The MetaCoq formalization(s)

Yannick Forster **Meven Lennon-Bertrand** Matthieu Sozeau Théo Winterhalter

MetaCoq tutorial – January 14th 2024

This is only an overview

- Explore the library
- Read the paper(s) (*Correct and Complete Type Checking and Certified Erasure for Coq, in Coq*, currently only on HAL)
- Ask on the **dedicated stream on the Coq Zulip**

Template Coq

- the entry point for meta-programming
- very close to Coq's kernel (*e.g.* `constr.ml` for the datatype of terms)

Main datastructures:

- terms
- (local) context
- (global) environment
- representation of universes

Template Coq

Terms

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  …
```

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Variable (de Bruijn index)

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Named variable (for goals)

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Existential variable (with a substitution)

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Universe (**Prop**/**Set**/**Type**), more on this later

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Type casting (`t : A`) (`kind` tells Coq which conversion algorithm to use)

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Dependent product type (`forall x : A, B`) (`aname` stores the name + relevance)

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Function (`fun x : A ⇒ B`)

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Let binder (`let x := t : A in u`)

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

n-ary application, use the smart `mkApps : term → list term → term`

Terms (I)

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident)
  | tEvar (ev : nat) (args : list term)
  | tSort (s : sort)
  | tCast (t : term) (kind : cast_kind) (v : term)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tApp (f : term) (args : list term)
  | tConst (c : kername) (u : Instance.t)
  ...
```

Global constant (definition or axiom)

Terms (II)

```
...
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info : predicate term)
    (discr : term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
...
```

(Co)inductive type: `inductive` = a `kername` + an integer to identify the type, and `Instance.t` is for universe polymorphic inductive types

Terms (II)

```
...
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info : predicate term)
    (discr : term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
...
```

(Co)inductive constructor

Terms (II)

```
…
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info : predicate term)
    (discr : term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
…
```

Pattern-matching **`match`** `discr` **`as`** … **`in`** … **`return`** `P` **`with`** `branches` **`end`**

`ci` contains basic info (the inductive, its number of parameters…)

Terms (II)

```
...
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info : predicate term)
    (discr : term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
...
```

Projection `t.(proj)`

```
...
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info : predicate term)
    (discr : term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
...
```

(Mutual) (co)fixed point: `mfixpoint` is a list of mutually-defined functions (bodies + types), `idx` is the function focused in the list

Terms (III)

...

```
| tInt (i : PrimInt63.int)
| tFloat (f : PrimFloat.float)
| tArray (u : Level.t) (arr : list term) (default : term) (type : term).
```

Primitive datatypes.

Example

```
fun x : nat ⇒ x
```

```
tLambda
  {| binder_name := nNamed "x"; binder_relevance := Relevant |}
  (tInd
    {|
      inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
      inductive_ind := 0
    |}
    [])
  (tRel 0)
```

Example

```
fun x : nat ⇒ x
```

```
tLambda
  {| binder_name := nNamed "x"; binder_relevance := Relevant |}
  (tInd
    {|
      inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat");
      inductive_ind := 0
    |}
    [])
  (tRel 0)
```

`MetaCoq Quote` is your friend!

Operations on terms

(Parallel) substitution `subst : list term → nat → term → term`

`subst l i t` substitutes `l` for the variables `tRel i`, `tRel (S i)`… in `t`.

`t{j := u}` for unary substitution

(Parallel) substitution `subst : list term → nat → term → term`

`subst l i t` substitutes `l` for the variables `tRel i`, `tRel (S i)`... in `t`.

`t{j := u}` for unary substitution

Lifting `lift : nat → nat → term → term`

`lift n k t` lifts variable after the `k`-th by `n`

Typical usage: `lift0 n := (lift n 0)`, to move from `Γ` to `Γ,,,Δ` (if `#|Δ| = n`).

(Parallel) substitution `subst : list term → nat → term → term`

`subst l i t` substitutes `l` for the variables `tRel i`, `tRel (S i)`... in `t`.

`t{j := u}` for unary substitution

Lifting `lift : nat → nat → term → term`

`lift n k t` lifts variable after the `k`-th by `n`

Typical usage: `lift0 n := (lift n 0)`, to move from Γ to Γ,,,Δ (if #|Δ| = n).

