Lightweight proof by reflection using a posteriori simulation of effectful computation

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

lgonzale@pps.univ-paris-diderot.fr

[1]PPS, team πr^2 (University Paris Diderot, CNRS, and INRIA)
[2]Max Planck Institute for Software Systems (MPI-SWS)

TYPES
April 24, 2013

Lightweight proof by reflection
using a posteriori simulation of effectful computation

Lightweight proof by reflection

using a posteriori simulation of effectful computation

Running example

How should we prove the following equivalence?

$$\frac{\begin{array}{llcl} \mathcal{H}_1 : & A_{11} & \sim & A_{12} \\ \mathcal{H}_2 : & A_{21} & \sim & A_{22} \\ & & \vdots & \\ \mathcal{H}_n : & A_{n1} & \sim & A_{n2} \end{array}}{\begin{array}{ccc} A & \sim & A' \end{array}}$$

Running example

How should we prove the following equivalence?

Manually using a script?

$$\begin{array}{rccc} \mathcal{H}_1 : & A_{11} & \sim & A_{12} \\ \mathcal{H}_2 : & A_{21} & \sim & A_{22} \\ & & \vdots & \\ \mathcal{H}_n : & A_{n1} & \sim & A_{n2} \\ \hline & A & \sim & A' \end{array}$$

Running example

How should we prove the following equivalence?

Using a **decision procedure**?

$$\frac{\begin{array}{rccc} \mathcal{H}_1 : & A_{11} & \sim & A_{12} \\ \mathcal{H}_2 : & A_{21} & \sim & A_{22} \\ & & \vdots & \\ \mathcal{H}_n : & A_{n1} & \sim & A_{n2} \end{array}}{A \sim A'}$$

Running example : A decision procedure in pseudo-code

```
let is_equivalent (hs, (i, j)) : bool =
  iter (fun (i,j) -> union i j) hs;
  return (find i == find j)
```

where

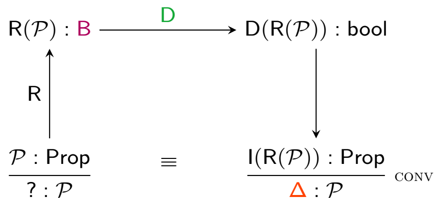
`union i j`	merges the equivalence classes of `i` and `j`
`find i`	returns the unique representative of `i`

How can we put this decision procedure in work?

Recipe for a proof by reflection

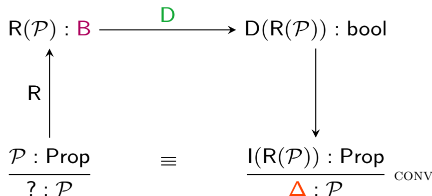
$$
\begin{array}{ccc}
\mathsf{R}(\mathcal{P}) : \mathsf{B} & \xrightarrow{\mathsf{D}} & \mathsf{D}(\mathsf{R}(\mathcal{P})) : \mathsf{bool} \\
\uparrow \mathsf{R} & & \downarrow \\
\dfrac{\mathcal{P} : \mathsf{Prop}}{? : \mathcal{P}} & \equiv & \dfrac{\mathsf{I}(\mathsf{R}(\mathcal{P})) : \mathsf{Prop}}{\Delta : \mathcal{P}}\ \text{CONV}
\end{array}
$$

Recipe for a proof by reflection



1. Define a type **B** for the targeted class of problems in Coq.

Recipe for a proof by reflection

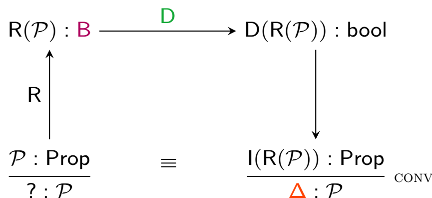


1. Define a type **B** for the targeted class of problems in Coq.

$(\mathsf{hs}, (\mathsf{i}, \mathsf{j})) : \mathsf{list}(\mathsf{atom} \times \mathsf{atom}) \times (\mathsf{atom} \times \mathsf{atom})$

where atom is a type with decidable equality

Recipe for a proof by reflection



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \rightarrow \mathsf{Prop}$.

