ProgLog workshop: Formalization of Mathematics

Anders Mörtberg

Mar 7, 2014

What is being done in the area?

- Mathematical components: Formalization of the four color theorem and Feit-Thompson theorem in Coq/SSReflect
- Flyspeck: Formal proof of Kepler conjecture in HOL light
- Homotopy type theory: Formalizing mathematics in univalent foundations

A common denominator of the formal proof of the four color theorem and Flyspeck is functional programming.

What are we doing in the area?

ForMath project:

- EU FP7 STREP project 2010-2013
- Collaborators in four different countries formalizing:
  - Constructive algebra
  - Algebraic topology
  - Real number computation and numerical analysis

What are we doing in the area?

Formalization in Coq/SSReflect of:

- Program refinements: Karatsuba polynomial multiplication, Strassen matrix multiplication, Sasaki-Murao algorithm...
- Data refinements: Better datastructures for computation, proof automation using parametricity...
- Constructive algebra: Finitely presented modules, elementary divisor rings, homological algebra...

Functional programming is used to write short elegant programs that we can prove correct.

Refinements

CoqEAL – The Coq effective algebra library

A refinement based library of computational algebra:

- Program refinements: Karatsuba polynomial multiplication, Strassen matrix multiplication, Sasaki-Murao algorithm...
- Data refinements: Binary integers, non-normalized rationals, list based polynomials and matrices, sparse polynomials...

Has been used by A. Mahboubi et. al. in formal proof that $\zeta(3)$ is irrational.

Refinements

- Program refinements: Transform a program into a more efficient one computing the same thing using a different algorithm, but preserving the involved types.

- Data refinements: Change data representation on which programs operate while preserving the algorithm. This kind of refinement is more subtle as it involves transporting both programs and their correctness proofs to the new data representation. Can be partially automated using parametricity.

We have developed a general methodology for data refinements using the technique of logical relations as in Reynolds' 1983 paper: "Types, abstraction, and parametric polymorphism"

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

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
```

Sasaki-Murao algorithm

- Problem with Bareiss: Division with 0?
- Solution: Sasaki-Murao algorithm:
  - Apply the algorithm to $M - xI$
  - Compute on $R[x]$ with pseudo-division instead of $a \mid b$
  - Put $x = 0$ in the result
- Benefits:
  - More general!
  - No problem with 0 (we have x along the diagonal)
  - Get characteristic polynomial for free
  - Algorithm does not change!

Correctness proof

```
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 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.
```

Data refinement: non-normalized rational numbers

Proof oriented definition of rational numbers:

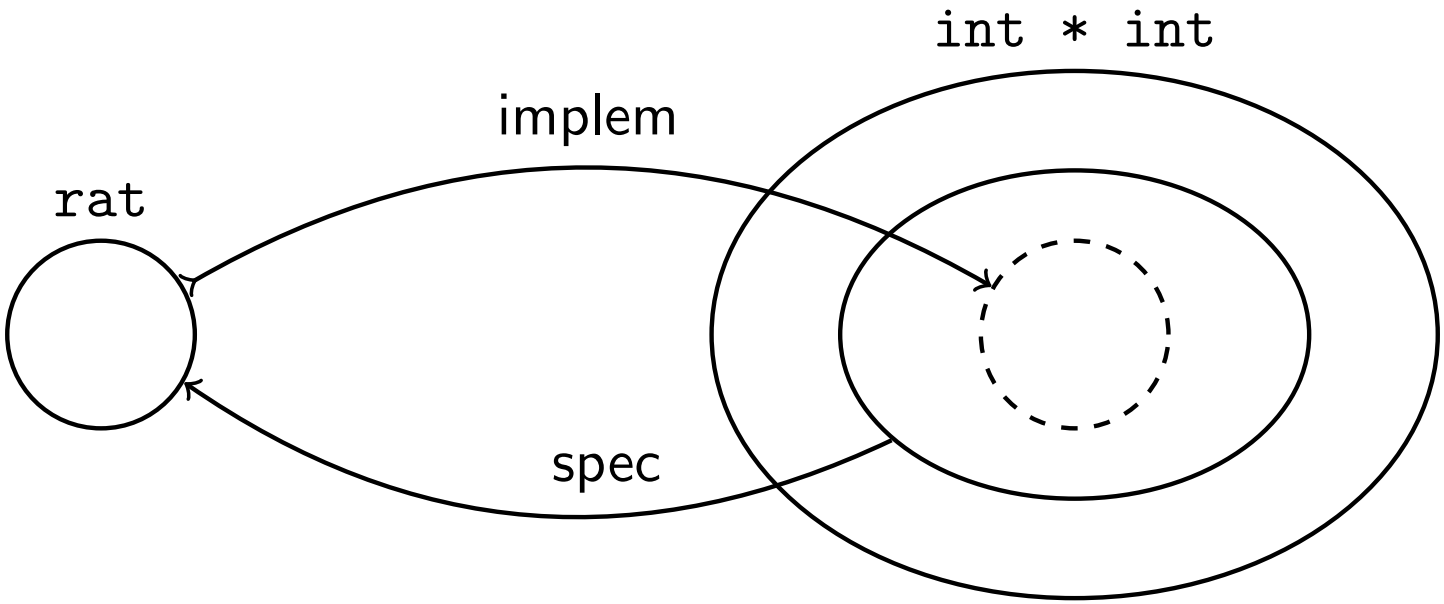
```
Record rat := Rat {
  val : int * int;
  _ : (0 < val.2) && coprime `|val.1| `|val.2|
}.

