to declarations. In Template-Coq, we provide both, and provide an erasing function `mind_decl_to_entry` from declarations to entries for inductive types.

Finally, local contexts are just list of local declarations: a type for lambda bindings and a type and a body for let bindings.

Quoting and unquoting the environment Template-Coq provides the command `Quote Recursively Definition` to quote an environment. This command crawls the environment and quote all declarations needed to typecheck a given term.

The other way, the commands `Make Inductive` allows declaring an inductive type from its entry. For instance the following redefines a copy of $\mathbb{N}$:

```
Make Inductive (mind_decl_to_entry {|
   ind_npars := 0;  ind_universes := [ ];
   ind_bodies := [{|
    ind_name := "nat";
    ind_type := tSort [(lSet, false)];
    ind_kelim := [InProp; InSet; InType];
    ind_ctors := [("O", tRel 0, 0); ("S", tProd nAnon (tRel 0) (tRel 1), 1)];
    ind_projs := [ ] |}] |}).
```

More examples of use of quoting/unquoting commands can be found in the file `test-suite/demo.v`.

```
Inductive cumul (Σ : global_context) (Γ : context) : term → term → Prop :=
| cumul_refl t u : leq_term (snd Σ) t u = true → Σ ; Γ ⊢ t ≤ u
| cumul_red_l t u v : red1 Σ Γ t v → Σ ; Γ ⊢ v ≤ u → Σ ; Γ ⊢ t ≤ u
| cumul_red_r t u v : Σ ; Γ ⊢ t ≤ v → red1 Σ Γ u v → Σ ; Γ ⊢ t ≤ u

where " Σ ; Γ ⊢ t ≤ u " := (cumul Σ Γ t u).

Inductive typing (Σ : global_context) (Γ : context) : term → term → Set :=
| type_Rel n : ∀ (isdecl : n < List.length Γ),
    Σ ; Γ ⊢ tRel n : lift0 (S n) (safe_nth Γ (exist _ n isdecl)).(decl_type)
| type_Sort (l : level) :
    Σ ; Γ ⊢ tSort (Universe.make l) : tSort (Universe.super l)
| type_Prod n t b s1 s2 :
    Σ ; Γ ⊢ t : tSort s1 → Σ ; Γ , vass n t ⊢ b : tSort s2 →
    Σ ; Γ ⊢ tProd n t b : tSort (max_universe s1 s2)
| type_App t l t_ty t' :
    Σ ; Γ ⊢ t : t_ty → typing_spine Σ Γ t_ty l t' →
    Σ ; Γ ⊢ tApp t l : t'
| ...
where " Σ ; Γ ⊢ t : T " := (typing Σ Γ t T)

with typing_spine Σ Γ : term → list term → term → Prop :=
| type_spine_nil ty : typing_spine Σ Γ ty [ ] ty
| type_spine_const hd tl na A B B' T :
    Σ ; Γ ⊢ T ≤ tProd na A B → Σ ; Γ ⊢ hd : A →
    typing_spine Σ Γ (subst0 hd B) tl B' →
    typing_spine Σ Γ T (cons hd tl) B'
```

Fig. 2. Typing judment for `terms`, excerpt

3 Type Checking Coq in Coq

In Fig. 2, we present (an excerpt of) the specification of the typing judgment of the kernel of Coq using the inductive type `typing`. It represents all the typing rules of Coq[6]. This includes the basic dependent lambda calculus with lets, global references to inductives and constants, the `match` construct and primitive projections. Universe polymorphic definitions and the well-formedness judgment for global declarations are dealt with as well.

The only ingredients missing are the guard check for fixpoint and productivity of cofixpoints and the positivity condition of mutual (co-) inductive types. They are work-in-progress.

The typing judgment `typing` is mutually defined with `typing_spine` to account for n-ary applications. Untyped reduction `red1` and cumulativity `cumul` can be defined separately.

[6] We do not treat metavariables which are absent from kernel terms and require a separate environment for their declarations.

