The algebraic hierarchy of the FTA project

Herman Geuvers, Randy Pollack, Freek Wiedijk, Jan Zwanenburg

University of Nijmegen

Abstract. We describe a framework for algebraic expressions for the proof assistant Coq. This framework has been developed as part of the FTA project in Nijmegen, in which a complete proof of the fundamental theorem of algebra has been formalized in Coq.

The algebraic framework that is described here is both abstract and structured. We apply a combination of record types, coercive subtyping and implicit arguments.

The algebraic framework contains a full development of the real and complex numbers and of the rings of polynomials over these fields. The framework is constructive. It does not use anything apart from the Coq logic. The framework has been successfully used to formalize non-trivial mathematics as part of the FTA project.

1 Introduction

1.1 Background

When we started working on the FTA project in the proof tool Coq [2] (see Sect. 2 for an overview of this project) we needed an algebraic framework for it. There were several requirements for such a framework.

- We wanted to translate a constructive proof of the fundamental theorem of algebra, so 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.

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 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 datatype specification without an implementation.) However, instantiating the abstract proof with any construction of the real number structure gives the proof full computational content.

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) and not as a toy exercise in abstract mathematics.

Paul Jackson, in his Ph.D. thesis [10], 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 subtyping, there is in general no inheritance between structures.

Loïc Pottier implemented a classical algebraic hierarchy for Coq. This hierarchy is more elaborate than ours and mimics the hierarchy of the Axiom computer algebra system [11]. This hierarchy has been presented in conferences (Calculemus ’98, Types ’99), but there is nothing about it in the proceedings. All that exists is a brief web page in French [18].

A constructive algebraic hierarchy for Lego in the work of Anthony Bailey has only been presented in his Ph.D. thesis [1], which is not readily available. (Bailey left research and has no web page.)

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 [5]. This development only gives axioms for the real numbers and does not implement a specific model of it. Again this work is not constructive.

The Mizar system [15,20] 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 is scattered throughout the Mizar library and has not been designed by one person. For instance the type $\text{Group}$ is in an article by Michal Muzalewski and Wojciech Skaba, the type $\text{Ring}$ is in an article by Henryk Oryszczyszyn and Krzysztof Prażmowski and the type $\text{Field}$ is in an article by Eugeniusz Kusak, Wojciech Leonczuk and Michal Muzalewski.
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. It 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 representation of the field of the real numbers.

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 developed contains many ‘calculation lemmas’. These are lemmas that give a calculation rule used to manipulate algebraic expressions. We 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 developed specific reasoning tools to make it easier to reason inside our framework without having to know the names of the lemmas [7].

1.5 Outline of the 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 subsetoid. 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 [4].

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 [14] and by John Harrison in HOL [9].

The people involved with the FTA formalization were:
- Herman Geuvers
- Randy Pollack
- Freek Wiedijk
- Jan Zwanenburg
- Milad Niqui
- Henk Barendregt

The first four people designed the algebraic hierarchy that is in this paper. Milad Niqui wrote the ‘implementation’ of the real numbers in this framework [6]. All together the formalization of the proof of the fundamental theorem of algebra took between three and four man-years.

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 translated to Coq was a constructive proof by Hellmuth and Martin Kneser [12,8]. Because of its constructive nature we could keep 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 of apartness (written \( x \neq y \)) 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: however many terms of \( x \) and \( y \) we 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:

```coq
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)
}.
```

A record type in Coq consists of labelled tuples, where the type of a field may be dependent on other fields. A term of type CSetoid is a tuple \( (A, R_1, R_2, p) \), with \( A : \text{Set}, R_1 : (\text{relation } A), R_2 : (\text{relation } A) \) and \( p : (\text{is_CSetoid } A \ 1 \ 2) \). (\( p \) is a proof that \( A, R_1, R_2 \) have property \( \text{is_CSetoid} \).) The labels allow projection of a specific field of the tuple: if \( S : \text{CSetoid} \), then \( (\text{cs_crr } S) : \text{Set} \) and \( (\text{cs_eq } S) : (\text{relation } (\text{cs_crr } S)) \). As a matter of fact, the projection \( \text{cs_crr} \) does not have to be written, because \( \text{cs_crr} \) is declared (by the annotation \( :> \)) to be a coercion function. This means that the type checker will insert this function when necessary, driven by typechecking. 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 \( x : (\text{cs_crr } S) \). This captures the common mathematical usage of confusing a structure with its carrier.

