Formalizing Refinements and Constructive Algebra in Type Theory

Anders Mörtberg

December 12, 2014

This thesis

Formally verify the correctness of the implementation of algorithms from computer algebra using intuitionistic type theory

Decrease the gap between algorithms in computer algebra and proof assistants, increase the reliability of algorithms in computer algebra and the computational capabilities of proof assistants

This thesis

Formalization in Coq/SSReflect of:

- Program refinements
- Data refinements
- Constructive algebra

Refinements

Refinements

- Program refinements: Transform a program into a more efficient one computing the same thing using a different algorithm, while preserving the types.
- Data refinements: Change the data representation on which the program operates into a more efficient one, while preserving the involved algorithms.

Program refinement: Sasaki-Murao algorithm

- Simple polynomial time algorithm that generalizes Bareiss' algorithm for computing the determinant over any commutative ring (not necessarily with division)
- Standard presentations have quite complicated correctness proofs, relying on Sylvester determinant identities
- We wrote a short and simple program using functional programming notations that we proved correct

Program refinement: Bareiss' algorithm

```
data Matrix a = Empty | Cons a [a] [a] (Matrix a)

dvd_step :: DvdRing a => a -> Matrix a -> Matrix a
dvd_step g M = mapM (\x -> g | x) M

bareiss_rec :: DvdRing a => a -> Matrix a -> a
bareiss_rec g M = case M of
  Empty -> g
  Cons a l c M ->
     let M' = a * M - c * l in
     bareiss_rec a (dvd_step g M')

bareiss :: DvdRing a => Matrix a -> a
bareiss M = bareiss_rec 1 M
```

Program refinement: Sasaki-Murao algorithm

- Problem with Bareiss: Division by 0
- Solution: Sasaki-Murao algorithm:
  - Apply the algorithm to $M - xI$
  - Compute on $R[x]$ with pseudo-division instead of division on R
  - Put $x = 0$ in the result
- Benefits:
  - More general
  - No problem of division by 0 (we have x along the diagonal)
  - Get characteristic polynomial for free
  - Algorithm is the same as Bareiss'
- Correctness proved with respect to standard definition:

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{\sigma(i),i}$$

Data refinement: unary and binary integers

Proof-oriented (unary) integers: `int`

Computation-oriented (binary) integers: `Z`

These two types are isomorphic, in general we consider any *related* types

"Types, abstraction, and parametric polymorphism" – Reynolds 1983

Data refinement: Polynomials

Proof oriented definition:

```
Record poly R := Poly {
  polyseq : seq R;
  _ : last 1 polyseq != 0
}.

Definition mul_poly (p q : poly R) : poly R :=
  \poly_(i < (size p + size q).-1)
    (\sum_(j < i.+1) p`_j * q`_(i - j)).
```

Data refinement: Sparse polynomials

Want to refine to computation-oriented implementation, for instance sparse Horner normal form:

```
Inductive sparse R :=
    Pc : R -> sparse R
  | PX : R -> pos -> sparse R -> sparse R.
```

where

$$\texttt{PX a n p} = \texttt{a} + \texttt{X}^{\texttt{n}}\texttt{p}$$

which can be used to define a relation:

```
Definition Rsparse : poly R -> sparse R -> Prop := ...
```

Data refinement: Sparse polynomials

We can define multiplication of sparse polynomials and express its correctness by:

```
Definition mul_sparse (p q : sparse R) : sparse R := ...

Lemma Rsparse_mul (x y : poly R) (x' y' : sparse R) :
  Rsparse x x' -> Rsparse y y' ->
  Rsparse (mul_poly x y) (mul_sparse x' y').
```

Data refinement: Polynomials over integers

This means that we have proved a refinement:

$$\texttt{mul\_poly int} \xrightarrow{\texttt{Rsparse\_mul}} \texttt{mul\_sparse int}$$

Data refinement: Polynomials over integers

This means that we have proved a refinement:

$$\texttt{mul\_poly int} \xrightarrow{\texttt{Rsparse\_mul}} \texttt{mul\_sparse int}$$

But, to compute efficiently we really want:

$$\texttt{mul\_poly int} \longrightarrow \texttt{mul\_sparse Z}$$

Data refinement: Polynomials over integers

This means that we have proved a refinement:

$$\texttt{mul\_poly int} \xrightarrow{\texttt{Rsparse\_mul}} \texttt{mul\_sparse int}$$

But, to compute efficiently we really want:

$$\texttt{mul\_poly int} \longrightarrow \texttt{mul\_sparse Z}$$

To get this we compose the first refinement with:

$$\texttt{mul\_sparse int} \longrightarrow \texttt{mul\_sparse Z}$$

Data refinements

The last step of the data refinement proof is found automatically by proof search (implemented using type classes) with parametricity theorems (provided by the library) and refinements of the parameters (provided by the user) as basic building blocks

Has been used in a recent formal proof that $\zeta(3)$ is irrational:

F. Chyzak, A. Mahboubi, T. Sibut-Pinote and E. Tassi. A Computer Algebra Based Formal Proof of the Irrationality of $\zeta(3)$. Interactive Theorem Proving 2014.

