# Embedding Untrusted Imperative ML Oracles into Coq Verified Code

Sylvain Boulmé, Thomas Vandendorpe

**To cite this version:**

Sylvain Boulmé, Thomas Vandendorpe. Embedding Untrusted Imperative ML Oracles into Coq Verified Code. 2019. hal-02062288v2

**HAL Id: hal-02062288**

**https://hal.science/hal-02062288v2**

Preprint submitted on 16 Jul 2019

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Embedding Untrusted Imperative ML Oracles into Coq Verified Code*

SYLVAIN BOULMÉ and THOMAS VANDENDORPE,
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, France,

This paper investigates a *lightweight approach* – combining Coq and OCaml typecheckers – in order to *formally verify* higher-order imperative programs in *partial correctness*. In this approach, the user does never formally reason about effects of imperative functions, but only about their results. Formal guarantees are obtained by combining *parametric reasoning over polymorphic functions* (i.e. "theorems for free" a la Wadler) with *verified defensive programming*. This paper illustrates the approach on several examples. Among them: first, the certification of a polymorphic memoized fixpoint operator using untrusted hash-tables; second, a certified Boolean SAT-solver, invoking internally any untrusted but state-of-the-art SAT-solver (itself generally programmed in C/C++).

Additional Key Words and Phrases: The Coq Proof Assistant, Monads, Polymorphism, Parametricity

## 1 INTRODUCTION

The CompCert certified compiler [Leroy 2009a,b] is the first C compiler with a formal proof of correctness that is used in industry [Bedin França et al. 2012; Kästner et al. 2018]. It is a major success of software verification, because CompCert does not have the bugs which are usually found in optimizing compilers [Yang et al. 2011]. Its success partly comes from its smart design, focusing the *formal proof* in Coq on *the partial correctness* of compilation passes, while reasonings on their *performance* (including thus their termination) remain *informal*. In particular, CompCert invokes untrusted oracles from the certified code. For example, register allocation in compilers is a NP-complete problem: finding a valid allocation is difficult, while checking the validity of an allocation is easy. In CompCert, the allocation is found by an *oracle*, i.e. an untrusted OCaml function; and, only a *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 or tuned for some specific cases, without requiring to reprove the checker.

In some certified software like the certified UNSAT prover of [Cruz-Filipe et al. 2017a], oracles are invoked *before* certified code which only checks their outputs. 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 OCaml code from certified code. However, there is no formal justification that using this FFI is sound. Section 1.1 details the main pitfall of this FFI. Section 1.2 sketches a proposal to fix this issue. Section 1.3 summarizes the contributions of the paper.

*This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR"

Authors' address: Sylvain Boulmé; Thomas Vandendorpe,
Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France,
sylvain.boulme@univ-grenoble-alpes.fr, thomas.vandendorpe@etu.univ-grenoble-alpes.fr.

2019. Manuscript submitted to ACM

### 1.1 Unsoundness of the Standard FFI w.r.t OCaml

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` 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`. Finally, a lemma `cong` is proved, using the fact that `test` is a function.

```
Definition one: nat := (S 0).
Axiom test: nat → bool.
Extract Constant test ⇒ "oracle".

Lemma cong: test one = test (S 0).
  auto.
Qed.
```

However, implementing `oracle` by "`let oracle x = (x == one)`" in OCaml makes the lemma `cong` false at runtime. Indeed, `(oracle one)` returns **`true`** whereas `(oracle (S 0))` returns **`false`**, because "==" tests equality between *pointers*. Hence, the Coq axiom is unsound w.r.t this implementation. A similar unsoundness is obtained with another implementation of `oracle`, that returns the value of a global reference, containing **`true`** at the first call, and **`false`** at the second call.[2] This unsoundness comes from the fact that a Coq function $f$ satisfies "$\forall x, (f\ x) = (f\ x)$", whereas 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`: the remaining of the compiler does not depend on whether `regalloc` is *pure* or not.[3]

### 1.2 Foreign Functions as Non-Deterministic Functions

[Fouilhé and Boulmé 2014] have proposed 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 `cong` 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 $\mathcal{P}(A)$, the type $A \longrightarrow \text{Prop}$ of predicates over $A$. Formally, the type transformer "$??.$" is axiomatized as a monad that provides a *may-return* relation $\leadsto_A : ??A \longrightarrow A \longrightarrow \text{Prop}$. Intuitively, when "$??A$" is seen as "$\mathcal{P}(A)$", then "$\leadsto$" simply corresponds to identity. At extraction, $??A$ is extracted like $A$, and its binding operator is efficiently extracted as an OCaml let-in. See details in Section 2.

For example, replacing the `test` axiom by "`Axiom test: nat → ??bool`" avoids the above unsoundness w.r.t the OCaml `oracle`. The `cong` property can still be expressed as below, but it is no longer provable – because it is not satisfied when interpreting $??A$ as $\mathcal{P}(A)$ and interpreting `test` as the function returning the trivially true predicate (in this interpretation, the goal below reduces to the false property that all Booleans are equals).

```
cong: ∀ b b', (test one)⇝b →(test (S 0))⇝b' →b=b'.
```

[1]See https://github.com/AbsInt/CompCert/blob/master/backend/Allocation.v
[2]For example defined with "`let oracle = let h=ref false in (fun x -> h:=not !h; !h)`".
[3]The current implementation of `regalloc` uses imperative hash-tables: it is not obvious if it is observationally pure or not – in the Coq sense.

Of course, this approach does not suffice to avoid all pitfalls of axiomatizing oracle types in Coq. Some other pitfalls are detailed in the paper, with proposal of remedies.

### 1.3 Contributions of the Paper

The *may-return* monad of [Fouilhé and Boulmé 2014] aims to ensure that Coq proofs like those of CompCert do not rely on the purity of external oracles. Based on a variant of the *may-return* monad, this paper proposes a library called Impure which is conjectured to provide a safe FFI for almost any well-typed OCaml function (see details in Section 2). Hence, this FFI allows to embed many impure OCaml features into Coq certified code. This is illustrated along the paper on I/O operations, exception-handling, mutable data-structures and physical equality. Whereas this approach inherits of the full programming power of OCaml, its reasoning power is rather limited as detailed just below.

For example, on I/O operations, Impure does not provide any reasoning support unlike the coq.io library of Guillaume Claret. Actually, embedding I/O in Coq code is already very convenient, even without any formal proof about those I/O. In particular, this allows us to write the main function of our executables in Coq: when such a main function is sufficiently small, it can be considered itself as a part of the formal specification (see Figure 15 at page 16).

On exception-handling and on polymorphic mutable data-structures, Impure supports "*theorems for free*" *a la* Wadler [Wadler 1989] that are derived by embedding *invariants* into the polymorphic types of OCaml oracles. Even if this technique suffers from a very incomplete reasoning power (for example, it cannot even prove that "`x:=1; !x`" returns "`1`"), it suffices to formally prove many interesting properties "for free".

Actually, in counter-part of this limited reasoning power, there is only a very small bureaucratic overhead with respect to reasoning about pure code.[4] The only overhead comes from the fact that every impure computation is encapsulated in a monad. And, in realistic developments like CompCert, even pure computations are often handled through an error monad. So, we believe that replacing the error monad of CompCert by the may-return monad would not make the proofs heavier.[5]

In other words, the Impure library provides a framework to combine Coq and OCaml type-checkers. Surprisingly, in a defensive style (i.e. with dynamic checks of oracle results), this simple framework is very effective. This is illustrated on two main examples: first, the certification of a polymorphic memoized fixpoint operator using untrusted hash-tables; second, a certified frontend to any untrusted but state-of-the-art SAT-solver. This second—very significant—example is partly inspired by the Coq-verified checker of UNSAT certificates proposed by Cruz-Filipe et al. [2017a]. Our main contribution on this example is to illustrate how our "theorems for free" technique helps to develop a code, which is *much* faster than this previous one—with a very modest development effort.

This paper is more experimental than theoretical. It proposes a very pragmatic solution to the formal verification of complex software like SAT solvers or compilers. In particular, with some of our colleagues, we have also fruitfully applied our approach to extend a CompCert backend with an efficient postpass scheduling. This other significant development (20Kloc of Coq + 4Kloc of OCaml) is presented in [Six et al. 2019].

[4]Full reasoning on imperative functions often requires to write (and prove) bureaucratic specifications of functions with respect to their effect on the global environment. For examples, see Frama-C [Kirchner et al. 2015] or CFML [Charguéraud 2011]. The drastic simplicity of our approach comes from the fact that it handles only two effects: "*nothing*" (for pure Coq functions) or "*everything-compatible-with-typing*" (for Coq functions embedding OCaml code).

[5]However, such a replacement would require a significant amount of work, because the two monads differs.

### 1.4 Overview of the Paper

Section 2 presents our proposal of FFI (through the Impure library) and conjectures its soundness. Section 3 applies it to extend the Coq programming language with some polymorphic impure operators: exception-handling and fixpoints. Finally, Section 4 applies it to certified SAT-solving. An appendix details some technical points of the paper.

## 2 TOWARD A SOUND FFI W.R.T OCAML THROUGH MAY-RETURN MONADS

As sketched in introduction, this section will define a type $A \rightarrow ??B$ to represent the type of *impure functions* from type $A$ into type $B$. Informally, we interpret the type $??B$ as $\mathcal{P}(B)$ the type of predicates characterizing the possible results. This interpretation represents thus each impure function as a function of $A \rightarrow \mathcal{P}(B)$, or equivalently, as a relation of $\mathcal{P}(A \times B)$, because of the bijection between this two types. Section 2.1 defines type $??B$ using axioms in order to provide an efficient extraction into OCaml, where " ??" are simply removed. Based on this notion of impure computations, Section 2.2 presents our FFI and conjectures its soundness. Section 2.3 explains how this conjecture is related to a parametricity property of the underlying "Coq+OCaml" type system. Finally, Section 2.4 extends the FFI with pointer equality.

### 2.1 Definition of the May-Return Monad in the Impure library

This section introduces in an informal syntax the theory of the may-return monads and presents the informal interpretation of these axioms. See the sources online[6] for the full Coq syntax with the proofs. The definition of may-return monads in this paper – given below – is inspired by the original definition of [Fouilhé and Boulmé 2014], itself inspired by the structure of monads in functional programming languages [Wadler 1995]. There are however two differences between the definition below and the original one. First, in this paper, the congruence "≡" over computations has been omitted. Indeed, in the VPL, the Verified Polyhedra Library of [Fouilhé and Boulmé 2014], this congruence is only needed in order to prove a property on a higher-order operator that is absent of the case studies of this paper. Moreover, as discussed in Appendix A, the meaning of such an equality with respect to the extracted code is counter-intuitive: an issue that we keep out of the scope of this paper. Second, this paper introduces the "$\text{mk_annot}_A$" operator, that is invoked in order to prove properties on higher-order operators by parametricity (see Section 3).

*Definition 2.1 (May-return monad).* For any type $A$, type $??A$ represents impure computations returning values of type $A$, and provides a may-return relation

$$\leadsto_A : ??A \rightarrow A \rightarrow \text{Prop}$$

where "$k \leadsto a$" means that "$k$ *may return* $a$". It also provides the three following operators

- Operator $\gg\!=_{A,B} : ??A \rightarrow (A \rightarrow ??B) \rightarrow ??B$ encodes an impure OCaml sequence "**let** $x = k_1$ **in** $k_2$" into "$k_1 \gg\!= \lambda x, k_2$". This operator must satisfy

$$k_1 \gg\!= k_2 \leadsto b \;\Rightarrow\; \exists a, k_1 \leadsto a \wedge k_2\, a \leadsto b$$

- Operator $\text{ret}_A : A \rightarrow ??A$ lifts a pure value as an impure computation. It must satisfy

$$\text{ret}\, a_1 \leadsto a_2 \;\Rightarrow\; a_1 = a_2$$

- Operator $\text{mk_annot}_A : \forall (k : ??A),\, ??\{\, a \mid k \leadsto a\}$ annotates the result of a computation $k$ with an assertion expressing that it has been returned by $k$.

[6] http://github.com/boulme/Impure/blob/master/ImpMonads.v

In the Coq code, “$k_1 \gg= \lambda x, k_2$” is written with a “DO” notation reminiscent of Haskell “`DO x <~ k1;; k2`” (or “`k1;; k2`” if `x` does not appear in `k2`). And ret is also written with cases “`RET`” to increase readability of impure code.

