Defensive Certification in Coq with ML Type-Safe Oracles

Sylvain Boulmé
Univ. Grenoble-Alpes, VERIMAG, F-38000 Grenoble, France
sylvain.boulme@imag.fr

Abstract. Initially promoted by CompCert, the embedding of untrusted OCaml code into extracted code from Coq – through a skeptical approach – significantly simplifies Coq developments of formally proved software. However, as illustrated by various examples of this paper, such an embedding could be unsound. This paper conjectures sufficient conditions to ensure soundness. And, it illustrates the power of these conditions on the ultra-lightweight certification of an UNSAT-prover: its Coq sources (less than 250 lines) have been developed in around one man-day.

1 Introduction

Designing a certified software in a formal proof assistant, like Coq, involves usually to choose between two approaches: the autarkic one[1] where the software is fully described in the proof assistant, and the skeptical one[2] where some part of the software is unproved and untrusted but produces a certificate, i.e. a trace of its computations, such that this certificate allows to build a certified result. Of course, skeptical approach can only establish partial correctness properties like “if the software gives a result then the result has property $P$”. However, even if autarkic approach in CoQ requires to prove software termination, this proof uses an abstract model of computations. Hence, at runtime, the software may still abort because of a lack of memory, or may not end before years. Hence, practical guarantees offered by both approaches are quite similar. I now briefly recall interests and limits of skeptical approach.

Skeptical approach aims to scale down the proof effort by reducing the whole certification of the software to the certification of a small part of it: a checker. It induces a modular design, where this certified checker may be independently linked to different external oracles, as soon as these oracles are able to produce certificates in the expected format. The oracles may use untrusted aggressive tricks and invoke code from a low-level programming language like C. The formal proof of the checker ensures that it builds correct results from certificates output by oracles. Of course, recomputing the results from the certificates induces some redundant computations. But, this defensive style may be largely compensated by the gain of generating these certificates from a very optimized oracle.

* This work was partially supported by ANR project VERASCO (INS 2011).
For instance, skeptical approach is suited for verifying the satisfiability of a propositional formula: a certificate corresponds here to an assignment of the Boolean variables in the formula. However, for verifying the unsatisfiability, finding a “good” notion of certificates is less obvious: resolution proofs provide a notion of certificates, but it is not easy to design an efficient solver that generates them. Moreover, these certificates may become so huge that they require some adhoc “management”.

In short, skeptical approach seems only suited in a context where “efficient searching” is much more complex than “verifying”. This criterion depends on the state-of-the-art. But, many NP or co-NP hard problems could satisfy it.

The “Certificate Verification” Pattern The most common skeptical approach is called here “certificate verification” and became very famous with its instance as “Proof Carrying Code” [3]. It is designed as two functions:

\[
\text{oracle: input } \rightarrow \text{ certificate} \\
\text{checker: input } \rightarrow \text{ certificate } \rightarrow \text{ output}
\]

Only \text{checker} is certified with a lemma of the following form, for a given predicate \(P : \text{input } \rightarrow \text{ output } \rightarrow \text{ Prop}\)

\[\text{checker_correctness : } \forall i c, P i (\text{checker i c}).\]

In the context of Coq, this pattern is typically used in reflexive tactics [4,5,6,7,8]. From a given input \(i\), such a tactic (written in Ocaml) invokes \text{oracle} to get a certificate \(c\). Then, it builds a Coq term “\text{checker_correctness i c}”. At last, Coq kernel checks this term and evaluates its type. Hence, \text{oracle} can be an arbitrary Ocaml function without almost any unsoundness risk \(^1\).

In order to run \text{checker} in a standalone program, outside of Coq kernel, it can be itself extracted to Ocaml. Actually, extraction also allows to link \text{checker} to \text{oracle} in a much simpler way: there is no need to handle Coq abstract syntax trees. However, as detailed in this paper, the soundness of this link of extracted code to untrusted code is currently not fully ensured by automated typechecking.

The “Defensive Certification” Pattern This extraction-oriented approach has been slightly generalized in CompCert [9] by providing a certified API (Application Programming Interface) where the oracle and certificates are not visible. In other words, the oracle is an untrusted backend invoked by a certified frontend which is only run after extraction. The Coq certified API is like

\[
\text{frontend: input } \rightarrow \text{ output} \\
\text{frontend_correctness : } \forall i, P i (\text{frontend i}).
\]

This more general pattern is typically used in certified libraries which are basic bricks to build large certified software [10,11,12]. In this case, the frontend and the backend may actually correspond to some collections of functions (e.g.

\(^1\) The only risk seems that in the case of a reflexive tactic running inside Coq, the kernel state may be silently altered through a memory corruption, see Section 3.
modules). On the contrary to certificate verification, oracles are embedded inside certified code: the backend is directly invoked by the frontend through an internal API. This allows for complex interactions between the certified part and the untrusted part, as detailed below. I call this pattern “defensive certification” because it may be understood as “defensive programming that certifies”.

However, this approach still raises delicate correctness issues. Technically, defensive certification requires to declare backend functions with their type in Coq. This Coq API of the backend must be designed with care. In particular, developers must carefully check that backend functions in OCAML are sound with respect to their Coq type. Here, they may be helped by the fact that OCAML and Coq are both typed lambda-calculi. But this similarity may also lead to dangerous illusions. In particular, an OCAML function is not a Coq function. In Coq, a function \( f \) is implicitly pure: hence \( \forall x, f x = f x \) is provable. On the contrary, a OCAML function may use an implicit state such that, two distinct calls on the same input give different results. Moreover, using OCAML physical equality, it may also distinguish between input values that are logically equal.

**Defensive Certification in the VPL (Verimag Polyhedra Library)** The certified API of the VPL frontend (see [13,14,15]), consists in a datatype of polyhedra with certified operations on this datatype\(^2\). Actually, the frontend and the backend both introduce their concrete representations of polyhedra. Typically, whereas in the frontend concrete representation works with Coq rationals (where integers are represented as lists of bits), the backend representation uses more efficient rationals based on GMP, an optimized multiple precision arithmetic library written in C. This allows backend operations to involve an efficient linear programming solver (written in OCAML). The design of the VPL takes care to avoid rebuilding whole polyhedra at each operation.