Universe substitution

`subst_instance:forall {A : Type}, UnivSubst A → Instance.t → A → A`

Type-class, for all objects containing universe levels that can be substituted

Notation `t@[u]`

Template Coq

Contexts and environments

Usually written Γ, changes during type-checking.

```
Record context_decl : Type := mkdecl
  { decl_name : aname;  decl_body : option term;  decl_type : term }.

Definition context := list context_decl.
```

Beware: it can contain local definitions!

Local context

Usually written Γ, changes during type-checking.

```
Record context_decl : Type := mkdecl
  { decl_name : aname;  decl_body : option term;  decl_type : term }.

Definition context := list context_decl.
```

Beware: it can contain local definitions!

Written in snoc fashion:

```
Γ ,, vass na A        Γ ,, vdef na t A        Γ ,,, Δ
```

Global environment

Fixed during type-checking, extended at each new definition. Rougly:

```
Inductive global_decl :=
  | ConstantDecl : constant_body → global_decl (* Definition or Axiom *)
  | InductiveDecl : mutual_inductive_body → global_decl.
    (* (Co)inductive/record type *)

Record global_env :=
{ universes : ContextSet.t; (* Universe levels + constraints *)
  declarations : list (kername * global_decl)
  retroknowledge : Retroknowledge.t (* For primitive types *)}.
```

Global environment

Fixed during type-checking, extended at each new definition. Rougly:

```
Inductive global_decl :=
  | ConstantDecl : constant_body → global_decl (* Definition or Axiom *)
  | InductiveDecl : mutual_inductive_body → global_decl.
      (* (Co)inductive/record type *)

Record global_env :=
{ universes : ContextSet.t; (* Universe levels + constraints *)
  declarations : list (kername * global_decl)
  retroknowledge : Retroknowledge.t (* For primitive types *)}.
```

Definitions are done in a `global_env` + universes:

```
Definition global_env_ext : Type := global_env * universes_decl.
```

```
Record constant_body := {
  cst_type : term;
  cst_body : option term;
  cst_universes : universes_decl; (* For polymorphic universes *)
  cst_relevance : relevance (* For SProp *) }.
```

Declarations

```
Record constant_body := {
  cst_type : term;
  cst_body : option term;
  cst_universes : universes_decl; (* For polymorphic universes *)
  cst_relevance : relevance (* For SProp *) }.
```

I will spare you the inductive declarations `mutual_inductive_body`…

Template Coq

Universes

To have a vague idea of what happens under the hood: hopefully, you will not need to touch it too much.

```
Inductive Level : Set :=
| lzero (* represents Set *)
| level (s : string) (* global/monomorphic level *)
| lvar (n : N). (* local/polymorphic level *)
```

```
Inductive Level : Set :=
| lzero (* represents Set *)
| level (s : string) (* global/monomorphic level *)
| lvar (n : N). (* local/polymorphic level *)

Definition LevelExpr := Level × nat.
...
Inductive sort :=
  sProp | sSProp | sType nonEmptyLevelExprSet.
```

In reality, some indirections. Using the `MSet` library for efficient implementations of sets.

Levels and sorts

```
Inductive Level : Set :=
| lzero (* represents Set *)
| level (s : string) (* global/monomorphic level *)
| lvar (n : N). (* local/polymorphic level *)

Definition LevelExpr := Level × nat.
...
Inductive sort :=
  sProp | sSProp | sType nonEmptyLevelExprSet.
```

In reality, some indirections. Using the `MSet` library for efficient implementations of sets.

Example: **`Prop`** is represented as `tSort sProp`, **`Set`** as (roughly) `tSort (sType (singleton (lzero,0)))`...

Constraint sets

```
Inductive ConstraintType : Set := Le (z : Z) | Eq.
Definition UnivConstraint : Set := Level * ConstraintType * Level.
```

Constraint sets

```
Inductive ConstraintType : Set := Le (z : Z) | Eq.
Definition UnivConstraint : Set := Level * ConstraintType * Level.
```

Mostly used through

```
leq_sort : UnivConstraintSet.t → sort → sort → Prop
```

and similar accessors (`eq_sort`, `leq_universe`...).