*2.1.1 Interpretations of May-Return Monads.* Here is the informal interpretation of “ $??A$” as the type of predicates “$\mathcal{P}(A)$”: $\leadsto_A$ is identity on $\mathcal{P}(A)$; $\text{ret}_A$ is the identity relation of $A \rightarrow \mathcal{P}(A)$; $\gg=_{A,B}$ returns the image of a predicate on $A$ by a binary relation of $A \rightarrow \mathcal{P}(B)$; mk_annot returns the trivially true predicate. Theses definitions are formalized in Figure 1. They satisfy axioms of may-return monads.

Actually, it is worth noticing that usual monads are naturally embedded as a may-return monad. For example, Figure 2 corresponds to the embedding of the identity monad. And, Figure 3 corresponds to the embedding of the state-monad on a given global state of type $S$.

$$??A \triangleq A \rightarrow \text{Prop} \qquad k \leadsto a \triangleq (k\ a) \qquad \text{ret}\, a \triangleq \lambda x, a = x$$

$$k_1 \gg= k_2 \triangleq \lambda x, \exists a, (k_1\ a) \wedge (k_2\ a\ x) \qquad \text{mk_annot}\, k \triangleq \lambda x, \text{True}$$

Fig. 1. Power-set instance of may-return monads

$$??A \triangleq A \qquad k \leadsto a \triangleq k = a \qquad \text{ret}\, a \triangleq a \qquad k_1 \gg= k_2 \triangleq (k_2\ k_1)$$

$$\text{mk_annot}\, k \triangleq \text{exist}_{\leadsto}\ k\ \text{eq_refl}_k$$

where
- $\text{exist}_{\leadsto}$ is the constructor of the dependent pair $\{ a \mid k \leadsto a \}$
- $\text{eq_refl}_k$ is a proof of $k = k$

Fig. 2. Identity instance of may-return monads

$$??A \triangleq S \rightarrow A \times S \qquad k \leadsto a \triangleq \exists s, \text{fst}(k\ s) = a \qquad \text{ret}\, a \triangleq \lambda s, (a, s)$$

$$k_1 \gg= k_2 \triangleq \lambda s, \text{let}\ (a, s') := (k_1\ s)\ \text{in}\ (k_2\ a\ s')$$

$$\text{mk_annot}\, k \triangleq \lambda s, \text{let}\ (a, s') := (k\ s)\ \text{in}\ (\text{exist}_{\leadsto}\ a\ p_{k,s},\ s')$$

where $p_{k,s}$ is a proof of $\exists s_0, \text{fst}(k\ s_0) = \text{fst}(k\ s)$

Fig. 3. State-transformer instance on a global state of type $S$

In order to handle impure computations in Coq, the Impure library declares an abstract may-return monad (i.e. its implementation remains hidden). It is extracted as like as the identity may-return monad of Figure 2 except that, in order to enforce the expected evaluation order, operator $\gg=$ is extracted to operator `(|>)` of OCaml.[7]

*2.1.2 Reasoning on Impure Computations with Weakest-Liberal-Preconditions.* Having introduced axioms for impure computations in Definition 2.1, we automate Coq reasonings about such computations, by reusing a weakest-precondition calculus introduced by [Fouilhé and Boulmé 2014] and programmed as a very simple Ltac tactic. They define in Coq an operator

[7] See http://github.com/boulme/Impure/blob/master/ImpConfig.v

$$\text{“wlp}_A : ??A \longrightarrow (A \longrightarrow \text{Prop}) \longrightarrow \text{Prop”} \qquad \text{such that} \qquad \text{“wlp } k\ P \triangleq \forall a, k \rightsquigarrow a \Rightarrow (P\ a)\text{”.}$$

In other words, (wlp $k$ $P$) expresses the weakest (liberal) precondition ensuring that any result returned by computation $k$ satisfies postcondition $P$. More simply, when $??A$ is interpreted as $\mathcal{P}(A)$, wlp corresponds to inclusion of predicates. Now, we define the notion of WLP-theorems.

*Definition 2.2 (WLP-theorems).* A WLP-theorem is a Coq theorem with a conclusion of the form "(wlp $k$ $P$)". Such a theorem means that (under the hypotheses of the theorem),

For all $r$, if the extraction of $k$ returns the extraction of $r$, then $r$ satisfies $P$.

In particular, when the extraction of $k$ does not terminate or raises an uncaught exception, WLP-theorems do not give any useful information (as usual in *partial correctness*). In our Coq code, we write (wlp $f$ $\lambda r, P$) with the notation "`WHEN f ⇝ r THEN P`". For example, let us consider the following Coq code:

```
Variable f: nat → ?? nat.
Definition g (x:nat): ?? nat := DO r ⇜ f x;; RET (r+1).
Lemma triv: ∀ x, WHEN g x ⇝ r THEN r > 0.
```

The Ltac tactic simplifies this goal into the trivial property "$\forall n : \mathbb{N}, n+1 > 0$" (See [Fouilhé and Boulmé 2014] for details).

### 2.2 The Foreign-Function-Interface provided by Impure

As shown in introduction, declaring external OCaml oracles in Coq may be *unsound*, by authorizing Coq theorems that can be false at runtime. The may-return monad has been introduced in order to avoid the pitfall of embedding impure computations as pure functions. But this is not sufficient to ensure soundness. To this end, we need to define a class "*permissive*" of Coq types and a class "*safe*" of OCaml values satisfying Conjecture 2.4 below, with "*being permissive*" and "*being safe*" automatically checkable, and as expressive as possible. In this paper, we consider the following definition for "*safe*". The definition of "*permissive*" will be gradually introduced in Section 2.2.1 upto Definition 2.5.

*Definition 2.3 (Safe OCaml value).* An OCaml value is "*safe*" **iff** it is well-typed and without calls to external values like `Obj.magic`, and without using cyclic values like "`let rec v = S v`" on Coq extracted types.[8] NB: The last restriction does not forbid recursive functions nor cyclic mutable data-structures in *safe* OCaml values.

Conjecture 2.4 (Soundess of permissive Coq types). Every "*permissive*" Coq type $T$ according to Definition 2.5 satisfies the following property:

every safe OCaml value compatible with the extraction of $T$ is "*soundly*" axiomatized in Coq with type $T$ – in the sense that WLP-theorems deduced from the axiom cannot be falsified when running the extracted code, in which the axiom has been replaced by the OCaml value.

Ideally, we aim to extend Coq with a "`Import Constant`" construct of the form:

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

and acting like "`Axiom ident: permissive_type`", but with additional checks during Coq and OCaml typechecking in order to ensure soundness of extraction. However, defining precisely such typechecking algorithms is left for future work.

[8] Appendix B details issues of cyclic values.

*2.2.1 Definition of Permissivity.* This section gradually introduces our definition of *permissivity* in order to enforce Conjecture 2.4 on its soundness. We first give examples of unsound types, and examples that are conjectured to be sound. Section 1.1 have illustrated that type `nat→bool` is unsound: thus, it cannot be permissive. On the contrary, type `nat→??bool` is conjectured to be sound. We also conjecture that `nat→??nat` is sound. But, type `nat→??{n:nat | n ≤ 10}` – extracted to "`nat->nat`" – is not. Indeed, such a Coq type corresponds to assume a *postcondition* on the oracle that the OCaml typechecker cannot ensure.

Similarly, type `nat→??(nat→nat)` – extracted to "`nat->(nat->nat)`" – is unsound because the Coq side expresses that the result of type `nat→nat` is pure, whereas this (implicit) postcondition cannot be ensured by OCaml typechecker. Actually, the same phenomenon happens with `nat→(nat→??nat)` (extracted on the same OCaml type): the partial application on the first argument is declared pure in Coq, whereas this cannot be ensured by OCaml typechecker.

On the contrary, types `nat→??(nat→??nat)` and `(nat→??nat)→??nat` are conjectured to be sound. And also `{n | n ≤ 10}→??nat`. On this last example, the Coq axiom requires a *precondition* that OCaml typechecker can safely ignore. A similar phenomenon happens with `(nat→nat)→??nat` (the purity of the parameter is an implicit precondition that OCaml typechecker can safely ignore). Note that currying in sound oracle types—like in `nat→??(nat→??nat)`—allows for more imperative OCaml implementations (at the price of more bureaucracy on the Coq side) than tupling—like in `nat*nat→??nat`.

In the general case, permissitivity can be viewed as a given *supertyping* relation between Coq types and OCaml types: a Coq type is permissive if and only if it is a supertype of its extraction. In this view, permissivity of arrow types requires to distinguish "inputs" (negative occurrences) from "outputs" (positive occurrences): outputs are covariant and inputs are contravariant. We thus also need to introduce a *subtyping* relation between Coq types and OCaml types. On "basic" types, motivated by the correctness of Coq extraction[Letouzey 2008], we consider that a Coq type is always a subtype of its extraction. For example, `{n:nat | n ≤ 10}` is a subtype of OCaml `nat`. In order to deal with more complex types, the subtyping relation must be mutually defined together with the supertyping relation. Here, we leave the precise definition of the subtyping relation for future works. These considerations leads us to the following definition of permissivity.

*Definition 2.5 (Permissivity).* The permissivity of a Coq type is defined inductively over the syntax of types:

- An inductive type is permissive whenever its sort is not **`Prop`** and whenever the type of each input of each constructor is permissive. See example below.
- An arrow type is permissive whenever the arrow is followed by a "`??`", and its output type is permissive, and its input type is a subtype of its extraction.
- ML polymorphism – i.e. prenex universal polymorphism – preserves permissivity.

For example, given type `foo` below, type `nat→??foo` is permissive. But this would not be the case if constructor `Bar` has no "`??`" in the type of its argument.

```
Inductive foo := Bar: (nat → ??nat) → foo
```

A more advanced example of permissive type is given by the polymorphic type of `make_cref` in Figure 4. Section 2.3 illustrates that Conjecture 2.4 implies a powerful parametricity property on such a polymorphic oracle.

*2.2.2 Application to Imperative Programming in Coq.* Let us start exploring basic imperative programming in Coq, by using mutable data-structures and I/O. Let us first consider the embedding

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

Fig. 4. A Coq FFI of mutable references

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

Fig. 5. Standard OCaml implementation of `make_cref`

```
let make_cref x =
  let h = ref [x] in {
    set = (fun y -> h:=y::!h);
    get = (fun () -> List.nth !h (Random.int (List.length !h))) }
```

Fig. 6. Iconic variant of `make_cref`

of mutable references with the Coq code of Figure 4: it defines the record type `cref` that represents references in a kind of object-oriented style (as the pair of a mutator `set` and a selector `get`), and declares an oracle `make_cref` building values of this type. On the OCaml side, type `cref` is extracted to "`type 'a cref = { set: 'a -> unit; get: unit -> 'a }`". Then, we define `make_cref: 'a -> 'a cref` such that it allocates a fresh reference `r` and returns the pair of `set`/`get` function to update/access the content of `r` (see the code in Figure 5). Conjecture 2.4 states that it is sound to implement `make_cref` by any *safe* OCaml function of type `'a -> 'a cref` like in Figure 5. Having implemented `make_cref` according to Figure 5, the user can thus program with mutable references in Coq. However, most properties of this implementation cannot be *formally* proven in Coq.

Indeed, from the point-of-view of the formal logic, any *safe* OCaml function of type `'a -> 'a cref` is admitted as a sound implementation of `make_cref`, including the *iconic* implementation of Figure 6. This implementation typifies what any OCaml implementation of `make_cref` can do: store inputs of `make_cref` and `set` such that `get` outputs one of the previously stored input. For example, every execution using the implementation of Figure 5 can be emulated by an execution using the implementation of Figure 6 where each call to `Random.int` returns 0: in this way, `get` outputs the last received input.

Hence, all formal properties provable from the interface of Figure 4 should be satisfied by the oracle of Figure 6. Thus, they can only express that if all the inputs of a given reference satisfy some given *invariant*, then the value returned by `get` will also satisfy this invariant. Such a property can be partly expressed in Coq by instantiating the parameter `A` of `cref` in Figure 4 on a $\Sigma$-type that constrains this reference to preserve the given invariant. Whereas this technique seems a bit weak on this example, Sections 3.3 and 4 present interesting applications of this lightweight technique for constraining polymorphic mutable data-structures (like hash-tables). Finally, let us note that our embedding of ML references does not forbid aliases as soon as they are compatible with Coq typing: see details in Appendix C.

However, extending extracted code with an OCaml main could in theory break some properties proved on the Coq side (see also Appendix C). It is thus safer to define the main function of executables on the Coq side. This motivates to embed some I/O functions in Coq. Such an embedding is very easy. Currently, the Impure library provides a few wrapper of some functions of OCaml standard library, like the below two (where `pstring` is a Coq type to represent strings).

```
Axiom read_line: unit → ?? pstring.  (* reads a line from stdin *)
Axiom println: pstring → ?? unit.    (* prints a line on stdout *)
```

However, the Impure library does not provide any formal reasoning support on these I/O functions. Hence, in this approach, reasoning with I/O on Coq code remains informal – more or less like on OCaml code. The programmer is only much more protected against stupid mistakes when combining formally proved code and trusted (but informally verified) code, because the Coq type system is more accurate.