Hence, a polyhedron of this certified API is implemented as a pair associating a certified representation to an untrusted representation of type Bcknd.poly – an abstract datatype for the frontend but concrete for the backend. When the frontend invokes some operation of the backend, the certificate of type cert returned by this operation allows the frontend to update its own concrete certified representation of polyhedra. For example, the operations provided by the backend have a type like the one below where affCstr is some input of the operation (here a new face “added” to the polyhedra)

\[
\text{Bcknd.guard}: \text{Bcknd.poly*affCstr} \rightarrow (\text{Bcknd.poly option})*\text{cert}
\]

Thus, the certified representation and the untrusted one remain implicitly synchronized. If this synchronization is broken because of an unexpected bug, this can only result in a loss of precision: this can not break correctness.

VPL illustrates a more complex interaction between untrusted oracles and certified code than in the classic certificate verification pattern. VPL avoids unsoundness by declaring in Coq its backend functions as computations of a “may-return” monad (recalled here in Section 4.1).

\(^2\) This datatype is an abstract domain of Verasco certified static analyzer [16].
Overview of the Paper

This paper investigates the paradigm of defensive certification one step further. First, Sections 2 and 3 detail examples of backends which are unsound with their Coq API; Section 4 conjectures sufficient conditions at which a Coq-certified program may soundly invoke an untrusted Ocaml oracle; Section 5 illustrates the power of these conditions for the certification of an UNSAT prover. Here, the defensive certification uses a LCF proof construction style [17] in order to have very lightweight certificates and to delegate most of the “certificate management” to the untrusted backend. This style exploits ML polymorphic typechecking to ensure that the untrusted oracle preserves data-invariants of the frontend. The sources of examples in Section 2 and Section 5 are available online [18].

2 The hdiscr Anti-Example

This section gives a tutorial on how to embed untrusted code into Coq certified code at extraction. However, its example does not really correspond to defensive certification, since it does not involve certificate checking. Actually, it introduces unsoundness issues.

Given a polymorphic hashing function hash, let us build a certified function able to discriminate between values of the same type. First, we declare hash function in Coq: for any value of type A, it returns an unbounded binary integer of Coq. The Extract directive below instructs Coq extraction to replace hash calls by Oracle.hash, an Ocaml function defined later.

\[
\text{Axiom hash: } \forall \{A\}, A \rightarrow Z. \\
\text{Extract Inlined Constant hash } \Rightarrow \text{"Oracle.hash".}
\]

Hence, we define a polymorphic discriminating function hdiscr such that hdiscr x y tests whether hash-codes of x and y differ. In this case, tactic congruence proves that x and y are distincts. Otherwise, we do not know.

\[
\text{Program Definition hdiscr } \{A\} (x y:A): \{ b | b=\text{true } \rightarrow x\neq y \} :\rightleftharpoons \text{if } \text{Z.eq_dec (hash x)} (\text{hash y)} \text{then false else true.} \\
\text{Solve Obligations with congruence.}
\]

Axiom hash is consistent: it is realized in Coq by any constant function returning the same hash-code for all inputs. Obviously, such hashing functions have no interest, because they lead hdiscr to return always false.

I rather build an implementation of Oracle.hash in Ocaml. I first define a function z_of_int of type int -> coq_Z where coq_Z is the Ocaml type resulting from extraction of Coq type Z. The straightforward implementation of z_of_int is omitted here. Then, I define Oracle.hash by invoking Hashtbl.hash of type 'a -> int from the standard Ocaml library:

\[
\text{let hash: 'a } \rightarrow \text{coq}_Z = \text{fun } x \rightarrow z\_\text{of}\_\text{int (Hashtbl.hash x)}
\]

In order to see hdiscr in action, I introduce type ord of enumerable ordinals.

\[
\text{Inductive ord } := \text{Zero } \rightarrow \text{ord } | \text{Succ } \rightarrow \text{ord } | \text{Lim } (\text{nat } \rightarrow \text{ord }) \rightarrow \text{ord}
\]
At runtime, \texttt{hdiscr} discriminates between some values of type \texttt{ord}: hence, “\texttt{hdiscr (Succ Zero) (Lim (fun _ ⇒ Zero))}” returns \texttt{true}. Moreover, I prove in Coq the following property: given two equal values \(x\) and \(y\), then “\texttt{hdiscr x y}” returns \texttt{false}. Given \texttt{test_neg} defined below, we observe that “\texttt{test_neg (Succ Zero) (Succ Zero)}” returns \texttt{false} as expected.

<table>
<thead>
<tr>
<th>Program Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{test_neg {A} x (y:A</td>
</tr>
</tbody>
</table>

\textbf{Unsoundness of \texttt{Oracle.hash}.} Actually, \texttt{test_neg} provides soundness tests for \texttt{Oracle.hash}. Whereas “\texttt{test_neg e e}” is proved to return \texttt{false}, if it returns \texttt{true} for some Coq expression \(e\), then \texttt{Oracle.hash} implementation is unsound w.r.t. its axiomatization in CoQ. Such an unsound result is given by

\texttt{test_neg (Lim (fun _ ⇒ Zero)) (Lim (fun _ ⇒ Zero))}

Indeed, when its input value contains a closure, \texttt{Hashtbl.hash} returns a hash-code computed from the memory address of the function invoked in the closure. Therefore, \texttt{Hashtbl.hash} distinguishes functions of distinct addresses which are logically equals for CoQ.

\textbf{Unsoundnesses of Oracles Axiomatized as Pure Functions.} I now illustrate that even a monomorphic \texttt{hdiscr} of type \texttt{nat → bool} involving only elementary types in axioms raises unsoundness issues. I assume a constant \texttt{one} defined in the frontend as \((\mathrm{S} \ 0)\). Below, I give two alternative functions \texttt{hash: nat → int} as replacements of \texttt{Hashtbl.hash} in \texttt{Oracle.hash}. These two functions are unsound, because they lead “\texttt{test_neg (S \ 0) one}” to return \texttt{true} whereas it is proved to be \texttt{false}.

1. I use operator \texttt{==} to compare physical addresses.
   \texttt{let hash x = if x==one then 1 else 0}

2. Function \texttt{hash} below returns the number of times it has been called (thus, it ignores its argument).
   \texttt{let r = ref 0}
   \texttt{let hash _ = (let v = !r in r:=v+1 ; v)}

The first example illustrates that an OCAML function may discriminate more values than logical equality by accessing to their low-level representation. The second one illustrates that side-effects may also make results of OCAML functions depend on the computation history. In practice, such bad behaviors are not so obvious, because they may appear as implicit effects of nested function calls.

