

A refinement-based approach to computational algebra in COQ

Maxime Dénès, Anders Mörtberg, Vincent Siles

▶ **To cite this version:**

Maxime Dénès, Anders Mörtberg, Vincent Siles. A refinement-based approach to computational algebra in COQ. ITP - 3rd International Conference on Interactive Theorem Proving - 2012, Aug 2012, Princeton, United States. pp.83-98, 10.1007/978-3-642-32347-8_7 . hal-00734505

HAL Id: hal-00734505

https://inria.hal.science/hal-00734505

Submitted on 22 Sep 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A refinement-based approach to computational algebra in Coq*

Maxime Dénès[1], Anders Mörtberg[2], and Vincent Siles[2]

[1]INRIA Sophia Antipolis - Méditerranée
[1]Maxime.Denes@inria.fr
[2]University of Gothenburg
[2]{mortberg,siles}@chalmers.se

September 22, 2012

Abstract

We describe a step-by-step approach to the implementation and formal verification of efficient algebraic algorithms. Formal specifications are expressed on rich data types which are suitable for deriving essential theoretical properties. These specifications are then refined to concrete implementations on more efficient data structures and linked to their abstract counterparts. We illustrate this methodology on key applications: matrix rank computation, Winograd's fast matrix product, Karatsuba's polynomial multiplication, and the gcd of multivariate polynomials.

1 Introduction

In the past decade, the range of application of proof assistants has extended its traditional ground in theoretical computer science to mainstream mathematics. Formalised proofs of important theorems like the fundamental theorem of algebra [2], the four colour theorem [6] and the Jordan curve theorem [10] have advertised the use of proof assistants in mathematical activity, even in cases when the pen and paper approach was no longer tractable.

But since these results established proofs of concept, more effort has been put into designing an actually scalable library of formalised mathematics. The *Mathematical Components* project (developing the SSReflect library [8] for the Coq proof assistant) advocates the use of small scale reflection to achieve a nearly comparable level of detail to usual mathematics on paper, even for advanced theories like the proof of the Feit-Thompson theorem. In this approach, the user expresses

*The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

significant deductive steps while low-level details are taken care of by small computational steps, at least when properties are decidable. Such an approach makes the proof style closer to usual mathematics.

One of the main features of these libraries is that they heavily rely on rich dependent types, which gives the opportunity to encode a lot of information directly into the type of objects: for instance, the type of matrices embeds their size, which makes operations like multiplication easy to implement. Also, algorithms on these objects are simple enough so that their correctness can easily be derived from the definition. However in practice, most efficient algorithms in modern computer algebra systems do not rely on dependent types and do not provide any proof of correctness. We show in this paper how to use this rich mathematical framework to develop efficient computer algebra programs *with proofs of correctness*. This is a step towards closing the gap between proof assistants and computer algebra systems.

The methodology we suggest for achieving this is the following: we are able to prove the correctness of some mathematical algorithms having all the high-level theory at our disposal and we then refine them to an implementation on simpler data structures that will be actually running on machines. In short, we aim at formally linking convenient high-level properties to efficient low-level implementations, ensuring safety of the whole approach while enjoying better performance thanks to the separation of proofs and computational content.

In the next section, we describe the methodology of refinements. Then, we give two examples of such refinements for matrices in Section 3, and polynomials in Section 4. In Section 5, we give a solution to unify both examples by describing CoqEAL[1], a library built using this methodology on top of the SSReflect libraries.

2 Refinements

Refinements are commonly used to describe successive steps when verifying a program. Typically, a specification is expressed in Hoare logic, then the program is described in a high-level language and finally implemented in C. Each step is proved correct with respect to the previous one. By using several formalisms, one has to trust every translation step or prove them correct in yet another formalism.

Our approach is similar: we refine the definition of a concept to an efficient algorithm described on high-level data structures. Then, we implement it on data structures that are closer to machine representations, once we no longer need rich theory to prove the correctness. Thus the implementation is an immediate translation of the algorithm, see Fig. 1.

However, in our approach, the three layers can be expressed in the same formalism (the Calculus of Inductive Constructions), though they do not use exactly the same features. On one hand, the high-level layers use rich dependent types that are very useful when describing theories because they allow abuse of notations

[1] Documentation available at `http://www-sop.inria.fr/members/Maxime.Denes/coqeal/`

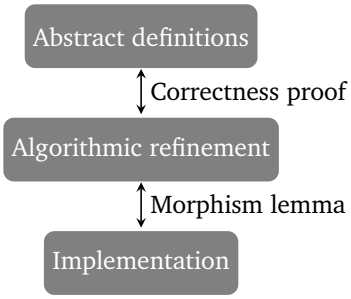


Figure 1: The three steps of refinement

and concise statements which quickly become necessary when working with advanced mathematics. On the other hand, the efficient implementations use simple types, which are closer to standard implementations in traditional programming languages. The main advantage of this approach is that the correctness of translations can easily be expressed in the formalism itself, and we do not rely on any additional external proofs.