Definition fracq : int * int -> rat := ...

Definition mulq (x y : rat) :=
  let (x1,x2) := val x in
  let (y1,y2) := val y in
  fracq (x1 * y1, x2 * y2).
```

Non-normalized rationals

In order to be able to compute efficiently we would like to refine this to pairs of integers that are not necessarily normalized:



Non-normalized rationals

Define multiplication (and other operations) for pairs of integers:

```
Definition mul_int2 (x y : int * int) : int * int :=
  (x.1 * y.1, x.2 * y.2).
```

Correctness can now be stated as:

```
Definition Rrat : rat -> int * int -> Prop :=
  fun x y => y.2 <> 0 /\ x = fracq y.

Lemma Rrat_mul : ∀ (x y : rat) (x' y' : int * int),
  Rrat x x' -> Rrat y y' ->
  Rrat (mul_rat x y) (mul_int2 x' y').
```

The proof of this is easy, but in general we can have more complicated data structures and then this proof is more interesting.

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)).
```

Polynomials

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

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

Definition Rhpoly : poly R -> hpoly R -> Prop := ...

Definition mul_hpoly (p q : hpoly R) : hpoly R := ...
```

If we instantiate R with rat we can prove:

```
Lemma Rhpoly_mul :
  ∀ (x y : poly rat) (x' y' : hpoly rat),
  Rhpoly x x' -> Rhpoly y y' ->
  Rhpoly (mul_poly x y) (mul_hpoly x' y').
```

Polynomials

This means that we have proved a refinement:

$$\texttt{mul\_poly rat} \longrightarrow \texttt{mul\_hpoly rat}$$

Polynomials

This means that we have proved a refinement:

$$\texttt{mul\_poly rat} \longrightarrow \texttt{mul\_hpoly rat}$$

But, to compute efficiently we really want:

$$\texttt{mul\_poly rat} \longrightarrow \texttt{mul\_hpoly (int * int)}$$

Polynomials

This means that we have proved a refinement:

$$\texttt{mul\_poly rat} \longrightarrow \texttt{mul\_hpoly rat}$$

But, to compute efficiently we really want:

$$\texttt{mul\_poly rat} \longrightarrow \texttt{mul\_hpoly (int * int)}$$

To get this we compose the first refinement with:

$$\texttt{mul\_hpoly rat} \longrightarrow \texttt{mul\_hpoly (int * int)}$$

The proof of this is found automatically by proof search (implemented using type classes) with parametricity theorems as basic building blocks (provided by the library).

Problem: No internal parametricity in Coq

Constructive algebra

Constructive module theory

The concept of a module over a ring is a generalization of the notion of vector space over a field, where the scalars are elements of an arbitrary ring.

We restrict to *finitely presented modules* as these are the ones used in applications (control theory, algebraic topology...).

Finitely presented modules

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

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

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

Problem: How do we express that this is a restriction of the standard abstract mathematical definition of modules?

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.

If we assume that we can solve systems of equations over R we get algorithms to compute the kernel and cokernel of morphisms.

Abelian categories