### 2.3 Coq "Theorems for Free" about Polymorphic ML Oracles

According to Definition 2.5, a polymorphic type like "∀ A, A→??A" is permissive. Together with Conjecture 2.4, this implies a "theorem for free" on *safe* OCaml values of the corresponding extracted type. For example, we now prove that any safe OCaml value `pid` of type `'a -> 'a` satisfies

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

In the following, we say that a function `pid` satisfying the above property is a pseudo-identity (indeed, such a `pid` may not be the identity because it may not terminate normally or produce side-effects).

In order to prove that any safe "`pid:'a -> 'a`" is a pseudo-identity, we first declare `pid` as an external function in Coq. Then, we build a Coq function `cpid`, which is proved to be a pseudo-identity, and which is extracted to OCaml as "`let cpid x = pid x |> (fun z -> z)`". In the Coq source, for a type `B` and a value `x:B`, `(cpid x)` invokes `pid` on the type `{y:B|y=x}`, which constrains it to produce a value that is equal to `x`. Below, `` `z `` returns the first component of the dependent pair `z` of type `{y:B|y=x}`; the **Program** environment allows for terms with "holes" (like here in the implicit coercion of `x:B` into a value of `{y:B|y=x}`) and generates static proof obligations to fill the holes.[9]

```
Axiom pid: ∀ A, A→??A.
Program Definition cpid{B}(x:B):??B := DO z ⇜ pid {y|y=x} x;; RET `z.
Lemma cpid_correct A (x y:A): WHEN (cpid x) ⇝ y THEN y=x.
```

Let us point out that we cannot prove in Coq that `pid` – declared as the axiom given above – is a pseudo-identity. Indeed, we provide a model of this axiom where `pid` detects – through some dynamic typing operators – if its parameter `x` has a given type `Integer` and in this case returns a constant value, or otherwise returns `x`. Such a counter-example already appears in [Vytiniotis and Weirich 2007]. This function is now provided in Java syntax.

```
static <A> A pid(A x) {
  if (x instanceof Integer) // A==Integer, because Integer is final
    return (A)(new Integer(0));
  return x;
}
```

The soundness of `cpid` extraction is thus related to a nice feature of ML: type-safe polymorphic functions cannot inspect the type to which they are applied. In other words, type erasure in ML semantics ensures that functions handle polymorphic values in a uniform way.

However, a similar counter-example can be built for OCaml by using an external C function that is sound with type `'a -> 'a` (i.e. for all ML type $T$, it behaves like a function of type $T \longrightarrow T$). Such a function inspects the bit of its parameter that tags unboxed integers, and returns integer 0

[9] This small example also illustrates how our approach benefits from powerful features of Coq like **Program**.

when instantiated on type `int`, or behaves like an identity otherwise. This explains why such a counter-example must be rejected by Definition 2.3.

In summary, our Coq proof is not about `pid`, but about `cpid` which instantiates `pid` on a dependent type. Actually, `cpid` and `pid` coincide, but only *in the extracted code*. This proof can be viewed as a “theorem for free” in the sense of Wadler [1989]: it is a parametricity proof for a *unary* relation, i.e. a predicate that we call here an *invariant*. Bernardy and Moulin [2012, 2013] have previously demonstrated that parametricity reasoning can be constructively internalized in the logic from an erasure mechanism. Here, in our “Coq+OCaml” logic of programs, it derives from the fact that the invariant instantiating the polymorphic type variable in the Coq proof is syntactically removed by Coq extraction.

But, whereas parametricity of pure system F has been established a long time ago by Reynolds [1983], its adaptation to imperative languages with higher-order references *a la ML* is much more recent [Birkedal et al. 2011]. Indeed, because higher-order references allows to build recursive functions without explicit recursion (see Figure 9 page 12), it is even hard to define what is a predicate over such a higher-order reference. See [Ahmed et al. 2002; Appel et al. 2007; Hobor et al. 2010]. This paper leaves Conjecture 2.4 for future works, and focuses on its poweful applications.

### 2.4 Axioms of the Trusted Equality of Pointers

We now extend the FFI described at Section 2.2, by embedding the physical equality (i.e. pointer equality) of OCaml into Coq. At the difference of all other oracles in this paper, we impose the `phys_eq` oracle to satisfy an axiom – called `phys_eq_true` – in addition to its declaration. Thus, the implementation of this oracle must be trusted.

```
Axiom phys_eq: ∀ {A}, A → A → ?? bool.
Extract Constant phys_eq ⇒ "(==)".
Axiom phys_eq_true: ∀ A (x y: A), phys_eq x y ⇝ true →x=y.
```

As illustrated on the example of Section 1.1, because “`(==)`” distinguishes pointers: it can distinguish values that the Coq logic cannot. It is thus necessary to declare `phys_eq` as a non-deterministic function. The `phys_eq_true` axiom is useful in order to replace some tests about structural equality by faster tests using physical equality instead. This axiom is similar to the one introduced by Breitner et al. [2018] to model the pointer equality of Haskell. Section 3.3 gives an example. It also been used to implement a verified hash-consing mechanism in the CompCert backend of Six et al. [2019].

## 3 CERTIFYING “FOR FREE” POLYMORPHIC IMPERATIVE FUNCTIONS

This section applies our FFI in order to extend the Coq programming language with some polymorphic impure operators[10]: exception-handling at Section 3.1, loops at Section 3.2 and fixpoints at Section 3.3. Our goal is to formally prove the *usual rules of Hoare logic* for these operators *in partial correctness*. This is achieved by applying the technique of “*parametricity by invariants*” (introduced at Section 2.3): we derive these correctness rules by instantiating the polymorphic type of well-chosen oracles on a well-chosen sigma-type. In other words, we illustrate that “parametricity by invariants” interprets ML polymorphic types as “*higher-order invariants*”, i.e. invariant properties (of ML values) depending on type variables which names themselves some invariant. Hence, with this interpretation, ML typecheckers are powerful engine to infer higher-order invariants in partial correctness.

[10]The full Coq/OCaml code of these examples is online at https://github.com/boulme/ImpureDemo

### 3.1 Exception-Handling Operators

First, we declare an external function `fail` which is (informally) expected to raise an error parametrized by a string.

```
Axiom fail: ∀ {A}, pstring → ?? A.
```

This axiom is safely implemented by the following OCaml function `fail: pstring -> 'a`.

```
exception ImpureFail of pstring
let fail msg = raise (ImpureFail msg)
```

But, this axiom is also safely implemented by the following OCaml function `fail: 'a -> 'b`.

```
let rec fail msg = fail msg
```

Actually, while our *formal* Coq reasonings will be valid for any of these implementations, our *informal* reasonings will only consider the first implementation.

For its formal correctness, `fail` does never return a value, or equivalently it returns only values satisfying any predicate. In order, to get this property "for free", we wrap `fail` into a function `FAILWITH` of the same type, but which internally call `fail` on the empty type `False`. For any value `r:False` returned by `fail`, we are thus able to build any value of any type (by destructing `r`).

```
Definition FAILWITH {A:Type} msg: ?? A :=
  DO r ⇜ fail (A:=False) msg;; RET (match r with end).

Lemma FAILWITH_correct A msg (P: A → Prop):
  WHEN FAILWITH msg ⇝ r THEN P r.
```

Now, we use `FAILWITH` to define dynamic assert checking. Below, "`assert_b`" ensures that a (pure) Boolean expression is `true` or aborts the computation otherwise.

```
Program Definition assert_b (b: bool) (msg: pstring): ?? b=true :=
  match b with
  | true ⇒ RET _
  | false ⇒ FAILWITH msg end.
Lemma assert_correct msg b: WHEN assert_b b msg ⇝ _ THEN b=true.
```

This approach is extended to exception-handling with the following oracles, which are used in Figure 15. See Appendix D for details about formal reasoning on exception-handling.

```
Axiom exn: Type.   Extract Inlined Constant exn ⇒ "exn".
Axiom raise: ∀ {A}, exn → ?? A.  Extract Constant raise ⇒ "raise".
Axiom try_with_any: ∀ {A}, (unit → ?? A) * (exn → ??A) → ??A.
Notation "'TRY' k1 'WITH_ANY' e '⇒' k2" :=
  (try_with_any (fun _ ⇒ k1, fun e ⇒ k2)) ...
```

Here `try_with_any` is implemented in OCaml by

```
let try_with_any (k1, k2) = try k1() with e -> k2 e
```

### 3.2 Generic Loops in Coq

This section defines a verified WHILE-loop in partial correctness. Let us first introduce our untrusted oracle for generic loops. We use type `A` as the type of "(potential) reachable states" in the loop (i.e. `A` is the loop invariant). We also use type `B` as the type of "(potential) final states" (i.e. `B` is the post-condition of the loop). Our `loop` oracle is parametrized by an initial state of type `A` and by a function "`step:A -> ??(A+B)`" computing the next state from a non-final state (see the

declaration of `loop` in Figure 7). Typically, the Coq type "`(A+B)`" being extracted on OCaml

```
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 → WHEN body s ⇝ s' THEN 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).
```

Fig. 7. Implementation of a WHILE-loop in Coq

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

Fig. 8. Standard OCaml implementation of oracle `loop` by a tail-recursive loop

```
let loop (a0, step) =
  let fix = ref (fun _ -> failwith "init") in
  (fix := fun a -> match step a with
                   | Coq_inl a' -> (!fix) a'
                   | Coq_inr b -> b);
  (!fix) a0
```

Fig. 9. Emulating recursion in OCaml with a cyclic higher-order reference

type "`('a, 'b) sum`" defined in Figure 8, we implement this `loop` oracle by the tail-recursive function of Figure 8. Any safe OCaml implementation of a compatible type is also admitted, like the alternative implementation of Figure 9. In this alternative implementation, recursion is not explicit in the code, but is emulated by a reference `fix` containing a function accessing `fix`. Here, the OCaml typechecker is able to verify that this obfuscated piece of code has the expected type.

After defining the `wli` predicate (acronym for "while-loop-invariant"), Figure 7 defines our verified `while` function. It is parametrized by a pure test `cond`, by an impure state-transformer `body`, by a predicate `I` preserved by one iteration of the loop (`wli` condition) and by an initial state `s0`. Parameter `A` (resp. `B`) of `loop` is instantiated on the loop invariant (resp. the postcondition). On this code, the **`Program`** plugin generates 3 trivial proof obligations:

(1) "`I s0 → I s0`".

(2) “`(I s0 → I s) → cond s = true → body s⇝s' →(I s0 →I s')`” (trivial from `wli` hypothesis).
(3) “`(I s0 → I s) → cond s = false → (I s0 → I s) ∧ cond s = false`”.

Let us remark that `mk_annot` is necessary to get the appropriate hypothesis on `s'` in the second proof obligations. Figure 21 in Appendix E illustrates how to apply this while-loop operator to an iterative computation of Fibonacci's numbers.

This technique could be applied to other kind of generic loops. For example, Figure 17 on page 24 defines a generic loop dedicated to refutation of unreachability properties. This generic loop is applied in Figure 18 to check an UNSAT property, as detailed in Section 4.4.3.

### 3.3 Generic Fixpoints in Coq

This section now extends the previous approach to generic fixpoints of functions. The simplest version of such a fixpoint in OCaml is given by `fixp` function in Figure 10. The `fixp` function computes the fixpoint of `step` a function performing one unfolding step of a recursive computation. For example, it is instantiated for the naive recursive computation of Fibonacci's number as “`fixp (fun fib p -> if p <= 2 then 1 else fib(p-1)+fib(p-2))`”.

Of course, with the implementation in Figure 10, this naive computation of Fibonacci's number performs an exponential number of additions. By using the memoized implementation on Figure 11, the number of additions remains linear. However, a bug in the implementation of `fixp` like in Figure 12 leads to incorrect results. Here, the implementation in Figure 12 represents an erroneous version of the memoized version of Figure 11 where all recursive results are crashed into a single memory cell (instead of associating each recursive result to its corresponding input into a dedicated memory cell).

```
let fixp (step: ('a -> 'b) -> 'a -> 'b): 'a -> 'b =
  let rec f x = step f x in f
```

Fig. 10. Standard fixpoint in OCaml

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

Fig. 11. Memoized fixpoint in OCaml

```
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
```

Fig. 12. An erroneous memoized fixpoint in OCaml

Hence, Figure 12 gives a *safe* implementation of type `(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b` that does not compute a correct fixpoint, even in partial correctness. This illustrates that the property "*be a correct fixpoint*" cannot be derived by *pure* parametric reasoning (on the contrary of the WHILE-loop of Section 3.2). However, we build a verified fixpoint operator from any fixpoint oracle of type `(('a -> 'b) -> 'a -> 'b) -> 'a -> 'b`, by combining parametricity-by-invariants and (inexpensive) defensive checks. In the case of implementation in Figure 12, the incorrect fixpoint computations will abort (because of the defensive checks). Hence, we declare the following oracle in Coq. And, we build a formally correct fixpoint operator by wrapping this oracle.

