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

Sylvain Boulmé

Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France
{sylvain.boulme}@univ-grenoble-alpes.fr

The CompCert certified compiler [Leroy, 2009] is a success story of software verification in Coq (see [Yang et al., 2011, Bedin França et al., 2012, Kästner et al., 2018]). This success partly comes from the use of untrusted oracles invoked from the certified code. For example, register allocation in compilers – being related to a graph coloring problem – is a NP-complete problem: finding a valid allocation is difficult, while checking the validity of an allocation is easy. In CompCert, finding the allocation is done by an *oracle*, i.e. an untrusted OCaml function; and, only the *checker* of the allocation is programmed and certified correct in Coq [Rideau and Leroy, 2010]. Generally speaking, introducing such an oracle has the following benefits: first, this avoids to program and prove a difficult algorithm in Coq; second, this offers the opportunity to use (or even reuse) an efficient imperative implementation as the oracle; at last, this makes the software more modular. Indeed, the checker is actually certified for a family of oracles: the oracle can still be improved, without requiring to reprove the checker.

In some certified software like certified SAT solvers, oracles are invoked *before* certified code which only checks their outputs [Cruz-Filipe et al., 2017]. This is not the case in CompCert: oracles are directly invoked in the middle of certified transformations of the input. Hence, CompCert uses a standard FFI (*Foreign Function Interface*) of the Coq programming language, in order to invoke external code from certified code. However, there is no formal justification that using this FFI is sound. Below, I will illustrate some pitfalls of this FFI and propose how to overcome them. Moreover, I will conjecture that by using an adequate FFI, we can derive "*theorems for free*" *a la* [Wadler, 1989] in Coq from the OCaml type of polymorphic oracles, and thus discharge a part of the certification on the OCaml typechecker. However, my proposal raises more issues than it solves: in other words, it opens a new topic of research. **I propose to discuss these ideas in a talk of the Coq Workshop 2018.**

Unsoundness of the Standard FFI. The register allocation of CompCert is declared in Coq by[1]

```
Parameter regalloc: RTL.function → res LTL.function.
```

Here, "`Parameter`" is synonymous of "`Axiom`" and "`res`" is quite similar to the "`option`" type transformer. Some Coq directive in CompCert instructs Coq extraction [Letouzey, 2008] to replace this "`regalloc`" axiom by a function of the same name from the `Regalloc.ml` OCaml module. While very common, this approach is potentially unsound.

Let us consider the Coq example on the right hand-side. It first defines a constant `one` as the Peano's natural number representing 1. Then, it declares an axiom `test` replaced at extraction by a function `oracle`. At last, a lemma `congr` is proved, using the fact that `test` is a function. But, implementing `oracle` by "`let oracle x = (x == one)`" in OCaml makes the lemma `congr` false at runtime.

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

Axiom test: nat → bool.
Extract Constant test ⇒ "oracle".

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

Indeed, `(oracle one)` returns `true` whereas `(oracle (S O))` returns `false`, because `==` tests the equality between *pointers*. Hence, the Coq axiom is unsound w.r.t this implementation. A similar unsoundness is obtained if `oracle` uses a reference in order to return `true` at the first call, and `false` otherwise.

This unsoundness comes fundamentally from the fact that a Coq function f satisfies $\forall x, (f\,x) = (f\,x)$. But, an OCaml function may not satisfy this property. Actually, CompCert is probably free from such a bug, because its Coq proof does probably not depend on this property of `regalloc`.

[1] See `https://github.com/AbsInt/CompCert/blob/master/backend/Allocation.v`

Toward a formalized FFI. [Fouilhé and Boulmé, 2014] propose to avoid this unsoundness by axiomatizing external OCaml functions using a notion of non-deterministic computations. For example, if the result of `test` is declared to be non-deterministic, then the property `congr` is no more provable. For a given type A, type $??A$ represents the type of non-deterministic computations returning values of type A: it can be interpreted as $\mathcal{P}(A)$. Formally, the type transformer "$??$." is axiomatized as a monad that provides a *may-return* relation $\rightsquigarrow_A : ??A \to A \to \text{Prop}$. Intuitively, when "$k : ??A$" is seen as "$k \in \mathcal{P}(A)$", then "$k \rightsquigarrow a$" means that "$a \in k$". At extraction, $??A$ is extracted like A, and its binding operator is efficiently extracted as an OCaml let-in. See [Fouilhé and Boulmé, 2014] for more details.

For example, replacing the `test` axiom by "`Axiom test: nat → ??bool`" avoids the above unsoundness w.r.t the OCaml `oracle`. The lemma `congr` can still be expressed as below, but it is no longer provable – because it is not satisfied when interpreting $??A$ as $\mathcal{P}(A)$.

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

Of course, this approach does not suffice to avoid all pitfalls of axiomatizing oracle types in Coq. In the following, a Coq type T is said *permissive* iff any "*safe*" OCaml function[2] compatible with the extraction of T can be soundly axiomatized in Coq with type T.

Typically, `nat→bool` is not permissive while I conjecture that `nat→??bool` is permissive. But, `nat→??{n:nat | n ≤ 10}` is not. And, `nat→??(nat→nat)` neither. Indeed, for these two last examples, the Coq axiom assumes a *postcondition* that the OCaml typechecker can not ensure. On the contrary, I conjecture that `nat→??(nat→??nat)` and `(nat→??nat)→??nat` are permissive. And also `{n | n ≤ 10}→??nat`. On this last example, the Coq axiom requires a *precondition* that OCaml typechecker can safely ignore. Actually, a similar phenomenon happens with `(nat→nat)→??nat`.

