Embedding Impure & Untrusted ML Oracles into Coq Verified Code

December 2018

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

Motivations from COMP\textsc{Cert} 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

(Overview mainly inspired from Käster-et-al@ERTS’18)

**Input** most of ISO C99 + a few extensions

**Output** (32&64 bits) code for PowerPC, ARM, x86, RISC-V

**Developed** since 2005 by Leroy-et-al at Inria

≈ 100 Kloc of Coq @ 6 person-years

Commercial support since 2015 by AbsInt (German Company)

Industrial uses in Avionics (Airbus) & Nuclear Plants (MTU)

for safety-critical embedded systems

**Performance of generated code** (for PowerPC and ARM)

2\times faster than gcc -00

10\% slower than gcc -01 and 20\% than gcc -03.

**Example of success story**

In MTU systems (German provider of Nuclear Power Plants)

28\% smaller WCET than with a previous unverified compiler!
Two “modesty” lessons from COMPCERT success

Success depicted by Yang, Chen, Eide & Regehr @PLDI’11 (concluding a huge campaign of random testing on C compilers)

“COMPCERT is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. [...] developing compiler optimizations within a proof framework [...] has tangible benefits for compiler users.”

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

(no formal guarantee on output existence)

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 0).
Axiom oracle : nat → bool.
Lemma congr : oracle one = oracle (S 0).
    auto.
Qed.
```

With the Ocaml implementation "let oracle x = (x == one)"

**Unsound!** Because at runtime, (oracle one) returns true whereas (oracle (S 0)) returns false.

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

**NB** \( \mathbb{P}(A \times B) \simeq A \rightarrow \mathbb{P}(B) \) where "\( \mathbb{P}(B) \)" is "\( B \rightarrow \text{Prop} \)"
Oracles are declared as pure functions
Example of register allocation:

\textbf{Axiom} \texttt{regalloc: RTL.func \to option LTL.func}.

implemented by imperative \texttt{OCAML} code using hash-tables.

Not a real issue because

\textit{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 == (physical equality) in CoQ.

**This talks provides a formal (partial) axiom about ==**  
Suffices for a proof of Tristan’s checker with 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. physical equality)
  In this case, the “oracle” must be trusted!
Contents

Motivations from COMP'CERT 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 constants such that

a Coq type $T$ is “permissive” iff
any “safe” constant 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_constant".
```

that acts like “Axiom ident: permissive_type”,
but with additional checks during Coq and Ocaml typechecking.

Example safe=“well-typed” $\Rightarrow$ “nat$\rightarrow$bool” not permissive.
May-return monads [Fouilhé, Boulmé’14]

**Axiomatize “**\(P(A)\)**” 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** “\(\text{let } x = k_1 \text{ in } k_2\)” as “\(k_1 \gg= (\text{fun } x \Rightarrow k_2)\)”

- \(\text{mk_annot}_A(k : ??A) : ??\{ a | k \leadsto a\}\) (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

NB For oracle : nat → ??bool the below property is not provable

∀ b b’, (oracle one)⇝b → (oracle (S 0))⇝b’ → 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 \text{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 phys. equality also forbids cyclic values

In presence of the following axioms

\begin{align*}
\text{Axiom } \text{phys}_\text{eq}: & \forall \{A\}, A \to A \to \text{bool}.
\text{Axiom } \text{phys}_\text{eq}_\text{true}: & \forall A \; (x \; y: A),
\text{phys}_\text{eq} \; x \; y \; \leadsto \; \text{true} \; \to \; x=y.
\end{align*}

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

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

since at runtime “pred fuel == fuel”,
whereas it is easy to prove the following CoQ goal

\begin{align*}
\text{Goal } & \forall (n:\text{nat}), \; \text{pred} \; n = n \to n = 0.
\end{align*}

and to write a 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>Remarks</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>Remarks</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

```coq
Record cref\{A\} := \{set: A \to \text{unit}; \text{get}: \text{unit} \to A\}.
```

**Axiom** `make_cref`: \(\forall \{A\}, A \to \text{cref } A\).

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

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

but also like

```coq
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)))
  }
```

\(\Rightarrow\) No formal guarantee on reference contents except **invariant preservations** encoded in **instances** of A.
Permissivity of polymorphism $\Rightarrow$ unary parametricity

Conjecturing that “$\forall A, A \rightarrow ??A$” 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: $\forall A, A \rightarrow ??A$.

