The MetaCoq Project

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, Théo Winterhalter

Noname manuscript No.
(will be inserted by the editor)

The MetaCoq Project

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau and Théo Winterhalter

Received: date / Accepted: date

Abstract The MetaCoq project[1] aims to provide a certified meta-programming environment in Coq. It builds on Template-Coq, a plugin for Coq originally implemented by Malecha (2014), which provided a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Recently, it was used in the CertiCoq certified compiler project (Anand et al., 2017), as its front-end language, to derive parametricity properties (Anand and Morrisett, 2018). 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 Polymorphic Calculus of Cumulative Inductive Constructions (pCUIC), 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 and a certifying extraction to call-by-value λ-calculus. We also advocate the use of MetaCoq as a foundation for higher-level tools.

M. Sozeau
Pi.R2 Project-Team, Inria Paris and IRIF, France

S. Boulier, N. Tabareau, T. Winterhalter
Gallinette Project-Team, Inria Nantes, France

C. Cohen
Université Côte d'Azur, Inria, France

Y. Forster, F. Kunze
Saarland University, Germany

A. Anand, G. Malecha
BedRock Systems, USA

[1] `https://metacoq.github.io/metacoq`

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 allows 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 (Chapman, 2009; Altenkirch and Kaposi, 2016). 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 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 unification of the syntax and its well-typedness makes it 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 (Devriese and Piessens, 2013; Van der Walt and Swierstra, 2013) (the introduction of (Devriese and Piessens, 2013) provides a comprehensive review of related work in this area). This can be seen as a type-theoretic version of the functional programming language designs such as Template Haskell (Sheard and Jones, 2002a) or MetaML (Taha and Sheard, 1997). This is also the approach taken by Malecha in his thesis (Malecha, 2014) where he introduced 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 (Anand et al., 2017), which uses it as its front-end language. It was also used by Anand and Morrisett (2018) to formalize a modified parametricity translation, and to extract Coq terms to a CBV λ-calculus (Forster and Kunze, 2016). 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.

The MetaCoq project described in this paper remedies this situation by providing a formal semantics of Coq's 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 access to Coq logical environments. We achieve this using the `TemplateMonad`, which 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 MTac project (Ziliani et al., 2015), Lean's tactic monad (Ebner et al., 2017), or Agda's reflection framework (Van der Walt and Swierstra, 2013), our ultimate goal is not to interface with Coq's unification and type-checking algorithms, but to provide a self-hosted, bootstrappable and verifiable implementation of these algorithms. This opens the possibility to verify the kernel's implementation, a problem tackled by Barras (1999) using set-theoretic models. In addition, we advocate for the use of MetaCoq as a foundation to build higher-level tools. For example, translations, boilerplate generators, domain-specific proof languages, or even general purpose tactic languages.

Terminologically, we reserve the use of the name Template-Coq to denote reification of the internal syntax and logical environment of Coq, and also for the reification of the type-checking algorithm. We otherwise use the name MetaCoq when talking about definition of the formal semantics and certification of the algorithms.

Outline of the paper. In Section 2, we present the complete reification of Coq terms, covering the entire CIC and present a formal specification of typing derivations of these terms. In Section 3, we show the definition of the `TemplateMonad` for general manipulation of Coq's logical environment and use it to define plugins for various translations from Coq to Coq or λ-calculus (Section 4). Section 5 covers a modification to `TemplateMonad` that enables plugins to be run natively in OCaml. Finally, we discuss related and future work in Section 6.

2 A Formal Specification of Coq

In this section, we give a formal specification for Coq by giving syntax and semantics. We will proceed as follows. First, we give the syntax of Coq terms (Section 2.1) and environments (Section 2.2):

```
term : Set          context : Set
```

Then, we give the formal semantics of those terms by defining the typing relation (Section 2.3), the reduction relation and the conversion relation (Section 2.4):

```
typing : context → term → term → Type
red    : context → term → term → Type
conv   : context → term → term → Type
```

Finally, Section 2.5 is devoted to typing environment and inductive types while Section 2.6 explains the management of universes.

2.1 Reification of Terms

The central piece of MetaCoq is the inductive type `term` (Figure 1) which represents the syntax of Coq terms (this language is called Gallina). This inductive follows directly the `constr` datatype of Coq terms in the implementation of Coq, except

```
Inductive term : Set :=
| tRel       (n : nat)
| tVar       (id : ident)
| tEvar      (ev : nat) (args : list term)
| tSort      (s : universe)
| tCast      (t : term) (kind : cast_kind) (v : term)
| tProd      (na : name) (ty : term) (body : term)
| tLambda    (na : name) (ty : term) (body : term)
| tLetIn     (na : name) (def : term) (def_ty : term) (body : term)
| tApp       (f : term) (args : list term)
| tConst     (c : kername) (u : universe_instance)
| tInd       (ind : inductive) (u : universe_instance)
| tConstruct (ind : inductive) (idx : nat) (u : universe_instance)
| tCase      (ind_and_nbparams : inductive * nat) (type_info : term)
             (discr : term) (branches : list (nat * term))
| tProj      (proj : projection) (t : term)
| tFix       (mfix : mfixpoint term) (idx : nat)
| tCoFix     (mfix : mfixpoint term) (idx : nat).
```

Fig. 1 MetaCoq's representation of Coq terms mirrors Coq's `constr` type.

for the use of OCaml's native arrays and strings[2]. Some familiar constructions are recognizable: sorts, lambdas, applications, ... Let's review the different constructors.

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:

```
Definition ident := string.
Inductive  name  :=  nAnon | nNamed (_ : ident).
```

Sorts are represented with `tSort`, which takes a `universe` as argument. A universe can be either `Prop`, `Set` or a more complex expression representing one of the `Type` universes. The details are given in Section 2.6.

Type casts `(t : A)` are given by `tCast`.

n-ary application is introduced by `tApp`. In `tApp t l`, `t` is expected not to be an application, and l to be a non-empty list.

Example 1 The function `fun (f : Set → Set) (A : Set) ⇒ f A` is represented by:

```
tLambda (nNamed "f")
  (tProd nAnon (tSort [(Level.lSet, false)]) (tSort [(Level.lSet, false)]))
  (tLambda (nNamed "A") (tSort [(Level.lSet, false)]) (tApp (tRel 1) [tRel 0]))
```

The three constructors `tConst`, `tInd` and `tConstruct` represent references to constants declared in a global environment. The first is for definitions or axioms, the second for inductive types, and the last for constructors of inductive types. In Coq, constants can be universe polymorphic, meaning that they can be used at different universe levels. In such a case, said universe levels are given in the `universe_instance` which is a list of levels. If the constant is not universe polymorphic, the instance is expected to be empty.

`tCase` represents pattern-matchings, `tProj` primitive projections, `tFix` fixpoints and `tCoFix` cofixpoints.

[2] An upcoming extension of Coq (Armand et al., 2010) with such features could address this mismatch.

Example 2 The addition on natural numbers

```
Fixpoint add (a b : nat) : nat :=
  match a with
    | 0 => b
    | S a => S (add a b)
  end.
```

Is represented by:

```
tFix [{|
   dname := nNamed "add";
   dtype := tProd (nNamed "a") (tInd inat [])
              (tProd (nNamed "b") (tInd inat []) (tInd inat []));
   dbody := tLambda (nNamed "a") (tInd inat [])
              (tLambda (nNamed "b") (tInd inat [])
                 (tCase (inat, 0)
                    (tLambda (nNamed "a") (tInd inat []) (tInd inat []))
                    (tRel 1)
                    [(0, tRel 0);
                     (1, tLambda (nNamed "a") (tInd inat [])
                                 (tApp (tConstruct inat 1 [])
                                       [tApp (tRel 3) [tRel 0; tRel 1]]))]));
   rarg := 0 |}] 0
```

where `inat` is a notation for the inductive representing `nat`:

```
{| inductive_mind := "Coq.Init.Datatypes.nat"; inductive_ind := 0 |}
```

`tVar` is for named variables introduced in Coq sections or during interactive proofs. `tEvar` represents for existential variables, *i.e.* holes to be filled in terms. Typing of these two constructions is not defined in MetaCoq for the moment.

2.2 Reification of environment

In Coq, the meaning of a term is relative to an environment, which must be reified as well. We distinguish the global environment which is constant through a typing derivation, from the local context which may vary. The type of typing relation is:

```
typing : global_context → context → term → term → Type
```

(similar for `red` and `conv`)

The *local context* records the types and potential bodies (for *let-ins*) of de Bruijn indexes:

```
Record context_decl := mkdecl {
  decl_name : name ;
  decl_body : option term ;
  decl_type : term
}.
Definition context := list context_decl.
```

The de Bruijn index 0 is bound to the head of the list. Contexts are written in *snoc* order: we use the notation `Γ ,, d` for adding `d` to the head of `Γ`. We also use the abbreviations `vass x A` and `vdef x t A` for the two ways to build a `context_decl` (with or without a body). Last, we use the notation `Γ ,,, Γ'` for context concatenation.

Remark 1 Contrarily to MetaCoq, in the OCaml code of Coq de Bruijn indices start at 1 for historical reasons.

The *global environment* consists of two parts: the graph of universes (described in Section 2.6) and a list of declarations, properly ordered according to dependencies.

```
Definition global_declarations := list global_decl
Definition global_context := global_declarations * uGraph.t.
```

A declaration is either the declaration of a constant (a definition or an axiom, according to the presence of body) or of a block of mutual inductive types (which brings both the inductive types and their constructors to the context).

```
Inductive global_decl :=
| ConstantDecl  : kername → constant_body         → global_decl
| InductiveDecl : kername → mutual_inductive_body → global_decl.
```

The kernel name `kername` is a fully qualified name (among modules), for instance the kernel name corresponding to `nat` is `Coq.Init.Datatypes.nat`. `kername` as a type is a synonym to `string`.

The declaration of a constant is fairly easy:

```
Record constant_body := {
  cst_type : term;
  cst_body : option term;
  cst_universes : universe_context
}.
```

The `universe_context` indicates whether the constant is polymorphic or not. If so, it contains the constraints that the universe instances have to satisfy.

Declarations of inductives are more involved, they are described in Section 2.5.

2.3 Typing judgements

Now that we have terms and environments, we can describe formally all the typing rules of Coq. This is done by defining an inductive family `typing` whose definition looks like:

```
Inductive typing (Σ : global_context) (Γ : context) : term → term → Set :=
| type_Rel n :
  All_local_env typing Σ Γ →
  nth_error Γ n = Some decl →
  Σ ;;; Γ ⊢ tRel n : lift0 (S n) decl.(decl_type)

| type_Sort (l : level) :
  All_local_env typing Σ Γ →
  Σ ;;; Γ ⊢ tSort (Universe.make l) : tSort (Universe.super l)

| ...

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_cons hd tl na A B s T B' :
  Σ ;;; Γ ⊢ tProd na A B : tSort s →
  Σ ;;; Γ ⊢ T ≤ tProd na A B →
  Σ ;;; Γ ⊢ hd : A →
  typing_spine Σ Γ (subst10 hd B) tl B' →
  typing_spine Σ Γ T (hd :: tl) B'.
```

The typing rules include the basic dependent λ-calculus with let-bindings, global references to inductives and constants, pattern-maching, primitive projections and (co)fixed-points. Universe polymorphic definitions and the well-formedness judgment for global declarations are dealt with as well. The only ingredients missing are the termination check for fixed-points and productivity check for cofixed-points. They are work-in-progress.

