Packaging mathematical structures

François Garillot[1], Georges Gonthier[2], Assia Mahboubi[3], Laurence Rideau[4]

[1] Microsoft Research - INRIA Joint Centre Francois.Garillot@inria.fr
[2] Microsoft Research Cambridge gonthier@microsoft.com
[3] Inria Saclay and LIX, École Polytechnique Assia.Mahboubi@inria.fr
[4] Inria Sophia-Antipolis – Méditerranée Laurence.Rideau@inria.fr

Abstract. This paper proposes generic design patterns to define and combine algebraic structures, using dependent records, coercions and type inference, inside the Coq system. This alternative to telescopes in particular supports multiple inheritance, maximal sharing of notations and theories, and automated structure inference. Our methodology is robust enough to handle a hierarchy comprising a broad variety of algebraic structures, from types with a choice operator to algebraically closed fields. Interfaces for the structures enjoy the convenience of a classical setting, without requiring any axiom. Finally, we present two applications of our proof techniques: a key lemma for characterising the discrete logarithm, and a matrix decomposition problem.

Key words: Formalization of Algebra, Coercive subtyping, Type inference, Coq, SSReflect

1 Introduction

Large developments of formalized mathematics demand a careful organization. Fortunately mathematical theories are quite organized, e.g., every algebra textbook [1] describes a hierarchy of structures, from monoids and groups to rings and fields. There is a substantial literature [2–7] devoted to their formalization within formal proof systems.

In spite of this body of prior work, however, we have found it difficult to make practical use of the algebraic hierarchy in our project to formalize the Feit-Thompson Theorem in the Coq system; this paper describes some of the problems we have faced and how they were resolved. The proof of the Feit-Thompson Theorem covers a broad range of mathematical theories, and organizing this formalization into modules is central to our research agenda. We've developed [8] an extensive set of modules for the combinatorics and set and group theory required for the "local analysis" part of the proof, which includes a rudimentary algebraic hierarchy needed to support combinatorial summations [9].

Extending this hierarchy to accommodate the linear algebra, Galois theory and representation theory needed for the "character theoretic" part of the proof has proved problematic. Specifically, we have found that well-known encodings of algebraic structures using dependent types and records [2] break down in the face of complexity; we address this issue in section 2 of this paper.

Many of the cited works focused on the definition of the hierarchy rather than its use, making simplifying assumptions that would have masked the problems we encountered. For example some assume that only one or two structures are involved at any time, or that all structures are explicitly specified. The examples in section 4 show that such assumptions are impractical: they involve several different structures, often within the same expression, and some of which need to be synthesized for existing types.

We have come to realize that algebraic structures are not "modules" in the software engineering sense, but rather "interfaces". Indeed, the mathematical theory of, say, an abstract ring, is fairly thin. However, abstract rings provide an interface that allows "modules" with actual contents, such as polynomials and matrices, to be defined and, crucially, *composed*. The main function of an algebraic structure is to provide common notation for expressions and for proofs (e.g., basic lemmas) to facilitate the composition and application of these generic modules. Insisting that an interface be instantiated explicitly each time it is used negates this function, so it is critical that structures be inferred on the fly; we'll see in the next section how this can be accomplished.

Similarly, we must ensure that our algebraic interfaces are consistent with the *other* modules in our development: in particular they should integrate the existing combinatoric interfaces [8], as algebra requires equality. As described in section 3, we have therefore adapted classical algebra to our constructive combinatorics. In addition to philosophical motivations (viz., allowing constructive proof of a finitary result like the Feit-Thompson Theorem), we have practical uses for a constructive framework: it provides basic but quite useful proof automation, via the small-scale reflection methodology supported by the SSReflect extension to Coq [10].

Due to space constraints, we will assume some familiarity with the Coq type system [11] (dependent types and records, proof types, type inference with implicit terms and higher-order resolution) in section 2, and with the basic design choices in the Feit-Thompson Theorem development [8] (boolean reflection, concrete finite sets) in sections 3 and 4.

2 Encoding structures

2.1 Mixins

An algebraic or combinatorial structure comprises representation types (usually only one), constants and operations on the type(s), and axioms satisfied by the operations. Within the propositions-as-types framework of Coq, the interface for all of these components can be uniformly described by a collection of dependent types: the type of operations depends on the representation type, and the statement (also a "type") of axioms depends on both the representation type and the actual operations.

For example, a path in a combinatorial graph amounts to

- a representation type T for nodes
- an edge relation e : `rel` T
- an initial node $x_0 : T$

– the sequence p : `seq` T of nodes that follow x_0
– the axiom p_P : `path` $e\ x_0\ p$ asserting that e holds pairwise along $x_0\ ::\ p$.

The path "structure" is actually best left unbundled, with each component being passed as a separate argument to definitions and theorems, as there is no one-to-one relation between any of the components (there can be multiple paths with the same starting point and relation, and conversely a given sequence can be a path for different relations). Because it depends on all the other components, only the axiom p_P needs to be passed around explicitly; type inference can figure out T, e, x_0 and p from the type of p_P, so that in practice the entire path "structure" can be assimilated to p_P.

