Article Submitted to Journal of Symbolic Computation

A Constructive Algebraic Hierarchy in Coq

Herman Geuvers, Randy Pollack, Freek Wiedijk and Jan Zwanenburg

Department of Computer Science, Nijmegen University, the Netherlands

Abstract

We describe a framework of algebraic structures in the proof assistant Coq. We have developed this framework as part of the FTA project in Nijmegen, in which a constructive proof of the Fundamental Theorem of Algebra has been formalized in Coq.

The algebraic hierarchy that is described here is both abstract and structured. Structures like groups and rings are part of it in an abstract way, defining e.g. a ring as a tuple consisting of a group, a binary operation and a constant that together satisfy the properties of a ring. In this way, a ring automatically inherits the group properties of the additive subgroup. The algebraic hierarchy is formalized in Coq by applying a combination of labeled record types and coercions. In the labeled record types of Coq, one can use *dependent types*: the type of one label may depend on another label. This allows to give a type to a dependent-typed tuple like $\langle A, f, a\rangle$, where A is a set, f an operation on A and a an element of A. Coercions are functions that are used implicitly (they are inferred by the type checker) and allow, for example, to use the structure $\mathcal{A} := \langle A, f, a\rangle$ as synonym for the carrier set A, as is often done in mathematical practice. Apart from the inheritance and reuse of properties, the algebraic hierarchy has proven very useful for reusing notations.

1. Introduction

1.1. Background

When we started working on the FTA (Fundamental Theorem of Algebra) project in the proof tool Coq [2] (see Sect. 2 for an overview of this project) we needed an algebraic framework for it. The requirements for such a framework included the following.

- We wanted to formalize a *constructive* proof of the fundamental theorem of algebra, so we needed support for constructive algebra.
- We wanted to reason about *real numbers* and polynomials over the reals.

- We wanted to use an *abstract* presentation of the real numbers that could be instantiated with various constructions in the future. In intensional type theory, such as Coq, this requires the use of *setoids*, which are a type and an equivalence relation packaged together. This allows to deal with the set-theoretic notion of a quotient, which any actual construction of the reals is a typical example of.

No algebraic developments in Coq at that time could be a base for our work, so we decided to develop our own *algebraic hierarchy* for Coq. This would be a constructive theory of the real numbers and beyond, built on a pervasive notion of constructive setoid.

1.2. Approach

We did not want our mathematics to be dependent on specific representations of the various algebraic structures, so we decided to use an *axiomatic* approach. Instead of defining the real numbers as, e.g. Cauchy sequences or Dedekind cuts, we have defined a notion of *real number structure* of which both these constructions would be an instance. For any 'implementation' of this notion all of our theory would immediately be available.

Constructive mathematics is supposed to have computational content, but a proof of FTA based on an abstract real number structure, although using only constructive principles, cannot compute the number that the theorem claims exists, because the real number structure is like an abstract data-type specification without an implementation. However, after instantiating the abstract proof with a concrete construction of the real number structure, the proof has full computational content. Indeed, the proof of FTA that we have formalized contains a root-finding algorithm. (As a matter of fact, the proof can be seen as a correctness proof of this algorithm.) It should be pointed out that, in order to do feasible computations with this algorithm the actual representation of the real numbers is crucial. The straightforward representation of reals as Cauchy sequences over $\mathbb{Q}$ the rationals, as used in [11], does not yield feasible computations.

1.3. Related work

Other algebraic hierarchies are similar to ours. However, they often have not been *used* in a large proof development. Our framework has been written and optimized for real-life use (to prove the fundamental theorem of algebra).

Paul Jackson, in his Ph.D. thesis [15], presents a constructive development of algebra in Nuprl and uses it to prove some results in abstract algebra. Nuprl, like Coq, is based on type theory, but it uses an extensional equality. This makes several constructions easier (e.g. quotienting), but renders type checking undecidable. As Nuprl does not have coercive sub-typing, there is in general no inheritance between structures.

A constructive algebraic hierarchy for Lego appears in the Ph.D. thesis of Anthony Bailey [1].

Loïc Pottier has implemented a classical algebraic hierarchy for Coq [23]. This hierarchy is more elaborate than ours and mimics the hierarchy of the Axiom computer algebra system [16].

In the FOC project, Sylvain Boulmé, Thérèse Hardin, Valérie Ménissier-Morain, Virgile Prévosto and Renaud Rioboo are building an environment to develop certified programs for symbolic computation [6]. This environment will contain a classical algebraic hierarchy. It is built from a certain kind of records with dependent labeled fields, on which several operations (refinement, extension, redefinition) have been defined [5]. A Coq formalization of the FOC algebraic hierarchy is planned. Currently only the framework of the FOC records has been formalized.