In the above definition, \( \text{is_CSetoid} \) is a predicate, the conjunction of the defining properties of a constructive setoid. It is itself defined as a record:

```coq
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: \( x \# y \rightarrow y \# x \)
- apartness is cotransitive: \( x \# y \rightarrow x \# z \vee z \# y \)
- 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. Because equality seems such an important notion we didn’t do this.

The field \( \text{cs_eq} \) of a \( \text{CSetoid} \) record is the function that represents the equality. Hence the Coq expression:

\[
(\text{cs_eq } S \ x \ y)
\]

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 \((\text{cs_eq } S \ x \ y)\). Similarly we can define \( x[=]y \) as an abbreviation of \((\text{cs_ap } S \ x \ y)\).

The notion setoid allows 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:

\[
x \approx y \rightarrow f(x) \approx f(y)
\]

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

\[
\text{Definition fun_well_def } [f:S1\rightarrow S2] : \text{Prop} := \\
(x,y:S1) (x [=] y) \rightarrow ((f x)[=] (f y)).
\]

Constructively, as apartness is more fundamental than equality, so the property of \emph{strong extensionality}

\[
\text{Definition fun_strong_ext } [f:S1\rightarrow S2] : \text{Prop} := \\
(x,y:S1) ((f x)[=] (f y)) \rightarrow (x[=] y).
\]

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.

There naturally occur 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 \emph{functions}, which are ‘just’ terms \( f \) of functional type \( S1 \rightarrow S2 \)
and setoid functions, which should also be strongly extensional (and hence respect the equality).\footnote{Bishop \cite{Bishop} calls these operations and functions respectively.} Note that the non-well-definedness of some functions is unavoidable and is not caused by the constructivity; classically, the \(n\)-th root is 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 subtyping

We 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.

\begin{verbatim}
CSetoid     constructive setoids
CSemi_grp   semi-groups
CMonoid     commutative monoids
CGroup      groups
CRing       rings
CField      fields
COrdField   ordered fields
CReals      'real number structures'
\end{verbatim}

We lack the space to present the definitions of these types in detail, and refer the interested reader to the FTA files which are available on the web at URL <http://www.cs.kun.nl/~freek/fta/>. The definitions of the types of the algebraic hierarchy have been extracted in the file Spec_CReals.v.

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:

\begin{verbatim}
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)
}.
\end{verbatim}

The function \(\text{cr\_crr}\) 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 \(\text{CRing}\) is a coercive subtype of \(\text{CGroup}\). For details of coercive subtyping in Coq see \cite{Bishop}.

The multiplication operation of a ring is a setoid function: it respects setoid equality. We have types for such setoid functions called \(\text{CSetoid\_bin\_fun}\) and \(\text{CSetoid\_bin\_op}\) (the second is a specialized case of the first in which the domain and range setoids are the same).

The defining property \(\text{is\_CRing}\) is:
Record is_CRing
  [G:Group; one:G; mult:(CSetoid_bin_op G)] : Prop :=
  { ax_mult_assoc : (Associative mult);
    ax_mult_mon :
      (is_CMono (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 [17].) 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.

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 symmetry of the addition is present three times:

Lemma plus_sym : (x,y:nat)(plus x y)=(plus y x).
Lemma Zplus_sym : (x,y:Z)(Zplus x y)=(Zplus y x).
Axiom Rplus_sym : (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:


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 symmetry law:

Lemma Gplus_sym :
(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 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, CMonoid, ... Addition is declared at the level of semi-groups,\(^2\) and inherited at more highly specified levels.
- Addition is not just an intensional function, but a setoid function.

\(^2\) In fact, for no particular reason, symmetry of addition is declared at the level of monoids.
Each structure is a 'subtype' of each simpler structure by a chain of forgetful coercions

```haskell
  cm_crr : CMonoid->CSemi_grp.
  csg_crr : CSemi_group->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:

```haskell
  csg_op : (G:CSemi_group)
         (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:

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

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