Constructive algebra

Constructive module theory

We take an approach similar to the one in the SSReflect library where finite dimensional vector spaces are represented using matrices and all subspace operations are defined from Gaussian elimination

- Finite dimensional vector spaces $\Longrightarrow$ Finitely presented modules
- Gaussian elimination $\Longrightarrow$ Coherent and strongly discrete rings

Finitely presented modules

An R-module $\mathcal{M}$ is **finitely presented** if it is finitely generated and there are a finite number of relations between the generators.

$$R^{m_1} \xrightarrow{M} R^{m_0} \xrightarrow{\pi} \mathcal{M} \longrightarrow 0$$

M is a matrix representing the m_1 relations among the m_0 generators of the module $\mathcal{M}$.

Finitely presented modules: example

The $\mathbb{Z}$-module $\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$ is given by the presentation:

$$\mathbb{Z} \xrightarrow{\begin{pmatrix} 0 & 2 \end{pmatrix}} \mathbb{Z}^2 \longrightarrow \mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z} \longrightarrow 0$$

as if $\mathbb{Z} \oplus \mathbb{Z}/2\mathbb{Z}$ is generated by (e_1, e_2) there is one relation, namely $0e_1 + 2e_2 = 0$.

Finitely presented modules: morphisms

A morphism between finitely presented R-modules is given by the following commutative diagram:

$$\begin{array}{ccccccc} R^{m_1} & \xrightarrow{M} & R^{m_0} & \longrightarrow & \mathcal{M} & \longrightarrow & 0 \\ \downarrow{\scriptstyle \varphi_R} & & \downarrow{\scriptstyle \varphi_G} & & \downarrow{\scriptstyle \varphi} & & \\ R^{n_1} & \xrightarrow{N} & R^{n_0} & \longrightarrow & \mathcal{N} & \longrightarrow & 0 \end{array}$$

This means that morphisms between finitely presented modules can be represented by pairs of matrices. All operations can be defined by manipulating these matrices.

Coherent and strongly discrete rings

To convieniently represent morphisms and compute their kernel the underlying ring needs to be:

- Coherent: it is possible to compute generators of the kernel of any matrix
- Strongly discrete: membership in finitely generated ideals is decidable

Examples: fields (Gaussian elimination), $\mathbb{Z}$ (Smith normal form), Bézout domains, Prüfer domains...

These rings provide the basis of the HOMALG (M. Barakat *et. al.*) computer algebra package for computational homological algebra

Abelian categories

We have formalized that the category of finitely presented modules over coherent and strongly discrete rings satisfies the axioms of **abelian categories**:

```
(* Any monomorphism is a kernel of its cokernel *)
Lemma mono_ker (M N : fpmodule R) (phi : 'Mono(M,N)) :
  is_kernel (coker phi) phi.
Proof.
split=> [|L X]; first by rewrite mulmorc.
apply: (iffP idP) => [|Y /eqmorMr /eqmor_ltrans <-]; last first.
  by rewrite -mulmorA (eqmor_ltrans (eqmorMl _ (mulmorc _))) mulmor0.
rewrite /eqmor subr0 /= mulmx1 => /dvd_col_mxP [Y Ydef].
suff Ymor : pres M %| pres L *m Y.
  by exists (Morphism Ymor); rewrite /= -dvdmxN opprB.
have := kernel_eq0 phi; rewrite /eqmor subr0 /= => /dvdmx_trans -> //.
rewrite dvd_ker -mulmxA -[Y *m phi](addrNK X%:m) mulmxDr dvdmxD.
by rewrite ?dvdmx_morphism // dvdmxMl // -dvdmxN opprB.
Qed.
```

This means that this provides a good setting for doing homological algebra.

Conclusions

CoqEAL – The Coq effective algebra library[1]

A refinement based library of computational algebra:

- Program refinements: Karatsuba polynomial multiplication $\mathcal{O}(n^{1.58})$, Strassen matrix multiplication $\mathcal{O}(n^{2.8})$, Sasaki-Murao algorithm $\mathcal{O}(n^3)$
- Data refinements: Binary integers, non-normalized rationals, list based polynomials and matrices, sparse polynomials...
- Constructive algebra: Finitely presented modules over coherent strongly discrete rings, elementary divisor rings, homological algebra...

[1] `https://github.com/CoqEAL/CoqEAL/`

Thank you for your attention!