Many provers have a formalization of the real numbers. Often such a formalization is 'flat' – the type of the real numbers is not an instance of a notion of 'field' – but some are part of an algebraic hierarchy.

A flat development of the real numbers has been added to the Coq library by Micaela Mayero [8]. This development gives classical axioms for the real numbers but does not implement a specific model of it.

The Mizar system [20; 25] has both flat real numbers as well as an algebraic hierarchy. Interestingly, it is the flat real numbers that is used in most proofs that need the real numbers. The Mizar algebraic hierarchy has not been designed by one person. The various types, like `Group`, `Ring` and `Field` are defined in various articles by various authors.

1.4. Contribution

We have built a library for doing algebra and analysis in Coq. It is completely self-contained and completely constructive. That is, it uses the Coq logic (the calculus of inductive constructions) and contains no axioms. The algebraic hierarchy that we describe in this paper is part of this library. It ranges from *constructive setoids* to *real number structures*. (In [11] it is shown that the notion real number structure is categoric: any two real number structures are isomorphic.) The library also contains a construction of the complex numbers (out of a real number structure) and of the polynomials over an arbitrary ring. The complex numbers are shown to be an instance of a field and the polynomials are shown to be a ring, which allows the inheritance of properties and notation from fields, respectively rings. Finally, the library contains a construction of the real numbers as Cauchy sequences, so despite the fact that our theory is axiomatic in spirit we also give a specific construction of the field of the real numbers (see [11]).

Subjects treated in our framework are:

- groups, rings and fields
- finite sums
- polynomials

- finite dimensional vector spaces
- the real and complex numbers
- real valued functions and continuity
- the intermediate value theorem
- real and complex roots

All together our Coq code consists of approximately 35000 lines or 860 kilobytes.

The theory that we have developed contains many 'calculation lemmas'. These are lemmas that give a calculation rule used to manipulate algebraic expressions. We have created an automatic way to keep track of these lemmas in the form of a LaTeX document, in the spirit of 'literate programming'. Also we have developed specific reasoning tools to make it easier to reason inside our framework without having to know the names of the lemmas [12].

1.5. Outline of this paper

In Sect. 2 an overview of the FTA project is presented. Sect. 3 gives the notion of constructive setoid. Building on this, Sect. 4 presents the algebraic hierarchy. Sect. 5 develops our approach to representing the inheritance of common notions in the hierarchy. Sect. 6 describes the way our framework models partial functions such as division. For this we introduce the notion of *sub-setoid*. Finally Sect. 7 discusses the syntax of expressions and the complexity of the underlying Coq terms.

In order to understand this paper one must have some basic familiarity with the Coq system [2].

2. The Fundamental Theorem of Algebra project

In 1999 and 2000 in the group of Henk Barendregt at the University of Nijmegen, a proof of the fundamental theorem of algebra was formalized in the Coq proof assistant [2]. This work was done in the spirit of the QED Manifesto [7]. See [9; 10; 13] for further details of this formalization project.

The proof of the FTA project is constructive. Before the FTA project had been finished, a classical proof of the fundamental theorem of algebra had already been formalized by Robert Milewski in Mizar [19] and by John Harrison in HOL [14].

The four authors of the present paper designed the algebraic hierarchy that is part of the FTA project.

The fundamental theorem of algebra states that every non-constant polynomial over the complex numbers has a zero. In other words it says that the field of the complex numbers is algebraically closed. The proof that was formalized in Coq was a constructive proof by Hellmuth and Martin Kneser [17; 13]. We have kept the formalization free of axioms. All that we needed was already present in the Coq logic, the calculus of inductive constructions.

The final statement that we proved in the formalization was:

```
Lemma fta :
  (f:(cpoly_cring CC))(nonConst ? f)->(EX z | f!z[=]Zero).
```

This says that for every non constant polynomial f over the complex numbers there exists some complex number z such that $f(z) = 0$.

3. Setoids, constructive setoids and setoid functions

Coq is an intensional type theory. A notion called *Leibniz equality* is definable; it is the smallest reflexive relation. In a context with no assumptions about Leibniz equality, it is equivalent to the meta theoretic (intensional) *definitional equality*. On concrete constructions, such as natural numbers, where equality and identity coincide, Leibniz equality coincides with the definable structural identity. However, in abstract mathematics, this is not a very useful relation. For instance consider the representation of real numbers as Cauchy sequences. Two different Cauchy sequences (i.e. intensionally distinguished by Leibniz equality) can represent the same (extensionally) real number. If we were to assume axioms restricting Leibniz equality to behave as intended for an abstract real number structure, it would be impossible to implement that structure with a construction.

We want to make a *quotient type* by dividing out the equivalence relation 'represent the same real'. The solution is to work with *setoids* instead of raw types. A setoid is a type together with an equivalence relation on it. (This equivalence, called *setoid equality*, or *book equality*, is written `x[=]y` in our use of Coq notation.) Quotienting a setoid is achieved by changing its equivalence relation.