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

Usually, proving the correctness of a (non-tail)recursive functions requires to prove that a given *relation* between inputs and outputs is preserved by the unfolding step of recursion. Here, we need to encode this *binary* relation – called `R` below – into the *unary* invariant `B`. The trick is thus to store both the input (of type `A`) and the output (of type `B`) in this invariant. In the following, `A B: Type` and `R: A → B → Type` are implicit parameters of the formally proved fixpoint operator.

```
Record answ := { input:A; output:B; correct:R input output }.
```

Then, we add a **defensive check** on each recursive result `r` – returned through the oracle – that `(input r)` "*equals to*" the actual input of the call.

Thus, our fixpoint operator is also parametrized by an equality test `beq: A → A → ?? bool` that is expected to satisfy the following formal property.

```
∀ x y, WHEN beq x y ⇝ b THEN b=true → x=y.
```

For example, `beq` could be instantiated by the pointer equality `phys_eq` or a more structural equality test (as detailed later).

Then, we introduce a wrapper `wapply` of the application, such that each recursive call `k` returning a value of type `answ` is converted into a function `(wapply k)` returning a value of type `B`, but with a defensive check that the `input` field equals to the `x` parameter.

```
Definition wapply (k: A → ?? answ) (x:A): ?? B :=
  DO a ⇜ k x;;
  DO b ⇜ beq x (input a);;
  assert_b b msg;;
  RET (output a).
Lemma wapply_correct k x: WHEN wapply k x ⇝ y THEN R x y.
```

The parameter "`step:(A → ?? B) → A → ?? B`", that unfolds one step of recursion, is expected to preserve relation `R`, as formalized by `step_preserv` predicate.

```
Definition step_preserv (step: (A → ?? B) → A → ?? B) := ∀ f x,
  WHEN step f x ⇝ z THEN (∀ x', WHEN f x' ⇝ y THEN R x' y) → R x z.
```

Our proved `rec` operator is thus defined by:

```
Program Definition rec step (H:step_preserv step R): ?? (A → ?? B) :=
  DO f ⇜ fixp (B:=answ R)
              (fun k x ⇒
                DO y ⇜ mk_annot (step (wapply k) x);;
                RET {| input := x; output := `y |}
              );;
  RET (wapply f).
```

```
Lemma rec_correct step (H:step_preserv step R):
  WHEN rec step H ⇝ f THEN  ∀ x, WHEN f x ⇝ y THEN R x y.
```

Appendix E illustrates how to instantiate this **rec** operator on the naive recursive computation of Fibonacci's numbers. Actually, for performance issue, beq must be chosen at instantiation of operator **rec** according to fixp implementation. If beq is too much discriminating, then it may reject valid computations. On the contrary, if beq inspects too much the structure of its inputs, then it may slow down computations. For example, phys_eq is well-suited for the fixpoint implementation in Figure 10. But it is too much discriminating for the fixpoint implementation of Figure 11. Actually, for the latter, beq must correspond to the equality test involved in the hash-table implementation: here structural equality. Hence, our approach could be improved by passing beq as a parameter of the oracle, which then could use it as the equality test of the hash-table instead of structural equality.

## 4 CERTIFYING A CHECKER OF (BOOLEAN) SAT-SOLVER ANSWERS

This section presents a major contribution of our paper, by applying the Impure library to a realistic use-case: SatAnsCert, a verifier of SAT-solver answers, itself certified in Coq. Actually, for verifying UNSAT answers, we were inspired by a previous Coq development, called "lrat checker", documented in [Cruz-Filipe et al. 2017a] and available online[11]. Our main contribution is to illustrate how our "theorems for free" technique helps to develop a code, which is *much* scalable than this previous one—for a very modest development effort.[12]

### 4.1 Overview of SatAnsCert and its formal correctness

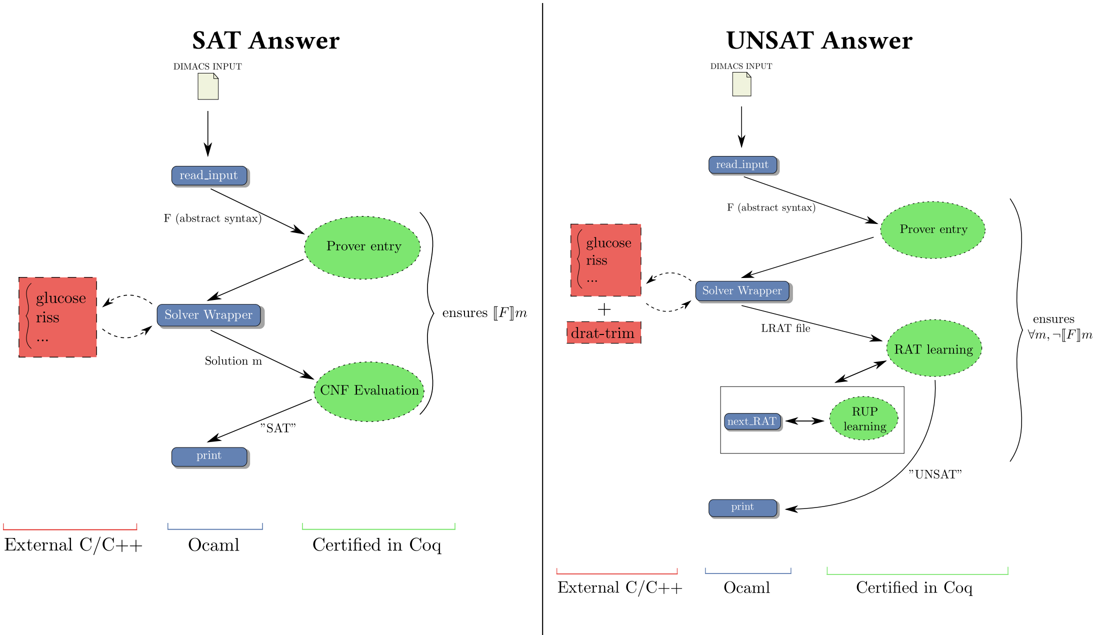


Fig. 13. Overview of SatAnsCert

[11] https://imada.sdu.dk/~petersk/lrat/

[12] The full Coq/OCaml code of these examples is online at https://github.com/boulme/satans-cert.

```
Definition var := positive.
Record literal := { is_pos: bool ; ident: var }.
Definition model := var → bool. (* boolean mapping *)
Definition clause := list literal.  (* syntactic clause *)
Fixpoint sat (c: clause) (m: model): Prop :=
  match c with
  | nil ⇒ False
  | l::c' ⇒ m(ident l)=(is_pos l) ∨ sat c' m
  end.
Definition iclause := clause_id * clause.  (* clause with an id *)
Definition cnf := list iclause.     (* syntactic cnf *)
Fixpoint sats (f: cnf) (m: model): Prop :=
  match f with
  | nil ⇒ True
  | c::f' ⇒ sat (snd c) m ∧ sats f' m
  end.
```

Fig. 14. Coq definitions of the abstract syntax of a CNF

```
1  Program Definition main: ?? unit :=
2    TRY
3      DO f ⇜ read_input();; (* Command-line + CNF parsing *)
4      DO a ⇜ sat_solver f;; (* solver(+drat-trim) wrapper *)
5      match a with
6      | SAT_Answer mc ⇒
7        assert_b (satProver f mc) "wrong SAT model";;
8        ASSERT (∃ m, ⟦f⟧ m);;
9        println "SAT !"
10     | UNSAT_Answer ⇒
11       unsatProver f;;
12       ASSERT (∀ m, ¬⟦f⟧ m);;
13       println "UNSAT !"
14   WITH_ANY e ⇒
15     DO s ⇜ exn2string e;;
16     println ("Certification failure: " +; s).
```

Fig. 15. (simplified) Coq code of the main function of SatAnsCert

SatAnsCert reads a proposition $f$ in Conjunctive Normal Form and outputs whether $f$ is *"SAT"* or *"UNSAT"* (see Definition 4.1 below). This proposition $f$ must be syntactically given in DIMACS file – a standard format[13] of SAT competitions. Internally, SatAnsCert invokes – according to options on its command line – some state-of-the-art SAT-solver like Glucose[14], Riss[15], CryptoMiniSat[16] or CaDiCaL[17]. This SAT-solver is expected to produce a witness of its answer (such a witness is mandatory for SAT competitions since 2016). SatAnsCert thus checks this witness before to output the answer or to fail on an error. The execution of SatAnsCert is depicted in Figure 13.

[13]https://www.satcompetition.org/2009/format-benchmarks2009.html
[14]http://www.labri.fr/perso/lsimon/glucose
[15]http://tools.computational-logic.org/content/riss.php
[16]https://github.com/msoos/cryptominisat
[17]http://fmv.jku.at/cadical

The external SAT-solver is run in a separate process and communicates with SatAnsCert through the file system. As later detailed, SatAnsCert also invokes some OCaml oracles through the FFI of the Impure library (presented in Section 2): these oracles are thus part of the SatAnsCert process. The external SAT-solver is actually invoked through one of this OCaml oracle.

*Definition 4.1 (Conjunctive Normal Form).* 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 "$m\,x$". A CNF is said *"SAT"*, if it has a model, and *"UNSAT"* otherwise.

Our Coq definitions of CNF abstract syntax is given in Figure 14. These definitions involve external clause identifiers of type `clause_id` without formal semantics. These identifiers are intended to relate clauses to their name in the UNSAT witness during its parsing by an untrusted oracle (which is later introduced). Here, type `clause_id` is opaque for the Coq proof: it remains uninterpreted. In the following, we use the bracket notations $[\![.]\!]$ for both predicates "`sat`" and "`sats`".

We now describe the formal property proved on SatAnsCert in Coq+Impure+OCaml. First, like in CompCert, I/O (ie parsing and printing) are not formally proved and thus must be trusted. More precisely, the formal correctness of SatAnsCert only deals with the abstract syntax (defined in Figure 14) of the input CNF. And, it is directly expressed in the main function of SatAnsCert through statically proved "`ASSERT`" (see Figure 15). Here, "`ASSERT P`" (where `P`:**`Prop`**) is simply a macro for "`RET (A:=P) _`": it declares a proof of proposition `P` that must be (statically) provided as a proof obligation generated by "**`Program Definition`**". We consider that the ability to use imperative code in Coq with statically verified assertions improves the approach of CompCert– where formally proved components and unproved (but trusted) components are linked together in OCaml only.

Hence, our code in Figure 15, thus combines *static* assertions ("`ASSERT`") and *dynamic* assertions, like "`assert_b`" defined on page 11. The static "`ASSERT`" proved at line 8 derives from the defensive check of line 7: `satProver` simply evaluates CNF `f` in the model `mc` found by the SAT-solver. Similarly, the static "`ASSERT`" proved at line 13 derives from a defensive check of line 12: `unsatProver` checks that the UNSAT witness (here implicit) provided by the SAT-solver is valid, or fails otherwise. The next sections sketch how this latter verification is achieved. Section 4.3 defines a first simple version with type:

```
unsatProver (f: cnf): ?? (∀ m, ¬⟦f⟧m)
```

And, Section 4.4 defines a second refined version with the equivalent type:

```
unsatProver (f: cnf): ?? ¬(∃ m, ⟦f⟧m)
```

### 4.2 Certifying UNSAT answers of SAT-solvers: a brief overview

Since the pioneering works of [Goldberg and Novikov 2003] and [Zhang and Malik 2003], the verification of UNSAT answers has been well studied. Several proof formats have been proposed, and currently, the DRAT format [Heule 2016; Wetzler et al. 2014] is the standard format in SAT competitions. Actually, most SAT-solvers generate only DRUP proofs [Gelder 2008; Heule et al. 2013] – a previous format that DRAT has later extended with RAT clauses [Wetzler et al. 2013]. In theory, using RAT clauses may lead to exponentially shorter proofs than using only pure (D)RUP

proofs. But, in practice, the SAT-solving community is still looking for efficient algorithms to find such RAT proofs [Heule et al. 2017].

