Formally Verified Defensive Programming (FVDP)

efficient COQ-verified computations from untrusted ML oracles

Habilitation (HDR) of Sylvain Boulmé — Sep 27, 2021

Reviewers

Andrew W. Appel	Professor at Princeton University
Sandrine Blazy	Professeur à l'Université de Rennes 1
Greg Morrisett	Professor, Dean of Cornell Tech

Examiners

Hugo Herbelin	Directeur de Recherche à l'Inria
Xavier Leroy	Professeur au Collège de France
Jean-François Monin	Professeur à l'UGA

Université Grenoble Alpes — Verimag — Grenoble INP Ensimag

thesis & slides on `http://www-verimag.imag.fr/~boulme/hdr.html`

Contents

Scientific proposal

Challenge

Formal verification of software
that produces/verifies safety-critical systems :
compilers, analyzers & verifiers.

Example : prevent compilers from introducing critical bugs
with a formal (mechanized) proof of the compiler correctness.

How ? I propose to
bind OCAML (the programming language)
to COQ (the interactive theorem prover)

$\gg=$

and to apply Formally Verified Defensive Programming

CompCert, the 1st formally proved C compiler

Major success of software verification

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

Developed since 2005 by Leroy & collaborators (Blazy, etc)
More than 100Kloc of Coq & OCaml

Lesson *"If the formal-verification problem is too complex, then change it for a simpler one !"*

- Drop *noncritical* requirements, e.g. *termination* : only consider *partial correctness*.
- Introduce *untrusted oracles*...

Formally Verified **Defensive** Programming (FVDP)

Idea : complex computations by **efficient** functions, called **oracles**,
with an *untrusted* & *hidden* implem. for the formal proof
⇒ only a **defensive test** of their result is formally verified

Example of CompCert register allocator [Rideau,Leroy'10]

- *finding* an *efficient* allocation is difficult
- *checking* the *correctness* of a given allocation is easier

⇒ Register allocation provided by an OCaml imperative oracle
Only a checker is programmed and proved in Coq.

Typical applications NP-hard problems,
complex fixpoints (e.g. memoization or dynamic programming)...

Benefits of FVDP

simplicity + efficiency + **modularity**

OCaml oracles need to appear in Coq as "*foreign functions*"...

The issue of foreign OCaml functions in Coq

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

Example of Coq proof

```
Axiom oracle: nat → bool.  Extract Constant oracle ⇒ "foo".
Lemma oracle_pure: ∀ n, oracle n = oracle n.
  congruence.
Qed.
```

Example of OCaml implementation

```
let foo =
  let b = ref false in
  fun (_:nat) -> (b:=not !b; !b)
```

INCORRECT `oracle_pure` is wrong for two "successive" calls

OCaml "functions" are not functions in the math sense.
Rather view them as "relations", ie "nondeterministic functions"

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

Oracles in CompCert : a soundness issue ?

CompCert 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 formal proof !

I propose to formally ensure such a claim [VSTTE'14],
by modeling OCaml foreign functions in Coq as
"nondeterministic functions"
Successfully applied in the VPL (Verified Polyhedra Library)
[Boulmé, Fouilhé, Maréchal, Monniaux, Périn, etc, 2013-2018]

A Coq model of OCaml pointer equality (==)

OCaml "==" cannot be modeled as a "pure" Coq function.
However, a trusted "==" seems useful for FVDP.

Example of **Instruction scheduling** in CompCert
Very elegant **FVDP design** of [Tristan,Leroy@POPL'08]
based on **symbolic execution** (of [King'76]).
But, still not in CompCert because of *checkers inefficiency*!

I have shown how to *fix this efficiency issue*
with the help of another **FVDP design** where

a "nondeterministic" model of == in Coq
suffices to verify the answers of **hash-consing oracles**.

See [Six,Boulmé,Monniaux@OOSPLA'20] & [Six-Phd'21].

A "good" FVDP design is the key !

The FVDP-design **trade-off** (for a given application) :

Simplicity of formal verification

versus

Reduced overhead of "defensive tests"

FVDP designs in my HDR thesis for

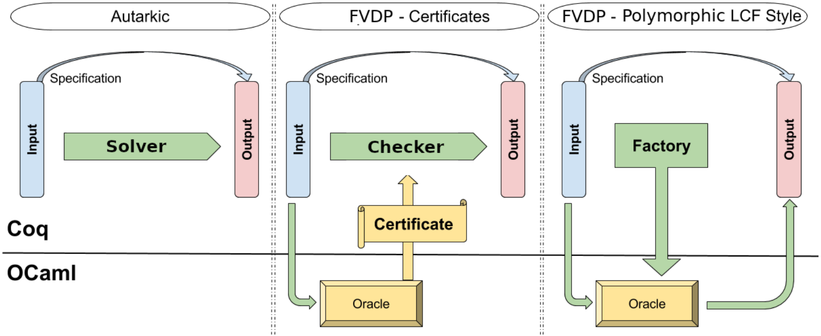
- instruction scheduling in CompCert (optimizing compiler)
- abstract domain of polyhedra (VPL) for the Verasco static analyzer (on the top of CompCert)
- Boolean SAT-solving (SatAnsCert)

Central Issue

How "oracles" may help "defensive tests" without being too hindered ?

