Integration of Multiple Formal Matrix Models in Coq

ZhengPu Shi and Gang Chen(✉)

Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China
zhengpushi@nuaa.edu.cn, gangchensh@qq.com

Abstract. Matrices are common tools in mathematics and computer science, and matrix formalization in proof assistants can provide strong support for verifying system behaviors related to matrix operations. The Coq community has proposed at least five formal matrix models, although the Coq standard library does not implement them. Developers who require a formal matrix library could have difficulty choosing what model to use. More importantly, once a choice is made, switching to another model later can be expensive. Although these matrix models have formalized matrix theory to a certain scale, demonstrating the ability to support the development of matrix theory, they have not identified the absolute advantages over other models, making it difficult for developers to choose. Moreover, the models have different data structures with completely different signatures of function and theorem, forcing developers to virtually completely rewrite matrix-related scripts when switching to a new matrix model. To address these problems, herein we undertake the following. First, we propose a unified matrix interface and integrate existing formal matrix models in the Coq community based on this interface. Secondly, we construct bijective functions between the different models to form isomorphisms, thus establishing connections between these models. We also provide technical comparison conclusions to help developers make choices. Hence, matrix formalization developers have a reference guide in the early stage and can switch to other models at a low cost at a later stage or use multiple models simultaneously with conversion assistance.

Keywords: Coq theorem prover · Formal matrix theory · Interface and implementation · Isomorphic mapping

1 Introduction

Matrix theory and linear algebra are important mathematical tools in science and engineering and are widely used in different fields such as control systems, signal processing, and neural networks [1]. For example, matrix theory is used to address kinematic and dynamic equations in flight control systems; optimization to improve the speed of matrix operations in deep learning also relies on matrix theory. Errors are easily introduced when writing matrix algorithms. Owing to

W. Dong and J.-P. Talpin (Eds.): SETTA 2022, LNCS 13649, pp. 169–186, 2022.
https://doi.org/10.1007/978-3-031-21213-0_11

the large scale of the problem space to be verified, such errors are difficult to eliminate by software testing or automatic verification methods, whereas using theorem proving can fundamentally guarantee the reliability of matrix algorithms [2–4]. Currently, there are numerous theorem-proving platforms including Coq [5], Isabelle [6], Lean [7], and HOL [8], where each platform has its independent software ecosystem. Herein, we discuss the Coq theorem prover.

At least five formal matrix libraries are available for Coq community projects [9,11,12,14,15]. Note that we use the terms matrix model, matrix library, and matrix scheme interchangeably to refer to these different matrix libraries because each library has a unique model and scheme consisting of a set of related operations and properties. Each scheme implements a partial formalization of matrix theory, including basic matrix algebra; however, the majority of these schemes do not implement the difficult parts. For example, basic operations such as matrix addition, scalar multiplication, transposition, and multiplication have been implemented, but research on the matrix canonical form, factorization, matrix sequence, and generalized inverse matrix has not been implemented. This is partly because implementing complex matrix theories in a theorem prover requires long development circles. These models appear to compete with each other, forcing users to make choices.

For developers working on formal matrix theory, easily choosing a suitable matrix scheme from these schemes is difficult, and switching to another model at a later stage is costly. Matrix theory is broad and the upper-level verifications that rely on formal matrix libraries are more diverse, making the requirement for formal matrices a critical action item. However, these existing schemes cannot support multiple requirements. First, the full-featured matrix library expected by these upper-level applications currently does not exist, as mentioned previously. Secondly, we cannot predict what scheme will necessarily be optimal in the future. Because each scheme has unique technical characteristics, the progress of the verification process could be different. For example, a theorem A could be easy to be proven in Scheme 1 while a theorem B is easy in Scheme 2; however, A could be difficult to be proven in Scheme 2 or require a longer time. Again, switching costs can be high when the chosen model does not meet the actual requirements and one wants to appraise another model. Because different models have different types of matrices, with different function and theorem types, the scripts in these schemes are not portable. Hence, developers can be reluctant to choose a certain matrix model to meet long-term project requirements in the early stage; nor can they afford the huge costs of switching to new models in the later stage, resulting in slow verification progress.