While this unbundling allows for maximal flexibility, it also induces a proliferation of arguments that is rapidly overwhelming. A typical algebraic structure, such as a ring, involves half a dozen constants and even more axioms. Moreover such structures are often nested, e.g., for the Cayley-Hamilton theorem one needs to consider the ring of polynomials over the ring of matrices over a general commutative ring. The size of the terms involved grows as C^n, where C is the number of separate components of a structure, and n is the structure nesting depth. For Cayley-Hamilton we would have $C = 15$ and $n = 3$, and thus terms large enough to make theorem proving impractical, given that algorithms in user-level tactics are more often than not nonlinear.

Thus, at the very least, related operations and axioms should be packed using Coq's dependent records (Σ-types); we call such records *mixins*. Here is, for example, the mixin for a **Z**-module, i.e., the additive group of a vector space or a ring:

```
Module Zmodule.
Record mixin_of (M : Type) : Type := Mixin {
  zero : M; opp : M -> M; add : M -> M -> M;
  _ : associative add;  _ : commutative add;
  _ : left_id zero add; _ : left_inverse zero opp add
}. ...
End Zmodule.
```

Here we are using a Coq `Module` solely to avoid name clashes with similar mixin definitions.

Note that mixins typically provide only part of a structure; for instance a ring structure would actually comprise a representation type and three mixins: one for equality, one for the additive group, and one for the multiplicative monoid together with distributivity. A mixin can depend on another one: e.g., the ring multiplicative mixin depends on the additive one for its distributivity axioms.

Since types don't depend on mixins (it's the converse) type inference usually cannot fill in omitted mixin parameters; however, the *type class* mechanism of Coq 8.2 [12] can do so by running ad hoc tactics after type inference.

2.2 Packed structures

The geometric dependency of C^n on n is rather treacherous: it is quite possible to develop an extensive structure package in an abstract setting (when $n = 1$)

that will fail dramatically when used in practice for even moderate values of n. The only case when this does not occur is with $C = 1$ — when each structure is encapsulated into a single object. Thus, in addition to aesthetics, there is a strong pragmatic rationale for achieving full encapsulation.

While mixins provide some degree of packaging, it falls short of $C = 1$.

However, mixins require one object per level in the structure hierarchy. This is far from $C = 1$ because theorem proving requires deeper structure hierarchies than programming, as structures with identical operations can differ by axioms; indeed, despite our best efforts, our algebraic hierarchy is nine levels deep.

For the topmost structure in the hierarchy, encapsulation just amounts to using a dependent record to package a mixin with its representation type. For example, the top structure in our hierarchy, which describes a type with an equality comparison operation (see [8]), could be defined as follows:

```
Module Equality.
Record mixin_of (T : Type) : Type :=
 Mixin {op : rel T; _ : forall x y, reflect (x = y) (op x y)}.
Structure type : Type :=
 Pack {sort :> Type; mixin : mixin_of sort}.
End Equality.
Notation eqType := Equality.type.
Notation EqType := Equality.Pack.
Definition eq_op T := Equality.op (Equality.mixin T).
Notation "x == y" := (@eq_op _ x y).
```

Coq provides two features that support this style of interface, `Coercion` and `Canonical Structure`. The `sort :> Type` declaration above makes the `sort` projection into a *coercion* from `type` to `Type`. This form of explicit subtyping allows any T : `eqType` to be used as a `Type`, e.g., the declaration $x : T$ is understood as x : `sort` T. This allows `x == x` to be understood as `@eq_op T x x` by simple first-order unification in the Hindley-Milner type inference, as `@eq_op` α expects arguments of type `sort` α.

Coercions are mostly useful for establishing generic theorems for abstract structures. A different mechanism is needed to work with specific structures and types, such as integers, permutations, polynomials, or matrices, as this calls for construing a *more* specific `Type` as a structure object (e.g., an `eqType`): coercions and more generally subtyping will not do, as they are constrained to work in the *opposite* direction.

Coq solves this problem by using higher-order unification in combination with `Canonical Structure` hints. For example, assuming `int` is the type of signed integers, and given

```
Definition int_eqMixin := @Equality.Mixin int eqz ...
Canonical Structure int_eqType := EqType int_eqMixin.
```

Coq will interpret `2 == 2` as `@eq_op int_eqType 2 2`, which is convertible to `eqz 2 2`. Thanks to the `Canonical Structure` hint, Coq finds the solution $\alpha =$ `int_eqType` to the higher-order unification problem `sort` $\alpha \equiv_{\beta\iota\delta}$ `int` that arises during type inference.

2.3 Telescopes

The simplest way of packing deeper structures of a hierarchy consists in repeating the design pattern above, substituting "the parent structure" for "representation type". For instance, we could end `Module Zmodule` with

```
Structure zmodType : Type := Pack {sort :> eqType; _ : mixin_of sort}.
```

This makes `zmodType` a subtype of `eqType` and (transitively) of `Type`, and allows for the declaration of generic operator syntax $(0,\ x+y,\ -x,\ x-y,\ x*i)$, and the declaration of canonical structures such as

```
Canonical Structure int_zmodType := Zmodule.Pack int_zmodMixin.
```

Many authors [2, 13, 7, 5] have formalized an algebraic hierarchy using such nested packed structures, which are sometimes referred to as *telescopes* [14], the term we shall use henceforth.