In the next sections, we are going to use the following methodology to build efficient algorithms from high-level descriptions:

1. Implement an abstract version of the algorithm using SSReflect's structures and use the libraries to prove properties about them. Here we can use the full power of dependent types when proving correctness.

2. Refine this algorithm into an efficient one using SSReflect's structures and prove that it behaves like the abstract version.

3. Translate the SSReflect structures and the efficient algorithm to the low-level data types, ensuring that they will perform the same operations as their high-level counterparts.

3 Matrices

Linear algebra is a natural first test-case to validate our approach, as a pervasive and inherently computational area of mathematics, which is well covered by the SSReflect library [7]. In this section, we will detail the (quite simple) data structure we use to represent matrices and then review two fundamental examples: rank computation and efficient matrix product.

3.1 Representation

Matrices are represented by finite functions over pairs of ordinals (the indices):

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

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

This encoding makes many properties easy to derive, but it is inefficient for evaluation. Indeed, a finite function over `'I_m * 'I_n` is internally represented as a flat list of $m \times n$ values which has to be traversed whenever the function is evaluated. Moreover, having the size of matrices encoded in their type allows to state concise lemmas without explicit side conditions, but it is not always flexible enough when getting closer to machine-level implementation details.

To be able to implement efficient matrix operations we introduce a low-level data type `seqmatrix` representing matrices as lists of lists. A concrete matrix is built from an abstract one by mapping canonical enumerations (`enum`) of ordinals to the corresponding coefficients in the abstract matrix:

```
Definition seqmx_of_mx (M : 'M[R]_(m,n)) : seqmatrix :=
  [seq [seq M i j | j <- enum 'I_n] | i <- enum 'I_m].
```

To ensure the correct behaviour of concrete matrices it is sufficient to prove that `seqmx_of_mx` is injective (`==` denotes boolean equality):

```
Lemma seqmx_eqP (M N : 'M[R]_(m,n)) :
  reflect (M = N) (seqmx_of_mx M == seqmx_of_mx N).
```

Operations like addition are straightforward to implement, and their correctness is expressed through a morphism lemma, stating that the concrete representation of the sum of two matrices is the concrete sum of their concrete representations:

```
Definition addseqmx (M N : seqmatrix) : seqmatrix :=
  zipwith (zipwith (fun x y => add x y)) M N.

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

Here `morph` is notation meaning that `seqmx_of_mx` is an additive morphism from abstract to concrete matrices. It is worth noting that we could have stated all our morphism lemmas with the converse operator (from concrete matrices to abstract ones). But these lemmas would then have been quantified over lists of lists, with poorer types, which would have required a well-formedness predicate as well as premises expressing size constraints. The way we have chosen takes full advantage of the information carried by richer types.

Like the `addseqmx` operation, we have developed concrete implementations of most of the matrix operations provided by the SSReflect library and proved the corresponding morphism lemmas. Among these operations we can cite: subtraction, scaling, transpose and block operations.

3.2 Computing the rank

Now that the basic data structure and operations have been defined, it is possible to apply our approach to an algorithm based on Gaussian elimination which computes the rank of a matrix $A = (a_{i,j})$ over a field K. We first specify the algorithm using abstract matrices and then refine it to the low-level structures.

An elimination step consists of finding a nonzero pivot in the first column of A. If there is none, it is possible to drop the first column without changing the rank. Otherwise, there is an index i such that $a_{i,1} \neq 0$. By linear combinations of rows (preserving the rank) A can be transformed into the following matrix B:

$$B = \begin{bmatrix} 0 & a_{1,2} - \frac{a_{1,1}\times a_{i,2}}{a_{i,1}} & \cdots & a_{1,n} - \frac{a_{1,1}\times a_{i,n}}{a_{i,1}} \\ 0 & \vdots & & \vdots \\ a_{i,1} & a_{i,2} & \cdots & a_{i,n} \\ 0 & \vdots & & \vdots \\ 0 & a_{n,2} - \frac{a_{n,1}\times a_{i,2}}{a_{i,1}} & \cdots & a_{n,n} - \frac{a_{n,1}\times a_{i,n}}{a_{i,1}} \end{bmatrix} = \begin{bmatrix} 0 & & \\ \vdots & R_1 & \\ 0 & & \\ a_{i,1} & \cdots & a_{i,n} \\ 0 & & \\ \vdots & R_2 & \\ 0 & & \end{bmatrix}$$

Now pose $R = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$, since $a_{i,1} \neq 0$, this means that $rank\ A = rank\ B = 1 + rank\ R$. Hence the current rank can be incremented and the algorithm can be recursively applied on R.

In our development we defined a function `elim_step` returning the matrix R above and a boolean b indicating if a pivot has been found. A wrapper function `rank_elim` is in charge of maintaining the current rank and performing the recursive call on R:

```
Fixpoint rank_elim (m n : nat) {struct n} : 'M[K]_(m,n) -> nat :=
  match n return 'M[K]_(m,n) -> nat with
   | q.+1 => fun M =>
     let (R,b) := elim_step M in (rank_elim R + b)%N
   | _ => fun _ => 0%N
  end.
```

Note that booleans are coerced to natural numbers: b is interpreted as 1 if true and 0 if false. The correctness of `rank_elim` is expressed by relating it to the `\rank` function of the SSReflect library:

```
Lemma rank_elimP n m (M : 'M[K]_(m,n)) : rank_elim M = \rank M.
```

The proof of this specification relies on a key invariant of `elim_step`, relating the ranks of the input and output matrices:

```
Lemma elim_step_rank m n (M : 'M[K]_(m, 1 + n)) :
  let (R,b) := elim_step M in \rank M = (\rank R + b)%N.
```

Now the proof of `rank_elimP` follows by induction on n. The concrete version of this algorithm is a direct translation of the algorithm using only concrete matrices

and executable operations on them. This executable version (called `rank_elim_seqmx`) is then linked to the abstract implementation by the lemma:

```
Lemma rank_elim_seqmxE : forall m n (M : 'M[K]_(m, n)),
  rank_elim_seqmx m n (seqmx_of_mx M) = rank_elim M.
```

The proof of this is straightforward as all of the operations on concrete matrices have morphism lemmas which means that the proof can be done simply by expanding the definitions and applying the translation morphisms.

3.3 Fast matrix product

In the context we presented, the naïve matrix product (i.e. with cubic complexity) of two matrices M and N can be implemented by transposing the list of lists representing N and then for each i and j compute $\sum_k M_{i,k} N^{\mathrm{T}}_{j,k}$:

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

Lemma mulseqmxE (M : 'M[R]_(m,p)) (N : 'M[R]_(p,n)) :
  mulseqmx (seqmx_of_mx M) (seqmx_of_mx N) = seqmx_of_mx (M *m N).
```

`*m` is SSReflect's notation for the matrix product. Once again, the rich type information in the quantification of the morphism lemma ensures that it can be applied only if the two matrices have compatible sizes.

In 1969, Strassen [19] showed that 2×2 matrices can be multiplied using only 7 multiplications without requiring commutativity. This yields an immediate recursive scheme for the product of two $n \times n$ matrices with $\mathcal{O}(n^{\log_2 7})$ complexity.[2] This is an important theoretical result, since matrix multiplication was commonly thought to be intrinsically of cubic complexity, it opened the way to many further improvements and gave birth to a fertile branch of algebraic complexity theory.

However, Strassen's result is also still of practical interest since the asymptotically best algorithms known today [4] are slower in practice because of huge hidden constants. Thus, we implemented a variant of this algorithm suggested by Winograd in 1971 [20], decreasing the required number of additions and subtractions to 15 (instead of 18 in Strassen's original proposal). This choice reflects the implementation of matrix product in most of modern computer algebra systems. A previous formal description of this algorithm has been developed in ACL2 [17], but it is restricted to matrices whose sizes are powers of 2. The extension to arbitrary matrices represents a significant part of our development, which is to the best of our knowledge the first complete formally verified description of Winograd's algorithm.

We define a function expressing a recursion step in Winograd's algorithm. Given two matrices A and B and an operator f representing matrix product, it reformulates the algebraic identities involved in the description of the algorithm:

[2] $\log_2 7$ is approximately 2.807

```
Definition winograd_step {p : positive} (A B : 'M[R]_(p + p)) f :=
  let A11 := ulsubmx A in let A12 := ursubmx A in
  let A21 := dlsubmx A in let A22 := drsubmx A in
  let B11 := ulsubmx B in let B12 := ursubmx B in
  let B21 := dlsubmx B in let B22 := drsubmx B in
  let X := A11 - A21 in let Y := B22 - B12 in
  let C21 := f X Y in
  let X := A21 + A22 in let Y := B12 - B11 in
  let C22 := f X Y in
  let X := X - A11 in let Y := B22 - Y in
  let C12 := f X Y in
  let X := A12 - X in
  let C11 := f X B22 in
  let X := f A11 B11 in
  let C12 := X + C12 in let C21 := C12 + C21 in
  let C12 := C12 + C22 in let C22 := C21 + C22 in
  let C12 := C12 + C11 in
  let Y := Y - B21 in
  let C11 := f A22 Y in let C21 := C21 - C11 in
  let C11 := f A12 B21 in let C11 := X + C11 in
  block_mx C11 C12 C21 C22.
```

