What is the Foreign Function Interface of the Coq Programming Language?

July 2018

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

Introduction to the quest of a sound FFI for Coq

Foreign Functions as Non-Deterministic Functions

Coq “Theorems for Free” about Polymorphic Foreign Functions

Applications to Certify UNSAT Answers from Oracles
Several kinds of Foreign Functions in Coq programs

1. Extend Coq with I/O, exceptions & threads. 
   Already possible with http://coq.io/ 
   ⇒ provides support to reason about I/O effects in Coq. 
   ⇒ efficiently extracted to OCaml.

2. Introduce external oracles in complex computations 
   e.g. “register allocation” of CompCert (see next slides). 
   No reasoning on their effects, only on returned values!

   What do we need: oracles + axioms on oracles? 
   or, something more specific to each external system?

This talk = kind 2: “foreign functions” as “untrusted oracles”
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! Because at runtime, \((\text{oracle one})\) returns true whereas \((\text{oracle (S O)})\) 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 in success of COMPCERT [Leroy et al., 2006-2018]

Success story of software certification in Coq:
the safest C optimizing compiler [Yang et al., 2011]
commercially available since 2015
compile critical software for airplanes & power plants.

Uses “untrusted oracles” invoked from the certified code.
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 certified in Coq

Benefits of untrusted oracles
simplicity + efficiency + modularity
Issues of oracles in CompCert

Oracles are declared as pure functions.
Example of register allocation:

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

Not a real issue because
their purity is not used in the compiler proof!

This talk proposes an approach to ensure such a claim...
The quest proposed in 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!

This 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.
Contents

Introduction to the quest of a sound FFI for Coq

Foreign Functions as Non-Deterministic Functions

Coq “Theorems for Free” about Polymorphic Foreign Functions

Applications to Certify UNSAT Answers from Oracles
May-Return Monads [Fouilhé, Boulmé’14]

Axiomatize \( \mathbb{P}(A) \) as type “??A”

\[ \text{to represent “impure computations of type } A \]

and “\( a \in k \)” as proposition “\( k \leadsto a \)”

\[ \text{read “computation } k \text{ may return value } a \]

with formal type \( \leadsto_A : ??A \to A \to \text{Prop} \)

Formal operators and axioms

\[ \text{ret}_A : A \to ??A \quad (\text{interpretable as identity relation}) \]

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

\[ \gg=_{A,B} : ??A \to (A \to ??B) \to ??B \quad (\text{interpretable as the image of a predicate by a relation}) \]

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

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

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 et. al, 2013-2018]
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

```coq
Inductive empty : Type := Absurd : 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

```coq
let rec loop : empty = Absurd 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

Axion phys_eq \( \forall \{A\}, A \times A \rightarrow \text{?? bool} \).

Axion phys_eq_true \( \forall A (x \ y: A), \text{phys_eq}(x,y) \rightarrow \text{true} \rightarrow x = y \).

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 \( \forall (n: \text{nat}), \text{pred} \ n = n \rightarrow n = 0 \).

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

\[
\begin{align*}
& \text{nat} \rightarrow \text{bool} \quad (* \text{extracted as nat} \rightarrow \text{bool} \quad *) \\
& \text{nat} \rightarrow ??\{ \text{n: nat} \mid \text{n} \leq 10\} \quad (*) \quad \text{nat} \rightarrow \text{nat} \quad (*) \\
& \text{nat} \rightarrow ??(\text{nat} \rightarrow \text{nat}) \quad (*) \quad \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat}) \quad (*)
\end{align*}
\]

Conjecture the following types are permissive

\[
\begin{align*}
& \text{nat} \rightarrow ??(\text{nat} \rightarrow ?? \text{nat}) \quad (*) \quad \text{nat} \rightarrow (\text{nat} \rightarrow \text{nat}) \quad (*) \\
& \{ \text{n: nat} \mid \text{n} \leq 10\} \rightarrow ?? \text{nat} \quad (*) \quad \text{nat} \rightarrow \text{nat} \quad (*) \\
& (\text{nat} \rightarrow ?? \text{nat}) \rightarrow ?? \text{nat} \quad (*) \quad (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \quad (*) \\
& (\text{nat} \rightarrow \text{nat}) \rightarrow ?? \text{nat} \quad (*) \quad (\text{nat} \rightarrow \text{nat}) \rightarrow \text{nat} \quad (*) \\
& \forall \ A, \ A * A \rightarrow ??(\text{list} \ A) \quad (*) \quad 'a * 'a \rightarrow ('a \text{ list}) \quad (*)
\end{align*}
\]
Contents

Introduction to the quest of a sound FFI for Coq

Foreign Functions as Non-Deterministic Functions

Coq “Theorems for Free” about Polymorphic Foreign Functions

Applications to Certify UNSAT Answers from Oracles
A first “Theorem for Free” in Coq

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 :=
  (pid { y | y = x } x) >>= (fun z ⇒ ret (proj1_sig z)).