As the coercion of a telescope to a representation `Type` is obtained by transitivity, it comprises a chain of elementary coercions: given T : `zmodType`, the declaration $x : T$ is understood as x : `Equality.sort`(`Zmodule.sort` T). It is this explicit chain that drives the resolution of higher-order unification problems and allows structure inference for specific types. For example, the implicit α : `zmodType` in the term $2 + 2$ is resolved as follows: first Hindley-Milner type inference generates the constraint `Equality.sort`(`Zmodule.sort` α) $\equiv_{\beta\iota\delta}$ `int`. Coq then looks up the `Canonical Structure int_eqType` declaration associated with the pair (`Equality.sort`, `int`), reduces the constraint to `Zmodule.sort` $\alpha \equiv_{\beta\iota\delta}$ `int_eqType` which it solves using the `Canonical Structure int_zmodType` declaration associated with the pair (`Zmodule.sort`, `int_eqType`). Note that `int_eqType` is an `eqType`, not a `Type`: canonical projection values are not restricted to types.

Although this clever double use of coercion chains makes telescopes the simplest way of packing structure hierarchies, it raises several theoretical and practical issues for deep or complex hierarchies.

Perhaps the most obvious one is that telescopes are restricted to single inheritance. While multiple inheritance is rare, it does occur in classical algebra, e.g., rings can be unitary and/or commutative. It is however possible to fake multiple inheritance by extending one base structure with the mixin of a second one (similarly to what we do in Section 3.2), provided this mixin was not inlined in the definition of the second base structure.

A more serious limitation is that the head constant of the representation type of any structure in the hierarchy is always equal to the head of the coercion chain, i.e., the `Type` projection of the topmost structure (`Equality.sort` here). This is a problem because for both efficiency and robustness, coercions and canonical projections for a type are determined by its head constant, and the topmost projection says very little about the properties of the type (e.g., only that it has equality, not that it is a ring or field).

There is also a severe efficiency issue: the complexity of Coq's term comparison algorithm is exponential in the length of the coercion chain. While this is clearly a problem specific to the current Coq implementation, it is hard and unlikely to be resolved soon, so it seems prudent to seek a design that does not run into it.

2.4 Packed Classes

We now describe a design that achieves full encapsulation of structures, like telescopes, but without the troublesome coercion chains. The key idea is to introduce an intermediate record that bundles all the mixins of a structure, but *not* the representation type; the latter is packed in a second stage, similarly to the top structure of a telescope. We call this intermediate record a *class*, by analogy with open-recursion models of objects, and Haskell type classes; hence in our design structures are represented by *packed classes*.

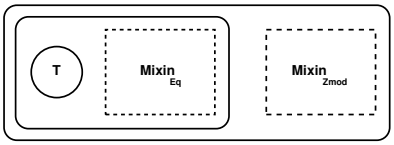

Fig. 1. Telescopes for Equality and Zmodule

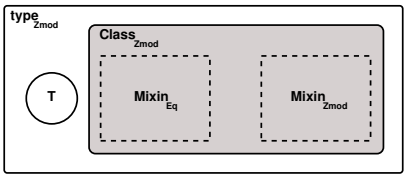

Fig. 2. Packed class for Zmodule

Here is the code for the packed class for a **Z**-module:

```
Module Zmodule.
Record mixin_of (T : Type) : Type := ...
Record class_of (T : Type) : Type :=
 Class {base :> Equality.class_of T; ext :> mixin_of T}.
Structure type : Type :=
 Pack {sort :> Type; class : class_of sort; _ : Type}.
Definition unpack K (k : forall T (c : class_of T), K T c) cT :=
 let: Pack T c _ := cT return K _ (class cT) in k _ c.
Definition pack :=
 let k T c m := Pack (Class c m) T in Equality.unpack k.
Coercion eqType cT := Equality.Pack (class cT) cT.
End Zmodule.
Notation zmodType := Zmodule.type.
Notation ZmodType := Zmodule.pack.
Canonical Structure Zmodule.eqType.
```

The definitions of the `class_of` and `type` records are straightforward; `unpack` is a general dependent destructor for `cT : type` whose type is expressed in terms of `sort cT` and `class cT`. Almost all of the code is fixed by the design pattern[5]; indeed the definitions of `type` and `unpack` are literally identical for all packed classes, while usually only the name of the parent class module (here, `Equality`) changes in the definitions of `class_of` and `pack`.

Indeed, the code assumes that `Module Equality` is similarly defined. Because `Equality` is a top structure, the definitions of `class_of` and `pack` in `Equality` reduce to

[5] It is nevertheless impractical to use the Coq `Module` construct to package these three fixed definitions, because of its verbose syntax and technical limitations.

```
Notation class_of := mixin_of.
Definition pack T c := @Pack T c T.
```

While `Pack` is the primitive constructor for `type`, the usual constructor is `pack`, whose only explicit argument is a **Z**-module mixin: it uses `Equality.unpack` to break the packed `eqType` supplied by type inference into a type and class, which it combines with the mixin to create the packed `zmodType` class. Note that `pack` ensures that the canonical `Type` projections of the `eqType` and `zmodType` structure are *exactly* equal.

The inconspicuous `Canonical Structure Zmodule.eqType` declaration is the keystone of the packed class design, because it allows Coq's higher order unification to unify `Equality.sort` and `Zmodule.sort`. Note that, crucially, `int_eqType` and `Zmodule.eqType int_zmodType` and are *convertible*; this holds in general because `Zmodule.eqType` merely rearranges pieces of a `zmodType`. For a deeper structure, we will need to define one such conversion for each parent of the structure. This is hardly inconvenient since each definition is one line, and the convertibility property holds for any composition of such conversions.

