# Embedding Impure & Untrusted ML Oracles into Coq Verified Code

December 2018

Sylvain.Boulme@univ-grenoble-alpes.fr

# Contents

# CompCert, the 1st formally proved C compiler

**100Kloc** of Coq, developed since 2005 by Leroy-et-al at Inria

**Major success story** of software verification

- the "*safest C optimizing compiler*" from Regher-et-al@PLDI'11
- Commercial support since 2015 by AbsInt (German Company)
- Compile critical software for Avionics & Nuclear Plants
- See Käster-et-al@ERTS'18.

**Lesson 1** Focus on proving *critical* properties (e.g. functional correctness) instead of *non-critical* properties (e.g. performance). Actually, only consider **partial correctness**.

**Lesson 2** Use *untrusted* oracles when possible

# Untrusted oracles in CompCert

**Principle :** delegate computations to efficient external functions without having to prove them
⇒ only a checker of the result is verified
i.e. *verified defensive programming !*

**Example** of *register allocation* – a NP-complete problem
- finding a *correct* and *efficient* allocation is difficult
- verifying the *correctness* of an allocation is easy

⇒ only "*allocation checking*" is verified in Coq

**Benefits of untrusted oracles**

simplicity + efficiency + modularity

**NB** oracles needs to appear in Coq as "*foreign functions*"...

# Foreign functions in Coq : an unsound example

Standard method to declare a foreign function in Coq
"*Use an axiom declaring its type ; replace this axiom at extraction*"

```
Definition one: nat := (S O).

Axiom oracle: nat → bool.

Lemma congr: oracle one = oracle (S O).
  auto.
Qed.
```

With the OCaml implementation "`let oracle x = (x == one)`"

**Unsound** `(oracle one) = true` vs `(oracle (S O)) = false`
*Similar behavior* with side-effects instead.

**NB** OCaml "functions" are not functions in the math sense.
They are rather "non-deterministic functions" (ie "relations")

$\mathbb{P}(A \times B) \simeq A \rightarrow \mathbb{P}(B)$ where "$\mathbb{P}(B)$" is "$B \rightarrow$ **Prop**"

# Oracles in CompCert : a soundness issue ?

**Oracles are declared as pure functions**

Example of register allocation :

```
Axiom regalloc: RTL.func → option LTL.func.
```

implemented by imperative OCaml code using hash-tables.

Not a real issue because

**their purity is not used in the compiler proof !**

This talk proposes an approach to ensure such a claim...

# Limits of some experimental checkers in CompCert

Example of **Instruction scheduling** (yet another NP-hard pb)
Very elegant **translation validation** of J-B. Tristan's PhD (2009).
But still not in CompCert because the checker blows up !

This blow up could be "simply" fixed with hash-consing...
but, require to handle == (pointer equality) in Coq.

**This talks provides a formal (partial) axiom about ==**
Suffices for a proof of Tristan's checker with external hash-consing !

# Foreign Functions := *untrusted* oracles (in this talk)

- Embedding of arbitrary imperative ML functions into Coq.
(e.g. aliasing in Coq code is allowed)

- No reasoning on *effects*, only on returned values.
Intuition : *oracles* could have bugs, only their type is ensured
⇒ Foreign Functions are non-deterministic...
(e.g. for I/O reasoning, use `http://coq.io/` instead)

- Polymorphism to get "*theorems-for-free*" about
    - (some) invariant preservations by mutable data-structures
    - arbitrary recursion operators (needs a small defensive test)
    - exception-handling
    - ...

- Exceptionally : additional axioms (e.g. pointer equality)
In this case, the "*oracle*" must be trusted !

# Contents

# The (still open) quest of this talk

Define a class "*permissive*" of Coq types and a class "*safe*" of OCaml values such that

> a Coq type $T$ is "*permissive*" iff
>
> > any "*safe*" value compatible with the extraction of $T$ is soundly axiomatized in Coq with type $T$ (for partial correctness)

with "*being permissive*" and "*being safe*" automatically checkable and as expressive as possible !

