Verified Extraction and Type-Checking for Coq in Coq

20-01-2025
Dick Arends
MFoCS Seminar

PAPERS

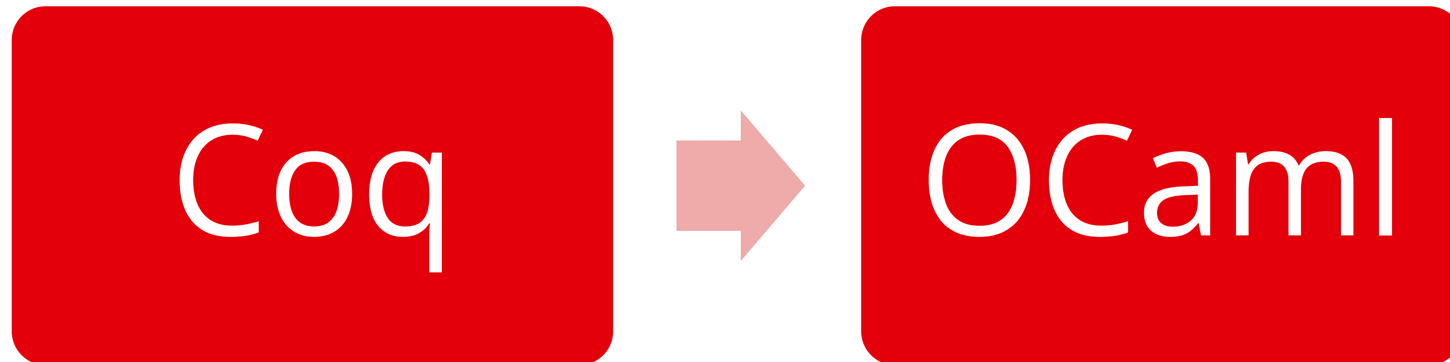
Verified Extraction from Coq to OCaml

Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. Proc. ACM Program. Lang. 8, PLDI, Article 149 (June 2024), 24 pages. https://doi.org/10.1145/3656379

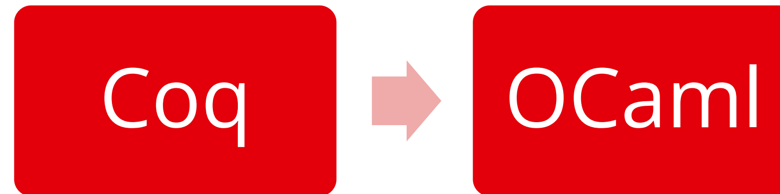
Coq Coq correct! Verification of type checking and erasure for Coq, in Coq

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. Proc. ACM Program. Lang. 4, POPL, Article 8 (January 2020), 28 pages. https://doi.org/10.1145/3371076

EXTRACTION

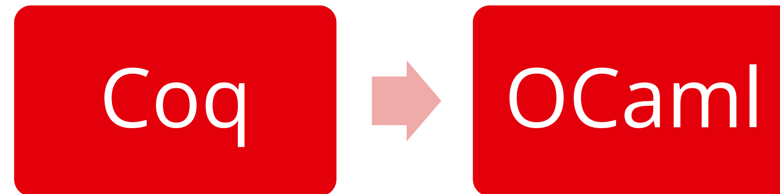


EXTRACTION



Coq	OCaml
Enjoy the expressiveness of Coq's type system	Interoperate with existing code
Dependent types	No dependent types
Purely functional	Can be effectful
Proof objects	No proof objects

SIMPLE EXTRACTION EXAMPLE




```coq
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.

Definition simple_coq_function : nat -> nat := S.
```

```ocaml
type nat = O | S of nat

(** val simple_coq_function : nat -> nat **)
let simple_coq_function x = S x
```

OCAML'S TYPE SYSTEM IS TOO WEAK

EXTRACTION EXAMPLE

```
Definition pred_or_false : forall n : nat,
  if n == 0 then bool else nat :=
    fun n : nat => match n with
      | O => false
      | S n => n
    end.
```

```
(** val pred_or_false : nat -> Obj.t **)

let pred_or_false = function
| O    -> Obj.magic False
| S n0 -> Obj.magic n0
```

EXTRACTION EXAMPLE

```
Definition pred_or_false : forall n : nat,
  if n == 0 then bool else nat :=
    fun n : nat => match n with
      | O => false
      | S n => n
    end.
```

```
(** val pred_or_false : nat -> Obj.t **)

let pred_or_false = function
| O    -> Obj.magic False
| S n0 -> Obj.magic n0
```

Module Obj

```
module Obj: sig .. end
```

Operations on internal representations of values.