Implementation. To test this specification, we have implemented the basic algorithms for type-checking in Coq, that is, we implement type inference: given a context and a term, output its type or produce a type error. All the rules of type inference are straightforward except for cumulativity. The cumulativity test is implemented by comparing head normal forms for a fast-path failure and potentially calling itself recursively, unfolding definitions at the head in Coq's kernel in case the heads are equal. We implemented weak-head reduction by mimicking Coq's kernel implementation, which is based on an abstract machine inspired by the KAM. Coq's machine optionally implements a variant of lazy, memoizing evaluation (which can have mixed results, see Coq's PR #555 for example), that feature has not been implemented yet.

The main difference with the OCaml implementation is that all of the functions are required to be shown terminating in Coq. One possibility could be to prove the termination of type-checking separately but this amounts to prove in particular the normalization of CIC which is a complex task. Instead, we simply add a fuel parameter to make them syntactically recursive and make `OutOfFuel` a type error, *i.e.*, we are working in a variant of the option monad.

Bootstrapping it. We can extract this checker to OCaml and reuse the setup described in Section 2 to connect it with the reifier and easily derive a (partialy verified) alternative checker for Coq's `.vo` object files. Our plugin provides a new command `Template Check` for typechecking definitions using the alternative checker, that can be used as follows:

```
Require Import Template.TemplateCoqChecker List. Import ListNotations.
Definition foo := List.map (fun x ⇒ x + 3) [0; 1].
Template Check foo.
```

Our initial tests indicate that it runs in reasonable time on medium-sized proof terms.

4 Reification of Coq Commands

Coq plugins need to interact with the environment, for example by repeatedly looking up definitions by name, declaring new constants using fresh names, or performing computations. It is desirable to allow such programs to be written in Coq (Gallina) because of the two following advantages. Plugin-writers no longer need to understand the OCaml implementation of Coq and plugins are no longer sensitive to changes made in the OCaml implementation. Also, when plugins implementing syntactic models are proven correct in Coq, they provide a mechanism to add axioms to Coq without compromising consistency (§5.3).

In general, interactions with the environment have side effects, e.g. the declaration of new constants, which must be described in Coq's pure setting. To overcome this difficulty, we use the standard "free" monadic setting to represent the operations involved in interacting with the environment, as done for instance in Mtac [24].

`TemplateMonad` is an inductive family (Fig. 3) such that `TemplateMonad A` represents a program which will finally output a term of type `A`. There are special

```
Inductive TemplateMonad : Type → Prop :=
(* Monadic operations *)
| tmReturn : ∀ {A}, A → TemplateMonad A
| tmBind : ∀ {A B}, TemplateMonad A → (A → TemplateMonad B) → TemplateMonad B

(* General operations *)
| tmPrint : ∀ {A}, A → TemplateMonad unit
| tmFail : ∀ {A}, string → TemplateMonad A
| tmEval : reductionStrategy → ∀ {A}, A → TemplateMonad A
| tmDefinition : ident → ∀ {A}, A → TemplateMonad A
| tmAxiom : ident → ∀ A, TemplateMonad A
| tmLemma : ident → ∀ A, TemplateMonad A
| tmFreshName : ident → TemplateMonad ident
| tmAbout : ident → TemplateMonad (option global_reference)
| tmCurrentModPath : unit → TemplateMonad string

(* Quoting and unquoting operations *)
| tmQuote : ∀ {A}, A  → TemplateMonad term
| tmQuoteRec : ∀ {A}, A  → TemplateMonad program
| tmQuoteInductive : kername → TemplateMonad mutual_inductive_entry
| tmQuoteConstant : kername → bool → TemplateMonad constant_entry
| tmMkDefinition : ident → term → TemplateMonad unit
| tmMkInductive : mutual_inductive_entry → TemplateMonad unit
| tmUnquote : term  → TemplateMonad {A : Type & A}.
| tmUnquoteTyped : ∀ A, term → TemplateMonad A
```

Fig. 3. The monad of commands

constructor `tmReturn` and `tmBind` to provide (freely) the basic monadic operations. We use the standard monadic syntactic sugar `x ← t ; u` for `tmBind t (fun x ⇒ u)`.

The other operations of the monad can be classified in two categories: the traditional Coq operations (`tmDefinition` to declare a new definition, . . .) and the quoting and unquoting operations to move between Coq term and their syntax or to work directly on the syntax (`tmMkInductive` to declare a new inductive from its syntax for instance). A overview is given in Table 1.