Could lead to a Coq "`Import Constant`" construct

```
Import Constant ident: permissive_type
  := "safe_ocaml_value".
```

that acts like "`Axiom ident: permissive_type`",
but with additional checks during Coq and OCaml typechecking.

**Example** *safe*="well-typed" ⇒ "`nat→bool`" not *permissive*.

# May-return monads [Fouilhé, Boulmé'14]

**Axiomatize** "$\mathbb{P}(A)$" as type "$??A$"
to represent "*impure computations of type A*"
and "$(k\ a)$" as proposition "$k \rightsquigarrow a$"
with formal type $\rightsquigarrow_A : ??A \to A \to \mathrm{Prop}$
read "*computation k may return value a*"

**Formal operators and axioms**

- $\mathtt{ret}_A : A \to ??A$ *(interpretable as identity relation)*

$$(\mathtt{ret}\ a_1) \rightsquigarrow a_2 \quad \to \quad a_1 = a_2$$

- $\gg\!=_{A,B} : ??A \to (A \to ??B) \to ??B$
*(interpretable as the image of a predicate by a relation)*

$$(k_1 \gg\!= k_2) \rightsquigarrow b \quad \to \quad \exists a, k_1 \rightsquigarrow a \wedge k_2\ a \rightsquigarrow b$$

encodes OCaml "**let** $x = k_1$ **in** $k_2$" as "$k_1 \gg\!= (\mathtt{fun}\ x \Rightarrow k_2)$"

- $\mathrm{mk\_annot}_A(k : ??A) : ??\{\, a \mid k \rightsquigarrow a \}$
*(returns the* True *predicate)*

**NB** another interpretation is "$??A := A$" used for extraction !

# Usage of may-return monads

Used to declare oracles in the Verified Polyhedra Library
[Fouilhé, Maréchal, Monniaux, Périn, et. al, 2013-2018]
`github.com/VERIMAG-Polyhedra/VPL`

However, soundness of VPL design is currently only a conjecture !

**Example of Conjecture**
"`nat` $\rightarrow$ `??bool`" is *permissive* for any welltyped OCAML constant

**NB** For `oracle : nat` $\rightarrow$ `??bool` the below property is not provable

```
∀ b b', (oracle one)⇝b → (oracle (S O))⇝b' → b=b'.
```

# The issue of cyclic values

Consider the following Coq type

```
Inductive empty: Type:= Succ: empty → empty.
```

This type is proved to be empty. (Thm : `empty → False`).

Then, a function of `unit→??empty` is proved to never return.

Thus, `unit→??empty` is not permissive in presence of OCaml cyclic values like

```
let rec loop: empty = Succ loop
```

**My proposal**

Add an optional tag on OCaml type definitions to **forbid** cyclic values (typically, for inductive types extracted from Coq).

# Axioms of pointer equality also forbids cyclic values

In presence of the following axioms

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool.
Axiom phys_eq_true: ∀ A (x y: A),
  phys_eq x y ⇝ true → x=y.
```

where `phys_eq x y` is extracted on `x==y`,
the following OCaml value is unsound...

```
let rec fuel: nat = S fuel
```

**since** at runtime "`pred fuel == fuel`",
whereas it is easy to prove the following Coq goal

```
Goal ∀ (n:nat), pred n = n → n = O.
```

and to write a Coq function distinguishing `fuel` from `O`.

# Counter-examples and conjectures of "being permissive"

Here "safe" OCAML functions correspond to
"well-typed" functions (without "obj.magic" tricks)
and without cyclic-values on extracted types.

**Counter-Examples** the following types are not permissive

```
nat → bool                      (* extracted as   nat → bool      *)
nat → ??{ n:nat | n ≤ 10}       (*          nat → nat             *)
nat → ??(nat → nat)             (*          nat → (nat → nat)     *)
```

**Conjecture** the following types are permissive

