Final Report for Deliverable Nr. 2.3

Executable library of Basic Constructive Algebra

Responsible: Thierry Coquand, coquand@chalmers.se
Written by: Anders Mörtberg, mortberg@chalmers.se
Site: University of Gothenburg, Sweden

Deliverable Date: July 2013

1 Introduction

This report presents the formalization of two notions in constructive algebra: the *Sasaki-Murao algorithm* for computing the characteristic polynomial of a matrix over any commutative ring in polynomial time and, *coherent* and *strongly discrete rings* which are rings on which it is possible to solve systems of equations.

2 Sasaki-Murao algorithm

This section presents of a formal proof of the Sasaki-Murao algorithm [17]. This is an elegant algorithm for computing the characteristic polynomial (and hence determinant) of a square matrix over an arbitrary commutative ring in polynomial time. Usual presentations of this algorithm are quite complex, and rely on some Sylvester identities [1]. We believe that the proof we shall present, which was obtained by formalizing this algorithm in type theory (more precisely in the SSReflect [10] extension to Coq [18]) is simpler. It does not rely on Sylvester identities and indeed gives a proof of some of them as corollaries. It provides also a good example of how one can use a library of formalized mathematical results to prove formally a computer algebra program. The whole formalization can be found at [16].

2.1 Matrices

For any $n \in \mathbb{N}$, we define $I_n = \{i \in \mathbb{N} \mid i < n\}$ (with $I_0 = \emptyset$). If R is a set, a $m \times n$ matrix of elements of the set R is a function $I_m \times I_n \to R$. We can also view any such matrix as a family of elements (m_{ij}) for $i \in I_m$ and $j \in I_n$.

If M is a $m \times n$ matrix, f a function of type $I_p \to I_m$ and g a function of type $I_q \to I_n$, we define the $p \times q$ sub-matrix[1] $M(f, g)$ by

$$M(f, g)(i, j) = M(f\ i, g\ j)$$

[1] In the usual definition of sub-matrix, only some lines and columns are removed, which would be enough for the following proofs. But our more general definition make the Coq formalization easier to achieve.

We often use the following operation on finite maps: if $f : I_p \to I_m$, we defined $f^+ : I_{1+p} \to I_{1+m}$ such that

$$\begin{array}{rcl} f^+0 & = & 0 \\ f^+(1+x) & = & 1+(f\ x) \end{array}$$

If R is a ring, let 1_n be the $n \times n$ identity matrix. We can also define addition and multiplication of matrices as usual. We can decompose a nonempty $m \times n$ matrix M in four components:

- the top-left element m_{00}, which is an element of R
- the top-right line vector $L = m_{01}, m_{02}, \ldots, m_{0(n-1)}$
- the bottom-left column vector $C = m_{10}, m_{20}, \ldots, m_{(m-1)0}$
- the bottom-right $(m-1) \times (n-1)$ matrix $N_{ij} = m_{(1+i,1+j)}$

$$\left(\begin{array}{c|c} m_{00} & L \\ \hline C & N \end{array}\right)$$

With this decomposition, we define the central operation of our algorithm, which defines a $(m-1) \times (n-1)$ matrix:

$$M' = m_{00}N - CL$$

This operation $M \longmapsto M'$ transforms a $m \times n$ matrix into a $(m-1) \times (n-1)$ matrix is crucial in the Sasaki-Murao algorithm. In the special case where $m = n = 2$ the matrix M' (of size 1×1) can be identified with the determinant of M.

Lemma 2.1 *For any $m \times n$ matrix M, for any map $f : I_p \to I_{m-1}$ and any map $g : I_q \to I_{n-1}$, we have the following identity:*

$$M'(f,g) = M(f^+, g^+)'$$

Proof. This lemma is easy to prove once one has realized two facts:

1. Selecting a sub-matrix commutes with most of the basic operations about matrices. In particular, $(M-N)(f,g) = M(f,g) - N(f,g)$, $(aM)(f,g) = aM(f,g)$. For multiplication, we have $(MN)(f,g) = M(f,id)N(id,g)$ where id is the identity function.
2. For any matrix M described as a block $(r\ L\ C\ N)$, we have that $M(f^+, g^+)$ is the block $(r\ L(id,g)\ C(f,id)\ N(f,g))$

From this two observations, we then have:

$$\begin{array}{rcl} M'(f,g) & = & (rN - Cl)(f,g) \\ & = & rN(f,g) - C(f,id)L(id,g) \end{array}$$

and

$$M(f^+, g^+)' = rN(f,g) - C(f,id)L(id,g)$$

So, we can conclude that $M'(f,g) = M(f^+, g^+)'$. $\square$

The block decomposition suggests the following possible representation of matrices in a functional language using the data type (where $[R]$ is the type of lists over the type R, using HASKELL notation):

$$Mat\ R\ ::=\ Empty \mid Mat\ R\ [R]\ [R]\ (Mat\ R)$$

So a matrix M is either the empty matrix or a compound matrix $Mat\ m\ L\ C\ N$. It is direct, using this representation, to define the operations of addition, multiplication on matrices, and the operation M' on non-empty matrices. From this representation, we can also compute other standard views of a $m \times n$ matrix, such as a list of lines $l_1, \ldots, l_m$ or as a list of columns $c_1, \ldots, c_n$.

If M is a square $n \times n$ matrix over a ring R we write $|M|$ the determinant of M. A *k-minor* of M is a determinant $|M(f, g)|$ for any strictly increasing maps $f : I_k \to I_n$ and $g : I_k \to I_n$. A leading principal minor of M is a determinant $|M(f, f)|$ where f is the inclusion of I_k into I_n.

2.2 The algorithm

We present Sasaki-Murao algorithm using functional programming notations. This algorithm computes in polynomial time, not only the determinant of a matrix, but also its characteristic polynomial. We assume that we have a representation of polynomials over the ring R and that we are given an operation p/q on $R[X]$ which should be the quotient of p by q when q is a *monic* polynomial. This operation is directly extended to an operation M/q of type $Mat\ R[X] \to R[X] \to Mat\ R[X]$. We define then an auxiliary function $\phi\ a\ M$ of type $R[X] \to Mat\ R[X] \to R[X]$. The definition is:

$$\begin{array}{lcl} \phi\ a\ Empty & = & a \\ \phi\ a\ (Mat\ m\ L\ C\ N) & = & \phi\ m\ ((mN - CL)/a) \end{array}$$

From now on, we assume R to be a commutative ring.