```haskell
  (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:

```haskell
  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 symmetry of addition is declared by a field in the axioms of CMonoid. The form in which this property is used is a lemma with some definitions unfolded.

**Lemma cm_commutes_unfolded:** \((\mathcal{M}:\text{CMonoid}; x,y:\mathcal{M}) (x[++]y [=] y[++]x)\).

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 coercion, necessary to maintain the meaning of the implicit notations. For example, it is not possible to declare that the additive and multiplicative monoids of a ring both coerce to the same type of structure.

## 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 \textit{subsetoid} of non-zero elements. Informally, the division function might be written:

\[
/\!\!/: F \times F_{\neq 0} \to F.
\]

Formally we define a type corresponding to \( F_{\neq 0} \) using the notion of a subsetoid.

The elements of a \texttt{subsetoid\_crr}, the carrier of a subsetoid, are pairs of an element of setoid \( S \), and a proof that the element satisfies property \( P \).

Record \texttt{subsetoid\_crr} [\( S:C\text{Setoid}; P:S\to Prop \): Set :=

\[
\{\text{scs\_elem} : S;\text{scs\_prf} : (P\text{ scs\_elem})\}\).
\]

An instance of \texttt{subsetoid\_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 \texttt{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 subsetoid construction we can define the setoid of the non-zeroes of a ring, together with functions \texttt{nzinj} and \texttt{nzpro} (injection and projection) that relate it to the original setoid. (We only give the types of the latter two functions.)

\[
\text{Variable } F : CRing.
\]

Definition \texttt{NonZeroP} [\( x:F \) : Prop := \( x[\#]Zero \).

Definition \texttt{NonZeros} : CSetoid :=

\[
\text{Build\_SubCSetoid } F \text{ NonZeroP}.
\]

Definition \texttt{nzinj} : \texttt{NonZeros}\(\to F := \ldots

Definition \texttt{nzpro} : (x:F)(x[\#]Zero)\(\to \text{NonZeros} := \ldots

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

\[
\text{cf\_rcpcl} : (C\text{Setoid\_un\_op } (\text{NonZeros } F))
\]

Division therefore has type

\[
\text{cf\_div} : (F:C\text{Field})F\to(\text{NonZeros } F)\to F
\]

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

\[
\text{(cf\_div } F \text{ x (nzpro } F \text{ y H)})
\]
The expression \( \text{nzpro} F \ y \ H \) 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:

\[
\frac{x}{y} \div z = \frac{x}{y \cdot z}
\]

is:

\[
\text{Lemma div_div} : (x, y, z : F) (\text{nz} y [\#] \text{Zero}) (\text{nz} z [\#] \text{Zero})
\]

\[
(\text{nz} y [\#] z [\#] \text{Zero})
\]

\[
(\text{nz} y [\#] z [\#] \text{Zero})
\]

\[
(\text{nz} y [\#] z [\#] \text{Zero})
\]

In this lemma the condition \( y [\#] z [\#] \text{Zero} \) is superfluous, as it is implied by \( y [\#] \text{Zero} \) and \( z [\#] \text{Zero} \). However, if we omit it (and plug in a proof using \( \text{nz} \) and \( \text{nz} \) in the place of \( \text{nz} y z \)), 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:

\[
(x[\times]y)[\cdot](2) = x[\cdot](2)[\times]Z\]\n
\[
(x[\times]y)[\cdot](2)
\]

and the equation:

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

becomes:

\[
p!x[-](\text{Sum (0) \; \{ i:nat \} (a \; i) \; [\times] \; i))
\]

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 \( x, y : \mathbb{R} \) (where \( \mathbb{R} \) is the type of the real numbers) then

\[
x[\times]y
\]

is an abbreviation of:

\[
(\text{cshr_fun} \ (\text{csg_crr} \ (\text{cm_crr} \ (\text{cg_crr} \ (\text{cr_crr} \ (\text{cf_crr} \ (\text{cof_crr} \ (\text{crl_crr} \ \mathbb{R})))))\)))
\]

\[
(\text{csg_crr} \ (\text{cm_crr} \ (\text{cg_crr} \ (\text{cr_crr} \ (\text{cf_crr} \ (\text{cof_crr} \ (\text{crl_crr} \ \mathbb{R})))))))
\]
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 subtyping [13] 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 subtyping
- implicit arguments

Similar features are also available in other systems (like 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:

**Better record types.** One would like the multiplicative monoid of a ring to be a coercive super-type of the ring type itself. Also, one would like a subsetoid to be a subtype of the setoid it is derived from. Both of these coercions don’t work well in the current version of Coq. Further research on how this situation can be improved is necessary. In [17] a start has been made with these investigations.
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 because they didn’t occur in our work.

One place where the hierarchy might be more refined is between CField, COrdField and CReals. Currently a number of the 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 so for it those results don’t apply. 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.

Partial functions. The current way Coq deals with partiality is through proof terms in the expressions. This is unnatural. The PVS system [16] 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.

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.

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

Thanks to Dan Synek and the anonymous referees for valuable comments. Due to space limits we have not been able to incorporate all of them.

References