```
nat → ??(nat → ?? nat)           (*     nat → (nat → nat)     *)
{ n:nat | n ≤ 10} → ?? nat       (*     nat → nat             *)
(nat → ?? nat) → ?? nat          (*     (nat → nat) → nat     *)
(nat → nat) → ?? nat             (*     (nat → nat) → nat     *)
∀ A, A*A → ??(list A)            (*     'a*'a → ('a list)     *)
```

# Embedding Imperative References into Coq

**Conjecture** permissivity of

```
Record cref{A}:={set: A→??unit; get: unit→??A}.

Axiom make_cref: ∀ {A}, A → ?? cref A.
```

Compatible with OCaml constants of "`'a -> 'a cref`", like

```
let make_cref x =
    let r = ref x in {
        set = (fun y -> r := y);
        get = (fun () -> !r) }
```

but also like

```
let make_cref x =
  let h = ref [x] in {
    set = (fun y -> h := y::!h);
    get = (fun () -> List.nth !h (Random.int (List.length !h))) }
```

⇒ No formal guarantee on reference contents
except **invariant preservations** encoded in **instances** of `A`.

# Permissivity of polymorphism ⇒ unary parametricity

Conjecturing that "`∀ A, A→??A`" is permissive,
we prove that any *safe* OCaml "`pid:'a -> 'a`" satisfies

when `(pid x)` returns normally some `y` then `y = x`.

## Proof

```
Axiom pid: ∀ A, A→??A.

(* We define below cpid:∀{B}, B → ??B *)
Program Definition cpid {B} (x:B): ?? B :=
  DO z ← pid { y | y = x } x ;;
  RET `z.

Lemma cpid_correct A (x y:A): (cpid x) ⇝ y → y=x.
```

At extraction, we get "`let cpid x = (let z = pid x in z)`".

⇒ mimicks a *"theorems for free"* of [Wadler'89]
i.e. a (unary) *parametricity proof* of [Reynolds'83]

# Unary parametricity for imperative type-systems

**Counter-example** : no parametricity with dynamic types *a la* Java

```
<A> A pid(A x) {
  if (x instanceof Integer)
    return (A)(new Integer(0));
  return x;
}
```

- Parametricity comes *intuitively* from the type-erasure semantics : polymorphic values must be handled uniformly.
- But, even hard to *formally define* with higher-order references : no elementary model of "*predicates over recursive heaps*" !
- Has been proved for a variant of system F with references by [Birkedal'11] (from the works of [Ahmed'02] and [Appel'07]).
- **Open Conjecture** for "Coq + ??. + OCaml"

# Contents

# Unary Parametricity : ML type → 2nd-order invariant

**Example** deriving a while-loop for Coq in partial correctness
from a (possibly non-terminating) ML oracle such that
ML type of the oracle ⇒ usual rule of Hoare Logic

Given definition of `wli` (while-loop-invariant)

```
Definition wli{S}(cond:S→bool)(body:S→??S)(I:S→Prop)
:= ∀ s, I s → cond s = true →
              ∀ s', (body s) ⇝ s' → I s'.
```

I aim to define

```
while {S} cond body (I: S→Prop | wli cond body I):
   ∀ s0, ??{s | (I s0 → I s) ∧ cond s = false}.
```

# Polymorphic oracle for loops

**Declaration of the oracle** in Coq

```
Axiom loop: ∀ {A B}, A * (A → ?? (A+B)) → ?? B.
```

$\begin{cases} \texttt{A} \mapsto \text{invariant} & \text{i.e. type of “may-reachable states”} \\ \texttt{B} \mapsto \text{post-condition} & \text{i.e. type of “may-final states”} \end{cases}$

**Implem.** in OCaml

```
let rec loop (a, step) =
  match step a with
  | Coq_inl a' -> loop (a', step)
  | Coq_inr b -> b
```

**Another implem** with recursion from a higher-order reference