A program `prog` of type `TemplateMonad A` can be executed with the command `Run TemplateProgram prog`. This command is thus an interpreter for `TemplateMonad` programs, implemented in OCaml as a traditional Coq plugin. The term produced by the program is discarded but, and it is the point, a program can have many side effects like declaring a new definition or a new inductive type, printing something, ...

Let us look at some examples. The following program adds the definitions `foo := 12` and `bar := foo + 1` to the current context.

```
Run TemplateProgram (foo ← tmDefinition "foo" 12 ;
                     tmDefinition "bar" (foo +1)).
```

Vernacular command	Reified command with arguments	Description
`Eval`	`tmEval red t`	Returns the evaluation of `t` following the evaluation strategy `red` (`cbv`, `cbn`, `hnf`, `all` or `lazy`)
`Definition`	`tmDefinition id t`	Makes the definition `id` ≔ `t` and returns the created constant `id`
`Axiom`	`tmAxiom id A`	Adds the axiom `id` of type `A` and returns the created constant `id`
`Lemma`	`tmLemma id A`	Generates an obligation of type `A`, returns the created constant `id` after all obligations close
`About` or `Locate`	`tmAbout id`	Returns `Some gr` if `id` is a constant in the current environment and `gr` is the corresponding global reference. Returns `None` otherwise.
	`tmQuote t`	Returns the syntax of `t` (of type `term`)
	`tmQuoteRec t`	Returns the syntax of `t` and all the declarations on which it depends
	`tmQuoteInductive kn`	Returns the declaration of the inductive `kn`
	`tmQuoteConstant kn b`	Returns the declaration of the constant `kn`, if `b` is `true` the implementation bypass opacity to get the body of the constant
`Make Definition`	`tmMkDefinition id tm`	Adds the definition `id` ≔ `t` where `t` is denoted by `tm`
`Make Inductive`	`tmMkInductive d`	Declares the inductive denoted by the declaration `d`
	`tmUnquote tm`	Returns the pair (`A`;`t`) where `t` is the term whose syntax is `tm` and `A` it's type
	`tmUnquoteTyped A tm`	Returns the term whose syntax is `tm` and checks that it is indeed of type `A`

Table 1. Main Template-Coq commands

The program below asks the user to provide an inhabitant of ℕ (here we provide $3 * 3$) and records it in the lemma `foo` ; prints its normal form ; and records the syntax of its normal form in `foo_nf_syntax` (hence of type `term`). We use Program's obligation mechanism to ask for missing proofs, running the rest of the program when the user finishes providing it. This enables the implementation of *interactive* plugins.

```
Run TemplateProgram (foo ← tmLemma "foo" ℕ ;
            nf  ← tmEval all foo ; tmPrint "normal form: " ; tmPrint nf ;
            nf_ ← tmQuote nf ;
            tmDefinition "foo_nf_syntax" nf_).
Next Obligation. exact (3 * 3). Defined.
```

5 Writing Coq plugins in Coq

The reification of syntax, typing and commands of Coq allow writing a Coq plugin directly inside Coq, without requiring another language like OCaml and an external compilation phase.

In this section, we describe three examples of such plugins: (i) a plugin that adds a constructor to an inductive type, (ii) a re-implementation of Lasson's parametricity plugin[7], and (iii) an implementation of a plugin that provides an extension of CIC—using a syntactic translation—in which it is possible to prove the negation of functional extensionality [8].

5.1 A Plugin to Add a Constructor

Our first example is a toy example to show the methodology of writing plugins in Template-Coq. Given an inductive type `I`, we want to declare a new inductive type `I'` which corresponds to `I` plus one more constructor.

For instance, let's say we have a syntax for lambda calculus:

```
Inductive tm : Set := var : nat → tm | lam : tm → tm | app : tm → tm → tm.
```

And that in some part of our development, we want to consider a variation of `tm` with a new constructor, *e.g.,* `let in`. Then we declare `tm'` with the plugin by:

```
Run TemplateProgram (add_constructor tm "letin" (fun tm' ⇒ tm' → tm' → tm')).
```

This command has the same effect as declaring the inductive `tm'` by hand:

```
Inductive tm' : Set := var' : nat → tm' | lam' : tm' → tm' | app' : tm' → tm' → tm'
| letin : tm' → tm' → tm'.
```

but with the benefit that if `tm` is changed, for instance by adding one new constructor, then `tm'` is automatically changed accordingly. We provide other examples in the file `test-suite/add_constructor.v`, e.g. with mutual inductives.

