Lightweight proof by reflection using a posteriori simulation of effectful computation

Guillaume Claret[1], Lourdes del Carmen González Huesca[1],
Yann Régis-Gianas[1], and Beta Ziliani[2]

[1] PPS, team πr^2 (University Paris Diderot, CNRS, and INRIA)
`{guillaume.claret,lgonzale,yann.regis-gianas}@pps.univ-paris-diderot.fr`
[2] Max Planck Institute for Software Systems (MPI-SWS)
`beta@mpi-sws.org`

Abstract. Proof-by-reflection is a well-established technique that employs decision procedures to reduce the size of proof-terms. Currently, decision procedures can be written either in Type Theory—in a purely functional way that also ensures termination— or in an effectful programming language, where they are used as oracles for the certified checker. The first option offers strong correctness guarantees, while the second one permits more efficient implementations.
We propose a novel technique for proof-by-reflection that marries, in Type Theory, an effectful language with (partial) proofs of correctness. The key to our approach is to use *simulable* monads, where a monad is simulable if, for all terminating reduction sequences in its equivalent effectful computational model, there exists a witness from which the same reduction may be simulated *a posteriori* by the monad. We encode several examples using simulable monads and demonstrate the advantages of the technique over previous approaches.

1 Introduction

In Type Theory, types may embed computation, thereby allowing for a proof technique called *proof by reflection*. This technique reduces the time to typecheck a proof by replacing potentially large proof-terms by small proof-terms, whose verification consists of computing at the type level.

For instance, say that verifying a proof Δ of $\mathsf{P}\,\mathsf{a}$ is computationally expensive, for $\mathsf{P} : \mathsf{A} \to \mathsf{Prop}$, with A a type, and $\mathsf{a} : \mathsf{A}$. Let B be a type such that there exists an *interpretation* function I from B to A, and a decision procedure $\mathsf{D} : \mathsf{B} \to \mathsf{bool}$. Furthermore, let us assume that D *decides* P, that is, there is a theorem

$$\mathsf{sound} \ : \ \forall \mathsf{x} : \mathsf{B}, \mathsf{D}\,\mathsf{x} = \mathsf{true} \to \mathsf{P}\,(\mathsf{I}\,\mathsf{x})$$

which states that for every element x of B, if the decision procedure returns true for this element, then property P holds for the interpretation of x. With these definitions at hand, then if we have some $\mathsf{b} : \mathsf{B}$ such that $\mathsf{I}\,\mathsf{b} = \mathsf{a}$, we can replace the original proof-term Δ with

$$\mathsf{sound}\,\mathsf{b}\,(\mathsf{refl\_equal}\,\mathsf{true})$$

where $\mathsf{refl\_equal}$ has type $\forall \mathsf{x} : \mathsf{bool}, \mathsf{x} = \mathsf{x}$. Typechecking that the proof-term above has the expected type $(\mathsf{P}\ \mathsf{a})$ effectively amounts to (i) executing the procedure D on b, (ii) checking that its result is equal to true, (iii) and checking that the interpretation of b is equal to a.[3]

Previous works [11,4] have exposed several advantages and weaknesses of proof by reflection, especially in comparison with the traditional LCF proof style [9]. In a nutshell, the former is considered more robust to change, while the latter is easier to write. Indeed, proving by reflection has a price: the decision procedure D must usually be written in a constrained programming language with only total functions and no imperative features. Furthermore, soundness proofs are often complex and thus difficult to construct [8]. These two problems, the restricted language and the need for keeping the proof of soundness simple, incite the proof developer to write inefficient decision procedures, which is regrettable since proof search is intrinsically a computationally expansive process.

There is a variation of proof by reflection that alleviates some of these problems, called *certifying* proof by reflection [3,10]. In this technique, the decision procedure is written in a general purpose programming language, and used by the proof assistant as an *untrusted oracle*. The decision procedure returns a certificate, which is mechanically verified by the proof assistant *via* a certificate checker written in Type Theory. This checker and its proof of correctness are usually kept simple, whereas the untrusted oracle can be as sophisticated as necessary to implement the decision procedure efficiently. However, this technique has its drawbacks. First, it is not as efficient as one may expect, as the certificate embedded in the resulting proof-term can be large and, in addition, there is a cost of executing the oracle, plus verifying the certificate with the checker. Second, the proof developer is forced to write the certificate checker *and* the decision procedure (or adapt an existing one in order to produce the certificate). Third, the implementation of an oracle usually gives only weak guarantees about its applicability (a perfectly valid but useless oracle could fail on every input) because proving completeness properties about a program written in a general purpose programming language is notoriously hard.

In this paper, we propose a novel style of proof by reflection that allows for writing an *efficient* decision procedure *in Type Theory*. Our idea is to use an (untrusted) compiled version of a monadic decision procedure written in Type Theory as an efficient oracle for itself. Like in the certifying style, the decision procedure is developed within an effectful language and used as an oracle by the theorem prover. However, unlike in the certifying style, the decision procedure is written in Type Theory, in a language extended with monads as commonly found in Haskell programs [16]. In this way, programmers have a full set of effects at their hand (references, exceptions, non-termination), together with dependent types to enforce (*partial*) correctness. This decision procedure is then automatically compiled into an impure programming language with an efficient computational model. This compiled code is executed, and a small piece

[3] Usually there is also a previous step where a b is constructed for the given a. This step is called *reification* in the literature.

of information is collected to efficiently *simulate* this execution *in Type Theory* using the initial monadic decision procedure.

To formalize this idea we define the concept of *a posteriori* simulation of effectful computations in Type Theory. Roughly speaking, it involves determining, for a computation C encapsulated in the monadic type $\mathsf{M}\,\mathsf{A}$, the conditions for which there exists a piece of information p such that the evaluation of C, using p, can witness an inhabitant of type A.

