Lightweight proof by reflection using a posteriori simulation of effectful computation

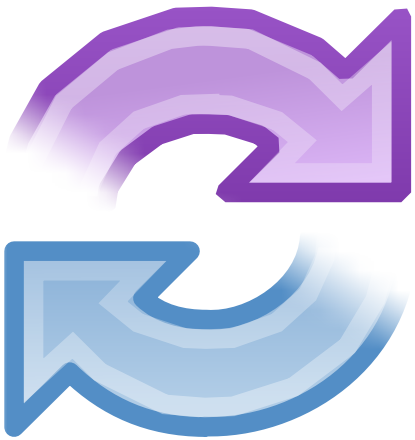
Guillaume Claret [1]
Lourdes del Carmen González Huesca [1]
Yann Régis-Gianas [1]
{guillaume.claret,lgonzale,yann.regis-gianas}@pps.univ-paris-diderot.fr
Beta Ziliani [2]
beta@mpi-sws.org

[1]PPS, team πr^2 (University Paris Diderot, CNRS, and INRIA)

[2]Max Planck Institute for Software Systems (MPI-SWS)

Conference/Workshop name
March 21, 2013

Proof by reflection



Let us take an example

- Problem: to decide if two elements are equivalent according to some known equivalence properties
- T is a type with $\sim$ an equivalence relation
- the hypothesis are:

$$\begin{array}{rrcl} H_1: & e_{i_1} & \sim & e_{j_1} \\ & & \vdots & \\ H_n: & e_{i_n} & \sim & e_{j_n} \end{array}$$

- the goal is to prove $e_i \sim e_j$

Decide equivalence

Proof by hands, small instance

- hypothesis:

$$\begin{array}{rrcl} H_{ab}: & a & \sim & b \\ H_{bc}: & b & \sim & c \\ H_{cd}: & c & \sim & d \\ H_{de}: & d & \sim & e \end{array}$$

- goal: $a \sim e$
- solution: apply 3 times the transitivity

Decide equivalence

Proof by hands, big instance

- hypothesis:

$$\begin{array}{rrcl} H_{1,2}: & e_1 & \sim & e_2 \\ H_{2,3}: & e_2 & \sim & e_3 \\ & & \vdots & \\ H_{999,1000}: & e_{999} & \sim & e_{1000} \end{array}$$

- goal: $e_1 \sim e_{1000}$
- solution: apply 998 times the transitivity

Automatized solutions

Coq

- using tactics: a proof style where the goal is transformed to a proof term which can be:
  - ↳ huge
  - ↳ slow to type-check
  - ↳ unsafe using $\mathcal{L}_{tac}$ (untyped language)
- proof by reflection

Proof by reflection: reification

Step 1

- From the *Coq* terms:

$$\begin{array}{rrcl} H_{1,2} : & e_1 & \sim & e_2 \\ & & \vdots & \\ H_{999,1000} : & e_{999} & \sim & e_{1000} \end{array}$$

- transform into reified terms:

$$\underbrace{[(1,2),\ldots,(n-1,n)],(1,1000)}_{x} : \underbrace{\mathsf{list}\ (\mathsf{nat} \times \mathsf{nat}) \times (\mathsf{nat} \times \mathsf{nat})}_{A}$$

- and define an interpretation function: $I : A \to \mathsf{Prop}$

Proof by reflection: decision procedure

Step 2

Two things:

- a decision procedure for A: $\mathsf{D} : A \to \mathsf{bool}$
- a proof of the soundness of D

Pseudo code

$$
\begin{array}{l}
D\ (hs, (i, j)) : \mathsf{bool}\ = \\
\quad \mathsf{let}\ a\ =\ \mathsf{ref}\ [\ ]\ \mathsf{in} \\
\quad \mathsf{map}\ ((i, j) \mapsto \mathsf{union}\ a\ i\ j)\ hs; \\
\quad \mathsf{let}\ i'\ =\ \mathsf{find}\ a\ i\ \mathsf{in} \\
\quad \mathsf{let}\ j'\ =\ \mathsf{find}\ a\ j\ \mathsf{in} \\
\quad i' = j'
\end{array}
$$

Problems

- no reference because *Coq* is purely functional
- how can we prove this algorithm is sound?