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

At extraction, we get “let cpid x = pid x”.

Coq “Theorems for Free” about Polymorphic Foreign Functions
Permissiveness of Polymorphism ⇒ Parametric Invariance

Permissiveness of
“∀ A, (A→A→A) → ??(A → ??(list A))” implies that
any safe OCAML

foo: ('a -> 'a -> 'a) -> 'a -> ('a list)
preserves any invariant (like 7N) attached to type variable 'a.

Example: “(foo (+) 7)” can only return lists of 7N.

A property of polymorphism sometimes called “unary parametricity”
or “parametricity over unary logical relations”
I prefer “parametric invariance”.

NB “theorems for free” from the type of polymorphic oracles!
Parametric Invariance for ML

- Comes *intuitively* from the type-erasure semantics: types are removed from runtime code (hence polymorphic functions must uniformly treat polymorphic values).
- Even hard to *formally define*: What are “invariants” about a higher-order reference (which can thus refer to itself)?
- Has been proved for a variant of system F with references by [Birkedal’11] (from the works of [Ahmed’02] and [Appel’07]).
- Requires some restrictions on polymorphic references parametric invariance is unsound on function calls creating some alias on an effective argument.

Example on type “int ref -> 'a ref -> 'a”
let f x y = ( x:=0; !y )

**Unsound Parametric Reasoning** on “f x x” (returning 0).
⇒ forbids to “import” polymorphic references in Coq???
Contents

Introduction to the quest of a sound FFI for Coq

Foreign Functions as Non-Deterministic Functions

Coq “Theorems for Free” about Polymorphic Foreign Functions

Applications to Certify UNSAT Answers from Oracles
Certifying UNSAT Answers from Oracles

Examples UNSAT on Boolean CNF or in linear arithmetic; no valid register allocation; etc...

Usually reduced to check some certificate (e.g. a resolution proof) from the oracle.

Alternatively might be done with Polymorphic LCF style: Oracles computes directly “correct-by-construction” results through an API certified from Coq where type abstraction comes from polymorphism.
Examples

- 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] or [Boulmé, Maréchal, preprint’17].

- **In this talk**: a tiny UNSAT prover on Boolean CNF

On the top of state-of-the-art CDCL SAT solvers + drat-trim

Based on verification of “Backward Resolution Chains”
(introduced as “Restricted RUP” by [Cruz-Filipe et al, 2016])
(work in progress with Thomas Vandendorpe)
Specification of the Refutation Prover

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

**Literal** \(\ell \triangleq x \text{ or } \neg x\).

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

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

\[
\text{unsat: list clause} \rightarrow \text{ bool}.
\]

**Lemma** \(\text{unsat_correct: } \forall F, (\text{unsat } F) \Rightarrow \text{true} \rightarrow \forall m, \neg[F]m.\)
Background on Backward Resolution

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

\[
\text{Axiom} \quad \frac{C \in F}{C} \quad \text{Resol} \quad \frac{\{\ell\} \cup C_1'}{C_1' \cup C_2' \cup C} \quad \frac{\{-\ell\} \cup C_2'}{C_1' \cup C_2'}
\]

**Rule** Resol equivalently split in two rules for **backward checking**

\[
\text{BckRsl} \quad \frac{\{\ell\} \cup C_1'}{C_2'} \quad \frac{\{-\ell\} \cup C}{C_1', C_2'} \quad \text{Trivial} \quad \frac{C_2'}{C} \quad \frac{C_1'}{C_2'}
\]

equivalently rewritten in

\[
\text{BckRsl} \quad \frac{C_1}{C} \quad \frac{\{-\ell\} \cup C}{C_1 \setminus C = \{\ell\}} \quad \text{Trivial} \quad \frac{C_1}{C} \quad \frac{C_1 \setminus C = \emptyset}{C_2'}
\]
Resolution Chains & Conflict-Driven Clause Learning DPLL