Constraint sets

```
Inductive ConstraintType : Set := Le (z : Z) | Eq.
Definition UnivConstraint : Set := Level * ConstraintType * Level.
```

Mostly used through

```
leq_sort : UnivConstraintSet.t → sort → sort → Prop
```

and similar accessors (`eq_sort`, `leq_universe`...).

```
Coercion global_ext_constraints : global_env_ext ↣ ConstraintSet.t.
```

Polymorphic Universes

```
Inductive universes_decl : Type :=
Monomorphic_ctx : universes_decl
| Polymorphic_ctx : list name × ConstraintSet.t → universes_decl.

Definition Instance : Set := list Level.
```

Reduction, cumulativity, typing

Reduction, cumulativity, typing

Reduction

Reduction (I)

```
Inductive red1 (Σ : global_env) (Γ : context) : term → term → Type :=
| red_beta na t b a l :
  red1 Σ Γ (tApp (tLambda na t b) (a :: l)) (mkApps (subst10 a b) l)

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

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

| 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 u body)
...
```

Reduction (II)

```
…
| 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) (mkApps fn args)

| 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)

| 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))
…
```

Reduction (III)

```
…
| red_iota ci mdecl idecl cdecl c u args p brs br :
    nth_error brs c = Some br →
    …
    red1 Σ Γ (tCase ci p (mkApps (tConstruct ci.(ci_ind) c u) args) brs)
        (iota_red ci.(ci_npar) args bctx br)

| red_proj p u args arg:
    nth_error args (p.(proj_npars) + p.(proj_arg)) = Some arg →
    red1 Σ Γ (tProj p (mkApps (tConstruct p.(proj_ind) 0 u) args)) arg
…
```

Reduction (III)

```
…
| red_iota ci mdecl idecl cdecl c u args p brs br :
    nth_error brs c = Some br →
    …
    red1 Σ Γ (tCase ci p (mkApps (tConstruct ci.(ci_ind) c u) args) brs)
        (iota_red ci.(ci_npar) args bctx br)

| red_proj p u args arg:
    nth_error args (p.(proj_npars) + p.(proj_arg)) = Some arg →
    red1 Σ Γ (tProj p (mkApps (tConstruct p.(proj_ind) 0 u) args)) arg
…

| app_red_l t t' u : red1 Σ Γ t t' → red1 Σ Γ (tApp t u) (mkApps t' u)
```

Reduction, cumulativity, typing

Cumulativity

Cumulativity

Least

- congruence and equivalence relation
- containing **red1**
- cumulative:

```
| cumul_Sort : forall s s',
  leq_sort Σ s s' →
  Σ ;;; Γ ⊢ tSort s ≤ tSort s'
```

Least

- congruence and equivalence relation
- containing `red1`
- cumulative:

```
| cumul_Sort : forall s s',
  leq_sort Σ s s' →
  Σ ;;; Γ ⊢ tSort s ≤ tSort s'
```

In practice, two presentations:

- as an inductive relation, in one go (“specification”)
- via reduction to terms equal up to universes (“algorithmic”)

Reduction, cumulativity, typing

Typing

Typing (I)

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

  | type_Sort s :
      wf_local Σ Γ →
      wf_sort Σ s →
      Σ ;;; Γ ⊢ tSort s : tSort (Sort.super s)

  …

  | type_Lambda na A t s B :
      Σ ;;; Γ ⊢ A : tSort s →
      Σ ;;; Γ ,, vass na A ⊢ t : B →
      Σ ;;; Γ ⊢ tLambda na A t : tProd na A B
```

Typing (II)

```
| type_LetIn na A t s u B :
    Σ ;;; Γ ⊢ A : tSort s →
    Σ ;;; Γ ⊢ t : A →
    Σ ;;; Γ ,, vdef na t A ⊢ u : B →
    Σ ;;; Γ ⊢ tLetIn na t A u : tLetIn na t A B

| type_App t l t_ty t' :
    Σ ;;; Γ ⊢ t : t_ty →
    isApp t = false → l <> [] → (* Well-formed application *)
    typing_spine Σ Γ t_ty l t' →
    Σ ;;; Γ ⊢ tApp t l : t'
(*
| type_App : forall t na A B u,
  Σ ;;; Γ ⊢ t : tProd na A B →
  Σ ;;; Γ ⊢ u : A →
  Σ ;;; Γ ⊢ tApp t u : B{0 := u} *)
```

Typing (III)