3 Description of the hierarchy

Figure 3 gives an account for the organization of the main structures defined in our libraries. Starred blocks denote algebraic structures that would collapse on an unstarred one in either a classical or an untyped setting. The interface for each structure supplies notation, definitions, basic theory, and generic connections with other structures (like a field being a ring).

In the following, we comment on the main design choices governing the definition of interfaces. For more details, the complete description of all the structures and their related theory, see module ssralg on `http://coqfinitgroup.gforge.inria.fr/`.

We do not package as interfaces all the possible combinations of the mixins we define: a structure is only packaged when it will be populated in practice. For instance integral domains and fields are defined on top of commutative rings as in standard textbooks [1], and we do not develop a theory for non commutative algebra, which hardly shares results with its commutative counterpart.

3.1 Combinatorial structures

SubType structures To handle mathematical objects like "the units of $\mathbf{Z}/n\mathbf{Z}$", one needs to define new types in comprehension-style, by giving a specification over an existing type. The Coq system already provides a way to build such new types, by the means of Σ–types (dependent pairs). Unfortunately, in general, to compare two inhabitants of such a Σ-type, one needs to compare *both* components of the pairs, i.e. comparing the elements *and* comparing the related proofs.

To take advantage of the proof-irrelevance on boolean predicates when defining these new types, we use the following `subType` structure:

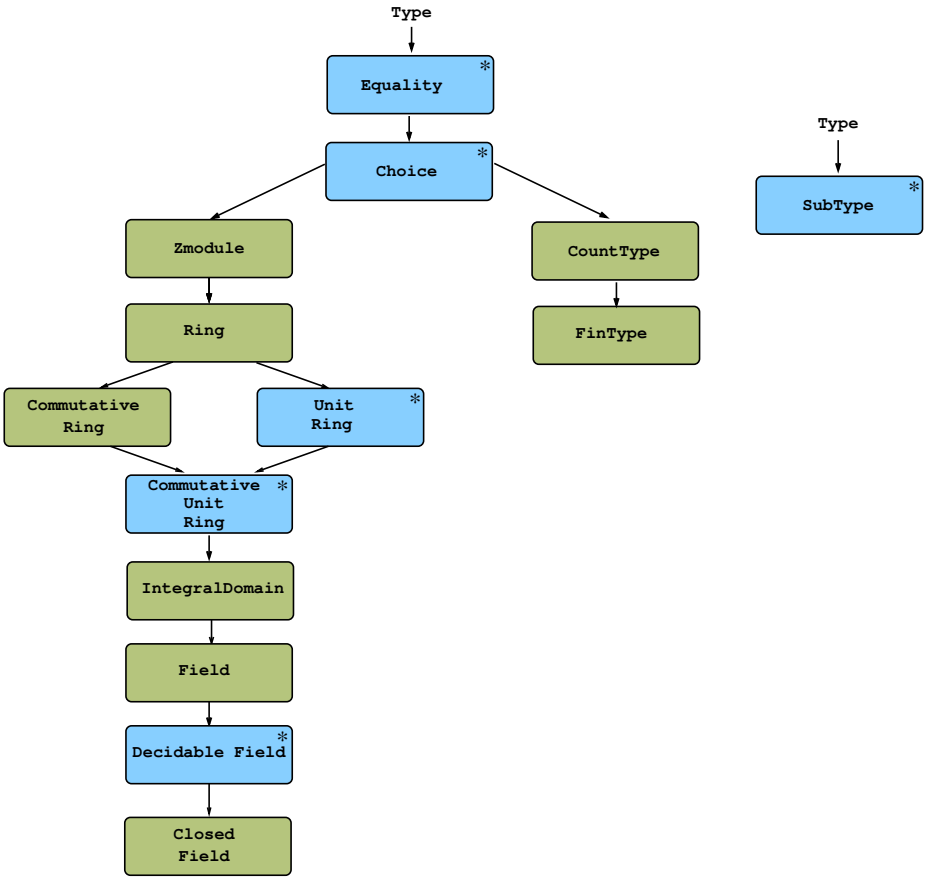

Fig. 3. The algebraic hierarchy in the ssreflect libraries

```
Structure subType (T : Type)(P : pred T): Type := SubType {
  sub_sort :> Type;
  val : sub_sort -> T;
  Sub : forall x, P x -> sub_sort;
  _ : forall K (_ : forall x Px, K (@Sub x Px)) u, K u;
  _ : forall x Px, val (@Sub x Px) = x}.
```

This interface gathers a new type `sub_sort` for the inhabitants of type `T` satisfying the boolean predicate `P`, with a projection `val` on type `T`, a `Sub` constructor, and an elimination scheme. Now, the `val` projection can be proved injective: to compare two elements of a `subType` structure on type `T` it is enough to compare their projections on `T`. A simple example of `subType` structure equips the type of finite ordinals:

```
Inductive ordinal (n : nat) := Ordinal m of m < n.
```

where `<` stands for the boolean strict order on natural numbers. Crucially, replacing a primitive Coq Σ-type by this encoding makes it possible to *coerce* `ordinal` to `nat`. Moreover, in Coq, the definition of this inductive type automatically generates the `ord_rect` associated elimination scheme. We can hence easily build a (canonical) structure of `subType` on top of `ordinal`, by providing `ord_rect` to the `SubType` constructor, as the other arguments are trivial.