Note that the typing rules use substitution and lifting operations of de Bruijn indexes (`lift0`, `subst`, ...), their definitions are standard. The typing relation also relies on the subtyping relation. It is described in Section 2.4.

We shall now take time to explain in details the rules one by one.

Variables. A variable is well typed when its de Bruijn index corresponds to a declaration in the (local) context Γ. The following rule is not saying much more despite its looks.

```
type_Rel n decl :
  All_local_env typing Σ Γ →
  nth_error Γ n = Some decl →
  Σ ;;; Γ ⊢ tRel n : lift0 (S n) decl.(decl_type)
```

`decl` is a declaration of type `context_decl`. The rule attests that the `n`th variable corresponds to the `n`th most recent declaration in the context and thus has the ascribed type. The latter is however *lifted* because the context contains `n` declarations after it:

```
Γ = Δ, decl_n, ..., decl₁, decl₀
```

with `decl_n` typed in Δ, so Γ is Δ extended with `S n` declarations, hence the `lift0 (S n)`. Finally, `All_local_env typing` Σ Γ is asserting that the local context Γ is well-formed in global context Σ. Later on this property is called `wf_local` Σ Γ but here the dependency on `typing` is being made explicit.

Sorts. Any sort corresponding to a level (without a `+1`) can be typed with its successor universe (with a `+1`), provided the context is well-formed.

```
type_Sort l :
  All_local_env typing Σ Γ →
  Σ ;;; Γ ⊢ tSort (Universe.make l) : tSort (Universe.super l)
```

Remark 2 With this rule, only non-algebraic universes can be typed (see Section 2.6 for the definition of non-algebraic universes).

Type-casts. In Coq, a type-cast happens when you give a type explicitly to an expression: `(t : A)`. `t` is checked to have type `A` and the whole expression is also typed with `A`.

```
type_Cast t k A s :
  Σ ;;; Γ ⊢ A : tSort s →
  Σ ;;; Γ ⊢ t : A →
  Σ ;;; Γ ⊢ tCast t k A : A
```

In the rule it is required that `A` is *well-sorted*, meaning that there exists (constructively) a sort `s` such that `A` is of type `tSort s`. In Coq's kernel, the `k : cast_kind` indicates which algorithm is used to check the conversion between `A` and the type of `t`. We ignore it for the moment in MetaCoq.

Dependent products. The dependent product, or Π-type, `∀ x : A, B` is well typed when both `A` and `B` are well typed (the latter in the context extended with assumption `x : A`).

```
type_Prod n A B s1 s2 :
  Σ ;;; Γ ⊢ A : tSort s1 →
  Σ ;;; Γ ,, vass n A ⊢ B : tSort s2 →
  Σ ;;; Γ ⊢ tProd n A B : tSort (Universe.sort_of_product s1 s2)
```

The sort in which the product lives in roughly the maximum of the sorts of its components, accounting for impredicativity of `Prop`:

```
Definition sort_of_product domsort rangsort :=
  match (domsort, rangsort) with
  | (_, [(Level.lProp,false)])  ⇒ rangsort
  | (u1, u2) ⇒ Universe.sup u1 u2
  end.
```

λ-abstractions. Similarly the rule governing the typing of `fun x : A ⇒ t` is not surprising.

```
type_Lambda n n' A t s1 B :
  Σ ;;; Γ ⊢ A : tSort s1 →
  Σ ;;; Γ ,, vass n A ⊢ t : B →
  Σ ;;; Γ ⊢ tLambda n A t : tProd n' A B
```

Notice the names `n` and `n'` that are different in the term and the type; they are only printing *hints* and are irrelevant to typing, which is why this rule doesn't force them to be the same.

`let in` expression. `tLetIn x b B t` reifies `let x := b : B in t` for which typing is pretty straightforward. Assuming `t : A` the whole expression has type `let x := b : B in A`.

```
type_LetIn x b B t s1 A :
  Σ ;;; Γ ⊢ B : tSort s1 →
  Σ ;;; Γ ⊢ b : B →
  Σ ;;; Γ ,, vdef x b B ⊢ t : A →
  Σ ;;; Γ ⊢ tLetIn x b B t : tLetIn x b B A
```

Applications. Typing applications is usually simple, but because MetaCoq features n-ary applications, we need to be careful when handling them.

```
type_App t l t_ty t' :
  Σ ;;; Γ ⊢ t : t_ty →
  ~ (isApp t = true) → l ≠ [] → (* Well-formed application *)
  typing_spine Σ Γ t_ty l t' →
  Σ ;;; Γ ⊢ tApp t l : t'
```

The conditions `~ (isApp t = true)` and `l ≠ []` ensure that the application is well-formed: that is `t` is not a nested application and it is applied to at least one argument. Then `typing_spine Σ Γ t_ty l t'` states that a term of type `t_ty` applied to a list of arguments `l` will return a term of type `t'`. Let's have a closer look at it:

```
typing_spine Σ Γ : term → list term → term → Prop :=
| type_spine_nil ty : typing_spine Σ Γ ty [] ty
| type_spine_cons hd tl na A B s T B' :
  Σ ;;; Γ ⊢ tProd na A B : tSort s →
  Σ ;;; Γ ⊢ T ≤ tProd na A B →
  Σ ;;; Γ ⊢ hd : A →
  typing_spine Σ Γ (subst10 hd B) tl B' →
  typing_spine Σ Γ T (hd :: tl) B'.
```

Basically, it iterates over every argument of the function, checking each time that the new function has a function type and is being applied to something in its domain. The argument is then substituted in the codomain which then is matched against a function type again, until there are no arguments left and the type can be returned as is.

Global constants. A constant can either refer to a global definition (stemming from `Definition` or `Lemma` for instance), or to an axiom (`Axiom`). It has a name which is a `kername`. Such a declaration can be universe polymorphic, so when referring to a constant, one needs to provide it with a universe instance (*i.e.* values for the universe variables in the definition).

```
type_Const cst u :
  All_local_env typing Σ Γ →
  ∀ decl (isdecl : declared_constant (fst Σ) cst decl),
  consistent_universe_context_instance (snd Σ) decl.(cst_universes) u →
  Σ ;;; Γ ⊢ tConst cst u : subst_instance_constr u decl.(cst_type)
```

For a constant to be well typed, it first needs to indeed refer to a declared constant in the global context Σ, which is checked by `declared_constant (fst Σ) cst decl`, a synonym to `lookup_env (fst Σ) cst = Some (ConstantDecl cst decl)`.

`consistent_universe_context_instance` has a self-explanatory name: it checks that the instance is indeed an instance and verifies that if satisfies the constraints. The constant can thus be typed with the type found in the context `decl.(cst_type)`, where the universes are substituted with the instance.

Inductive types. Typing an inductive type is very similar to typing a constant. This time `ind` is of type `inductive` which consists of a `kername` (the name of the mutual-inductive block) and a natural number (the index of the considered inductive type in the block, starting at 0). Similarly to constants, inductive types can be universe polymorphic.

```
type_Ind ind u :
  All_local_env typing Σ Γ →
  ∀ mdecl idecl (isdecl : declared_inductive (fst Σ) mdecl ind idecl),
  consistent_universe_context_instance (snd Σ) mdecl.(ind_universes) u →
  Σ ;;; Γ ⊢ tInd ind u : subst_instance_constr u idecl.(ind_type)
```

Inductives are declared in the global context as well. `mdecl` corresponds to the mutual block and `idecl` corresponds to the inductive of that block we're interested in. `declared_inductive` checks that `ind` indeed corresponds to these declarations in Σ.

Constructors of an inductive type. Inductive types come with their constructors. If the inductive type is declared, and the constructor is indeed a constructor, then it is welltyped.

```
type_Construct ind i u :
  All_local_env typing Σ Γ →
  ∀ mdecl idecl cdecl
    (isdecl : declared_constructor (fst Σ) mdecl idecl (ind, i) cdecl),
    consistent_universe_context_instance (snd Σ) mdecl.(ind_universes) u →
    Σ ;;; Γ ⊢ tConstruct ind i u : type_of_constructor mdecl cdecl (ind, i) u
```

However, this time the constructor types come under the context corresponding to the mutual inductive types. Take for instance the mutual inductive types `even` and `odd`:

```
Inductive even : nat → Prop :=
| even0 : even 0
| evenS : ∀ n, odd n → even (S n)

with odd : nat → Prop :=
| oddS : ∀ n, even n → odd (S n).
```

In this case, `evenS` is typed in context `even : nat → Prop, odd : nat → Prop`, which is why it can refer to both types, even before they are defined.

The purpose of `type_of_constructor` is thus to substitute these variables by their actual definitions, as well as instantiating the universes.

Pattern matching. In the internals of Coq and MetaCoq, pattern-matching is refered to as `tCase`. Dependent pattern-matching with general inductive types is no small task so we shall try and break down the typing rule, and the `tCase` constructor.

```
type_Case ind u npar p c brs args :
  ∀ mdecl idecl
    (isdecl : declared_inductive (fst Σ) mdecl ind idecl),
    mdecl.(ind_npars) = npar →
    let pars := List.firstn npar args in
    ∀ pty, Σ ;;; Γ ⊢ p : pty →
    ∀ indctx pctx ps btys,
      types_of_case ind mdecl idecl pars u p pty =
      Some (indctx, pctx, ps, btys) →
      check_correct_arity (snd Σ) idecl ind u indctx pars pctx = true →
      Exists (fun sf ⇒ universe_family ps = sf) idecl.(ind_kelim) →
      Σ ;;; Γ ⊢ c : mkApps (tInd ind u) args →
      All2 (fun x y ⇒ (fst x = fst y) * (Σ ;;; Γ ⊢ snd x : snd y)) brs btys →
      Σ ;;; Γ ⊢ tCase (ind, npar) p c brs : mkApps p (List.skipn npar args ++ [c
     ])
```

In `tCase (ind, npar) p c brs`, `ind` is inductive type of the scrutinee `c`, `npar` is the number of parameters of the inductive (arguments that are constant across all the constructors), `p` is the predicate or return type, while `brs` is a list of branches comprised of the number of arguments of the constructor and the term corresponding to the branch (with abstractions for the arguments of the constructor). For instance, consider the following pattern-matching:

```
fun m P (P0 : P 0) (PS : ∀ n, P (S n)) ⇒
  match m as n return P n with
  | 0 ⇒ P0
  | S n ⇒ PS n
  end.
```

Ignoring the λs, it is quoted to

```
tCase
  (inat, 0)
  (tLambda (nNamed "n") (tInd inat []) (tApp (tRel 3) [ tRel 0 ]))
  (tRel 3) [
    (0, tRel 1) ;
    (1, tLambda (nNamed "n") (tInd inat []) (tApp (tRel 1) [ tRel 0 ]))
  ]
```

Let's focus on the rule now. As we did for inductive types, we check that the inductive type of the scrutinee is declared.