We believe that the concept of interfaces and modularity in software engineering should be used to improve this situation. Coq is a theorem prover of higher-order logic with strong expressive ability and provides techniques such as module signatures and functors to realize interface programming. Moreover, the operations and properties of matrices are mathematically well defined; hence, concrete implementations of matrix theory should have strong similarities. These two aspects support our idea that the formalization of matrix theory in Coq can

use the same interface. What is required is a method to abstract the matrix operations and properties implemented in these different models into a unified interface for external use. Because the research progress of these schemes is different, it would be helpful for these models to learn from each other by constructing transformations between themselves. In addition, because a matrix is a general concept, and its carrier can be a different type, we must also support polymorphic matrices. Hence, we have undertaken to complete the following.

1. Use Module Type to define a unified matrix interface.
2. Integrate five formal matrix schemes based on the interface.
3. Construct bijective conversion functions between these models.
4. Establish isomorphisms of certain operations between these models to provide another proof approach for certain properties.
5. Use a functor to implement polymorphic matrices and support different matrix element types such as rational numbers, real numbers, and functions.

Using this unified interface, the development of the underlying formal matrix library (FML) is effectively decoupled from the work of the upper-level matrix application (MApp), which facilitates multiteam collaboration. Matrix library developers gradually design the matrix library where multiple schemes can be performed in parallel based on a unified signature. MApp developers use this interface for matrix verification development. The relationship is shown in Fig. 1.

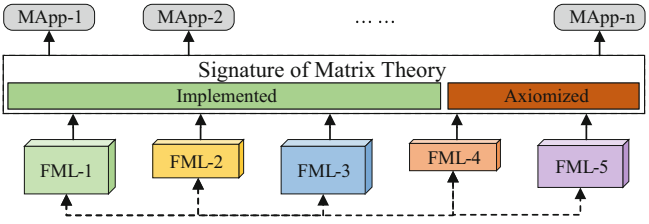


Fig. 1. Relationship between matrix interface, FML, and MApp.

The different sizes of *FML-1* to *FML-5* indicate that the existing scales of each library are different. The *Implemented* represents the formal matrix theory that has been implemented by at least one scheme. New matrix theory could be added to *Axiomized* in the form of abstract operations and axioms. After the formalization of this part, a complete verification result is obtained. The dashed lines at the bottom represent conversions between the FMLs.

Because matrices and vectors are mathematically transformable and vector theory has numerous applications, we also offer vector formalization. For example, a matrix with r rows and c columns can be viewed as either a column vector of length r or a row vector of length c. The design of the vector interface is similar to that of the matrix interface; it decouples the dependencies between the lower-level formal vector library (FVL) and upper-level vector application (VApp), as indicated in Fig. 2.

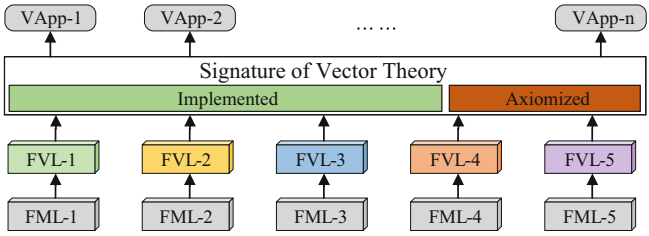


Fig. 2. Relationship between vector interface, FVL, and VApp.

FVL uses FML which is only a recommended practice. In fact, the vector signature does not force FVL to use FML because vectors and matrices are independent theoretically. In the current implementation, we define a vector of length n as an $n \times 1$ dimensional matrix followed mathematical conventions. Thus, vector theory reuses much of the matrix theory and avoids type conversions when operating between vectors and matrices. We do not overly discuss vector interfaces, which are not the focus of this paper.

The remainder of this paper is organized as follows. Section 2 presents the different existing formal matrix models. Section 3 describes the integration of the matrix library. In this section, Sect. 3.1 describes the mathematical properties used in this paper, Sect. 3.2 describes the polymorphism and hierarchy of matrix elements, Sect. 3.3 describes the unified matrix interface, Sect. 3.4 describes the implementation of the matrix, Sect. 3.5 describes the conversion between models, and Sect. 3.6 describes the isomorphism of models. Section 4 presents a comparative analysis of the models. Section 5 presents an example of using the matrix interface. Section 6 presents the conclusions of this study.