In constructive mathematics there is a little more to say, as the notion of *apartness* (written $x \# y$) is more fundamental than the notion of equality. No amount of information can show that concretely presented real numbers are equal. For example, consider real numbers, x and y, presented as Cauchy sequences: no matter how many terms of x and y we may have examined and found to be equal, we can not be sure that some later terms will not distinguish x from y. However, after some number of terms we may see that x and y are so far apart that, being Cauchy sequences, they cannot represent the same real. Two objects are *apart* if we 'positively' know (have evidence) that they are different. For instance a real number x is apart from 0 only if we can give some natural number n such that we know that $|x| > 1/n$.

Packaging a carrier set with an equivalence relation and an apartness relation, we get the notion of *constructive setoid*, called `CSetoid` in our framework. Apartness in such a `CSetoid` is written `x[#]y`. In Coq we define constructive setoid as a record type, as is shown on the top of the following page. A *record type* in Coq consists of *labeled tuples*, where the type of a field may be dependent on other fields. A term of type `CSetoid` is a tuple $\langle$`A,R1,R2,p`$\rangle$, with `A:Set`, `R1:(relation A)`, `R2:(relation A)` and `p:(is_CSetoid A R1 R2)`. (`p` is a proof that `A`, `R1`, `R2` have property `is_CSetoid`.) The labels allow projection of a specific field of the tuple: if `S:CSetoid`, then `(cs_crr S):Set` and `(cs_eq S):(relation (cs_crr S))`.

```
Record CSetoid : Type :=
  { cs_crr   :> Set;
    cs_eq    :  (relation cs_crr);
    cs_ap    :  (relation cs_crr);
    cs_proof :  (is_CSetoid cs_crr cs_eq cs_ap)
  }.
```

As a matter of fact, the projection `cs_crr` does not have to be written, because `cs_crr` is declared (by the annotation `:>`) to be a *coercion function*. This means that the type checker will insert this function when necessary, driven by type-checking. For example `S` is not a type, and cannot have inhabitants, but if we declare a variable `x:S`, the type checker implicitly forms the correct declaration `x:(cs_crr S)`. This captures the common mathematical usage of confusing a structure with its carrier.

In the above definition, `is_CSetoid` is a predicate, the conjunction of the defining properties of a constructive setoid. It is itself defined as a record:

```
Record is_CSetoid [A:Set; eq,ap:(relation A)] : Prop :=
  { ax_ap_irreflexive  : (irreflexive A ap);
    ax_ap_symmetric    : (symmetric A ap);
    ax_ap_cotransitive : (cotransitive A ap);
    ax_ap_tight        : (tight_apart A eq ap)
  }.
```

This says that a constructive setoid is a tuple $\langle A, \approx, \# \rangle$ that satisfies, for all x, y and z in A:

- apartness is irreflexive: $\neg(x \# x)$
- apartness is symmetric: $x \# y \to y \# x$
- apartness is cotransitive: $x \# y \to x \# z \vee z \# y$
- apartness is tight: $\neg(x \# y) \leftrightarrow x \approx y$

Because of the property of tightness we could have defined equality in terms of apartness, rather than carrying it in our setoid structure. However, equality is a very central notion in algebra (algebraic properties are equational) and we wanted the notion of constructive setoid to be a refinement of the notion of setoid. (Also to make the work readily accessible for classical mathematicians.) Therefore we didn't do this.

The field `cs_eq` of a `CSetoid` record is the function that represents the equality. Hence the Coq expression:

```
(cs_eq S x y)
```

represents $x \approx y$ in `S`. Since Coq can determine the argument `S` from the types of `x` and `y`, we can define an operator `[=]` such that `x[=]y` is a shorthand for (cs_eq S x y). Similarly we can define x[#]y as an abbreviation of (cs_ap S x y).

The notion of *setoid* allows an intensional formalization of *quotient*. A function f only induces a corresponding function on a quotient of its domain when it *respects* the equivalence $\approx$ that is being divided out:

$$x \approx y \to f(x) \approx f(y)$$

This is called *weak extensionality* or *well-definedness* of the function. It is defined in the Coq formalization as:

```
Definition fun_well_def [f:S1->S2] : Prop :=
  (x,y:S1)(x[=]y)->((f x)[=](f y)).
```

Constructively, as apartness is more fundamental than equality, so the property of *strong extensionality*

```
Definition fun_strong_ext [f:S1->S2] : Prop :=
  (x,y:S1)((f x)[#](f y))->(x[#]y).
```

is more fundamental than well-definedness. It is easily shown that strong extensionality implies well-definedness. The properties of well-definedness and strong extensionality are also defined for relations.