We will see that it is fairly easy to define this plugin using Template-Coq. The main function is `add_constructor` which take an inductive type `ind` (whose type is not necessarily `Type` if it is an inductive family), a name `idc` for the new constructor and the type `ctor` of the new constructor, abstracted with respect to the new inductive.

```
Definition add_constructor {A} (ind : A) (idc : ident) {B} (ctor : B)
  : TemplateMonad unit
  := tm ← tmQuote ind ;
    match tm with
    | tInd ind₀ _ ⇒
      decl ← tmQuoteInductive (inductive_mind ind₀) ;
      ctor ← tmQuote ctor ;
      d' ← tmEval lazy (add_ctor decl ind₀ idc ctor) ;
      tmMkInductive d'
    | _ ⇒ tmFail "The provided term is not an inductive"
    end.
```

[7] https://github.com/parametricity-coq/paramcoq

It works in the following way. First the inductive type `ind` is quoted, the obtained term `tm` is expected to be a `tInd` constructor otherwise the function fails. Then the declaration of this inductive is obtained by calling `tmQuoteInductive`, the constructor is reified too, and an auxiliary function is called to add the constructor to the declaration. After evaluation, the new inductive type is added to the current context with `tmMkInductive`.

It remains to define the `add_ctor` auxiliary function to complete the definition of the plugin. It takes a `minductive_decl` which is the declaration of a block of mutual inductive types and returns a `minductive_decl`.

```
Definition add_ctor (mind : minductive_decl) (ind₀ : inductive) (idc : ident)
           (ctor : term) : minductive_decl
 := let i₀ := inductive_ind ind₀ in
   {| ind_npars := mind.(ind_npars) ;
     ind_bodies := map_i (fun (i : nat) (ind : inductive_body) ⇒
                 {| ind_name := tsl_ident ind.(ind_name) ;
                   ind_type  := ind.(ind_type) ;
                   ind_kelim := ind.(ind_kelim) ;
                   ind_ctors :=
                    let ctors := map (fun '(id, t, k) ⇒ (tsl_ident id, t, k))
                                 ind.(ind_ctors) in
                    if Nat.eqb i i₀ then
                     let n := length mind.(ind_bodies) in
                     let typ := try_remove_n_lambdas n ctor in
                     ctors ++ [(idc, typ, 0)]
                    else ctors;
                   ind_projs := ind.(ind_projs) |})
                   mind.(ind_bodies) |}.
```

The declaration of the block of mutual inductive types is a record. The field `ind_bodies` contains the list of declarations of each inductive of the block. We see that most of the fields of the records are propagated, except for the names which are translated to add some primes and `ind_ctors`, the list of types of constructors, for which, in the case of the relevant inductive (`i₀` is its number), the new constructor is added.

5.2 Parametricity Plugin

We now show how Template-Coq permits to define a parametricity plugin that computes the translation of a term following Bernardy's parametricity [7], whose definition in the unary case is described in Fig. 4. The soundness theorem ensures that, for a term t of type A, $[t]_1$ computes a proof of parametricity of $[t]_0$ in the sense that it has type $[A]_1\ [t]_0$. The definition of the plugin goes in two steps: first the definition of the translation on the syntax of `term` in Template-Coq and then the instrumentation to connect it with terms of Coq using the `TemplateMonad`.