Σ `;;;` $\Gamma \vdash$ `c : mkApps (tInd ind u) args` checks that the scrutinee `c` is indeed in the right type, i.e., the inductive applied to some arguments. After checking that `npar` is indeed the number of parameters of the inductive type (`mdecl.(ind_npars) = npar`), we take them off the list of arguments (`pars := List.firstn npar args`). The rest are the indices of the inductive type and may vary depending on the branch.

Additionally, we check that the predicate (or return type) is well typed with Σ `;;;` $\Gamma \vdash$ `p : pty`.

`types_of_case` has the purpose of producing the typing information required to type the branches:

– `indctx` corresponds to the context of the inductive type where the parameters have been instantiated by `pars`, it thus contains only the indices, (e.g. `y : A` when matching against `p : @eq A u v`, `A` and `u` being the parameters);
– `pctx` is the same but for the type of the predicate `p` (in the example above it would just be `n : nat`);
– `ps` is the sort targeted by `p` (basically `p` quantifies over `pctx` to return `ps`—in particular it forces `p` to be a type once fully applied);
– `btys` is a list containing the expected type for each element of `brs`, the branches.

`check_correct_arity` verifies that `pctx` is equal (modulo α-renaming) to `indctx` extended with a variable of the inductive applied to the parameters `pars` and the variables of context `indctx`.

Then, `Exists (fun sf ⇒ universe_family ps = sf) idecl.(ind_kelim)` attests that the sort of the predcate `ps` belongs to one the universe families that the inductive type can be eliminated to (`ind_kelim`). The universe family may be `Prop`, `Set` or `Type` and some inductives have restrictions for elimination; most inductive types defined in `Prop` can only be eliminated into `Prop` itself, the only to bypass this restriction is using the so-called *singleton elimination*.

Finally, with `All2` we iterate over both `brs` and `btys` to check that the branches are indeed typed according to what is recorded in `btys`, all the while checking that they agree on the number of arguments of the constructors (with the `fst` part).

Primitive projections. In Coq there are two notions of record types. By default, when one defines the following record:

```
Record T := mk { pi₁ : bool ; pi₂ : nat }.
```

it is actually equivalent to the inductive type with one constructor

```
Inductive T := mk (pi₁ : bool) (pi₂ : nat).
```

along with the definitions of `pi`$_1$ and `pi`$_2$ by pattern-matching.

It however possible to define records in a more primitive way. Using the global option `Set Primitive Projections`, the former record definition is still internally represented as an inductive, but this time, additionally to constructors, it has projections, corresponding to `pi`$_1$ and `pi`$_2$. Projections can be called with the syntax `t.(pi`$_1$`)` or as regular functions.

```
type_Proj p c u :
  ∀ mdecl idecl pdecl
    (isdecl : declared_projection (fst Σ) mdecl idecl p pdecl) args,
    Σ ;;; Γ ⊢ c : mkApps (tInd (fst (fst p)) u) args →
    #|args| = ind_npars mdecl →
    let ty := snd pdecl in
    Σ ;;; Γ ⊢ tProj p c
            : subst0 (c :: List.rev args) (subst_instance_constr u ty)
```

As usual, `declared_projection` checks that Σ contains both the inductive and the projection declaration. The projection is applied to a term `c` of the record as ensured by the condition:

```
Σ ;;; Γ ⊢ c : mkApps (tInd (fst (fst p)) u) args
```

Here `projection` stands for `inductive * nat * nat`, that is an inductive, a number of parameters and the index of the projected argument. We verify that the inductive is fully applied with `#|args| = ind_npars mdecl`, stating that the number of arguments corresponds to the number of parameters of the inductive type. Finally, we substitute these arguments, `c`, and the universes in the type of the projection to get the type of the term.

Fixed-points. In Coq, the fixed-point operator is primitive and completes pattern-matching for performing induction. One usually writes a fixed-point using the aptly named command `Fixpoint`. It is however possible to write them directly in a term with `fix`. Let's consider the following mutual fixed-point:

```
fix  f1 (x1:X11) ... (xn1:X1n1) {struct xk1} : A1 := t1
with ...
with fn (x1:Xn1) ... (xnn:Xnnn) {struct xkn} : An := tn
for fj
```

This fixed-point will be of type `∀ (x1:Xj1) ... (xnj:Xjnj), Aj`. For it to be well typed there are three conditions:

- Each `Ai` has to be a type;
- Each `ti` has to be of type `Ai` in a context extended by the signatures of the fixed-points (allowing the recursive calls in the body):

$$\Gamma, f_1 : A_1, \ldots f_n : A_n, x_1 : X_{i1}, \ldots x_{n_i} : X_{in_i} \vdash t_i : A_i;$$

- A termination criterion has to be fulfilled. Such a criterion has not yet been implemented in MetaCoq.

Internally, a fixed-point is represented with `tFix mfix idx` where `mfix : list (def term)` represents the mutual fixed-points, and `idx : nat` specifies which of them we want to refer to. `def` is the following record:

```
Record def (term : Set) : Set := mkdef {
  dname : name; (* the name fi **)
  dtype : term; (* the type Ai **)
  dbody : term; (* the body ti (a lambda-term).
                   Note, this may mention other (mutually-defined) names **)
  rarg  : nat   (* the index ki of the recursive argument, 0 for cofixpoints **)
}.
```

The formal typing rule is the following:

```
type_Fix mfix n decl :
  let types := fix_context mfix in
  nth_error mfix n = Some decl →
  All_local_env typing Σ (Γ ,,, types) →
  All (fun d ⇒
    Σ ;;; Γ ,,, types ⊢ d.(dbody) : lift0 #|types| d.(dtype)) *
    (isLambda d.(dbody) = true
  ) mfix →
  Σ ;;; Γ ⊢ tFix mfix n : decl.(dtype)
```

First, we build a context containing the assumptions of the different definitions with `types := fix_context mfix`, and verify that the composite context `Γ ,,, types` is well-formed. Then we check that `idx` indeed corresponds to one of the definitions of the block (`nth_error mfix n = Some decl`). Finally, for each of the definitions, we check that the body has the ascribed type (in the extended context, hence the `lift0`) and that they all correspond to functions. The return type is the ascribed type.

Cofixed-points. Co-fixed-points are handled in a very similar fashion to regular fixed-points. Even their representation is the same. Again, productivity conditions remain unchecked for the time being.

```
type_CoFix mfix n decl :
  let types := fix_context mfix in
  nth_error mfix n = Some decl →
  All_local_env typing Σ (Γ ,,, types) →
  All (fun d ⇒
    Σ ;;; Γ ,,, types ⊢ d.(dbody) : lift0 #|types| d.(dtype)
  ) mfix →
  Σ ;;; Γ ⊢ tCoFix mfix n : decl.(dtype)
```

Conversion rules. We conclude with the usual conversion rule.

```
type_Conv t A B s :
  Σ ;;; Γ ⊢ t : A →
  Σ ;;; Γ ⊢ B : tSort s →
  Σ ;;; Γ ⊢ A ≤ B →
  Σ ;;; Γ ⊢ t : B
```

It is here stated with cumulativity (allowing to increase universes in contravariant positions), and it requires the new type to be well-sorted as well. We shall explain conversion and cumulativity in more details in the next subsection.

2.4 Conversion, Cumulativity and Reduction

The cumulativity, or subtyping, relation, is defined from one-step reduction `red1` as follows:

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

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

Basically, `A ≤ B` when `A` and `B` respectively reduce to `A'` and `B'` such that cumulativity can be checked syntactically with `leq_term`. `leq_term` operates as a congruence and invokes universe comparison when reaching sorts.

Conversion is derived from cumulativity going both ways:

```
Definition conv Σ Γ T U :=
  (Σ ;;; Γ ⊢ T ≤ U) * (Σ ;;; Γ ⊢ U ≤ T).

Notation " Σ ;;; Γ ⊢ t = u " := (conv Σ Γ t u).
```

It is equivalent to having both terms reduce to α-convertible terms.

The main point of interest is thus how one-step reduction `red1` is defined. It is introduced with the following command:

```
Inductive red1 (Σ : global_declarations) (Γ : context) : term → term → Type
```

however, we will not put here all of its constructors. Most of them are congruence rules. For instance, for `tLambda`, the congruences are as follows.

```
| abs_red_l na M M' N :
  red1 Σ Γ M M' →
  red1 Σ Γ (tLambda na M N) (tLambda na M' N)
| abs_red_r na M M' N :
  red1 Σ (Γ ,, vass na N) M M' →
  red1 Σ Γ (tLambda na N M) (tLambda na N M')
```

A term reduces to another in one step, if one of its subterms does. It holds for all term constructors so we will now focus on actual computation rules.

β-reduction. A λ-abstraction may consume its first argument to reduce.

```
red_beta na t b a l :
  red1 Σ Γ (tApp (tLambda na t b) (a :: l)) (mkApps (subst10 a b) l)
```

`let` expressions. A let expression can be unfolded as a substitution right away (this is called ζ-reduction):

```
red_zeta na b t b' :
  red1 Σ Γ (tLetIn na b t b') (subst10 b b')
```

It can also be unfolded later, by reducing a reference to the `let`-binding:

```
red_rel i body :
  option_map decl_body (nth_error Γ i) = Some (Some body) →
  red1 Σ Γ (tRel i) (lift0 (S i) body)
```

It checks that the `i`th variable in Γ corresponds to a definition and replaces the variable with it. It needs to be lifted because the body was defined in a smaller context.

Pattern-matching. A `match` expression can be reduced with ι-reduction when the scrutinee is a constructor.

```
red_iota ind pars c u args p brs :
  red1 Σ Γ (tCase (ind, pars) p (mkApps (tConstruct ind c u) args) brs)
           (iota_red pars c args brs)
```

Herein, `iota_red` is defined as follows:

```
Definition iota_red npar c args brs :=
  mkApps (snd (List.nth c brs (0, tDummy))) (List.skipn npar args).
```

As `List.nth` takes a default value, `(0, tDummy)` can be ignored, it basically picks the branch corresponding to the constructor and applies it to the indices of the inductive (`List.skipn npar args`).

Fixed-point unfolding. Even after they are checked to be terminating, fixed-points cannot be unfolded indefinitely. There is a syntactic guard to only unfold a fixed-point when its recursive argument is a constructor.

```
red_fix mfix idx args narg fn :
  unfold_fix mfix idx = Some (narg, fn) →
  is_constructor narg args = true →
  red1 Σ Γ (tApp (tFix mfix idx) args) (tApp fn args)
```

`unfold_fix mfix idx` allows to recover both the body (`fn`) and the index of the recursive argument (`narg`) while `is_constructor narg args` checks that the said argument is indeed a constructor.