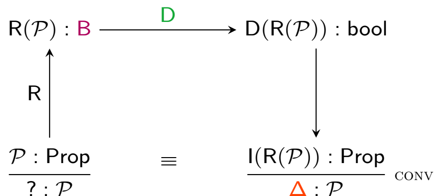
Recipe for a proof by reflection

$$\begin{array}{ccc} \mathsf{R}(\mathcal{P}) : \mathsf{B} & \xrightarrow{\mathsf{D}} & \mathsf{D}(\mathsf{R}(\mathcal{P})) : \mathsf{bool} \\ \uparrow \mathsf{R} & & \downarrow \\ \dfrac{\mathcal{P} : \mathsf{Prop}}{? : \mathcal{P}} & \equiv & \dfrac{\mathsf{I}(\mathsf{R}(\mathcal{P})) : \mathsf{Prop}}{\Delta : \mathcal{P}}\ \textsc{conv} \end{array}$$

1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \rightarrow \mathsf{Prop}$.

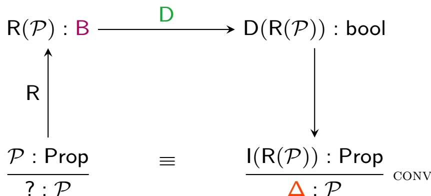
I (hs, (i, j)) : Prop.

Recipe for a proof by reflection



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function I : B → Prop.
3. Write the decision procedure **D : B → bool** in Coq.

Recipe for a proof by reflection

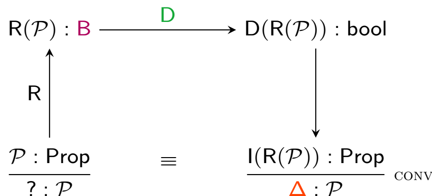


1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \to \mathsf{Prop}$.
3. Write the decision procedure **D** : **B** → **bool** in Coq.

$\mathsf{D}\ (\mathsf{hs}, (\mathsf{i}, \mathsf{j})) : \mathsf{bool}.$

Critical: *How should we implement* `union` *and* `find`?

Recipe for a proof by reflection



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function I : B → Prop.
3. Write the decision procedure **D : B → bool** in Coq.
4. Prove the **soundness** of D

Recipe for a proof by reflection

$$\begin{array}{ccc} \mathsf{R}(\mathcal{P}) : \mathsf{B} & \xrightarrow{\mathsf{D}} & \mathsf{D}(\mathsf{R}(\mathcal{P})) : \mathsf{bool} \\ \uparrow \mathsf{R} & & \downarrow \\ \dfrac{\mathcal{P} : \mathsf{Prop}}{? : \mathcal{P}} & \equiv & \dfrac{\mathsf{I}(\mathsf{R}(\mathcal{P})) : \mathsf{Prop}}{\Delta : \mathcal{P}}\ \textsc{conv} \end{array}$$

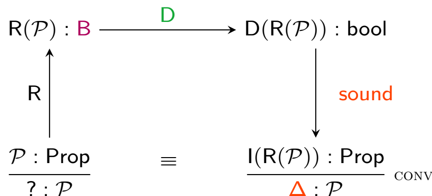
1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \to \mathsf{Prop}$.
3. Write the decision procedure $\mathbf{D} : \mathbf{B} \to \mathbf{bool}$ in Coq.
4. Prove the **soundness** of D

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

if $\mathsf{D}(\mathsf{hs}, (\mathsf{i}, \mathsf{j})) = \mathsf{true}$ then $a \sim a'$

Critical: *The development cost of this proof depends on the implementation choice for* `union` *and* `find`.

Recipe for a proof by reflection

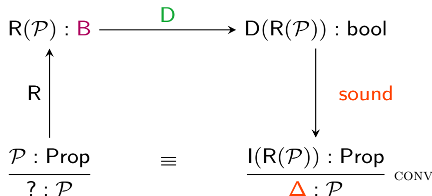


1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \to \mathsf{Prop}$.
3. Write the decision procedure $\mathbf{D} : \mathbf{B} \to \mathbf{bool}$ in Coq.
4. Prove the **soundness** of D

For a specific instance $\mathbf{b} = \mathbf{R}(\mathcal{P})$, the ***proof-term*** for I b has the form:

$$\mathsf{sound}\ \ \mathsf{b}\ \left(\mathsf{refl_eq}\left(\mathsf{D}(\mathsf{b})\right)\right)$$

Recipe for a proof by reflection



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \rightarrow \mathsf{Prop}$.
3. Write the decision procedure $\mathbf{D} : \mathbf{B} \rightarrow \mathbf{bool}$ in Coq.
4. Prove the **soundness** of D