The presentation of the parametricity translation of Fig. 4 is total and syntax directed, the two components of the translation $[\ ]_0$ and $[\ ]_1$ are implemented by

$$
\begin{aligned}
[t]_0 &= t \\
[x]_1 &= x^t \\
[\forall (x : A).B]_1 &= \lambda f.\forall (x : [A]_0)(x^t : [A]_1 x).[B]_1 (f\ x) \\
[\lambda (x : A).t]_1 &= \lambda (x : [A]_0)(x^t : [A]_1 x).[t]_1 \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : [A]_0, x^t : [A]_1\, x
\end{aligned}
$$

$$
\frac{\Gamma \vdash t : A}{\begin{array}{c} [\![\Gamma]\!] \vdash [t]_0 : [A]_0 \\ [\![\Gamma]\!] \vdash [t]_1 : [A]_1\, [t]_0 \end{array}}
$$

Fig. 4. Unary parametricity translation and soundness theorem, excerpt (from [7])

two recursive functions `tsl_param`$_0$ and `tsl_param`$_1$. The translation is presented in a named setting, so the introduction of new variables does not change references to existing ones. That's why, $[\ ]_0$ is the identity.

In the De Bruijn setting of Template-Coq, the translation has to take into account the shift induced by the duplication of the context. Therefore, the implementation `tsl_param`$_0$ of $[\ ]_0$ is not the identity anymore. The argument `n` of `tsl_param`$_0$ represents the De Bruijn level from which the variables have been duplicated. There is no need for such an argument in `tsl_param`$_1$, the implementation of $[\ ]_1$, because in this function all variables are duplicated.

```
Fixpoint tsl_param0 (n : nat) (t : term) {struct t} : term :=
match t with
| tRel k ⇒ if k >= n then (* global variable *) tRel (2*k−n+1)
                    else (* local  variable *) tRel k
| tProd na A B ⇒ tProd na (tsl_param0 n A) (tsl_param0 (n+1) B)
| _ ⇒ ...
end.

Fixpoint tsl_param1 (E : tsl_table) (t : term) : term :=
match t with
| tRel k ⇒ tRel (2 ∗ k)
| tSort s ⇒ tLambda (nNamed "A") (tSort s) (tProd nAnon (tRel 0) (tSort s))
| tProd na A B ⇒ let A0 := tsl_param0 0 A in let A1 := tsl_param1 E A in
                 let B0 := tsl_param0 1 B in let B1 := tsl_param1 E B in
   tLambda (nNamed "f") (tProd na A0 B0)
   (tProd na (lift0 1 A0) (tProd (tsl_name na) (subst_app (lift0 2 A1) [tRel 0])
   (subst_app (lift 1 2 B1) [tApp (tRel 2) [tRel 1] ])))
| tConst s univs ⇒ match lookup_tsl_table E (ConstRef s) with
                     | Some t ⇒ t
                     | None ⇒ default_term
                   end
| _ ⇒ ...
end.
```

The parametricity plugin not only has to be defined on terms of CIC but also on additional terms dealing with the global context. In particular, constants are translated using a translation table which records the previous constant's translations.

```
Definition tsl_table := list (global_reference * term).
```

If a constant is not in the translation table we return a dummy `default_term`, considered as an error (this could also be handled by an option monad).

We have also implemented the translation of inductives and pattern matching. For instance the translation of the equality type `eq` produces the inductive type:

```
Inductive eqᵗ A (Aᵗ : A → Type) (x : A) (xᵗ : Aᵗ x) : ∀ H, Aᵗ H → x = H → Prop :=
  eq_reflᵗ : eqᵗ A Aᵗ x xᵗ x xᵗ eq_refl.
```

Then $[\texttt{eq}]_1$ is given by $\texttt{eq}^t$ and $[\texttt{eq\_refl}]_1$ by $\texttt{eq\_refl}^t$.

Given $\texttt{tsl\_param}_0$ and $\texttt{tsl\_param}_1$ the translation of the declaration of a block of mutual inductive types is not so hard to get. Indeed such a declaration mainly consists of the arities of the inductives and the types of constructors. For the translated inductive, they are roughly produced by translation of the original ones.

```
Definition tsl_mind_decl (E : tsl_table) (kn : kername)
           (mind : minductive_decl) : minductive_decl.
```

In a similar manner, we can translate pattern-matching. Note that the plugin does not support fixpoints and cofixpoints for the moment.

