Experiments towards efficient computations

in formalized linear algebra

Maxime Dénès

Marelle Team, INRIA Sophia-Antipolis
ForMath European Project

November 10, 2010

General objective:

Efficient computations in linear algebra proofs.

General objective:

Efficient computations in linear algebra proofs.

Motivations

- Proofs relying on (concrete) heavy computations
- Usability of certified algorithms inside provers

Context

Context

- Calculus of Inductive Constructions

Context

- Calculus of Inductive Constructions
- Reduction rules encode computations

Context

- Calculus of Inductive Constructions
- Reduction rules encode computations
- Conversion rule embeds computational steps in reasoning

Context

- Calculus of Inductive Constructions
- Reduction rules encode computations
- Conversion rule embeds computational steps in reasoning

Context

- Calculus of Inductive Constructions
- Reduction rules encode computations
- Conversion rule embeds computational steps in reasoning

Conversion rule:

$$\frac{\Gamma \vdash U : s \quad \Gamma \vdash t : T \quad \Gamma \vdash T \equiv_{\beta\delta\iota\zeta} U}{\Gamma \vdash t : U}\ \text{conv}$$

In Coq, computational steps are often plugged in proofs through reflection.

$$\Gamma \vdash \mathtt{fP} : \forall (x : T), f(x) = \mathtt{true} \rightarrow P(x)$$

$$\Gamma, x : T \vdash \mathtt{fP}\, x\, (\mathtt{refl\_equal}\, \mathtt{true}) : P(x)$$

We distinguish small scale reflection:

```
Lemma existsP : forall P,
  reflect (exists x, P x) (existsb x, P x).
```

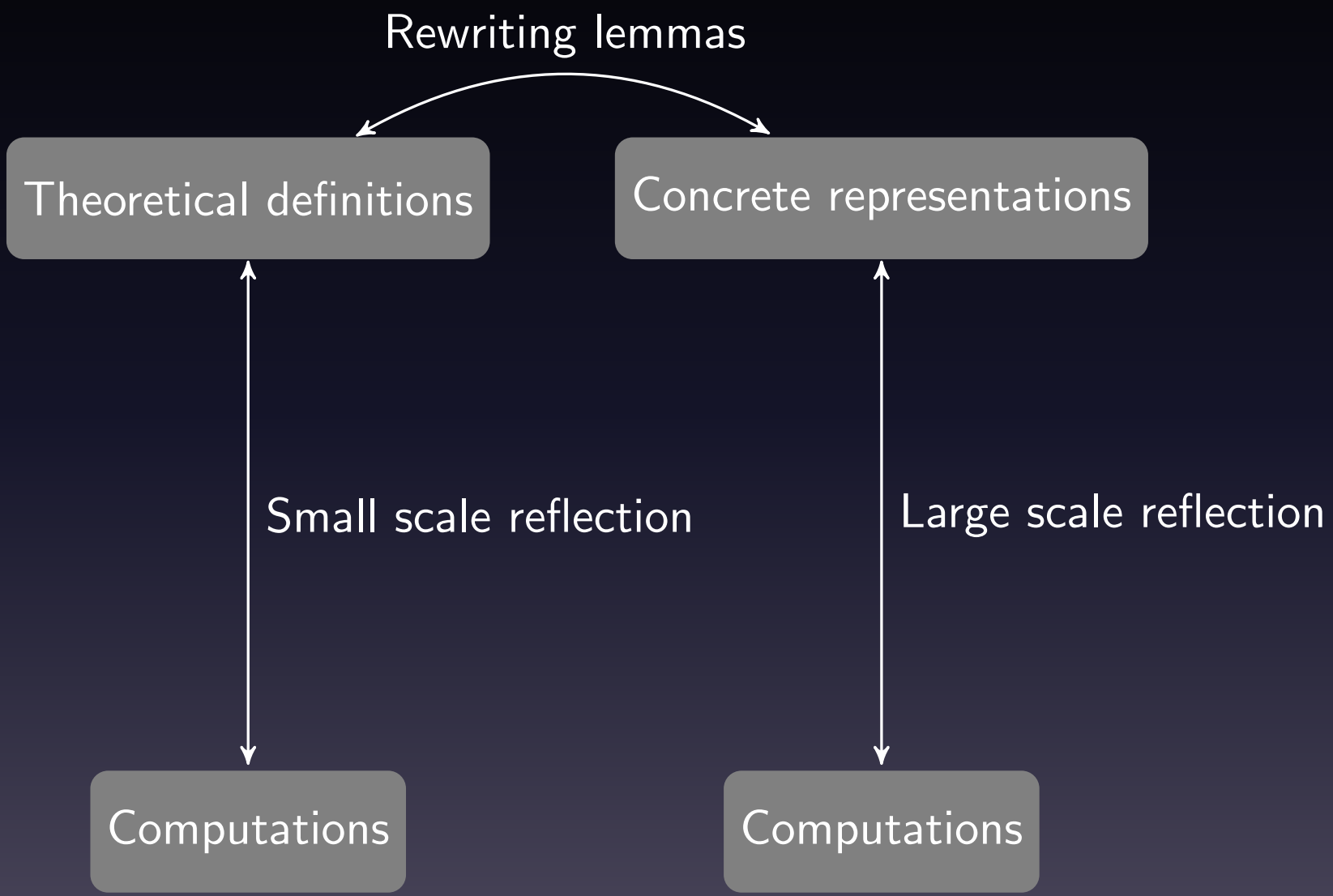
which enables computations in small steps, with fine grained control