We believe this technique to be more lightweight than existing approaches because neither a full proof of correctness nor a certificate checker is required to execute a decision procedure once it is written in our monad.

To sum things up, our contributions are (i) a technique to perform *a posteriori* simulations of effectful computations in Type Theory in order to promote these computations as genuine proofs by reflection; (ii) an informal discussion of different simulable effects; (iii) a plugin[4] for the Coq proof assistant, which enables the effectful computation as an interactive decision procedure of a Coq function written in monadic style; (iv) several examples of proofs by reflection in this new style, showing its simplicity and efficiency.

2 Simulation-based proof by reflection

In this section we give an informal presentation of the simulation-based style of proof by reflection, in Coq. As a running example, we consider the problem of determining if a conjunction of inequalities $\bigwedge_{i \in I} \mathsf{A}_i \leqslant \mathsf{B}_i$ between ground terms of type T logically implies $\mathsf{A} \leqslant \mathsf{B}$ by transitivity, for some A and B. A simple decision procedure for this problem boils down to a depth-first traversal of the graph induced by the hypotheses. In the following procedure implemented in pseudo-code, infinite loops are avoided by marking all of the visited terms:

```
decide ($\bigwedge_{i \in I} \mathsf{A}_i \leqslant \mathsf{B}_i \Rightarrow \mathsf{A} \leqslant \mathsf{B}$) =
  let rec traverse : $\mathsf{T} \to \mathsf{bool} = \lambda \mathsf{C}$
    if $\mathsf{C} = \mathsf{B}$ then $\top$
    else if marked $\mathsf{C}$ then $\bot$
    else
      mark $\mathsf{C}$;
      choice $\mathsf{D}$ s.t. $\exists j, \mathsf{C} \leqslant \mathsf{D} \equiv \mathsf{A}_j \leqslant \mathsf{B}_j \wedge$ traverse $\mathsf{D}$
    in traverse $\mathsf{A}$
```

This procedure cannot be implemented in Coq as it is, for the simple reason that it uses side effects (marks) and is not obviously terminating (of course, it is, but the argument is not syntactical as Coq requires). As mentioned in the introduction, we are going to implement procedures with side effects using a monad $\mathsf{M}\,\Sigma\,\mathsf{T}'$, where Σ represents the type of the state and T' the returning type of the monad. This procedure is then used as an oracle for itself, as we are going to see in the second part of this section.

[4] The plugin and the Coq developments of this paper are downloadable online at http://cybele.gforge.inria.fr.

Here is our encoding of the decide procedure:

```
01 Program Definition decide (f: formula) : M Σ (interpret f) :=
02   let (a, b) := goal f in
03   letrec! traverse x [ interpret_hypotheses f → x ⩽ b) ] :=
04     if x =?= b then return (▻ eq_refl x)
05     else if! marked x then error "Not Found"
06     else do! mark x in
07       choice ( f → x ⩽ b) (successors x (hypotheses f))
08         (λ (s : { y : T & interpret_hypotheses f → x ⩽ y }) ⇒
09            let! Hyb := traverse (π1 s) in
10            return (▻ (λ (hs : interpret_hypotheses f) ⇒ le_trans x (π1 s) b))
11         )
12   in ▻ (traverse a).
```

At high level, the code looks like an ML implementation of the pseudo-algorithm, annotated with dependent types. We are going to explain line by line why this procedure is a faithful representation of the pseudo-code shown above, while introducing the notations used in the rest of the paper.

We start by describing the type formula in line 1. It is a record containing a list of pairs of elements $(\mathsf{A}_i, \mathsf{B}_i)$—the hypotheses—and a pair of elements (A, B)—the goal. When an element f of this type is interpreted using the function interpret, it produces the type $\bigwedge_{i \in I} \mathsf{A}_i \leqslant \mathsf{B}_i \to \mathsf{A} \leqslant \mathsf{B}$. This is the type returned by the monad.

Line 2 is straightforward: it binds the pair of elements being compared in the goal of f to variables a and b. In line 3, the keyword **letrec!** introduces a (potentially nonterminating) recursive function. Behind this syntactic sugar is hidden the application of a dependently-typed general fixpoint operator. The returning type of the local fixpoint traverse is specified between brackets. It returns a proof that the inequality $\mathsf{x} \leqslant \mathsf{b}$ holds under the hypotheses of formula f. As we can see in line 12, the argument x is instantiated with the element a from the goal, therefore effectively proving $\mathsf{a} \leqslant \mathsf{b}$.

In line 4 the current element is compared with b, assuming that the type of the elements, T, has decidable equality. If it is equal, then the reflexivity proof is returned using the standard unit monadic combinator **return** [16]. We defer the explanation of the operator ▻.

In line 5, an error is raised if the element x is already marked. We do not show the implementations of functions mark and is_marked (used in next line), but they are straightforward. In line 6 we mark the element, and in lines 7-11 we try to find a proof of $\mathsf{x} \leqslant \mathsf{b}$ by transitivity, by finding a c such that $\mathsf{x} \leqslant \mathsf{c}$ and $\mathsf{c} \leqslant \mathsf{b}$. For that, we make a list with all the *successors* of x, that is, all c such that $\mathsf{x} \leqslant \mathsf{c}$ is in the list of hypotheses. Then, we call the function choice:

```
Fixpoint choice A {T} (cs : list T) (pred : T → M Σ A) : M Σ A :=
  match cs with
    | nil ⇒ Error "Not found"
    | c :: cs ⇒ try! pred c with _ ⇒ choice A cs pred
    end.
```

The choice operator iterates over a list to find an element c that successfully produces a result using the function pred. At each step of the iteration, the function makes use of the exception mechanism to catch failed attempts and recurse on the tail of the list. Coming back to traverse, in line 9 we call the function recursively using the standard monadic bind operator **let!** $x = e_1$ **in** e_2 [16]. The resulting proof Hyb of $c \leqslant b$ is then used to prove $x \leqslant b$ by transitivity.