Now, it remains to connect this translation of terms in `term` to terms of Coq. For this, we define the new command `tTranslate` in the `TemplateMonad`.

```
Definition tTranslate (E : tsl_table) (id : ident) : TemplateMonad (tsl_table).
```

When `id` is a definition, the command recovers the body of `id` (as a `term`) using `tmQuoteConstant` and then translates it and records it in a new definition $\texttt{id}^t$. The command returns the translation table `E` extended by ($\texttt{id}$, $\texttt{id}^t$). In the case `id` is an inductive type or a constructor then the command does basically the same but extends the translation table with both the inductive and the constructors. If `id` is an axiom or not a constant the command fails.

Here is an illustration coming from the work of Lasson [15] on the automatic proofs of (ω-)groupoid laws using parametricity. We show that all function of type `ID := ∀ A x y, x = y → x = y` are identity functions. First we need to record the translations of `eq` and `ID`.

```
Run TemplateProgram (table ← tTranslate [ ] "eq" ;
                     table ← tTranslate table "ID" ;
                     tmDefinition "table" table).
```

Then we show that every parametric function on `ID` is pointwise equal to the identity using the predicate `fun y ⇒ x = y`.

```
Lemma param_ID (f : ID) : IDᵗ f → ∀ A x y p, f A x y p = p.
Proof.
  intros H A x y p. destruct p.
  destruct (H A (fun y ⇒ x = y) x eq_refl x eq_refl eq_refl (eq_reflᵗ _ _)).
  reflexivity.
Qed.
```

$$
\begin{aligned}
[x] &:= x & [\lambda(x:A).\,t] &:= (\lambda(x:[A]).\,[t],\, \text{true}) \\
[t\ u] &:= (\pi_1\ t)\ u & [\forall(x:A).\,B] &:= (\forall(x:[A]).\,[B]) \times \mathbb{B}
\end{aligned}
$$

Fig. 5. Intensional Function Translation, excerpt (from [8])

Then we define a function `myf` $:= p \mapsto p \centerdot p^{-1} \centerdot p$ and get its parametricity proof using the plugin.

```
Definition myf : ID := fun A x y p ⇒ eq_trans (eq_trans p (eq_sym p)) p.

Run TemplateProgram (table ← tTranslate table "eq_sym" ;
                     table ← tTranslate table "eq_trans" ;
                     tTranslate table "myf").
```

It is then possible to deduce automatically that $p \centerdot p^{-1} \centerdot p = p$ for all $p : x = y$.

```
Definition free_thm_myf : ∀ A x y p, myf A x y p = p := param_ID myf myfᵗ.
```

5.3 Intensional Function Plugin

Our last illustration is a plugin that provides an intensional flavour to functions and thus allows negating functional extensionality (FunExt). This is a simple example of syntactical translation which enriches the logical power of Coq, as opposed to the parametricity translation which is conservative over CIC. See [8] for an introduction to syntactical translations and a complete description of the intensional function translation.

Even if the translation is very simple as it just adds a boolean to every function (Fig. 5), this time, it is not fully syntax directed. Indeed the notation for pairs hide some types:

```
[fun (x:A) ⇒ t] := pair (∀ x:[A]. ?T) bool (fun (x:[A]) ⇒ [t]) true
```

and we can not recover the type `?T` from the source term. There is thus a mismatch between the lambdas which are not fully annotated and the pairs which are.[8]

However we can use the type inference algorithm of Section 3 implemented on Template-Coq terms to recover the missing information.

```
[fun (x:A) ⇒ t] := let B := infer Σ (Γ, x:[A]) t in
                   pair (∀ (x:[A]). B) bool (fun (x:[A]) ⇒ [t]) true
```

Compared to the parametricity plugin, the translation function has a more complex type as it requires the global and local contexts. However, we can generalize the `tTranslate` command so that it can be used for both the parametricity and the intensional function plugins.

[8] Note that there is a similar issue with applications and projections, but which can be circumvented this time using (untyped) primitive projections.

Extending Coq *using plugins.* The intensional translation extends the logical power of Coq as it is possible for instance to negate FunExt. In this perspective, we defined a new command:

```
Definition tImplement (Σ : global_context ∗ tsl_table) (id : ident) (A : Type)
  : TemplateMonad (global_context ∗ tsl_table).
```

which computes the translation A' of A, then asks the user to inhabit the type A' by generating a proof obligation[9] and then safely adds the axiom id of type A to the current context. By safely, we mean that the correction of the translation ensures that no inconsistencies are introduced.