A **Backward Resolution Chain** (BRC) w.r.t a list of axioms \( F \)

\( = \) specialization of \( \text{BckRsl} \) and \( \text{Trivial} \) when \( C_1 \in F \)

\[
\begin{align*}
\text{UNIT} & \quad C_1 \quad \text{\{\neg \ell\} \cup C} \\
\text{C} & \quad \{ C_1 \in F \} \\
\text{C_1 \setminus C} & = \{ \ell \}
\end{align*}
\]

**Conflict**

\[
\begin{align*}
\text{C_1} & \quad \{ C_1 \in F \} \\
\text{C_1 \setminus C} & = \emptyset
\end{align*}
\]

**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

\[
\begin{align*}
\text{UNIT} & \quad C_{n-1} \\
\text{Conflict} & \quad C_n \\
\text{UNIT} & \quad C_1
\end{align*}
\]

- ... and then adds \( C \) in \( F \)

CDCL **minimizes** \( C \) before learning!
UNSAT Certificates from Learned Clauses

- UNSAT answer when clause $\emptyset$ is learned

- UNSAT certificates for CDCL in DRUP format
  $\vdash$ a sequence of learned clauses until $\emptyset$
  (We also support RAT clauses: out the scope of this talk)

- The \texttt{DRAT-trim} tool of [Heule et al, 2013-2017] outputs
  a backward resolution chain $[C_1; \ldots; C_n]$ for each learned
  clause $C$ (LRAT format).
Learning Clauses in Coq from Backward Resolution Chains

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

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

Then, we define emptiness test:

```
Definition isEmpty: \forall \{s\}, cc s \rightarrow \text{boolean} := ...
Lemma isEmpty_correct:
\forall s (c: cc s), isEmpty c=\text{true} \rightarrow \forall m, \neg(s \ m).
```

Learning a clause (from a BRC) is defined by

```
learn: \forall\{s\}, \text{list}(cc s) \rightarrow \text{clause} \rightarrow \text{option}(cc s)
```

such that (learn l c) returns

- (Some c’) with (\text{rep c’})=c on a correct BRC.
- None otherwise.
Toward “Logical Consequence Factories” (LCF)

**Idea** our oracle (≈ a LRAT parser) computes directly “certified learned clauses” through a certified API (called a LCF).
⇒ No need of an explicit “proof object”!

**For the following benefits**

- Backward Resolution Chains are verified “on-the-fly”, in the oracle (much easier to debug)
- very low memory footprint: deletion of “learned clauses” in memory directly & only managed by the oracle.
- very simple & small Coq code
Polymorphic LCF Style Oracle

- Data-abstraction is provided by polymorphism! Type $A$ is abstract type of “learned clause” here, $lcf = \text{abstraction of certified clause learning}$
- In input, each clause both given as a concrete value of $\text{clause}$ and an abstract “axiom” of type $A$.
- On an UNSAT input, the oracle returns some $\text{learned clause}$ (built from inputs and $lcf$ operations) and we only check its emptiness.

**Definition**

\[
\text{lcf } A := (\text{list } A) \rightarrow \text{clause} \rightarrow \text{option } A.
\]

**Axiom**

\[
\text{oracle: } \forall \{A\}, (\text{lcf } A)\times \text{list}(\text{clause}\times A) \rightarrow \text{!!(option } A\text{).}
\]
Using the Polymorphic Oracle in Coq

Implementation of unsat

Definition mkInput (f: list clause):
    lcf(cc[f]) * list(clause*(cc[f]))
:= ...

Definition unsat f :=
    oracle (mkInput f) >>= (fun o ⇒
        ret (match o with
            | Some c ⇒ isEmpty c
            | None ⇒ false end)).

Good results from our first experiments on some “large” examples (from SAT-competition 2017)
Verifying Backward Resolution Chains with certified code from Coq is faster than the corresponding SAT-solver run...
(Partial) Conclusion

- Study of “Foreign Functions” in Coq
  → new proof paradigms, combining Coq and other tools

- I propose to combine Coq and OCaml typecheckers to get
  “Theorems for free!” almost for free!

- Only need to understand the meta-theory of this proposal
  Is there any interested type-theorist in the room?