```
let loop (a0, step) =
  let fix = ref (fun _ -> failwith "init") in
  (fix := fun a -> match step a with
                   | Coq_inl a' -> (!fix) a'
                   | Coq_inr b -> b);
  (!fix) a0
```

# Definition of the while-loop in Coq

```
Axiom loop: ∀ {A B}, A*(A → ?? (A+B)) → ?? B.
```

```
Definition wli{S}(cond:S→bool)(body:S→??S)(I:S→Prop)
:= ∀ s, I s → cond s = true →
            ∀ s', (body s) ⇝ s' → I s'.

Program Definition
  while {S} cond body (I:S→Prop | wli cond body I) s0
  : ??{s | (I s0 → I s) ∧ cond s = false}
:=
  loop (A:={s | I s0 → I s})
       (s0,
          fun s ⇒
          match (cond s) with
          | true ⇒
             DO s' ← mk_annot (body s) ;;
             RET (inl (A:={s | I s0 → I s }) s')
          | false ⇒
             RET (inr (B:={s | (I s0 → I s)
                               ∧ cond s = false}) s)
          end).
```

# A simple example using the while-loop in Coq

```
(* Specification of Fibonacci's numbers by a relation *)
Inductive isfib: Z → Z → Prop :=
 | isfib_base p: p ≤ 2 → isfib p 1
 | isfib_rec p n1 n2: isfib p n1 → isfib (p+1) n2 → isfib (p+2) (n1+n2).

(* Internal state of the iterative computation *)
Record iterfib_state := { index: Z; current: Z; old: Z }.

Program Definition iterfib (p:Z): ?? Z :=
  if p ≤? 2
  then RET 1
  else
    DO s ←
      while (fun s ⇒ s.(index) <? p)                       (* cond *)
            (fun s ⇒ RET {| index := s.(index)+1;          (* body *)
                            current := s.(old) + s.(current);
                            old:= s.(current) |})
            (fun s ⇒ s.(index) ≤ p                         (* I    *)
                     ∧ isfib s.(index) s.(current)
                     ∧ isfib (s.(index)-1) s.(old))
            {| index := 3; current := 2; old := 1 |};;     (* s0   *)
    RET (s.(current)).

(* Correctness of the iterative computation *)
Lemma iterfib_correct p r: iterfib p ⇝ r →isfib p r.
```

# Generalization to arbitrary recursion operators

For any oracle compatible with

```
fixp: ∀ {A B}, ((A → ?? B) → A → ?? B) → ?? (A → ?? B).
```

But, usual reasoning on **recursive functions** requires
a **relation** between inputs and outputs.
How to encode a *binary* relation into the "*unary invariant*" `B` ?

**Solution** use in Coq "`(B:=answ R)`" where

```
Record answ {A O} (R: A → O → Prop) := {
  input: A ;
  output: O ;
  correct: R input output
}.
```

+ a **defensive check** on each recursive result `r` that
`(input r)` "*equals to*" the actual input of the call

# Such a defensive check is needed...

because of well-typed oracles like

```
let fixp (step: ('a -> 'b) -> 'a -> 'b): 'a -> 'b =
  let memo = ref None in
  let rec f x =
    match !memo with
    | Some y -> y
    | None ->
       let r = step f x in
       memo := Some r;
       r
  in f
```

$\Rightarrow$ a memoized fixpoint with “a bug”
crashing all recursive results into a single memory cell.

Defensive check detects it and raises an exception (as later shown).

# But any `fixp` implementation is supported !

Standard fixpoint (== is sufficient in defensive check)

```
let fixp (step: ('a -> 'b) -> 'a -> 'b): 'a -> 'b =
  let rec f x = step f x in f
```

Memoized fixpoint (require structural equality in defensive check)