Co-fixed-point unfolding. There are two cases where a co-fixed-point gets unfolded. One of them is when it is matched against.

```
red_cofix_case ip p mfix idx args narg fn brs :
  unfold_cofix mfix idx = Some (narg, fn) →
  red1 Σ Γ (tCase ip p (mkApps (tCoFix mfix idx) args) brs)
           (tCase ip p (mkApps fn args) brs)
```

As for fixed-points, `unfold_cofix` returns the body.

A co-fixed-point can also be unfolded when projected, behaving exactly the same way.

```
red_cofix_proj p mfix idx args narg fn :
  unfold_cofix mfix idx = Some (narg, fn) →
  red1 Σ Γ (tProj p (mkApps (tCoFix mfix idx) args))
           (tProj p (mkApps fn args))
```

δ-reduction. δ-reduction allows to unfold a constant (from the global context Σ).

```
red_delta c decl body (isdecl : declared_constant Σ c decl) u :
  decl.(cst_body) = Some body →
  red1 Σ Γ (tConst c u) (subst_instance_constr u body)
```

It can only be done if a definition is indeed found. Its universes (if it is universe polymorphic) are then instantiated.

Projection. When a constructor of a record is projected, it can be reduced to the corresponding field.

```
red_proj i pars narg args k u arg :
  nth_error args (pars + narg) = Some arg →
  red1 Σ Γ (tProj (i, pars, narg) (mkApps (tConstruct i k u) args)) arg
```

2.5 Typing environments

Local environment. As already mentionned in the typing rules, a local context Γ is wellformed if `wf_local Σ Γ` holds. This type is an abbreviation of `All_local_env typing Σ Γ` where `All_local_env` is defined by:

```
Inductive All_local_env (Σ : global_context) : context → Type :=
| localenv_nil :
  All_local_env Σ []

| localenv_cons_abs Γ na t u :
  All_local_env Σ Γ →
  typing Σ Γ t (tSort u) →
  All_local_env Σ (Γ ,, vass na t)

| localenv_cons_def Γ na b t :
  All_local_env Σ Γ →
  typing Σ Γ b t →
  All_local_env Σ (Γ ,, vdef na b t).
```

Hence, the empty context is well-formed. A variable assumption is well-formed if the type is well-sorted and a variable definition is well-formed if the body is indeed of the given type.

The well-typedness of the local context is enforced in every typing judgment:

```
typing_wf_local: ∀ Σ Γ t T, wf Σ → Σ ;;; Γ ⊢ t : T → wf_local Σ Γ
```

Global environment. As opposed to local contexts, the well-typedness of the global environment is *not* enforced in typing judgments and have thus to be stated additionally with the predicate `wf Σ` (as above for instance). This predicate is defined as `on_global_decls (fst Σ) (snd Σ)` where we have:

```
Definition on_constant_decl Σ d :=
  match d.(cst_body) with
  | Some trm ⇒ typing Σ [] trm d.(cst_type)
  | None ⇒ {u : universe & typing Σ [] d.(cst_type) (tSort u)}
  end.

Definition on_global_decl Σ decl :=
  match decl with
  | ConstantDecl id d ⇒ on_constant_decl Σ d
  | InductiveDecl ind inds ⇒ on_inductive Σ ind inds
  end.

Inductive on_global_decls φ : global_declarations → Type :=
| globenv_nil : consistent (snd φ) → on_global_decls φ []
| globenv_decl Σ d :
    on_global_decls φ Σ →
    fresh_global (global_decl_ident d) Σ →
    on_global_decl (Σ, ϕ) d →
    on_global_decls ϕ (d :: Σ).
```

The empty environment is wellformed when the graph of universes has no inconsistencies. Well-formedness of constants is the same as for local contexts. Well-formedness of inductive declarations is outlined below. For each new declaration, the identifier is required to be fresh with respect to the previous ones.

Inductive declarations. In Coq, a block of mutual inductive types is declared as follows:

```
Inductive I1 params : A1 := c11 : T11 | ... | c1n1 : T1n1
...
with      Ip params : Ap := cp1 : Tp1 | ... | cpnp : Tpnp.
```

`I1`, … `Ip` are the names of the inductive types. `A1`, … `Ap` are the arities. The `cij` are the constructors and the `Tij` their types. `params` is the context of parameters. This context can contain some let-bindings, we will write $x_1, \dots x_n$ for the variables without body bound in this context.

Remark 3 With respect to *indices*, parameters $x_1, \dots x_n$ have to be constant in all the conclusions of the types of constructors. However, they may vary in the types of arguments of constructors. A parameter is called *uniform* if it is constant through the whole inductive type, and *non uniform* otherwise.

In MetaCoq, a mutual block of inductive types is formally represented by a `mutual_inductive_body` which, itself, consists mainly in a list of `one_inductive_body`, one for each block.

```
(* Declaration of one inductive type *)
Record one_inductive_body := {
  ind_name  : ident;
  ind_type  : term;  (* closed arity: ∀ params, Ai *)
  ind_kelim : list sort_family;  (* allowed elimination sorts *)
  (* name, type, number of arguments for each constructor *)
  ind_ctors : list (ident * term * nat);
  (* name and type for each projection (if any) *)
  ind_projs : list (ident * term)
}.

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

A block `mutual_inductive_body` is well-formed when:

– the context of parameters is well-formed: `wf_local` Σ `ind_params`;
– `ind_npars` is the number of assumptions (*i.e.* without let-in) in `ind_params`;
– each `one_inductive_body` is well-formed.

And a declaration of type `one_inductive_body` is well-formed when:

– the arity `ind_type` is well-sorted in the empty context and starts with at least `ind_npars` foralls "∀" (skipping the lets and casts);

– for each triplet (id,T,n) of the list of constructors ind_ctors,
  – T is well-sorted under the context of arities:

$$I_1 : A'_1, \ldots I_n : A'_n \vdash T : s \qquad \text{where } A'_i \text{ is } \forall \texttt{params}, A_i;$$

  – T is of the shape $\forall \texttt{params args}, I_i\ x_1\ \ldots\ x_n\ t_1\ \ldots\ t_k$ where args are the real arguments of the constructor and I_i is the corresponding de Bruijn index;
– for each pair (id, T) of the list of projections ind_projs:
  – the inductive type has no index;
  – T is well-sorted in the context of parameters extended by the considered inductive type:

$$\texttt{params}, x : I_i\, x_1\ \ldots\ x_n \vdash T : s.$$

This specification of inductive types is not fully complete: for instance ind_kelim is not checked yet. The main missing feature is the positivity criterion.

Remark 4 In Coq internals, there are in fact two ways of representing a declaration: either as a "body" or as an "entry". The kernel takes entries as input, type-checks them and elaborates them into bodies. In MetaCoq, we provide both, as well as an erasing function mind_body_to_entry from bodies to entries for inductive types.

2.6 Universes

The treatment of universes in Coq is both a strong feature and something hard to understand. We hope that MetaCoq can shed some light on it.

Coq relies on a hierarchy of universes: Prop, Set, $\texttt{Type}_0$, $\texttt{Type}_1$, $\texttt{Type}_2$, ... The universe Set can be seen as a strict synonym of $\texttt{Type}_0$.

The hierarchy behaves as follows for typing:

$$\texttt{Prop} : \texttt{Type}_1$$
$$\texttt{Type}_0 : \texttt{Type}_1 : \texttt{Type}_2 \ldots$$

And as follows with respect to cumulativity:

$$\texttt{Prop} \subseteq \texttt{Type}_0 \subseteq \texttt{Type}_1 \subseteq \texttt{Type}_2 \ldots$$

In Coq, the user does not have to provide the universe level i of $\texttt{Type}_\texttt{i}$ but can instead use typical ambiguity and simply write Type. The Coq system has then the responsibility of instantiating the universe levels properly. For flexibility, the universe levels are not definitely determined at declaration time. Instead, a *universe variable* for the level is introduced and only the most general *constraints* on this variable are recorded. In technical cases, the user can enforce the universe variable with the notation Type@{l}.

For instance, the following definition

```
Definition T : Type@{l₁} := ∀ (A : Type@{l₂}), A → Set.
```

will generate the constraints Set < $\texttt{l}_1$ and $\texttt{l}_2$ < $\texttt{l}_1$ where $\texttt{l}_1$ and $\texttt{l}_2$ are universe variables. Here, the set of constraints is satisfiable: it can be instantiated with, for instance, ($\texttt{l}_1$:= 2, $\texttt{l}_2$:= 1).

The Coq system maintains a set of constraints and updates it each time a new universe variable is introduced. The Coq system also manipulates some *algebraic* universes which are of the form Type@{max($\texttt{l}_1$,$\texttt{l}_2$+1)}, as introduced in Herbelin and Spiwack

(2013). The level of these universes is uniquely determined by `l₁` and `l₂`. Thanks to the `Set` keyword, $\texttt{Type}_0$ is the only $\texttt{Type}_i$ that can be given explicitly by the user.

Formally, 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. Polymorphic levels are used when type checking a polymorphic declaration (constant or inductive).

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

A universe is called *non-algebraic* if it is a level (that is, of the form `[(l, false)]`), and algebraic otherwise.

A constraint is given by two levels and a `constraint_type`:

```
Inductive constraint_type := Lt | Le | Eq.
Definition univ_constraint := Level.t * constraint_type * Level.t.
```

The set of constraints (`constraints`) is implemented by sets as lists without duplicates coming from the Coq standard library. A valuation is an instance for all monomorphic and polymorphic levels in natural numbers. Monomorphic (global) levels are required to be positive so that we have `Prop : Type` for any instance.

```
Record valuation :=
  { valuation_mono : string → positive ;
    valuation_poly : nat → nat }.
```

We define the evaluation of valuation on monomorphic levels and then on universes.

```
Fixpoint val0 (v : valuation) (l : Level.t) : Z :=
  match l with
  | lProp ⇒ -1
  | lSet ⇒ 0
  | Level s ⇒ Zpos (v.(valuation_mono) s)
  | Var x ⇒ Z.of_nat (v.(valuation_poly) x)
  end.