\section{Pitfalls of Linking Certified Code after Coq Extraction}

On example of Section 2, some people could prefer to turn \texttt{hash} into a parameter of \texttt{hdiscr} instead of an axiom. However, all previous unsound examples would
still happen in instantiating \texttt{hdiscr} from “hand-written” OCAML code. This illustrates that, whatever is the mean, linking extracted COQ code to untrusted OCAML code may lead to serious pitfalls. Let us make this more precise.

Extraction [19] generates OCAML code from COQ code. It ensures that the generated functions behave like their corresponding sources: on the same input, they produce the same output. It roughly proceeds by removing all the proof-related information from the COQ code which OCAML type system is not powerful enough to represent, and then by translating the resulting typed lambda-terms into OCAML syntax.

Now, let us consider an axiom (or variable) \( f \) of type \( T \) in COQ which is replaced at extraction by an untrusted OCAML value \( \overline{f} \). Let \( \overline{T} \) be the OCAML type of \( \overline{f} \) given by OCAML typechecker. Type \( \overline{\text{E}}(T) \) – the extracted version of type \( T \) – must match \( T \): the OCAML compiler will report an error otherwise. However, this is not sufficient to ensure soundness. Here is a list of pitfalls.

**Implicit Axioms.** A COQ inductive type \( T \) (e.g. \( \{ x : \text{nat} \mid x < 5 \} \)) may be extracted into a strictly larger extracted type \( \overline{\text{E}}(T) \) (e.g. \( \text{nat} \)). This introduces an implicit requirement on \( \overline{f} \) (i.e. its results are lower than 5) that OCAML typechecker cannot ensure. Hence, such a COQ type should be avoided in a backend API.

**Cyclic Values of Inductive Types.** Let us consider the COQ function below that for a given \( n \), returns a \( p \) distinct of \( n \). The correctness of this implementation reduces to the trivial goal that \( \forall p : \text{nat}, (S p) \not= p \).

\[
\text{Program Definition } \text{find\_other} (n : \text{nat}) : \{ p : \text{nat} \mid p \not= n \} := \text{match } n \text{ with } 0 \Rightarrow (S 0) \mid (S p) \Rightarrow p \text{ end.}
\]

When extracted in OCAML and applied on \texttt{loop} where \texttt{let rec loop = S loop}, we observe that \texttt{find\_other loop \Rightarrow loop} returns \texttt{true}. This contradicts the specification of \texttt{find\_others} if we consider that physical equality implies logical equality. Yet, such a property seems very desirable, even if it is only observable from OCAML. Otherwise, certified extracted code may have counter-intuitive specifications. This issue comes from OCAML cyclic values (e.g. rational trees) built with “\texttt{let rec}”. They do not exist in COQ inductive types : they make type \texttt{nat} of OCAML strictly larger than type \texttt{nat} of COQ.

**Implicit Purity Axiom.** Semantics of \( \rightarrow \) are different in COQ and in OCAML. In COQ, a function \( f \) is implicitly pure: hence \( \forall x, f x = f x \) is provable. In other words, axiomatizing \( f \) with COQ type \( A \rightarrow B \) introduces an implicit functional requirement: \( f \) is observationally pure w.r.t. Coq equality. As illustrated by Section 2, checking this property is non-trivial. Even invoking predefined functions like \( \texttt{==} \) is dangerous: the backend may silently discriminate between equal values from COQ point-of-view.

\[3\] Some COQ users define a cyclic value of type \texttt{nat} at extraction to turn bounded recursion of functions waiting for “fuel” into unbounded recursion. As discussed above, such a hack seems a bit unsafe.
Memory Corruption. A major interest of defensive certification comes from the ability to build a backend using arbitrary libraries like the GMP library — written in C — through an OCAML frontend. A bug in such low-level libraries may corrupt arbitrary memory locations. However, it seems very unlikely that such a bug breaks soundness silently. Hence, we may admit as an hypothesis that the backend can not corrupt frontend results. Of course, this hypothesis is only sensible if the backend does not use malicious code that intends to corrupt silently frontend results.

Type Violations. When oracles shortcut the typechecker using `Obj.magic` (which casts any value to any type) — or similar constructs based on declaration of external constants — they can also forge values that do not exist in the Coq logic. Let us consider the following example in Coq which invokes a backend my_id of OCAML polymorphic type `'a -> 'a`.

```coq
Axiom my_id : ∀ {A}, A -> A.
Program Definition test_my_id : { b | b = true } := my_id (A := { b | b = true }) true.
```

The following implementation of my_id makes silently test_my_id be evaluated as the Boolean false whereas it is proved true in Coq.

```coq
let my_id x = if (Obj.magic x) then (Obj.magic false) else x
```

This implementation of my_id is unsafe. For instance, evaluating expression `((my_id my_id) 2)` of type `int` leads to a segmentation fault. Hence, we may admit that no sane programmer would write such a code (even unintentionally). However, such type violations may happen when using external low-level code.

4 Toward a Sound Approach to Defensive Certification

In the next, I admit absence of memory corruption as an hypothesis. All other pitfalls listed in Section 3 comes from cases where OCAML types contain values that do not exist in the corresponding Coq type from the API. In order to ensure soundness of oracles w.r.t. their Coq API, I propose the following approach:

- Axiomatize OCAML \( \mathcal{F} \) function of type \( \hat{\mathcal{E}}(A) \rightarrow \hat{\mathcal{E}}(B) \) as its “may-return” relation of type \( A \rightarrow B \rightarrow \text{Prop} \). More precisely, \( \mathcal{F} \) is axiomatized as a function \( A \rightarrow \mathcal{B} \) where \( \mathcal{B} \) is an abstract monad of predicates \( B \rightarrow \text{Prop} \). Typically, with this restriction, a “corrected” version of hdiscr example can not exist: such examples, relying on the fact that oracles are pure functions, can not be expressed. See Section 4.1 and Section 4.2.
- In backend APIs, only accept Coq inductive types that correspond to ML datatypes. It means that an admissible Coq type \( T \) must correspond to a superset of the set of values in \( \hat{\mathcal{E}}(T) \). See Section 4.2.
- Section 4.3 restricts OCAML typechecking on casts and on cyclic values in order to ensure that execution of extracted code preserves its Coq type.
4.1 Abstracting “Impure Computations” in May-Return Monads

Let us recall the definition of may-return monads from [14].