This is an implementation of matrix multiplication that is clearly not suited for proving algebraic properties, like associativity. The correctness of this function is expressed by the fact that if f is instantiated by the multiplication of matrices, `winograd_step A B` should be the product of `A` and `B` (`=2` denotes extensional equality):

```
Lemma winograd_stepP (p : positive) (A B : 'M[R]_(p + p)) f :
  f =2 mulmx -> winograd_step A B f = A *m B.
```

This proof is made easy by the use of the `ring` tactic (the script is two lines long). Since version 8.4 of Coq, `ring` is applicable to non-commutative rings, which has allowed its use in our context.

Note that the above implementation only works for even-sized matrices. This means that the general procedure has to implement a strategy for handling odd-sized matrices. Several standard techniques have been proposed, which fall into two categories. Some are static, in the sense that they preprocess the matrices to obtain sizes that are powers of 2. Others are dynamic, meaning that parity is tested at each recursive step. Two standard treatments can be implemented either statically or dynamically: padding and peeling. The first consists of adding rows and/or columns of zeros as required to get even dimensions (or a power of 2), these lines are then simply removed from the result. Peeling on the other hand removes rows or columns when needed, and corrects the result accordingly.

We chose to implement dynamic peeling because it seemed to be the most challenging technique from the formalisation point of view, since the size of matrices involved depend on dynamic information and the post processing of the result is

more sophisticated than using padding. Another motivation is that dynamic peeling has shown to give good results in practice.

The function that implements Winograd multiplication with dynamic peeling is called `winograd` and it is proved correct with respect to the usual matrix product:

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

The concrete version is called `winograd_seqmx` and it is also just a direct translation of `winograd` using only concrete operations on `seq` based matrices. In the next section, Fig. 2 shows some benchmarks of how well this implementation performs compared to the naïve matrix product, but we will first discuss how to implement concrete algorithms based on dependently typed polynomials.

4 Polynomials

Polynomials in the SSReflect library are represented as records with a list representing the coefficients and a proof that the last of these is nonzero. The library also contains basic operations on this representation like addition and multiplication and proofs that the polynomials form a commutative ring using these operations. The implementation of these operations use big operators [3] which means that it is not possible to compute with them.

To remedy this we have implemented polynomials as lists without any proofs together with executable implementations of the basic operations. It is very easy to build a concrete polynomial from an abstract polynomial, simply apply the record projection (called `polyseq`) to extract the list from the record. The soundness of concrete polynomials is proved by showing that the pointwise boolean equality on the projected lists reflects the equality on abstract polynomials:

```
Lemma polyseqP p q : reflect (p = q) (polyseq p == polyseq q).
```

Basic operations like addition and multiplication are slightly more complicated to implement for concrete polynomials than for concrete matrices as it is necessary to ensure that these operations preserve the invariant that the last element is nonzero. For instance multiplication is implemented as:

```
Fixpoint mul_seq p q := match p,q with
  | [::], _ => [::]
  | _, [::] => [::]
  | x :: xs,_ => add_seq (scale_seq x q) (mul_seq xs (0%R :: q))
  end.

Lemma mul_seqE : {morph polyseq : p q / p * q >-> mul_seq p q}.
```

Here `add_seq` is addition of concrete polynomials and `scale_seq x q` means that every coefficient of `q` is multiplied by `x` (both of these are implemented in such a way that the invariant that the last element is nonzero is satisfied). Using this approach we have implemented a substantial part of the SSReflect polynomial library, including pseudo-division, using executable polynomials.

4.1 Fast polynomial multiplication

The naïve polynomial multiplication algorithm presented in the previous section requires $\mathcal{O}(n^2)$ operations. A more efficient algorithm is Karatsuba's algorithm [1, 11] which is a divide and conquer algorithm based on reducing the number of recursive calls in the multiplication. More precisely, in order to multiply two polynomials written as $aX^k + b$ and $cX^k + d$ the ordinary method

$$(aX^k + b)(cX^k + d) = acX^{2k} + (ad + bc)X^k + cd$$

requires four multiplications (as the multiplications by X^n can be implemented efficiently by padding the list of coefficients by n zeroes). The key observation is that this can be rewritten as