Not for the casual user.

EXTRACTION EXAMPLE

```
Definition id_or_zero : forall b : bool, if b then bool -> bool else nat :=
  fun b : bool =>
  match b with
  | true => fun x => x
  | false => 0
  end.

Definition apply_or_false : forall b : bool, (if b then bool -> bool else nat) -> bool :=
  fun b : bool =>
  match b with
  | true => fun f => f true
  | false => fun _ => false
  end.

Definition assumes_pure : (unit -> bool) -> bool :=
  fun f => apply_or_false (f tt) (id_or_zero (f tt)).
```

SEGFAULTS

EXTRACTION EXAMPLE

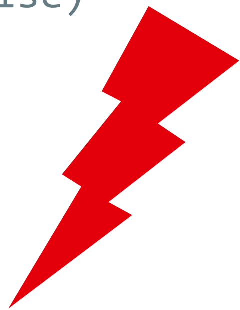
```
Definition assumes_pure : (unit -> bool) -> bool :=
fun f =>
  apply_or_false (f tt) (id_or_zero (f tt)).
```

```
(** val assumes_pure : (unit -> bool) -> bool **)
let assumes_pure f =
apply_or_false (f tt) (id_or_zero (f tt))
```

```
let x : bool ref = ref False;;

let impure : unit -> bool = fun _ -> match !x with
| False -> (x:=True; False)
| True  -> True;;

assumes_pure impure;;
```



CURRENT EXTRACTION

Due to Letouzey

Proves:

Higher order functions, for example $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, behave correctly when they are called on functions that extensionally agree to a Coq function of the same type

FOCUS POINTS OF THE PAPER

1. “Due to a lack of formal semantics for OCaml, the extraction process is not and cannot be formally specified and thus not proved correct.”

💡 Extract to (untyped) Malfunction instead of OCaml. Provide formal semantics for Malfunction.

2. “Even though a type interface is provided by extraction, the use of `Obj.magic` means that there are no guarantees by OCaml’s type checker.”

💡 Extract to Malfunction.

3. “Interoperation of higher-order functions with (potentially) nonterminating or effectful OCaml programs is not captured by the correctness guarantees, because they do not extensionally agree with a Coq program.”

💡 Everything is fine, if we work in the realms where the two main central correctness theorems hold.

IN WORDS

CENTRAL CORRECTNESS THEOREMS

1. If a Coq program *t* of first-order data type is convertible to some *v* and *v* is irreducible, then its compilation to Malfunction evaluates to the value translation of *v*.

2. A Malfunction program obtained using extraction of a first-order function can be applied to *any* piece of OCaml code with the correct type.
I.e. an example like `assumes_pure` cannot be produced for first-order functions.

HYPOTHESES #1

FIRST CORRECTNESS THEOREM

```
Variable Σ : global_env_ext.
Variable (ΣH : wf_ext Σ).
Variable (Σexp : expanded_global_env Σ).
Variable t : term.
Variable expt : expanded Σ [] t.
Variables (i : inductive) (u : Instance.t) (args : list PCUICAst.term).
Variable fo : firstorder_ind Σ (firstorder_env Σ) i.
Variable typing : Σ ; [] ⊢ t : mkApps (tInd i u) args.
Variable noParam : ∀ i mdecl, lookup_env Σ i = Some (InductiveDecl mdecl) → ind_npars
mdecl = 0.
```

HYPOTHESES #1
FIRST CORRECTNESS THEOREM

```
Variable Σ : global_env_ext.
Variable (ΣH : wf_ext Σ).
Variable (Σexp : expanded_global_env Σ).
Variable t : term.
Variable expt : expanded Σ [] t.
Variables (i : inductive) (u : Instance.t) (args : list PCUICAst.term).
Variable fo : firstorder_ind Σ (firstorder_env Σ) i.
Variable typing : Σ ; [] ⊢ t : mkApps (tInd i u) args.
Variable noParam : ∀ i mdecl, lookup_env Σ i = Some (InductiveDecl mdecl) → ind_npars
mdecl = 0.
```