For a specific instance $\mathbf{b} = \mathbf{R}(\mathcal{P})$, the ***proof-term*** for $\mathsf{I}\,\mathsf{b}$ has the form:

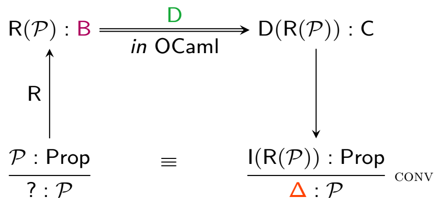
$$\mathsf{sound}\ \ \mathsf{b}\ \left(\mathsf{refl_eq}\left(\mathsf{D}(\mathsf{b})\right)\right)$$

This term has type $\mathbf{I}\,\mathbf{b}$ only if $\mathsf{D}\,\mathsf{b}$ is ***convertible*** to true.

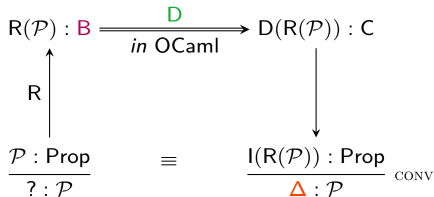
Original proof by reflection

A certified decision procedure
written in a **total** language
is a **robust** tool for the proof developer.
But,
it has a **high cost of development**,
often leading to **simplifications**
destroying efficiency.

Recipe for a proof by reflection with an untrusted oracle



Recipe for a proof by reflection with an untrusted oracle



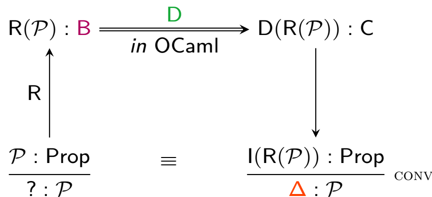
1. Define a type **B** for the targeted class of problems in Coq.

Recipe for a proof by reflection with an untrusted oracle

$$\begin{array}{ccc} R(\mathcal{P}) : B & \xrightarrow[\textit{in OCaml}]{D} & D(R(\mathcal{P})) : C \\ \uparrow R & & \downarrow \\ \dfrac{\mathcal{P} : \mathrm{Prop}}{? : \mathcal{P}} & \equiv & \dfrac{I(R(\mathcal{P})) : \mathrm{Prop}}{\Delta : \mathcal{P}}\ \textsc{conv} \end{array}$$

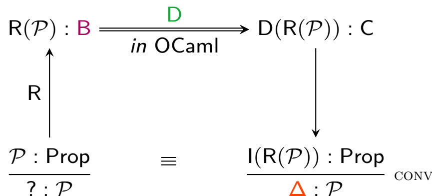
1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $I : B \to \mathrm{Prop}$.

Recipe for a proof by reflection with an untrusted oracle



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $I : B \to \text{Prop}$.
3. Write an (untrusted) oracle **D** in ML.

Recipe for a proof by reflection with an untrusted oracle

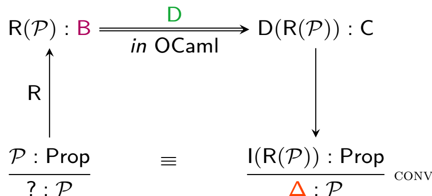


1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function I : B → Prop.
3. Write an (untrusted) oracle **D** in ML.

D : B → C

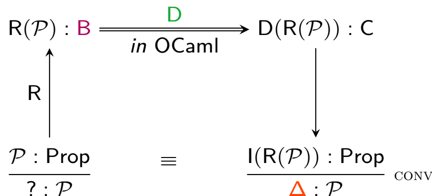
Critical: D *gives a certificate that must be checked.*

Recipe for a proof by reflection with an untrusted oracle



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $I : B \rightarrow \mathrm{Prop}$.
3. Write an (untrusted) oracle **D** in ML.
4. Write a simple **certificate checker** in Coq.

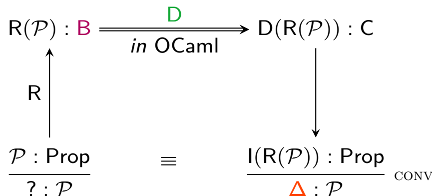
Recipe for a proof by reflection with an untrusted oracle



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \rightarrow \mathsf{Prop}$.
3. Write an (untrusted) oracle **D** in ML.
4. Write a simple **certificate checker** in Coq.