$$(aX^k + b)(cX^k + d) = acX^{2k} + ((a + b)(c + d) - ac - bd)X^k + bd$$

which only requires three multiplication: ac, $(a + b)(c + d)$ and bd. Now if the two polynomials have 2^n coefficients and the splitting is performed in the middle at every point then the algorithm will only require $\mathcal{O}(n^{\log_2 3})$ which is better than the naïve algorithm.[3] If the polynomials do not have 2^n coefficients it is possible to split the polynomials at for example $\lfloor n/2 \rfloor$ as the formula above holds for any $k \in \mathbb{N}$ and still obtain a faster algorithm. This algorithm has been implemented in Coq previously for binary natural numbers [15] and for numbers represented by a tree-like structure [9]. But as far as we know, it has never been implemented for polynomials before. When implementing this algorithm we first implemented it using dependently typed polynomials as:

```
Fixpoint karatsuba_rec (n : nat) p q := match n with
  | 0%N => p * q
  | n'.+1 => if (size p <= 2) || (size q <= 2) then p * q else
     let m := minn (size p)./2 (size q)./2 in
     let (p1,p2) := splitp m p in
     let (q1,q2) := splitp m q in
     let p1q1 := karatsuba_rec n' p1 q1 in
     let p2q2 := karatsuba_rec n' p2 q2 in
     let p12 := p1 + p2 in
     let q12 := q1 + q2 in
     let p12q12 := karatsuba_rec n' p12 q12 in
     p1q1 * 'X^(2 * m) + (p12q12 - p1q1 - p2q2) * 'X^m + p2q2
  end.
```

Here `splitp` is a function that splits the polynomial at the correct point using `take` and `drop`. There is also a wrapper function named `karatsuba` that calls `karatsuba_seq` with the greatest degree of `p` and `q`. The correctness of this algorithm is expressed by:

```
Lemma karatsubaE : forall p q, karatsuba p q = p * q.
```

[3] $\log_2 3$ is approximately 1.585.

As p and q are SSReflect polynomials this lemma can be proved using all of the theory in the library. The next step is to implement the executable version (karatsuba_seq) of this algorithm which is done by changing all the operations in the above version to executable operations on concrete polynomials. The correctness of the concrete algorithm is then proved by:

```
Lemma karatsuba_seqE :
  {morph polyseq : p q / karatsuba p q >-> karatsuba_seq p q}.
```

The proof of this is straightforward as all of the operations have morphism lemmas for translating back and forth between the concrete representation and the high-level ones.

In Fig. 2 the running time of the different multiplication algorithms that we have implemented is compared:

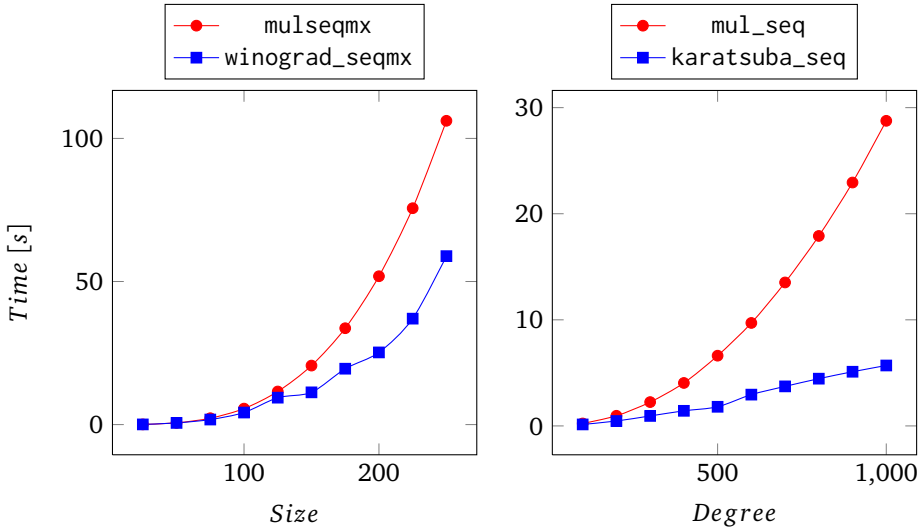


Figure 2: Benchmarks of Winograd and Karatsuba multiplication

The benchmarks have been done by computing the square of integer matrices and polynomials using the Coq virtual machine (i.e. by running vm_compute). It is clear that both the implementation of Winograd matrix multiplication and Karatsuba polynomial multiplication is faster than their naïve counterparts, as expected.