Finally, notice that in line 1 we use the standard Coq keyword **Program** [15]. This keyword allows for writing a partial term, where the holes are exposed to the user as proof obligations. In our case, the holes come from type coercions, noted as $\rhd$, and they are solved automatically by Coq.

The compiled decision procedure as an oracle. The type system of Coq will not let us apply decide as it is on some formula f to prove the goal. The reason is simple: an infinite loop would lead to breaking soundness of the prover. Instead, in order for decide to be evaluated, it needs some extra information, which we call *prophecy*. For instance, in our example this extra information is the number of steps that leads to a successful result.

To get this information, we execute a compiled version $\mathcal{C}$(decide) in OCaml, which performs the effectful computation. A central property of the system is that $\mathcal{C}(\cdot)$ maps the effectful computations of the monad in Coq to effectful terms in OCaml, in such a way that a relation of *a posteriori* simulation stands between the compiled term $\mathcal{C}(t)$ and the initial monadic term t. Intuitively, if a compiled term $\mathcal{C}(t)$, with t of type M T,[5] converges to a value v, then the same evaluation can be simulated *a posteriori* in Coq, using some prophecy p. This prophecy *completes* computation t in order to get a term convertible to **return** t' for some term t' of type T. We instrument the compiled code $\mathcal{C}(t)$ to produce the prophecy along its execution.

Coming back to our example, the following is the (slightly beautified) extracted OCaml code of the function decide:

```
let rec fix f x = incr_nbstep (); f (fix f) x

let rec choice cs pred0 = match cs with
  | Nil → failwith "error"
  | Cons (c, cs0) → try pred0 c with _ → choice cs0 pred0

let decide f =
  let (a, b) = goal f in
  let traverse = fix (fun traverse x →
      match O.eq_dec x b with
      | Left → ()
      | Right → if marked x then failwith "error" else (
          mark x;
          choice (successors x (hypothesis f)) (fun s0 → traverse (projT1 s0))
        ))
  in traverse a
```

[5] For presentation purposes we leave out the parameter Σ representing the type of the state.

The compiled program has almost the same shape as the source term except that every term in Prop has been erased and that the primitives of the monad are replaced with combinators defined in OCaml. These combinators implement an effect and also contribute in determining the prophecy. For instance, the fix combinator not only implements a general fixpoint but also stores the number of iterations that are performed by the oracle in a global variable.

Once applied to a specific formula, this compiled function may diverge or fail. In the setting of *interactive* theorem proving, divergence is not an important issue because the user stays in front of the screen waiting for an answer, and he or she can always interrupt the oracle if it takes too much time to respond. In the case of a successful execution of the oracle, a prophecy of type nat is extracted from the final value of the mutable cell incremented by fix.

The final proof-term. The resulting proof-term corresponding to the application of the procedure to some formula f is

$$\mathsf{unit\_witness}\,(\mathsf{decide}\,\mathsf{f})\,p\,(\mathsf{refl\_equal}\,\mathsf{true})$$

where p has type $\mathsf{Prophecy}$ (in this case, a natural number), and for any type T,

$$\begin{aligned}\mathsf{unit\_witness} &: \forall \mathsf{x} : \mathsf{M}\,\mathsf{T}, \mathsf{Prophecy} \to \mathsf{is\_unit}\,\mathsf{x} = \mathsf{true} \to \mathsf{T}\\ \mathsf{is\_unit} &: \mathsf{M}\,\mathsf{T} \to \mathsf{bool}\end{aligned}$$

The execution time of checking that this term has type $\mathsf{interpret}\,\mathsf{f}$ is split between the execution time of typechecking the prophecy p and the weak head normalization of the procedure, using p to guide the reduction. The overall execution time of the proof-by-reflection results from executing the decision procedure in OCaml plus typechecking the final proof-term, which as we just mentioned, essentially consists of executing the decision procedure a second time in Coq.[6] One can wonder if it is not a waste of time to execute the decision procedure twice, but, as it turns out, using the hints in p, the execution time of the simulation can be tremendously reduced in comparison with the execution of the oracle. This optimization is the subject of Section 5.2.

Putting all the pieces together, our plugin performs the following steps when proving a goal with a monadic procedure proc: (1) Translates and compiles proc into OCaml. (2) Executes the compiled code $\mathcal{C}(\mathsf{proc})$ and obtains prophecy p. (3) Builds proof term $\mathsf{unit\_witness}\,\mathsf{proc}\,p\,(\mathsf{refl\_equal}\,\mathsf{true})$. Notice that the proof developer only has to develop the procedure.

3 *A posteriori* simulation of effects

In this section we formalize the principle of *a posteriori* simulation of effectful computations. The interested reader is invited to read the proofs from the companion technical report [6]. In order to promote a clear formalization we will focus only on simply typed λ-calculus, but the results presented are easily

[6] We assume compilation time not to be significant.

extensible to full Type Theory and OCaml, for the pure and impure calculus, respectively. More precisely, we define two languages: λ, a purely functional and strongly normalizing programming language with monadic constructs, and $\lambda_{v,\perp}$, a non-terminating functional programming language. The definition of λ is parameterized by a monad M, which is abstractly specified by a set of requirements. Accordingly, $\lambda_{v,\perp}$ offers impure operators that match the effectful primitives of the monad M.

Conventions. We write $\overrightarrow{e}$ for a sequence $e_1\, e_2\, \ldots\, e_n$ where $n \geqslant 0$. If a function $\mathcal{F}$ is defined over e then we abusively write $\mathcal{F}\,(\overrightarrow{e})$ for the pointwise extension of $\mathcal{F}$ to a sequence of e. For the sake of conciseness, we often omit universal quantifiers in types when they appear in outermost prenex position.