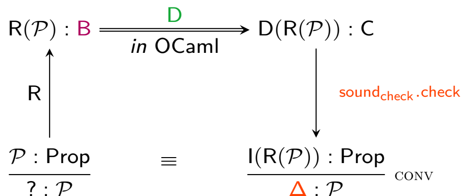
$$\text{check} : \forall x : \mathsf{B},\ \mathsf{C} \rightarrow \text{bool}$$

Recipe for a proof by reflection with an untrusted oracle



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $I : B \rightarrow \text{Prop}$.
3. Write an (untrusted) oracle **D** in ML.
4. Write a simple **certificate checker** in Coq.
5. Prove the soundness of the checker

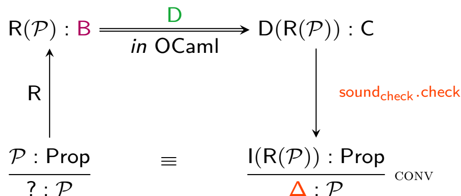
Recipe for a proof by reflection with an untrusted oracle



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \to \mathsf{Prop}$.
3. Write an (untrusted) oracle **D** in ML.
4. Write a simple **certificate checker** in Coq.
5. Prove the soundness of the checker

$$\mathsf{sound}_{\mathsf{check}} : \forall \mathsf{x} : \mathsf{B},\ \mathsf{y} : \mathsf{C},\ \mathsf{check}\,\mathsf{x}\,\mathsf{y} = \mathsf{true} \to \mathsf{I}\,\mathsf{x}$$

Recipe for a proof by reflection with an untrusted oracle



1. Define a type **B** for the targeted class of problems in Coq.
2. Write an interpretation function $\mathsf{I} : \mathsf{B} \rightarrow \mathsf{Prop}$.
3. Write an (untrusted) oracle **D** in ML.
4. Write a simple **certificate checker** in Coq.
5. Prove the soundness of the checker

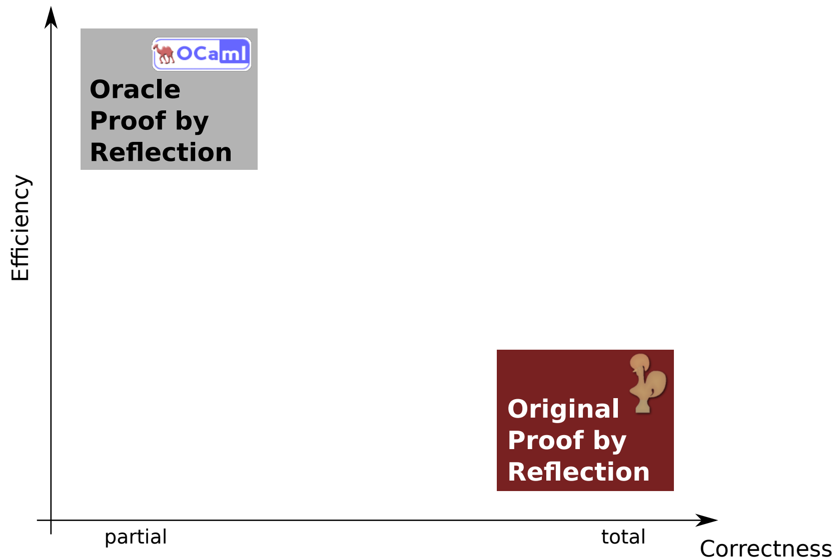
For a specific instance $\mathbf{b} = \mathbf{R}(\mathcal{P})$, a proof-term for **I b** has the form:

$$\mathsf{sound}_{\mathsf{check}}\ \ \mathsf{b}\ \ \mathsf{D}(\mathsf{b})\ \ \big(\mathsf{refl_eq}\big(\mathsf{check}\,\mathsf{D}(\mathsf{b})\big)\big)$$