Fixpoint val (v : valuation) (u : universe) (Hu : u ≠ []) : Z := ...
```

Satisfaction of constraints is defined as expected. Then, a set of constraints is said to be consistent if there exists a valuation satisfying the constraints:

```
Definition consistent ctrs := ∃ v, satisfies v ctrs.
```

Last, given a set of constraints, two universes are said equal when they are equal for all valuation satisfying the constraints (idem for $\leq$):

```
Definition eq_universe (φ : uGraph.t) u Hu u' Hu' :=
  ∀ v, satisfies v (snd φ) → val v u Hu = val v u' Hu'.
Definition leq_universe (φ : uGraph.t) u Hu u' Hu' :=
  ∀ v, satisfies v (snd φ) → val v u Hu ≤ val v u' Hu'.
```

The functions `eq_term` and `leq_term` used in conversion and cumulativity relations are defined as congruence on terms calling those two functions on sorts.

2.7 Toward Coq bootstrap

The reification of syntax is a first step toward the bootstrap of Coq. From this, one can reimplement some algorithm of the kernel such as type inference, type checking, the test of conversion/cumulativity and so on. On the other hand, the reification of semantics is then a first step toward the certification of such reimplementation. From here, we can dream of a proof assistant whose critical algorithms are certified.

As a preliminary stage, we implemented the three aforementioned algorithms:

```
(* typing_result is an error monad *)
check_conv: Fuel→ global_ctx → context → term → term → typing_result unit
infer     : Fuel→ global_ctx → context → term → typing_result term
check     : Fuel→ global_ctx → context → term → term → typing_result unit
```

Type checking is given by type inference followed by a conversion test. All the rules of type inference are straightforward except for cumulativity. The cumulativity test is implemented by comparing recursively head normal forms for a fast-path failure. We implemented weak-head reduction by mimicking Coq 's implementation, which is based on an abstract machine inspired by the KAM. Coq 's machine optionally implements a variant of lazy, memoizing evaluation (the `lazy` reduction strategy). That feature has not been implemented yet. A major difference with the OCaml implementation is that all of functions are required to be shown terminating in Coq. One possibility could be to prove the termination of type-checking separately but this requires 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 `makeOutOfFuel` a type error.

We also implemented, the satisfiability check of universe constraints. In Coq, the set of constraints is maintained as a weighted graph called the *universe graph*. The nodes are the introduced level variables, and the edges are given by the constraints. Each edge has a weight which corresponds to the minimal distance needed between the two nodes:

```
Definition edges_of_constraint (uc : univ_constraint) : list edge :=
  let '((l, ct),l') := uc in
  match ct with
  | Lt ⇒ [(l,-1,l')]
  | Le ⇒ [(l,0,l')]
  | Eq ⇒ [(l,0,l'); (l',0,l)]
  end.
```

We implemented some functions to manipulate the graph:

```
      init_graph : uGraph.t  (* contains only Prop and Set *)
            add_node : Level.t → uGraph.t → uGraph.t
     add_constraint : univ_constraint → uGraph.t → uGraph.t
```

And some functions to query the graph:

```
         check_leq_universe : uGraph.t → universe → universe → bool
          check_eq_universe : uGraph.t → universe → universe → bool
no_universe_inconsistency: uGraph.t → bool (* the graph has no negative cycle *)
```

For the moment they all rely on a naive implementation of the Bellman-Ford algorithm as presented in Cormen et al. (2009).

3 The Template-Coq Plugin

Along with the formal specification of Coq, the MetaCoq project also provides a plugin, called Template-Coq, which allows to move back and forth from concrete syntax (the syntax of Coq as entered by the user) to reified syntax (as defined in the previous section).

$$\begin{tikzcd}\text{concrete syntax} \arrow[r, bend left, "\textit{quote}"] & \text{reified syntax} \arrow[l, bend left, "\textit{unquote}"]\end{tikzcd}$$

The plugin can reflect all kernel Coq terms.

We start by presenting the basic commands provided by the plugin to quote and unquote (Section 3.1), and then we describe in Section 3.2 the reification of the main Coq vernacular commands which can be used to automatize the use of quoting and unquoting. This makes it possible in particular to write plugins directly in Coq by combining such commands.

3.1 Basic commands

Quoting and unquoting of terms. The command `Test Quote` reifies the syntax of a term and prints it. For instance,

```
Test Quote (fun x ⇒ x + 0).
```

outputs the following

```
(tLambda (nNamed "x")
   (tInd {| inductive_mind := "Coq.Init.Datatypes.nat"; inductive_ind := 0 |} [])
   (tApp (tConst "Coq.Init.Nat.add" [])
      [tRel 0; tConstruct {| inductive_mind := "Coq.Init.Datatypes.nat";
                             inductive_ind := 0 |} 0 []]))
```

The command `Quote Definition f := (fun x ⇒ x + 0)` records the reification of the term in the definition `f` to allow further manipulations.

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

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

where `mkInd na k : inductive` is the $\texttt{k}^{\text{th}}$ inductive of the mutual block of the name `na`.

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

For instance, the command `Quote Recursively Definition mult_syntax := mult` (the multiplication on natural numbers) will define `mult_syntax` of type `global_declarations * term`. This first component is the list of declarations needed to typecheck the term `mult`. Namely, the declaration of the inductive `nat` and of the constants `add` and `mult`. The second component is the reified syntax of the term, here it is only: `tConst "Coq.Init.Nat.mult" []`.

The command `Make Inductive` provides a way to declare an inductive type from its syntax. For instance, the following command defines a copy of ℕ:

```
Make Inductive (mind_body_to_entry
    {| ind_npars := 0;  ind_universes := [];
       ind_bodies := [{|
Inductive TemplateMonad : Type → Prop :=
(* Monadic operations *)
| tmReturn : ∀ {A}, A → TemplateMonad A
| tmBind : ∀ {A B}, TemplateMonad A → (A → TemplateMonad B)
                                       → TemplateMonad B

(* General commands *)
| tmPrint : ∀ {A}, A → TemplateMonad unit
| tmMsg   : string → 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 : qualid → TemplateMonad (option global_reference)
| tmCurrentModPath : unit → TemplateMonad string
| tmExistingInstance : qualid → TemplateMonad unit
| tmInferInstance : option reductionStrategy → ∀ A, TemplateMonad (option A)

(* Quoting and unquoting commands *)
| tmQuote : ∀ {A}, A  → TemplateMonad term
| tmQuoteRec : ∀ {A}, A  → TemplateMonad (global_declarations * term)
| tmQuoteInductive : qualid → TemplateMonad mutual_inductive_body
| tmQuoteUniverses : TemplateMonad uGraph.t
| tmQuoteConstant : qualid → bool → TemplateMonad constant_entry
| tmMkInductive : mutual_inductive_entry → TemplateMonad unit
| tmUnquote : term  → TemplateMonad {A : Type & A}
| tmUnquoteTyped : ∀ A, term → TemplateMonad A.
```

Fig. 2 The monad of commands

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

3.2 Reification of Coq Commands

Along with the reification of Coq terms, Template-Coq provides the reification of the main vernacular commands of Coq. This way, one can write plugins by combining such commands. To combine commands while taking into account that commands have side effects (notably by interacting with global environment), we use the "free" monadic setting to represent those operations. A similar approach was for instance used in Mtac (Ziliani et al., 2015).

The syntax of reified commands is defined by the inductive family `TemplateMonad` (Fig. 2). In this family, `TemplateMonad A` represents a program which will eventually output a term of type `A`. There are special constructors `tmReturn` and `tmBind` to provide (freely) the basic monadic operations. We use the monadic syntactic sugar `x ← t ;; u` for `tmBind t (fun x ⇒ u)` and `ret` for `tmReturn`.

The other operations of the monad can be classified in two categories:

– the traditional Coq operations (`tmDefinition` to declare a new definition, etc.)
– 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).

An overview of available commands is given in Table 1.

Vernacular command	Reified command with its arguments	Description
`Eval`	`tmEval red t`	Returns the evaluation of `t` following the evaluation strategy `red` (`cbv`, `cbn`, `hnf`, `all`, `lazy` or `unfold`)
`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` when all obligations are closed
`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
	`tmPrint t` `tmMsg msg`	Prints a term or a message
	`tmFail msg`	Fails with error message `msg`
	`tmQuote t`	Returns the syntax of `t` (of type `term`)
	`tmQuoteRec t`	Returns the syntax of `t` and of 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 Inductive`	`tmMkInductive d`	Declares the inductive denoted by the declaration `d`
	`tmUnquote tm`	Returns the dependent 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

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. It is 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, declaring a new inductive type or printing something. Typically, we run programs of type `TemplateMonad unit`.

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

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

The program below asks the user to provide an inhabitant of `nat` (here we provide `3 * 3`), 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[3] 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.
```

The basic commands of TEMPLATE-COQ described in 3.1 are implemented with such `TemplateProgram`. For instance:

```
Definition tmMkDefinition id (tm : term) : TemplateMonad unit
  := tmBind (tmUnquote tm)
            (fun t' ⇒ tmBind (tmEval all (my_projT2 t'))
            (fun t'' ⇒ tmBind (tmDefinition id t'')
            (fun _ ⇒ tmReturn tt))).
```

4 Writing COQ plugins in COQ

The reification of commands of COQ allows users to write COQ plugins directly inside COQ, without requiring another language like OCAML or 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 plugin for extending COQ via syntactic translation as advocated in (Boulier et al., 2017) and (iii) a plugin extracting COQ functions to weak-call-by-value λ-calculus.

4.1 A Toy Example: 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 that we have a syntax for lambda calculus:

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

[3] In COQ, a proof obligation is a goal which has to be solved to complete a definition. Obligations were introduced by Sozeau (2007) in the PROGRAM mode.

And that in some part of our development, we want to consider a variation of `tm` with a new constructor, *e.g.* a “let in” constructor. 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 annotating the lambda or adding one new constructor, then `tm’` is automatically changed accordingly. We provide other examples, *e.g.* with mutual inductives, in the file `test-suite/add_constructor.v`.

We will see that it is fairly easy to define this plugin using Template-Coq. The main function is `add_constructor` which takes 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.
```

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 `mutual_inductive_body` which is the declaration of a block of mutual inductive types and returns another `mutual_inductive_body`.

```
Definition add_ctor (mind : mutual_inductive_body) (ind₀ : inductive)
                    (idc : ident) (ctor : term) : mutual_inductive_body
  := 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 (`i`$_0$ is its number), the new constructor is added.

4.2 The Program Translations Plugin

The following plugin expects a syntactic translation as defined in Boulier et al. (2017). It makes it possible to manipulate translated terms and, ultimately, to justify some logical extensions of Coq by postulating safe axioms. It is implemented in the file `translations/translation_utils.v`.

Two examples of syntactic translations are presented here: the parametricity translation, and a "times bool" translation (which justifies the negation of functional extensionality). A few other examples are available in the directory `translations`.

In all generality, a translation is given by two functions $[\,\_\,]$ and $[\![\,\_\,]\!]$ from Coq terms to Coq terms such that they enjoy at least computational soundness and typing soundness:

$$\frac{M \equiv N}{[M] \equiv [N]} \qquad\qquad \frac{\Gamma \vdash M : A}{[\![\Gamma]\!] \vdash [M] : [\![A]\!]}$$

Given such a translation, the plugin provides four commands:

- `Translate` which computes the translation $[M]$ of a term M.
- `TranslateRec` which computes the translation of a term and of all constants on which it depends.
- `Implement`. This command computes the translation $[\![$`Ax`$]\!]$ of a type `Ax` and asks the user to inhabit $[\![$`Ax`$]\!]$ in proof mode. If the user succeeds (but not before), it declares an axiom of type `Ax`. If the program translation is sound (*cf.* Boulier et al. (2017)), it ensures that the axiom does not break consistency.
- `ImplementExisting` which is used to provide the translation of some terms by hand. It can be used to "implement" an existing axiom. It is also useful to experiment with translations only partially defined; for instance to provide the translation of a particular inductive type without defining the translation of all inductive types.

