Embedding Impure & Untrusted ML Oracles into CoQ Verified Code

December 2018

Sylvain.Boulme@univ-grenoble-alpes.fr
Contents

Motivations from COMPCERT successes and weaknesses

A Foreign Function Interface for Coq (programming)

Coq “Theorems for free” about Polymorphic Foreign Functions

Certifying Answers of (State-of-the-art) Boolean SAT-Solvers

Conclusions
**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"

\begin{Verbatim}
Definition one: nat := (S O).
Axiom oracle: nat \to bool.
Lemma congr: oracle one = oracle (S O).
   auto.
Qed.
\end{Verbatim}

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")
\[ \mathbb{P}(A \times B) \simeq A \to \mathbb{P}(B) \]
where \( \mathbb{P}(B) \) is \( B \to \text{Prop} \)
Oracles in CompCert: a soundness issue?

Oracles are declared as pure functions
Example of register allocation:

\[
\text{Axiom } \text{regalloc: RTL.func } \rightarrow \text{ option LTL.func.}
\]

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)
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

Motivations from COMPCERT successes and weaknesses

A Foreign Function Interface for CoQ (programming) ??

CoQ “Theorems for free” about Polymorphic Foreign Functions

Certifying Answers of (State-of-the-art) Boolean SAT-Solvers

Conclusions
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” $\Rightarrow$ “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 “\(k \leadsto a\)"

with formal type \(\leadsto_{A}: ??A \rightarrow A \rightarrow \text{Prop}\)

read “computation \(k\) may return value \(a\)”

Formal operators and axioms

- \(\text{ret}_A : A \rightarrow ??A\) (interpretable as identity relation)

  \[(\text{ret} \ a_1) \leadsto a_2 \rightarrow a_1 = a_2\]

- \(\gg =_{A,B} : ??A \rightarrow (A \rightarrow ??B) \rightarrow ??B\)

  (interpretable as the image of a predicate by a relation)

  \[(k_1 \gg = k_2) \leadsto b \rightarrow \exists a, k_1 \leadsto a \land k_2 \ a \leadsto b\]

  encodes OCAML “let \(x = k_1\) in \(k_2\)” as “\(k_1 \gg = (\text{fun} \ x \Rightarrow k_2)\)”

- \(\text{mk\_annot}_A(k : ??A) : ??\{ a \mid k \leadsto a\}\)

  (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 → ??bool” is permissive for any welltyped OCAML constant

\( \forall b b', (\text{oracle one}) \rightarrow b \rightarrow (\text{oracle (S 0)}) \rightarrow b' \rightarrow b = b'. \)
The issue of cyclic values

Consider the following Coq type

\[
\text{Inductive empty : Type := Succ : empty } \rightarrow \text{ empty.}
\]

This type is proved to be empty. (Thm : empty \rightarrow False).

Then, a function of \( \text{unit } \rightarrow \text{ empty} \) is proved to never return.

Thus, \( \text{unit } \rightarrow \text{ empty} \) is not permissive in presence of Ocaml cyclic values like

\[
\text{let rec loop : empty = Succ loop}
\]

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

\begin{align*}
\textbf{Axiom} \quad \text{phys\_eq} &: \forall \ A, A \rightarrow A \rightarrow \text{bool}. \\
\textbf{Axiom} \quad \text{phys\_eq\_true} &: \forall \ A \ (x \ y : A), \\
&\quad \text{phys\_eq} \ x \ y \ \Rightarrow \ true \ \rightarrow \ x = y.
\end{align*}

where \text{phys\_eq} \ x \ y is extracted on \text{x==y},
the following \text{OCAML} value is unsound...

\begin{center}
let rec fuel: nat = S fuel
\end{center}

\textbf{since} at runtime “pred fuel == fuel”,
whereas it is easy to prove the following \text{COQ} goal

\begin{align*}
\textbf{Goal} \quad \forall \ (n:\text{nat}), \ \text{pred} \ n = n \ \rightarrow \ n = 0.
\end{align*}

and to write a \text{COQ} function distinguishing fuel from 0.
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

<table>
<thead>
<tr>
<th>Type</th>
<th>注释</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nat → bool</code></td>
<td>（* extracted as <code>nat → bool</code> *）</td>
</tr>
<tr>
<td>`nat → ??{ n:nat</td>
<td>n ≤ 10}`</td>
</tr>
<tr>
<td><code>nat → ??(nat → nat)</code></td>
<td>（* <code>nat → (nat → nat)</code> *）</td>
</tr>
</tbody>
</table>