*4.2.1 Background on Resolution, RUP proofs and CDCL (Conflict-Driven Clause Learning).* In a first step, this paper focuses only on (D)RUP proofs: they are simpler to understand. Actually, we even consider a strict subset of RUP proofs, introduced as "*restricted RUP* proofs" in [Cruz-Filipe et al. 2017b], that we rename (for clarity) into "*backward resolution* proofs". Indeed, we consider a variant of the *resolution proof system* where the resolution rule is *specialized* for backward reasoning through the rule BckRsl of Definition 4.2 below. Together with rule Triv, we recover the usual resolution rule: for any literal $\ell$, any clauses $c_1$ and $c_2$, there exists a list of clauses $f$ included in the list of two clauses "$\{\ell\} \cup c_1, \{\neg\ell\} \cup c_2$" such that "$f \vdash^{\text{BRC}} c_1 \cup c_2$". Indeed, if $\ell \notin c_1 \cup c_2$, then we define $f$ as the whole list, and "$f \vdash^{\text{BRC}} c_1 \cup c_2$" because $(\{\ell\} \cup c_1)\backslash(c_1 \cup c_2) = \{\ell\}$ (BckRsl) and $(\{\neg\ell\} \cup c_2)\backslash(\{\neg\ell\} \cup c_1 \cup c_2) = \emptyset$ (Triv). Otherwise, we define $f$ as the single clause "$\{\ell\} \cup c_1$", and we have $f\backslash(c_1 \cup c_2) = \emptyset$ (Triv).

*Definition 4.2 (Backward Resolution Chain).* Given these two clause derivation rules,

$$\textsc{BckRsl}\ \frac{c_1 \qquad \{\neg\ell\} \cup c_2}{c_2}\ c_1\backslash c_2 = \{\ell\} \qquad\qquad \textsc{Triv}\ \frac{c_1}{c_2}\ c_1\backslash c_2 = \emptyset$$

for $n \geq 1$, we write "$c_1, \ldots, c_n \vdash^{\text{BRC}} c$" **iff** there is a bottom-up derivation – like on the right hand-side – that first iterates BckRsl from $c$ on the list $c_1, \ldots, c_{n-1}$ and then concludes by Triv on $c_n$.

$$\textsc{BckRsl}\ \cfrac{c_1 \qquad \textsc{BckRsl}\ \cfrac{c_{n-1} \qquad \textsc{Triv}\ \cfrac{c_n}{\ldots}}{\ldots}}{c}$$

When "$f \vdash^{\text{BRC}} c$", we say that $f$ is a *Backward Resolution Chain* (BRC) of $c$.

The correctness & completeness of the resolution proof system is rephrased by Theorem 4.3.

Theorem 4.3 (Refutation correctness & completeness). *A CNF $f$ is UNSAT **iff** there exists a sequence $c_1, \ldots, c_n$ (with $n \geq 1$) such that*

- *forall $i \in [1, n]$, there exists a list of clauses $f_i \subseteq f \cup \{c_1, \ldots, c_{i-1}\}$ such that $f_i \vdash^{\text{BRC}} c_i$*
- *and, $c_n = \emptyset$*

*Such a sequence $c_1, \ldots, c_n$ is called a* RUP proof *of the unsatisfiability of $f$.*

Now, we sketch how RUP proofs are naturally found by CDCL SAT-solvers, a refinement of DPLL algorithms, at the heart of modern SAT-solvers (see [Silva et al. 2009] for details). The BckRsl rule corresponds to the fact that, under its side-condition, the proposition "$c_1 \wedge \neg c_2$" implies the proposition "$\ell \wedge \neg c_2$", the latter being equivalent to "$\neg(\neg\ell \vee c_2)$". Actually, this corresponds exactly in DPLL SAT-solving to a *unit-propagation* on clause $c_1$ where "$\neg c_2$" represents the assignment of literals before the propagation and "$\neg(\{\neg\ell\} \cup c_2)$" represents the assignment after the propagation. Similarly, Triv corresponds to the fact that, under its side-condition, the proposition "$c_1 \wedge \neg c_2$" is UNSAT. Hence, Triv corresponds exactly to a *conflict* on clause $c_1$ where "$\neg c_2$" represents the current assignment of literals. A CDCL SAT-solver *learns* lemma (under assumption of the input CNF) from conflicts: each of this lemma is actually a clause provable from a BRC involving the input clauses and previously learned clauses. The solver answers "UNSAT", when it has learned the empty clause: the sequence of its learned clauses is then exactly a RUP proof.

*4.2.2 Checking DRUP proofs.* Historically, some CDCL SAT-solvers have dumped full resolution proofs on UNSAT answers (see [Zhang and Malik 2003]). Certifying a resolution proof checker is not too difficult and, in Coq, a first checker has been certified by [Armand et al. 2010]. However,

instrumenting SAT-solvers to output full resolution proofs is very intrusive. Thus, RUP proofs have been proposed as a very lightweight alternative for the design of SAT-solvers [Gelder 2008]. In counterpart, checking RUP proofs requires to recover all BRCs, typically by replaying unit-propagations (RUP is the acronym of "*Reverse Unit Propagation*"). In practice, a RUP-checker does not need all the heuristics of a CDCL SAT-solver, but the data-structures necessary for unit-propagation (e.g. *two-watched literals*).

The DRUP proof format [Heule et al. 2013] is an ASCII file format to describe a RUP proof as a list of clauses, one by line. There are also lines to delete clauses which are no more involved in remaining resolution chains. The standard checker of DRUP proofs in SAT competitions is currently DRAT-TRIM[18] of [Wetzler et al. 2014]. Of course, it also checks DRAT proofs, a conservative extension of DRUP with RAT clauses (detailed at Section 4.4).

Actually, checking DRAT proofs is still a complex task (see [Rebola-Pardo and Cruz-Filipe 2018]) and DRAT-TRIM is an untrusted program written in C. Hence, DRAT-TRIM has been designed to output the full BRC of learned clauses, in an other proof format called LRAT. As indicated by its name, DRAT-TRIM first prunes from the proof (by processing it backward) many learned clauses that are not necessary to derive the empty clause. This reduces a lot the size of LRAT proofs (and of DRAT-TRIM running times).

Then, [Cruz-Filipe et al. 2017a] have developed two certified checkers of LRAT proofs: one certified in Coq and extracted to OCaml; the other certified in ACL2 and extracted to C. As shown in Figure 16 – built from the benchmark table published by Peter Schneider-Kamp on his webpage[11] – their Coq/OCaml version is terribly slow compared to their ACL2/C version.

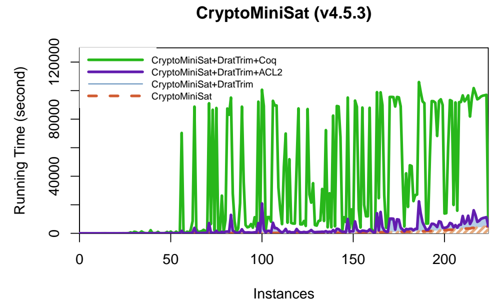


Fig. 16. Benchmark of [Cruz-Filipe et al. 2017a]

Our work illustrates that, by using the Impure library, we can improve the efficiency of the Coq/OCaml implementation, while simplifying substantially the Coq proof.

### 4.3 Verification of (D)RUP proofs in SatAnsCert

This section describes how `unsatProver` introduced at page 17 is implemented by checking the LRAT file generated with DRAT-TRIM from a DRUP proof (itself generated by the SAT-solver, as represented in Figure 13).

*4.3.1 A Shallow-Embedded RUP Checker in Coq.* First, we introduce our shallow embedding of RUP proofs in Coq. In our implementation, besides the type `iclause` of the abstract syntax, we have a more computational representation of clauses, called `cclause`, where a clause is represented as two finite sets of positive integers: one for the positive literals, and one for the negative literals. Such finite sets are efficiently defined in the standard library of Coq using radix trees. For the sake of simplicity, the Coq definitions of our paper omit this type `cclause` and use `iclause` instead.

Given `f:cnf`, we define the type "`consc`$[\![f]\!]$" of clauses that are "*logical consequences*" of $f$. Actually, type `consc` is parametrized by a set of models `s` and constrains its field `rep` to satisfy all models of $s$ (through `rep_sat` property).

[18] Available at https://www.cs.utexas.edu/~marijn/drat-trim

```
Record consc(s:model→Prop): Type :=
  { rep: iclause; rep_sat: ∀ m, s m →⟦snd rep⟧ m }.
```

Then, we define emptiness test of the following type. Actually, `assertEmpty c` terminates iff `(rep c)` is the empty clause. Otherwise, it raises an exception.

```
assertEmpty {s}: consc s → ??(∀ m, ¬(s m)).
```

Checking a Backward Resolution Chain is defined by the following function, called `learn` (it builds a new consequence of the set of models).

```
learn: ∀{s}, list(consc s) → iclause → ??(consc s)
```

It is implemented (for "performance" only) such that if `l` $\vdash^{\mathrm{BRC}}$ `c` then `(learn l c)` returns `c'` where `(rep c')=c`. An exception is raised on an invalid BRC.

*4.3.2 Embedding the verified RUP-Checker into an untrusted LRAT Parser.* The `unsatProver` function needs to parse the LRAT file and to check that it corresponds to a valid RUP proof of the input CNF. It delegates the parsing of the LRAT file to an external *untrusted* OCaml oracle. Moreover, it exploits the cooperation mechanism of Coq and OCaml typechecker in order to make this *untrusted* oracle compute directly "*certified learned clauses*" through a certified API. This API is called a *Logical Consequence Factory* (LCF) and builds correct-by-construction proofs, without an explicit "proof object" – in the style of the old LCF prover [Gordon et al. 1978].

The LRAT parser is declared in Coq by the `rup_lratParse` axiom (see below). This function is parametrized by:

- an abstract type of clause: this type – called `C` – is abstract for the untrusted parser but instantiated by "`consc`⟦$f$⟧" in the Coq proof;
- a logical consequence factory of type "`(rupLCF C)`": this factory allows the oracle to build logical consequences (ie new abstract clauses) with a BRC from existing ones thanks to `rup_learn` (instantiated by the previous `learn` in the Coq proof).[19]
- the input CNF $f$ given as a list of "*axioms*", ie abstract clauses of type `C`.

```
Record rupLCF C :=
  { rup_learn:(list C) → iclause → ?? C;  get_id: C → clause_id }.
Axiom rup_lratParse: ∀ {C}, (rupLCF C)*list(C) → ?? C.
```

By using the `get_id` function, the parser first builds a map from clause identifiers in the DIMACS input to their corresponding abstract clause (ie axiom). Then, it maintains this map while parsing the LRAT file, ie when deleting clauses or adding new learned clauses. On a non-RUP clause or on unexpected issues in the LRAT file, it raises an exception. Otherwise, it eventually returns the abstract clause corresponding to the empty clause.

Thus, `unsatProver` is simply defined by the code below. It first calls the `mkInput` function that builds the parameters expected by the parser (we omit the details here). Afterwards, `unsatProver` simply invokes the parser and checks that its result is the empty clause. Here, the polymorphism over "logical consequences" in the untrusted OCaml parser ensures that this latter cannot forge unsound clauses.

[19]Note that, type `rupLCF` only appear in input of our oracle: it is thus not constrained by permissivity checking. Here, `rup_learn` is declared impure because it may raise exceptions: alternatively, we also could have use an option monad. However, this would probably produce a slightly less efficient extracted code.

```
Definition mkInput (f: cnf): rupLCF(consc⟦f⟧) * list(consc⟦f⟧) ≔..
Definition unsatProver f: ?? (∀ m, ¬⟦f⟧m) ≔
  DO c ⇜ rup_lratParse (mkInput f);; assertEmpty c.
```

The *Polymorphic LCF style* design of our RUP checker has the following benefits w.r.t. the design of the prover found in [Cruz-Filipe et al. 2017b] (a preliminary version of the Coq implementation of the LRAT prover of [Cruz-Filipe et al. 2017a]): BRCs are verified "on-the-fly" in the oracle, and this is much easier to debug; the dictionary mapping *clause identifiers* to *clause values* is only managed by the OCaml oracle (in a efficient hash-table); hence, the deletion of clauses from memory is also only managed by the oracle; the Coq code is thus very simple and very small.

Polymorphic LCF style is also strictly more powerful than standard LCF style, where type abstraction is provided by an abstract type. Indeed, standard LCF style requires to represent each "learned RUP clause" as a sequent of the form "$f \vdash c$" which means that clause $c$ is a logical consequence of CNF $f$. Handling such sequents enables to forbid derivations "$c_1, \dots, c_n \vdash^{\mathrm{BRC}} c$" where some $c_i$ are consequences of two distinct CNFs: otherwise, when called several times during a run, an erroneous oracle could mix consequences of a CNF with consequences of a previous (and maybe contradictory) one. In Polymorphic LCF style, the antecedent $f$ is represented by a type variable: mixing consequences of distinct CNFs is statically forbidden by typechecking. On the contrary, in standard LCF style, this is only prevented by a dynamic check: it is both less simple and less efficient.