Polymorphic LCF Style (= Shallow Embeddings of Certificates)

Design patterns for a solver that bounds the set of solutions



Inspired by old LCF prover, **I propose** "Polymorphic LCF Style" as a "lightweight certificate handling".

See [Boulmé,Maréchal,Monniaux,Périn,Yu@SYNASC'2018]

Feedback from the Verified Polyhedra Library

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

- Code size at the interface COQ/OCAML divided by 2 :
  shallow versus *deep* embedding (of certificates).
- Oracles debugging much easier :
  interleaved executions of untrusted and certified computations.

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].

FVDP by Data-Refinement

Two sources of "bureaucratic reasoning" in large FVDP proofs

1. optimized data-representations (wrt more naive ones)
2. impure computations (wrt pure ones)

Data-refinement helps in reducing both of them, *simultaneously* !

Examples

- Data-refinement for FVDP of Symbolic Execution [Six,Boulmé,Monniaux@OOSPLA'20]
- Data-refinement for FVDP of Abstract Interpretation [Boulmé,Maréchal@JAR'19].

Contents

Features of my approach

- Almost any OCaml function embeddable into Coq.

(e.g. mutable data-structures with aliasing in Coq)

- No formal reasoning on *effects*, only on results :

foreign functions could have bugs, only their type is ensured.
⇒ Considered as nondeterministic.
e.g. for I/O reasoning, use FreeSpec or InteractionTrees instead.

- OCaml polymorphism provides "*theorems-for-free*" about
  - (some) invariant preservations by mutable data-structures
  - arbitrary recursion operators (needs a small defensive test)
  - exception-handling

- Exceptionally : additional axioms on results (e.g. pointer equality)

In this case, the foreign function must be trusted !

Introduction to my IMPURE library

Impure computation := COQ code embedding OCAML code.

Based on *may-return monads* of [Fouilhé,Boulmé@VSTTE'14]

- **Axiomatize** (in COQ) "$A \to \mathrm{Prop}$" as type "$??A$"
  to represent "*impure computations of type A*"
  with "$(k\ a)$" as proposition "$k \leadsto a$"
  with formal type $\leadsto_A : ??A \to A \to \mathrm{Prop}$
  read "*computation k may return value a*"
  and composition operators (on next slide)
- "$??A$" extracted like "A".

For any "`Axiom oracle:nat→??bool`", determinism is **unprovable**

```
∀ n b1 b2, (oracle n)⇝b1 →(oracle n)⇝b2 →b1=b2.
```

because, it reduces to contradiction "`∀ (b1 b2:bool), b1=b2`" when interpreting proposition "`(oracle n)⇝b`" as "`True`".

May-return monads operators (and axioms)

Currently, only 3 operators with 2 additional axioms :

- $\texttt{RET}_A : A \to ??A$
  with axiom $(\texttt{RET}\ a_1) \leadsto a_2 \ \to\ a_1 = a_2$
  formally interpretable as the identity relation

  extracted as the identity function

- $\gg\!=_{A,B} : ??A \to (A \to ??B) \to ??B$
  with axiom $(k_1 \gg\!= k_2) \leadsto b \ \to\ \exists a, k_1 \leadsto a \wedge (k_2\ a) \leadsto b$
  formally interpretable as the image of a predicate by a relation

  "$k_1 \gg\!= k_2$" actually written in COQ "$\texttt{DO}\ a \leftsquigarrow k_1;;\ \ k_2\, a$ "
  extracted to OCAML as "**let** a=... **in** ... "

- $\mathrm{mk\_annot}_A : \forall (k : ??A),\ ??\{\, a \mid k \leadsto a \}$
  without axiom
  formally interpretable as the trivially "True" relation

  extracted as the identity function

Declaration of oracles : a Coq user wish

I would wish some "`Import Constant`" like

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

that acts like

```
Axiom ident: permissive_type.
Extract Constant ident ⇒ "safe_ocaml_value".
```

but with **additional typechecking** ensuring that

any "`safe_ocaml_value`" compatible with
the OCaml extraction of "`permissive_type`"
satisfies Coq theorems proved from the axiom.

} soundness of *permissive* type

Should reject "`Import Constant ident: nat → bool :=...`"
because "`nat → bool`" is not *permissive*,
but accept "`nat → ??bool`" as *permissive*.

Permissivity

Currently, only an informal notion (i.e. "human expertise").
Hence, the Coq type of OCaml oracles is part of the TCB.

Counter-Examples Coq types which are not permissive

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

Examples Coq types which are permissive
(i.e. they are conjectured to be sound Coq types for oracles)

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

More detailed explanation in my HDR thesis.