4.2 gcd of multivariate polynomials

An important feature of modern computer algebra systems is to compute the greatest common divisor (gcd) of multivariate polynomials. The main idea of our implementation is based on the observation that in order to compute the gcd of elements in $R[X_1,\ldots,X_n]$ it suffices to show how to compute the gcd in $R[X]$ given that it is

possible to compute the gcd of elements in R. So, for example, to compute the gcd of elements in $\mathbb{Z}[X,Y]$ we model it as $(\mathbb{Z}[X])[Y]$, i.e. as univariate polynomials in Y with coefficients in $\mathbb{Z}[X]$, and then use that there is a gcd algorithm in $\mathbb{Z}$.

The algorithm that we implemented is based on the presentation of Knuth in [12] which uses that in order to compute the gcd of two multivariate polynomials it is possible to instead consider the task of computing the gcd of *primitive* polynomials, i.e. polynomials where all coefficients are coprime. Using that any polynomial can be split in a primitive part and a non-primitive part by dividing by the gcd of its coefficients (this is called the *content* of the polynomial) we get an algorithm for computing the gcd of any two polynomials. Below is our implementation of this algorithm together with explanations of the operations:

```
Fixpoint gcdp_rec (n : nat) (p q : {poly R}) :=
  let r := modp p q in
  if r == 0 then q
            else if n is m.+1 then gcdp_rec m q (pp r) else pp r.

Definition gcdp p q :=
  let (p1,q1) := if size p < size q then (q,p) else (p,q) in
  let d := (gcdr (gcdsr p1) (gcdsr q1))%:P in
  d * gcdp_rec (size (pp p1)) (pp p1) (pp q1).
```

- `modp p q` computes the remainder after pseudo-dividing `p` by `q`.
- `pp p` computes the primitive part of `p` by dividing it by its content.
- `gcdsr p` computes the content of `p`.
- `gcdr (gcdsr p1)(gcdsr q1)` computes the gcd (using the operation in the underlying ring) of the content of `p1` and the content of `q1`.

The correctness of this algorithm is now expressed by:

```
Lemma gcdpP : forall p q g, g %| gcdp p q = (g %| p) && (g %| q).
```

Here `p %| q` computes whether `p` divides `q` or not. As divisibility is reflexive this equality is a compact way of expressing that the function actually computes the gcd of `p` and `q`.

Our result is stated in constructive algebra [14] as: If R is a gcd domain then so is $R[X]$. Our algorithmic proof is different (and arguably simpler) than the one in [14]; for instance, we do not go via the field of fractions of the ring.

As noted in [12], this algorithm may be inefficient when applied on the polynomials over integers. The reference [12] provides a solution in this case, based on subresultants. This would be a further refinement of the algorithm, which would be interesting to explore since subresultants have been already analysed in Coq [13].

The executable version (`gcdp_seq`) of the algorithm has also been implemented and is linked to the abstract version above by:

```
Lemma gcdp_seqE :
  {morph polyseq : p q / gcdp p q >-> gcdp_seq p q}.
```

But when running the concrete implementation there is a quite subtle problem: the `polyseq` projection links the abstract polynomials with the concrete polynomials of type `seq R` where `R` is a *ring* with a gcd operation. Let us consider multivariate polynomials, for example $R[x,y]$. In this case the concrete type will be `seq (seq R)`, but `seq R` is not a ring so our algorithm is not applicable! The next section explains how to resolve this issue so that it is possible to implement computable algorithms of the above kind that rely on the computability of the underlying ring.

5 Algebraic hierarchy of computable structures

As noted in the previous section there is a problem when implementing multivariate polynomials by iterating the polynomial construction, i.e. by representing $R[X,Y]$ as $(R[X])[Y]$. The same problem occurs when considering other structures where the computation relies on the computability of the underlying ring as is the case when computing the characteristic polynomial of a square matrix for instance. For this, one needs to compute with matrices of polynomials which will require a concrete implementation of matrices with coefficients being a concrete implementation of polynomials.

However, both the list based matrices and polynomials have something in common: we can guarantee the correctness of the operations on a subset of the low-level structure. This can be used to implement another hierarchy of computable structures corresponding to the SSReflect algebraic hierarchy.

5.1 Design of the library

We have implemented computable counterparts to the basic structures in this hierarchy, e.g. $\mathbb{Z}$-modules, rings and fields. These are implemented in the same manner as presented in [5] using canonical structures. Here are a few examples of the mixins we use:

```
Record trans_struct (A B: Type) : Type := Trans {
  trans : A -> B;
  _ : injective trans
}.

(* Mixin for "Computable" Z-modules *)
Record mixin_of (V : zmodType) (T: Type) : Type := Mixin {
  zero : T;
  opp : T -> T;
  add : T -> T -> T;
  tstruct : trans_struct V T;
  _ : (trans tstruct) 0 = zero;
  _ : {morph (trans tstruct) : x / - x >-> opp x};
```

```
  _ : {morph (trans tstruct) : x y / x + y >-> add x y}
}.

(* Mixin for "Computable" Rings *)
Record mixin_of (R : ringType) (V : czmodType R) : Type := Mixin {
  one : V;
  mul : V -> V -> V;
  _ : (trans V) 1 = one;
  _ : {morph (trans V) : x y / x * y >-> mul x y}
}.
```

The type `czmodType` is the computable $\mathbb{Z}$-module type parametrized by a $\mathbb{Z}$-module. The `trans` function is the translation function from SSReflect structures to the computable structures and the only property that is required of it is that it is injective, so we are sure that different high-level objects are mapped to different computable objects.