and large scale reflection:

```
Lemma cfred232: cfreducible (Config 11 33 37 H 2 H 13
Y 5 H 10 H 1 H 1 Y 3 H 11 Y 4 H 9 H 1 Y 3 H 9 Y 6 Y 1
Y 1 Y 3 Y 1 Y Y 1 Y).
Proof. apply check_reducible_valid; by compute. Qed.
```

We would like to perform large scale reflection on objects from linear algebra (matrix,...).

The SSReflect extension (for the Coq proof assistant) provides powerful tools and theories on these objects, but not full computational features.


Rewriting lemmas
Theoretical definitions
Concrete representations
Small scale reflection
Large scale reflection
Computations
Computations

We need:

- Efficient techniques for evaluation
- Concrete and adapted representations
- Connections between abstract and concrete descriptions

Example: addition of matrices

```
Definition comp_matrix_add (M N:comp_matrix) :=
  mapi (fun i u => comp_row_add u (get N i)) M.

Lemma comp_matrix_addE: forall (M N:comp_matrix),
  (matrix_of_comp_matrix M) + (matrix_of_comp_matrix N)
  (matrix_of_comp_matrix (comp_matrix_add M N)).
```

Example: addition of matrices

```
Goal m1 + (m2 + m3) = m3 + (m2 + m1).
Proof.
rewrite /m1 /m2 /m3.
rewrite !comp_matrix_of_funE !comp_matrix_addE.
by apply:comp_matrix_eqP ; compute.
Qed.
```

Large scale reflection requires efficient techniques for evaluation of terms, that scale up well to heavy computations.

Weak reduction

Used in most functional languages.

```
# let rec f x = f x;;
  val f : 'a -> 'b = <fun>
# f;;
  - : 'a -> 'b = <fun>
# f 2;;
```

Weak reduction

Used in most functional languages.

```
# let rec f x = f x;;
  val f : 'a -> 'b = <fun>
# f;;
  - : 'a -> 'b = <fun>
# f 2;;
```

Strong reduction

Enables comparison of functions body.

```
Coq < Eval compute in (fun (x:nat) => (fun y => y) x).
      = fun x : nat => x
      : nat -> nat
```

Main issue

Which techniques for strong reduction ?

Main issue

Which techniques for strong reduction ?

Interpretation

Abstract machine manipulating a representation of terms

Main issue

Which techniques for strong reduction ?

Interpretation

Abstract machine manipulating a representation of terms

Example: standard evaluator of Coq (Barras, 1999)

- Uses a lazy strategy

Main issue

Which techniques for strong reduction ?

Interpretation

Abstract machine manipulating a representation of terms

Example: standard evaluator of Coq (Barras, 1999)

- Uses a lazy strategy
- Enables some control over heuristics

Main issue

Which techniques for strong reduction ?

Interpretation

Abstract machine manipulating a representation of terms

Example: standard evaluator of Coq (Barras, 1999)

- Uses a lazy strategy
- Enables some control over heuristics
- No specialization

Main issue

Which techniques for strong reduction ?

Interpretation

Abstract machine manipulating a representation of terms

Example: standard evaluator of Coq (Barras, 1999)

- Uses a lazy strategy
- Enables some control over heuristics
- No specialization
- Limited performance

Main issue

Which techniques for strong reduction ?

