Experiments with computable matrices in the Coq system[1]

Maxime Dénès, Yves Bertot

Marelle Team, INRIA Sophia-Antipolis

September 5, 2011

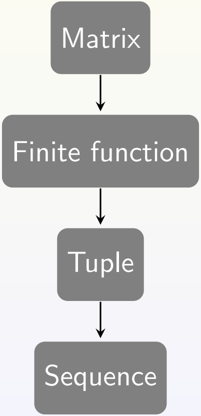
[1]This work has been partially funded by the FORMATH project, nr. 243847, of the FET program within the 7th Framework program of the European Commission.

SSReflect matrices

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```

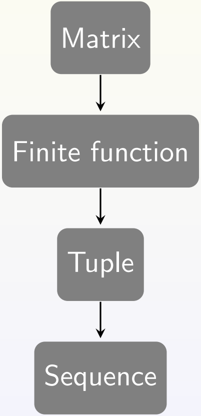
SSReflect matrices

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```



SSReflect matrices

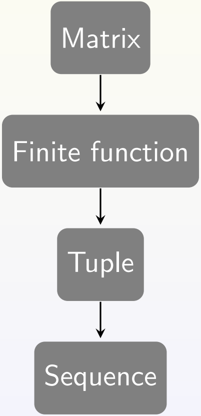
```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```



- Fine-grained architecture

SSReflect matrices

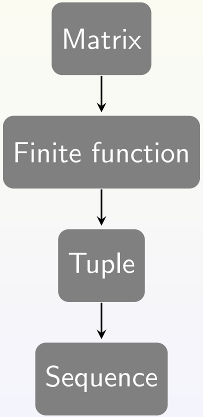
```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```



- Fine-grained architecture
- Easier to get the properties

SSReflect **matrices**

```
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.
```



- Fine-grained architecture
- Easier to get the properties
- Had to be locked to avoid term size explosion

Our objective

Our objective

- Define concrete and executable representations and operations on matrices

Our objective

- Define concrete and executable representations and operations on matrices
- Devise a way to link them with the theory in SSReflect

Our objective

- Define concrete and executable representations and operations on matrices
- Devise a way to link them with the theory in SSReflect
- Still be able to use convenient tools to reason about the algorithms we implement

Our objective

- Define concrete and executable representations and operations on matrices
- Devise a way to link them with the theory in SSReflect
- Still be able to use convenient tools to reason about the algorithms we implement

Our objective

- Define concrete and executable representations and operations on matrices
- Devise a way to link them with the theory in SSReflect
- Still be able to use convenient tools to reason about the algorithms we implement

Parametrized datatypes are useful in mathematical description, they can reduce the need for side conditions in specifications.

```
Variable M : matrix R (m1 + m2) (n1 + n2).
Lemma submxK :
block_mx (ulsubmx M) (ursubmx M) (dlsubmx M) (drsubmx M) = M.
```

A traditional approach to vectors

```
Inductive vect (A : Type) : nat -> Type :=
| Vnil : vect A 0
| Vcons : forall n, A -> vect A n -> vect A (S n).
```

- Traditionnally used to justify dependent datatypes
- Refers to a known weakness of conventional programming languages
  - Array overflow
  - Static verification of bound checking

```
Vnth : forall n p, p < n -> vect A n -> A
```

Problems with binary functions

```
Fixpoint Vzip A B C (op : A -> B -> C) n (v : vect A n) :=
  match v in vect _ n return vect B n -> vect C n with
    Vnil => fun w => Vnil
  | Vcons n' x v' => fun w : vect B (S n') =>
    match w in vect _ k return k = S n' -> vect C k with
      Vnil => fun _ => Vnil
    | Vcons p' y w' => fun h =>
      Vcons (op x y)
         (Vzip A B C op p'
            (eq_rect n' (vect A) v' _ (eqS h)) w')
    end (refl_equal _)
  end.
```

Problems with casts

- the expression `(eq_rect n' (vect A) v' p' ...)` changes the type of `v'`, not the value,
- It is a formally verified type cast,
- This clutters statements,
- Human beings prefer to be typeless in these conditions
- Proofs also become unwieldy
  - especially because of (refl_equal _)

Relaxed implementation

- Use directly unsized lists (or sequences in `ssreflect` jargon)
- No size guarantee expressed by result type
- The procedure terminates when the end of the shorter list is reached

```
Fixpoint zip A B C (op : A -> B -> C)
   (l1 : list A) (l2 : list B) :=
match l1, l2 with
  a::l1', b::l2' => op a b::zip A B C op l1' l2'
| _, _ => nil
end.
```