This way we can implement all the basic operations of the algebraic structures the way we want (for example using fast matrix multiplication as an implementation of `*m` instead of a naïve one), and the only thing we have to prove is that the implementations behave the same as SSReflect's operations *on the subset of "well-formed terms"* (e.g. for polynomials, lists that do not end with 0). This is done by providing the corresponding morphism lemmas.

The operations presented in the previous sections can then be implemented by having computable structures as the underlying structure instead of dependently typed ones. This way one can prove that polynomials represented as lists is a computable ring by assuming that the coefficients are computable and hence get ring operations that can be applied on multivariate polynomials built by iterating the construction.

It is interesting to note that the equational behavior of an abstract structure is carried as a parameter, but does not appear in its computable counterpart, which depends only on the operations to be implemented. For instance, the same computable ring structure can implement a commutative ring or an arbitrary one, only its parameter varies.

5.2 Example: computable ring of polynomials

Let us explain how the list based polynomials can be made a computable ring. First, we define:

```
Variable R : comRingType.
Variable CR : cringType R.
```

This says that `CR` is a computable ring parametrized by a commutative ring which makes sense as any commutative ring is a ring. Next we need to implement the translation function from `{poly R}` to `seq CR` and prove that this translation is injective:

```
Definition trans_poly (p : {poly R}) : seq CR :=
```

```
  map (@trans R CR) (polyseq p).

Lemma inj_trans_poly : injective trans_poly.
```

Assuming that computable polynomials already are an instance of the computable $\mathbb{Z}$-module structure it is possible to prove that they are computable rings by implementing multiplication (exactly like above) and then prove the corresponding morphism lemmas:

```
Lemma trans_poly1 : trans_poly 1 = [:: (one CR)].

Lemma mul_seqE :
  {morph trans_poly : p q / p * q >-> mul_seq p q}.
```

At this point, we could also have used the `karatsuba_seq` implementation of polynomial multiplication instead of `mul_seq` since we can prove its correctness using the `karatsubaE` and `karatsuba_seqE` lemmas. Finally this can be used to build the `CRing` mixin and make it a canonical structure.

```
Definition seq_cringMixin := CRingMixin trans_poly1 mul_seqE.

Canonical Structure seq_cringType :=
  Eval hnf in CRingType {poly R} seq_cringMixin.
```

5.3 Examples of computations

This computable ring structure has also been instantiated by the Coq implementation of $\mathbb{Z}$ and $\mathbb{Q}$ which means that they can be used as basis when building multivariate polynomials. To multiply $2+xy$ and $1+x+xy+x^2y^2$ in $\mathbb{Z}[x,y]$ one can write:

```
Definition p := [:: [:: 2]; [:: 0; 1]].
Definition q := [:: [:: 1; 1]; [:: 0; 1]; [:: 0; 0; 1]].

> Eval compute in mul p q.
  = [:: [:: 2; 2]; [:: 0; 3; 1]; [:: 0; 0; 3]; [:: 0; 0; 0; 1]]
```

The result should be interpreted as $(2+2x)+(3x+x^2)y+3x^2y^2+x^3y^3$. The gcd of $1+x+(x+x^2)y$ and $1+(1+x)y+xy^2$ in $\mathbb{Z}[x,y]$ can be computed by:

```
Definition p := [:: [:: 1; 1] ; [:: 0; 1; 1] ].
Definition q := [:: [:: 1]; [:: 1; 1]; [:: 0; 1]].

> Eval compute in gcdp_seq p q.
  = [:: [:: 1]; [:: 0; 1]]
```

The result is $1+xy$ as expected. The following is an example over $\mathbb{Q}[x,y]$:

```
Definition p := [:: [:: 2 # 3; 2 # 3]; [:: 0; 1 # 2; 1 # 2]].
Definition q := [:: [:: 2 # 3]; [:: 2 # 3; 1 # 2]; [:: 0; 1 # 2]].
```

```
> Eval compute in gcdp_seq p q.
  = [:: [:: 1 # 3]; [:: 0; 1 # 4]]
```

The two polynomials are $\frac{2}{3}+\frac{2}{3}x+\frac{1}{2}xy+\frac{1}{2}x^2y$ and $\frac{2}{3}+\frac{2}{3}y+\frac{1}{2}xy+\frac{1}{2}xy^2$. The resulting gcd should be interpreted as $\frac{1}{3}+\frac{1}{4}xy$.