2 Different Formal Matrix Models

This section describes five formal matrix schemes presently available in the Coq community with a particular focus on matrix models. We analyze the definitions of the different matrix types that determine the type of all subsequent functions and theorems. Moreover, we briefly discuss the matrix functions implemented in the library and their technical characteristics.

2.1 DepList: Dependent List

Because this definition is similar to a list and is a dependent type, we call it *Dependent List* (*DepList* or *DL* for short). DepList is a formal matrix model used in the CoLoR project [14] and Nicolas Magaud's work [13]. CoLoR is a math library in Coq designed for theoretical research such as rewriting theory, lambda calculus, and termination; it was released in 2003 and remains valid today. The model is based on the vector type in the vector library provided by the Coq standard library. Matrix type is defined as follows:

```
Inductive vec (A : Type) : ℕ → Type :=
  | nil : vec A 0 | cons : A → ∀ n : ℕ, vec A n → vec A (S n).
Definition mat (r c : ℕ) := vec (vec A c) r.
```

Here, *Inductive* is a keyword in Coq that defines an inductive type. The vector type (called *vec*) has two cases: *nil* is a vector of length zero and *cons* is a vector of length $n + 1$ that is formed by a value of type A and a vector of length n. We use the notation $V[A]_n$ to represent a vector with a carrier of type A with n elements. Thus, a matrix type (*mat*) is regarded as a vector of vectors. Therefore, the operations on matrices are actually operations on vectors. The library implements the following: matrix definition on *setoid*, obtain the matrix elements, identity matrix, transposition, addition, multiplication, and related properties. A setoid is a set equipped with an equivalence relation, which is a general case of equality. In addition, the library supports polymorphic matrices using a *functor*. A functor is a *module* with a parameter called the *module type*; this *module* is a common technique in Coq for packaging a set of operations. The library requires the carrier to be *SemiRing*, meaning that a ring is not required to provide an additive inverse. Therefore, data types such as $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ can be used as carriers to form a matrix. The features of this model are as follows.

- **Advantage**
  - Development based on the Coq standard library easily attracts developers.
  - Inductively defined data structures are easy to prove and list-like data structures are easier to understand.
  - The definition of setoid-based equality has wider applicability than Leibniz equality.
  - It is dimensionally scalable because higher-dimensional arrays can be defined by a nested vec structure.
- **Disadvantage**
  - The content of the vector library has not yet reached the maturity of the list library and requires significant development.
  - Proofs are roughly trickier than the ordinary list structures.

2.2 DepPair: Dependent Pair

Because this definition is similar to a pair and is a dependent type, we call it *Dependent Pair* (*DepPair* or *DP* for short). DepPair is a matrix model adopted in the Coquelicot project [12], a research project for real analysis funded by the French Foundation Centre. They proposed a compact representation of the matrices as follows:

```
Fixpoint Tn (A : Type) (n : ℕ) : Type :=
  match n with O ⇒ unit | S n ⇒ prod A (Tn A n) end.
Definition mat A (r c : ℕ) := Tn (Tn A c) r.
```

Here, the *Tn* function constructs a nested pair of length n of base type A. *Fixpoint* is a keyword for defining recursive functions in Coq, *unit* is a single-element set with only one member *tt*, and *prod* is product type with the constructor *pair*. Then, the *Tn* type represents a vector of length n, and the matrix type is defined as the Tn of Tn, i.e., a vector of vectors. This definition is similar to that of DepList in Sect. 2.1, except that the inductive definition is replaced by a recursive function. The library implements the following: construction of matrices, identity matrix, addition, multiplication, and related properties. In addition, the matrix carrier types are organized into a hierarchy in this scheme. For example, the carrier of matrix addition forms an abelian group, the carrier of matrix multiplication forms a monoid, and matrix addition and multiplication together form a ring. The features of this model are as follows.