The proof relies on the notion of *regular* element of a ring: a *regular* element of R is an element a such that $ax = 0$ implies $x = 0$. An alternative (and equivalent) definition is to say that multiplication by a is injective or that a can be cancelled from $ax = ay$ giving $x = y$.

Theorem 2.2 *Let P be a square matrix of elements of $R[X]$. If all leading principal minors of P are monic, then $\phi\ 1\ P$ is the determinant of P. In particular, if $P = X1_n - M$ for some square matrix M of elements in R, $\phi\ 1\ P$ is the* characteristic polynomial *of M.*

This gives a remarkably simple (and polynomial time [1]) algorithm for computing the characteristic polynomial $\chi_M(X)$ of a matrix M. The determinant of M is then $\chi_{-M}(0)$.

2.3 Correctness proof

We first start to prove some auxiliary lemmas:

Lemma 2.3 *If M is a $n \times n$ matrix, $n > 0$ then we have*

$$m_{00}^{n-1}|M| = m_{00}|M'|.$$

In particular, if m_{00} is regular and $n > 1$, then we have

$$m_{00}^{n-2}|M| = |M'|.$$

Proof. Let us view the matrix M as a list of lines $l_0, \ldots, l_{n-1}$ and let N_1 be the matrix $l_0,\ m_{00}l_1, \ldots,\ m_{00}l_{n-1}$. The matrix N_1 is computed from M by multiplying all of its lines (except the first one) by m_{00}. By the properties of the determinant, we can assert that $|N_1| = m_{00}^{n-1}|M|$.

Let N_2 be the matrix $l_0,\ m_{00}l_1 - m_{10}l_0,\ \ldots,\ m_{00}l_{n-1} - m_{(n-1)0}l_0$. The matrix N_2 is computed from N_1 by subtracting a multiple of l_0 from every line except l_0:

$$m_{00}l_{1+i} \leftarrow m_{00}l_{1+i} - m_{(1+i)0}l_0.$$

By the properties of the determinant, we can assert that $|N_2| = |N_1|$.

Using the definition of the previous subsection, we can also view the matrix M as the block matrix $(m_{00}\ L\ C\ N)$, and then the matrix N_2 is the block matrix $(m_{00}\ L\ 0\ M')$. Hence we have $|N_2| = m_{00}|M'|$. From this equality, we can now prove that

$$m_{00}^{n-1}|M| = |N_1| = |N_2| = m_{00}|M'|.$$

If m_{00} is regular and $n > 2$, this equality simplifies to $m_{00}^{n-2}|M| = |M'|$ □

Corollary 2.4 *Let M be a $n \times n$ matrix with $n > 0$. If f and g are two strictly increasing maps from I_k to I_{n-1}, then $|M'(f,g)| = m_{00}^{k-1}|M(f^+, g^+)|$ if m_{00} is regular.*

Proof. Using Lemma 2.1, we know that $M'(f,g) = M(f^+, g^+)'$, so this corollary follows from Lemma 2.3. □

Let a be an element of R and M a $n \times n$ matrix. We say that a and M are related if and only if

1. a is regular
2. a^k divides each $k+1$ minor of M
3. each principal minor of M is regular

Lemma 2.5 *Let a be a regular element of R and M a $n \times n$ matrix, with $n > 0$. If a and M are related, then a divides every element of M'. Furthermore if $aN = M'$ then m_{00} and N are related and if $n > 1$*

$$m_{00}^{n-2}|M| = a^{n-1}|N|$$

Proof. Let us start by stating two simple facts: m_{00} is a 1×1 principal minor of M and for all i, j, M'_{ij} is a 2×2 minor of M. Therefore, since a and M are related, m_{00} is regular and a divides all the M'_{ij} (by having $k = 1$), so a divides M'.

Let us write $M' = aN$, we now need to show that m_{00} and N are related, and if $n > 1$,

$$m_{00}^{n-2}|M| = a^{n-1}|N|$$

Let us consider two strictly increasing maps $f : I_k \to I_{n-1}$, $g : I_l \to I_{n-1}$, we have $|M'(f, g)| = u^{k-1}|M(f^+, g^+)|$ by Corollary 2.4. From the definition of related, we also know that a^k divides $|M(f^+, g^+)|$. Since $M' = aN$ we have $|M'(f, g)| = a^k|N(f, g)|$. If we write $ba^k = |M(f^+, g^+)|$, we have that $ba^k u^{k-1} = a^k|N(f, g)|$. Since a is regular, this equality implies $bu^{k-1} = |N(f, g)|$, and we see that u^{k-1} divides each k minor of N. This also shows that $|N(f, g)|$ is regular whenever $|M(f^+, g^+)|$ is regular. In particular, each principal minor of N is regular. Finally, since $|M'| = a^{n-1}|N|$ we have $m_{00}^{n-2}|M| = a^{n-1}|N|$ by Lemma 2.3. □

Since any monic polynomial is also a regular element of the ring of polynomials, Theorem 2.2 follows directly from Lemma 2.5 by performing a straightforward induction over the size n. In the case where P is $X1_n - M$ for some square matrix M over R, we can use the fact that any principal minor of $X1_n - M$ is the characteristic polynomial of a smaller matrix, and thus is always monic. In the end, the second part of the conclusion follows directly for the first: $\phi\ 1\ (X1_n - M) = \chi_M(X)$.

Now, we explain how to derive some Sylvester equalities from Lemma 2.5. If we look at the computation of $\phi\ 1\ P$ we get a chain of equalities

$$\phi\ 1\ P = \phi\ u_1\ P_1 = \phi\ u_2\ P_2 = \cdots = \phi\ u_{n-1}\ P_{n-1}$$

and we have that u_k is the $k : th$ leading principal minor of P, while P_k is the $(n-k) \times (n-k)$ matrix

$$P_k(i, j) = |P(f_{i,k}, f_{j,k})|$$

where $f_{i,k}(l) = l$ if $l < k$ and $f_{i,k}(k) = i + k$. (We have $P_0 = P$.) Lemma 2.5 shows that we have for $k < l$

$$|P_k|u_l^{n-l-1} = |P_l|u_k^{n-k-1}$$

This is a Sylvester equality for the matrix $P = X1_n - M$. If we evaluate this identity at $X = 0$, we get the corresponding Sylvester equality for the M matrix over an arbitrary commutative ring.