```
| type_Const cst u :
    wf_local Σ Γ →
    forall decl (isdecl : declared_constant Σ.1 cst decl),
    consistent_instance_ext Σ decl.(cst_universes) u →
    Σ ;;; Γ ⊢ (tConst cst u) : (decl.(cst_type))@[u]
...
| type_Conv t A B s :
    Σ ;;; Γ ⊢ t : A →
    Σ ;;; Γ ⊢ B : tSort s →
    Σ ;;; Γ ⊢ A ≤ B → Σ ;;; Γ ⊢ t : B
```

Typing (IV)

```
| type_Case (ci : case_info) p c brs indices ps mdecl idecl :
  let predctx := case_predicate_context ci.(ci_ind) mdecl idecl p in
  let ptm := it_mkLambda_or_LetIn predctx p.(preturn) in
  declared_inductive Σ ci.(ci_ind) mdecl idecl →
  case_side_conditions Σ Γ ci p ps mdecl idecl indices predctx →
  Σ ;;; Γ ++ predctx ⊢ p.(preturn) : tSort ps →
  Σ ;;; Γ ⊢ c : mkApps (tInd ci.(ci_ind) p.(puinst)) (p.(pparams) ++ indices) →
  case_branch_typing Σ Γ ci p ps mdecl idecl ptm brs →
  Σ ;;; Γ ⊢ tCase ci p c brs : mkApps ptm (indices ++ [c])

| type_Proj p c u mdecl idecl cdecl pdecl args :
    declared_projection Σ p mdecl idecl cdecl pdecl →
    #|args| = ind_npars mdecl →
    Σ ;;; Γ ⊢ c : mkApps (tInd p.(proj_ind) u) args →
    Σ ;;; Γ ⊢ tProj p c : subst0 (c :: List.rev args) (pdecl.(proj_type))@[u]
```

```
| type_Fix mfix n decl :
    fix_guard Σ Γ mfix →
    nth_error mfix n = Some decl →
    wf_local Σ Γ →
    All (on_def_type (lift_typing1 (typing Σ)) Γ) mfix →
    All (on_def_body (lift_typing1 (typing Σ)) (fix_context mfix) Γ) mfix →
    wf_fixpoint Σ mfix →
      Σ ;;; Γ ⊢ tFix mfix n : decl.(dtype)
...
| type_Int p prim_ty cdecl :
    wf_local Σ Γ →
    primitive_constant Σ primInt = Some prim_ty →
    declared_constant Σ prim_ty cdecl →
    primitive_invariants primInt cdecl →
    Σ ;;; Γ ⊢ tInt p : tConst prim_ty []
...
```

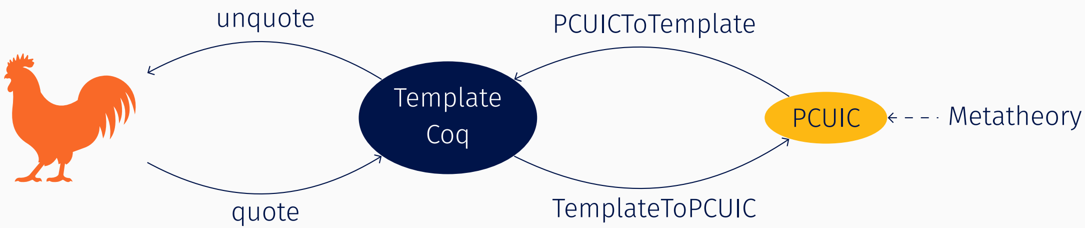
Environment typing

- names should be unique
- everything should be well-typed
- extra conditions on inductive types:
    - positivity (no general recursive types!)
    - cumulative universes
    - allowed elimination

Mechanized Meta-theory

PCUIC

A slightly simplified version of Template, better suited for meta-theory



Meta-theory

Substitution

- substitution calculus
- universe and term substitution for cumulativity, typing, etc.

Confluence & Simulation

$$\begin{tikzcd}[column sep=small]
 & t \arrow[dl, squiggly, "\star"'] \arrow[dr, squiggly, "\star"] & \\
t_1 \arrow[dr, squiggly, dashed] & & t_2 \arrow[dl, squiggly, dashed] \\
 & u \arrow[u, phantom, "\star\star"] &
\end{tikzcd}$$

$$\begin{tikzcd}
t \arrow[d, squiggly] & \texttt{leq_term} & u \arrow[d, squiggly, dashed] \\
t' & \texttt{leq_term} & u'
\end{tikzcd}$$

Subject reduction