3.1 λ, a purely functional language

The language λ is the simply typed λ-calculus *à la Curry* with constants:

$$\begin{array}{rlcrl} t,\ u & ::= x \mid \lambda x.t \mid t\,t \mid \mathbf{c} & \qquad & \mathsf{T} & ::= \mathsf{T} \to \mathsf{T} \mid \mathsf{C}\,\overrightarrow{\mathsf{T}} \\ \mathbf{c} & ::= \mathsf{unit} \mid \mathsf{bind} \mid \Downarrow \mid \nabla & & \mathsf{C} & ::= \mathsf{M} \mid \mathsf{P} \end{array}$$

Constants include the usual monadic combinators for effects in the spirit of [16]: unit lifts a term of type T as a computation of type $\mathsf{M\,T}$, and bind composes two computations. Effectful primitives of the monad are kept abstract by regrouping them in the syntactic category ∇. The types include functional types and type constructor applications which are assumed well-formed. M and P are the type constructors for monad and prophecy, respectively. We omit the typing rules but they are standard.

The constant $\Downarrow$ and the type constructor P are unusual. The role of $\Downarrow$ is to perform *a posteriori* simulation using a value p of type P produced by the oracle. We read $\Downarrow_p t$ as "the reduced computation of t using the prophecy p". We require the existence of a total order $\leqslant$ over values of type P and a minimal element $\perp$ for this order. A reduced computation is still a computation, so $\Downarrow$ has type $\mathsf{P} \to \mathsf{M\,T} \to \mathsf{M\,T}$.

We are interested in reasoning on $\beta\delta$-convertibility between terms (where the δ-reduction is the unfolding of constant definitions). We write $\star t$ for $\Downarrow_\perp \mathsf{unit}\ t$ and we say that a computation has converged if there exist a prophecy p and a term t' such that $\Downarrow_p t$ is convertible to $\star t'$.

Finally, the standard notion of monad is extended with a mechanism of simulation directed by a prophecy.

Definition 1 (Simulable monad). *A type constructor M is a simulable monad if it is equipped with unit, bind, $\Downarrow$ and an associated type for prophecies P, such that the requirements 1, 2, 3 and 4 are fulfilled.*

Requirement 1 (Standard monadic laws)

$$\begin{aligned} \mathit{bind}\,(\mathit{unit}\ t)\ f &= f\ t \\ \mathit{bind}\,t\,(\lambda x.\mathit{unit}\ x) &= t \\ \mathit{bind}\,(\mathit{bind}\,t_1\,t_2)\,t_3 &= \mathit{bind}\,t_1\,(\lambda x.\mathit{bind}\,(t_2\,x)\,t_3) \end{aligned}$$

Requirement 2 (Reduction)

$$\begin{array}{l}
\forall\ t, p_1, p_2,\ \Downarrow_{p_1} \textsf{\textit{unit}}\ t = \Downarrow_{p_2} \textsf{\textit{unit}}\ t. \\
\forall\ t, u, p_1, p_2,\ p_1 \leqslant p_2 \ \textit{and}\ \Downarrow_{p_1} t = \star u \ \textit{implies that}\ \Downarrow_{p_2} t = \star u. \\
\forall\ p,\ \Downarrow_p \textsf{\textit{bind}}\, t_1\, t_2 = \Downarrow_p \textsf{\textit{bind}}\,(\Downarrow_p t_1)\, t_2
\end{array}$$

3.2 $\boldsymbol{\lambda_{v,\perp}}$, a call-by-value impure functional language

The impure functional and non-terminating language $\lambda_{v,\perp}$ has the same syntax as λ, except that now constants only consist of effectful operators. The language is equipped with an *instrumented* big-step operational semantics for a weak call-by-value reduction strategy. The executions are carried out under environments η assigning closed values v to variables: $\eta ::= \cdot \mid \eta; x \mapsto v,\ v ::= \boldsymbol{c}\overrightarrow{v} \mid (\lambda x.u)\,[\eta]$. Closed values comprise full applications of effectful constants to values and closures. The judgment in this instrumented semantics is $\eta \vdash u \Downarrow_{p \to p'} v$, which is intended to be read as "the execution of a term u under the environment η converges to a value v and computes a prophecy p' from an initial prophecy p". We keep abstract the rules for constants: they will be characterized by the requirement 4.

$$\textsc{R-Var}\ \frac{}{\eta \vdash x \Downarrow_{p \to p} \eta(x)} \qquad\qquad \textsc{R-Lam}\ \frac{}{\eta \vdash \lambda x.u \Downarrow_{p \to p} (\lambda x.u)\,[\eta]}$$

$$\textsc{R-App}\ \frac{\begin{array}{c}\eta \vdash u_1 \Downarrow_{p \to p_1} (\lambda x.u)\,[\eta'] \\ \eta \vdash u_2 \Downarrow_{p_1 \to p_2} v_1 \qquad \eta'; x \mapsto v_1 \vdash u \Downarrow_{p_2 \to p'} v\end{array}}{\eta \vdash u_1\, u_2 \Downarrow_{p \to p'} v}$$

The purpose of the instrumentation of the compiled code is to monotonically refine the prophecy at each step of the computation:

Requirement 3 (Monotonicity of prophecy computation)

$$\forall\ p,\ p',\ \eta \vdash u \Downarrow_{p \to p'} v \ \textit{implies}\ p \leqslant p'.$$

Compilation Now, we define the compilation function $\mathcal{C}(\cdot)$ from λ to $\lambda_{v,\perp}$.