A library of effective matrices

Abstract definitions

A library of effective matrices

Abstract definitions

Algorithmic refinement

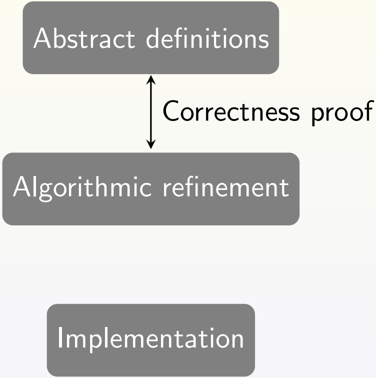
A library of effective matrices

Abstract definitions

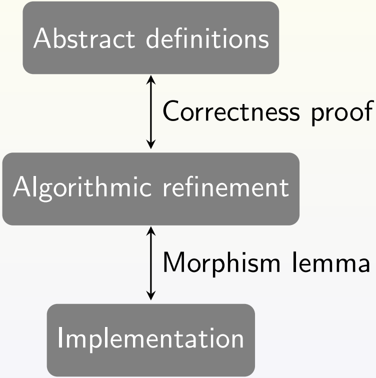
Algorithmic refinement

Implementation

A library of effective matrices



A library of effective matrices



List-based representation of matrices

```
Variable R : ringType.
Definition seqmatrix := seq (seq R).
```

List-based representation of matrices

```
Variable R : ringType.
Definition seqmatrix := seq (seq R).

Definition seqmx_of_mx (M:'M_(m,n)) : seqmatrix :=
 map (fun i => map (fun j => M i j) (enum 'I_n)) (enum 'I_m).
```

List-based representation of matrices

```
Variable R : ringType.
Definition seqmatrix := seq (seq R).

Definition seqmx_of_mx (M:'M_(m,n)) : seqmatrix :=
 map (fun i => map (fun j => M i j) (enum 'I_n)) (enum 'I_m).

Lemma seqmx_of_funE : forall (f : 'I_m -> 'I_n -> R),
  seqmx_of_mx (\matrix_(i < m, j < n) f i j) = mkseqmx_ord f.
```

List-based representation of matrices

```
Variable R : ringType.
Definition seqmatrix := seq (seq R).

Definition seqmx_of_mx (M:'M_(m,n)) : seqmatrix :=
 map (fun i => map (fun j => M i j) (enum 'I_n)) (enum 'I_m).

Lemma seqmx_of_funE : forall (f : 'I_m -> 'I_n -> R),
  seqmx_of_mx (\matrix_(i < m, j < n) f i j) = mkseqmx_ord f.

Lemma seqmx_eqP: forall (M N:'M[mT]_(m,n)),
  seqmx_of_mx M == seqmx_of_mx N -> M = N.
```

Large scale reflection on matrices

```
Goal \matrix_(i, j < n) (f i j) = \matrix_(i, j < n) (h i j).
Proof.
by apply/seqmx_eqP ; rewrite seqmx_of_funE ; native_compute.
Qed.
```

Addition

```
Definition addseqmx (M N : seqmatrix) : seqmatrix :=
  map2 (map2 (fun x y => x + y)) M N.

Lemma addseqmxE:
  {morph (@seqmx_of_mx m n) : M N / M + N >-> addseqmx M N}.
```

Naive product

```
Definition mulseqmx (M N: seqmatrix) : seqmatrix :=
  let N := transposeseqmx N in
  let f := foldl2 (fun z x y => x * y + z) 0 in
  map (fun r => map (f r) N) M.

Lemma mulseqmxE (M:'M_(m,p)) (N:'M_(p,n)) :
  mulseqmx (seqmx_of_mx M) (seqmx_of_mx N) =
    seqmx_of_mx (M *m N).
```

Winograd (1971)