2.4 Representation in type theory

As we previously said, this is a simple functional program, but its correctness involves non-trivial mathematics. We choose to use the SSReflect library to formalize the proof because it already contains many results that we need. The main scheme is to translate this program using SSReflect data types, prove its correctness and then prove that both implementations output the same results on valid inputs following the methodology presented in [8].

First, here is a description of the SSReflect data types we need:

```
(* 'I_n *)
Inductive ordinal (n: nat) : predArgType := Ordinal m of m < n.

Variable R : ringType.

(* 'M[R]_(m,n) a.k.a. 'M_(m,n) *)
Inductive matrix R m n := Matrix of {ffun 'I_m * 'I_n -> R}.

(* {poly R} *)
Record polynomial := Polynomial {
    polyseq :> seq R;
    _ : last 1 polyseq != 0
}.
```

Here dependent types are used to express well-formedness. For example, polynomials are encoded as lists (of their coefficients) with a proof that the last one is not zero. With this restriction, we are sure that one list exactly represent a unique polynomial. Matrices are described as finite functions over the finite sets of indexes.

With this definition, it is easy to define the sub-matrix $M(f, g)$ along with minors:

```
(* M(f,g) *)
Definition submatrix m n p q (f : 'I_p -> 'I_m) (g : 'I_q -> 'I_n)
  (A : 'M[R]_(m,n)) : 'M[R]_(p,q) :=
  \matrix_(i < p, j < q) A (f i) (g j).

Definition minor m n p (f : 'I_p -> 'I_m) (g : 'I_p -> 'I_n)
  (A : 'M[R]_(m,n)) : R := \det (submatrix f g A).
```

Using SSReflect notations and types, we can now write the steps of the functional program (where rdivp is the pseudo-division operation [11] of $R[X]$):

```
Fixpoint bareiss_rec m a : 'M[{poly R}]_(1 + m) -> {poly R} :=
  match m return 'M[_]_(1 + m) -> {poly R} with
    | S p => fun (M: 'M[_]_(1 + _)) =>
      let d := M 0 0 in (* up left *)
      let l := ursubmx M in (* up right *)
      let c := dlsubmx M in (* down left *)
      let N := drsubmx M in (* down right *)
      let M' := d *: N - c *m l in
      let M'' := map_mx (fun x => rdivp x a) M' in
        bareiss_rec d M''
    | _ => fun M => M 0 0
  end.

Definition bareiss n (M : 'M[{poly R}]_(1 + n)) := bareiss_rec 1 M.

Definition bareiss_char_poly n (M : 'M[R]_(1 + n)) :=
  bareiss (char_poly_mx M).

Definition bdet n (M : 'M[R]_(1 + n)) := (bareiss_char_poly (-M))`_0.
```

The main achievement of this paper is the formalized proof of correctness (detailed in the previous subsection) of this program:

```
Lemma bareissE n (M : 'M[{poly R}]_(1 + n))
```

```
  (H : forall p (h h' : p < 1 + n), pminor h h' M \is monic) :
  bareiss M = \det M.

Lemma bareiss_char_polyE n (M : 'M[R]_(1 + n)) :
  bareiss_char_poly M = char_poly M.

Lemma bdetE n (M : 'M[R]_(1 + n)) : bdet M = \det M.
```

The formal correctness proof is included in appendix 4 which shows that the length of the formal proof is in fact shorter than the paper proof in the previous section.

In order to be able to compute more efficiently we rely on CoqEAL [7], a library built on top of SSReflect libraries that we are currently developing. It allows to mirror the main algebraic hierarchy of SSReflect with more concrete data types, for example, here we mirror the matrix type 'M[R]_(m,n) by the concrete type Matrix:

```
Inductive Matrix : Type :=
 | eM (* the empty matrix *)
 | cM of CR & seq CR & seq CR & Matrix.
```

We can easily prove that translating a SSReflect matrix into a Matrix always lead to a "valid" Matrix, and there is a bijection between SSReflect matrices and "valid" matrices, so we are sure that our program computes the correct determinant for all valid inputs.

2.5 Computations

In this section the formalization of a polynomial time algorithm for computing the determinant over any commutative ring has been presented. In order to be able to do the formalization in a convenient way a new correctness proof more suitable for formalization has been found. The formalized algorithm has also been refined to a more efficient version on simple types, following the methodology of [8]. This work can be seen as an indication that this methodology works well on more complicated examples involving many different computable structures, in this case matrices of polynomials.

We have tested the implementation on randomly generated matrices with $\mathbb{Z}$ coefficients:

```
(* Random 3x3 matrix *)
Definition M3 :=
  cM 10%Z [:: (−42%Z); 13%Z] [:: (−34)%Z; 77%Z]
     (cM 15%Z [:: 76%Z] [:: 98%Z]
          (cM 49%Z [::] [::] (@eM _ _))).

Time Eval vm_compute in ex_bdet 3 M3.
     = (−441217)%Z
  Finished transaction in 0. secs (0.006667u,0.s)

Definition M10 := (* Random 10x10 matrix *).

Time Eval vm_compute in ex_bdet 10 M10.
     = (−406683286186860)%Z
  Finished transaction in 1. secs (1.316581u,0.s)

Definition M20 := (* Random 20x20 matrix *).
```

```
Time Eval vm_compute in ex_bdet 20 M20.
     = 757280501074819691276943718 61%Z
  Finished transaction in 63. secs (62.825904u,0.016666s)
```

This indicates that the implementation is indeed quite efficient, we believe that the slow-down of the last computation is due to the fact that the size of the determinant is so large and that the intermediate arithmetic operations has to be done on very big numbers. We have verified this by extracting the function to Haskell and the determinant of the 20×20 matrix can then be computed in 0.273 seconds. The main reasons for this is that the Haskell program has been compiled and have an efficient implementation of arithmetic operations for large numbers.