- **Advantage**
  - Matrix construction is simple; only using a pair structure can create a matrix.
  - The structure is extensible because the carrier is carefully designed based on mathematical hierarchy.
  - Dimensionally scalable because higher-dimensional arrays can be defined by the nested Tn structure.
- **Disadvantage**
  - Not a significant amount of matrix theory has been implemented.
  - The skill for dependently typed proof is required to complete the proof.

2.3 DepRec: Dependent Record

Because the definition is in the form of a *Record*, and the fields of the record depend on the parameters and previous fields, we call it *Dependent Record* (*DepRec* or *DR* for short). In computer science, record is a data structure that can hold multiple named fields. The model was proposed by Zhenwei et al. [9].

```
Fixpoint width {A : Type} (dl : list (list A)) (n : ℕ) : Prop :=
  match dl with nil ⇒ True | cons x t ⇒ (length x = n) ∧ width t n end.
Record matrix {A : Type} {r c : ℕ} : Type := mkMatrix {
  mdata: list (list A);   matH: length mdata = r;   matW: width mdata c}.
```

Here, *length* returns the length of a list, and *width* indicates that each list item in the type *list(list A)* has a given length. A matrix is defined as a record with three fields: a data *mdata*, a proof *matH* indicating that mdata has r rows, and a proof *matW* indicating that mdata has c columns. The library implements the following: matrix construction, zero matrix, identity matrix, addition, scalar multiplication, transposition, multiplication, and related properties. Besides, a block matrix where the matrix elements are a matrix of a specified shape is also implemented [10]. This scheme separates the computation from the proof with clear logic. The matrix construction is divided into two parts. The first part is the computation, mainly based on list operations, which have a rich implementation in the standard library. The second part is the matrix row proof

matH and column proof matW, which are easy to perform. In addition, when extracting functional programs such as OCaml, the model can obtain the most concise code that completely corresponds to the list type in OCaml and thus reduces the storage space overhead. The features are as follows.

- **Advantage**
  - Mature list libraries provide abundant support for matrix development.
  - The separation of computation and proof make the logic clear.
  - The extracted OCaml program has a concise data structure.
- **Disadvantage**
  - The script is longer because it requires separate construction of the data and provides two separate proofs.
  - It is marginally less scalable because it is currently fixed on two-dimensional arrays and cannot implement higher-dimensional arrays directly.

2.4 NatFun: Function with Natural Indexing

A matrix is defined as a function of two natural numbers (corresponding to the index of row and column) to a complex number; therefore, we call it a *Function with Natural indexing* (*NatFun* or *NF* for short). This scheme appeared in the quantum computing work of Rand et al. [15,16]. Because their work was oriented towards complex numbers, no other types of carriers were considered.

```
Definition Matrix (r c : ℕ) := ℕ → ℕ → ℂ.
```

Here, the function *Matrix* uses two arguments, r and c, as the number of rows and columns of the matrix, and returns a function. The problem is that these two arguments fail to constrain the shape of the matrix, which we address in Sect. 3.4. The library implements the following: conversion between list and matrix, zero matrix, identity matrix, addition, scalar multiplication, multiplication, transposition, trace, and related properties. Because the functional style does not have to maintain the storage structure, the matrix operations are extremely concise. Examples are defined as follows:

```
Variable C0 C1 : ℂ.  Variable Cplus Cmult : ℂ → ℂ → ℂ.
Infix "+" := Cplus. Infix "×" := Cmult.
Fixpoint Csum (f : ℕ → ℂ) (n : ℕ) : ℂ :=
  match n with O ⇒ C0 | S n' ⇒ Csum f n' + f n' end.
Definition I (n : ℕ) : Matrix n n := fun i j ⇒ if (i =? j) then C1 else C0.
Definition Mmult {r c s: ℕ} (A: Matrix r c) (B: Matrix c s) : Matrix r s :=
  fun x z ⇒ Csum (fun y ⇒ A x y × B y z) c.
```

Here, *Csum* is the sum of the sequence, I is the identity matrix, and function *Mmult* is matrix multiplication. As can be observed, this definition is concise and expressive. The features of this model are as follows.