$$\begin{array}{rcl|c}
\mathcal{C}(x) = x & \qquad \mathcal{C}(\mathsf{unit}) = \lambda x.x & & \mathcal{C}(\mathsf{M}\ \mathsf{T}) = \mathcal{C}(\mathsf{T}) \\
\mathcal{C}(\lambda x.t) = \lambda x.\mathcal{C}(t) & \qquad \mathcal{C}(\mathsf{bind}) = \lambda x, y.y\, x & & \mathcal{C}(\mathsf{C}\ \overrightarrow{\mathsf{T}}) = \mathsf{C}\,(\mathcal{C}(\overrightarrow{\mathsf{T}})) \\
\mathcal{C}(t_1\, t_2) = \mathcal{C}(t_1)\, \mathcal{C}(t_2) & \qquad \mathcal{C}(\Downarrow_p) = \text{undefined} & & \mathcal{C}(\mathsf{T}_1 \to \mathsf{T}_2) = \mathcal{C}(\mathsf{T}_1) \to \mathcal{C}(\mathsf{T}_2)
\end{array}$$

The translation replaces the monadic constructs unit and bind with their respective definitions in the identity monad, and converts each effectful primitive of the monad to the corresponding impure construction of $\lambda_{v,\perp}$. The type for prophecies is kept fully abstract to the programmer. As a consequence, only the instrumented compiled code is allowed to generate prophecies. Therefore, the compilation of $\Downarrow_p$ is explicitly *undefined* because this operator cannot appear in a well-typed user-written monadic term.

The compilation of an effectful monadic constant must extend the prophecy in a sufficient way to make the simulation converge.

Requirement 4 (Adequate instrumented compilation) $\forall p_0, \ldots, p_{n+1}, p,$

$$\textit{if} \begin{cases} \forall i, & \eta \vdash \mathcal{C}(t_i) \Downarrow_{p_i \to p_{i+1}} v_i \\ & \eta \vdash \mathcal{C}(\boldsymbol{c}\,(t_0, \ldots, t_n)) \Downarrow_{p_0 \to p} v \end{cases} \quad \textit{then} \;\; \exists u, \Downarrow_p \boldsymbol{c}\,(t_0, \ldots, t_n) = \star u$$

3.3 Examples of simulable monads

The "trace" prophecy. Given a monad M with an underlying effectful computation model specified by a reduction relation, there is always a prophecy to simulate a converging effectful reduction: the reduction chain itself. However, such a naive implementation of prophecies is obviously inefficient.

Non-termination and partiality. The type $\mathsf{nat} \to \mathsf{option}\,\mathsf{T}$ defines an adequate monad to represent non-terminating computations of type T. A general fixpoint operator is defined by induction over the input natural number. If the number of iterations is sufficient then the computation produces a term $\mathsf{Some}\,\mathsf{t}$, otherwise None. For this monad, the natural type for prophecies is nat and the instrumentation only has to compute an over-approximation of the number of iterations for all the fixpoints of the program. Therefore, a single global variable is enough to represent the prophecy.

State. The type $\mathsf{state} \to \mathsf{T} \times \mathsf{state}$ defines an adequate monad to model stateful computations. A state monad is naturally simulable without a need for prophecies because the operations read and write are total. Yet, if the monad also provides an operation ref to dynamically allocate mutable references, it is hard to ensure statically that a given reference belongs to the state. In that case, the state monad has to be composed with the partiality monad and inherit its type for prophecies. Furthermore, the prophecy can also embed the initial state used to evaluate the monadic term: this is an opportunity to import some precomputed results from the oracle (see Section 4).

Non-determinism. The type $\mathsf{list}\,\mathsf{T}$ defines an adequate monad to model nondeterministic computations, which is useful in proof search procedures. An important operator of this monad is choice of type $\mathsf{M}\mathsf{T} \to (\mathsf{T} \to \mathsf{M}\,\mathsf{T}') \to \mathsf{M}\,\mathsf{T}'$, a partial function that picks an arbitrary choice in all the possibilities. If the list is empty, there is no such choice. But, if a computation had converged, there exist a list of choices that leads to a result. An interesting prophecy is exactly this list of choices (see Section 5.2).

3.4 A posteriori simulation

The main theorem states that, if the evaluation of $\mathcal{C}(t)$ converges for some computation t, then there exists a prophecy p to simulate t back in λ.

Theorem 1 (A posteriori simulation). *Let $\cdot \vdash t : M\,T$ a computation which compilation converges to a value, that is $\cdot \vdash \mathcal{C}(t) \Downarrow_{p \to p'} v$ holds. Then there exists a term t' such that $\Downarrow_{p'} t = \star t'$.*

4 Implementation

We provide a plugin for Coq to develop proofs using the method described in this work. The plugin includes (i) a library with the definition of a simulable monad to write effectful decision procedures; (ii) a tactic called coq waiting for a monadic term t of type M T to try to solve a goal T. Behind the scene, the tactic compiles the monadic term into an OCaml program, executes this program and if its execution converged, uses the resulting prophecy to produce a proof-term in Coq.

The formal notion of simulable monad served as a guideline for the implementation: we defined a compilation function from Coq to OCaml as well as a simulable monad in Coq that respect the requirements drawn by our formal study. However, to improve the usability and the efficiency of our tool, some practical aspects of the implementation differ from the formal specification.

4.1 A simulable monad in Coq

Our monad combines[7] a partiality monad, a non-termination monad, a state monad and a printing monad. It is still possible to implement nondeterminism in it, as we will see in the example from Section 5.2. The monad is parametrized by a signature $\mathit{\Sigma}$ to type the memory (see below). Its type definition is:

$$\mathsf{M}\ \mathit{\Sigma}\ \alpha = \mathsf{State.t}\ \mathit{\Sigma} \to (\alpha + \mathsf{string}) \times \mathsf{State.t}\ \mathit{\Sigma}$$

The monad takes a state and returns a new state plus a value of type α if the computation is successful, or an error message in case of failure. The state is implemented as a dependent record containing: (i) the number of steps allowed in recursion, (ii) a list of messages (used for debugging by the printing monad), and (iii) the memory.