3 Coherent and strongly discrete rings

We present a formalization of coherent and strongly discrete rings in type theory. This is a fundamental structure in constructive algebra that represents rings in which it is possible to solve linear systems of equations. These structures have been instantiated with Bézout domains (for instance $\mathbb{Z}$ and $k[x]$) and Prüfer domains (generalization of Dedekind domains) so that we get certified algorithms solving systems of equations that are applicable on these general structures. This work can be seen as basis for developing a formalized library of linear algebra over rings.

3.1 Coherent rings

Given a ring R (in our setting commutative but it is possible to consider non-commutative rings as well [2]) one important problem to study is how to solve linear systems over R. If R is a field, then we have a nice description of the space of solution by a basis of solutions. Over an arbitrary ring R there is in general no basis.[2] But an important weaker property is that there is a finite number of solutions which generate all solutions. We say that the ring is *coherent* if this is the case.

More concretely, given a rectangular matrix M over R we want to find a finite number of solutions $X_1, \ldots, X_n$ of the system $MX = 0$ such that any solution is of the form $a_1X_1 + \cdots + a_nX_n$ where $a_1, \ldots, a_n \in R$. If this is possible, we say that the module of solutions of the system $MX = 0$ is finitely generated. This can be reformulated with matrices: we want to find a matrix L such that

$$MX = 0 \;\leftrightarrow\; \exists Y.X = LY$$

A ring is *coherent* if for any matrix M it is possible to compute a matrix L such that this holds. If this is the case it follows that $ML = 0$.

For this it is enough to consider the case where M has only one line. Indeed, assume that for any $1 \times n$ matrix M we can find a $n \times m$ matrix L such that $MX = 0$ iff $X = LY$ for

[2] For instance over the ring $R = k[X, Y, Z]$ where k is a field, the equation $pX + qY + rZ = 0$ has no basis of solutions. It can be shown that a generating system of solutions is given by $(-Y, X, 0)$, $(Z, 0, -X)$, $(0, -Z, Y)$.

some Y. To solve the system

$$M_1X = \cdots = M_kX = 0$$

where each M_i is a $1 \times n$ matrix first compute L_1 such that $M_1X = 0$ iff $X = LY_1$ for some Y_1. Next compute L_2 such that $M_2L_1Y_1 = 0$ iff $Y_1 = L_2Y_2$. At the end we obtain $L_1, \ldots, L_k$ such that $M_1X = \cdots = M_kX = 0$ iff X is of the form $L_1 \cdots L_kY$ and so $L_1 \cdots L_k$ provide a system of generators for the solution of the system.

Hence it is sufficient to formulate the condition for coherent rings as: For any *row* matrix M it is possible to find a matrix L such that

$$MX = 0 \;\leftrightarrow\; \exists Y.X = LY$$

Note that the notion of coherent is not stressed in classical presentations of algebra since Noetherian rings are automatically coherent, but in a computationally meaningless way. It is however fundamental, both conceptually [13, 14] and computationally. The system HOMALG [3] for instance takes this notion as the central one.

In the development, coherent rings have been implemented as in [9] using the Canonical Structure mechanism of COQ. In general the size of L cannot be predicted so we need an extra function that computes this:

```
Record mixin_of (R : ringType) : Type := Mixin {
 size_solve : forall m, 'rV[R]_m -> nat;
 solve_row : forall m (V : 'rV[R]_m), 'M[R]_(m,size_solve V);
 _ : forall m (V : 'rV[R]_m) (X : 'cV[R]_m),
  reflect (exists Y : 'cV[R]_(size_solve V), X = solve_row V *m Y)
            (V *m X == 0)
}.
```

Here *m denotes matrix multiplication and V *m X == 0 is the boolean equality of matrices, so the specification says that this equality is reflected by the existence statement. An alternative to having a function computing the size would be to output a dependent pair but this has the undesired behavior that the pair has to be destructed when stating lemmas about it which in turn would mean that these lemmas would be cumbersome to use as it would not be possible to rewrite with them directly.

Using this we have implemented the algorithm for computing the generators of a system of equations:

```
Fixpoint solveMxN (m n : nat) :
  forall (M : 'M_(m,n)), 'M_(n,size_solveMxN M) :=
   match m return forall M : 'M_(m,n), 'M_(n,size_solveMxN M) with
    | S p => fun (M : 'M_(1 + _,n)) =>
      let L1 := solve_row (usubmx M)
      in L1 *m solveMxN (dsubmx M *m L1)
    | _ => fun _ => 1%:M
   end.

Lemma solveMxNP : forall m n (M : 'M[R]_(m,n)) (X : 'cV[R]_n),
  reflect (exists Y : 'cV_(size_solveMxN M), X = solveMxN M *m Y)
            (M *m X == 0).
```

In order to instantiate this structure one can of course directly give an algorithm that computes the solution of a single row system. However there is another approach that will be used in the rest of the paper that is based on the intersection of finitely generated ideals.

3.1.1 Ideal intersection and coherence

In the case when R is an integral domain one way to prove that R is coherent is to show that the intersection of two finitely generated ideals is again finitely generated. This amounts to given two ideals $I = (a_1, \ldots, a_n)$ and $J = (b_1, \ldots, b_m)$ compute generators $(c_1, \ldots, c_k)$ of $I \cap J$. For $I \cap J$ to be the intersection of I and J it should satisfy

$$I \cap J \subseteq I$$

$$I \cap J \subseteq J$$

$$\forall x.\ x \in I \wedge x \in J \rightarrow x \in I \cap J$$

The first two of these mean that the generators of $I \cap J$ should be possible to write as a linear combination of the generators of both I and J. The third property states that if x can be written as a linear combination of the generators of I and J then it can be written as a linear combination of the generators of $I \cap J$.