HYPOTHESES #2

FIRST CORRECTNESS THEOREM

```
Variable v : term.
Variable v_red : Σ ; [] ⊢ t ⇝* v.
Variable v_irred : ∀ t', ( Σ ; [] ⊢ v ⇝ t' ) → False.
```

STATEMENT

FIRST CORRECTNESS THEOREM

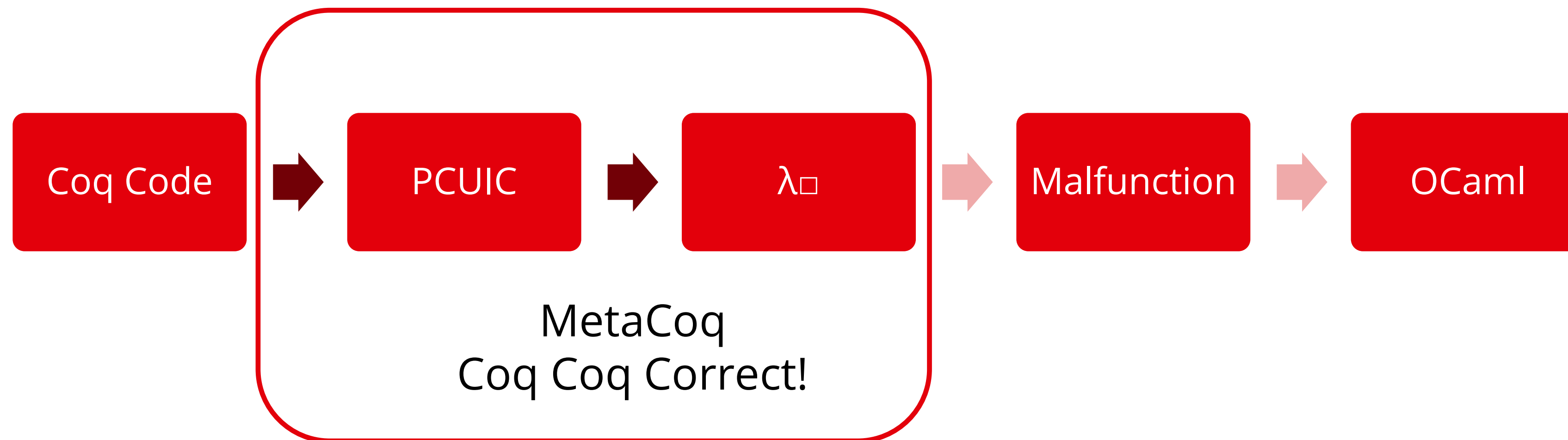
```
Theorem verified_malfunction_pipeline_theorem : ∀ (h:heap),
  eval Σ' empty_locals h (compile_malfunction_pipeline expt typing)
  h (compile_value_mf Σ v).
```

ALL DONE WITHIN COQ

- Note that both theorems are stated within Coq itself.
- The proof is also mechanized within Coq.

A VERIFIED EXTRACTION PIPELINE

THE VERIFIED EXTRACTION PIPELINE



PCUIC

- Equivalent to the calculus implemented in Coq.
- (Meta-theoretical) properties have been formally verified about PCUIC *within Coq.*

DEFINITION

PCUIC

PCUIC

```
Inductive term :=
| tRel (n : nat)
| tVar (i : ident)
| tEvar (n : nat) (l : list term)
| tSort (u : sort)
| tProd (na : aname) (A B : term)
| tLambda (na : aname) (A t : term)
| tLetIn (na : aname) (b B t : term)
| tApp (u v : term)
| tConst (k : kername) (ui : Instance.t)
| tInd (ind : inductive) (ui : Instance.t)
| tConstruct (ind : inductive) (n : nat) (ui : Instance.t)
| tCase (indn : case_info) (p : predicate term) (c : term) (brs : list (branch term))
| tProj (p : projection) (c : term)
...
```

QUOTATION
EXAMPLE

```
true : bool
tConstruct Ind_bool 0 []

fun p : Prop => p
tLambda name_p (tSort sProp) (tRel 0)

fun x : nat => x
tLambda name_x (tInd Ind_nat []) (tRel 0)
```