Interpretation

Abstract machine manipulating a representation of terms

Example: standard evaluator of Coq (Barras, 1999)

- Uses a lazy strategy
- Enables some control over heuristics
- No specialization
- Limited performance
- Better suited to small computations

Bytecode compilation (Grégoire and Leroy, 2002)

Bytecode compilation (Grégoire and Leroy, 2002)

- Encodes strong reduction of CIC by weak reduction of symbolic calculus

Bytecode compilation (Grégoire and Leroy, 2002)

- Encodes strong reduction of CIC by weak reduction of symbolic calculus
- Uses a modified version of OCaml's virtual machine

Bytecode compilation (Grégoire and Leroy, 2002)

- Encodes strong reduction of CIC by weak reduction of symbolic calculus
- Uses a modified version of OCaml's virtual machine
- Compiles terms to bytecode

Native code compilation

- Same framework as Coq's VM

Native code compilation

- Same framework as Coq's VM
- But produces OCaml source code, instead of bytecode

Native code compilation

- Same framework as Coq's VM
- But produces OCaml source code, instead of bytecode
- Enables the use of OCaml's standard native code compiler

Native code compilation

- Same framework as Coq's VM
- But produces OCaml source code, instead of bytecode
- Enables the use of OCaml's standard native code compiler

Native code compilation

- Same framework as Coq's VM
- But produces OCaml source code, instead of bytecode
- Enables the use of OCaml's standard native code compiler

Limits of this approach

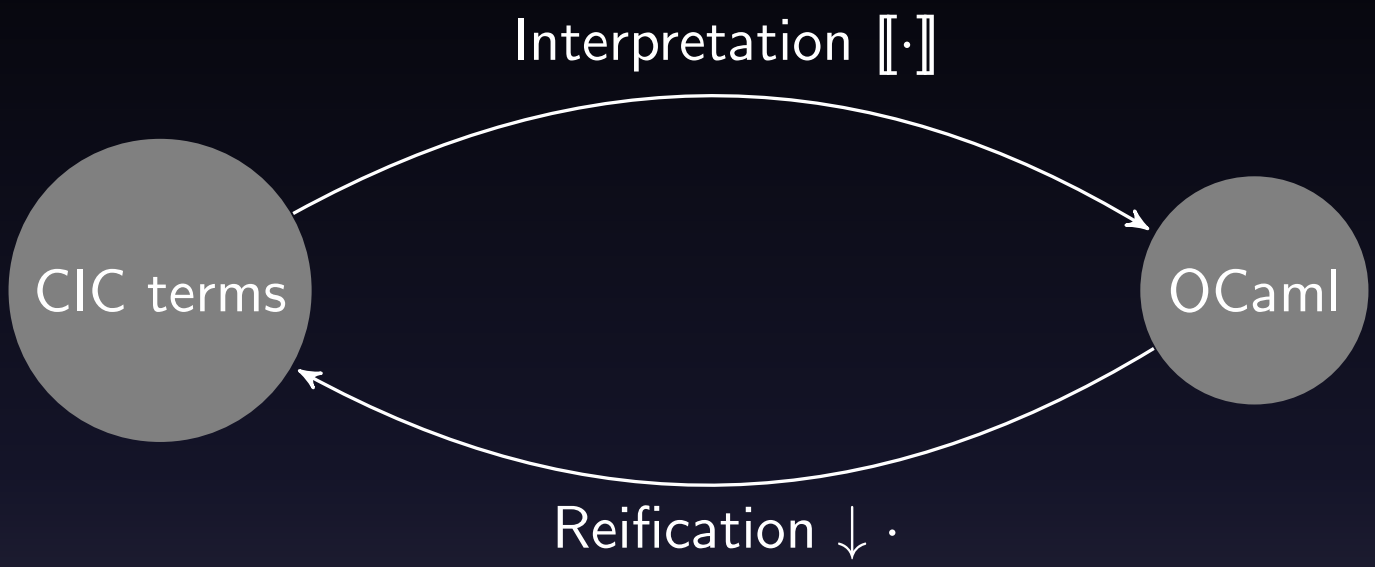
- Well suited only for full evaluation involving heavy computations

Native code compilation

- Same framework as Coq's VM
- But produces OCaml source code, instead of bytecode
- Enables the use of OCaml's standard native code compiler