The size of the memory has to be dynamic, but at the same time the memory has to be statically typed. Our solution is to parametrize the memory by a signature $\mathit{\Sigma}$, containing the exact list of types $T_1, T_2, \ldots, T_n$ that will be used. Then, the memory is a list of n regions of types $T_1, \ldots, T_n$ respectively. The content of a region is unbounded. For instance each region may contain a list of elements. A reference has type $\mathrm{Ref.t}\ \mathit{\Sigma}\ T_i$, and its implementation is simply the natural number i corresponding to the i-th type in the signature $\mathit{\Sigma}$. All in all, here are the effectful operations offered by the monad:

$$\begin{array}{ll}
\mathsf{ref} & : T_i \to \mathsf{M}\ \mathit{\Sigma}\ (\mathrm{Ref.t}\ \mathit{\Sigma}\ T_i) \\
\mathsf{read} & : \mathrm{Ref.t}\ \mathit{\Sigma}\ T \to \mathsf{M}\ \mathit{\Sigma}\ T \\
\mathsf{write} & : \mathrm{Ref.t}\ \mathit{\Sigma}\ T \to T \to \mathsf{M}\ \mathit{\Sigma}\ () \\
\mathsf{print} & : \alpha \to \mathsf{M}\ \mathit{\Sigma}\ ()
\end{array}
\qquad
\begin{array}{ll}
\mathsf{error} & : \mathsf{string} \to \mathsf{M}\ \mathit{\Sigma}\ \alpha \\
\mathsf{try\_with} & : (() \to \mathsf{M}\ \mathit{\Sigma}\ \alpha) \to \\
 & \quad (\mathsf{string} \to \mathsf{M}\ \mathit{\Sigma}\ \alpha) \to \mathsf{M}\ \mathit{\Sigma}\ \alpha \\
\mathsf{dependentfix} & : (\mathcal{F} \to \mathcal{F}) \to \mathcal{F} \\
 & \quad \text{with } \mathcal{F} = \forall (x : A). \mathsf{M}\ \mathit{\Sigma}\ (B\ x)
\end{array}$$

[7] In this work, unlike in Haskell, we are not interested in a fine grain control of effects so we provide only one monad with all the effectful operations we found useful.

Pre-computation The memory is partitioned into two parts: InputMem and TmpMem. TmpMem is initially empty and corresponds to the memory in the usual state monad. InputMem is initialized by the OCaml program and given as the initial (read-only) memory to Coq as a prophecy. Roughly speaking, the order on the prophecies is induced by the distance between the contents of this initial memory and the information needed to compute the same result in Coq as in OCaml *i.e.* the required number of fixpoint iterations and the values that were pre-computed in OCaml. Inside the implementation of the monad, this forces us to program differently for these two environments and, for this reason, we defined a low-level internal operator, select, of type $\forall\,\alpha, (() \to \alpha) \to (() \to \alpha) \to \alpha$ which is defined in Coq as $\mathsf{select}(f, g) = f()$ and compiled in OCaml as $\mathcal{C}(g())$. For more information about select, we defer the reader to Section 5.2. To fulfill the requirements to ensure that our monad is simulable, we make sure that the OCaml version of each operator only refines the contents of the InputMem during its effectful execution.

4.2 In OCaml

The compilation of a monadic term written in Coq to a program in OCaml is implemented by customizing the existing extraction mechanism of Coq [13], where the new monadic constructs are extracted as follows:

$$\begin{array}{rlrl}
\mathsf{M}\ \Sigma\ \alpha & \mapsto \alpha & \mathsf{error} & \mapsto \textbf{fun}\ x \to \mathrm{failwith}\, x \\
\mathsf{unit} & \mapsto \textbf{fun}\ x \to x & \mathsf{try\_with} & \mapsto \textbf{fun}\ f\, h \to \textbf{try}\ f\ () \\
\mathsf{bind} & \mapsto \textbf{fun}\ x\, f \to f\ x & & \quad\ \textbf{with}\ s \mapsto h\ s \\
\mathsf{print} & \mapsto \textbf{fun}\ x \to \mathrm{print\_endline}\, x & \mathsf{tmp\_ref} & \mapsto \textbf{fun}\ i\, v \to \mathrm{ref}\ v \\
\mathsf{dependentfix} & \mapsto \textbf{let rec}\ \mathsf{fix}\ f = \textbf{fun}\ x \to & \mathsf{input\_ref} & \mapsto \textbf{fun}\ i\, v \to \mathrm{register\_ref}\ i\ v \\
& \qquad \mathrm{incr\_nbsteps}(); f\ (\mathsf{fix}\ f)\ x & \mathsf{read} & \mapsto \textbf{fun}\ r \to !r \\
& \quad \textbf{in}\ \mathsf{fix}\ f\ x & \mathsf{write} & \mapsto \textbf{fun}\ r\, v \to r := v
\end{array}$$

Since we are using the built-in effectful mechanisms provided by OCaml, the monad is converted into the identity monad and, thus, the bind and unit combinators are defined accordingly.

The print and partiality monad are implemented with the standard print function and exceptions. The fixpoint operator adds instrumentation to count the number of iterations in a global variable. The memory operators are handled by OCaml's references. References are divided into tmp_ref and input_ref. The first ones are just normal OCaml's references, while the second ones are registered in an array, using the function register_ref. Thus, we can collect the values of all the input references at the end of the execution to pass them to Coq.