Types with a choice function Our intentional, proof-irrelevant representation of finite sets was sufficient to address quotients of finite objects like finite groups [8]. However, this method does not apply to an infinite setting, where arguments like the incomplete basis theorem are pervasive.

The construction of quotients and its practice inside type theory based proofs assistants has been quite intensively studied. In classical systems like HOL, the infrastructure needed to work with quotient types is now well understood [15].

Coq provides support for `Setoid`s [16], which is a way to define quotients by explicitly handling the involved equivalence relation, and the proved substitutive contexts. In our case, quotients have to be dependent types, the dependent parameters often being themselves (dependent) quotients. This combination of dependent types with setoids, which has proved successful in an extensional setting like NuPRL [2], is not adapted to an intentional theory like the one of Coq. Crafting and implementing a Curry-Howard based system featuring the appropriate balance between intentional and extensional type theories, as well as an internalized quotient construction is still work in progress [17].

To circumvent this limitation, we combine the structure of types with equality with a choice operator on decidable predicates in a `Choice` structure. This structure, at the top of the hierarchy, is embedded in every lower level algebraic structure.

To construct objects like linear bases, we need to choose *sequences* of elements. Yet a choice operator on a given type does not canonically supply a choice operator on sequences of elements of this type. This would indeed require a canonical encoding of `(seq T)` into `T` which is in general not possible: for the empty `void` type, `(seq void)` has a single inhabitant, while `(seq (seq void))` is isomorphic to `nat`. The solution is to require a choice operator on `(seq (seq T))`. This leads to a canonical structure of choice for `T` and any `(seq .. (seq T))`, using a Gödel-style encoding.

Thus we arrive at the following definition for the `Choice` mixin and class:

```
Module Choice.
Definition xfun (T : Type) := forall P : pred T, (exists x, P x) -> T.
Definition correct (f : xfun) := forall (P : pred T) xP, P (f P xP).
Definition extensional (f : xfun) := forall P Q xP xQ,
 P =1 Q -> f P xP = f Q xQ.

Record mixin_of (T : Type) : Type := Mixin {
 xchoose : xfun T;
 xchooseP : correct xchoose;
 eq_xchoose : extensional xchoose}.
```

```
Record class_of (T : Type) : Type := Class {
  base :> Equality.class_of T; ext2 : mixin_of (seq (seq T)) }.
...
End Choice.
```

The `xfun` choice operator for boolean predicates should return a witness satisfying `P`, given a proof of the existence of such a witness. It is extensional with respect to both the proofs of existence and the predicates.

Countable structures A choice structure will still not be transmitted to any desired construction (like product) over types featuring themselves a choice structure. Types with countably many inhabitants on the other side are more amenable to transmit their countability. This leads us to define a structure for these countable types, by requiring an injection `pickle : T -> nat` on the underlying type `T`.

Since the Calculus of Inductive Constructions [11] validates the axiom of countable choice, it is possible to derive a `Choice` structure from any countable type. However since a generic choice construction on arbitrary countable types would not always lead to the expected choice operator, we prefer to embed a `Choice` structure as base class for the `Countable` structure.

Finite types structures The structure of types with a finite number of inhabitants is at the heart of our formalization of finite quotients [8]. The `Finite` mixin still corresponds to the description given in this reference, but the `FinType` structure now packs this mixin with a `Countable` base instead of an `eqType`. Proofs like the cardinal of the cartesian product of finite types make the most of this computational content for the enumeration. Indeed the use of (computations of) list iterators shrinks the sizes of such proofs by a factor of five compared to the abstract case.

3.2 Advanced algebraic structures

Commutative rings, rings with units, commutative rings with units We package two different structures for both commutative and plain rings, as well as rings enjoying a decidable discrimination of their units. The latter structure is for instance the minimum required on a ring for a polynomial to bound the number of roots of a polynomial on that ring by the number of its roots (see lemma `max_ring_poly_roots` in module `poly`). For the ring $\mathbf{Z}/n\mathbf{Z}$, this `unit` predicate selects coprimes to n. For matrices, it selects those having a non-zero determinant. Its semantic and computational content can prove very efficient when developing proofs.

Yet we also want to package a structure combining the `ComRing` structure of commutative ring and the `UnitRing` deciding units, equipping for instance $\mathbf{Z}/n\mathbf{Z}$. This `ComUnitRing` structure has no mixin of its own:

```
Module ComUnitRing.
Record class_of (R : Type) : Type := Class {
  base1 :> ComRing.class_of R;
```

```
  ext :> UnitRing.mixin_of (Ring.Pack base1 R)}.
Coercion base2 R m := UnitRing.Class (@ext R m).
...
End ComUnitRing.
```

Its class packages the class of a `ComRing` structure with the mixin of a `UnitRing` (which reflects a natural order for further instantiation). The `base1` projection coerces the `ComUnitRing` class to its `ComRing` base class. Note that this definition does *not* provide the required coercion path from a `ComUnitRing` class to its underlying `UnitRing` class, which is only provided by `base2`. Now the canonical structures of `ComRing` *and* `UnitRing` for a `ComUnitRing` structure will let the latter enjoy both theories with a correct treatment of type constraints.