To work, the plugin needs a translation. It is given by the following record:

```
Class Translation :=
  { tsl_id  : ident → ident ;
    tsl_tm  : tsl_context → term → tsl_result term ;
    tsl_ty  : option (tsl_context → term → tsl_result term) ;
    tsl_ind : tsl_context → string → kername → mutual_inductive_body
              → tsl_result (tsl_table * list mutual_inductive_body) }.
```

This record is a `Class` so that, using type classes inference, when a translation is provided, it is automatically found by Coq.

- `tsl_ident` is how identifiers are translated. It will always be `(fun id ⇒ id ++ "ᵗ")` for us.

- `tsl_tm` is the main translation function implementing $[\,\_\,]$. It takes a `term` and returns a `term`. The translation context contains the global environment and the previously translated constants, see below. The result is in the `tsl_result` monad which is an error monad:

  ```
  Inductive tsl_error :=
  | NotEnoughFuel             | TranslationNotFound (id : ident)
  | TranslationNotHandeled    | TypingError (t : type_error).
  ```

  The returned term can be of any type. `tsl_tm` is used by the commands `Translate` and `TranslateRec`.
- `tsl_ty` is the function translating types $[\![\,\_\,]\!]$. This time, the returned term is expected to be a type. This function is used by the commands `Implement` and `ImplementExisting` which are not available when `tsl_ty` is not provided. This is the case for models which do not translate a type by a type (for instance: the standard model, the setoid model, . . .).
- Last, `tsl_ind` is the function translating inductive types. It returns:
  - an extended translation table with the translations of the inductive type and its constructor;
  - a list of inductive declarations which are used in the translation of the inductive type. Generally, an inductive is translated either by itself (in which case the list is empty), or by a new inductive whose constructors are the the translation of the original constructors (in which case the list is of length one).

  The second argument of `tsl_ind` is technical: it is the path to the module in which the new inductives will be declared.

Translation context. In the translation plugin, the constants (definitions, axioms, inductive types and constructors), are translated one by one. They are recorded in a *translation table* so that the constants are not retranslated each time they appear. This association table is implemented as the list of the translated constants together with their translation.

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

Thus, the `tConst` case in the `tsl_tm` functio is generally implemented by:

```
  | tConst s univs ⇒ lookup_tsl_table table (ConstRef s)
```

and similarly for `tInd` and `tConstruct`.

Some translations that we implemented need to access the global environment in which the considered term makes sense. That's why we define a translation context to be a global environment and a translation table:

```
Definition tsl_context := global_context * tsl_table.
```

4.2.1 Parametricity

Let's describe the use of the plugin for the parametricity translation. Its implementation can be found in `translations/param_original.v`.

The translation that we use here follows Reynolds'parametricity (Reynolds, 1983; Wadler, 1989). We follow the already known approaches of parametricity for dependent type theories (Bernardy et al., 2012; Keller and Lasson, 2012). We get an alternative

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

Fig. 3 Unary parametricity translation and soundness theorem, excerpt (from Bernardy et al. (2012))

implementation Lasson's plugin ParamCoq[4]. For the moment, only the unary case is implemented. The translation is reminded in Figure 3.

The two components of the translation $[\_]_0$ and $[\_]_1$ are implemented by two recursive functions `tsl_param₀` and `tsl_param₁`.

```
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 ⇒ lookup_tsl_table' E (ConstRef s)
| _ ⇒ ...
end.
```

In Figure 3, the translation is presented in a named setting. As a consequence, the introduction of new variables does not change references to existing ones and 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_param₀` of $[\_]_0$ is no longer the identity. The argument `n` of `tsl_param₀` represents the de Bruijn level from which the variables have to be duplicated. There is no need for such an argument in `tsl_param₁`, the implementation of $[\_]_1$, because in this function all variables are duplicated. The implemented cases include pattern matching. Fixed-points are still work in progress.

Given those two functions, we can already translate some terms. For example, the translation of the type of polymorphic identity functions can be obtained by:

```
Definition ID := ∀ A, A → A.
```

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

```
Run TemplateProgram (Translate emptyTC "ID").
```

`emptyTC` is the empty translation context. This defines `ID`^{t} to be:

```
fun f : ∀ A, A → A ⇒ ∀ A (Aᵗ : A → Type) (x : A), Aᵗ x → Aᵗ (f A x)
```

We have also implemented `tsl_mind_body` the translation of inductive types. For instance, the translation of the equality type `eq` produces the following inductive:

```
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 `eq`^{t} and $[\texttt{eq\_refl}]_1$ by `eq_refl`^{t}.

The translation of the declarations of a block of mutual inductive types are similar declarations, with the arities and the types of constructors translated accordingly.

All put together, the translation is declared by:

```
Instance param : Translation :=
  {| tsl_id := fun id ⇒ id ++ "ᵗ" ;
     tsl_tm := fun ΣE t ⇒ ret (tsl_param₁ (snd ΣE) t) ;
     tsl_ty := None ;
     tsl_ind := fun ΣE mp kn mind ⇒ ret (tsl_mind_body (snd ΣE) mp kn mind) |}.
```

For each constant c of type A, it is $[c]_1$ (of type $[A]_1$ $[c]_0$) which is recorded in the translation table. There is no implementation of `tsl_ty` because there is no meaningful function $[\![\_]\!]$ for this presentation of parametricity.

Example. With this translation, the only commands that can be used are `Translate` and `TranslateRec`. Here is an illustration of their use coming from the work of Lasson on the automatic proofs of ω-groupoid laws using parametricity Lasson (2014). We show that all functions which have type `∀ (A:Type) (x y:A). x = y → x = y` are identity functions. Let `IDp` be this type. First we compute the translation of `IDp` using `TranslateRec`.

```
Run TemplateProgram (table ← TranslateRec emptyTC "IDp" ;;
                     tmDefinition "table" table).
```

The second line defines `table` as the new translation context, so that we can reuse it later. Then we show that every parametric function of type `IDp` is pointwise equal to the identity by using the predicate `fun y ⇒ x = y`.

```
Lemma param_IDp (f : IDp) : IDpᵗ 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.
```

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

```
Definition myf: IDp := fun A x y p ⇒ eq_trans (eq_trans p (eq_sym p)) p.
Run TemplateProgram (TranslateRec table "myf").
```

We reuse here `table` in which the translation of equality has been recorded. It is then possible to deduce automatically that $p \centerdot p^{-1} \centerdot p = p$ for all p:

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

4.2.2 Times bool translation

We describe here the use of the plugin with the times bool translation. This translation is a model of Coq[5] which negates function extensionality. It will give an example of the use of the command `Implement`. This example can be found in `translations/times_bool_fun.v`.

The translation is defined as follows on variables and dependent products (see Boulier et al. (2017) for a more complete description):

$$[x]_f := x \qquad [\lambda\, x : A.\ M]_f := (\lambda\, x : [A]_f.\ [M]_f, \mathbf{true})$$
$$[M\ N]_f := \pi_1([M]_f)\ [N]_f \qquad [\forall\, x : A.\ B]_f := (\forall\, x : [A]_f.\ [B]_f) \times \mathbb{B}$$

For this translation, terms and types are translated the same way, hence $[\![\,\_\,]\!]_f = [\,\_\,]_f$.

Even if the translation is very simple, this time, going from the ideal world of calculus of constructionsto the real world of Coq is not as simple as for parametricity. Indeed, when written in Coq, the translation is no longer fully syntax directed. In Coq, pairs (M, N) are typed, M and N are not the only arguments, their types are also required:

```
pair : ∀ (A B : Type), A → B → A × B
```

Hence, in the case of lambdas in the definition of the translation, those types have to be provided:

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

`true` is always of type `bool`, but for the left hand side term, we cannot 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. There is a similar issue with applications and projections, but this one can be circumvented using primitive projections which are untyped.

A solution is to use the type inference algorithm of Section 2.7 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
```

Here we need to have kept track of the global context Σ and of the local context Γ.

The translation function $[\,\_\,]_f$ is thus implemented by:

```
Fixpoint tsl_rec (fuel : nat) (Σ : global_context) (E : tsl_table)
                 (Γ : context) (t : term) {struct fuel}
  : tsl_result term :=
  match fuel with
  | O ⇒ raise NotEnoughFuel
  | S fuel ⇒
  match t with
  | tRel  n ⇒ ret (tRel n)
  | tSort s ⇒ ret (tSort s)

  | tProd n A B ⇒ A’ ← tsl_rec fuel Σ E Γ A ;;
                  B’ ← tsl_rec fuel Σ E (Γ ,, vass n A) B ;;
                  ret (timesBool (tProd n A’ B’))
```

[5] In fact, this translation is not completely a model of Coq: Coq features η-conversion on functions, which is incompatible with this translation.

```
  | tLambda n A t ⇒ A’ ← tsl_rec fuel Σ E Γ A ;;
                    t’ ← tsl_rec fuel Σ E (Γ ,, vass n A) t ;;
                    match infer Σ (Γ ,, vass n A) t with
                    | Checked B ⇒
                      B’ ← tsl_rec fuel Σ E (Γ ,, vass n A) B ;;
                      ret (pairTrue (tProd n A’ B’) (tLambda n A’ t’))
                    | TypeError t ⇒ raise (TypingError t)
                    end

  ...
  end
  end.
```

We use a fuel argument because of the non-structural recursive call on `B` in the case of lambdas.

We also implemented the translation of some inductive types. For instance, the translation of the inductive `foo` generates the new inductive $\texttt{foo}^t$:

```
Inductive foo :=
| bar : (nat → foo) → foo.

Inductive fooᵗ :=
| barᵗ : (natᵗ → fooᵗ) × bool → fooᵗ
```

and the translation is extended by:

```
[ foo ] = fooᵗ
[ bar ] = (barᵗ ; true)
```

Example. Let’s demonstrate how to use the plugin to negate function extensionality.

The type of the axiom we will add to our theory is:

```
Definition NotFunext := (∀ A B (f g:A→B), (∀ x:A, f x = g x) → f = g) →
     False.
```

We use `TranslateRec` to get the translation of `eq` and `False` and then we use `Implement` to inhabit the translation of the `NotFunext`:

```
Run TemplateProgram (TC ← TranslateRec emptyTC NotFunext ;;
                     Implement TC "notFunext" NotFunext).
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_reflᵗ _ _; true).
  inversion H.
Defined.
```

The `Implement` command generates an obligation whose type is the translation of `NotFunext`, that is:

```
((∀ A, (∀ B, (∀ f : (A → B) × bool, (∀ g : (A → B) × bool,
  ((∀ x : A, eqᵗ B (π1 f x) (π1 g x)) × bool → eqᵗ ((A → B) × bool) f g)
   × bool) × bool) × bool) × bool) × bool → Falseᵗ) × bool
```

There are a lot of “`× bool`”, that’s why it is convenient that this type is automatically computed. We fill the obligation with the tactics `tIntro` and `tSpecialize` which are variants of `intro` and `specialize` dealing with the boolean:

```
Tactic Notation "tSpecialize" ident(H) uconstr(t)
  := apply π1 in H; specialize (H t).
Tactic Notation "tIntro" ident(H)
  := refine (fun H ⇒ _; true).
```

After the obligation is closed (and not before), an axiom `notFunext` of type `NotFunext` is declared in the current environment, as it would have been done by:

```
Axiom notFunext : NotFunext.
```

A constant `notFunext`^{t} whose body is the term provided in the obligation is also declared and the mapping (`notFunext`, `notFunext`^{t}) is added in the translation table.