PCUIC

```
Inductive term :=
| tRel (n : nat)
| tVar (i : ident)
| tEvar (n : nat) (l : list term)
| tSort (u : sort)
| tProd (na : aname) (A B : term)
| tLambda (na : aname) (A t : term)
| tLetIn (na : aname) (b B t : term)
| tApp (u v : term)
| tConst (k : kername) (ui : Instance.t)
| tInd (ind : inductive) (ui : Instance.t)
| tConstruct (ind : inductive) (n : nat) (ui : Instance.t)
| tCase (indn : case_info) (p : predicate term) (c : term)
    (brs : list (branch term))
| tProj (p : projection) (c : term)
...
```

TYPING

```
typing : global_env_ext → context → term → term → Type
```

`Σ ; Γ ⊢ t : T` for `typing Σ Γ t T`

Example for application of `t u`:

```
type_App t na A B u :
  Σ ; Γ ⊢ t : tProd na A B →
  Σ ; Γ ⊢ u : A →
  Σ ; Γ ⊢ tApp t u : B{0 := u}
```

TYPE CHECKING

Define a type inference algorithm *within Coq* that for a given term `t` returns the type `T` such that `t : T` and a proof stating `Σ ; Γ ⊢ t : T`

Type checker performance is around 1 order of magnitude slower than the existing one.

But this implementation is verified according to a specification.

META THEORETICAL PROPERTIES OF PCUIC

Authors assume theory/specification for Coq to be correct.
From there prove that the implementation is correct.

One of the properties assumed to hold is strong normalization.

Idea: Trust the theory not the implementation.

FORMALLY VERIFIED META THEORETICAL PROPERTIES

- Confluence of reductions
  $\forall\, t, u, v$ we have $t \rightarrow^* u \land t \rightarrow^* v \Rightarrow \exists x$ s.t. $u \rightarrow^* x \land v \rightarrow^* x$

- Validity of typing
  If $\Gamma \vdash t : T$ then we have that T is well-typed.

- Subject reduction
  If $e_1 : T$ and e_1 reduces to some e_2 then $e_2 : T$

- Principality
  Every typeable term has a principal type

ERASURE

Removing types and propositional content.

Or replace with □ if it cannot be removed.

Function : PCUIC → λ□

For first-order types: If we have $\Sigma; \Gamma \vdash t$ evaluates to v then $\varepsilon\Sigma \vdash \varepsilon t$ evaluates to εv.

ε erasure function.

EXAMPLE

LAMBDA BOX

PCUIC

```
Inductive term :=
| tRel (n : nat)
| tVar (i : ident)
| tEvar (n : nat) (l : list term)
| tSort (u : sort)
| tProd (na : aname) (A B : term)
| tLambda (na : aname) (A t : term)
| tLetIn (na : aname) (b B t : term)
| tApp (u v : term)
| tConst (k : kername) (ui : Instance.t)
| tInd (ind : inductive) (ui : Instance.t)
| tConstruct (ind : inductive) (n : nat) (ui : Instance.t)
| tCase (indn : case_info) (p : predicate term) (c : term)
    (brs : list (branch term))
| tProj (p : projection) (c : term)
...
```

λ□

```
Inductive term :=
| tBox
| tRel (n : nat)
| tVar (i : ident)
| tEvar (n : nat) (l : list term)
| tLambda (na : name) (t : term)
| tLetIn (na : name) (b t : term)
| tApp (u v : term)
| tConst (k : kername)
| tConstruct (ind : inductive) (n : nat) (args : list
   term)
| tCase (indn : inductive * nat) (c : term) (brs : list
   (list name * term))
| tProj (p : projection) (c : term)
...
```

EXAMPLE

LAMBDA BOX

PCUIC

```
Inductive term :=
| tRel (n : nat)
| tVar (i : ident)
| tEvar (n : nat) (l : list term)
| tSort (u : sort)
| tProd (na : aname) (A B : term)
| tLambda (na : aname) (A t : term)
| tLetIn (na : aname) (b B t : term)
| tApp (u v : term)
| tConst (k : kername) (ui : Instance.t)
| tInd (ind : inductive) (ui : Instance.t)
| tConstruct (ind : inductive) (n : nat) (ui : Instance.t)
| tCase (indn : case_info) (p : predicate term) (c : term)
    (brs : list (branch term))
| tProj (p : projection) (c : term)
...
```