4.3 Communication from OCaml to Coq

Once the execution of the OCaml code is done, we generate the prophecy for Coq. It contains two parts: the number of steps and the memory in InputMem. The first part is easy to communicate back, as it is just a natural number. As for InputMem, it is more tricky since we need to reify OCaml data into Coq terms. Notice that this is not possible in general, for example for abstractions or for proof terms, since the extraction to OCaml erases too much information from the source term. Our solution is to provide an ad-hoc reification mechanism using binary trees: for every type T in the input memory signature, the user needs to provide a morphism between T and a binary tree.

5 Examples

We now show examples of Coq programs written using a simulable monad. The first example describes how to write effectful programs, while the second example illustrates how the performance of an algorithm is greatly improved by using compilation to OCaml and cross-stage memoization.

5.1 Congruence-Closure

The congruence-closure problem is about proving equality of two first-order terms, given a set of known equalities. It can be solved efficiently using the union-find algorithm [2]. In [7], a reflexive version of the algorithm is presented, which is purely functional and proven correct. A large part of the code is devoted to prove termination and implementing functional arrays. We wrote this algorithm in our system using the partiality monad to avoid proving termination. We focus on the `Find` function:

```
Program Definition Find hash u : M Σ {u': Index.t | u ≡ u'} :=
  dependentfix (λ i ⇒ {j: Index.t | i ≡ j}) (λ find i ⇒
    let! eq_proof := MHash.Read hash i in
    let (i', j, Hij) := eq_proof in
    if i ≡ i' then (* case i = i': should always be the case *)
      if i ≡ j then (* case i = j: we find it *) return (exist _ j Hij)
      else (* case i <> j: we have to continue from j *)
        let! r := find j in
        let (k, Hjk) := r in
        do! MHash.Write hash i (EqProof.Make (i := i) (j := k) _) in
        return (exist _ k _)
    else (* case i <> i': unexpected *) error "Find: i ≢ i'")
  u.
```

At high level this function retrieves the representative u' of the equivalent class of u, along with a proof of the equality among u and u'. It iterates over a hash-table `hash` from expressions (`Index.t` in the code) to expressions, crawling the hash table until an element points to itself. If that is the case, then we reach the representative. The hash table also contains the proof of equality, which is used transitively to compute the resulting equality proof.

Programming with effects in Coq. Proving termination of the algorithm is hard since it requires to maintain the invariant that the table is not cyclic. Luckily, we are exempt to do such proof, thanks to the `dependentfix` operator that allows for non-termination. The hash-table is a mutable structure with a `read` and a `write` operation. It is implemented as a mutable map from expressions to expressions, with an additional proof of equality.

Dependently-typed programming with partial functions. We keep the power of the Coq type system despite the fact that we are working in a monad. The `Find` function has a dependent type specifying that the result is the representative term u', equal to the input term u. The proof term is generated in the monad, so we can rely on run-time checks, which may fail, instead of proving invariants. For example the invariant $i = i'$ holds but does not have to be statically proven. Instead it is checked dynamically (comparison of i and i' on line 5). The result is used to coerce a proof of $i' = j$ to $i = j$ (done automatically by the `Program` command in our example). If the check fails, we raise an exception handled by the partiality monad. In this way we can partially specify our programs. Notice that we are not forced to use partial programs, we can also use pure Coq functions leading to stronger static guarantees. This flexibility is not available in mainstream functional languages like OCaml.

5.2 A tactic for Lattices

James and Hinze [12] present a reflection-based tactic to solve lattice (in)equalities based on the algorithm proposed by Whitman [17], which is known for being exponential in the worst case. In this work, the authors made the following remark:

> "Possible future work is to turn our current implementation [...] into one that uses dynamic programming to memoize the recursive calls. However, this is not a trivial task. Coq's programming language is purely functional [...], so any data-structure that we use for memoization must be purely functional and operations on that data-structure must all be proved terminating."

In this section we provide a tactic similar to James and Hinze's, but that uses memoization. In our case, unlike in the recommendation made in the quoted text, we use a form of cross-stage memoization to remember the successful path of execution made by the OCaml version and to transmit it to the Coq version. In this way, the exponential algorithm is executed only in OCaml, while Coq just recreates the successful path made by the OCaml version mimicking what a certificate checker would do. Unsurprisingly, the implementation presented here greatly outperforms the one presented by James and Hinze. For details, we refer to the original work cited above or to the accompanying code.

Whitman's algorithm. The algorithm, as written by James and Hinze with some simplifications and syntax sugaring follows.

```
Program Fixpoint leq (t u : Term) : {b : bool | b → t ⩽ u} :=
  match (t,u) with
  | (Var m, Var n) ⇒ m ≡ n
  | (Join t1 t2, u) ⇒ leq t1 u ∧ leq t2 u
  | (t, Meet u1 u2) ⇒ leq t u1 ∧ leq t u2
  | (Var m, Join u1 u2) ⇒ leq t u1 ∨ leq t u2
  | (Meet t1 t2, Var n) ⇒ leq t1 u ∨ leq t2 u
  | (Meet t1 t2, Join u1 u2) ⇒ leq t1 u ∨ leq t2 u ∨ leq t u1 ∨ leq t u2
  end.
```

What is important to notice is the ∨ branching in the last three cases. In particular, the last case requires the algorithm to branch four times! This is the culprit for the exponential time taken by the algorithm in some examples.