Limits of this approach

- Well suited only for full evaluation involving heavy computations
- No control over the heuristics of the underlying evaluator



Where :

$$\forall t\, \forall u, t \equiv_{\beta\delta\iota\zeta} u \implies [\![t]\!] = [\![u]\!]$$

$$\forall t \text{ in normal form}, \downarrow [\![t]\!] = t$$

Representation of terms

$$\mathrm{Var} \ni \underline{x}$$
$$\mathrm{Term} \ni \underline{t} ::= \underline{x} | \underline{t_1}\ \underline{t_2} | \underline{v}$$
$$\mathrm{Val} \ni \underline{v} ::= \underline{\lambda x.t} | [\tilde{x}\ v_1 \ldots v_n]$$

Weak reduction of symbolic calculus

$$(\underline{\lambda x.\ t})\ \underline{v} \to_{\beta_v} \underline{t}[x \leftarrow v] \qquad (\beta_v)$$
$$[\tilde{x}\ v_1 \ldots v_n]\ \underline{v} \to_{\beta_s} [\tilde{x}\ v_1 \ldots v_n\ v] \qquad (\beta_s)$$

Strong reduction by iteration of weak reduction

$$\mathcal{N}(t) = \downarrow_0 [\![t]\!]_0$$
$$\downarrow_n \underline{\lambda x.t} = \lambda \widehat{n}.\ \downarrow_{n+1} ((\underline{\lambda x.t})\,[\tilde{n}])$$
$$\downarrow_n [\tilde{x}\, v_1 \dots v_n] = (n - x)\,(\downarrow_n \underline{v_1}) \dots (\downarrow_n \underline{v_n})$$

Key ideas of the implementation

- Applications are directly interpreted by the ones of the underlying language.

Key ideas of the implementation

- Applications are directly interpreted by the ones of the underlying language.
- Representation of accumulators requires to manipulate OCaml's internal objects.

Key ideas of the implementation

- Applications are directly interpreted by the ones of the underlying language.
- Representation of accumulators requires to manipulate OCaml's internal objects.
- Objective: encode β_s and β_v rules by OCaml's β-réduction.

Representation of accumulators

Usual OCaml's closures look like the following:

$tag = closure\_tag$	$code$	env

We just change the tag and adapt the code zone, so that the arguments accumulate.

$tag = 0$	$code$	env	$atome$	arg_1	$\cdots$	arg_n

Interpretation of terms of λ-calculus

$$\begin{aligned} [\![t_1\ t_2]\!]_n &= [\![t_1]\!]_n\ [\![t_2]\!]_n \\ [\![x]\!]_n &= \widehat{n - x} \\ [\![\lambda x.t]\!]_n &= \texttt{fun}\ \widehat{n}\ \texttt{->}\ [\![t]\!]_{n+1} \end{aligned}$$

Then we extend the calculus with inductive types, fixpoints, sorts,...

For the sake of efficiency, large scale reflection requires primitive structures natively interpreted in the underlying language (e.g. integers, arrays).

```
Register int : Set as int31_type.
Register add : int -> int -> int as int31_add.
```

```
⟦add⟧ = fun x y → if (is_int x) && (is_int y)
    then Int31.add x y else mk_id_accu "add" x y
```

Examples of applications :

- Primality tests
- SAT prover trace verification (Armand et al., 2010)
- Computations on concrete matrices (arrays)

Native arrays

- Seen inside Coq as usual functional arrays

Native arrays

- Seen inside Coq as usual functional arrays
- Interpreted by persistent arrays, based on OCaml's imperative features (Conchon and Filliâtre, 2007)

Native arrays

- Seen inside Coq as usual functional arrays
- Interpreted by persistent arrays, based on OCaml's imperative features (Conchon and Filliâtre, 2007)
- Provided primitive operations (fold, map,...)