Embedding ML references into Coq

```
Record cref{A}:={set: A→??unit; get: unit→??A}.
Axiom make_cref: ∀ {A}, A → ?? cref A.
```

where "`∀ {A}, A → ?? cref A`" (permissive) is considered sound 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 hist = ref [x] in {
    set = (fun y -> hist := y::!hist);
    get = (fun () -> nth !hist (Random.int (length !hist))) }
```

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

Contents

Soundness of permissivity ⇒ unary parametricity of OCaml

MetaThm Assuming that permissivity of (∀ A, A→??A) is sound, any *safe* OCaml "pid:'a -> 'a" satisfies

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

Proof

1) a Coq "wrapper" of pid, called cpid is a pseudo-identity

```
Axiom pid: ∀ {A}, A→??A.

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

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

2) at extraction : `let cpid x = (let z = pid x in z)`

This meta-theorem is a *"theorem for free"* for [Wadler'89]
ie a proof by *"(unary) parametricity of polymorphism"*
for [Reynolds'83]

Unary parametricity for imperative higher-order languages

- Parametricity comes from the type-erasure semantics : polymorphic values must be handled uniformly.
- Has been proved for a variant of system F with references by [Ahmed, Dreyer, Birkedal, Rossberg@POPL+LICS'09] (from seminal works of Appel & co started around 2000).
- **Open Conjecture** for "Coq + ??. + OCaml"

Unary parametricity : ML type → 2nd-order invariant

Example

Deriving a while-loop for Coq (in partial correctness)
from a 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 DIRECTLY computing "while" results

Declaration of the oracle in Coq

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

- A ↦ loop invariant i.e. type of "reachable states"
- B ↦ post-condition i.e. type of "final states"

Implem. in OCaml

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

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).
```

Generalization to impure recursion (e.g. with memoization)

Wrap into a **certified** recursion operator, any oracle declared as

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

But, formal correctness of **recursive functions** requires a **relation** `R` between inputs and outputs.
How to encode a *binary* relation into the "*unary postcondition*" `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 such as

```
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 this bug...
...and aborts the recursive computation...
...by exception raising (as shown after next slide)

Any `fixp` implementation is supported !

Standard fixpoint (pointer equality 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 (defensive check of `Hashtbl.find` equality)

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

See my HDR thesis for details.

Verification “for free” of higher-order impure operators

- (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.
```

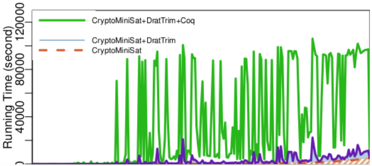
- **Polymorphic LCF Style**
  Design pattern for oracles (example next slide)

Certifying UNSAT proofs of Boolean SAT-solvers

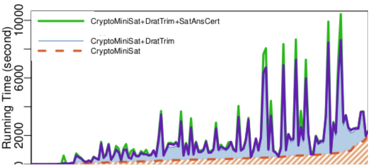
```
Record resolLCF C := { binary_resolution: C → C → ?? C;
                       get_id: C → clause_id }.
Axiom refute: ∀ {C}, (resolLCF C)*(list C) → ?? C.
```

where `(resolLCF C)` is the type of a "Logical Consequences Factory" by binary resolution on clauses of type `C`

Application (with T. Vandendorpe)
Redesign of the Coq-verified checker of [Cruz-Filipe+@CADE'17] into SatAnsCert



Certificate (Abstract Syntax)



Polymorphic LCF style

Contents

Projects with results covered by my HDR thesis

- VPL [2012-2018]
  D. Monniaux and M. Périn (Verimag)
  with their Phd students A. Fouilhé and A. Maréchal (Verimag)
  + French ANR Verasco [2012-2016]
  Gallium & Abstraction & Toccata (Inria Paris) ;
  Celtique (Irisa Rennes).

- SatAnsCert [June-July 2018]
  T. Vandendorpe (UGA Bachelor internship)

- CompCert for Kalray VLIW [2018-2021]
  D. Monniaux (Verimag) and B. Dupont de Dinechin (Kalray)
  with our Phd student C. Six (grant CIFRE Kalray-Verimag)
  + Xavier Leroy (Inria - Collège de France).

Projects uncovered by my HDR thesis

- CompCert for a secure RiscV with CFI protections [2018-2020]
  M-L. Potet and D. Monniaux (Verimag)
  with our post-doc P. Torrini (grant of IRT Nanoelec - Pulse)
  + O. Savry, T. Hiscock (CEA LETI)
- CompCert Verimag-Kalray student internships [06/19-08/21]
  (co-supervized with D. Monniaux and C. Six)
  T. Vandendorpe, L. Chelles, J. Fasse, L. Chaloyard, P. Goutagny and N. Nardino.

- CompCert for in-order embedded RiscV cores [10/20-09/23]
  F. Pétrot (UGA-TIMA) and D. Monniaux (Verimag)
  with our Phd student L. Gourdin (grant of labex Persyval UGA)
  + D. Demange (Irisa Rennes)
- CompCert front-end for a subset of Rust/MIR [10/21-09/24]
  D. Monniaux (Verimag) and F. Wagner (UGA-LIG)
  with our Phd student D. Carvalho (grant of IRT Nanoelec - Pulse)
  + TODO ?