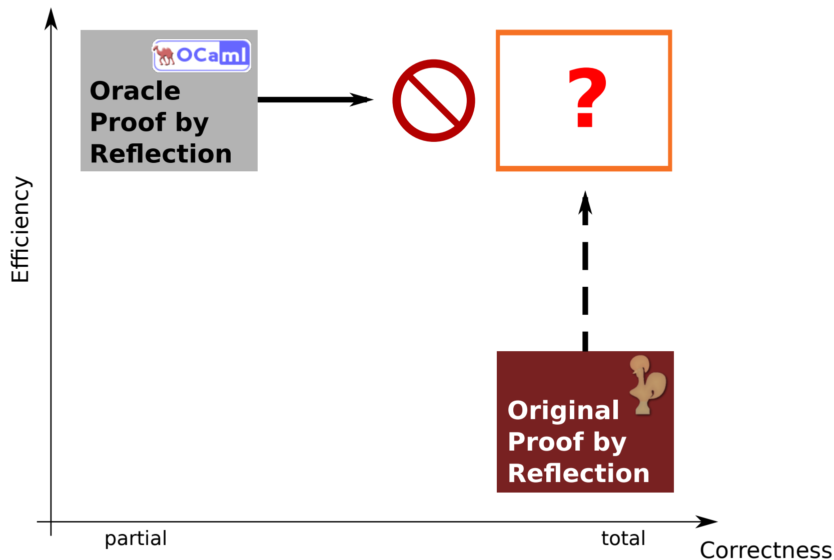
Proof by reflection using an untrusted oracle

The oracle implementation is very **efficient**.
The checker is usually **simple to prove**.
But,
we get only **weak guarantees**
about the oracle implementation.

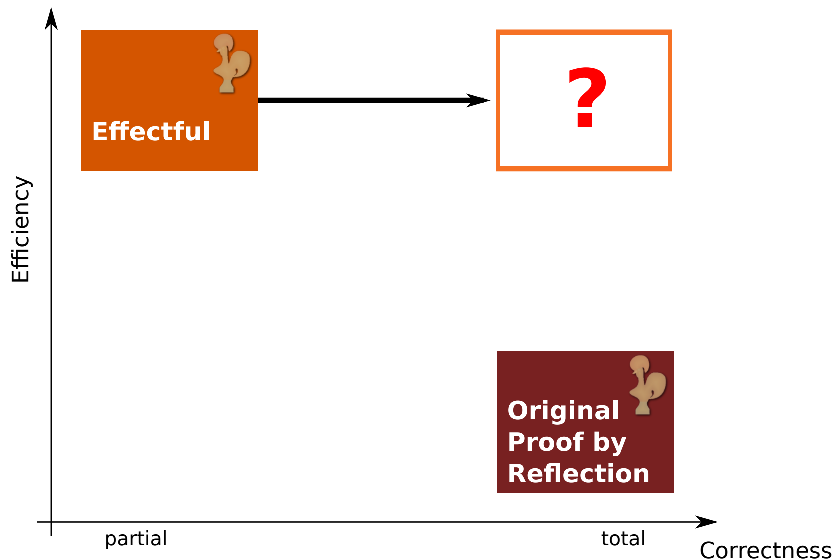
How to implement total **and** efficient decision procedures?



How to implement total **and** efficient decision procedures?



How to implement total **and** efficient decision procedures?



Listen to the old wisdom of Haskell's programmers

Listen to the old wisdom of Haskell's programmers

We can **represent** inside Coq

effectful and partial computations

Listen to the old wisdom of Haskell's programmers

We can **represent** inside Coq

effectful and partial computations

thanks to **monads**!

Running example: using a monad *à la* Haskell

```
Definition is_equivalent (hs,(i, j)) : M bool:=
  iterM (fun (i, j) -> union i j) hs >>
  let! x = find i in
  let! y = find j in
  return (x == y)
```

Running example: using a monad *à la* Coq

```
Definition is_equivalent (hs,(i, j)) : M (I (hs,(i,j))):=
  iterM (fun (i, j) -> union i j) hs >>
  let! x = find i in
  let! y = find j in
  return (x == y)
```