### 4.4 Generalization to (D)RAT proofs

A RUP proof can be thought as a sequence of transformations on the input CNF: each learned clause is added to the CNF. These transformations preserves logical equivalence. The motivation of RAT clauses – introduced in [Wetzler et al. 2013] – is to allow transformations which may break logical equivalence but preserve satisfiability. This could dramatically reduce the size of the CNF, and thus the size of its potential UNSAT proof.

*Example 4.4.* Let us define two CNFs $f_1$ and $f_2$ over arbitrary literals $(l_i)_{i\in[1,n]}$ and $(l'_j)_{j\in[1,p]}$ and over a distinct variable $x$:

$$f_1 \;=\; \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{p} (l_i \vee l'_j) \qquad\qquad f_2 \;=\; (\bigwedge_{i=1}^{n} (\neg x \vee l_i)) \wedge \bigwedge_{j=1}^{p} (x \vee l'_j)$$

Whereas $f_1$ has $n \cdot p$ clauses (of two literals), $f_2$ has only $n + p$ clauses (of two literals). These two CNF are equisatisfiable, which is easy to check by rewriting each of them into an equivalent DNF:

$$f_1 \;\Leftrightarrow\; (\bigwedge_{i=1}^{n} l_i) \vee (\bigwedge_{j=1}^{p} l'_j) \qquad\qquad f_2 \;\Leftrightarrow\; (x \wedge \bigwedge_{i=1}^{n} l_i) \vee (\neg x \wedge \bigwedge_{j=1}^{p} l'_j)$$

But, $f_1$ and $f_2$ are generally not equivalent, because $f_2$ constrains $x$ whereas $f_1$ does not.

*4.4.1 Introduction to RAT bunches.* In this section, following [Lammich 2017a], we slightly generalize the definition of RAT clauses of [Cruz-Filipe et al. 2017a] by considering the learning at once of a "bunch" of several RAT clauses on the same pivot. We first need to reintroduce the notion of RUP clause originally defined by [Gelder 2008].

*Definition 4.5 (RUP clause).* Given a CNF $f$ and a clause $c$, we say that "*$c$ is RUP w.r.t $f$*" – and we write $f \vdash^{\mathrm{RUP}} c$ – **iff** one of the two following conditions is verified:

(1) there exists $l$ such that $\{l, \neg l\} \vdash^{\mathrm{BRC}} c$ (ie $c$ is a trivial tautology)
(2) or, there exists $f'$ with $f' \subseteq f$ such that $f' \vdash^{\mathrm{BRC}} c$.

It is obvious that "$f \vdash^{\mathrm{RUP}} c$" implies "$f \Rightarrow c$".

*Definition 4.6 (RAT bunch).* Given two CNFs $f_1$ and $f_2$ and a literal $l$, we say that $f_2$ is a *bunch of RAT clauses w.r.t. $f_1$ for pivot $l$* – and we write $f_1 \vdash_l^{\text{RAT}} f_2$ – **iff** for each clause $c_2 \in f_2$ the two following conditions are satisfied:
(1) $l \in c_2$ ; (2) $f_1 \vdash^{\text{RUP}} (c_1 \backslash \{\neg l\}) \cup c_2$ for each clause $c_1$ of $f_1$.

Lemma 4.7 (SAT preservation of RAT). *Let us assume $f_1 \vdash_l^{\text{RAT}} f_2$ and $[\![f_1]\!]m$. Then, there exists $m'$ such that $[\![f_1 \wedge f_2]\!]m'$.*

Proof. If $[\![f_2]\!]m$ then the property is trivially satisfied for $m' = m$. Otherwise, let $m'$ be the model defined from $m$ by assigning $l$ to true. By condition (1), we have $[\![f_2]\!]m'$. Let $c_2 \in f_2$ such that $\neg[\![c_2]\!]m$. For all $c_1 \in f_1$, from $[\![f_1]\!]m$ and condition (2) we deduce that $[\![(c_1 \backslash \{\neg l\}) \cup c_2]\!]m$, and thus $[\![c_1 \backslash \{\neg l\}]\!]m$, and thus $[\![c_1]\!]m'$. Hence, we have also $[\![f_1]\!]m'$. □

Let us remark that if $c_1 = c_1 \backslash \{\neg l\}$ then condition (2) of Definition 4.6 is trivially satisfied. This leads to introduce the notion of "*basis*" by Definition 4.8 below. Indeed, it suffices to only check condition (2) on clauses $c_1$ that are in the basis of $f_1$ w.r.t. pivot $l$.

*Definition 4.8 (Basis).* Given a CNF $f_1$ and a literal $l$, the *basis of $f_1$ w.r.t. pivot $l$* is defined as the set of clauses in $f_1$ containing $\neg l$.

*Example 4.9 (RAT bunches of Example 4.4).* Clauses of $f_2$ are checked w.r.t $f_1$ in two RAT bunches:
(1) $f_1 \vdash_{\neg x}^{\text{RAT}} \bigwedge_{i=1}^{n}(\neg x \vee l_i)$: checking this RAT bunch is trivial because the basis is empty.
(2) $f_1 \wedge \bigwedge_{i=1}^{n}(\neg x \vee l_i) \vdash_x^{\text{RAT}} \bigwedge_{j=1}^{p}(x \vee l'_j)$: here the basis is $\bigwedge_{i=1}^{n}(\neg x \vee l_i)$. We simply check that for all $(i, j) \in [1, n] \times [1, p]$, we have $(l_i \vee l'_j) \vdash^{\text{BRC}} (l_i \vee x \vee l'_j)$ with $(l_i \vee l'_j) \in f_1$.

From Theorem 4.7, we deduce that if $f_1$ is SAT then $f_1 \wedge f_2$ is also SAT, and finally that $f_2$ is SAT (deleting clauses also trivially preserves satisfiability).

*Example 4.10 (Contradictory RAT bunches).* Given $x$ and $y$ two distinct variables. We check the two following RAT bunches: $x \vdash_{\neg y}^{\text{RAT}} \neg y$ and $x \vdash_y^{\text{RAT}} y$. This check is trivial because the basis is empty in both cases.

This last example shows that two *contradictory* RAT clauses can be learned from the same satisfiable CNF. Hence, "learning" a RAT clause **is not like** "learning" a new lemma: "learning" a RAT clause **is like** adding an axiom which preserves consistency.

*4.4.2 Formalization of RAT bunches.* In the syntax of LRAT files (see [Cruz-Filipe et al. 2017a] for details), each RAT clause comes with a list of BRC, one for each clause of the basis. Note that a valid BRC is at least of length 1. Here, by convention, a BRC of length 0 simply encodes the case (1) of Definition 4.5 (trivial tautology). Moreover, when these lists of BRC share a common prefix, this prefix can be given separately. We reflect these syntactic informations of LRAT files in the following Coq structure: field `clause_to_learn` is the clause to learn, `propag` is the common prefix of the BRC, and `rup_proofs` is the list of suffix of the BRC (one by clause of the basis). Here type `C` represents the type of clauses that are logical consequences of the current CNF (like in Section 4.3.2).

```
Record RatSingle C: Type :=
 { clause_to_learn:iclause; propag:list C; rup_proofs:list(list C) }.
```

Learning a RAT bunch is defined in Coq by the function `learnRat` below. In this function, parameter `s` is the set of models of the current CNF. The bunch is given in field `bunch` of parameter `R` where `pivot` is the pivot and `basis` (resp. `rem` – for remainder) is a list of clauses *containing* (resp. *not containing*) the negation of the pivot. If $f_2$ is the list of clause to learn in `bunch`, then

`learnRat` either returns the CNF "(`basis` ∧ `rem` ∧ $f_2$)" or fails if it cannot prove that the `bunch` is a correct RAT bunch.

```
Record RatInput C: Type :=
 { pivot:literal; rem:list C; basis:list C; bunch:list(RatSingle C) }.
Definition learnRat {s:model→Prop} (R:RatInput(consc s)):??cnf :=...
Lemma learnRat_correct (s: model → Prop) (R: RatInput (consc s)):
  WHEN learnRat R ⇝ f THEN ∀ m, s m → ∃ m', ⟦f⟧ m'.
```

*Example 4.11 (Learning RAT bunches of Example 4.9).* The running example can be turned into two successive formal invocations of `learnRat`:

(1) On the first time, we learn CNF "$f_1 \wedge \bigwedge_{i=1}^{n}(\neg x \vee l_i)$" with the empty basis, with $\bigwedge_{i=1}^{n}(\neg x \vee l_i)$ as the bunch, and with $f_1$ as remainder;
(2) On the second time, we learn CNF "$f_2$" with $\bigwedge_{i=1}^{n}(\neg x \vee l_i)$ as the basis, with $\bigwedge_{j=1}^{p}(x \vee l'_j)$ as the bunch, and with the **empty** remainder.

In the second case, it is formally not necessary to give $f_1$ as the remainder: $f_1$ already appears in the `rup_proofs` field of the bunch. Hence, it is useless to put $f_1$ in the remainder if we aim to delete it from the current CNF just after.

*4.4.3 Formalization of the RAT checker.* In order to define and prove the main loop of `unsatProver` with RAT checking, it is convenient to introduce a generic loop, called `loop_until_None`, dedicated to refutation of unreachability properties. This loop – defined in Figure 17 – iterates a body of type `S → ??(option S)` until to reach a `None` value. This body is assumed to preserve an invariant and to never reach `None` under the assumption of this invariant. Hence, if `None` is finally reached, then the invariant was false in the initial state. The `loop_until_None` loop reuses the `loop` oracle of Figure 7 and is very similar to the generic WHILE-loop.

At last, we extend our untrusted LRAT parser of Section 4.3.2. As discussed on Example 4.10, "learning" a RAT clause replaces the whole CNF by a new one. Thus, our parser learns RUP clauses until it finds a bunch of RAT clauses. Then, it stops, requiring the CNF to be updated. Afterwards, if the RAT bunch is correct, the certified checker restarts the untrusted parser for the updated CNF. This loop runs until the parser finds an empty RUP clause w.r.t. the current CNF. The untrusted parser, called `next_RAT` in Figure 18, behaves as an iterator over RAT bunches. This iterator is expected to return either the empty clause (left case) or a new RAT bunch to learn (right case). The looping process in `unsatProver` is a simple instance of `loop_until_None`: see Figure 18.

### 4.5 Performances & Comparison with other works

Our evaluation of SatAnsCert is split according to SAT and UNSAT answers. Our SAT benchmark – illustrated in Figure 19 – has been established with the CaDiCaL SAT-solver over 120 instances of the SAT competition 2018. Considering the logarithmic scales, the running times of the SAT checker of SatAnsCert in Figure 19 are negligible w.r.t. those of the solver. And, as expected, the running times of our SAT checker are linear w.r.t the size of the input CNF (being given either in number of clauses or in number of literals).

The UNSAT benchmark has been established by using two different solvers: CaDiCaL (sc18) which generates only RUP clauses and CryptoMiniSat (v4.5.3) which produces both RUP and RAT clauses. It is based on more than 170 instances from the SAT competition 2015, 2016 and 2018. Figure 20 represents – for each tested instance – the contribution of each tool in the running time, by cumulating their runtimes on upward ordinates. Along the abscissia axis, the instances are ordered by running times of the SAT-solver. By comparing the overhead of the Coq checkers w.r.t DRAT-trim in Figure 16 and in Figure 20, we see that our LRAT checker is much faster

```
Let luni {S} (body: S → ??(option S)) (I: S → Prop) :=
  ∀ s, I s → WHEN (body s) ⇝ s'
             THEN match s' with Some s1 ⇒ I s1 | None ⇒ False end.
Program Definition loop_until_None{S} body (I:S→Prop|luni body I) s0
  : ?? ¬(I s0)
  := loop (A:={s | I s0 → I s})
       (s0, fun s ⇒
            DO s' ⟵ mk_annot (body s) ;;
            match s' with
            | Some s1 ⇒ RET (inl (A:={s | I s0 → I s }) s1)
            | None ⇒ RET (inr (B:=¬(I s0)) _)
            end).
```

Fig. 17. A Generic Loop to Refute Unreachability Properties

```
Axiom next_RAT: ∀ {C}, (rupLCF C) * (list C) → ??(C + RatInput C).
Program Definition unsatProver: ∀ (f:cnf), ?? ¬(∃ m, ⟦f⟧m) :=
  loop_until_None
     (fun f ⇒ (* loop body *)
      DO step ⟵ next_RAT (mkInput f) ;;
      match step with
      | inl c ⇒
         assertEmpty (rep c);;
         RET None
      | inr ri ⇒ (* build a new CNF from the RAT bunch *)
         DO f' ⟵ learnRat ri;;
         RET (Some f')
       end)
     (fun f ⇒ ∃ m, ⟦f⟧m). (* loop invariant *)
```

Fig. 18. The RAT prover of SatAnsCert

than the Coq/OCaml checker of [Cruz-Filipe et al. 2017a] which has inspired it. We believe that our lightweight design, based on parametric reasoning has a significant impact on performances here (and it makes the formal proof much more simpler). As also shown in Figure 20, our LRAT checker is most often slower than the ACL2/C checker of [Cruz-Filipe et al. 2017a]. We could probably significantly improve the performance of SatAnsCert, by encoding literals with native integers instead of Coq positives (aka lists of bits), and by encoding clauses with native persistent arrays instead of radix-trees. These native data-structures were experimentally introduced in Coq by [Armand et al. 2010] and had a positive impact on their resolution checker. Currently, they have however still an experimental status in Coq.