- **Advantage**
  - The definition of the function is concise.
  - The proof of the property is concise.

– **Disadvantage**
  - There are no corresponding structured data types. Although functions are first class in functional languages and have no impact on them.
  - The two parameters used to describe the shape in the matrix definition fail to constrain this type. However, a small improvement can fix this.

2.5 FinFun: Function with Finite Indexing

This definition is similar to NatFun in Sect. 2.4; just replaces the natural indexing with a finite set of elements; hence we call it *Function with Finite indexing* (*FinFun* or *FF* for short). This model appears in the *mathematical component* library (MC) [11], which forms the infrastructure for machine-checked proofs of the well-known four-color theorems. Although this scheme was introduced last, the MC library is the most complete and complex formalization of matrices. Following is the definition of the matrix.

```
From mathcomp Require Import eqtype seq finfun fintype.
Variant matrix {R : Type} {m n : ℕ } : predArgType :=
  Matrix of {ffun 'I_m × 'I_n → R}.
```

It can be observed that the definition of the matrix introduces several types and notations. These are internally defined as follows:

```
Inductive ordinal (n : ℕ) : Type := Ordinal (m : ℕ) (_ : (S m) ≤? n).
Notation " 'I_' n" := (ordinal n).
Variant phant (p : Type) : Prop := Phant : phant p.
Notation "{ 'ffun' fT }" := (finfun_of (Phant fT)) (at level 0).
Variables (aT : finType) (rT : aT →Type).
Inductive finfun_on : seq aT → Type := | finfun_nil : finfun_on [::]
  | finfun_cons x s of rT x & finfun_on s : finfun_on (x :: s).

Variant finfun_of (ph : phant (∀ x, rT x)) : predArgType :=
  FinfunOf of finfun_on (enum aT).
```

Here, *ordinal n* represents a set of ordinal numbers less than n. The *phant* type is used to generate constraints to maintain the consistency of the type. The *finfun_on* type wraps lists of different lengths into different types. The *funfun_of* type specifies funfun_on as the only source of construction. It can be observed that the internal structure of the matrix type is very complex.

The formal matrix theory of this library is extremely rich, covering virtually all schemes in previous sections, and adds the following: block matrix of different shapes, determinant, adjoint matrix, inverse matrix, and LU decomposition. In addition, proof scripts are typically short because of deep extensions to the *SSR* language. For example, the commutative law of matrix multiplication can be performed in only three lines, whereas schemes using lists require tens to hundreds of lines. In addition, the library creates an 11-level hierarchy that corresponds well with the mathematical concepts. However, the SSR language is not friendly to new users because of its large number of notations, and its complex hierarchy requires more mathematical knowledge and programming skills. A more important problem is that a number of the expressions cannot be reduced to a simple

form; therefore, we cannot obtain the results of matrix operations intuitively. For example, obtaining matrix elements produces a complex expression instead of the expected value. The features of this model are as follows.

- **Advantage**
  - Implemented matrix theory is the most numerous of all schemes
  - The proof is concise owing to the SSR language extension.
  - It allows the careful organization of matrix operations and properties owing to a clear mathematical hierarchy.
- **Disadvantage**
  - Requires more math background to start; hence, it is not "newbie" friendly.
  - SSR extensions and too many notations are difficult to understand.
  - The inability to reduce to a simple form prevents symbolic computation.

3 Integrated Matrix Library

As can be observed from Sect. 2, each of these schemes has advantages and disadvantages, and not one can satisfy all cases. The first three schemes are more suitable for matrix symbolic computation, whereas the latter two are more suitable for verifying mathematical properties. Developers can make different choices based on the requirements such as technical difficulty, scalability of the design, or simplicity of the extracted functional code. As envisioned in Sect. 1, our next step is to integrate these disparate schemes to reduce the technical barriers. First, definitions of the mathematical properties used are provided. Subsequently, several approaches to polymorphic matrices and hierarchies are discussed. Then, we propose a matrix interface based on which of the above models are integrated. Furthermore, conversions and isomorphisms between the models are established.