Drama

Drama

There is **no** **total function** of type

$$\mathrm{M\,T} \rightarrow \mathrm{T}$$

Lightweight proof by reflection

using a posteriori simulation of effectful computation

A posteriori simulation of effectful computation

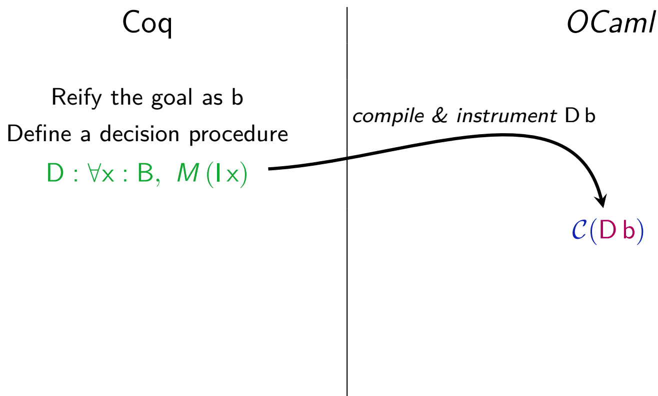
Coq

OCaml

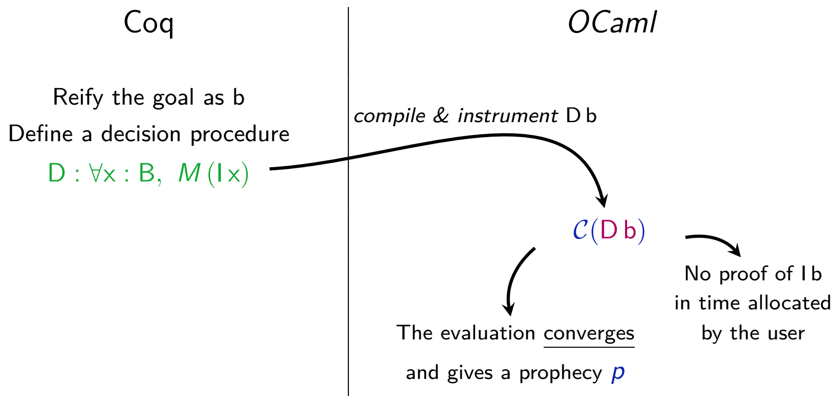
A posteriori simulation of effectful computation

Coq	*OCaml*
Reify the goal as b Define a decision procedure $D : \forall x : B,\ M\,(I\,x)$	

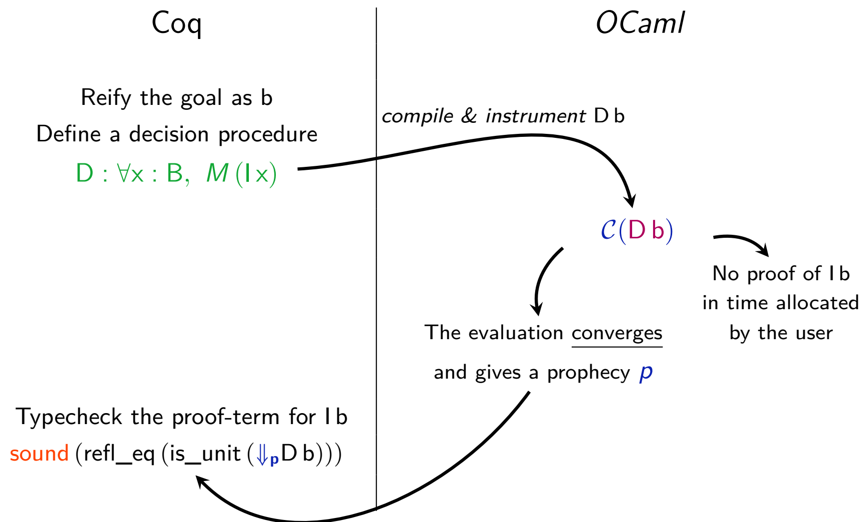
A posteriori simulation of effectful computation



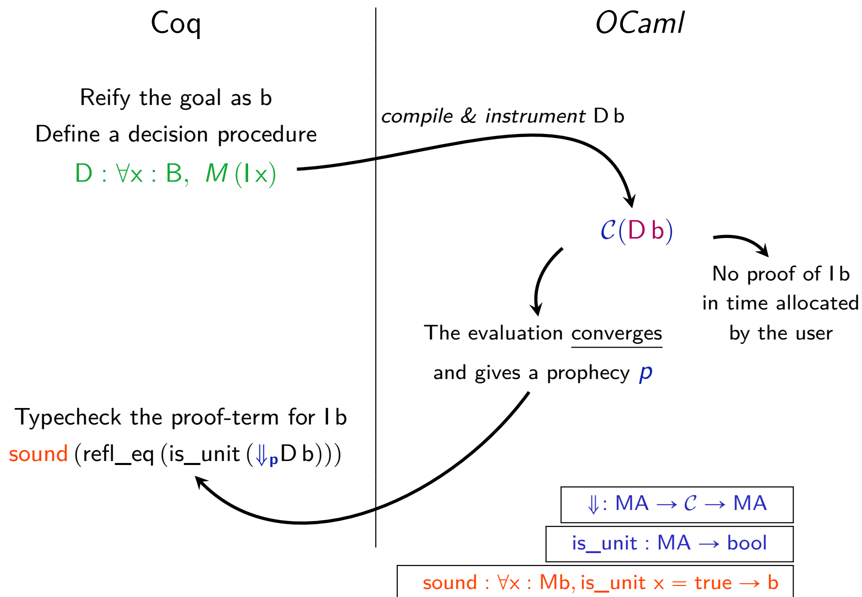
A posteriori simulation of effectful computation