**Conjecture** the following types are permissive

<table>
<thead>
<tr>
<th>Type</th>
<th>注释</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nat → ??(nat → ?? nat)</code></td>
<td>（* <code>nat → (nat → nat)</code> *）</td>
</tr>
<tr>
<td>`{ n:nat</td>
<td>n ≤ 10} → ?? nat`</td>
</tr>
<tr>
<td><code>(nat → ?? nat) → ?? nat</code></td>
<td>（* <code>(nat → nat) → nat</code> *）</td>
</tr>
<tr>
<td><code>(nat → nat) → ?? nat</code></td>
<td>（* <code>(nat → nat) → nat</code> *）</td>
</tr>
<tr>
<td><code>∀ A, A*A → ??(list A)</code></td>
<td>（* <code>’a*’a → (’a list)</code> *）</td>
</tr>
</tbody>
</table>
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 \(\Rightarrow\) unary parametricity

Conjecturing that "\(\forall\ A,\ A \rightarrow \text{??}A\)" is permissive, we prove that any safe OCAML "\(\text{pid:}'a \rightarrow 'a\)" satisfies when \((\text{pid} \ x)\) returns normally some \(y\) then \(y = x\).

Proof

Axiom \(\text{pid}: \forall\ A,\ A \rightarrow \text{??}A\).

\((\text{* We define below \(\text{cpid:}\forall\{B\},\ B \rightarrow ?B\ *)})\)

Program Definition \(\text{cpid}\ \{B\}\ (x:B): \text{??} B :=\)

\(\text{DO}\ \ z \leftarrow \text{pid} \ \{ y \mid y = x \} \ x \ \;;\)

\(\text{RET} \ 'z.\)

Lemma \(\text{cpid\_correct}\ A\ (x\ y:A): (\text{cpid} \ x) \leadsto y \rightarrow y=x.\)

At extraction, we get "let \(\text{cpid} \ x = (\text{let} \ z = \text{pid} \ x \ \text{in} \ z)\)".

\(\Rightarrow\) 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

```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

Motivations from COMPCERT successes and weaknesses

A Foreign Function Interface for Coq (programming)

Coq “Theorems for free” about Polymorphic Foreign Functions

Certifying Answers of (State-of-the-art) Boolean SAT-Solvers

Conclusions
Unary Parametricity: ML type $\rightarrow 2^{nd}$-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 $\Rightarrow$ usual rule of Hoare Logic

Given definition of $\text{wli}$ (while-loop-invariant)

```
Definition wli{S}(cond:S$\rightarrow$bool)(body:S$\rightarrow$??S)(I:S$\rightarrow$Prop)
:= $\forall$ s, I s $\rightarrow$ cond s = true $\rightarrow$
    $\forall$ s', (body s) $\Rightarrow$ s' $\rightarrow$ I s'.
```

I aim to define

```
while {S} cond body (I: S$\rightarrow$Prop | wli cond body I):
    $\forall$ s0, ??{s | (I s0 $\rightarrow$ I s) $\land$ cond s = false}.
```
Polymorphic oracle for loops

**Declaration of the oracle in CoQ**

Axiom loop: \( \forall \{A B\}, A \times (A \to ?? (A+B)) \to ?? B \).

\[
\begin{align*}
A & \mapsto \text{invariant} \quad \text{i.e. type of “may-reachable states”} \\
B & \mapsto \text{post-condition} \quad \text{i.e. type of “may-final states”}
\end{align*}
\]

**Impl. 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: \( \forall \{A\ B\}, A*(A \rightarrow ?? (A+B)) \rightarrow ?? B. \)

Definition \( \text{wli}\{S\}(\text{cond:}S\rightarrow\text{bool})(\text{body:}S\rightarrow??S)(I:S\rightarrow\text{Prop}) \) := \( \forall s, I\ s \rightarrow \text{cond}\ s = \text{true} \rightarrow \forall s', (\text{body}\ s) \leadsto s' \rightarrow I\ s'. \)

Program Definition
while \{S\} \text{cond} \ \text{body} \ (I:S\rightarrow\text{Prop} \ | \ \text{wli} \ \text{cond} \ \text{body} \ I)\ s0 := ??\{s \mid (I\ s0 \rightarrow I\ s) \land \text{cond}\ s = \text{false}\}\ :
:=
loop \((A:=\{s \mid I\ s0 \rightarrow I\ s\})\)
(s0,
fun s \Rightarrow
match (\text{cond} s) \text{ with}
| \text{true} \Rightarrow
DO s' \leftarrow \text{mk_annot} (\text{body} s) ;;
RET (\text{inl} (A:=\{s \mid I\ s0 \rightarrow I\ s \})\ s')
| \text{false} \Rightarrow
RET (\text{inr} (B:=\{s \mid (I\ s0 \rightarrow I\ s) \land \text{cond} s = \text{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 |};;
  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

\[ \text{fixp: } \forall \{A B\}, ((A \to ?? B) \to A \to ?? B) \to ?? (A \to ?? B). \]

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

\[
\text{Record answ } \{A O\} (R: A \to O \to \text{Prop}) := \{
\begin{array}{l}
\text{input: } A ; \\
\text{output: } O ; \\
\text{correct: } R \text{ input output}
\end{array}
\}.
\]

+ a defensive check on each recursive result r that

(\text{input r} \text{ "equals to" } \text{the actual input of the call}
Such a defensive check is needed...

because of well-typed oracles like