```
let fixp (step: ('a -> 'b) -> 'a -> 'b): 'a -> 'b  =
  let memo = Hashtbl.create 10 in
  let rec f x =
    try
      Hashtbl.find memo x
    with
      Not_found ->
        let r = step f x in
        Hashtbl.replace memo x r;
        r
  in f
```

# Properties of impure higher-order operators "for free"

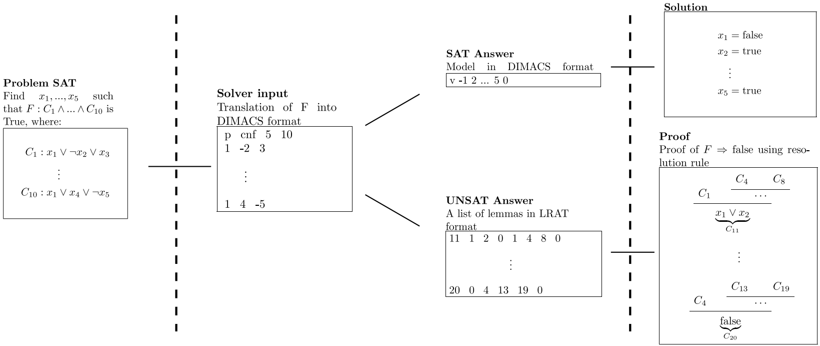
- (more adhoc) operators for loops and fixpoints
- raising and catching exceptions like in

```
Axiom fail: ∀ {A}, string → ?? A.

Definition FAILWITH {A} msg: ?? A :=
  DO r ← fail (A:=False) msg;;
  RET (match r with end).

Lemma FAILWITH_correct A msg (P:A → Prop):
  ∀ r, FAILWITH msg ⇝ r → P r.
```

- a "*design pattern*" where *all* oracles
  are polymorphic higher-order operators (as soon as it's useful)

# Contents

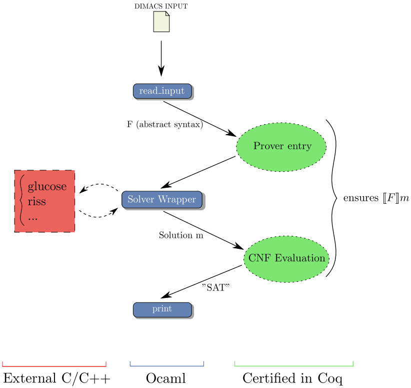
# Certifying boolean SAT-solvers answers (state-of-the-art)



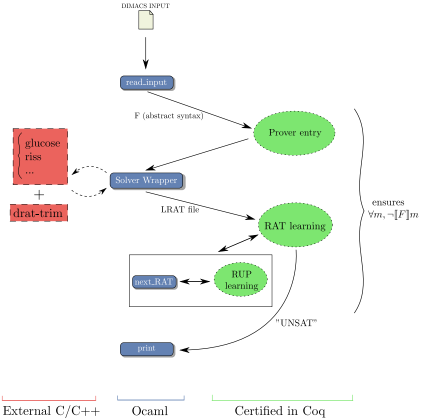

**UNSAT certificates** mandatory for SAT compet' since 2016.
Main format : DRUP/DRAT
Translated by the DRAT-TRIM untrusted checker (written in C)
into the more detailed LRAT-format verified by
a *certified* checker *extracted in* C *from* ACL2
Tool-chain from [Heule et al, 2013-2017].

# Architecture of our SatAnsCert (with T. Vandendorpe)

## SAT Answer

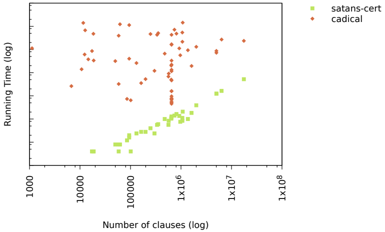


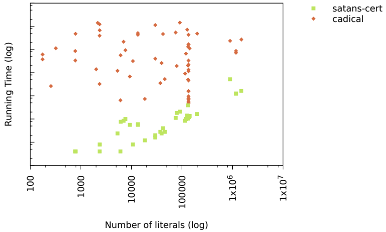

## UNSAT Answer



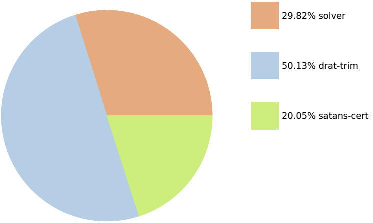
# Mean running times of SatAnsCert

**SAT** with the CaDiCaL SAT-solver
on the 120 instances of the SAT competition 2018 benchmarks.




**UNSAT** with both CaDiCaL and CryptoMiniSat SAT-solvers
on 306 instances from the SAT competition 2015,2016,2018 benchmarks



# Introduction to the correctness of SatAnsCert

Formal proof from CNF abstract syntax :
I/O of SatAnsCert are not verified !

Main written in Coq with **statically verified** “ASSERT”