A convenient way to express this in Coq is to use strongly discrete rings, which are discussed in section 3.2. For now we just assume that we can find matrices V and W such that $IV = I \cap J$ and $JW = I \cap J$ (with matrix multiplication and ideals represented by row-vectors containing the generators). Using this there is an algorithm to compute generators of the solutions of a system:

$$m_1 x_1 + \cdots + m_n x_n = 0$$

The main idea is to compute generators, M_0, of the solution for $m_2 x_2 + \cdots + m_n x_n = 0$ by recursion and also compute generators $t_1, \ldots, t_p$ of $(m_1) \cap (-m_2, \ldots, -m_n)$ together with V and W such that

$$(m_1)V = (t_1, \ldots, t_p)$$
$$(-m_2, \ldots, -m_n)W = (t_1, \ldots, t_p)$$

The generators of the module of solutions are then given by:

$$\begin{bmatrix} V & 0 \\ W & M_0 \end{bmatrix}$$

This has been implemented by:

```
Fixpoint solve_int m : forall (M : 'rV_m),'M_(m,size_int M) :=
  match m return forall (M : 'rV_m), 'M_(m,size_int M) with
    | S p => fun (M' : 'rV_(1 + p)) =>
              let m1 := lsubmx M' in
              let ms := rsubmx M' in
              let M0 := solve_int ms in
              let V := cap_wl m1 (−ms) in
```

```
            let W := cap_wr m1 (−ms) in
            block_mx (if m1 == 0 then delta_mx 0 0 else V) 0
                     (if m1 == 0 then 0 else W) M0
  | 0 => fun _ => 0
end.

Lemma solve_intP : forall m (M : 'rV_m) (X : 'cV_m),
  reflect (exists Y : 'cV[R]_(size_int M), X = solve_int M *m Y)
          (M *m X == 0).
```

Here cap_wl computes V and cap_wr computes W. Note that some special care has to be taken if m_1 is zero, if this is the case we output a matrix:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & M_0 \end{bmatrix}$$

However it would be desirable to output just

$$\begin{bmatrix} 1 & 0 \\ 0 & M_0 \end{bmatrix}$$

But this would not have the correct size. This could be solved by having a more complicated function that outputs a sum type with matrices of two different sizes. As this would give slightly more complicated proofs we decided to pad with zeroes instead. In section 3.6 we will discuss how to implement a more efficient algorithm, without any padding, that is more suitable for computation.

3.2 Strongly discrete rings

An important notion in constructive mathematics is the notion of *discrete* ring, that is, rings with decidable equality. Another important notion is *strongly discrete* rings, these are rings where membership in finitely generated ideals is decidable and if $x \in (a_1, \ldots, a_n)$ there is an algorithm computing $w_1, \ldots, w_n$ such that $x = \sum_i a_i w_i$.

Examples of such rings are multivariate polynomial rings over discrete fields (via Gröbner bases [5, 12]) and Bézout domains with explicit divisibility, that is, whenever $a \mid b$ one can compute x such that $b = xa$. We have represented strongly discrete rings in Coq as:

```
CoInductive member_spec (R : ringType) n (x : R) (I : 'rV[R]_n)
  : option 'cV[R]_n -> Type :=
| Member J of x%:M = I *m J : member_spec x I (Some J)
| NMember of (forall J, x%:M != I *m J) : member_spec x I None.

Record mixin_of (R : ringType) : Type := Mixin {
  member : forall n, R -> 'rV_n -> option 'cV_n;
  _ : forall n (x : R) (I : 'rV_n), member_spec x I (member x I)
}.
```

The structure of strongly discrete rings contains a function taking an element and a row vector (with the generators of the ideal) and return an option type with a column vector. This is Some J if x can be written as I *m J and if it is None then there should also be a proof that there cannot be any J satisfying x = I *m J. Note that the use of CoInductive has nothing

to do with coinduction but it should be seen as a datatype without any recursion schemes (as opposed to datatypes defined using Inductive) on which one can do case-analysis, for more information see [10].

3.3 Ideal theory

In the development we have chosen to represent finitely generated ideals as row vectors, so an ideal in R with n generators is represented as a row matrix of type 'rV[R]_n. This way operations on ideals can be implemented using functions on matrices and properties can be proved using the matrix library of SSReflect.

A nice property of strongly discrete rings is that the inclusion relation of finitely generated ideals is decidable. This means that we can decide if $I \subseteq J$ and if this is the case express every generator of I as a linear combination of the generators of J. We have implemented this as the function subid with correctness expressed as:

```
Notation "A <= B" := (subid A B).
Notation "A == B" := ((A <= B) && (B <= A)).

Lemma subidP : forall m n (I : 'rV[R]_m) (J : 'rV[R]_n),
  reflect (exists D, I = J *m D) (I <= J)%IS.
```

Note that this is expressed using matrix multiplication, so subidP says that if I <= J then every generator of I can be written as a linear combination of generators of J.

Ideal multiplication is an example where it is convenient to represent ideals as row vectors. As the product of two finitely generated ideals is generated by all products of generators of the ideals this can be expressed compactly using matrix operations:

```
Definition mulid m n (I : 'rV_m) (J : 'rV_n) : 'rV_(m * n) :=
  mxvec (I^T *m J).

Notation "I *i J" := (mulid I J).
```

Here mxvec flattens a matrix of type 'M[R]_(m,n) into a row vector of type 'rV[R]_(m * n) and I^T is the transpose of I. By representing ideals as row vectors we get compact definitions and quite simple proofs as the theory already developed about matrices can be used when proving properties of ideal operations.

It is also convenient to specify what the intersection of I and J is: it is an ideal K such that K <= I, K <= J and forall (x : R), member x I -> member x J -> member x K. So in order to prove that an integral domain is coherent it suffices to give an algorithm that computes K and prove that it satisfies these three properties. The cap_wr and cap_wl functions used in solve_with_int can then be implemented easily by explicitly computing D in subidP.