We have a record type of *constructive setoid functions*, consisting of pairs $\langle$f, p$\rangle$, with f:S1->S2 and p:(fun_strong_ext f). (p is a proof that f is strongly extensional.)

```
Record CSetoid_fun [S1,S2:CSetoid] : Set :=
  { csf_fun    :> S1 -> S2;
    csf_strext :  (fun_strong_ext csf_fun)
  }.
```

Due to the implicit coercions, the term csf_fun is actually a term of type (cs_crr S1) -> (cs_crr S2). This record type of functions also illustrates another use of implicit coercions: if F:(CSetoid_fun S1 S2) and a:S1, then we can apply F to a. The term (F a) is expanded by the type checker to ((csf_fun F) a).

Constructively, there are naturally occurring functions that are not well-defined in the above sense. For instance the *n-th root* $\sqrt[n]{x}$ in the complex plane; it is possible to define the n-th root of a complex number, but different representations of the same complex number will sometimes have different complex roots (not just different representations of the same complex root). So although the n-th root function can be defined it does not respect the setoid equality. Thus it is necessary to distinguish between *functions*, which are 'just' terms f of functional type S1->S2, and *setoid functions*, which should also be strongly extensional (and hence respect the equality).[*] The non-well-definedness (constructively) of some functions is unavoidable. Classically, this non-well-definedness is mirrored by the

[*]Bishop [4] calls these *operations* and *functions* respectively.

fact that a function is not continuous and is not computable on the representations. For example, the n-th root is classically non-continuous as a function from $\mathbb{C}$ to $\mathbb{C}$. (It can be made continuous by mapping to *sets* of roots, with an appropriate topology on those sets.)

4. Algebraic structures and coercive sub-typing

We have defined a number of types in Coq representing algebraic structures of which the carriers are constructive setoids. Each algebraic type is defined in terms of the previous one.

`CSetoid`	constructive setoids
`CSemi_grp`	semi-groups
`CMonoid`	abelian monoids
`CGroup`	abelian groups
`CRing`	rings
`CField`	fields
`COrdField`	ordered fields
`CReals`	‘real number structures’

We will not present the definitions of these types in detail, but refer the interested reader to the FTA files [9]. The structures of fields, ordered fields and real numbers are explained in detail in [11]. A more elaborate discussion of all the structures and the whole FTA project will appear in [10].

From the level of `CMonoid` up, all structures we deal with are assumed to be abelian. Henceforth, we will not explicitly mention this property when talking about the hierarchy: when we speak about groups we mean abelian groups.

In this paper we only present the definition of the type of rings in terms of the type of groups; the other definitions follow the same pattern. The type of rings is defined:

```
Record CRing : Type :=
  { cr_crr   :> CGroup;
    cr_one   :  cr_crr;
    cr_mult  :  (CSetoid_bin_op cr_crr);
    cr_proof :  (is_CRing cr_crr cr_one cr_mult)
  }.
```

The function `cr_crr` that gives the underlying group is a coercion (indicated by the annotation `:>`) which Coq can silently insert, as explained above (Sect. 3). So the type `CRing` is a coercive subtype of `CGroup`. For details of coercions in Coq see [24; 2]. For a more general introduction into coercive sub-typing see [3; 18; 22].

The multiplication operation of a ring is a setoid function: it respects setoid equality. We have types for such setoid functions called `CSetoid_bin_fun` and

CSetoid_bin_op (the second is a specialized case of the first in which the domain and range setoids are the same).

The defining property is_CRing is:

```
Record is_CRing
    [G:CGroup; one:G; mult:(CSetoid_bin_op G)] : Prop :=
  { ax_mult_assoc : (Associative mult);
    ax_mult_mon : (is_CMonoid
        (Build_CSemi_grp G one mult ax_mult_assoc));
    ax_dist : (Distributive mult (csg_op G));
    ax_non_triv : one[#]Zero
  }.
```

This completes the definition of rings in terms of groups.

The general scheme of defining an algebraic structure B in terms of an algebraic structure A is:

```
Record BName : Type :=
  { b_crr :> AName;
    b_opName$_1$ : $\sigma_1$;
      ⋮
    b_opName$_n$ : $\sigma_n$;
    b_proof : (is_BName b_crr b_opName$_1$ ... b_opName$_n$)
  }.

Record is_BName
    [A:AName; opName$_1$:$\sigma_1$; ...; opName$_n$:$\sigma_n$] : Prop :=
  { ax_propName$_1$ : $P_1$;
      ⋮
    ax_propName$_m$ : $P_m$
  }.
```

Note that *BName* is not a structural subtype of *AName* in the sense of having at least all the fields of *AName*. Instead *AName* occurs as a field in *BName*. (Records in Coq are *right associative* and not *extensible*, in the classification of [22].) However *BName* is a subtype of *AName* by the coercion *b*_crr, so that wherever an *AName* is expected, a *BName* can be used instead.