λ□

```
Inductive term :=
| tBox
| tRel (n : nat)
| tVar (i : ident)
| tEvar (n : nat) (l : list term)
| tLambda (na : name) (t : term)
| tLetIn (na : name) (b t : term)
| tApp (u v : term)
| tConst (k : kername)
| tConstruct (ind : inductive) (n : nat) (args : list
   term)
| tCase (indn : inductive * nat) (c : term) (brs : list
   (list name * term))
| tProj (p : projection) (c : term)
...
```

ERASURE

EXAMPLE

```
true : bool
tConstruct Ind_bool 0 []   PCUIC
tConstruct Ind_bool 0 []   λ□

fun x : nat => x
tLambda name_x (tInd Ind_nat []) (tRel 0) PCUIC
tLambda name_x (tRel 0)                   λ□
```

λ□

```
Inductive term :=
| tBox
| tRel (n : nat)
| tVar (i : ident)
| tEvar (n : nat) (l : list term)
| tLambda (na : name) (t : term)
| tLetIn (na : name) (b t : term)
| tApp (u v : term)
| tConst (k : kername)
| tConstruct (ind : inductive) (n : nat) (args : list
    term)
| tCase (indn : inductive * nat) (c : term) (brs : list
    (list name * term))
| tProj (p : projection) (c : term)
...
```

ERASURE RELATION

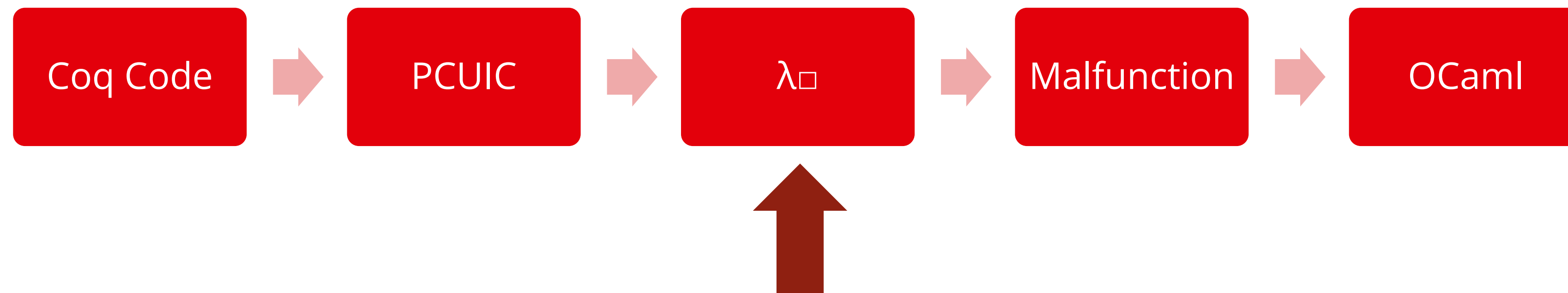
```
Inductive erases (Σ : global_context) (Γ : context) : term → eterm → Prop :=
| erases_tRel : ∀ i : N, Σ; Γ ⊢ tRel i ⇝ℰ E.tRel i
| erases_tVar : ∀ n : ident, Σ; Γ ⊢ tVar n ⇝ℰ E.tVar n
| erases_tLambda : ∀ (na : name) (b t : term) (t' : eterm),
    Σ; (na :a b :: Γ) ⊢ t ⇝ℰ t' → Σ; Γ ⊢ tLambda na b t ⇝ℰ E.tLambda na t'
```

```
fun x : nat => x
tLambda name_x (tInd Ind_nat []) (tRel 0)
tLambda name_x (tRel 0)
```