3.4 Coherent strongly discrete rings

If a ring R is both coherent and strongly discrete it is not only possible to solve homogeneous systems $MX = 0$ but also any system $MX = A$. The algorithm for computing this is expressed by induction on the number of equations where the case of one equation follow directly from the fact that the ring is strongly discrete. In the other case the matrix looks like:

$$\begin{bmatrix} R_1 \\ M \end{bmatrix} X = \begin{bmatrix} a_1 \\ A \end{bmatrix}$$

Now compute generators G_1 for the module of system of solutions of the homogeneous system $R_1X = 0$ and also test if $a_1 \in R_1$, if this is not the case the system is not solvable otherwise get W_1 such that $R_1W_1 = a_1$. Now compute by recursion the solution S of $MG_1X = A - MW_1$ such that $MG_1S = A - MW_1$. The solution to the system is then $W_1 + G_1S$ as

$$\begin{bmatrix} R_1 \\ M \end{bmatrix} (W_1 + G_1S) = \begin{bmatrix} R_1W_1 + R_1G_1S \\ MW_1 + MG_1S \end{bmatrix} = \begin{bmatrix} a_1 \\ A \end{bmatrix}$$

This algorithm has been implemented and proved correct as the function solve_general. Together with solveMxN this constitutes the only operations used as basis in the libraries of the HOMALG project [3].

3.4.1 Bézout domains are strongly discrete and coherent

The first example of coherent strongly discrete rings that we studied were Bézout domains with explicit divisibility. These are integral domains where every finitely generated ideal is principal (generated by a single element). The two main examples of Bézout domains are $\mathbb{Z}$ and $k[x]$ where k is a discrete field.

Bézout domains can also be characterized as rings with a GCD operation in which there is a function computing the elements of the Bézout identity. This means that given a and b one can compute x and y such that $xa + by$ is associate[3] to $gcd(a, b)$. Based on this it is straightforward to implement a function that given a finitely generated ideal $(a_1, \ldots, a_n)$ computes g such that $(a_1, \ldots, a_n) \subseteq (g)$ and $(g) \subseteq (a_1, \ldots, a_n)$ where this g is the greatest common divisor of all the a_i. To test if $x \in (a_1, \ldots, a_n)$ in a Bézout domain first compute a principal ideal (g) and then test if $g \mid x$ and if this is the case we we can construct the witness and otherwise we know that $x \notin (a_1, \ldots, a_n)$.

For showing that Bézout domains are coherent let I and J be two finitely generated ideals and compute principal ideals such that $I = (a)$ and $J = (b)$. Now it easy to prove that $I \cap J = (lcm(a, b))$, where $lcm(a, b)$ is the lowest common multiple of a and b which is computable in our setting as any Bézout ring is a GCD domain with explicit divisibility. Hence we have now proved that $\mathbb{Z}$ and $k[x]$ are both coherent and strongly discrete which means that we can solve arbitrary systems of equations over them.

3.5 Prüfer domains

Another class of rings that are coherent are *Prüfer domains*. These can be seen as non-Noetherian analogues of Dedekind domains and have many different characterizations, for instance does Bourbaki list fourteen of them [4]. The one we choose here is the one in [13]

[3] a and b are *associates* if $a \mid b$ and $b \mid a$ or equivalently that there exists a unit $u \in R$ such that $a = bu$.

that says that a Prüfer domain is an integral domain where given any x and y there exists u, v and w such that

$$ux = vy$$

and

$$(1 - u)y = wx$$

This is implemented in Coq by:

```
Record mixin_of (R : ringType) : Type := Mixin {
  prufer: R -> R -> (R * R * R)%type;
  _ : forall x y, let: (u,v,w) := prufer x y in
      u * x = v * y /\ (1 - u) * y = w * x
}.
```

As we require that Prüfer domains have explicit divisibility it is possible to prove that they are strongly discrete which in turn means that we can use the library of ideal theory developed for strongly discrete rings when proving that they are coherent. However it would be possible to prove that Prüfer domains are coherent without assuming explicit divisibility.

The most basic examples of Prüfer domains are Bézout domains (in particular $\mathbb{Z}$ and $k[x]$). However there are many other examples, for instance if R is a Bézout domain then the ring of elements integral over R is a Prüfer domain, this gives examples from algebraic geometry like $k[x, y]/(y^2 + x^4 - 1)$ and algebraic number theory like $\mathbb{Z}[\sqrt{-5}]$.

3.5.1 Principal localization matrices and strong discreteness

The key algorithm in the proof that Prüfer domains with explicit divisibility are both strongly discrete and coherent is an algorithm computing a *principal localization matrix* of an ideal [6]. This means that given a finitely generated ideal $(x_1, \ldots, x_n)$ one can compute a $n \times n$ matrix $M = (a_{ij})$ such that:

$$\sum_{i=1}^{n} a_{ii} = 1$$

and

$$\forall ijl.\ a_{lj}x_i = a_{li}x_j$$

Note that there is no constraint $n \neq 0$ which means that the first of these is a bit problematic as if $n = 0$ the sum will be empty and hence 0. To remedy this we express the property formally as:

```
Definition P1 n (M : 'M[R]_n) :=
  \big[+%R/0]_(i: 'I_n) (M i i) = (0 < n)%:R.

Definition P2 n (I : 'rV[R]_n) (M : 'M[R]_n) :=
  forall (i j l : 'I_n), (M l j) * (I 0 i) = (M l i) * (I 0 j).

Definition isPLM n (I : 'rV[R]_n) (M : 'M[R]_n) := P1 M /\ P2 I M.
```

The first statement uses an implicit coercion from booleans to rings where false is coerced to 0 and true to 1. The algorithm computing a principal localization matrix, plm, is quite involved so we have omitted it from this presentation, the interested reader should have a

look in the formal development and at the proofs in [6] and [13]. We have proved that this algorithm satisfies the above specification:

```
Lemma plmP : forall n (I : 'rV[R]_n), isPLM I (plm I).
```

The reason that principal localization matrices are interesting is that they give a way to compute the *inverse of a finitely generated ideal* I, this is a finitely generated ideal J such that IJ (with ideal multiplication) is principal. In fact if $I = (x_1, \ldots, x_n)$ and $M = (a_{ij})$ is its principal localization matrix then every column of M is an inverse to I. This means that we can define an algorithm for computing the inverse of ideals in Prüfer domains:

```
Definition inv_id n : 'I_n -> 'rV[R]_n -> 'rV[R]_n := match n with
  | S p => fun (i : 'I_(1 + p)%N) (I : 'rV[R]_(1 + p)%N) =>
    (col i (plm I))^T
  | _ => fun _ _ => 0
  end.

Lemma inv_idP n (I : 'rV[R]_n) i :
  (inv_id i I *i I == (I 0 i)%:M)%IS.
```

Using this it is possible to prove that Prüfer domains with explicit divisibility are strongly discrete. To compute if $x \in I$ first compute J such that $IJ = (a)$. Now $x \in I$ iff $(x) \subseteq I$ iff $xJ \subseteq (a)$. This can be decided if we can decide when an element is divisible by a.