```
Program Definition main: ?? unit :=
  TRY
    DO f ← read_input();;  (* Command-line + CNF parsing *)
    DO a ← sat_solver f;;  (* solver(+drat-trim) wrapper *)
    match a with
    | SAT_Answer mc ⇒
      assert_b (satProver f mc) "wrong SAT model";;
      ASSERT (∃ m, ⟦f⟧ m);;
      println "SAT !"
    | UNSAT_Answer ⇒
      unsatProver f;;
      ASSERT (∀ m, ¬⟦f⟧ m);;
      println "UNSAT !"
  WITH e ⇒
    DO s ← exn2string e;;
    println ("Certification failure: " +; s).
```

# Specification of a "simplified" refutation prover

**(Boolean) variable** $x$ (encoded as a positive).

**Literal** $\ell \triangleq x$ or $\neg x$.

**Clause** $C \triangleq$ a finite disjunction of literals
(encoded as a finite set of literals).

**CNF** $F \triangleq$ a finite conjunction of clauses
(encoded as a list of clauses).

```
unsatProver (f: list clause): ?? (∀ m, ¬⟦f⟧ m)
```

In the following, a simplified sketch of the implementation...
Full code on `github.com/boulme/satans-cert`

# Background on backward resolution proofs ($\subseteq$ RUP proofs)

**Def** given the derivation rules

$$\textsc{Triv}\ \frac{C_1}{C_2}\ C_1 \backslash C_2 = \emptyset \qquad\qquad \textsc{BckRsl}\ \frac{C_1 \qquad \{\neg\ell\} \cup C_2}{C_2}\ C_1 \backslash C_2 = \{\ell\}$$

We write "$C_1, \ldots, C_n \vdash C$" iff

$$\textsc{BckRsl}\ \frac{C_1 \qquad \textsc{BckRsl}\ \dfrac{C_{n-1} \qquad \textsc{Triv}\ \dfrac{C_n}{\cdots}}{\cdots}}{C}$$

**Thm** $F$ is UNSATISFIABLE iff
it exists a sequence of $C_1, \ldots, C_n$ such that

- forall $i \in [1, n-1]$, it exists $L \subseteq F \cup \{C_1, \ldots, C_{i-1}\}$ with $L \vdash C_i$
- $C_n = \emptyset$

# UNSAT certificates from learned clauses

learned clause = RUP lemma found by the CDCL SAT-solver

- **DRUP format** from CDCL solver
  a list of learned claused ended by clause $\emptyset$

- **LRAT format** from DRAT-TRIM
  for each learned clause $C$,
  a list of *previously* learned clauses (or axioms) $L$
  such that $L \vdash C$
  i.e. L is "**Backward Resolution Chain** learning $C$"

**NB** We also support RAT clauses : out the scope of this talk !

# Learned clauses in Coq from Backward Resolution Chains

On `F:(list clause)`, define type `cc`$[\![F]\!]$ of "*consequences*" of $F$.