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

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) $\Rightarrow$ y $\rightarrow$ y=x.

At extraction, we get “let cpid x = (let z = pid x 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
Deriving “for free” a while-loop in partial correctness

Unary parametricity: \( ML \text{ type } \rightarrow 2^{nd}\text{-order (impredicative)} \text{ invariant} \)

Example
Let us define a while-loop in \( Coq \)
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)

\[
\text{Record wli \{S\} (cond:S \rightarrow \text{bool}) (body:S \rightarrow ?? S) (s0:S) (I:S \rightarrow \text{Prop}) : \text{Prop} :=
\begin{array}{l}
\{ \text{while_init: } I \ s0; \\
\text{while_preserv: } \forall \ s, \ I \ s \rightarrow \text{cond s = true } \rightarrow \forall \ s', \ (\text{body s}) \leadsto s' \rightarrow I \ s' \}
\end{array}
\]

I aim to define

\[
\text{while cond body:} \forall \ s0 \ I, \ (\text{wli cond body s0 I}) \rightarrow ?? \{s \mid I \ s \land \text{cond s = false}\}.
\]
Polymorphic oracle for loops

**Declaration of the oracle in CoQ**

\[ \text{Axiom } \text{loop} : \forall \{A \ B\}, A \times (A \rightarrow ??? (A+B)) \rightarrow ??? B. \]

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

**Implem. in OCAML**

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

**Another implem** with a cyclic higher-order reference

```ocaml
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
```


Coq “Theorems for free” about Polymorphic Foreign Functions
Definition of the while-loop in CoQ

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

Record wli \{S\} (cond:S \rightarrow bool) (body:S \rightarrow ?? S) (s0:S) (I:S \rightarrow Prop): Prop :=
{ while_init: I s0;
  while_preserv: \( \forall \ s, I \ s \rightarrow \text{cond} \ s = \text{true} \rightarrow \forall \ s', (\text{body} \ s) \Rightarrow s' \rightarrow I \ s' \}.

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

Coq “Theorems for free” about Polymorphic Foreign Functions
Generalization to arbitrary recursion operators

For any oracle compatible with

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

But, **usual rule** for **recursive functions** requires a **relation** between inputs and outputs.

How to encode a **binary** relation into the "**unary invariant**" \(B\) ?

**Solution** use \((B \coloneqq \text{answ } R)\) with

\[
\begin{align*}
\text{Record } \text{answ} \ \{A \ O\} \ (R: A \rightarrow O \rightarrow \text{Prop}) & := \{ \\
& \text{input: } A \\
& \text{output: } O \\
& \text{correct: } R \ \text{input} \ \text{output}
\}.
\end{align*}
\]

+ a **defensive check** on each recursive result \(r\) that \((\text{input } r) \ "equals to" \) the actual input of the call
Such a defensive check is needed...

because of the following well-typed ML implementation

```ml
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 “bugs”
where recursive results are crashed 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)

```ocaml
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)

```ocaml
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

Motivations from \textsc{Compcert} successes and weaknesses

A Foreign Function Interface for \textsc{Coq} (programming)

\textsc{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 $x_1, \ldots, x_5$ such that $F : C_1 \land \ldots \land C_{10}$ is True, where:

- $C_1 : x_1 \lor \neg x_2 \lor x_3$
- $\vdots$
- $C_{10} : x_1 \lor x_4 \lor \neg x_5$

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

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

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