Decidable fields The `DecidableField` structure models fields with a decidable first order theory. One motivation for defining such a structure is our need for the decidability of the irreductibility of representation of finite groups, which is a valid but highly non trivial [18] property, pervasive in representation theory.

For this purpose, we define a reflected representation of first order formulas. The structure requires the decidability of satisfiability of atoms and their negation. Proving quantifier elimination leads to the decidability for the full first-order theory.

Closed fields Algebraically closed fields are defined by requiring that any non constant monic polynomial has a root. Since such a structure enjoys quantifier elimination, any closed field canonically enjoys a structure of decidable field.

4 Population of the hierarchy

The objective of this section is to give a hint of how well we meet the challenge presented in section 1: defining a concrete datatype and extending it externally with several algebraic structures that can be used in reasoning on this type. We aim at showing that this method works smoothly by going through the proofs of easy lemmas that reach across our algebraic hierarchy and manipulate a variety of structures.

4.1 Multiplicative finite subgroups of fields

Motivation, notations and framework Our first example is the well-known property that *a finite multiplicative subgroup of a field is cyclic*. When applied to F^*, the multiplicative group of non-null elements of a finite field F, it is instrumental in defining the discrete logarithm, a crucial tool for cryptography. Various textbook proofs of this result exist [19, 1], prompting us to state it as:

```
1  Lemma field_mul_group_cyclic : forall (gT: finGroupType)
2          (G : {group gT}) (F : fieldType) (f : gT -> F),
3     {in G & G, {morph f : u v / u * v >-> (u * v)%R}} ->
4     {in G, forall x, f x = 1%R <-> x = 1} ->
5    cyclic G.
```

The correspondence of this lemma with its natural language counterpart becomes straightforward, once we dispense with a few notations:

`%R :` is a scope notation for ring operations.

`{group gT} :`
The types we defined in section 3 are convenient for framing their elements in a precise algebraic setting. However, since a large proportion of the properties we have to consider deal with relations between sets of elements sharing such an algebraic setting, we have chosen to define the corresponding set-theoretic notions, for instance sets and groups, as a selection of elements of their underlying type, as covered in [8].

`{morph f : u v / u * v >-> (u * v)%R} :`
This reads as :$\forall x, y,\ f(x * y) = (fx) *_R (fy)$.

`{in G, P} :`
If `P` is of the form $\forall x,\ Q(x)$ this means $\forall\ x\ \in G,\ Q(x)$. Additional `&` symbols (as in line 3 above) extend the notation to relativize multiple quantifiers.

The type of `f` along with the scope notation `%R`, allows Coq to infer the correct interpretation for `1` and the product operator on line 3. `field_mul_group_cyclic` therefore states that any finite group `G` mapped to a field `F` by `f`, an injective group morphism for the multiplicative law of `F`, is `cyclic`.[6]

Fun with polynomials Our proof progresses as follows: if a is an element of any such group C of order n, we already know that $a^n = 1$. C thus provides at least n distinct solutions to $X^n = 1$ in any group G it is contained in. Moreover, reading the two last lines of our goal above, it is clear that f maps injectively the roots of that equation in G to roots of $X^n = 1$ in F. Since the polynomial $X^n - 1$ has at most n roots in F, the arbitrarily chosen C is exactly the collection of roots of the equation in G.

This suffices to show that for a given n, G contains at most one cyclic group of order n. Thanks to a classic lemma ([19], 2.17), this means that G is cyclic.

An extensive development of polynomial theory on a unitary ring allows us to simply state the following definition in our proof script:

```
pose P : {poly F} := ('X^n - 1)%R.
```

The construction of the ring of polynomials with coefficients in a unitary ring (with F canonically unifying with such a ring) is triggered by the type annotation, and allows us to transparently use properties based on the datatype of polynomials, such as degree and root lemmas, and properties of the `Ring` structure built on the aforementioned datatype, such as having an additive inverse.

This part of the proof can therefore be quickly dispatched in Coq. The final lemma on cyclicity works with the cardinal of a partition of G, and is a good use case for the methods developed in [9]; we complete its proof in a manner similar to the provided reference.

Importing various proof contexts inside a proof script is therefore a manageable transaction : here, we only had to provide Coq with the type of a mapping to an appropriate unitary ring for it to infer the correct polynomial theory.

[6] Unlike in [8], `cyclic` is a boolean predicate that corresponds to the usual meaning of the adjective.

4.2 Practical linear algebra

Motivations Reasoning on an algorithm that aims at solving systems of linear equations seems a good benchmark of our formalization of matrices. Indeed, the issue of representing fixed-size arrays using dependent types has pretty much become the effigy of the benefits of dependent type-checking, at least for its programatically-minded proponents.

However, writing functions that deal with those types implies some challenges, among which is dealing with size arguments. We want our library to simplify this task, while sharing operator symbols, and exposing structural properties of objects as soon as their shape ensures they are valid.

LUP decomposition The LUP decomposition is a recursive function that returns, for any non-singular matrix A, three matrices P, L, U such that L is a lower-triangular matrix, U is an upper-triangular matrix, and P is a permutation matrix, and $PA = LU$.