```ocaml
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

⇒ 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)

```plaintext
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)

```plaintext
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 : forall {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):
    forall r, FAILWITH msg ⇒ r → P r.
```

- a “design pattern” where all oracles
  are polymorphic higher-order operators (as soon as it’s useful)
Contents

Motivations from COMPCERT successes and weaknesses

A Foreign Function Interface for COQ (programming) ??

COQ “Theorems for free” about Polymorphic Foreign Functions

Certifying Answers of (State-of-the-art) Boolean SAT-Solvers

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

**Problem SAT**
Find $x_1, ..., x_5$ such that $F : C_1 \land ... \land C_{10}$ is True, where:

- $C_1 : x_1 \lor \neg x_2 \lor x_3$
- $C_10 : x_1 \lor x_4 \lor \neg x_5$

**Solver input**
Translation of $F$ into DIMACS format:

```plaintext
p cnf 5 10
1 -2 3
: :
1 4 -5
```

**SAT Answer**
Model in DIMACS format:

```
v -1 2 ... 5 0
```

**UNSAT Answer**
A list of lemmas in LRAT format:

```
11 1 2 0 1 4 8 0
: :
20 0 4 13 19 0
```

**Solution**
```
z_1 = false
z_2 = true
: :
z_5 = true
```

**Proof**
Proof of $F \Rightarrow$ false using resolution rule:

```
C_1  C_4  C_8
\hline
x_1 \lor x_2
C_{11}
: :
\hline
C_4  C_{13}  C_{19}
\hline
\text{false}  
```

**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 \textbf{SatAnsCert} (with T. Vandendorpe)
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 \texttt{SatAnsCert}

Formal proof from CNF abstract syntax:

I/O of \texttt{SatAnsCert} are not verified!

Main written in \texttt{Coq} with \textit{statically verified} “\texttt{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, \llbracket f \rrbracket m);;
          println "SAT !"
        | UNSAT_Answer ⇒
          unsatProver f;;
          ASSERT (∀ m, \neg \llbracket f \rrbracket m);;
          println "UNSAT !"
      WITH e ⇒
        DO s ← exn2string e;;
        println ("Certification failure: " +; s).
```

Certifying Answers of (State-of-the-art) Boolean SAT-Solvers
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).

\[
\text{unsatProver (f: list clause): ?? (} \forall m, \neg[f] m)\]

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

**Def** given the derivation rules

\[
\begin{align*}
\text{Triv} & \quad \frac{C_1}{C_2} \quad C_1 \setminus C_2 = \emptyset \\
\text{BckRsl} & \quad \frac{C_1 \{\neg \ell\} \cup C_2}{C_2} \quad C_1 \setminus C_2 = \{\ell\}
\end{align*}
\]

We write “\(C_1, \ldots, C_n \vdash C\)” iff

\[
\begin{align*}
\text{Thm} & \quad F \text{ is UNSATISFIABLE} \quad \text{iff} \\
& \quad \text{it exists a sequence of } C_1, \ldots, C_n \text{ such that} \\
& \quad \quad \text{forall } i \in [1, n-1], \text{ it exists } L \subseteq F \cup \{C_1, \ldots, C_{i-1}\} \text{ with } L \vdash C_i \\
& \quad \quad \text{\large \checkmark} \quad C_n = \emptyset
\end{align*}
\]
UNSAT certificates from learned clauses

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

- **DRUP format** from CDCL solver
  a list of learned clauses ended by clause ∅

- **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 $F: (\text{list clause})$, define type $\text{cc}[F]$ of “consequences” of $F$.

\begin{verbatim}
Record cc(s:model → Prop): Type :=
    { rep: clause; rep_sat: ∀ m, s m → [rep] m }.
\end{verbatim}

Then, we define emptiness test:

\begin{verbatim}
assertEmpty {s}: cc s → ??(∀ m, ¬(s m)).
\end{verbatim}

Learning a clause (from a BRC) is defined by

\begin{verbatim}
learn: ∀{s}, list(cc s) → clause → ??(cc s)
\end{verbatim}

implemented such that (for “performance” only)

if $l \vdash c$ then $(\text{learn } l \ c)$ returns $c'$ where $(\text{rep } c')=c$. 

Certifying Answers of (State-of-the-art) Boolean SAT-Solvers
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 .mli files)
Polymorphic LCF style

**Declaration of the oracle in CoQ**

```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.

**Implm. of unsatProver in CoQ**

```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

Motivations from COMPCERT successes and weaknesses

A Foreign Function Interface for COQ (programming) ??

COQ “Theorems for free” about Polymorphic Foreign Functions

Certifying Answers of (State-of-the-art) Boolean SAT-Solvers

Conclusions
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?