```
Theorem subject_reduction Σ Γ t u T :
  wf Σ → Σ ;;; Γ ⊢ t : T → Σ ;;; Γ ⊢ t ⇝ u → Σ ;;; Γ ⊢ u : T.
```

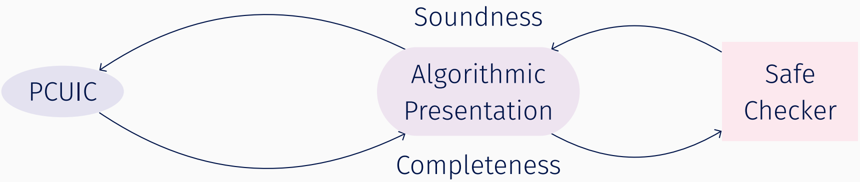
Axiomatized!

Axiomatized!

```
Class GuardCheckerCorrect :=
{
  guard_red1 b Σ Γ mfix mfix' idx :
    Σ ;;; Γ ⊢ tFixCoFix b mfix idx ⇝
      tFixCoFix b mfix' idx →
    guard b Σ Γ mfix → guard b Σ Γ mfix' ;
  …
}.
Axiom guard_checking_correct : GuardCheckerCorrect.

Axiom Normalization : forall Σ Γ t,
  wf_ext Σ → welltyped Σ Γ t → Acc (cored Σ Γ) t.
```

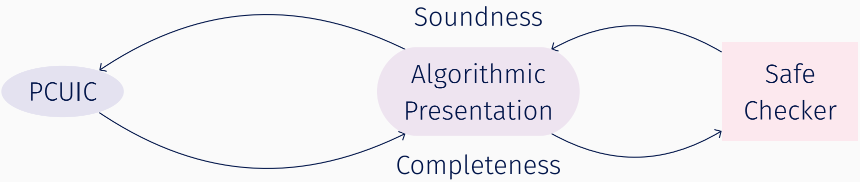
Algorithmic judgments & the safe-checker



Algorithmic:

- cumulativity = based on reduction and `leq_term`
- typing = bidirectional

Algorithmic judgments & the safe-checker



Algorithmic:

- cumulativity = based on reduction and `leq_term`
- typing = bidirectional

```
Equations check Σ Γ t A : ∥ wf_ext Σ ∥ → ∥ wf_local Σ Γ ∥ →
  : typing_result_comp (∥ Σ;;; Γ ⊢ t : A ∥) := …
```

$\lambda\square$

```
Inductive E.term : Set :=
  | tBox (* Represents all proofs *)
  | tRel (n : nat)
  ...
```

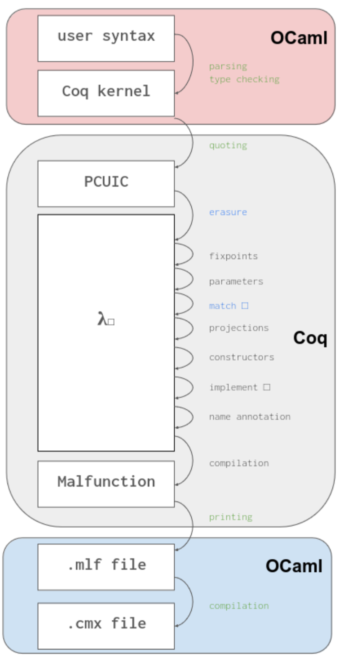
$\lambda\Box$

```
Inductive E.term : Set :=
  | tBox (* Represents all proofs *)
  | tRel (n : nat)
  …
```

Erasure: removes types and proofs