Back to our matrices example

```
Definition comp_matrix := array (array T).

Definition comp_matrix_add (M N:comp_matrix) :=
mapi (fun i u => comp_row_add u (get N i)) M.
```

```
Goal m1 + (m2 + m3) = m3 + (m2 + m1).
Proof.
rewrite /m1 /m2 /m3.
rewrite !comp_matrix_of_funE !comp_matrix_addE.
by apply:comp_matrix_eqP ; compute.
Qed.
```

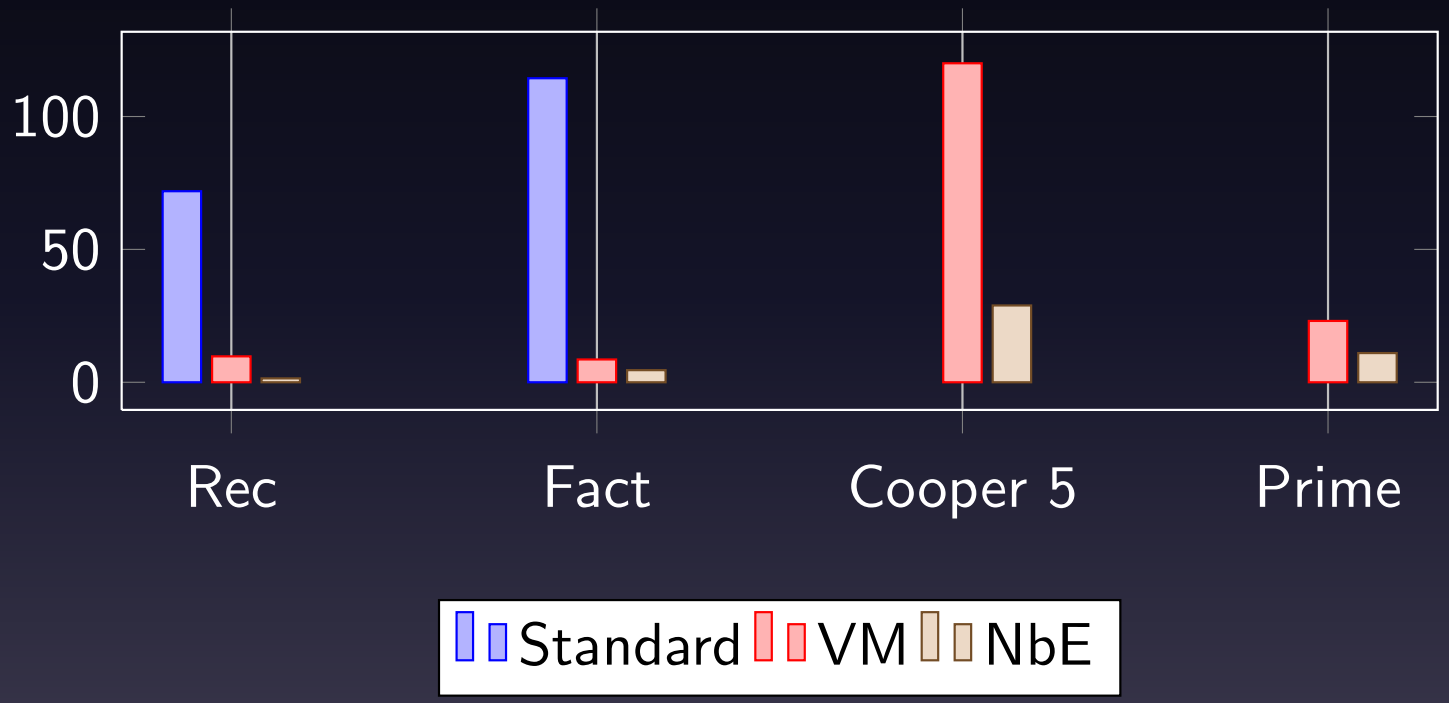
```
Goal m1 + (m2 + m3) = m3 + (m2 + m1).
Proof.
rewrite /m1 /m2 /m3.
rewrite !comp_matrix_of_funE !comp_matrix_addE.
by apply:comp_matrix_eqP ; native_compute.
Qed.
```

A few benchmarks

	Rec	Fact
Standard	71,89s (100 %)	114,4s (100 %)
VM Coq	9,75s (13,6 %)	8,58s (7,5 %)
NbE	1,46s (2,0 %)	4,56s (3,99 %)

	Cooper 5	Prime
Standard	untested	untested
VM Coq	120,03s (100 %)	23.08s (100 %)
NbE	28,9s (24 %)	10.99s (48 %)

A few benchmarks



What is (almost) done

- Native compilation of CIC terms
- Primitive integers and arrays in Coq's terms
- Compilation to native OCaml integers

Work in progress

- Universe constraints, coinductives and cofixpoints
- Compilation to native OCaml arrays
- Concrete implementation of SSReflect matrices and their operations

Perspectives

- Fully computable formalization of standard linear algebra algorithms
- Linear algebra proofs relying on heavy computations (suggestions welcome)