We have used this to prove that our implementation of Prüfer domains is strongly discrete which means that the theory about ideals developed for strongly discrete rings can be used when proving that they are coherent.

3.5.2 Coherence

The key property of ideals in Prüfer domains for computing the intersection is that finitely generated ideals I and J satisfy:

$$(I + J)(I \cap J) = IJ$$

This means that we can devise an algorithm for computing generators for the intersection by first computing $(I + J)^{-1}$ such that $(I + J)^{-1}(I + J) = (a)$ and then get that

$$I \cap J = \frac{(I + J)^{-1} IJ}{a}$$

Note the use of division here. In fact it is possible to compute the intersection without assuming division but then the algorithm is more complicated. Using this the function for computing generators of the intersection is:

```
Definition pcap (n m : nat) (I : 'rV[R]_n) (J : 'rV[R]_m) :
  'rV[R]_(pcap_size I J).+1 := match find_nonzero (I +i J) with
  | Some i => let sIJ := I +i J in
              let a := sIJ 0 i in
              let acap := inv_id i sIJ *i I *i J in
              (0 : 'M_1) +i (\row_i (odflt 0 (acap 0 i %/? a)))
  | None => 0
  end.
```

Here %/? is the explicit divisibility function of R. The reason to add 0 as a generator of the ideal is simply to have the correct size as the formalized proof that R is coherent if $I \cap J$ is computable requires that $I \cap J$ is nonempty. Also note the function find_nonzero which finds the first nonzero element in a row-vector. This could have been implemented using the pick function for picking an element satisfying a decidable predicate which is provided for all SSReflect rings. But in order to simplify the translation to an efficient version of the algorithm we avoid using it here.

To prove that pcap really computes the intersection we need to first prove the main property used above for finding the algorithm computing $I \cap J$:

```
Lemma pcap_id (n m : nat) (I : 'rV[R]_n) (J : 'rV[R]_m) :
  ((I +i J) *i pcap I J == I *i J)%IS.
```

Using this it is possible to prove that pcap compute the intersection:

```
Lemma pcap_subidl m n (I: 'rV_m) (J: 'rV_n): (pcap I J <= I)%IS.

Lemma pcap_subidr m n (I: 'rV_m) (J: 'rV_n): (pcap I J <= J)%IS.

Lemma pcap_member m n x (I : 'rV[R]_m) (J : 'rV[R]_n) :
  member x I -> member x J -> member x (pcap I J).
```

Hence we have now proved that Prüfer domains with explicit divisibility are coherent strongly discrete rings so not only can we solve homogeneous systems over them but also any linear system of equations.

3.5.3 Examples of Prüfer domains

As mentioned before, any Bézout domain is a Prüfer domain. The proof of this is easy:

```
Definition bezout_calc (x y: R) : (R * R * R)%type :=
  let: (g,c,d,a,b) := egcdr x y in (d * b, a * d, b * c).

Lemma bezout_calcP (x y : R) : let: (u,v,w) := bezout_calc x y in
  u * x = v * y /\ (1 - u) * y = w * x.
```

Here egcdr is the extended Bézout algorithm where g is the gcd of x and y, $x = ag$, $y = bg$ and $ca + db = 1$.

We have not yet formalized the proof that $\mathbb{Z}[\sqrt{-5}]$ and $k[x, y]/(y^2 - 1 + x^4)$ are Prüfer domains but we have previously implemented this in Haskell [15].

3.6 Computations

All of the above algorithms and instances has been implemented on simple types as well in order to be able to do some computations with $\mathbb{Z}$ using the algorithms for Bézout domains. First we can compute the generators of $(2) \cap (3, 6)$:

```
Eval vm_compute in (cbcap 1 2 [::[::2]] [::[::3; 6]]).
    = [:: [:: 6]]
```

Next we can test if $6 \in (2)$:

```
Eval vm_compute in (cmember 1%N 6 [::[:: 2]]).
    = Some [:: [:: 3]]
```

It is also possible to solve the homogeneous system:

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
Eval vm_compute in (csolveMxN 2 2 [::[:: 1;2];[::2;4]]).
    = [:: [:: 2; 0];
          [:: −1; 0]]
```

and the inhomogeneous system:

$$\begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 4 \\ 8 \end{bmatrix}$$

```
Eval vm_compute in (csolveGeneral 2 2 [::[:: 2; 3]; [:: 4; 6]]
                                      [::[:: 4];[:: 8]]).
    = Some [:: [:: −4];
               [:: 4]]
```

The system $2x = 1$ does not have a solution in $\mathbb{Z}$:

```
Eval vm_compute in (csolveGeneral 1 1 [::[:: 2]] [::[::1]]).
    = None
```

We can also do some computations on the algorithms for Prüfer domains using $\mathbb{Z}$:

```
Eval vm_compute in (cplm 3 [::[:: 2; 3; 5]]).
    = [:: [:: 8; 12; 20];
          [:: 12; 18; 30];
         [:: −10; −15; −25]]