A posteriori simulation of effectful computation



A posteriori simulation of effectful computation



What is a simulable monad?

What is a simulable monad?

It is a monad with

What is a simulable monad?

It is a monad with

$p : \mathcal{C}$, a type for prophecies equipped with a order $\prec$

What is a simulable monad?

It is a monad with

$p : \mathcal{C}$, a type for prophecies equipped with a order $\prec$

$\Downarrow : \mathsf{MA} \to \mathcal{C} \to \mathsf{MA}$, an operator to simulate an effectful computation using a prophecy

What is a simulable monad?

It is a monad with

$p : \mathcal{C}$, a type for prophecies equipped with a order $\prec$

$\Downarrow : \mathsf{MA} \rightarrow \mathcal{C} \rightarrow \mathsf{MA}$, an operator to simulate an effectful computation using a prophecy

such that:

What is a simulable monad?

It is a monad with

$p : \mathcal{C}$, a type for prophecies equipped with a order $\prec$

$\Downarrow : \text{MA} \rightarrow \mathcal{C} \rightarrow \text{MA}$, an operator to simulate an effectful computation using a prophecy

such that:

If a computation converges in the effectful computational model then there exists a prophecy to simulate this computation in the monad.

What is a simulable monad?

It is a monad with

$p : \mathcal{C}$, a type for prophecies equipped with a order $\prec$

$\Downarrow : \mathrm{MA} \rightarrow \mathcal{C} \rightarrow \mathrm{MA}$, an operator to simulate an effectful computation using a prophecy

such that:

If a computation converges in the effectful computational model then there exists a prophecy to simulate this computation in the monad.

(The instrumentation computes an over approximation of this prophecy.)

Simulable monads: Examples

Effect	$M\ \boldsymbol{T}$	***Prophecy***
non-termination	nat $\rightarrow$ option T	the maximal recursion depth
non-determinism	list T	a list of success choices
state	$S \rightarrow S \times$ T	an initial state

Simulable monads: Examples

Effect	$M\ \boldsymbol{T}$	***Prophecy***
non-termination	nat $\rightarrow$ option T	the maximal recursion depth
non-determinism	list T	a list of success choices
state	$S \rightarrow S \times$ T	an initial state

Simulating does not necessarily mean recomputing!

Programming with dependent types and partial functions

```
Program Definition Find hash u : { M Σ u': Index.t | u == u' } :=
  dependentfix (fun i => { j: Index.t | i == j }) (fun 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.
```

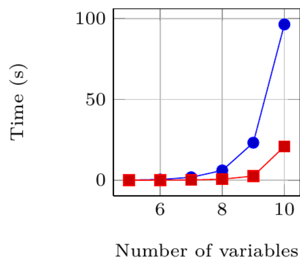
Experiments

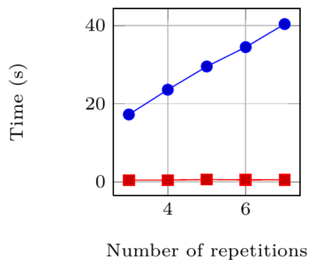
*"Possible future work is to turn our current implementation [...] into one that uses **dynamic programming to memoize** the recursive calls. However, this is not a trivial task. **Coq's programming language** is purely functional [...], so any data-structure that we use for memoization must be **purely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

A Reflection-based Proof Tactic for Lattices in COQ

[James & Hinze 2009]

Benefits in terms of efficiency





Final remarks

A plugin for Coq is available
and provides a simulable monad that includes
partiality, non-termination, state, and non-determinism.

Final remarks

A plugin for Coq is available
and provides a simulable monad that includes
partiality, non-termination, state, and non-determinism.

We invite the audience to try it and give us feedback!

http://cybele.gforge.inria.fr/