For instance, here is how to negate FunExt. We use for that two pairs (fun x ⇒ x; true) and (fun x ⇒ x; false) in the interpretation of functions from unit to unit, which are extensionally both the identity, but differ intensionally on their boolean.

```
Run TemplateProgram
  (TC ← tTranslate ([ ],[ ]) "eq" ; TC ← tTranslate TC "False" ;
   tImplement TC "notFunext"
     ((∀ (A B : Set) (f g : A → B), (∀ x:A, f x = g x) → f = g) → False)).
Next Obligation.
  tIntro H. tSpecialize H unit. tSpecialize H unit.
  tSpecialize H (fun x ⇒ x; true). tSpecialize H (fun x ⇒ x; false).
  tSpecialize H (fun x ⇒ eq_refl^t _ _; true).
  apply eq^t_eq in H; discriminate.
Defined.
```

where tIntro and tSpecialize are special versions of the corresponding intro and specialize tactics of Coq to deal with extra booleans appearing in the translated terms. After this command, we have inhabited the translation of the negation of FunExt, which ensures in particular that, providing the translation is correct, the negation of FunExt is consistent with Coq.

6 Related Work and Future Work

Meta-Programming is a whole field of research in the programming languages community, we will not attempt to give a detailed review of related work here. In contrast to most work on meta-programming, we provide a very rough interface to the object language: one can easily build ill-scoped and ill-typed terms in our framework, and staging is basic. However, with typing derivations we provide a way to verify meta-programs and ensure that they do make sense.

The closest cousin of our work is the Typed Syntactic Meta-Programming [11] proposal in Agda, which provides a well-scoped and well-typed interface to a denotation function, that can be used to implement tactics by reflection. We could also implement such an interface, asking for a proof of well-typedness on top of the tmUnquoteTyped primitive of our monad.

[9] In Coq, a proof obligation is a goal which has to be solved to complete a definition. Obligations were introduced by Sozeau [21] in the Program mode.

Intrinsically typed representations of terms in dependent type-theory is an area of active research. Most solutions are based on extensions of Martin-Löf Intensional Type Theory with inductive-recursive or quotient inductive-inductive types [9,2], therefore extending the meta-theory. Recent work on verifying soundness and completeness of the conversion algorithm of a dependent type theory (with natural numbers, dependent products and a universe) in a type theory with IR types [1] gives us hope that this path can nonetheless be taken to provide the strongest guarantees on our conversion algorithm. The intrinsically-typed syntax used there is quite close to our typing derivations.

Another direction is taken by the Œuf certified compiler [18], which restricts itself to a fragment of Coq for which a total denotation function can be defined, in the tradition of definitional interpreters advocated by Chlipala [10]. This setup should be readily accomodated by Template-Coq.

The translation+plugin technique paves the way for certified translations and the last piece will be to prove correctness of such translations. By correctness we mean computational soundness and typing soundness (see [8]), and both can be stated in Template-Coq. Anand has made substantial attempts in this direction to prove the computational soundness, in Template-Coq, of a variant of parametricity providing stronger theorems for free on propositions [3]. This included as a first step a move to named syntax that could be reused in other translations.

Our long term goal is to leverage this technique to extend the logical and computational power of Coq using, for instance, the forcing translation [14] or the weaning translation [19].

When performance matters, we can extract the translation to OCaml and use it like any ordinary Coq plugin. This relies on the correctness of extraction, but in the untyped syntax + typing judgment setting, extraction of translations is almost an identity pretty-printing phase, so we do not lose much confidence. We can also implement a template monad runner in OCaml to run the plugins outside Coq. Our first experiments show that we could gain a factor 10 for the time needed to compute the translation of a term. Another solution would be to use the certified CertiCoq compiler, once it supports a kind of foreign function interface, to implement the `TemplateMonad` evaluation.

The last direction of extension is to build higher-level tools on top of the syntax: the unification algorithm described in [25] is our first candidate. Once unification is implemented, we can look at even higher-level tools: elaboration from concrete syntax trees, unification hints like canonical structures and type class resolution, domain-specific and general purpose tactic languages. A key inspiration in this regard is the work of Malecha and Bengston [16] which implemented this idea on a restricted fragment of CIC.