```
11 1 2 0 1 4 8 0
1 4 -5
20 0 4 13 19 0
```

**Solution**
- $x_1 = \text{false}$
- $x_2 = \text{true}$
- $\vdots$
- $x_5 = \text{true}$

**Proof**
Proof of $F \Rightarrow \text{false}$ using resolution rule

- $C_1 \quad C_4 \quad C_8$
- $x_1 \lor x_2$
- $\vdots$
- $C_4 \quad C_{13} \quad C_{19}$
- $\text{false}$
- $C_{20}$

**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)
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!

(Simplified) main written in Coq with verified “ASSERT”

Program Definition main : ?? unit :=
  DO f ← read_input();; (* Command-line + CNF parsing *)
  DO a ← sat_solver f;; (* SAT-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 !".
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): ?? ($\forall m, \neg \text{[f]} m$)

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

**Thm (Resolution proof system)** \( F \) is UNSATISFIABLE iff clause \( \emptyset \) is derivable from

\[
\begin{align*}
\text{Axiom} & \quad C \in F \\
\text{Resol} & \quad \{\ell\} \cup C_1' \quad \{-\ell\} \cup C_2' \\
& \quad C_1' \cup C_2' \\
\end{align*}
\]

Rule \( \text{Resol} \) equivalently split in two rules for backward checking

\[
\begin{align*}
\text{BckRsl} & \quad \{\ell\} \cup C_1' \quad \{-\ell\} \cup C \\
& \quad C \quad C_1' \subseteq C \\
\text{Trivial} & \quad C_2' \quad C_2' \subseteq C
\end{align*}
\]

equivalently rewritten in

\[
\begin{align*}
\text{BckRsl} & \quad C_1 \quad \{-\ell\} \cup C \\
& \quad C \quad C_1 \setminus C = \{\ell\} \\
\text{Trivial} & \quad C_1 \quad C_1 \setminus C = \emptyset
\end{align*}
\]
Resolution Chains & Conflict-Driven Clause Learning DPLL

A **Backward Resolution Chain** (BRC) w.r.t a list of axioms $F$
$= \text{specialization of } \text{BckRsl} \text{ and } \text{Trivial}$ when $C_1 \in F$

\[
\begin{array}{c}
\text{UNIT } C_1 \{\neg \ell \} \cup C \\
\hline
C \end{array}
\begin{array}{l}
\{ C_1 \in F \\
C_1 \setminus C = \{ \ell \} \}
\end{array}
\begin{array}{c}
\text{CONFLICT } C_1 \\
\hline
C \end{array}
\begin{array}{l}
\{ C_1 \in F \\
C_1 \setminus C = \emptyset \}
\end{array}
\]

**Other interpretation**: two DPLL steps (read backward) where
$C$ is assumed FALSE while $F$ is assumed TRUE

On **CONFLICT**, DPLL backtracks: it **learns** some clause $C$ from $F$

- it proves “$F \Rightarrow C$” from

\[
\begin{array}{c}
\text{UNIT } C_{n-1} \\
\hline
C_n \end{array}
\begin{array}{c}
\text{CONFLICT } C_n \\
\hline
\end{array}
\]

- and then adds $C$ in $F$

CDCL **“minimizes”** $C$ before learning!
**UNSAT certificates from learned clauses**

learned clause = auxiliary lemma found by CDCL solver

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

- **LRAT format** from **DRAT-trim**
  for each learned clause $C$,
  a list of *previously* learned clauses (or axioms) $C_1, \ldots, C_n$
  representing the following proof of $C_1 \land \ldots \land C_n \Rightarrow C$

\[
\begin{align*}
  &C_1 \\
  \vdots \\
  &C_{n-1} \\
  &C_n \\
  \hline
  &C \quad \text{(Backward Resolution Chain learning $C$)}
\end{align*}
\]

**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 \( cc[F] \) of “consequences” of \( F \).

\[
\text{Record } cc(s : \text{model} \rightarrow \text{Prop}) : \text{Type} := \\
\{ \text{rep} : \text{clause}; \text{rep_sat} : \forall m, s m \rightarrow [\text{rep}] m \}.
\]

Then, we define emptiness test:

\[
\text{assertEmpty } \{s\} : cc s \rightarrow \neg(\forall m, \neg(s m)).
\]

Learning a clause (from a BRC) is defined by

\[
\text{learn} : \forall \{s\}, \text{list}(cc s) \rightarrow \text{clause} \rightarrow \neg(cc s)
\]

implemented such that (for “performance” only)

if \( l \) is a valid BRC of clause \( c \)
then \( \text{learn } l \ c \) returns \( c' \) such that \( \text{rep } c' = c \).
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 oracle

**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 clause"
  - type \( \text{lcf } A \) = 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**

```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
Polymorphic LCF style vs. Certificates

**Advantage of Polymorphic LCF style**
Oracles computes directly “correct-by-construction” results through an API certified from Coq (where type abstraction comes from polymorphism)
Since 2017, VPL fully rewritten in Polymorphic LCF style. **Benefits**:

- 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?