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

Winograd (1971)

$$\left(\begin{array}{c|c} A_{1,1} & A_{1,2} \\ \hline A_{2,1} & A_{2,2} \end{array}\right) \times \left(\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline B_{2,1} & B_{2,2} \end{array}\right) = \left(\begin{array}{c|c} C_{1,1} & C_{1,2} \\ \hline C_{2,1} & C_{2,2} \end{array}\right)$$

$$\begin{array}{lll}
S_1 = A_{2,1} + A_{2,2} & P_1 = A_{1,1} \times B_{1,1} & U_1 = P_1 + P_6 \\
S_2 = S_1 - A_{1,1} & P_2 = A_{1,2} \times B_{2,1} & U_2 = U_1 + P_7 \\
S_3 = A_{1,1} - A_{2,1} & P_3 = S_4 \times B_{2,2} & U_3 = U_1 + P5 \\
S_4 = A_{1,2} - S_2 & P_4 = A_{2,2} \times T_4 & C_{1,1} = P_1 + P_2 \\
T_1 = B_{1,2} - B_{1,1} & P_5 = S_1 \times T_1 & C_{1,2} = U_3 + P_3 \\
T_2 = B_{2,2} - T_1 & P_6 = S_2 \times T_2 & C_{2,1} = U_2 - P_4 \\
T_3 = B_{2,2} - B_{1,2} & P_7 = S_3 \times T_3 & C_{2,2} = U_2 + P_5 \\
T_4 = T_2 - B_{2,1} & &
\end{array}$$

Fast matrix product

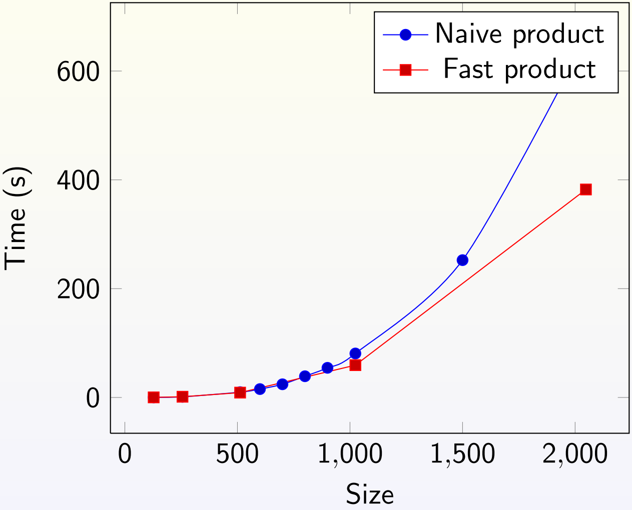
```
Fixpoint winograd {k} :=
  match k return let M := 'M[R]_(exp2 k) in M -> M -> M with
  | 0%N => fun A B => A *m B
  | l.+1 => fun A B =>
    if l <= C then A *m B else
    let A11 := ulsubmx A in
    ...
    let B22 := drsubmx B in
    let X := A11 - A21 in
    let Y := B22 - B12 in
    let C21 := winograd X Y in
    ...
    block_mx C11 C12 C21 C22
  end.

Lemma winogradP : forall n (M N : 'M[R]_(exp2 n)),
  (winograd M N) = (M *m N).
```

```
Fixpoint winogradI k :=
  match k return let M := seqmatrix R in M -> M -> M with
  | 0%N => fun A B => mulseqmx A B
  | l.+1 => fun A B =>
    if l <= C then mulseqmx A B else
    let off := exp2 l in
    let A11 := ulsubseqmx off off A in
    ...
    let B22 := drsubseqmx off off B in
    let X := subseqmx A11 A21 in
    let Y := subseqmx B22 B12 in
    let C21 := winogradI l X Y in
    ...
    block_seqmx C11 C12 C21 C22
  end.

Lemma winogradE :
 {morph (@seqmx_of_mx _ (exp2 k) (exp2 k)) :
    M N / winograd M N >-> winogradI k M N}.
```

Benchmarks


Naive product
Fast product
Time (s)
600
400
200
0
0
500
1,000
1,500
2,000
Size

Coq's arrays

- Implemented in a branch based on Coq's trunk
- Arrays are terms of a specific opaque type
- Equiped with operators and axioms
- Using persistent arrays (functional interface, imperative implementation)

Major improvements:

- Constant time access
- Efficient memory representation

Array-based matrices

Array-based row-major representation:

```
Variable R : ringType.
Definition arraymatrix := array (array R).
```

Link with matrices of the library:

```
Definition arraymx_of_mx n =
  match n with
  | O => fun M => make 0 (make 0 default)
  | p.+1 =>
      fun M => let k := of_Z (Z_of_nat n) in
      let row i :=
        let Mi := M (ord_of_int i) in
        init k (fun j => Mi (ord_of_int _ j)) default
      in
      init k row (make 0 default)
  end.
```