In Coq, any term can be declared as a coercion: if f:A->B, then we can declare f as a coercion by putting Coercion f:A>->B. (See [24] for a detailed account of coercions in Coq.) Declaring a coercion f means that the type checker will try to insert f if the term doesn't type check. If there are more coercions, the type checker will try them all, so coercions will be composed. There are some restrictions on coercions: for obvious reasons, the system does not allow to have two coercions with the same type at one time. This implies an important restriction for our algebraic hierarchy: we can not have both a coercion from a ring to its additive monoid *and* to its multiplicative monoid. So, multiple inheritance is not possible in full generality.

In our algebraic hierarchy, we have only used coercions that arise from a *field selection*, as shown in the example above where the coercion *b*_`crr` selects the first field of the labeled record type *BName*. This yields a linear hierarchy of structures, with field selecting coercions between them, as depicted in the following diagram, where the names of the coercions are omitted.

```
       CReals >-> COrdField >-> CField >-> CRing >-> CGroup
           CGroup >-> CMonoid >-> CSemi_grp >-> CSetoid
```

So, the only multiple inheritance arises from the composition of coercions: rings inherit structure (properties and operations) from groups *and* monoids *and* semigroups *and* constructive setoids.

5. Three ways to classify addition

We will now compare three ways to treat addition in Coq. The first is the way it is done in the standard Coq library, the second is a bridge to the third, which is the way it is done in the algebraic hierarchy.

5.1. The standard Coq library: separate additions

In the naive approach one defines an addition for every new type with addition that is introduced. In the Coq standard library there are three different additions for the natural numbers, the integers and the reals:

```
plus : nat->nat->nat
Zplus : Z->Z->Z
Rplus : R->R->R
```

The properties of these additions must be developed (or assumed) from scratch each time. For instance the commutativity of the addition is present three times:

```
Lemma plus_com : (x,y:nat)(plus x y)=(plus y x).
Lemma Zplus_com : (x,y:Z)(Zplus x y)=(Zplus y x).
Axiom Rplus_com : (x,y:R)(Rplus x y)=(Rplus y x).
```

In this approach, the multiplicity continues for every new type that is introduced: complex numbers, polynomials, matrices, functions, etc.

5.2. The algebraic structure as a parameter

The naturals, integers and reals are all commutative groups under addition, and one can develop the theory of addition uniformly for groups. This group addition is polymorphic in (i.e. parameterized by) the particular group:

```
Gplus : (G:Group)(crr G)->(crr G)->(crr G).
```

where the carrier function `crr:Group->Set` gives the underlying set of a group.

For any concrete structure we wish to define, (e.g. the naturals) we must still define addition and prove it satisfies the commutative group axioms, but two significant advantages are gained. First, the theory of commutative groups need only be developed once, and will be inherited by each particular group. We can also declare abstract groups, which immediately inherit all the properties of groups. Second, and very important in large scale formalization, there are uniform names for all the properties of abelian groups. Users of the formalization need only consider *one* commutative law:

```
Lemma Gplus_com :
  (G:Group; x,y:(crr G))(Gplus G x y)=(Gplus G y x).
```

This law is applicable to all groups.

5.3. Addition in the algebraic hierarchy

In the previous subsection, we have considered the advantages of classifying structures as groups. Two refinements are necessary.

- In our framework there is not just one type of structure, but a hierarchy of structures: `CSemi_grp`, `CMonoid`, `CGroup`, `CRing`, ... Addition is declared at the level of semi-groups, and inherited at more highly specified levels[†].
- Addition is not just an intensional function, but a setoid function.

Each structure is a 'subtype' of each simpler structure by a chain of forgetful coercions

```
cm_crr : CMonoid->CSemi_grp.
csg_crr : CSemi_grp->CSetoid.
cs_crr : CSetoid->Set.
```

applied in the proper order.

The addition function, called `csg_op`, is one of the fields of the `CSemi_grp` record. It has type:

```
csg_op : (G:CSemi_grp)
  (CSetoid_bin_fun (csg_crr G) (csg_crr G) (csg_crr G)).
```

This returns a `CSetoid_bin_fun` which is the type of a binary setoid function. From it one can retrieve the underlying type theoretic function by applying `csbf_fun`:

```
csbf_fun : (S1,S2,S3:CSetoid)(CSetoid_bin_fun S1 S2 S3)->
  (cs_crr S1)->(cs_crr S2)->(cs_crr S3).
```

[†]Commutativity of addition is declared at the level of monoids. This property is then inherited at higher levels. As has already been mentioned in Section 4, all structures from `CMonoid` up are abelian, also if not explicitly stated.