**Definition 1 (May-Return Monad).** For any type $A$, type $?A$ represents impure computations returning values of type $A$. Type transformer “?” is equipped with a monad [20] providing a may-return relation

- Operator $\gg_{A:B}: ?A \to (A \to ?B) \to ?B$ encodes OCAML “let $x = k_1 \text{in } k_2$” as “$k_1 \gg_{A:B} \lambda x, k_2$”.
- Operator $\varepsilon_A : A \to ?A$ lifts a pure computation as an impure one.
- Relation $\equiv_A : ?A \to ?A$ is a congruence (w.r.t. $\gg_{A:B}$) which represents equivalence of semantics between impure computations. Moreover, operator $\gg_{A:B}$ is associative and admits $\varepsilon$ as neutral element (w.r.t. $\equiv_{A:B}$).
- Relation $\Rightarrow_A : ?A \to A \to \text{Prop}$, where “$k \Rightarrow_A a$” means that “$k$ may return $a$”. This relation must be compatible with $\equiv_A$ and satisfies the axioms

$$\varepsilon a_1 \Rightarrow a_2 \Rightarrow a_1 = a_2 \quad k_1 \gg k_2 \Rightarrow b \Rightarrow \exists a, k_1 \Rightarrow a \land k_2 a \Rightarrow b$$

Given a type $S$ of global stores, the computation of a well-typed ML expression $e$ — returning a value of type $A$ — is represented by a big-steps relation $k$ of type $S \to A \to S \to \text{Prop}$. In this very abstract model, diverging or exception-raising computations correspond to empty relation. Moreover, when the typing of the store is unsound with $e$, this lead to a runtime error also encoded as empty. These big-steps relations are equipped with the may-return monad below: $\varepsilon$ and $\gg_{A:B}$ are associated to their usual big-steps semantics; observational equivalence is the extensional equivalence between big-steps computations; may-return relation expresses that there exists an execution of $k$ returning $a$.

$$?A \triangleq S \to A \to S \to \text{Prop} \quad \varepsilon a \triangleq \lambda s_0, \lambda a', \lambda s_1, a = a' \land s_0 = s_1$$

$$k_1 \gg k_2 \triangleq \lambda s_0, \lambda b, \lambda s_1, \exists a, \exists s_1, k_1 s_0 a s_1 \land k_2 a s_1 b s_2$$

$$k_1 \equiv k_2 \triangleq \forall s_0, \forall a, \forall s_1, k_1 s_0 a s_1 \leftrightarrow k_2 s_0 a s_1 \quad k \Rightarrow a \triangleq \exists s_0, \exists s_1, k s_0 a s_1$$

May-return monad allows to express that OCAML oracles are impure functions that actually encode relations. It is instantiated at extraction by providing the identity implementation below:

$$?A \triangleq A \quad \varepsilon a \triangleq a \quad k_1 \gg k_2 \triangleq k_2 k_1$$

Of course, this implementation remains hidden for our Coq proofs: they are thus valid for any instance of a may-return monad, like big-steps model above. A Coq tactic is provided in [14] to simplify reasoning on may-return relation.

4.2 Coq Types Considered as Admissible for Oracles

This section gives conditions on Coq types to consider them as admissible in axioms of backend API. The goal of these restrictions is to ensure that an admissible Coq type $T$ corresponds to a superset of the set of values in $\hat{E}(T)$ —
where $\mathcal{E}(T)$ is interpreted as an OCaml type, under restrictions of Section 4.3. Typically, dependent types are forbidden in admissible types: only ML polymorphism is admitted (i.e. universally quantified polymorphism in prenex form). Non-informative types (i.e. containing propositions in Prop) are also forbidden.

Moreover, in order to ensure that $A \rightarrow ?B$ is a superset of $\mathcal{E}(A) \rightarrow \mathcal{E}(B)$, it is sufficient to ensure that $\mathcal{E}(A)$ is a superset of $A$ (in this case $A$ is said “to be admissible for inputs”) and $B$ is a superset of $\mathcal{E}(B)$ (in this case $B$ is said “to be admissible for outputs”). Hence, rules for admissible types are split in two kinds: one for “inputs” and one for “outputs”. Typically, “?" are mandatory at the right of a “$\rightarrow$" on outputs, but not on inputs (since Coq typechecker will then ensure that the frontend instantiates such an input by a pure function).

Inductive types appearing in outputs must themselves be restricted. First, constructors have to be considered as special functions (formally, they are not functions in OCaml). Hence, the return type (in Coq) of a constructor should not be behind a “?". But, its arguments have to be considered as outputs. For instance, consider OCaml “type foo = Bar of (nat -> nat)”. In order to use foo in output of an oracle, it must be declared in Coq as:

```
Inductive foo := Bar : ( nat -> ?nat ) -> foo
```

Let us remark that this forbids to have type ord of Section 2 in output of an oracle: constructor Lim can not be both well-typed in Coq and have its higher-order arguments as potentially impure. In this paper, I do not consider the use of coinductive types in oracle axiomatizations (I forbid them).

At last, Coq backend APIs are allowed to declare external types which can then be used both as inputs or outputs. For the sake of simplicity, I only authorize monomorphic external types (otherwise, we get troubles with mutable values).

```
Axiom ref_int : Type.
Extract Inlined Constant ref_int ⇒ "(int ref)".
```

The logical equality on such types is included in OCaml $\mathcal{E}$.

### 4.3 Restrictions on OCaml Typechecking

Restrictions of Section 4.2 prevent most pitfalls of Section 3, except for “Cyclic Values” and “Type Violations”. Let us remark that an inductively empty type like “type foo = X of foo” does not lead to an inconsistent Coq axiom when used in outputs, because $?foo$ corresponds to empty relation in the big-steps model. But foo may lead to an unsound oracle because it is actually inhabited in OCaml by cyclic values. In order to avoid such unsound oracles, I annotate each inductive type extracted from Coq (e.g. by a “coq_inductive” tag) and forbid cyclic values on such types.