3.1 Mathematical Properties

The mathematical properties used in this paper are as follows:

```
Variable A B : Type.  Variable a b c : A.
Variable Aop1 Aop2: A → A → A.  Variable Bop1: B → B → B.
Infix "+" := Aop1.  Infix "×" := Aop2.  Infix "⊕" := Bop1.
Definition eqdec := {a = b} + {a ≠ b}.
Definition inj (f: A → B) := ∀ (a1 a2 : A), f a1 = f a2 → a1 = a2.
Definition surj (f: A → B) := ∀ (b : B), (∃ (a : A), f a = b).
Definition bij (f: A → B) := inj f ∧ surj f.
Definition comm:= a + b = b + a.
Definition assoc:= a + (b + c) = (a + b) + c.
Definition distr_l := (a + b) × c = a × c + b × c.
Definition homo (f:A → B) := f (a + b) = (f a) ⊕ (f b).
```

Here, *eqdec* indicates that the equality of set A is decidable; inj, $surj$, and bij indicate that the function f is injective, surjective, and bijective; $comm, assoc$ indicate that the binary operation $+$ satisfies the commutative or associative law; $distr\_l$ represents the left distributive law of $\times$ over $+$; *homo* means that f is a homomorphic mapping between $\langle A, + \rangle$ and $\langle B, \oplus \rangle$.

3.2 Polymorphism and Hierarchy of Matrix Elements

A matrix is a generic data structure whose carrier can be of any type, such as a Boolean, natural, integer, real, complex, function, or matrix. In Coq, polymorphic functions can be defined directly, or the functor can be used to implement polymorphism at the module level.

Polymorphic matrices are overly flexible, and fewer type constraints can lead to redundant code, which can be avoided when the design follows a hierarchy of matrix carriers. For example, when proving the associativity law for matrix addition, the associativity of the carrier must be explicitly provided and can be omitted on *monoid* structure. There are at least three techniques in Coq to implement hierarchy.

1. *Type Classes*, which appear in the Coq standard library [5] and the work [18,19]. This is a method of generating abstract structures by overloading notations and contexts, allowing flexible combinations of different fragments to build complex theories. However, this approach causes system complexity to grow rapidly with the size [20,21].
2. *Canonical Structures*, which appear in the MC project [11] and Assia Mahboubi's work [22]. The idea is to use the hint database to extend the unification algorithm of the Coq system to solve type-inference problems from general to individual. In the MC, an 11-levels hierarchical structure was implemented in this manner with acceptable scalability and standardization.
3. Use the built-in module type in Coq. This method does not require significant skill to address type issues; however, the build process could not be sufficiently flexible. For example, when combining an additive group $\langle A, + \rangle$ and multiplicative semigroup $\langle B, * \rangle$ into a ring $\langle R, +, * \rangle$, the types of carriers A and B cannot be unified to R easily.

Although there has been related work on algebraic structures, they still cannot address our requirements. For example, in the CoLoR [14], MC [11], and Coquelicot projects [12], and the work of Herman Geuvers et al. [17], these implementations are overly complex for our matrix integration work. We consider primarily the realization of the matrix theory of rational, real, complex numbers and funtions which are commonly used in engineering, particularly automatically solving equations on matrix elements using the *ring* and *field* tactics in Coq. Therefore, we only consider the two-level structure of the ring and field; fine-grained algebraic structures have not yet been added, such as Group, Monoid, and Semi-Group. To maintain consistency, we manually built the *RingSig* and *FieldSig* module types based on the Coq standard library, as follows.

```
Require Import Ring Field.
Module Type RingSig.
Parameters (A: Type) (A0 A1: A) (Aadd Amul: A → A → A) (Aopp: A → A).
Notation Asub := (fun x y ⇒ (Aadd x (Aopp y))).
Parameter Ring_thy : ring_theory A0 A1 Aadd Amul Asub Aopp eq.
Add Ring Ring_thy_inst : Ring_thy.  End RingSig.

Module Type FieldSig <: RingSig.
Parameter Ainv: A → A.  Notation Adiv:= (fun x y ⇒ (Amul x (Ainv y))).
Parameters (Field_thy: field_theory A0 A1 Aadd Amul Asub Aopp Adiv Ainv
eq).  Add Field Field_thy_inst : Field_thy.  End FieldSig.
```