Putting this together the sum of `x` and `y` in a semi-group `G` will be:

```
(csbf_fun (csg_crr G) (csg_crr G) (csg_crr G) (csg_op G) x y)
```

The syntax of Coq is powerful enough to allow this to be abbreviated by:

```
x[+]y
```

The `G` argument will be determined from the types of `x` and `y`. See Sect. 7 for discussion of the complexity of the underlying representation.

The fact that in `CMonoid` the right unit of addition is unique is stated by the following Lemma.

```
Lemma cm_unit_unique_rht:
    (M:CMonoid; x:M) ((y:M)(y[+]x [=] y)) -> (x [=] Zero).
```

Because of coercions, this lemma will work for every structure that can be coerced to a monoid. Those are the groups, rings, fields, the real and complex numbers, the polynomials, etc. The two advantages mentioned in the previous subsection hold across the whole algebraic hierarchy.

There are some technical restrictions on coercions, necessary to maintain the meaning of the implicit notations. For example, it is not possible to define a ring as the 'union' of an additive and a multiplicative monoid, because there can not be two coercions from rings to monoids, see Section 4.

6. Partial functions and subsetoids

One of the main problems in formal mathematics is how to deal with partial functions. The type theoretic way to treat this problem is to add proof objects as arguments to the functions; this is the approach that we followed in our framework.

The prototypical partial function is division. In the algebraic hierarchy the expression representing x/y will have three arguments, and be written

```
x[/]y[//]H
```

where `H:(y[#]Zero)` is a proof that `y` is apart from zero. This is actually managed by having a *subsetoid* of non-zero elements. Informally, the division function might be written:

$$\cdot/\cdot : F \times F_{\neq 0} \rightarrow F.$$

Formally we define a type corresponding to $F_{\neq 0}$ using the notion of a sub-setoid.

The elements of a `subcsetoid_crr`, the carrier of a sub-setoid, are pairs of an element of setoid `S`, and a proof that the element satisfies property `P`.

```
Record subcsetoid_crr [S:CSetoid; P:S->Prop]: Set :=
  { scs_elem :> S;
    scs_prf : (P scs_elem)
  }.
```

An instance of `subcsetoid_crr` can be turned into a setoid in a canonical way, by inheriting the apartness and equality of `S`, and showing that they satisfy the required properties. This is done via the map `Build_SubCSetoid`, which takes a setoid `S` and a predicate `P` over it, and returns the setoid of elements of `S` that satisfy `P`.

Using this sub-setoid construction we can define the setoid of the non-zeroes of a ring, together with functions `nzinj` and `nzpro` (injection and projection) that relate it to the original setoid. (We only give the types of the latter two functions.)

```
Variable F : CRing.
Definition NonZeroP [x:F] : Prop := x[#]Zero.
Definition NonZeros : CSetoid :=
                          (Build_SubCSetoid F NonZeroP).
Definition nzinj : NonZeros->F := ...
Definition nzpro : (x:F)(x[#]Zero)->NonZeros := ...
```

Division in our framework is defined from the reciprocal function. This is a setoid function on the sub-setoid of the non-zeroes:

```
cf_rcpcl : (CSetoid_un_op (NonZeros F))
```

Division therefore has type

```
cf_div : (F:CField)F->(NonZeros F)->F
```

and the expression `x[/]y[//]H` (parsed as `x[/](y[//]H)`) is shorthand for

```
(cf_div F x (nzpro F y H))
```

The expression `(nzpro F y H)` represents y considered as an element of $F_{\neq 0}$.

The proof terms that occur in expressions cause some calculation rules to have more conditions than one might expect. For instance the lemma formalizing:

$$\frac{x/y}{z} = \frac{x}{y \cdot z}$$

is:

```
Lemma div_div : (x,y,z:F)(nzy:y[#]Zero)(nzz:z[#]Zero)
  (nzyz:(y[*]z)[#]Zero)
    (((x[/]y[//]nzy)[/]z[//]nzz) [=] (x[/](y[*]z)[//]nzyz)).
```

In this lemma the condition `(y[*]z)[#]Zero` is superfluous, as it is implied by `y[#]Zero` and `z[#]Zero`. However, if we omit it (and plug in a proof using `nzy` and `nzz` in the place of `nzyz`), then the lemma is harder to apply in practice.

7. Syntax and complexity of terms

We can't use the customary operator symbols in our syntax because Coq doesn't support overloading and the customary symbols are already in use. We indicate that we are using a setoid analogue of a normal operation by putting the operator in square brackets `[` and `]`. So an equation like:

$$(x+y)^2 = x^2 + 2xy + y^2$$

becomes in our syntax:

```
(x[+]y)[^](2)[=]x[^](2)[+]Two[*]x[*]y[+]y[^](2)
```

and the equation:

$$p(X) = \sum_{i=0}^{n} a_i x^i$$

becomes:

```
p!x[=](Sum (0) n [i:nat](a i)[*]x[^]i)
```

Clearly our notation could be more readable.