We invite the reader to refer to ([20], 28.3) for more details about this notorious algorithm. Our implementation is strikingly similar to a tail-recursive version of its textbook parent. Its first line features a type annotation that does all of the work of dealing with matrix dimensions:

```
1  Fixpoint cormen_lup n : let M := 'M_n.+1 in M -> M * M * M :=
2   match n return let M := 'M_(1 + n) in M -> M * M * M with
3   | 0 => fun A => (1%:M, 1%:M, A)
4   | n'.+1 => fun A =>
5    let k := odflt 0 (pick [pred k | A k 0 != 0]) in
6    let A' := rswap A 0 k in
7    let Q := tperm_mx F 0 k in
8    let Schur := ((A k 0)^-1 *m: llsubmx A') *m ursubmx A' in
9    let: (P', L', U') := cormen_lup (lrsubmx A' - Schur) in
10   let P := block_mx 1 0 0 P' * Q in
11   let L := block_mx 1 0 ((A k 0)^-1 *m: (P' *m llsubmx A')) L' in
12   let U := block_mx (ulsubmx A') (ursubmx A') 0 U' in
13   (P, L, U)
14  end.
```

Here, in a fashion congruent with the philosophy of our archive, we return a value for any square matrix `A`, rather than just for non-singular matrices, and use the following shorthand:

`odflt 0 (pick [pred k:fT | P k])`
returns `k`, an inhabitant of the `finType fT` such that `P k` if it exists, and returns `0` otherwise.

`blockmx Aul Aur All Alr`
reads as $\left(\begin{smallmatrix} A_{ul} & A_{ur} \\ A_{ll} & A_{lr} \end{smallmatrix}\right)$.

`ulsubmx, llsubmx, ursubmx, lrsubmx`
are auxiliary functions that use the *shape*[7] of the dependent parameter of their argument `A` to return respectively $A_{ul}, A_{ll}, A_{ur}, A_{lr}$ when `A` is as repre-

[7] As crafted in line 2 above

sented above. Notice we will now denote their application using the same subscript pattern.

The rest of our notations can be readily interpreted, with `1` and `0` coercing respectively to identity and null matrices of the right dimension, and `(A i j)` returning the appropriate $a_{i,j}$ coefficient of `A` through coercion.

Correctness of the algorithm We will omit in this article some of the steps involved in proving that the LUP decomposition is correct: showing that `P` is a permutation matrix, for instance, involved building a theory about those matrices that correspond to a permutation map over a finite vector. But while studying the behavior of this subclass with respect to matrix operations gave some hint of the usability of our matrix library [8], it is not the part where our infrastructure shines the most.

The core of the correction lies in the following equation:

```
Lemma cormen_lup_correct : forall n A,
 let: (P, L, U) := @cormen_lup F n A in P * A = L * U.
```

Its proof proceeds by induction on the size of the matrix. Once we make sure that `A'` and `Q` (line 7) are defined coherently, it is not hard to see that we are proving is[9]:

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & & & \\ \vdots & & \mathtt{P'*A'} & \\ 0 & & & \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ \mathtt{a}_{k,0}^{-1} \cdot \mathtt{P'} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{ll}} & & & \mathtt{L'} \end{pmatrix} * \begin{pmatrix} \mathtt{A'}_{\mathtt{ul}} & \mathtt{A'}_{\mathtt{ur}} \\ 0 & \\ \vdots & \mathtt{U'} \\ 0 & \end{pmatrix} \tag{1}$$

where `P'`,`L'`,`U'` are provided by the induction hypothesis

$$\mathtt{P'}\left(\mathtt{A'}_{\mathtt{lr}} - \mathtt{a}_{k,0}^{-1} \cdot \mathtt{A'}_{\mathtt{ll}} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{ur}}\right) = \mathtt{L'*U'} \tag{2}$$

Notice that we transcribe the distinction Coq does with the three product operations involved: the scalar multiplication ($\cdot$), the square matrix product ($*$), and the matrix product, accepting arbitrary sized matrices ($*_{\mathtt{m}}$). Using block product expansion and a few easy lemmas allows us to transform (1) into:

$$\begin{pmatrix} \mathtt{A'}_{\mathtt{ul}} & \mathtt{A'}_{\mathtt{ur}} \\ \mathtt{P'} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{ll}} & \mathtt{P'} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{lr}} \end{pmatrix} = \begin{pmatrix} \mathtt{A'}_{\mathtt{ul}} & \mathtt{A'}_{\mathtt{ur}} \\ \mathtt{a}_{k,0}^{-1} \cdot \mathtt{P'} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{ll}} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{ul}} & \mathtt{a}_{k,0}^{-1} \cdot \mathtt{P'} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{ll}} *_{\mathtt{m}} \mathtt{A'}_{\mathtt{ur}} + \mathtt{L'} *_{\mathtt{m}} \mathtt{U'} \end{pmatrix} \tag{3}$$

At this stage, we would like to rewrite our goal with (2) —named `IHn` in our script—, even though its right-hand side does not occur exactly in the equation. However, SSReflect has no trouble expanding the definition of the ring multiplication provided in (2) to see it exactly matches the pattern[10] `-[L' *m U']IHn`.

[8] The theory, while expressed in a general manner, is less than ninety lines long.

[9] We will write block expressions modulo associativity and commutativity, to reduce parenthesis clutter.

[10] See [10] for details on the involved notation for the rewrite tactic.