Array size is limited, so lemmas express size constraints:

```
Hypothesis Hn0 : n <= max_array_length.

Lemma int_of_ordK : cancel (@int_of_ord n) ord_of_int.
Lemma ord_of_intK :
  {in [pred i| (i < of_Z (Z_of_nat n))%int31],
  cancel ord_of_int (@int_of_ord n)}.
```

Relaxed determinant implementation

- Matrix as sequences of sequences contain too little dimension information
- Extra argument for the dimension
- Expansion along the first column
  - Internal recursion for the expansion function
  - Use `behead` to remove first elements of each row
  - Use `rcons` to add an element at the end of sequence

Determinant code

```
Fixpoint sdet n (mat : seq (seq R)) : R :=
  if n is n'.+1 then
    let fix expand upper lower :=
      if lower is (a :: v) :: lower' then
        let minor := sdet n' (upper ++ map behead lower') in
        a * minor - expand (rcons upper v) lower'
      else 0
    in expand [::] mat
  else 1.
Lemma sdet_correct n M :
  size M <= n -> sdet n M = \det (mx_of_seqmx n n M).
Proof.
  (* 32 line proof *)
```

Determinant code (separating expand)

- expand is unaware of the matrix dimension

```
Fixpoint expand sign (mat : seq (seq R)) det acc : R :=
  if mat is a :: tl then
    sign * head 0 a * det (rev acc ++ (map behead tl)) +
    expand (- sign) tl det (behead a :: acc)
  else
    0.
```

- expand is unaware of the matrix dimension

```
Fixpoint sdet n (mat : seq (seq R)) : R :=
  if n is p.+1 then expand 1 mat (sdet p) [::] else 1.
```

Separate statement for expand

```
Lemma expand_correct :
  forall mat p k l d s acc v (h :(p.+1 = k + l)%N),
   size acc = k -> size mat <= l -> size v = k ->
    (forall mat', size mat' <= p ->
           d mat' = \det (mx_of_seqmx p p mat')) ->
    let r := mx_of_seqmx (k+l) p.+1 (rev (czip v acc) ++ mat)
         in
    expand s mat d acc =
      s * (-1)^+ k * \sum_(i < l) r (rshift k i) 0 *
      cofactor (castmx (refl_equal (k + l)%N, h) r)
         (rshift k i) (cast_ord h ord0).
Proof.
  (* 48 line proof *)
Qed.
```

Failed attempt

```
Lemma expand_correct :
 forall mat p k l d s acc (v:'M_(k,1))(h:(p.+1= k + l)%N),
   size acc = k -> size mat = l ->
    (forall mat', size mat' <= p ->
       d mat' = \det (mx_of_seqmx p p mat')) ->
    let ref := col_mx (row_mx v (mx_of_seqmx k p (rev acc)))
                (mx_of_seqmx l p.+1 mat) in
    expand s mat d acc =
      s * (-1)^+ k * \sum_(i < l) ref (rshift k i) 0 *
      cofactor (castmx ((refl_equal (k + l)%nat), h) ref)
         (rshift k i) (cast_ord h ord0).
```

Ongoing work

Ongoing work

- Development of a full library of effective linear algebra

Ongoing work

- Development of a full library of effective linear algebra
- Better interface with machine integers

Ongoing work

- Development of a full library of effective linear algebra
- Better interface with machine integers
- Other representations (sparse matrices,...)

Ongoing work

- Development of a full library of effective linear algebra
- Better interface with machine integers
- Other representations (sparse matrices,...)
- In place operations

Conclusion

Conclusion

- General approach to organize execution-oriented and properties-oriented definitions

Conclusion

- General approach to organize execution-oriented and properties-oriented definitions
- The usual pitfall: trying to prove correctness on a concrete implementation

Conclusion

- General approach to organize execution-oriented and properties-oriented definitions
- The usual pitfall: trying to prove correctness on a concrete implementation
- We make heavy use of the computational capabilities of Coq

Conclusion

- General approach to organize execution-oriented and properties-oriented definitions
- The usual pitfall: trying to prove correctness on a concrete implementation
- We make heavy use of the computational capabilities of Coq
- We have short proofs (4 lines for partial Winograd)

Conclusion

- General approach to organize execution-oriented and properties-oriented definitions
- The usual pitfall: trying to prove correctness on a concrete implementation
- We make heavy use of the computational capabilities of Coq
- We have short proofs (4 lines for partial Winograd)
- Realistic size problems can already be treated (4096×4096 dense matrices)

Thank you !