Solution: proof with invariants

$$
\begin{array}{l}
D\ (hs, (i, j)) : \text{option}\ e_i \sim e_j\ = \\
\quad \text{let}\ a\ =\ \text{ref}\ [\ ]\ \text{in} \\
\quad\quad \text{"where if}\ (i, j) \in a\ \text{then}\ e_i \sim e_j\text{"} \\
\quad \text{map}\ ((i, j) \mapsto \text{union}\ a\ i\ j)\ hs; \\
\quad \text{let}\ i'\ =\ \text{find}\ a\ i\ \text{in} \\
\quad \text{let}\ j'\ =\ \text{find}\ a\ j\ \text{in} \\
\quad \text{if}\ i' = j'\ \text{then} \\
\quad\quad \text{Some "the proof that}\ e_i \sim e_j\text{"} \\
\quad \text{else} \\
\quad\quad \text{None}
\end{array}
$$

- the proof is made along the computation
- the proof can be complex

Solution: an *OCaml* oracle

Method

1. an *OCaml* program generates a list of steps $[i_1, \ldots, i_n]$
2. a *Coq* program checks that $e_i \sim e_{i_1}, e_{i_1} \sim e_{i_2}, \ldots, e_{i_n} \sim e_j$

- complex computations made in an fast and effectful programming language
- *Coq* only used as a checker
- the oracle is unsafe
- the certificate can be big

Pure *Coq* style vs Oracle style

	Pure *Coq*	Oracle
language for writing "D"	**Type theory** no side-effects total functions	**general purpose PL** untrusted oracle
proving soundness	in Type theory	in a certified checker
size of the final proof-term	**small** is a proof of equality	**medium** includes a certificate $\mathcal{C}$
typechecking	**convertibility check**	conv. check + **check the certificate**
limitations	too strict	a checker for certificates

Why to choose?

We cannot have both but we can have a mix.

The *Coqbottom* project

http://coqbottom.gforge.inria.fr/

Compute with effects in *Coq*

Coqbottom is a *Coq* plugin providing:

- a monad $M\ T$
- new operators
- a tactic `coqbottom` to run the monad

Operators

References

- ref : $\forall i, T_i \to \mathsf{M}\ (\mathrm{Ref.t}\ T_i)$
- read : $\mathrm{Ref.t}\ T \to \mathsf{M}\ T$
- write : $\mathrm{Ref.t}\ T \to T \to \mathsf{M}\ ()$

Exceptions

- error : string $\to \mathsf{M}\ T$
- try_with : $(() \to \mathsf{M}\ T) \to (\mathsf{string} \to \mathsf{M}\ T) \to \mathsf{M}\ T$

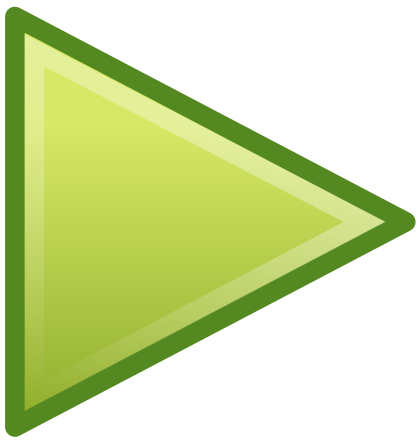
Non-termination

- dependent_fix : $(\mathcal{F}\ A\ B \to \mathcal{F}\ A\ B) \to \mathcal{F}\ A\ B$

Printing

- print : $T \to \mathsf{M}\ ()$

Demo



How to run the monad?

- has to be efficient
-  we cannot run the monad in pure *Coq*

There is no function of type:

$$M\ A \to A$$

- partiallity: we could do a:

$$\text{run} : M\ A \to \text{option}\ A$$

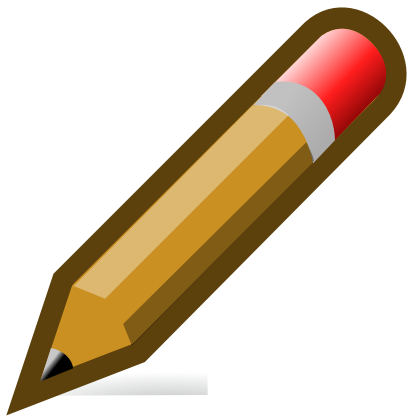
- non-termination: we need a prophecy p:

$$\text{run} : \text{prophecy} \to M\ A \to \text{option}\ A$$

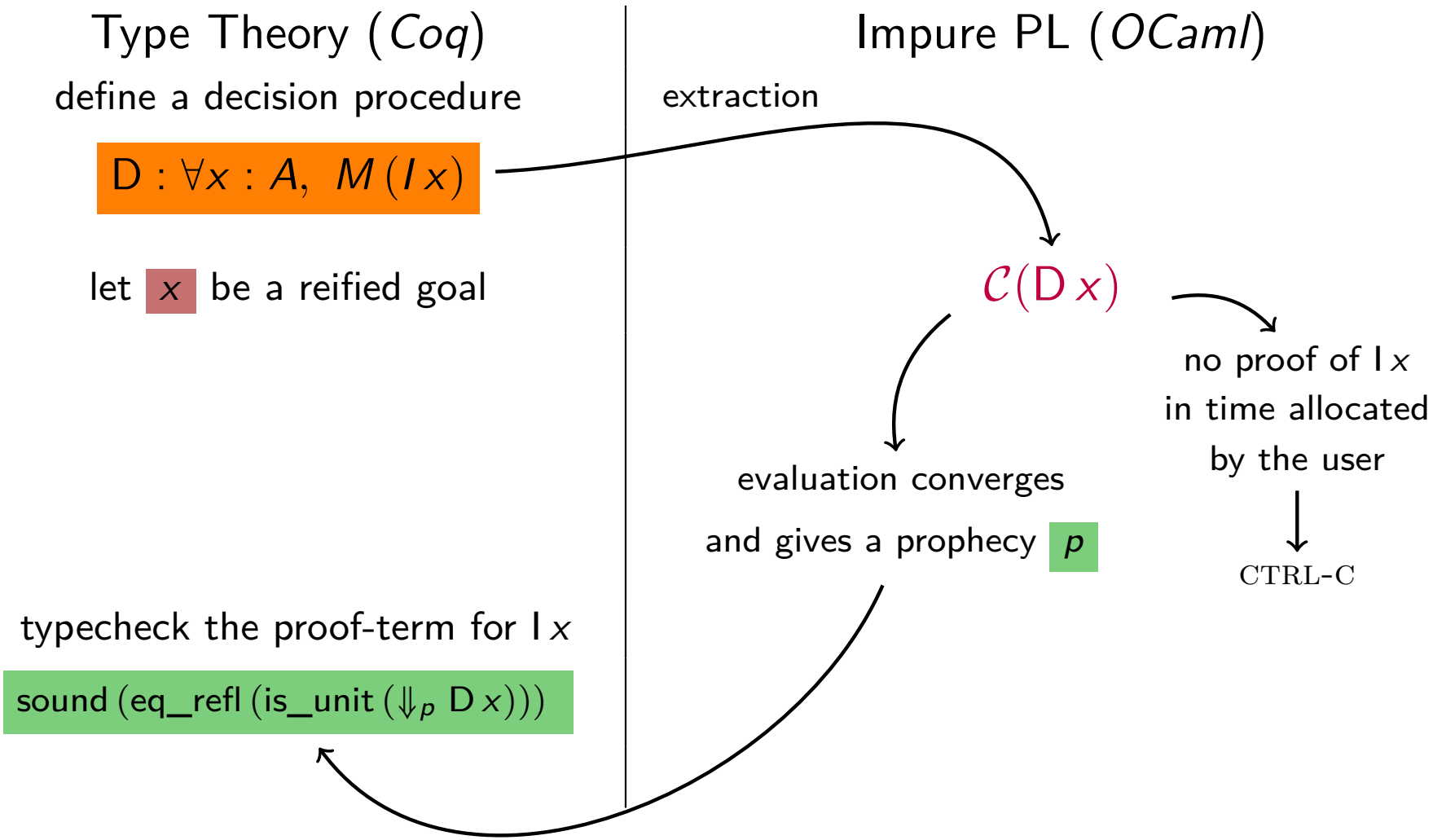
Prophecy

- the number of steps necessary to evaluate the fixpoints
- ensure all *Coq* terms are terminating
- cannot be computed inside *Coq*

Theory of simulable monads



Simulation-based proof by reflection



A posteriori simulation of effects

Functional language

We equip a purely functional language with what we call *simulable monad*.