Whereas type violations are dangerous, they are clearly useful to embed low-level code like GMP in OCaml. In the same spirit than above, I thus forbid the use of OCaml “external” on types containing “Coq inductive types”. More precisely, I also annotate – as “castable” – monomorphic types that are allowed
to be cast through an “external” directive. Hence, I check that OCaml inductives are never castable even through a type abstraction (a castable type abstraction should only have a castable implementation). Usual OCaml types like `int` are castable. Polymorphic types are generally forbidden in “external” directives (otherwise restrictions above are easily circumvented). We may only admit them inside the carefully checked implementation of standard library modules (e.g. for `Hashtbl.hash`). Using modules of the standard library like `Obj` that allows arbitrary type violations is forbidden.

4.4 Soundness Conjecture and its Applications

This paper conjectures that, in the absence of memory corruption, restrictions given in Section 4.2 and Section 4.3 are sufficient to ensure soundness of oracles w.r.t. their API axioms in Coq. Moreover, sources of the VPL [14,15] and of the UNSAT-prover described Section 5 [18] have been carefully reviewed to ensure that they satisfy these restrictions. Such a “manual” verification is not fully satisfying. But designing automated typecheckers of these restrictions is beyond the scope of this paper.

5 Ultra-Lightweight Certification of a Refutation Prover

As the archetype example of defensive certification, I detail the certification of a prover able to ensure unsatisfiability of a propositional logic formulas in CNF (Conjunctive Normal Form). This example is inspired from [6], where an untrusted DPLL procedure produces a resolution refutation proof instead of a simple “unsat” answer, which is then verified by a certified checker in Coq. In practice, such resolution proofs may be very large. In the certificate verification pattern of [6], the format of certificates has thus been designed with care. For instance, it allows to preserve sharing of subproofs, i.e. to express proofs as a directed acyclic graph.

Alternatively, I propose to combine defensive certification and LCF-style [17] by using an abstract data type for the resolution proof, such that the resolution tree is not explicitly built at runtime: its existence is only ensured by construction and by ML typechecking. More precisely, the only information kept at runtime on the resolution tree is the clause on its root. Here, the Coq proof (together with the ML typechecking of the backend) ensures that if the root of the resolution tree is empty clause then the input CNF is unsatisfiable.

Hence, my certificate format is very simple: a certificate is a single clause. The backend can only build such certificates through a certified API. The fact that the resolution tree shares some subproofs, simply corresponds to the fact that the backend avoids to recompute some clauses already computed. The memory management of certificates is thus completely delegated to the backend. As a result, the certified frontend is almost trivial. It relies on ML polymorphic typechecking to ensure that the backend can not forge unsound clauses.
This section is organized as follows. Paragraph 5.1 gives a very basic introduction to CDCL algorithm, a modern reformulation of DPLL, currently at the heart of state-of-the-art SAT-solvers. This paragraph actually explains why CDCL algorithm can naturally produce good refutation proofs on unsat CNF. Paragraph 5.2 presents the design of a certified unsat prover in Coq, assuming some oracle able to produce refutations proofs. At last, Paragraph 5.3 briefly reports the implementation of such an oracle.

5.1 Propositional Resolution and CDCL SAT-solving

A Boolean variable $x$ is a name and is encoded as a positive integer. A literal $\ell$ is either a variable $x$ or its negation $\neg x$. A clause $c$ is a finite disjunction of literals and is encoded as a set of literals. A $CNF$ $f$ is a finite conjunction of clauses and is encoded as a list of clauses. A model $m$ of $CNF$ $f$ is a mapping that assigns each variable to a Boolean such that "$f[m]$" is true – where "$f[m]$" is the Boolean value obtained by replacing in $f$ each variable $x$ by its value $m(x)$.

**Theorem 1 (Resolution Proof System).** $f$ is unsatisfiable (i.e. $f$ has no model) iff clause $\emptyset$ is derivable from the proof system defined by the 2 rules:

\[
\begin{align*}
  c & \in f & \text{Axiom} \\
  \ell \cup c_1 \cup \{\neg \ell\} & \cup c_2 & \text{Resolution}
\end{align*}
\]

Let us now examine how a SAT-solver may find that a $CNF$ $f$ is unsat. Actually, the solver tries to recursively build a model of $f$: if it fails, $f$ is unsat. As illustrated on Figure 1, the solver first assumes some literal $\neg 5$ in a given clause $c_1$. In this potential model, $c_1$ is thus trivially satisfied, like any other clause (e.g. $c_8$) containing $\neg 5$. Clauses containing its negation $5$ (e.g. $c_7$) are simplified by eliminating this literal. As there are still some unsolved clauses, the solver assumes a new literal $4$. Now, after simplifications, clause $c_2$ and $c_3$ become unitary: they contain a single unassigned literal. Hence, in the potential model, these literals are necessarily true: they are assigned accordingly. The propagation of these derived assignment leads to see $c_4$ and $c_5$ as satisfied, whereas $c_6$ is simplified as $\emptyset$ and is thus unsatisfiable. Such a clause – unsatisfiable in the current potential model – is said “in conflict”.

In presence of such a conflict, a simple DPLL algorithm would backtrack on the last assumption (here $4$) in order to propagate the dual assignment. A CDCL algorithm[21] firstly analyzes the conflict in order to learn more conflicting clauses, obtained by resolutions on the initial clauses (i.e. not the simplified ones). Hence, these learned clauses are logical consequences of $f$ – they are satisfied by any model of $f$. This analyze is illustrated on Figure 1, where each conflict found by the algorithm leads to a resolution tree on the right hand-side of this conflict: the corresponding learned clauses are nodes on this resolution tree. The first conflict produces two learned clauses: “1, $\neg 4$” and “$\neg 4$”. Actually, learned clauses are built by a resolution chain starting with the clause in conflict ($c_6$), and involving each clause from which a literal has been assigned since
Input CNF: \(-5, 1\) \(-1, -4\) \(2, -4\) \(-1, -3\) \(2, -3\) \(1, -2\) \(3, 4, 5\) \(3, 4, -5\)

\[
\begin{align*}
\text{try assume } -5 & \text{ from } c_1 \\
\text{try assume } 4 & \text{ from } c_7 \\
c_8: \text{SAT} \\
c_7: 3, 4
\end{align*}
\]

\[
\begin{align*}
c_2: & -1 \text{ propag.} \\
c_3: & 2 \text{ propag.} \\
c_4, c_5: & \text{SAT} \\
c_6: & \emptyset
\end{align*}
\]

or propag. \(-4\) from \(c_9\)

\[
\begin{align*}
c_2, c_3: & \text{SAT} \\
c_7: & 3 \text{ propag.} \\
c_4: & -1 \text{ propag.} \\
c_5: & 2 \text{ propag.} \\
c_6: & \emptyset
\end{align*}
\]

or propag. \(5\) from \(c_{11}\), \(-3\) from \(c_{10}\), \(-4\) from \(c_9\)

\[
\begin{align*}
c_2, c_3, c_4, c_5, c_7: & \text{SAT} \\
c_8: & \emptyset
\end{align*}
\]