We have formalized this using Coq/SSReflect and proved that it 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 finitely presented modules form a good setting for doing homological algebra.

Abelian categories: Problems

We want to formalize homological algebra at the level of abelian categories, but

Problem 1: What is a category in type theory? What notion of equality should one have for objects and morphisms?

Problem 2: Is the category of R-modules abelian?

Conclusions

Where do we want to go?

The main things we have found missing when formalizing constructive algebra and designing CoqEAL are:

- Quotients
- Internal parametricity
- Restricted to assume decidable equality (cannot perform localization of rings)
- Not clear how to define category theory, in particular we cannot conveniently work with abelian categories
- How do we express that finitely presented modules is a restriction of the standard abstract definition of modules?
- Better support for proof automation (compared to tactics, type classes, canonical structures...)

What is our vision?

A type theory that supports this!

Homotopy type theory is very promising – especially with the cubical set model that gives computational meaning to the univalence axiom

Internal parametricity in HoTT?

Thanks for your attention, now I'm going to...

Extra slides

Deciding isomorphism of finitely presented modules

It is in general impossible to decide if two matrices present isomorphic R-modules.

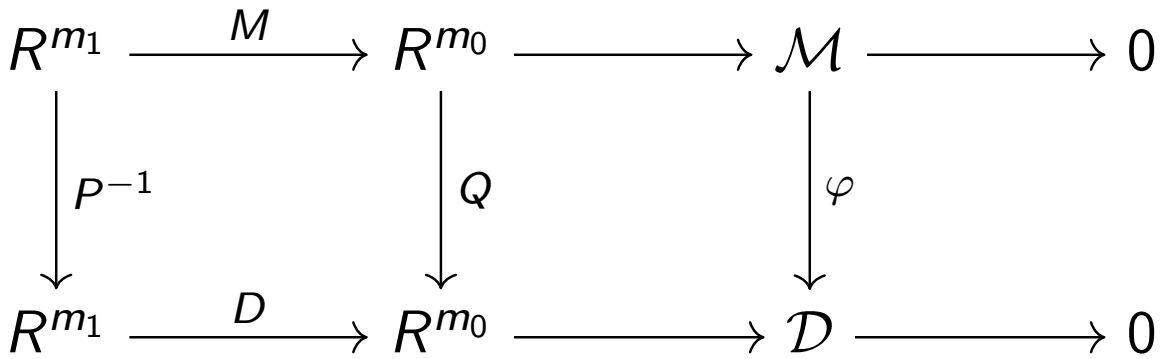
Elementary divisor rings are commutative rings where every matrix is equivalent to a matrix in Smith normal form:

$$\begin{pmatrix} d_1 & & 0 & \cdots & \cdots & 0 \\ & \ddots & & & & \vdots \\ 0 & & d_k & 0 & \cdots & 0 \\ \vdots & & 0 & 0 & & \vdots \\ \vdots & & \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & \cdots & 0 \end{pmatrix}$$

where $d_1 \mid d_2 \mid \ldots \mid d_k$.

Deciding isomorphism of finitely presented modules

Given M we get invertible P and Q such that $PMQ = D$:



Now φ is an isomorphism as P and Q are invertible.

This gives an algorithm to test if two finitely presented modules over an elementary divisor ring are isomorphic.

Bézout domains

Bézout domains are integral domains where for any two elements a and b there exists x and y such that $ax + by = gcd(a, b)$.

It is an open problem whether all Bézout domains are elementary divisor rings. We have formalized that Bézout domains extended with the following properties are:

1. Existence of a *gdco* operation that takes a and b and compute the greatest divisor of a that is coprime to b
2. Adequacy
3. Krull dimension ≤ 1 for any $a, u \in R$ there exists $v \in R$ and $m \in \mathbb{N}$ such that
$$a \mid u^m(1 - uv)$$
4. Strict divisibility is well-founded

Automated proof?

A Bézout domain is an elementary divisor ring if and only if it satisfies the **Kaplansky condition**: forall $a, b, c \in R$ with $gcd(a, b, c) = 1$ there exists $p, q \in R$ with $gcd(pa, pb + qc) = 1$.

As everything is first order we have tried to prove that Bézout domains satisfy the Kaplansky condition using automated theorem provers (without luck so far).