If the translation is correct, the consistency of Coq is preserved by the addition of this axiom. Let's insist on the fact that it is not fully the case because Coq has η-conversion, which is incompatible with this translation.

4.3 Extraction to λ-calculus

As a last example, we show how Template-Coq can be used to extract Coq functions to the weak-call-by-value λ-calculus (Forster and Kunze, 2019). It is folklore that every function definable in constructive type theory is computable in the classical sense, i.e. in a model of computation. While this statement can not be proven as a theorem inside the type theory of Coq, similar to parametricity, it is possible to give a computability proof in Coq for every concrete defined function. The translation from Coq functions to terms of the λ-calculus is essentially the identity, since the syntax of Coq can be seen as a feature-rich, type-decorated λ-calculus. Special care only has to be taken for fixed-points and inductive types (we do not cover co-inductives).

As concrete target language we use the (weak) call-by-value λ-calculus as used by Forster and Smolka (2017). The syntax is defined using de Bruijn indices:

$$s, t, u, v \;:\; \texttt{lterm} \;::=\; n \mid s\ t \mid \lambda s \qquad (n : \texttt{nat})$$

We follow their approach in employing Scott's encoding (Mogensen, 1992; Jansen, 2013) to incorporate inductive types and a fixed-point combinator ρ for recursion.

For instance, the Scott encoding of booleans is defined as $\varepsilon_{\texttt{bool}}\,\texttt{true} = \lambda xy.x$ and $\varepsilon_{\texttt{bool}}\,\texttt{false} = \lambda xy.y$, or $\lambda\lambda 1$ and $\lambda\lambda 0$ using de Bruijn indices, which we will avoid for examples. For natural numbers, the encodings are $\varepsilon_{\texttt{nat}}\,0 = \lambda zs.z$ and $\varepsilon_{\texttt{nat}}\,(S\ n) = \lambda zs.s(\varepsilon_{\texttt{nat}}\,n)$. Note that Scott encodings allow very direct encodings of `match`es: The Coq term `fun n : nat ⇒ match n with 0 ⇒ ... | S n' ⇒ ... end` can be directly translated to $\lambda n.\ n\ (\dots)\ (\lambda n'.\ \dots)$. We provide a command `tmEncode` which generates the Scott encoding function for an inductive datatype automatically. We restrict the generation to simple inductive types of the form

```
Inductive T (X1 ... Xp : Type) : Type :=
  ... | constr_i_T : A1 → ... → An → T X1 ... Xp | ... .
```

where `Aj` for $1 \leq$ `j` $\leq$ `n` is either encodable or exactly `T X1 ... Xn`. For such a fully instantiated inductive type `B = T X1 ... Xp` with `n` constructors we define the encoding function $\varepsilon_{\texttt{B}}$ as follows:

```
fix f (b : B) :=
  match b with
  | constr_i_T (x1 : A1) ... (xn : An) ⇒  λy1...yp.yi (f1 x1 ) ... (fn xn)
  | ...
  end
```

where `fj` for $1 \leq$ `j` $\leq$ `n` is a recursive call `f` if `Aj = B`, or $\varepsilon_{\mathtt{Aj}}$ otherwise. To be able to obtain the encoding function $\varepsilon_{\mathtt{Aj}}$, we could use translation tables as before. Instead, we demonstrate an alternative way using a type class of encodable types defined as follows:

```
Class encodable (A : Type) := enc_f : A → lterm.
```

Then, to generate, for instance, the Scott encoding of the type `lterm` itself, one first has to generate the Scott encoding for natural numbers:

```
Run TemplateProgram (tmEncode "nat_enc" nat).
Run TemplateProgram (tmEncode "lterm_enc" lterm).
```

This will define `nat_enc : encodable nat` and `lterm_enc : encodable lterm`. The second command uses the `tmInferInstance` operation of the `TemplateMonad` to find the instance of `encodable nat` defined before. If no instance is found, an obligation of type `encodable nat` is opened.

To extract functions, we proceed similarly. We restrict the extraction to a simple polymorphic subset of Coq without dependent types. We call a type A admissible if A is of the form $\forall X_1 \ldots X_n :$ `Type`. $B_1 \to \cdots \to B_m$ with $B_m \neq$ `Type`. Terms $a : A$ are admissible if A is admissible and if all constants $c : C$ that are proper subterms of a are either (a) admissible and occur syntactically on the left hand side of an application fully instantiating the type-parameters of c with constants or (b) of type `Type` and occur syntactically on the right hand side of an application instantiating type parameters. For instance, the definition of the function `@map A B : list A → list B` is admissible:

```
Definition map (A B : Type) : (A → B) → list A → list B := fun f ⇒
  fix map := match l with | [] ⇒ @nil B | a :: t ⇒ @cons B (f a) (map l) end.
```

We again define a type class to look up previously extracted terms:

```
Class extracted {A : Type} (a : A) := int_ext : lterm.
```

For constants (and constructors) occurring as subterms the `tmInferInstance` operation is used again to obtain the respective extractions. We define commands `tmExtract` and `tmExtractConstr` which can be used to extract functions and constructors. To extract the full polymorphic `map` function, we use Coq's section mechanism:

```
Section Fix_X_Y.
  Context { X Y : Type }. Context { encY : encodable Y }.
  Run TemplateProgram (tmExtractConstr "nil_lterm" (@nil X)).
  Run TemplateProgram (tmExtractConstr "cons_lterm" (@cons X)).
  Run TemplateProgram (tmExtract "map_lterm" (@map X Y)).
End Fix_X_Y.
```

This will define `map_lterm : ∀ X Y {H : encodable Y}, extracted (@map X Y)` and register it as an instance of the type class `extracted`.

To prove that the extracted terms are indeed correct, we provide a logical relation $t_a \sim a$ read as t_a *computes* a and a set of `Ltac` tactics which will automatically establish this relation. We wrap extracted terms together with the relation into a type class `computable`. We use MetaCoq's ability to run monadic operations inside tactics to implement a tactic `extract` which uses `tmExtract` and the `Ltac` tactics to allow for automatic computability proofs. Since this is not directly related to MetaCoq, we omit the details here and refer to Forster and Kunze (2019).

To automatically verify terms, we again use `tmInferInstance` to obtain the correctness proofs for previously extracted constants or constructors. The correctness lemma for `fix` w.r.t weak call-by-value reduction $\succ$ can be stated in general as $\rho\, u\, v \succ^* u\, (\rho\, u)\, v$

for closed abstractions u, v. For `match`, the correctness lemmas depend on the type of the discriminee and we provide an operation `tmGenEncode` generating both the encoding function and the correctness lemma for the corresponding `match`.

For instance, in order to prove the computability of addition, a user has to generate the encoding of natural numbers and extract the successor function first:

```
Run TemplateProgram (tmGenEncode "nat_enc" nat).
Hint Resolve nat_enc_correct : Lrewrite.

Instance lterm_S : computable S.
Proof. extract constructor. Qed.

Instance lterm_add : computable add.
Proof. extract. Qed.
```

5 Running plugins natively in OCaml

The approach of writing Coq plugins in Coq, as illustrated above, has several advantages. First, functions written in Coq are amenable to verification, and second, plugins can be written and iterated on quickly within a Coq buffer. However, one major disadvantage is that, Coq programs can not leverage efficient representations, algorithms, and compilers available for other languages, which makes Coq programs comparatively slow. This is especially a problem for our plugins which process the raw syntax of terms (`Ast.term`) which can be very verbose.

To mitigate the performance problem, it is common practice to run verified Coq programs after extraction to OCaml. Extraction gives us access, to both the efficiency of native code, and provides a declarative way to replace inefficient Coq types with efficient, machine-optimized types and operations in OCaml. During extraction, the Coq type `term` is extracted to an OCaml datatype, say `coq_term_ext`, and programs operate on that representation. To interface these computations with the Coq internals, which is necessary for plugins, we implemented functions that convert Coq's kernel representation of terms, i.e. `constr`, to `coq_term_ext`. Just the translation in this one direction provides sufficient functionality to implement plugins such as the CertiCoq compiler which translates Coq terms into CompCert's Clight intermediate language. More sophisticated plugins, such as the parametricity plugin, need to use both reification and reflection in a dynamic way. This poses the challenge of providing and implementation of `TemplateMonad` in OCaml so that it can be run after extraction.

Unfortunately, the use of meta-language Coq terms to represent Coq terms in the template monad, as opposed to abstract syntax terms, makes extracting `TemplateMonad` programs impossible. For example, consider the type of `tmPrint`, `∀ A, A → TemplateMonad unit`. Under extraction, the value of type `A` will be extracted to an OCaml value of the extracted type corresponding to `A`. This does not match the intended semantics of the template monad, however, because we wish to print the Coq term.

To address this problem, we define an extractable variant of the `TemplateMonad` which we call `TM` for the purposes of this presentation. Rather than using the (inlined) type `{t:Type & t}` to represent Coq terms, it instead uses the `Ast.term` type. Figure 4 shows the constructors that changed between `TM` and `TemplateMonad`. In addition to the modified constructors, `TM` drops `tmQuote`, `tmQuoteRec`, `tmUnqote`, and `tmUnquoteTyped`, none of which make sense with the new representation of terms. The unquote commands

```
Inductive TM : Type → Type :=
| tmPrint : Ast.term → TM unit
| tmMsg   : string → TM unit

| tmEval (red : reductionStrategy) (tm : Ast.term) : TM Ast.term
| tmDefinition (nm : ident) (type : option Ast.term) (term : Ast.term) : TM
    kername
| tmAxiom (nm : ident) (type : Ast.term) : TM kername
| tmLemma (nm : ident) (type : Ast.term) : TM kername
| tmInferInstance (type : Ast.term) : TM (option Ast.term)
| ...
```

Fig. 4 Modified constructors in `TM` and `TemplateMonad`.

could, theoretically, be implemented by an interpreter, but the interface between this interpreter and the native OCaml program would not be sensible.

As a by-product of the phase separation, we solve an additional problem that is necessary. The `TemplateMonad` type lives in `Prop` in order to get impredicativity and avoid problems when manipulating terms of higher universes. Concretely, note that we can not existentially quantify over universe levels, which one might have to do if unquoting a term of type `TemplateMonad`. While we have not needed to unquote `TemplateMonad` values, in practice, we have experienced universe issues with an earlier version of `TemplateMonad` which was defined in `Type`.