Remembering the past. To simulate only the interesting part of the proof search, the simulation must choose the right side of every disjunction. This optimization lies on the following function which advantageously replaces ∨:

```
Fixpoint tryBranches (ref: Ref.t Σ _) (n_branch: nat) (k : TermPairMap.key) B
  (branches: list (unit → M Σ B)) : M Σ B := select
  (* Coq *) (λ _ ⇒ let! map := !ref in
      let! n_branch := extract_some (TermPairMap.find k map) in
      let! branch := extract_some (nth_error branches n_branch) in branch tt)
  (* OCaml *) (λ _ ⇒ let! map := !ref in
      match TermPairMap.find k map with
      | Some n ⇒ let! branch := extract_some (nth_error branches n) in branch tt
      | None ⇒ match branches with
        | nil ⇒ error "No branch left to try"
        | branch :: branches' ⇒ try!
            let! r := branch tt in
            let! map := !ref in
            do! ref :=! TermPairMap.add k n_branch map in
            return r
          with _ ⇒ tryBranches ref (S n_branch) k branches'
        end
      end).
```

This function uses the select operator to behave differently in OCaml than in Coq. In OCaml, it tries to execute the code in all of the branches, and the returned value comes from the first branch succeeding in its execution. In addition, the position of the successful branch is added to the map referenced by ref. If no branch succeed, then it raises an error. Before exploring the branches, it first checks whether it is known which branch to take, and if this is the case, it executes the code from that branch only. In Coq, it first reads the position from the map, and executes only the code from the branch in this position. In both cases, the key k used to store the position in the map is given as a parameter. This key is instantiated with a pair containing the terms from both sides of the inequality under consideration. The Whitman's algorithm is changed to take advantage of the branching function just described.

There is a couple of important remarks that must be made about this optimization. First, the very powerful select operator can obviously break the theoretical requirements to achieve the *a posteriori* simulation. It is provided to allow the user to create primitive operators not present in the monad. Second, our implementation of the non determinism assumes that there is no side-effect in the failing branches that may affect the successful ones.

Performance. As expected, we get a great performance gain, shown in Figures 1 and 2. These plots show the time it takes Coq to typecheck the result, for two different classes of problems. The time to typecheck the result from the original purely functional algorithm is shown in rounded dots, while for the effectful code it is shown in squares. In Figure 1 we consider a problem with an increasing number of variables, where there is no repetition in the formula (therefore every combination should be taken into account). In Figure 2, we increment the number of times a certain pattern occurs in an inequality, showing how our method benefits from reusing previously computed paths. To sum things up, these plots clearly shows the benefit of using prophecies to help the typechecker save some computation.

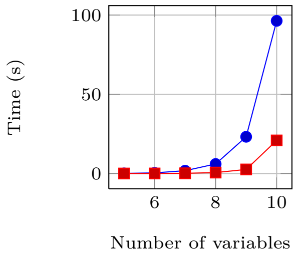

Fig. 1. Typechecking time for exponential proof terms.

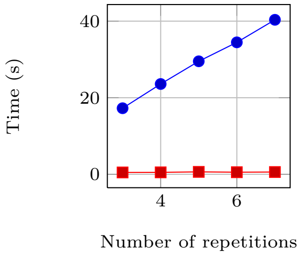

Fig. 2. Typechecking time for terms with a repetitive pattern.

6 Related work

Extending Coq with imperative features. Coq has been extended with imperative features [1]. The methodology behind this extension is to offer to Coq's user a functional interface to data structures that are efficiently compiled internally. This solution is transparent to the user: there is no need to write decision procedures in a monad to use imperative mechanisms. Yet, the trusted base, *i.e.* the kernel of Coq, had to be extended. Actually, the two systems can be used together: we could make use of the efficient data structures provided by this extension to define some of the effectful operators of our monad improving the performance of the *a posteriori* simulation done at Qed time.

Prophecies in Type Theory. Several works propose [5,14] methods to define and to reason on general recursive functions in Type Theory. Bove and Capretta [5] formally define a notion of prophecy, a coinductive predicate derived from a set of non-overlaping recursive equations characterizing the co-domain of the partial function defined by these equations. Our prophecies and Bove and Capretta's share the same role of prediction. However, our prophecies do not need to be coinductive because our monad uses them in direct style. Besides, our prophecies are computed outside Coq by an efficient computational model *i.e.* OCaml.

7 Conclusion

In this paper, we presented a novel technique to write decision procedures in Coq. We described its implementation as a plugin and we hope that it will simplify the development of proofs by reflection in the future.

Acknowledgments We would like to thank the anonymous reviewers for their thorough and helpful reviews.

References

1. M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with Imperative Features and its Application to SAT Verification. In *ITP*, 2010.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge Univ. Press, 1998.
3. J. O. Blech and B. Grégoire. Certifying compilers using higher-order theorem provers as certificate checkers. *FMSD*, 2011.
4. S. Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, 1997.
5. A. Bove and V. Capretta. Computation by prophecy. In *TLCA*, 2007.
6. G. Claret, L. González Huesca, Y. Régis-Gianas, and B. Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. Technical report, 2013. http://cybele.gforge.inria.fr/download/cybele_technical_report.pdf.
7. P. Corbineau. Autour de la clôture de congruence avec coq. Master's thesis, ENS, 2001. In French.
8. G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. ICFP, 2011.
9. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.
10. B. Grégoire, L. Pottier, and L. Théry. Proof certificates for algebra and their application to automatic geometry theorem proving. In *ADG*, 2008.
11. J. Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical report, SRI Cambridge, 1995.
12. D. W. H. James and R. Hinze. A Reflection-based Proof Tactic for Lattices in Coq. In *TFP*, 2009.
13. P. Letouzey. Coq Extraction, an Overview. In *LTA*, LNCS, 2008.
14. D. Pichardie and V. Rusu. Defining and Reasoning About General Recursive Functions in Type Theory: a Practical Method. Research report, IRISA, 2005.
15. M. Sozeau. Subset coercions in coq. In *TYPES*, 2006.
16. P. Wadler. Comprehending monads. *MSCS*, 1992.
17. P. Whitman. *Free Lattices.* Harvard University, 1941.