These examples illustrate that it would be useful to *formalize* such a notion of permissive Coq types in order to extend the correctness theorem of Coq extraction for (some) *open* Coq terms.[3] Such a work could also result in a new Coq directive "**External**" that declares an axiom and checks the permissivity of its type. Actually, as briefly introduced in the next paragrah, studying the class of *permissive* Coq types is both challenging and an opportunity to introduce new proof paradigms in Coq+OCaml.

Toward "*Theorems for Free*" about Polymorphic Oracles. Let us conjecture that the Coq type "`∀ A, A→??A`" is permissive. We can then formally prove that for all "safe" OCaml implementation `f` of type `'a -> 'a`, when `f` returns *normally* some output, this ouput equals the input. We say below that such a `f` is a *pseudo-identity*. Our formal proof mimicks a "theorem for free" *a la* [Wadler, 1989]: we derive a non-trivial theorem about `f` only from its polymorphic type. More exactly, assuming an oracle `f` of type "`∀ A, A→??A`", we build below a Coq function `cpid` of the same type and whose extraction is "`let cpid x = f x`", and which is proved to be a pseudo-identity. In the Coq source, for a type `B` and a value `x:B`, `(cpid x)` invokes `f` on the type `{y|y=x}`, and returns the first projection of its result. Here, operators `>>=` and `ret` are respectively the bind and unit operators of the may-return monad.

```
Axiom f: ∀ A, A → ?? A.
Program Definition cpid {B} (x:B): ?? B :=
  (f {y|y=x} x) >>= (fun z ⇒ ret (proj1_sig z)).
Lemma cpid_correct A (x y:A): (cpid x)⇝y → y=x.
```

This illustrates that the permissivity of such polymorphic types requires a parametricity theorem on the OCaml type-system. This theorem is very similar to the one of an imperative extension of System F, which has been progressively established by [Ahmed et al., 2002, Appel et al., 2007, Birkedal et al., 2011]. See [Boulmé and Maréchal, 2017] for details. This last preprint actually describes how such "theorems for free" avoid the need of the *certificate* format introduced in [Fouilhé and Boulmé, 2014] for our Verified Polyhedra Library. Hence, replacing certificate generating oracles by polymorphic oracles greatly simplifies both the OCaml oracles and the Coq code. Moreover, generic loops and some exception handling operators are also proved in Coq by such "theorems for free".

[2] In a first approximation, a "safe" OCaml function could be defined as a well-typed function without calls to external constants like `Obj.magic`. But this definition is probably both too restrictive in practice, and too permissive in theory...

[3] Currently, extraction is only proved correct for closed terms!

References

[Ahmed et al., 2002] Ahmed, A. J., Appel, A. W., and Virga, R. (2002). A stratified semantics of general references embeddable in higher-order logic. In *Symposium on Logic in Computer Science (LICS)*, page 75. IEEE.

[Appel et al., 2007] Appel, A. W., Melliès, P.-A., Richards, C. D., and Vouillon, J. (2007). A very modal model of a modern, major, general type system. In *Principles of Programming Languages (POPL)*, pages 109–122. ACM Press.

[Bedin França et al., 2012] Bedin França, R., Blazy, S., Favre-Felix, D., Leroy, X., Pantel, M., and Souyris, J. (2012). Formally verified optimizing compilation in ACG-based flight control software. In *ERTS2*.

[Birkedal et al., 2011] Birkedal, L., Reus, B., Schwinghammer, J., Støvring, K., Thamsborg, J., and Yang, H. (2011). Step-indexed kripke models over recursive worlds. In *Principles of Programming Languages (POPL)*, pages 119–132. ACM Press.

[Boulmé and Maréchal, 2017] Boulmé, S. and Maréchal, A. (2017). Toward Certification for Free! preprint.

[Cruz-Filipe et al., 2017] Cruz-Filipe, L., Heule, M. J. H., Hunt, W. A., Kaufmann, M., and Schneider-Kamp, P. (2017). Efficient certified RAT verification. In *CADE*, volume 10395 of *LNCS*, pages 220–236. Springer.

[Fouilhé and Boulmé, 2014] Fouilhé, A. and Boulmé, S. (2014). A certifying frontend for (sub)polyhedral abstract domains. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8471 of *LNCS*, pages 200–215. Springer.

[Kästner et al., 2018] Kästner, D., Barrho, J., Wünsche, U., Schlickling, M., Schommer, B., Schmidt, M., Ferdinand, C., Leroy, X., and Blazy, S. (2018). CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems*, pages 1–9, Toulouse, France. 3AF, SEE, SIE.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7).

[Letouzey, 2008] Letouzey, P. (2008). Extraction in Coq: An overview. In *Computability in Europe (CiE)*, volume 5028 of *LNCS*, pages 359–369. Springer.

[Rideau and Leroy, 2010] Rideau, S. and Leroy, X. (2010). Validating register allocation and spilling. In *Compiler Construction (CC 2010)*, volume 6011 of *LNCS*, pages 224–243. Springer.

[Wadler, 1989] Wadler, P. (1989). Theorems for free! In *Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359. ACM Press.

[Yang et al., 2011] Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI))*, pages 283–294. ACM Press.