The GRAT toolchain [Lammich 2017b] is an alternative for certified checking of DRAT files. As the DRAT-trim toolchain, it takes a CNF in DIMACS format and a DRAT file in input, generate some intermediate files through an untrusted C++ tool, and gives a certified answer from this intermediate files thanks to an Isabelle/MLton checker. According to [Lammich 2017a], the GRAT toolchain is faster than the DRAT-trim one. Because SatAnsCert is itself based on DRAT-trim, we did not find very significant to compare it experimentally to the GRAT toolchain.

In conclusion, SatAnsCert is not the most optimized DRAT checker. But the bottleneck of running times in our UNSAT checking is DRAT-trim (the standard checker in SAT competitions).

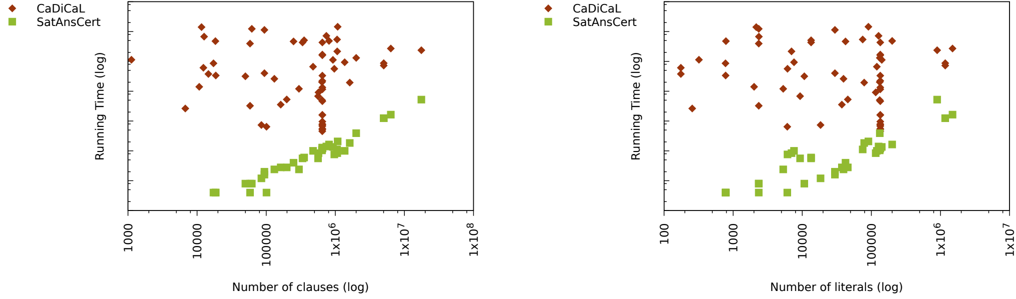


Fig. 19. Our SAT benchmark based on the CaDiCaL (sc18) SAT-solver

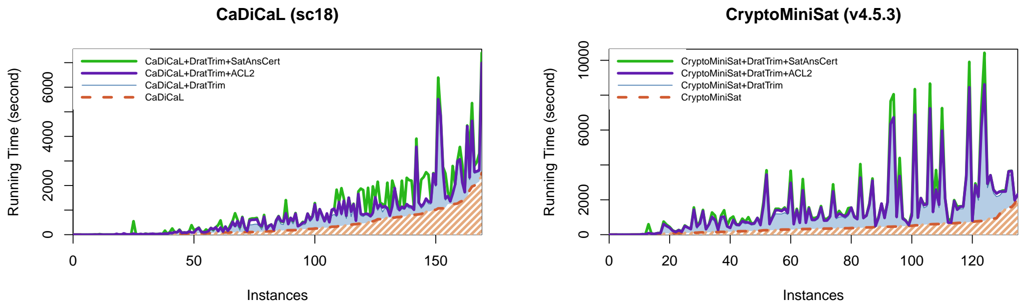


Fig. 20. Our UNSAT benchmarks

Indeed, on average of the UNSAT benchmark depicted in Figure 20, the solver takes 30% of the running time, DRAT-trim takes 50%, and our certified LRAT checker takes the 20% remaining. This demonstrates that SatAnsCert reasonably scales up on state-of-the-art SAT-solvers. One of our most noticeable achievement is that SatAnsCert only results from a modest effort: we evaluate the whole development at 2 person.months for 1Kloc of Coq (including all proof scripts) and 1Kloc of OCaml files (including `.mll` files). These figures exclude the development of the Impure library itself.

## 5 CONCLUSION AND FUTURE WORKS

This paper proposes a new FFI to embed OCaml code into Coq verified code. It illustrates its application to formal but lightweight reasonings about imperative functions. This FFI is based on may-return monads, originally introduced for the first version of the VPL (Verified Polyhedra Library) [Fouilhé and Boulmé 2014]. In this first version, each oracle of the VPL generated some terms (in a given abstract syntax), which were interpreted by the Coq certified frontend as monotonic transformations over convex polyhedra. Then, it appeared that the *deep embedding* of these monotonic transformations could be advantageously replaced by a *shallow embedding*. Hence, the VPL has been reimplemented [Maréchal 2017] with *Polymorphic LCF style* oracles: the oracles perform directly monotonic transformations through a certified API using polymorphism for abstracting types. This style resulted in a significant reduction of both Coq and OCaml code size. Moreover, the oracles were much more easier to debug. Finally, it was understood that Polymorphic LCF style exploits a kind of "*theorem for free*" corresponding to parametric reasonings with invariants. With respect to these previous works, our contribution in this work is to have extracted the Impure

library from the sources of the VPL and to have applied it to other contexts than convex polyhedra. This may contribute to convince other Coq users of the interest of this approach.

The theoretical foundations of this approach still remain to be investigated: our soundness conjecture needs to be formalized and proved, while permissivity checking needs to be formally defined and implemented. Moreover, as discussed in Appendix A, extending the approach to reasonings about program equivalences would also probably require to modify the extraction mechanism itself.

## REFERENCES


Amal J. Ahmed, Andrew W. Appel, and Roberto Virga. 2002. A Stratified Semantics of General References Embeddable in Higher-Order Logic. In *Symposium on Logic in Computer Science (LICS)*. IEEE, 75.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A Very Modal Model of a Modern, Major, General Type System. In *Principles of Programming Languages (POPL)*. ACM Press, 109–122.

Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. 2010. Extending Coq with Imperative Features and Its Application to SAT Verification. In *Interactive Theorem Proving (ITP) (LNCS)*, Vol. 6172. Springer, 83–98.

Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. 2012. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS2*. arXiv:hal-00653367

Jean-Philippe Bernardy and Guilhem Moulin. 2012. A Computational Interpretation of Parametricity. In *Symposium on Logic in Computer Science (LICS)*. IEEE Computer Society.

Jean-Philippe Bernardy and Guilhem Moulin. 2013. Type-theory in color. In *International Conference on Functional programming (ICFP)*. ACM Press.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke Models over Recursive Worlds. In *Principles of Programming Languages (POPL)*. ACM Press, 119–132.

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. 2018. Ready, Set, Verify! Applying Hs-to-coq to Real-world Haskell Code (Experience Report). *Proc. ACM Program. Lang.* 2, ICFP, Article 89 (July 2018), 16 pages. https://doi.org/10.1145/3236784

Arthur Charguéraud. 2011. Characteristic formulae for the verification of imperative programs. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. ACM, 418–430. https://doi.org/10.1145/2034773.2034828

Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. 2017a. Efficient Certified RAT Verification. In *CADE (LNCS)*, Vol. 10395. Springer, 220–236.

Luís Cruz-Filipe, Joao Marques-Silva, and Peter Schneider-Kamp. 2017b. Efficient Certified Resolution Proof Checking. In *Tools and Algorihtms for the Construction and Analysis of Systems (TACAS) (LNCS)*, Vol. 10205. Springer, 118–135.

Alexis Fouilhé and Sylvain Boulmé. 2014. A Certifying Frontend for (Sub)Polyhedral Abstract Domains. In *Verified Software: Theories, Tools, Experiments (VSTTE) (LNCS)*, Vol. 8471. Springer, 200–215.

Allen Van Gelder. 2008. Verifying RUP Proofs of Propositional Unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics*.

Evguenii I. Goldberg and Yakov Novikov. 2003. Verification of Proofs of Unsatisfiability for CNF Formulas. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*. IEEE Computer Society, 10886–10891. https://doi.org/10.1109/DATE.2003.10008

Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. 1978. A Metalanguage for Interactive Proof in LCF. In *Principles of Programming Languages (POPL)*. ACM Press, 119–130.

Marijn Heule, Warren A. Hunt Jr., and Nathan Wetzler. 2013. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 181–188. http://ieeexplore.ieee.org/document/6679408/

Marijn J. H. Heule. 2016. The DRAT format and DRAT-trim checker. *CoRR* abs/1610.06229 (2016). arXiv:1610.06229 http://arxiv.org/abs/1610.06229

Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. 2017. Short Proofs Without New Variables. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings (Lecture Notes in Computer Science)*, Vol. 10395. Springer, 130–147. https://doi.org/10.1007/978-3-319-63046-5_9

Aquinas Hobor, Robert Dockins, and Andrew W. Appel. 2010. A Theory of Indirection via Approximation. In *Principles of Programming Languages (POPL)*. ACM Press, 171–184.

Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 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*. 3AF,

SEE, SIE, Toulouse, France, 1–9. https://hal.inria.fr/hal-01643290

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

Peter Lammich. 2017a. Efficient Verified (UN)SAT Certificate Checking. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings (Lecture Notes in Computer Science)*, Vol. 10395. Springer, 237–254. https://doi.org/10.1007/978-3-319-63046-5_15

Peter Lammich. 2017b. The GRAT Tool Chain - Efficient (UN)SAT Certificate Checking with Formal Correctness Guarantees. In *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings (Lecture Notes in Computer Science)*, Vol. 10491. Springer, 457–463. https://doi.org/10.1007/978-3-319-66263-3_29

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009). arXiv:inria-00415861

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://xavierleroy.org/publi/compcert-backend.pdf

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Computability in Europe (CiE) (LNCS)*, Vol. 5028. Springer, 359–369.

Alexandre Maréchal. 2017. *New Algorithmics for Polyhedral Calculus via Parametric Linear Programming.* Ph.D. Dissertation. Université Grenoble Alpes. https://hal.archives-ouvertes.fr/tel-01695086

Adrian Rebola-Pardo and Luís Cruz-Filipe. 2018. Complete and Efficient DRAT Proof Checking. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018.* IEEE, 1–9. https://doi.org/10.23919/FMCAD.2018.8602993

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

Silvain Rideau and Xavier Leroy. 2010. Validating register allocation and spilling. In *Compiler Construction (CC 2010) (LNCS)*, Vol. 6011. Springer, 224–243.

João P. Marques Silva, Inês Lynce, and Sharad Malik. 2009. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press, 131–153.

Cyril Six, Sylvain Boulmé, and David Monniaux. 2019. Certified Compiler Backends for VLIW Processors Highly Modular Postpass-Scheduling in the CompCert Certified Compiler. (July 2019). https://hal.archives-ouvertes.fr/hal-02185883 preprint.

Dimitrios Vytiniotis and Stephanie Weirich. 2007. Free Theorems and Runtime Type Representations. *Electronic Notes in Theoretical Computer Science* 173 (2007), 357–373.

Philip Wadler. 1989. Theorems for Free!. In *Functional Programming Languages and Computer Architecture (FPCA).* ACM Press, 347–359.

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming (LNCS)*, Vol. 925. Springer.

Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. 2013. Mechanical Verification of SAT Refutations with Extended Resolution. In *Interactive Theorem Proving (ITP) (LNCS)*, Vol. 7998. Springer, 229–244.

Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Theory and Applications of Satisfiability Testing (SAT) (LNCS)*, Vol. 8561. Springer, 422–429.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)).* ACM Press, 283–294.

Lintao Zhang and Sharad Malik. 2003. Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany.* IEEE Computer Society, 10880–10885. https://doi.org/10.1109/DATE.2003.10014

## A THE ISSUE OF EQUALITY ON IMPURE COMPUTATIONS

When interpreting formal proofs based on the IMPURE library, the user must be aware that only WLP-theorems (defined in Section 2.1.2) have a meaning on the extracted code. In particular, the meaning of COQ equality on impure computations is currently very counter-intuitive as explained just now.