PCUIC

λ□

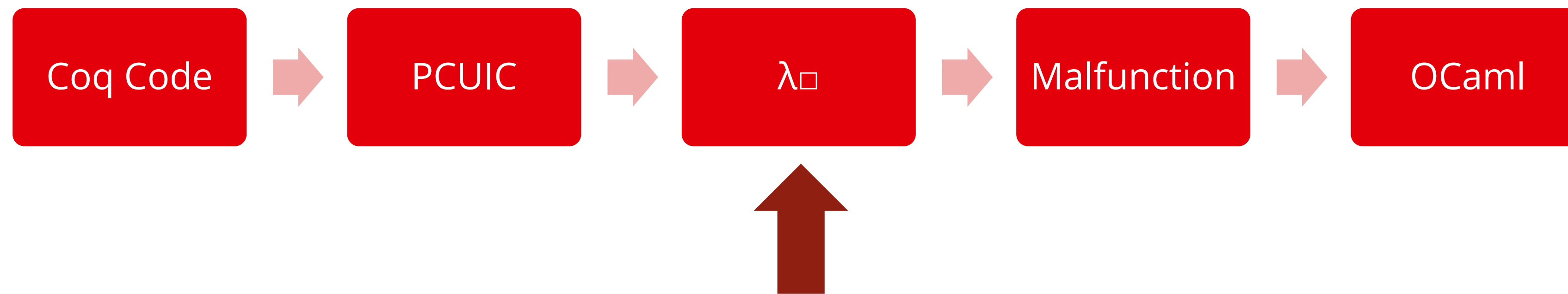
EXTRACTION PIPELINE



EXTRACTION PIPELINE

Difference between λ□ and Malfunction still very big.

Goal: Modify λ□ in small (manageable) steps to get close to semantics of Malfunction.



TRANSFORMATIONS

```
Record Transform.t :=
{ name : string;
  pre : program → Prop;
  post : program' → Prop;
  transform : ∀ p : program, pre p → program';
  correctness : ∀ input (p : pre input), post (transform input p);
  obseq : ∀ p : program, pre p → program' → value → value' → Prop;
  preservation : ∀ p v (pr : pre p), eval p v →
    let p' := transform p pr in
    ∃ v', eval' p' v' ∧ obseq p pr p' v v' }.
```

COMPOSING TRANSFORMATIONS

```
Definition Transform.compose :
  (o : Transform.t env env' term term' value value' eval eval’)
  (o' : Transform.t env' env'' term' term'' value' value'' eval' eval''),
  (∀ p : program env' term', post o p → pre o' p) →
  Transform.t env env'' term term'' value value'' eval eval''.

Notation " o ◁ o' " := (Transform.compose o o' _)
```

THE FULL PIPELINE

```
Definition erasure_pipeline : Transform.t :=
  build_template_program_env ◁
  eta_expand ◁
  template_to_pcuic_transform ◁
  erase_transform ◁
  guarded_to_unguarded_fix ◁
  remove_params_optimization ◁
  rebuild_wf_env_transform ◁
  optimize_prop_discr_optimization ◁
  rebuild_wf_env_transform ◁
  inline_projections_optimization ◁
  rebuild_wf_env_transform ◁
  constructors_as_blocks_transformation.
```

EXTRACTION AS DROP-IN REPLACEMENT

Current extraction has:

1. Set Extraction Optimize
2. .mli type interface
3. Extract separate .ml files following module structure.
4. Extract Constant and Extract Inductive.
5. Extract Inline.
6. It extracts terms of type $\{x : A \mid P\ x\}$, where P is a propositional predicate on A, to terms of type A in OCaml.

EXTRACTION AS DROP-IN REPLACEMENT

Current extraction has:

1. Set Extraction Optimize
2. .mli type interface
3. Extract separate .ml files following module structure.
4. Extract Constant and Extract Inductive.
5. Extract Inline.
6. It extracts terms of type $\{x : A \mid P\ x\}$, where P is a propositional predicate on A, to terms of type A in OCaml.

EXTRACTION AS DROP-IN REPLACEMENT

Current extraction has:

2. .mli type interface

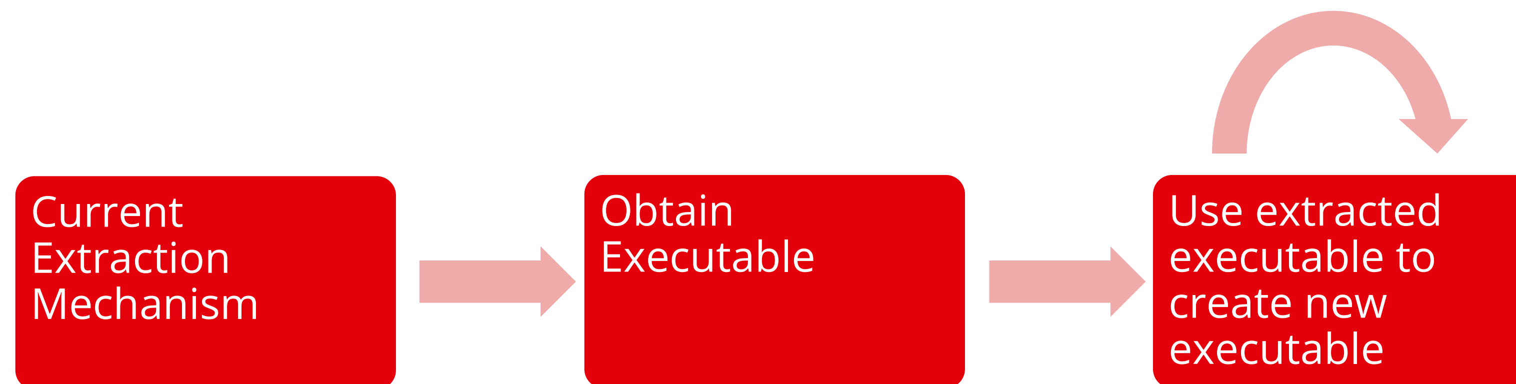
💡 Only generate such interface for code that will be interacted with.

6. It extracts terms of type $\{x : A \mid P\ x\}$, where P is a propositional predicate on A, to terms of type A in OCaml.

💡 Instead of outputting the pair $(x, \square)$, treat it as just an element of A.

BOOTSTRAPPED COMPILER

- Self-hosting compiler: Compiler that is implemented in the language it is also a compiler for.
- Chicken-and-egg problem: How to obtain the first executable compiler?



BENCHMARKS

Benchmark	ocamlc Extraction Optimize	ocamlopt Extraction Optimize	ocamlc	ocamlopt	CertiCoq gcc	CertiCoq gcc –O1	mlf –O0	mlf -O2
demo1	1.4	1.2	1.4	1.1	2.7	1.8	1.1	1.2
demo2	0.6	0.4	0.6	0.3	0.6	0.5	0.4	0.4
list_sum	5.2	1.8	4.2	1.7	4.1	5.2	1.9	1.7
vs_easy	1196.3	59.5	1390.6	74.4	190.2	154.2	181.1	65.5
vs_hard	5572.0	707.5	6331.9	684.9	1429.5	1268.6	1635.0	951.8
binom	971.0	182.5	963.1	141.5	166.6	174.5	150.4	150.1
color	X	X	X	X	785.5	706.1	1068.2	651.8
sha_fast	3076.5	1089.8	3167.9	914.9	1329.4	1306.2	1239.8	985.9

BENCHMARKS

Benchmark	ocamlc Extraction Optimize	ocamlopt Extraction Optimize	ocamlc	ocamlopt	CertiCoq gcc	CertiCoq gcc –O1	mlf –O0	mlf -O2
demo1	1.4	1.2	1.4	1.1	2.7	1.8	1.1	1.2
demo2	0.6	0.4	0.6	0.3	0.6	0.5	0.4	0.4
list_sum	5.2	1.8	4.2	1.7	4.1	5.2	1.9	1.7
vs_easy	1196.3	59.5	1390.6	74.4	190.2	154.2	181.1	65.5
vs_hard	5572.0	707.5	6331.9	684.9	1429.5	1268.6	1635.0	951.8
binom	971.0	182.5	963.1	141.5	166.6	174.5	150.4	150.1
color	X	X	X	X	785.5	706.1	1068.2	651.8
sha_fast	3076.5	1089.8	3167.9	914.9	1329.4	1306.2	1239.8	985.9

CONCLUSION

- Verified extraction pipeline

- Formally verified correctness of intermediate steps.
  - Some newly verified within ‘Verified extraction from Coq to OCaml’.
  - Some reused from ‘Coq Coq correct! Verification of type checking and erasure for Coq, in Coq‘.
- Can (in most cases) be used as a drop-in replacement to old extraction framework.
- Performance is on par.
- Bootstrapped compiler allows extraction to OCaml.

- Future work.