Using the Phase Split Monad The phase split comes at the cost of some convenience. In the original `TemplateMonad`, we could write, `tmDefinition "one" 1`. In the phase split monad, we must construct the `term` representation of `1` explicitly. To ease this, we define a the `<% t %>` notation, inspired by MetaOCaml's `.< t >.`, which desugars to the quoted version of `t` using Coq's tactics-in-terms feature. Using this feature, we can adapt the simple declaration above as `tmDefinition "one" <% 1 %>`.

Things become slightly more complicated when the term to quote is built dynamically, for example: `fun x y : Ast.term ⇒ tmDefinition "add_them" <% x + y %>`. Currently, to achieve this, we must build the syntax directly: `tApp <% plus %> (x :: y :: nil)`. Proper multi-stage languages, such as MetaOCaml, address this through a splicing operator where the above could be written `.< .~x + .~y >.`. We leave implementing improved splicing to future work.

Limitations of the Phase Split Monad While the programs can be slightly more verbose, from a practical point of view, the phase split does not decrease the expressivity of the monad [6]. To see this, consider an arbitrary program that uses `tmQuote`. The value to quote is either a literal in the program (in which case we can simply use `<% %>`), comes from the caller (in which case we ask the caller to pass the quoted value instead), or is a hybrid of the two (see above).

In general, we found that, in most instances, adapating code simply required phase-splitting the top-level function. For example, a template program that might previously have taken an arbitrary value now takes a `term`, and the caller of the function performs the quoting on their side. Readers familiar with TemplateHaskell Sheard and Jones (2002b) will note that this style is also employed there.

[6] One exception is with `tmQuoteRec` which requires recursion which can not be proved structurally recursive to implement.

Performance Our largest use case for running plugins after extraction is lens[7] generation for Coq records. This plugin takes the fully qualified name of a record in the environment and defines a lens for each field of the record. A lens for a field of record can be used to project that field or update that field (while keeping the other fields constant). The plugin's implementation invokes the `tmQuoteInductive` to get the definition of the record, computes the body and the type of the lens for each field, and then defines each of those lenses by using the `tmMkDefinition` command. Although in our verification work, we typically have records of only a few fields, to very roughly estimate the execution-time savings in general, we tested the lens plugin both with and without extraction on a record with 30 fields. The execution time was respectively 0.774 second and 0.047 second: the extracted version ran at least 10 times faster. We observed more speedups on records with more fields.

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 (Devriese and Piessens, 2013) 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.

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 (Chapman, 2009; Altenkirch and Kaposi, 2016), 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 (Abel et al., 2018) 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 (Mullen et al., 2018), 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 (2011). 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 Boulier et al. (2017)), and both can be stated in Template-Coq. Anand has made substantial attempts in this direction to prove, in Template-Coq, the computational soundness of a variant of parametricity providing stronger theorems for free on propositions (Anand and Morrisett, 2018). This included as a first step a move to named syntax that could be reused in other

[7] this is inspired by lenses in Haskell: http://lens.github.io

translations. Our long term goal is to leverage the translation+plugin technique to extend the logical and computational power of Coq using, for instance, the forcing translation (Jaber et al., 2016) or the weaning translation (Pédrot and Tabareau, 2017).

The last direction of extension is to build higher-level tools on top of the syntax: the unification algorithm described in (Ziliani and Sozeau, 2017) 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 Bengtson (2016) which implemented this idea on a restricted fragment of CIC.

Acknowledgments

This work is supported by the CoqHoTT ERC Grant 64399 and the NSF grants CCF-1407794, CCF-1521602, and CCF-1646417.

References

Abel A, Öhman J, Vezzosi A (2018) Decidability of conversion for type theory in type theory. PACMPL 2(POPL):23:1–23:29, DOI 10.1145/3158111, URL http://doi.acm.org/10.1145/3158111

Altenkirch T, Kaposi A (2016) Type theory in type theory using quotient inductive types. ACM, New York, NY, USA, POPL '16, pp 18–29, DOI 10.1145/2837614.2837638, URL http://doi.acm.org/10.1145/2837614.2837638

Anand A, Morrisett G (2018) Revisiting Parametricity: Inductives and Uniformity of Propositions. In: CoqPL'18, Los Angeles, CA, USA

Anand A, Appel A, Morrisett G, Paraskevopoulou Z, Pollack R, Belanger OS, Sozeau M, Weaver M (2017) CertiCoq: A verified compiler for Coq. In: CoqPL, Paris, France, URL http://conf.researchr.org/event/CoqPL-2017/main-certicoq-a-verified-compiler-for-coq

Armand M, Grégoire B, Spiwack A, Théry L (2010) Extending Coq with Imperative Features and Its Application to SAT Verification. In: Kaufmann M, Paulson LC (eds) Interactive Theorem Proving, Springer, pp 83–98

Barras B (1999) Auto-validation d'un système de preuves avec familles inductives. Thèse de doctorat, Université Paris 7, URL http://pauillac.inria.fr/~barras/publi/these_barras.ps.gz

Bernardy JP, Jansson P, Paterson R (2012) Proofs for free: Parametricity for dependent types. Journal of Functional Programming 22(2):107–152

Boulier S, Pédrot PM, Tabareau N (2017) The next 700 syntactical models of type theory. In: CPP'17, Paris, France, ACM, pp 182–194

Chapman J (2009) Type Theory Should Eat Itself. Electronic Notes in Theoretical Computer Science 228:21 – 36, DOI https://doi.org/10.1016/j.entcs.2008.12.114, URL http://www.sciencedirect.com/science/article/pii/S157106610800577X, proceedings of LFMTP 2008

Chlipala A (2011) Certified Programming with Dependent Types. MIT Press

Cormen TH, Leiserson CE, Rivest RL, Stein C (2009) Introduction to algorithms. MIT press

Devriese D, Piessens F (2013) Typed syntactic meta-programming. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ACM, ICFP '13, URL http://doi.acm.org/10.1145/2500365.2500575

Ebner G, Ullrich S, Roesch J, Avigad J, de Moura L (2017) A Metaprogramming Framework for Formal Verification. In: Proceedings of the 22st ACM SIGPLAN Conference on Functional Programming (ICFP 2017), ACM Press, Oxford, UK, pp 34:1–34:29

Forster Y, Kunze F (2016) Verified Extraction from Coq to a Lambda-Calculus. In: Coq Workshop 2016, URL https://www.ps.uni-saarland.de/~forster/coq-workshop-16/abstract-coq-ws-16.pdf

Forster Y, Kunze F (2019) A certifying extraction with time bounds from coq to call-by-value λ-calculus. CoRR abs/1904.11818, URL http://arxiv.org/abs/1904.11818, 1904.11818

Forster Y, Smolka G (2017) Weak call-by-value lambda calculus as a model of computation in Coq. In: ITP 2017, Springer, pp 189–206

Herbelin H, Spiwack A (2013) The rooster and the syntactic bracket. In: Matthes R, Schubert A (eds) 19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, LIPIcs, vol 26, pp 169–187, DOI 10.4230/LIPIcs.TYPES.2013.169, URL https://doi.org/10.4230/LIPIcs.TYPES.2013.169

Jaber G, Lewertowski G, Pédrot PM, Sozeau M, Tabareau N (2016) The definitional side of the forcing. In: LICS'16, New York, NY, USA, pp 367–376, DOI 10.1145/2933575.2935320, URL http://doi.acm.org/10.1145/2933575.2935320

Jansen JM (2013) Programming in the λ-calculus: From Church to Scott and back. In: The Beauty of Functional Code, LNCS, vol 8106, Springer, pp 168–180

Keller C, Lasson M (2012) Parametricity in an impredicative sort. CoRR abs/1209.6336, URL http://arxiv.org/abs/1209.6336, 1209.6336

Lasson M (2014) Canonicity of Weak ω-groupoid Laws Using Parametricity Theory. In: Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXX), DOI 10.1016/j.entcs.2014.10.013

Malecha G, Bengtson J (2016) Extensible and efficient automation through reflective tactics. In: ESOP 2016, DOI 10.1007/978-3-662-49498-1_21, URL http://dx.doi.org/10.1007/978-3-662-49498-1_21

Malecha GM (2014) Extensible proof engineering in intensional type theory. PhD thesis, Harvard University, URL http://gmalecha.github.io/publication/2015/02/01/extensible-proof-engineering-in-intensional-type-theory.html

Mogensen TÆ (1992) Efficient self-interpretations in lambda calculus. J Funct Program 2(3):345–363

Mullen E, Pernsteiner S, Wilcox JR, Tatlock Z, Grossman D (2018) Œuf: minimizing the coq extraction TCB. In: Proceedings of CPP 2018, pp 172–185, DOI 10.1145/3167089, URL http://doi.acm.org/10.1145/3167089

Pédrot P, Tabareau N (2017) An effectful way to eliminate addiction to dependence. In: LICS'17, Reykjavik, Iceland, pp 1–12, DOI 10.1109/LICS.2017.8005113, URL https://doi.org/10.1109/LICS.2017.8005113

Reynolds JC (1983) Types, abstraction and parametric polymorphism. In: IFIP Congress, pp 513–523

Sheard T, Jones SP (2002a) Template meta-programming for haskell. SIGPLAN Not 37(12):60–75, DOI 10.1145/636517.636528, URL http://doi.acm.org/10.1145/636517.636528

Sheard T, Jones SP (2002b) Template meta-programming for haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, ACM, New York, NY, USA, Haskell '02, pp 1–16, DOI 10.1145/581690.581691, URL http://doi.acm.org/10.1145/581690.581691

Sozeau M (2007) Program-ing Finger Trees in Coq. ACM, New York, NY, USA, ICFP '07, pp 13–24, DOI 10.1145/1291151.1291156, URL http://doi.acm.org/10.1145/1291151.1291156

Taha W, Sheard T (1997) Multi-stage programming with explicit annotations. ACM, New York, NY, USA, PEPM '97, pp 203–217, DOI 10.1145/258993.259019, URL http://doi.acm.org/10.1145/258993.259019

Wadler P (1989) Theorems for free! In: Functional Programming Languages and Computer Architecture, ACM Press, pp 347–359

Van der Walt P, Swierstra W (2013) Engineering Proof by Reflection in Agda. In: Implementation and Application of Functional Languages, Springer

Ziliani B, Sozeau M (2017) A Comprehensible Guide to a New Unifier for CIC Including Universe Polymorphism and Overloading. Journal of Functional Programming 27:e10, DOI 10.1017/S0956796817000028, URL http://www.irif.univ-paris-diderot.fr/~sozeau/research/publications/drafts/unification-jfp.pdf

Ziliani B, Dreyer D, Krishnaswami NR, Nanevski A, Vafeiadis V (2015) Mtac: A Monad for Typed Tactic Programming in Coq. Journal of Functional Programming 25, DOI 10.1017/S0956796815000118, URL https://doi.org/10.1017/S0956796815000118