However, the notational features of Coq provide significant benefits, and the official terms in our framework are much more complex than the notation suggests. For instance suppose we have `x,y:IR` (where `IR` is the type of the real numbers) then

```
x[+]y
```

is an abbreviation of:

```
(csbf_fun (csg_crr (cm_crr (cg_crr (cr_crr (cf_crr (cof_crr
    (crl_crr IR)))))))
  (csg_crr (cm_crr (cg_crr (cr_crr (cf_crr (cof_crr (crl_crr
    IR)))))))
  (csg_crr (cm_crr (cg_crr (cr_crr (cf_crr (cof_crr (crl_crr
    IR)))))))
  (csg_op (cm_crr (cg_crr (cr_crr (cf_crr (cof_crr (crl_crr
    IR))))))) x y)
```

Instead of what seems to be just one function symbol `[+]`, the term actually contains 33 function symbols. This shows that the terms in our framework are relatively 'heavy'. But note that most of this big term is inferable coercions. In Luo's coercive sub-typing [18] these parts of the term are actually elided, not just suppressed in printing. This may be an important optimization for large scale formal mathematics.

8. Conclusion

We presented a framework for writing algebraic expressions in the Coq proof assistant. The features of Coq that made our approach possible were:

- record types
- coercive sub-typing
- implicit arguments

Similar features are also available in other systems (for instance the Mizar system) and therefore something like our framework can be implemented in those systems.

In practice the framework that we have presented here works well. There is hardly any duplication of theory despite the great number of algebraic structures that we defined. For example, the theory of rings has been used for the rationals, the reals, the complex numbers and the polynomials. Apart from the reuse of theory, the reuse of notation (via a form of overloading introduced by the coercion mechanism) is also very convenient. Moreover this keeps user-level expressions reasonably concise.

8.1. Future work

There are various things that need to be investigated further:

8.1.1. Better record types.

As already pointed out, one would like both the multiplicative monoid and the additive monoid of a ring to be a coercive super-type of the ring type itself. This is not allowed, because it creates two coercions of the same type. Also, one would like a sub-setoid to be a subtype of the setoid it is derived from. This coercion doesn't work well in the current version of Coq. Further research on how coercions can be improved is necessary. In [22] a start has been made with these investigations.

8.1.2. Structure of the hierarchy.

The current hierarchy has been designed to make it possible to prove the fundamental theorem of algebra. This means that it is not as rich as one would like. For instance we don't have non-commutative structures (apart from the basis `CSemi_grp`) because they didn't occur in our work.

One place where the hierarchy might be more refined is between `CField`, `COrdField` and `CReals`. Currently, the useful properties of fields of characteristic zero are derived in `COrdField`. This is not the right place because the complex numbers are not an ordered field (but they are a field of characteristic zero) so for the complex numbers these results don't apply. (We have to restate and prove these results for the complex numbers separately.) The situation could be remedied by extending the hierarchy as described in Section 4 with

```
COrdField >-> CFieldCharzero >-> CField
```

Similarly a number of the convergence notions are now defined only for `CReals`.

However this is a subtype of `COrdField` so again these results don't apply to the complex numbers. But the complex numbers do have a metric structure, and it would be desirable to have a type `CMetricField` for this. This could be done by adding the following to the hierarchy

```
CMetricField >-> CField
```

In Coq, we can have both `CMetricField` and `CFieldCharzero` in the hierarchy (as indicated above), but we can not have coercions from the complex numbers to both types, because this would produce (by composition of coercions) two coercions from the complex numbers to `CField`, which is not allowed.

8.1.3. Partial functions.

The current way Coq deals with partiality is through proof terms in the expressions. This is unnatural. The PVS system [21] offers a different solution: in PVS a partial function is a total function on its domain. Whether a partial function is defined on an element is handled through so-called 'type check conditions', which may create extra proof obligations, but don't show up in syntax. A similar approach is used in Nuprl, that has subset types. This approach works very well, but the price to be paid is that type checking becomes undecidable. This is not felt as a serious problem among PVS users. It is valuable to investigate whether a similar approach can be adapted to Coq.

8.1.4. Better syntax.

The current syntax of our framework is not very readable. The integers and reals in the Coq standard library have custom parsers that allow for the more common algebraic notation. It would be valuable to build a parser like that for the algebraic hierarchy.

8.1.5. Classical logic.

The current algebraic hierarchy is completely constructive. For many people it is irrelevant whether their reasoning is constructive or not. In their case classical logic would be much easier, and it would be useful to have a classical variant of the algebraic hierarchy. One could for instance define a notion of *decidable setoid* in which the equality is decidable. Combining this notion with the types of the algebraic hierarchy then would give a classical algebraic hierarchy.