6 Conclusions and Further Work

In this paper, we showed how to use high-level libraries to prove properties of algorithms, while retaining good execution capabilities by providing efficient low-level implementations. The need of modularity of the executable structure appears naturally and the methodology explained in [5] works quite well. The only thing a user has to provide is a proof of an injectivity lemma stating that the translation behaves correctly.

The methodology we suggest has already been used in other contexts, like the CoRN library, where properties of real numbers described in [16] are obtained by proving that these real numbers are isomorphic to an abstract, pre-existing but less efficient version. We tried to show that this approach can be applied in a systematic and modular way.

The library we designed also helps to solve a restriction of SSReflect: due to a lot of computations during deduction steps, some of the structures are *locked* to allow type-checking to be performed in a reasonable amount of time. This locking prevents full-scale reflection on some of the most complex types like big operators, polynomials or matrices. Our implementation restores the ability to perform full-scale reflection on abstract structures, and more generally to compute. For instance, addition of two fully instantiated polynomials cannot be evaluated to its actual numerical result but we can refine it to a computable object that will reduce. This is a first step towards having in the same system definitions of objects on which properties can be proved and some of the usual features of a computer algebra system.

However, in its current state, the inner structure of our library is slightly more rigid than necessary: we create a type for computable $\mathbb{Z}$-modules, but in practice, all the operations it contains could be packaged independently. Indeed, on each of these operations we prove only a morphism lemma linking it to its abstract counterpart, whereas in usual algebraic structures, expressing properties like distributivity require access to several operations at once. This specificity would make it possible to reorganise the library and create independent structures for each operation, instead of creating one of them for each type. Also, we could use other packaging methods, like type classes [18], to simplify the layout of the library. However, modifying the library to use type classes on top of SSReflect's canonical structures is still on-going work, since we faced some incompatibilities between the different instance resolution mechanisms.

References

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

[2] H. Barendregt, H. Geuvers, R. Pollack, F. Wiedijk, and J. Zwanenburg. The "fundamental theorem of algebra" project. `http://www.cs.ru.nl/~freek/fta/`.

[3] Y. Bertot, G. Gonthier, S. Biha, and I. Pasca. Canonical big operators. In *Theorem Proving in Higher-Order Logics (TPHOLs'08)*, volume 5170 of *LNCS*, pages 86–101, 2008.

[4] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, Mar. 1990.

[5] 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.

[6] G. Gonthier. Formal proof—the four-color theorem. In *Notices of the American Mathematical Society*, volume 55, pages 1382–1393, 2008.

[7] G. Gonthier. Point-Free, Set-Free concrete linear algebra. In *Interactive Theorem Proving*, volume 6898 of *LNCS*, pages 103–118, 2011.

[8] G. Gonthier and A. Mahboubi. A Small Scale Reflection Extension for the Coq system. Technical report, Microsoft Research INRIA, 2009. `http://hal.inria.fr/inria-00258384`.

[9] B. Grégoire and L. Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In U. Furbach and N. Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 423–437. Springer, 2006.

[10] T. C. Hales. The jordan curve theorem, formally and informally. In *The American Mathematical Monthly*, volume 114, pages 882–894, 2007.

[11] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. In *USSR Academy of Sciences*, volume 145, pages 293–294, 1962.

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

[13] A. Mahboubi. Proving formally the implementation of an efficient gcd algorithm for polynomials. In *3rd International Joint Conference on Automated Reasoning (IJCAR), LNAI*, pages 438–452. Springer-Verlag, 2006.

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

[15] R. O'Connor. Karatsuba's multiplication. `http://coq.inria.fr/V8.2pl1/contribs/Karatsuba.html`.

[16] R. O'Connor. Certified exact transcendental real number computation in coq. In *Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *LNCS*, pages 246–261. Springer, 2008.

[17] F. Palomo-Lozano, I. Medina-Bulo, and J. Alonso-Jiménez. Certification of matrix multiplication algorithms. strassen's algorithm in ACL2. In *Supplemental Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics*, volume Informatics Research Report EDI-INF-RR-0046, Edinburgh, 2001.

[18] M. Sozeau and N. Oury. First-Class type classes. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, pages 278–293, 2008.

[19] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, Aug. 1969.

[20] S. Winograd. On multiplication of 2x2 matrices. *Linear Algebra and its Applications*, 4:381 – 388, 1971.