We conclude by identifying the blocks of (3) one by one. The most tedious step consists in treating the lower left block, which depends on whether we have been able to chose a non-null pivot in creating `A'` from `A`. Each alternative is resolved by case on the coefficients of that block, and it is only in that part that we use the fact that the matrix coefficients belong to a field. The complete proof is fourteen lines long.

5 Related Work

The need for packaging algebraic structures and formalizing their relative inheritance and sharing inside proof assistants is reported in literature as soon as these tools prove mature enough to allow the formalisation of significant pieces of algebra [2]. The set-theoretic Mizar Mathematical Library (MML) certainly features the largest corpus of formalized mathematics, yet covering rather different theories than the algebraic ones we presented here. Little report is available on the organization a revision of this collection of structures, apart from comments [7] on the difficulty to *maintain* it. The Isabelle/HOL system provides foundations for developing abstract algebra in a classical framework containing algebraic structures as first-class citizens of the logic and using a type-class like mechanism [6]. This library proves Sylow theorems on groups and the basic theory of rings of polynomials.

Two main algebraic hierarchies have been built using the Coq system: the seminal abstract Algebra repository [4], covering algebraic structures from monoids to modules, and the CCorn hierarchy [5], mainly devoted to a constructive formalisation of real numbers, and including a proof of the fundamental theorem of algebra. Both are axiomatic, constructive, and setoid based. They have proved rather difficult to extend with theories like linear or multilinear algebra, and to populate with more concrete instances. In both cases, limitations mainly come from the pervasive use of setoids and the drawbacks of telescope based hierarchies pointed in section 2.

The closest work to ours is certainly the hierarchy built in Matita [21], using telescopes and a more liberal system of coercions. This hierarchy, despite including a large development in constructive analysis [22], is currently less populated than ours. For example, no counterpart of the treatment of polynomials presented in section 4 is described in the Matita system.

We are currently extending our hierarchy to extend the infrastructure to the generic theory of vector spaces and modules.

References

1. Lang, S.: Algebra. Springer-Verlag (2002)
2. Jackson, P.: Enhancing the Nuprl proof-development system and applying it to computational abstract algebra. PhD thesis, Cornell University (1995)
3. Betarte, G., Tasistro, A.: Formalisation of systems of algebras using dependent record types and subtyping: An example. In: Proc. 7th Nordic workshop on Programming Theory. (1995)
4. Pottier, L.: User contributions in Coq, Algebra (1999) Available at `http://coq.inria.fr/contribs/Algebra.html`.

5. Geuvers, H., Pollack, R., Wiedijk, F., Zwanenburg, J.: A constructive algebraic hierarchy in Coq. Journal of Symbolic Computation **34**(4) (2002) 271–286
6. Haftmann, F., Wenzel, M.: Local theory specifications in Isabelle/Isar. In: Types for Proofs and Programs, TYPES 2008 International Workshop, Selected Papers. LNCS (2009)
7. Rudnicki, P., Schwarzweller, C., Trybulec, A.: Commutative algebra in the Mizar system. J. Symb. Comput. **32**(1) (2001) 143–169
8. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A Modular Formalisation of Finite Group Theory. In: TPHOLs 2007. Volume 4732 of LNCS. (2007) 86–101
9. Bertot, Y., Gonthier, G., Ould Biha, S., Pasca, I.: Canonical big operators. In: TPHOLs 2008. Volume 5170 of LNCS. (2008) 86–101
10. Gonthier, G., Mahboubi, A.: A small scale reflection extension for the Coq system. INRIA Technical report, `http://hal.inria.fr/inria-00258384`.
11. Paulin-Mohring, C.: Définitions Inductives en Théorie des Types d'Ordre Supérieur. Habilitation à diriger les recherches, Université Claude Bernard Lyon I (1996)
12. Sozeau, M., Oury, N.: First-Class Type Classes. In: TPHOLs 2008. Volume 5170 of Lecture Notes in Computer Science., Springer (August 2008) 278–293
13. Pollack, R.: Dependently typed records in type theory. Formal Aspects of Computing **13** (2002) 386–402
14. Bruijn, N.G.D.: Telescopic mappings in typed lambda calculus. Information and Computation **91** (1991) 189–204
15. Paulson, L.C.: Defining Functions on Equivalence Classes. ACM Transactions on Computational Logic **7**(4) (2006) 658–675
16. Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. Journal of Functional Programming **13**(2) (2003) 261–293
17. Altenkirch, T., McBride, C., Swierstra, W.: Observational equality, now! In: Proceedings of the PLPV'07 workshop, New York, NY, USA, ACM (2007) 57–68
18. Olteanu, G.: Computing the Wedderburn decomposition of group algebras by the Brauer-Witt theorem. Mathematics of Computation **76**(258) (2007) 1073–1087
19. Rotman, J.J.: An Introduction to the Theory of Groups. Springer (1994)
20. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 2nd edn. McGraw-Hill (2003)
21. Sacerdoti Coen, C., Tassi, E.: Working with Mathematical Structures in Type Theory. In: Proceedings of TYPES 2007: Conference of the Types Project. Volume 4941 of LNCS., Springer (2008) 157–172
22. Sacerdoti Coen, C., Tassi, E.: A constructive and formal proof of Lebesgue Dominated Convergence Theorem in the interactive theorem prover Matita. Journal of Formalized Reasoning **1** (2008) 51–89