8.2. Acknowledgements

We acknowledge the support from the EC Calculemus Network. Thanks to the anonymous referees for valuable comments. Furthermore, we want to thank Dan Synek, Henk Barendregt and Milad Niqui for valuable feedback and discussions.

References

[1] A. Bailey. *The machine-checked literate formalisation of algebra in type theory*. PhD thesis, Manchester University, 1999.

[2] B. Barras et al. *The Coq Proof Assistant Reference Manual*, 2000. URL: `<ftp://ftp.inria.fr/INRIA/coq/V6.3.1/doc/Reference-Manual-all.ps.gz>`.

[3] G. Barthe. Implicit coercions in type systems. In Stefano Berardi and Mario Coppo, editors, *Types for proofs and programs: international workshop TYPES '95, Torino, Italy: selected papers*, vol. 1185 of *LNCS*, pp. 1–15, Berlin, 1996. Springer-Verlag.

[4] E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, 1967.

[5] S. Boulmé. Opérateurs de raffinement sur les structures algébriques. In *Actes des Journées Francophones des Langages Applicatifs*, 2000.

[6] S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Proceedings of the Calculemus workshop of FLOC'99* (Federated Logic Conference, Trento, Italy), vol. 23 of ENTCS. Elsevier, 1999.

[7] R. Boyer. The QED Manifesto. In A. Bundy, editor, *Automated Deduction - CADE 12*, volume 814 of *LNAI*, pages 238–251. Springer-Verlag, 1994.

[8] D. Delahaye and M. Mayero. Field: une procédure de décision pour les nombres réels en Coq. In *Proceedings of JFLA'2001, INRIA*, 2001. URL: `<http://pauillac.inria.fr/jfla/2001/actes/04-delahaye.ps>`.

[9] The FTA project, `<http://www.cs.kun.nl/~freek/fta/>`.

[10] H. Geuvers, F. Wiedijk, J. Zwanenburg, H. Barendregt, M. Niqui, R. Pollack, Formalizing the Fundamental Theorem of Algebra, forthcoming.

[11] H. Geuvers and M. Niqui. Constructive Reals in Coq: Axioms and Categoricity. To appear in *Proceedings of the Workshop Types 2000, Durham UK,* Springer-Verlag, 2002.

[12] H. Geuvers, F. Wiedijk, and J. Zwanenburg. Equational Reasoning via Partial Reflection. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2000, Portland*, vol. 1869 of *LNCS*, pp. 162–178, Berlin, 2000. Springer-Verlag.

[13] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A Constructive Proof of the Fundamental Theorem of Algebra without using the Rationals. To appear in *Proceedings of the Workshop Types 2000, Durham UK,* Springer-Verlag, 2002.

[14] J. Harrison. Complex quantifier elimination in HOL. In Richard Boulton and Paul Jackson, editors, *TPHOLs 2001, Supplemental Proceedings*, Informatics Research Report, Division of Informatics, University of Edinburgh, pp. 159–174, 2001.

[15] P. Jackson. *Enhancing the Nuprl proof-development system and applying it to computational abstract algebra.* Ph.D. thesis, Cornell University, 1995.

[16] R. Jenks and R. Sutor. *AXIOM: The Scientific Computation System.* Springer-Verlag, Berlin, 1992.

[17] M. Kneser. Ergänzung zu einer Arbeit von Hellmuth Kneser über den Fundamentalsatz der Algebra. *Math. Z.*, 177:285–287, 1981.

[18] Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1), 1999.

[19] R. Milewski. Fundamental Theorem of Algebra. *Journal of Formalized Mathematics*, 12, 2000. MML Identifier: `POLYNOM5`.

[20] M. Muzalewski. *An Outline of PC Mizar.* Fond. Philippe le Hodey, Brussels, 1993.

[21] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, ed., *11th International Conference on Automated Deduction (CADE)*, vol. 607 of *LNAI*, pp. 748–752, Berlin, 1992. Springer-Verlag.

[22] R. Pollack. Dependently Typed Records for Representing Mathematical Structure. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2000, Portland*, vol. 1869 of *LNCS*, pp. 462–479, Berlin, 2000. Springer-Verlag.

[23] L. Pottier. Un début de formalisation de l'algèbre en Coq et Ctcoq. Web page, 2001. URL:
`<http://www-sop.inria.fr/croap/CFC/Ctcoqmath.html>`.

[24] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proc. of the 24th Symp. on Principles of Programming Languages (POPL'97)*, pp. 292–301, 1997.

[25] F. Wiedijk. Mizar: An impression. Unpublished, 1999. URL:
`<http://www.cs.kun.nl/~freek/mizar/mizarintro.ps.gz>`.