↳ the set of constants **c** include
- usual monadic combinators: *unit* and *bind*
- effectful primitives: $\Downarrow$ ∇

↳ the role of $\Downarrow_p\ t$ is to perform *a posteriori* simulation, it is read as:

"the reduced computation of t using the prophecy p"

$$\frac{\Gamma \vdash t : \mathrm{MT} \qquad \Gamma \vdash p : \mathrm{P}}{\Gamma \vdash \Downarrow_p\ t : \mathrm{MT}}$$

↳ the set ∇ includes the effectful operators

A posteriori simulation of effects

Effectful language

We give an instrumented semantics to a general purpose language.

↳ a big-step operational semantics for a weak call-by-value reduction strategy:

"the execution of term u under the environment η converges to a value v and computes a prophecy p' from an initial prophecy p"

$$\eta \vdash u \Downarrow_{p \to p'} v$$

Simulation

How we can ensure an effective simulation?

- We use monads to simulate effects in the purely language.
- The prophecy helps to simulate effects
- A successful simulation is done when a computation t converges.

t has converged if there exist p and t' such that $\Downarrow_p t = \star t'$

where $\star t =_{def} \Downarrow_\perp \text{unit } t$

$\Rightarrow$ What are the requirements for a monad to be simulable?

$\Rightarrow$ What function respects effects and binds the pure language with the effectful language?

Requirements I

- Monad laws
- The simulation must respect the operators unit and bind
  - $\Downarrow_{p_1} \text{unit } t = \Downarrow_{p_2} \text{unit } t$
  - $\Downarrow_p \text{bind } t_1\ t_2 = \Downarrow_p \text{bind } (\Downarrow_p\ t_1)\ t_2$
- The prophecies have an order which is also respected by the simulation:
  - if $p_1 \leqslant p_2$ and $\Downarrow_{p_1}\ t = \star u$ then $\Downarrow_{p_2}\ t = \star u$

Compilation

The link between the languages is achieved
by means of a compilation.

- instantiate the programs with the identity monad:

$$\begin{array}{rcl} \mathcal{C}(x) & = & x \\ \mathcal{C}(\lambda x.t) & = & \lambda x.\mathcal{C}(t) \\ \mathcal{C}(t_1\ t_2) & = & \mathcal{C}(t_1)\,\mathcal{C}(t_2) \end{array} \qquad \begin{array}{rcl} \mathcal{C}(\mathsf{unit}) & = & \lambda x.x \\ \mathcal{C}(\mathsf{bind}) & = & \lambda x, y.y\,x \\ \mathcal{C}(\Downarrow_p) & = & \text{undefined} \end{array}$$

$$\mathcal{C}(\mathsf{M}\,\mathsf{T}) \quad = \quad \mathcal{C}(\mathsf{T})$$

Requirements II

- The instrumented semantics ensures a prophecy growth:

$$\eta \vdash t \Downarrow_{p \to p'} v \text{ implies } p \leqslant p'.$$

- There is an adequate instrumented compilation for compiled terms

$$\text{if } \begin{cases} \forall i, & \eta \vdash \mathcal{C}(t_i) \Downarrow_{p_i \to p_{i+1}} v_i \\ & \eta \vdash \mathcal{C}(\mathbf{c}\,(t_0, \ldots, t_n)) \Downarrow_{p_0 \to p} v \end{cases}$$

$$\text{then } \exists u, \Downarrow_p \mathbf{c}\,(t_0, \ldots, t_n) = \star u$$

Simulation

Theorem

Finally, we relate semantics of both languages:

Theorem (A posteriori simulation)

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

Closer look at the implementation

The monad

State, exceptions, non-termination and printing:

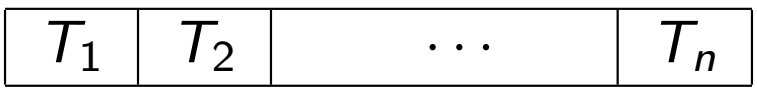
$$M\ \Sigma\ T = \text{State.t}\ \Sigma \to (T + \text{string}) \times \text{State.t}\ \Sigma$$

$$\begin{aligned}
&\text{State.t}\ \Sigma = \{ \\
&\quad \text{mem} : \text{Mem.t}\ \Sigma; \\
&\quad \text{nb_steps} : \text{nat}; \\
&\quad \text{output} : \text{list}\ \{T : \text{Type}\ \&\ T\}; \\
&\}
\end{aligned}$$

Something new: the signature Σ.