In the COQ logic, all reduction strategies are equivalent. In particular, for any term `foo` the COQ logic cannot distinguish between the two following $\beta$-convertible terms

`((fun x (_:unit) ⇒ x) foo)` versus `(fun (_:unit) ⇒ foo)`

But in OCAML, the two following expressions are very different

`((fun x (_:unit) -> x) (print_string "hello"))`

versus `(fun (_:unit) -> print_string "hello")`

The first expression prints "hello" whereas the second one is silent. This corresponds to the call-by-value semantics of OCAML.

Let us use this idea to build a counter-intuitive COQ theorem. Consider the following code, that defines the `repeat` operator, a higher-order iterator repeating `n` times a computation `k`. It is applied in `print3` to print a string three times.

```
Fixpoint repeat (n:nat) (k: unit → ?? unit): ?? unit :=
  match n with
  | 0 ⇒ RET()
  | S p ⇒ k();; repeat p k
  end.
Definition print3 (s:pstring):?? unit:= repeat 3 (fun _ ⇒ println s).
```

A careless user could instead provide the wrong implementation below, where `wprint3` prints the string only once. Actually, the careful user will have in mind that the parameter `k:??unit` of `wrepeat` is extracted to `k:unit` in OCAML. Thus, at extraction, `k` is `()` – the single value of type `unit`.

```
Fixpoint wrepeat (n:nat) (k: ?? unit): ?? unit :=
  match n with
  | 0 ⇒ RET()
  | S p ⇒ k;; wrepeat p k
  end.
Definition wprint3 (s:pstring): ?? unit := wrepeat 3 (println s).
```

Unfortunately, for the COQ logic, `print3` and `wprint3` are the same as attested by the following lemma.

```
Lemma wrong_IO_reasoning s: (print3 s)=(wprint3 s).
```

In order to avoid this counter-intuitive meaning of equality, we could use an alternative extraction, based on the *deferred* monad below, instead of the identity monad:

$$??A \triangleq \text{unit} \to A \qquad k \rightsquigarrow a \triangleq k() = a \qquad \text{ret}\, a \triangleq \lambda(), a \qquad k_1 \gg\!= k_2 \triangleq \lambda(), k_2\,(k_1())\,()$$

The extraction on the deferred monad is consistent with COQ equality, but it slows down the computations at runtime (and makes the type of OCAML oracles more heavyweight).

A better solution consists in keeping the extraction on the identity monad as much as possible, by building a type-system to detect COQ terms that are wrongly extracted in the identity monad (like `wrepeat` above) and extract them with the deferred monad instead. This feature requires a

non-trivial type-system and a non-trivial modification of the extraction, and is out of the scope of this paper.

However, we conjecture that this counter-intuitive equality cannot lead to wrong WLP-theorems, even for the extraction on the identity monad without restriction. In other words, we conjecture that while the results observed at runtime in the deferred monad or in the identity monad can differ, WLP-theorems can only state properties which are satisfied in both extractions.

## B THE ISSUES OF CYCLIC VALUES

Consider the following Coq code. It defines a type `empty` which is provably empty: the proposition `empty → False` is provable by induction. Thus, any function of `unit→?? empty` is proved to never return (normally).

```
Inductive empty: Type:= Absurd: empty → empty.
Lemma never_return_empty (f:unit→??empty): WHEN f() ⇝ _ THEN False.
```

Thus, `unit→?? empty` is not permissive in presence of OCaml cyclic values like the `loop` value defined below (with type `empty`).

```
let rec loop = Absurd loop
let f: unit -> empty = fun () -> loop
```

Besides this pathological case, forbidding cyclic values on Coq extracted types is also necessary for the soundness of the physical equality inside Coq introduced at Section 2.4. Indeed, otherwise there is an unsoundness issue with axiom `phys_eq_true`.

For example, let us consider the `phys_eq_pred` lemma about type `nat` of Peano's natural number, defined in the standard library. This lemma derives from the fact that `0` is the only `n:nat` such that `pred n = n`.

```
Definition is_zero (n:nat): bool :=
  match n with
  | O ⇒ true
  | (S _) ⇒ false
  end.
Lemma phys_eq_pred n:
  WHEN phys_eq (pred n) n ⇝ b THEN b=true → (is_zero n)=true.
```

Let us now consider the following cyclic value – called `fuel` – because some Coq users define such an "infinite fuel" in order to circumvent the structural recursion imposed by Coq.

```
let rec fuel: nat = S fuel
```

At runtime, the OCaml test "`pred fuel == fuel`" returns **`true`**, but "`is_zero fuel`" returns **`false`**. This contradicts the `phys_eq_pred` lemma. Hence, in order to formally reason about physical equality in Coq, it is necessary to forbid – in OCaml oracles – cyclic values on types extracted from Coq.

In conclusion, Definition 2.3 forbids oracles to define cyclic values on Coq extracted types. A way to check this property of oracles would consist in adding to the OCaml language an (optional) "`inductive`" tag on OCaml variant types that forbids cyclic values of these types. Then, Coq inductive types would be extracted on OCaml variant types tagged with "`inductive`".

## C MIXING COQ INVARIANTS AND ALIASES

This section illustrates interactions between aliases and Coq typing with examples using type `cref` defined in Figure 4 page 8 (for the implementation of the oracle given in Figure 5). First, we introduce the following Coq code:

```
Definition may_alias{A} (x:cref A) (y:cref nat):?? A:=
  y.(set) 0;; x.(get) ().
```

Now, let us consider `x: cref mydata` where `mydata` is constrained by invariant `bounded`. We are able to prove that `(may_alias x y)` returns a value satisfying this invariant as expressed by `mydata_preserved` lemma below:

```
Record mydata := { value: nat; bounded: value > 10 }.
Lemma mydata_preserved (x: cref mydata) (y: cref nat):
  WHEN may_alias x y ⇝ v THEN v.(value) > 10.
```

Let us remark that `mydata_preserved` property could be *broken* by extending the extracted code with arbitrary OCaml code (even for safe OCaml code). Indeed, in the extracted code, type `mydata` is extracted to `nat` (because `mydata` is a record type with a single field that is not a proposition). And, given any "`x:cref nat`", `(may_alias x x)` returns `0` (while changing the contents of `x` for this value). Actually, Conjecture 2.4 states that such an alias cannot break WLP-theorems proven in Coq if we consider only on the extracted code (linked to the oracle for `make_cref`). Informally, this is because the typing discipline of Coq itself forbids any alias that breaks Coq typing: in the Coq code, aliasing references of `(cref mydata)` with references of `(cref nat)` is forbidden.

However, this does not forbid the presence of all aliases in the Coq code itself. For example, the code below defines a reference `r2` containing a reference `r1`, and run `(may_alias r2 r1)` which thus changes the contents of the contents of `r2`.

```
Program Definition alias_example (r1: cref nat) : ?? { r | r=r1 } :=
  DO r2 ⇜ make_cref (exist (fun r ⇒ r = r1) r1 _);; may_alias r2 r1.
```

Here, through Coq typing, we also formally prove that the result of `(may_alias r2 r1)` is reference `r1`. But, the fact that `r1` contains `0` at the end cannot be formally proven (it depends on `make_cref` implementation).

The preceding example illustrates that extending extracted code with an OCaml main function could in theory break some properties proved on the Coq side. It seems thus important to define the main function of executables on the Coq side.

Moreover, the `cref` example illustrates that permissitivity checking is a bit more complex than the sketch of Section 2.2.1. In particular, the parameter `A` of type `cref` is both used in input (on `set`) and on output (on `get`). Thus, type `(cref nat)→??nat` and `nat → ??(cref nat)` are permissive, because type `nat` of Coq coincides with its OCaml extraction (in particular, because of the restriction on cyclic-values, see Appendix B). But `(cref mydata)→??nat` and `nat → ??(cref mydata)` are not permissive, because type `mydata` of Coq does not coincide with its extraction.

## D FORMAL REASONING ABOUT EXCEPTION HANDLERS IN IMPURE

This section presents how to derive a WLP property about the "`try_with_any`" operator of Section 3.1. Below, we define the following wrapper that requires from the user an additional *post-condition* `P` satisfied by both branches of the exception handler.

```
Definition is_try_post {A} (P: A → Prop) k1 k2: Prop :=
```

```
  wlp (k1 ()) P ∧ ∀ (e:exn), wlp (k2 e) P.
Program Definition try_catch_ensure
  {A} k1 k2 (P:A→Prop|is_try_post P k1 k2): ?? { r | P r } :=
  TRY DO r ⇜ mk_annot (k1 ());; RET (exist P r _)
  WITH_ANY e ⇒ DO r ⇜ mk_annot (k2 e);; RET (exist P r _).
```

Providing the following notation

```
Notation "'TRY' k1 'CATCH' e '⇒' k2 'ENSURE' P" :=
  (try_catch_ensure (fun _ ⇒ k1) (fun e ⇒ k2) (exist _ P _))...
```

This operator is illustrated in the following simple example which generates an (easy) proof obligation from the user to discharge the prove the `is_try_post` property.

```
Program Example tryex {A} (x y:A) :=
  TRY (RET x) CATCH _ ⇒ (RET y) ENSURE (fun r ⇒ r = x ∨ r = y).
```

Then, we can easily proves consequences of this postcondition as illustrated below.

```
Program Example tryex {A} (x y:A):
  WHEN tryex x y ⇝ r THEN `r <> x → `r = y.
```

Let us remark that on the above example, we cannot formalize the informal reasoning that `(tryex x y)` necessarily returns `x`. Indeed, our untrusted implementation of `try_with_any` could contain a bug while remaining sound w.r.t the formal declaration in Coq. In particular, for the following buggy implementation, `(tryex x y)` necessarily returns `y`.

```
let try_with_any (k1, k2) = try k2 (ImpureFail "") with _ -> k1()
```

More generally, except if we can prove that a given branch of a "`TRY`" cannot return normally like in "`TRY (FAILWITH "..") ..`", we can never formally prove which branch has returned. In other words, "`TRY`" should be considered formally as a non-deterministic operator. If this weakness becomes an issue, it is still possible to use option types instead of exceptions. In counterpart, these "formally weak" exceptions provide a nice feature: the formal specifications of functions have never to declare which exceptions may be raised or not.

## E INSTANTIATING GENERIC LOOPS AND FIXPOINTS OF SECTION 3

Figure 21 illustrates how to instantiate the while-loop operator of Figure 7 (page 12) to an iterative computation of Fibonacci's numbers. Figure 22 instantiates the fixpoint of Section 3.3 on a naive recursive computation of Fibonacci's numbers: given any correct `beqZ: Z -> Z -> ?? bool`, it derives a correct Fibonacci's implementation `fib`. The last paragraph of Section 3.3 explains how the definition of `beqZ` may impact the performance of `fib` according to the implementation of the `fixp` oracle.

```
(* Specification of Fibonacci's numbers by a relation *)
Inductive isfib: Z → Z → Prop :=
 | isfib_base p: p ≤ 2 → isfib p 1
 | isfib_rec p n1 n2:
     isfib p n1 → isfib (p+1) n2 → isfib (p+2) (n1+n2).

(* Internal state of the iterative computation *)
Record iterfib_state := { index: Z; current: Z; old: Z }.

Program Definition iterfib (p:Z): ?? Z :=
  if p ≤? 2
  then RET 1
  else
    DO s ⇜
      while (fun s ⇒ s.(index) <? p)
            (fun s ⇒ RET {| index := s.(index)+1;
                            current := s.(old) + s.(current);
                            old:= s.(current) |})
            (fun s ⇒ s.(index) ≤ p
                     ∧ isfib s.(index) s.(current)
                     ∧ isfib (s.(index)-1) s.(old))
            {| index := 3; current := 2; old := 1 |};;
    RET (s.(current)).

(* Correctness of the iterative computation *)
Lemma iterfib_correct p: WHEN iterfib p ⇝ r THEN isfib p r.
```

Fig. 21. Iterative computation of Fibonacci's numbers with the WHILE-loop

```
Parameter beqZ: Z → Z → ?? bool.
Parameter beqZ_correct: ∀ x y, WHEN beq x y ⇝ b THEN b=true → x=y.

Program Definition fib (z: Z): ?? Z :=
  DO f ⇜ rec beqZ isfib (fun (fib: Z → ?? Z) p ⇒
    if p ≤? 2
    then RET 1
    else
      let prev := p-1 in
      DO r1 ⇜ fib prev ;;
      DO r2 ⇜ fib (prev-1) ;;
      RET (r2+r1)) _;;
  (f z).

Lemma fib_correct (x: Z): WHEN fib x ⇝ y THEN isfib x y.
```

Fig. 22. Computation of Fibonacci's numbers with the generic fixpoint