Acknowledgments

This work is supported by the CoqHoTT ERC Grant 64399.

References

1. Abel, A., Öhman, J., Vezzosi, A.: Decidability of conversion for type theory in type theory. PACMPL 2(POPL), 23:1–23:29 (2018)
2. Altenkirch, T., Kaposi, A.: Type Theory in Type Theory Using Quotient Inductive Types. pp. 18–29. POPL '16, ACM, New York, NY, USA (2016)
3. Anand, A., Morrisett, G.: Revisiting Parametricity: Inductives and Uniformity of Propositions. In: CoqPL'18. Los Angeles, CA, USA (2018)
4. Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O.S., Sozeau, M., Weaver, M.: CertiCoq: A verified compiler for Coq. In: CoqPL. Paris, France (2017)
5. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with Imperative Features and Its Application to SAT Verification. In: Kaufmann, M., Paulson, L.C. (eds.) Interactive Theorem Proving. pp. 83–98. Springer (2010)
6. Barras, B.: Auto-validation d'un système de preuves avec familles inductives. Thèse de doctorat, Université Paris 7 (Nov 1999)
7. Bernardy, J.P., Jansson, P., Paterson, R.: Proofs for free: Parametricity for dependent types. Journal of Functional Programming 22(2), 107–152 (Mar 2012)
8. Boulier, S., Pédrot, P.M., Tabareau, N.: The Next 700 Syntactical Models of Type Theory. In: CPP'17, Paris, France. pp. 182–194. ACM (2017)
9. Chapman, J.: Type Theory Should Eat Itself. Electronic Notes in Theoretical Computer Science 228, 21 – 36 (2009), proceedings of LFMTP 2008
10. Chlipala, A.: Certified Programming with Dependent Types, vol. 20. MIT Press (2011)
11. Devriese, D., Piessens, F.: Typed syntactic meta-programming. In: ICFP (2013)
12. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A Metaprogramming Framework for Formal Verification pp. 34:1–34:29 (Sep 2017)
13. Forster, Y., Kunze, F.: Verified Extraction from Coq to a Lambda-Calculus. In: Coq Workshop 2016 (2016)
14. Jaber, G., Lewertowski, G., Pédrot, P.M., Sozeau, M., Tabareau, N.: The Definitional Side of the Forcing. In: LICS'16, New York, NY, USA. pp. 367–376 (2016)
15. Lasson, M.: Canonicity of Weak ω-groupoid Laws Using Parametricity Theory. Electronic Notes in Theoretical Computer Science 308, 229–244 (2014)
16. Malecha, G., Bengtson, J.: ESOP 2016, chap. Extensible and Efficient Automation Through Reflective Tactics, pp. 532–559. Springer Berlin Heidelberg (2016)
17. Malecha, G.M.: Extensible Proof Engineering in Intensional Type Theory. Ph.D. thesis, Harvard University (2014)
18. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: minimizing the Coq extraction TCB. In: Proceedings of CPP 2018. pp. 172–185 (2018)
19. Pédrot, P., Tabareau, N.: An effectful way to eliminate addiction to dependence. In: LICS'17, Reykjavik, Iceland. pp. 1–12 (2017)
20. Sheard, T., Jones, S.P.: Template Meta-programming for Haskell. SIGPLAN Not. 37(12), 60–75 (Dec 2002)
21. Sozeau, M.: Program-ing Finger Trees in Coq. pp. 13–24. ICFP '07, ACM, New York, NY, USA (2007)
22. Taha, W., Sheard, T.: Multi-stage Programming with Explicit Annotations. pp. 203–217. PEPM '97, ACM, New York, NY, USA (1997)
23. Van der Walt, P., Swierstra, W.: Engineering Proof by Reflection in Agda. In: Hinze, R. (ed.) Implementation and Application of Functional Languages. pp. 157–173. Springer Berlin Heidelberg (2013)

24. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A Monad for Typed Tactic Programming in Coq. Journal of Functional Programming 25 (2015)
25. Ziliani, B., Sozeau, M.: A Comprehensible Guide to a New Unifier for CIC Including Universe Polymorphism and Overloading. Journal of Functional Programming 27, e10 (2017)