Fig. 1. An Instance of CDCL Algorithm on an Unsat CNF & its Resolution Proof

the last assumption, in reverse order of the propagation chain. By construction, these learned clauses are themselves in conflict with the current potential model. Learned clauses allow two optimizations w.r.t. original DPLL. First, backtrack is skipped on a variable absent of the last learned clause, because assigning this variable would not solve the conflict: such a backtrack skipping is called a backjump. Second, during backtracking, the original CNF is extended with learned clauses which avoids redundant computations leading to similar conflicts. Such an extension is sound because learned clauses are consequences of the CNF.

In order to illustrate this second optimization, let us consider the remaining of Figure 1. Actually, from the two learned clauses of the first conflict, our algorithm memorizes only the last one as \(c_9\) (because it implies the other). While backtracking on \(4\), it finds a new conflict and learns two new clauses \(c_{10}\) and \(c_{11}\). The backtracking on \(5\) leads in one propagation chain to a conflict, thanks to the learned clauses. Moreover, at top level, on an unsat CNF, the CDCL algorithm has learned clause \(\emptyset\).

In conclusion, a CDCL algorithm is naturally able to produce a refutation proof by resolution when it finds an unsat CNF. Moreover, CDCL algorithms naturally produce reduced resolution proofs, where subproofs are shared through learned clauses. However, in practice, CDCL implementations do not use resolution proofs to analyze conflicts: they use a more efficient structure called implication graph[21]. Hence, replaying these computations by using resolutions instead may induce a serious overhead. I let this issue for future works.
5.2 Defensive Certification in Coq of a Resolution Prover

This section describes defensive certification of an oracle producing resolution proofs for unsat CNF. In our CoQ specification on Figure 2, a variable is coded as a CoQ positive (roughly, a list of binary digits); a literal is a pair of a Boolean and a positive; a syntactic clause (resp. CNF) is a list of literals (resp. syntactic clauses). The definition of predicate sats on Figure 2 is sufficient to understand the specification of unsat prover on Figure 4.

For efficiency reasons, the frontend computes resolutions with another type cclause for clauses, such that a clause is coded by two sets of positives: one for the set of positive literals, one for the set of negative ones. Such a set is indeed efficiently encoded by CoQ standard library as a radix tree. In the following, my
notations on cclause make its inhabitants as sets of literals. I also use $J^K_m$ for its satisfiability predicate. Hence, resolutions are computed by a total function resol defined below:

**Lemma 1 (Resolution Function).** Let resol defined by
\[
\text{resol } \ell \ c_1 \ c_2 \triangleq (c_1 \setminus \ell) \cup (c_2 \setminus \ell).
\]
Then, for all $m$,
\[
[J^K_{c_1} m \land J^K_{c_2} m \implies J^K_{\text{resol } \ell \ c_1 \ c_2} m].
\]

The API of our external oracle, named solver, is provided on Figure 3. Its purpose is to abstract “resol” implementation inside a polymorphic logger, playing the role of an abstract datatype for building “certified clauses” by resolution. Hence, for any type $A$ of “certified clause”, for any “logger” implementation log, and for any CNF $f$ given as a list(clause*A), solver is expected to return

- either (Some $c$), where $c$ is an empty “certified clause” of type $A$, built from log and from $f$, thus showing that $f$ is unsatisfiable;
- or None, if solver has failed to prove that $f$ is unsatisfiable.  

Here, as the type $A$ of clauses is polymorphic, ML typechecking ensures that solver can not build clauses $c$ of type $A$ that do not derive from $f$ through function log.resol. In particular, it can not forge an unsound empty clause $c$. Let us note that this would not be the case with a type abstracted through the module mechanism. Indeed, in that case, solver would be able to return an empty clause computed from a previous call on a unsatisfiable CNF.

Actually, polymorphism is crucial for the Coq implementation of our unsat prover Figure 4. Indeed, parameter $A$ of solver is instantiated as (aClause mod) where mod represents the set of models of CNF $f$, and (aClause mod) represents cclause satisfied by any model of mod. In other words, parameter $A$ is instantiated as a dependent type expressing that clauses produced by resol are satisfied by models of $f$. In particular, when solver returns (Some $c$), then $c$ is also satisfied by models of $f$. At last, unsat has only to check emptiness of $c$ using the trivial test defined below:

**Lemma 2 (Emptiness Test).** Let isEmpty defined by
\[
\text{isEmpty } c \triangleq c = \text{bool } \emptyset.
\]
Then, for all $m$,
\[
\text{isEmpty } c \text{ true } \implies \neg [c] m.
\]

### 5.3 Experiments with a Naive Oracle

In order to check the feasibility of generating these LCF-style certificates, I have implemented a naive oracle satisfying API of Figure 3. This oracle also allows to experiment with the running time overhead of certification.

Like modern DPLL implementations, my naive oracle performs a conflict analysis providing backjumps and some strategies of clause learning. It also features an internal data-structure for the set of clauses using watched literals

---

4 In order to keep this example as simple as possible, the frontend does not certify here “sat” answers of the SAT-solver. This is however a straightforward extension.
Defensive Certification in Coq with ML Type-Safe Oracles

Indeed, this imperative data-structure allows for a lazy propagation in less than \( O(s) \) tests on literals between each conflict detection – where \( s \) is the sum of all clause sizes. This data-structure is necessary in presence of clause learning which may greatly increase the number of clauses involved in propagations. Unlike standard CDCL implementations, my strategy of clause learning is not based on an implication graph, and is thus very naive. Moreover, my oracle misses other standard features of CDCL implementations, like selection of new literal to assume according to “VSIDS” heuristic[22]. A brief description of the solver architecture is given in Appendix A (or [18]).