Eval vm_compute in (cinv_id 2 0 [:: [:: 2; 3]]).
    = [:: [:: −2; 2]]
```

The first computation computes the principal localization matrix of $(2, 3, 5)$ and the second compute the inverse of the ideal $(2, 3)$.

References

[1] J. Abdeljaoued and H. Lombardi. *Méthodes matricielles - Introduction à la complexité algébrique*. Springer, 2004.

[2] M. Barakat and M. Lange-Hegermann. An Axiomatic setup for algorithmic homological algebra and an alternative approach to localization. *J. Algebra Appl.*, 10(2):269–293, 2011.

[3] M. Barakat and D. Robertz. HOMALG – A Meta-Package for Homological Algebra. *J. Algebra Appl.*, 7(3):299–317, 2008.

[4] N. Bourbaki. *Commutative algebra. Chapters 1–7*. Elements of Mathematics. Springer-Verlag, 1998.

[5] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms: An introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 2006.

[6] L. Ducos, H. Lombardi, C. Quitté, and M. Salou. Théorie algorithmique des anneaux arithmétiques, des anneaux de Prüfer et des anneaux de Dedekind. *Journal of Algebra*, 281(2):604–650, 2004.

[7] M. Dénès, A. Mörtberg, and V. Siles. CoqEAL, the Coq Effective Algebra Library, 2012. http://www-sop.inria.fr/members/Maxime.Denes/coqeal.

[8] M. Dénès, A. Mörtberg, and V. Siles. A refinement based approach to computational algebra in Coq. In *Interactive Theorem Proving*, volume 7406 of *LNCS*, pages 83–98, 2012.

[9] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Proceedings 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *LNCS*, pages 327–342, 2009.

[10] G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010. RR-7392 RR-7392.

[11] D. E. Knuth. *The art of computer programming, volume 2: seminumerical algorithms*. Addison-Wesley, 1981.

[12] H. Lombardi and H. Perdry. The Buchberger Algorithm as a Tool for Ideal Theory of Polynomial Rings in Constructive Mathematics. 1998.

[13] H. Lombardi and C. Quitté. *Algèbre commutative, Méthodes constructives: Modules projectifs de type fini*. Calvage et Mounet, 2011.

[14] R. Mines, F. Richman, and W. Ruitenburg. *A Course in Constructive Algebra*. Springer-Verlag, 1988.

[15] A. Mörtberg. Constructive Algebra in Functional Programming and Type Theory. Master's thesis, Chalmers University of Technology, 2010.

[16] A. Mörtberg and V. Siles. Formalization of Bareiss/Sasaki-Murao algorithm, 2012. http://www.cse.chalmers.se/~siles/coq/formalisation.html.

[17] T. Sasaki and H. Murao. Efficient Gaussian Elimination Method for Symbolic Determinants and Linear Systems. *ACM Trans. Math. Softw.*, 8(3):277–289, Sept. 1982.

[18] C. D. Team. The Coq Proof Assistant Reference Manual, version 8.3. Technical report, 2010.

4 Appendix 1

The formal correctness of the Bareiss/Sasaki-Murao algorithm:

```
Lemma bareiss_recE : forall m a (M : 'M[{poly R}]_(1 + m)),
  a \is monic ->
 (forall p (h h' : p < 1 + m), pminor h h' M \is monic) ->
 (forall k (f g : 'I_k.+1 -> 'I_m.+1), rdvdp (a ^+ k) (minor f g M)) ->
  a ^+ m * (bareiss_rec a M) = \det M.
Proof.
elim=> [a M _ _ _|m ih a M am hpm hdvd] /=.
  by rewrite expr0 mul1r {2}[M]mx11_scalar det_scalar1.
have ak_monic k : a ^+ k \is monic by apply/monic_exp.
set d := M 0 0; set M' := _ - _; set M'' := map_mx _ _; simpl in M'.
have d_monic : d \is monic.
  have -> // : d = pminor (ltn0Sn _) (ltn0Sn _) M.
  have h : widen_ord (ltn0Sn m.+1) =1 (fun _ => 0)
    by move=> x; apply/ord_inj; rewrite [x]ord1.
  by rewrite /pminor (minor_eq h h) minor1.
have dk_monic : forall k, d ^+ k \is monic by move=> k; apply/monic_exp.
have hM' : M' = a *: M''.
  pose f := fun m (i : 'I_m) (x : 'I_2) => if x == 0 then 0 else (lift 0 i).
  apply/matrixP => i j.
  rewrite !mxE big_ord1 !rshift1 [a * _]mulrC rdivpK ?(eqP am,expr1n,mulr1) //.
  move: (hdvd 1%nat (f _ i) (f _ j)).
  by rewrite !minor2 /f /= expr1 !mxE !lshift0 !rshift1.
rewrite -[M]submxK; apply/(@lregX _ d m.+1 (monic_lreg d_monic)).
have -> : ulsubmx M = d%:M by apply/rowP=> i; rewrite !mxE ord1 lshift0.
rewrite key_lemma -/M' hM' detZ mulrCA [_ * (a ^+ _ * _)]mulrCA !exprS -!mulrA.
rewrite ih // => [p h h'|k f g].
  rewrite -(@monicMl _ (a ^+ p.+1)) // -detZ -submatrix_scale -hM'.
  rewrite -(monicMl _ d_monic) key_lemma_sub monicMr //.
  by rewrite (minor_eq (lift_pred_widen_ord h) (lift_pred_widen_ord h')) hpm.
case/rdvdpP: (hdvd _ (lift_pred f) (lift_pred g)) => // x hx.
apply/rdvdpP => //; exists x.
apply/(@lregX _ _ k.+1 (monic_lreg am))/(monic_lreg d_monic).
rewrite -detZ -submatrix_scale -hM' key_lemma_sub mulrA [x * _]mulrC mulrACA.
by rewrite -exprS [_ * x]mulrC -hx.
Qed.

Lemma bareissE n (M : 'M[{poly R}]_(1 + n))
  (H : forall p (h h' : p < 1 + n), pminor h h' M \is monic) :
  bareiss M = \det M.
Proof.
rewrite /bareiss -(@bareiss_recE n 1 M) ?monic1 ?expr1n ?mul1r //.
by move=> k f g; rewrite expr1n rdvd1p.
Qed.

Lemma bareiss_char_polyE n (M : 'M[R]_(1 + n)) :
  bareiss_char_poly M = char_poly M.
Proof.
rewrite /bareiss_char_poly bareissE // => p h h'.
exact: pminor_char_poly_mx_monic.
Qed.
```

```
Lemma bdetE n (M : 'M[R]_(1 + n)) : bdet M = \det M.
Proof.
rewrite /bdet bareiss_char_polyE char_poly_det -scaleN1r detZ mulrA -expr2.
by rewrite sqrr_sign mul1r.
Qed.
```

As one can see the formal proof is not much more than one page long. We find this quite remarkable as the paper proof is in fact a bit longer.