```
Equations? erase Σ Γ t (Ht : welltyped Σ Γ t) : E.term :=
  { erase Γ t Ht with inspect_bool (is_erasableb Σ Γ t Ht) :=
  { | left he := E.tBox;
    | right he with t := {
      | tRel i := E.tRel i
      …
```

Er

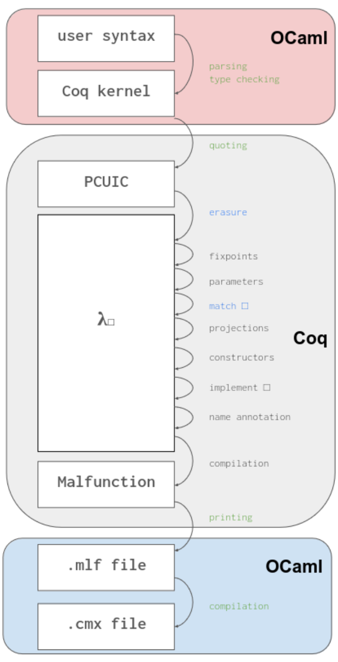


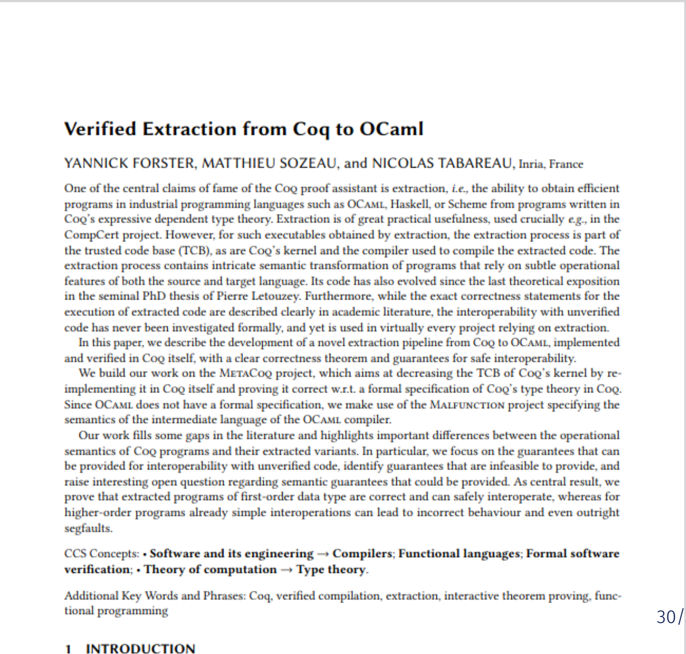

```
=
ll proofs *)
```

d proofs

```
Ht : welltyped Σ Γ t) : E.term :=
spect_bool (is_erasableb Σ Γ t Ht) :=

 {
 i
```





Verified Extraction from Coq to OCaml

YANNICK FORSTER, MATTHIEU SOZEAU, and NICOLAS TABAREAU, Inria, France

One of the central claims of fame of the Coq proof assistant is extraction, *i.e.*, the ability to obtain efficient programs in industrial programming languages such as OCaml, Haskell, or Scheme from programs written in Coq's expressive dependent type theory. Extraction is of great practical usefulness, used crucially *e.g.*, in the CompCert project. However, for such executables obtained by extraction, the extraction process is part of the trusted code base (TCB), as are Coq's kernel and the compiler used to compile the extracted code. The extraction process contains intricate semantic transformation of programs that rely on subtle operational features of both the source and target language. Its code has also evolved since the last theoretical exposition in the seminal PhD thesis of Pierre Letouzey. Furthermore, while the exact correctness statements for the execution of extracted code are described clearly in academic literature, the interoperability with unverified code has never been investigated formally, and yet is used in virtually every project relying on extraction.

In this paper, we describe the development of a novel extraction pipeline from Coq to OCaml, implemented and verified in Coq itself, with a clear correctness theorem and guarantees for safe interoperability.

We build our work on the MetaCoq project, which aims at decreasing the TCB of Coq's kernel by re-implementing it in Coq itself and proving it correct w.r.t. a formal specification of Coq's type theory in Coq. Since OCaml does not have a formal specification, we make use of the Malfunction project specifying the semantics of the intermediate language of the OCaml compiler.

Our work fills some gaps in the literature and highlights important differences between the operational semantics of Coq programs and their extracted variants. In particular, we focus on the guarantees that can be provided for interoperability with unverified code, identify guarantees that are infeasible to provide, and raise interesting open question regarding semantic guarantees that could be provided. As central result, we prove that extracted programs of first-order data type are correct and can safely interoperate, whereas for higher-order programs already simple interoperations can lead to incorrect behaviour and even outright segfaults.

CCS Concepts: • **Software and its engineering → Compilers**; **Functional languages**; **Formal software verification**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: Coq, verified compilation, extraction, interactive theorem proving, functional programming

1 INTRODUCTION

There's more to come!

Actively worked on:

- Guard and normalisation
- Support for η rules
- Sort polymorphism
- Modules
- …