By tuning the learning strategy of my oracle for a certain kind of problem (i.e. “pigeon holes”), I can make it a competitive solver for this kind of problems. In this configuration, the overhead of certification is acceptable (below 10% w.r.t. to oracle running time when this latter is above 0.1 second). I have also checked that with another learning strategy and other kinds of problems (e.g. “Sudokus”), this overhead remains acceptable (below 30%). However, in this case, my oracle is not quite competitive w.r.t. standard SAT-solvers. See Appendix B (or [18]).

6 Conclusions and Future Works

This paper illustrates that defensive certification allows to develop a certified UNSAT-prover in Coq, by focusing most of the development effort on an untrusted oracle in OCaml. In this experiment, the certified frontend has been developed using only around 1 man-day – with 30 Coq lines for the specification and around 200 Coq lines for the implementation (proof scripts included). The development of the untrusted backend represents 800 OCaml lines in around 1.5 man-month. While this certified prover is efficient on some examples, its backend still misses many crucial optimizations to be competitive with standard SAT-solvers. Actually, SAT-solving community has recently developed efficient certificate formats for UNSAT-checking like DRAT format[24]. Instead of reimplementing a solver in OCaml, it would probably be more useful to certify a DRAT checker in OCaml using the same kind of Coq-certified frontend. Indeed, the frontend presented in Section 5.2 could be adapted to support extended resolution[25]. The LCF-style presented here could also be adapted also to VPL[14], with the benefit of both improving the frontend efficiency and simplifying generation of certificates in the backend. Of course, having automated typechecking of Section 4 restrictions would increase confidence in such developments.

References

A Brief Description of the Ocaml Oracle

```ocaml
type 'a pcl = { cert: 'a; ... }
type 'a global = { resol: 'a logger; ... }

let resolution: 'a global -> literal -> 'a pcl -> 'a pcl
  = env l c1 c2 ->
    let c = { cert = env.resol l c1.cert c2.cert; ... } in
    ...
    c

let rec search: 'a scnf -> 'a pcl
  = fun sf ->
    match propagate sf with
    | Unsat proof -> proof
    | NewAssigned literals ->
      let (l, sf1) = select_some_literal sf in
      assume l sf1;
      let proof1 = search sf1 in
      let proof2 =
        if (∼l ∈ proof1) then (assume ∼l sf;
          resolution sf.env ∼l proof1 (search sf))
        else proof1
        in analyze_conflict literals sf.env proof2

let solver: 'a logger * (clause * 'a) list -> 'a option
  = fun (log, l) ->
    try
      let env = { resol = log; ... } in
      let sf = init env l in
      let proof = search sf in
      Some proof.cert
    with
    | Sat -> None
```

The above OCAML pseudo-code sketches my naive OCAML implementation of solver specified at Figure 3. In this pseudo-code, type variable 'a is the type of certificate that corresponds to type A in the Coq API. The heart of my solver is the search backtracking function, which implements the algorithm described in Section 5.1.

Let me explain this pseudo-code from the beginning. My oracle analyzes conflicts using resolution, but for efficiency reasons, it uses its own representation for clauses instead of the frontend ones. Hence, type 'a pcl represents type of “proved clauses”: this type thus relates a clause in the representation of the oracle to a certificate of the frontend. Type 'a global represents a shared state
between recursive searches: it contains the “logger” given by the frontend from Figure 3; it also contains the watched-literal data-structure (used by propagation algorithm). Function \texttt{resolution} allows to derive a new “proved clause” by resolution: it invokes resolution of the frontend, and may add the learned clause in the watched-literal data-structure. Type \texttt{a scnf} extends \texttt{a global} with informations that are local to the current recursive call of search.

Function \texttt{search} either returns a proof that the set of clauses in input is unsatisfiable or raises exception \texttt{Sat} otherwise. It implements an algorithm very closed to the one illustrated on Figure 1. It starts to propagate unit clauses via \texttt{propagate}. This either detects a conflict (case \texttt{Unsat}) or assigns a list of literals (case \texttt{NewAssigned}). In this latter case, \texttt{select_some_literal} looks for unassigned literals. If there is none, it raises \texttt{Sat}. Otherwise, it chooses one, named \texttt{l}. Then, \texttt{search} tries to recursively find a conflict \texttt{proof1} by assuming \texttt{l}. If exception \texttt{Sat} has not been raised, and if \texttt{¬l} appears in the learned clause \texttt{proof1}, then assuming \texttt{¬l} is also tried recursively. At last, if \texttt{Sat} has still not been raised, we have find a conflict \texttt{proof2} under the assumption of the previous propagations. Function \texttt{analyze_conflict} learns new clauses from this conflict.

At last, \texttt{solver} first initializes the oracle data-structures from the input (through \texttt{init} function) and then runs \texttt{search}. If exception \texttt{Sat} is raised, then it returns \texttt{None}. Otherwise, some certificate is returned to the front-end, which corresponds to empty clause if there is no bug.

\section{Comparison with some Standard SAT-Solvers}

I have compared my oracle with various CDCL SAT-solvers: CLASP 2.1.4, MiniSAT 2-070721, PICOSAT 954, SAT4J 2.3.2, ZCHAFF 2007.3.12 and Z3 4.4.0. Among them, I have especially considered PICOSAT and ZCHAFF which provide an option to output a resolution proof in a file on UNSAT results. Moreover, I have also considered the \texttt{smtcoq} 1.2 Coq library [8] that provides – by extraction from Coq – a certified checker of resolution proofs produced by ZCHAFF [22]. All experiments have been run on a laptop under Ubuntu 14.04.

Clearly, my oracle is not competitive w.r.t. these SAT-solvers. Nevertheless, my experiments indicate that the overhead of my checker would remain acceptable if it were plugged onto a more competitive CDCL oracle. As explained in section 5.1, the number of resolutions involved in the checker is bounded by the number of propagation steps in the oracle. In my experiments, I have measured this overhead as the ratio of the checker running time to the oracle running time. And, I have considered two learning strategies for the oracle: \texttt{systematic learning} when all consequences of a conflict are learned, and \texttt{unit learning} when consequences reduced to a unit clause are learned. These two strategies represent two extreme cases for the solver.