The signature Σ

To type the memory operators we need to statically type the memory.
A memory with $\Sigma = [T_1, T_2, \ldots, T_n]$:



- ref : $\forall\, \Sigma, i, T_i \to \mathsf{M}\ \Sigma\ (\mathrm{Ref.t}\ \Sigma\ T_i)$
- read : $\forall\, \Sigma, \mathrm{Ref.t}\ \Sigma\ T \to \mathsf{M}\ \Sigma\ T$
- write : $\forall\, \Sigma, \mathrm{Ref.t}\ \Sigma\ T \to T \to \mathsf{M}\ \Sigma\ ()$

Memory still extensible if $T_i = \mathsf{list}\ T_i'$.

Two memories

- tmp_mem: as usual
- input_mem: written by *OCaml*, read by *Coq*
  - some pre-computations can be done in *OCaml*
  - backtracking can be solved in *OCaml*
  - has to be reifiable to be transmitted in the prophecy:
    - no abstraction
    - no proposition

The prophecy

Non-termination and input memory:

$$
\begin{aligned}
&\text{Prophecy.t } \Sigma = \{ \\
&\quad \text{nb\_steps} : \text{nat}; \\
&\quad \text{input\_mem} : \text{Mem.t } \Sigma_{\text{input}}; \\
&\}
\end{aligned}
$$

Extraction to *OCaml*

- extraction to the identity monad:

$$\begin{aligned} \text{unit } x &\mapsto x \\ \text{bind } f\ x &\mapsto f\ x \end{aligned}$$

- effectful operators extracted to their direct *OCaml* equivalent:

$$\text{read } r \mapsto !\, r$$

- prophecy filled by side-effects:

```
let rec fix f x =
  CoqbottomState.observe_recursive_call ();
  f (fix f) x
```

Operator *select*

Allows the user to control the extraction process:

$$\begin{array}{lccccc} \text{select}: & ((\,) \to T) & \to & ((\,) \to T) & \to & T \\ & f & \mapsto & g & \mapsto & \begin{cases} f() & \text{in coq} \\ g() & \text{in ocaml} \end{cases} \end{array}$$

- used to hack with the input memory
- out of the simulation theory

Conclusions and future work

We presented a novel technique of proof by reflection performing *a posteriori* simulations of effectful computations in Type Theory.
Long-term goals:

- optimize the back-end
  - ↳ proven extraction so the computations are made once?
- use *Coq* as a typed tactic language for *Coq*
  - ↳ extend the monad to allow parallel computations
  - ↳ deeper integration into the proof engine
  - ↳ reflect *Coq*'s meta-theory in the monad

We invite the audience to visit:

`http://coqbottom.gforge.inria.fr/`

Questions?

Separate the proof from the algorithm

$$
\begin{array}{l}
D\ (hs, (i, j)) : \mathsf{bool}\ = \\
\quad \mathsf{let}\ a\ =\ \mathsf{ref}\ [\ ]\ \mathsf{in} \\
\quad \mathsf{map}\ ((i, j) \mapsto \mathsf{union}\ a\ i\ j)\ hs; \\
\quad \mathsf{let}\ i'\ =\ \mathsf{find}\ a\ i\ \mathsf{in} \\
\quad \mathsf{let}\ j'\ =\ \mathsf{find}\ a\ j\ \mathsf{in} \\
\quad i' = j'
\end{array}
$$

Soundness: "if it answers true then the goal is satisfied":

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

Advantage

No need to compute the proof.

Reification: pseudo code

$$
\begin{array}{l}
R_{\text{aux}}\ \text{hyps}\ = \\
\quad \text{match hyps with} \\
\quad \mid \langle \text{true} \rangle \mapsto [\ ] \\
\quad \mid \langle e_i \sim e_j \wedge \text{hyps'} \rangle \mapsto (i,j) :: R_{\text{aux}}\ \text{hyps'}
\end{array}
$$

$$
\begin{array}{l}
R\ = \\
\quad \text{match goal with} \\
\quad \mid \langle \text{hyps} \vdash e_i \sim e_j \rangle \mapsto (R_{\text{aux}}\ \text{hyps}, (i,j))
\end{array}
$$

$$
\begin{array}{l}
I : A \to \text{Prop}\ =\ (hs, (i,j)) \mapsto \\
\quad \text{let } hs\ =\ \text{map}\ ((i,j) \mapsto e_i \sim e_j)\ hs \text{ in} \\
\quad \text{let } hs\ =\ \text{fold}\ (\wedge)\ hs\ \text{true in} \\
\quad hs \Rightarrow e_i \sim e_j
\end{array}
$$