```
Record cc(s:model→Prop): Type :=
  { rep: clause; rep_sat: ∀ m, s m →⟦rep⟧m }.
```

Then, we define emptyness test :

```
assertEmpty {s}: cc s → ??(∀ m, ¬(s m)).
```

Learning a clause (from a BRC) is defined by

```
learn: ∀{s}, list(cc s) → clause → ??(cc s)
```

implemented such that (for "performance" only)
if `l` $\vdash$ `c` then `(learn l c)` returns `c'` where `(rep c')=c`.

# Toward "*Logical Consequence Factories*" (LCF)

**Idea** an oracle (≈ a LRAT parser) computes directly "certified learned clauses" through a certified API (called a LCF).
⇒ No need of an explicit "proof object" (like in old LCF prover) !

**For the following benefits**

- Backward Resolution Chains are verified "on-the-fly", **in the oracle** (much easier to debug)
- map of *clause identifiers* to *clause values* : only managed by the oracle (in a efficient hash-table)
- deletion of clauses from memory : only managed by the oracle.
- very simple & small Coq code

**Dev of whole SatAnsCert** in 2 person.months for 1Kloc of Coq + 1Kloc of OCaml files (including `.mll` files)

# Polymorphic LCF style

## **Declaration of the oracle** in Coq

```
Definition lcf A := (list A) → clause → ?? A.
Axiom lratParse: ∀ {A}, (lcf A)*list(clause_ident*A) → ?? A.
```

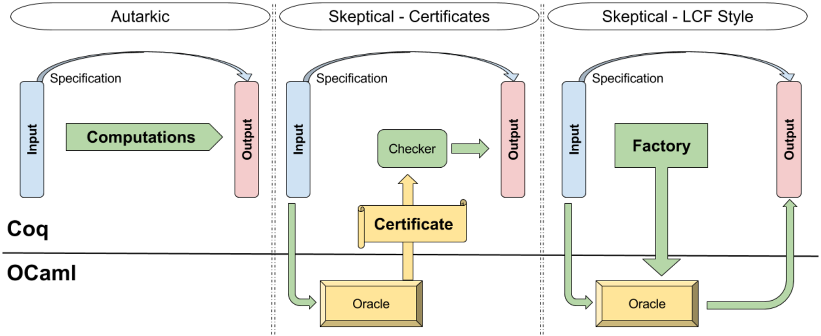
- Data-abstraction is provided by polymorphism !
  type "`A`" is abstract type of *learned clauses*
  type "`lcf A`" = abstraction of certified BRC checking
- In input, each clause both given as an ident and an abstract "axiom" of type `A`.

## **Implem.** of `unsatProver` in Coq

```
Definition mkInput (f: list clause):
  lcf(cc⟦f⟧) * list(clause_ident*(cc⟦f⟧))
:= ...

Definition unsatProver f: ?? (∀ m, ¬⟦f⟧m) :=
  DO c ← lratParse (mkInput f);;
  assertEmpty c.
```

# Contents

# 3 styles of Coq verified code




**In this talk** *Polymorphic LCF style*

Oracles computes directly "correct-by-construction" results through an API certified from Coq

(where type abstraction comes from polymorphism)

# Feedback from the Verified Polyhedra Library

**Benefits** of switching from "Certificates" to "LCF style".

- Code size on the interface Coq/Ocaml divided by 2 :
  *shallow* versus *deep* embedding (of certificates).
- Interleaved execution of untrusted and certified computations :
  Oracles debugging much easier.

See [Maréchal Phd'17].

Generating certificates still possible from LCF style oracles.
See our Coq tactic for learning equalities in linear rational arithmetic [Boulmé & Maréchal @ ITP'18].

# (Partial) Conclusion

I propose to combine Coq and OCaml typecheckers to get

Imperative programming with "Theorems for free !"
and all this for *almost* free !

Mostly need to understand the meta-theory of this proposal
*Is there any motivated type-theorist in the room ?*