Figure 5 compares my oracle for \texttt{systematic learning} strategy with ZCHAFF on Hn inputs. Such a Hn corresponds to the standard SAT query “to put \(n + 1\) pigeons into \(n\) holes, with at most one pigeon by hole” (which is unsatisfiable). This problem is a worst-case for DPLL procedures: the number of backtracks
(or resolutions) explodes exponentially. In “ZCHAFF” and “oracle” columns, running times are in seconds. Of course, this comparison is unfair for ZCHAFF since systematic learning is a very good strategy for Hn problems5, but explodes for other kinds of problems.

The interest of this comparison lies in “ratio” columns which represents the overhead of certifying the results. While, this overhead is given in seconds by “cert” column, “ratio” gives it in percents of the solver running time. In both “ratio” columns, the overhead of certification reduces while input size increases, even if the certification processes are different. Indeed, for ZCHAFF, “cert” column gives the overhead of generating the proof in a file (roughly, 20% of the overhead on “pigeon holes”) plus the overhead of checking this proof with smtcoq. Hence, this running time includes generating the file from ZCHAFF and then parsing it by smtcoq, which is not needed by my approach. However, in smtcoq, which uses an experimental extension of Coq, literals are represented with integers from the hardware, and clause are represented with imperative arrays. On the contrary, my checker uses only pure functional data-structures of standard Coq which are less efficient.

Figure 6 illustrates my experiments to compare my oracle for unit learning strategy with other solvers. Here, “Sn” inputs are based on Sudoku puzzles; “SAT” column indicates the expected result; “nv” the number of variable and “nc” the number of clauses. This figure presents the running times of CLASP, the best solver with ZCHAFF – among those I have tested – on this experiment. It presents the running times of SAT4J – a very popular solver running in Java virtual machine – showing that running times of my oracle are not ridiculous w.r.t. those of this solver. 6 It also presents the running times of PICOSAT, and its overhead when a trace (i.e. a resolution proof) is asked for UNSAT results: checking this trace is not included here in this overhead.

In this figure, on “pigeon holes”, my oracle – even with this bad learning strategy – remains competitive with all solvers except ZCHAFF. And, the checking ratio (in “%” column) is still low. But, as expected, it is higher than when the solver has a learning strategy optimized for such a problem. In conclusion, my experiments on “pigeon holes” indicate that the cost of my propagation algorithm is not too bad w.r.t. other solvers. Hence, on other kinds of problem, the checking overhead would remain acceptable even if the checker were plugged on solvers having less propagation steps than my oracle (typically, because of better strategies for selecting literals to assume and for learning clauses).

5 The subtree sharing in the resolution proof of Hn induced by systematic learning divides the number of resolutions by a factor from 200 for H9 up to $10^6$ for H13.

6 Due to measure imprecisions, the ratio between small times is very imprecise.

7 Running times of SAT4J given here includes at least 0.2s for launching Java virtual machine on my laptop. Moreover, according to http://www.sat4j.org/, “a SAT solver in Java is about 3.25 times slower than its counterpart in C++”. This comparison with SAT4J only illustrates that speed may not be the only criterion on which people select a given SAT-solver.
### Fig. 5. Systematic learning versus ZCHAFF/SMTCOQ on “pigeon holes”

<table>
<thead>
<tr>
<th>SAT</th>
<th>nv</th>
<th>nc</th>
<th>CLASP</th>
<th>SAT4J</th>
<th>PICO</th>
<th>trace</th>
<th>%</th>
<th>ZCHAFF</th>
<th>cert</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>S9.1</td>
<td>Y</td>
<td>729</td>
<td>7399</td>
<td>0</td>
<td>0.58</td>
<td>0</td>
<td>+0</td>
<td>0.01</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>S9.2</td>
<td>Y</td>
<td>729</td>
<td>7393</td>
<td>0.01</td>
<td>0.95</td>
<td>0.04</td>
<td>+0.01 +25</td>
<td>0.08</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>S9.1</td>
<td>N</td>
<td>729</td>
<td>7400</td>
<td>0.01</td>
<td>0.50</td>
<td>0</td>
<td>+0</td>
<td>0.01</td>
<td>+0.03 +300</td>
<td>+0</td>
</tr>
<tr>
<td>S9.2</td>
<td>N</td>
<td>729</td>
<td>7394</td>
<td>0.02</td>
<td>1.45</td>
<td>0.04</td>
<td>+0.01 +25</td>
<td>0.08</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>S16</td>
<td>N</td>
<td>4K</td>
<td>80K</td>
<td>0.04</td>
<td>1.17</td>
<td>0.04</td>
<td>+0.01 +25</td>
<td>0.08</td>
<td>+0</td>
<td>+0</td>
</tr>
<tr>
<td>H8</td>
<td>N</td>
<td>72</td>
<td>297</td>
<td>0.23</td>
<td>5.29</td>
<td>0.62</td>
<td>+0.15 +24</td>
<td>0.15</td>
<td>+0.04 +27</td>
<td></td>
</tr>
<tr>
<td>H9</td>
<td>N</td>
<td>90</td>
<td>415</td>
<td>2.02</td>
<td>37.79</td>
<td>9.29</td>
<td>+1.95 +21</td>
<td>0.84</td>
<td>+0.17 +20</td>
<td></td>
</tr>
<tr>
<td>H10</td>
<td>N</td>
<td>110</td>
<td>561</td>
<td>19.14</td>
<td>timeout</td>
<td>892</td>
<td>+422 +53</td>
<td>3.69</td>
<td>+0.57 +15</td>
<td></td>
</tr>
<tr>
<td>S9.1</td>
<td>N</td>
<td>72</td>
<td>297</td>
<td>0.23</td>
<td>5.29</td>
<td>0.62</td>
<td>+0.15 +24</td>
<td>0.15</td>
<td>+0.04 +27</td>
<td></td>
</tr>
<tr>
<td>S9.2</td>
<td>N</td>
<td>90</td>
<td>415</td>
<td>2.02</td>
<td>37.79</td>
<td>9.29</td>
<td>+1.95 +21</td>
<td>0.84</td>
<td>+0.17 +20</td>
<td></td>
</tr>
<tr>
<td>S16</td>
<td>N</td>
<td>110</td>
<td>561</td>
<td>19.14</td>
<td>timeout</td>
<td>892</td>
<td>+422 +53</td>
<td>3.69</td>
<td>+0.57 +15</td>
<td></td>
</tr>
</tbody>
</table>

### Fig. 6. Unit learning versus others SAT-solvers