We first import two libraries, *Ring* and *Field*, to enable ring and field theories in Coq. The keyword *Module Type* is used to create modules that can be used as arguments for other modules. *A* is the type of carrier; *Aadd, Amul, Aopp,* and *Asub* are addition, multiplication, additive inverse, and subtraction operations; *A*0 and *A*1 are the additive and multiplicative identities. *Ring_thy* is a proof for constructing a ring structure. Next, the syntax *Add Ring* registers this ring structure with Coq, enabling the ring tactic on A. FieldSig is similar to RingSig, adding more operations such as multiplicative inverse and division, and registering the field structure to Coq.

After building a concrete module that satisfies the signature of RingSig or FieldSig, a ring or field structure can be obtained. For example, the code for Qc (canonical rational number) is as follows:

```
Require Export Qcanon.
Module RingQc <: RingSig.  Definition A:= Qc.
Definition A0:= 0.  Definition A1:= 1.  Definition Aadd:= Qcplus.
Definition Amul := Qcmult. Definition Aopp:= Qcopp.
Lemma Ring_thy: ring_theory A0 A1 Aadd Amul Qcminus Aopp eq.
Add Ring Ring_thy_inst : Ring_thy.  End RingQc.
```

In this manner, we can define more ring and field structures for other data types such as real and complex numbers.

3.3 Matrix Interface

The matrix interface or matrix theory signature is defined in *MatrixThySig*. For simplicity, it is not organized based on a carrier hierarchy; rather, it adopts a field structure that meets general requirements, as follows.

```
Module Type MatrixThySig.
Parameter A: Type.  Variable r c s t: ℕ.
Parameters (mat: ℕ → ℕ → Type) (meq_dec: eqdec (@mat r c)
(l2m: list (list A) → mat r c)  (m2l: mat r c → list (list A).
(l2m_bij: bij l2m)  (m2l_bij: bij m2l)  (l2m_m2l_id: ∀ m, l2m (m2l m) = m)
(m2l_l2m_id: ∀ dl, length dl = r → width dl c → m2l (l2m dl) = dl)
(mat0 mat1 : mat r c)  (madd : mat r c → mat r c → mat r c)
(mcmul : A → mat r c→ mat r c)  (mtrans : mat r c → mat c r)
```

```
(mmul : mat r c → mat c s → mat r s).  Infix "×" := mmul.
Parameters (mmul_assoc : ∀ (m1 : mat r c) (m2 : mat c s)
  (m3 : mat s t), (m1 × m2) × m3 = m1 × (m2 × m3)).
End MatrixThySig.
```

Here, A denotes the type of carrier, and matrix type mat is defined as a function determined by the number of rows and columns; $eqdec$ indicates that matrix equality is decidable; $l2m$ and $m2l$ denote conversion operations between matrices and lists (list A); $l2m\_bij$, $m2l\_bij$, $l2m\_m2l\_id$, and $m2l\_l2m\_id$ indicate that l2m and m2l are bijective functions and inverse functions of each other; $mat0$ and $mat1$ generate a zero matrix and unit matrix; $madd, mcmul, mtrans$, and $mmul$ are matrix addition, scalar multiplication, transposition, and multiplication. Owing to space limitations, other operations and properties of the matrices are not listed.

3.4 Matrix Implementation

We restructured or reimplemented all the schemes based on the MatrixThySig interface and finally combined them. Only the key issues are discussed below; the specific implementation is not introduced here.

Framework for Matrix Theory Implementation. A typical framework for matrix theory is a functor with a parameter of type $FiledSig$ as indicated below.

```
Module MatrixThy (E : FieldSig) <: MatrixThySig.
  Import X.  Definition mat := @matrix E.A.  (* other things ... *)
End MatrixThy.
```

Here, X represents a specific scheme that should implement full matrix operations in its own style. The $MatrixThy$ module calls the content in X and completes it based on the unified interface MatrixSig.

Corrected Definition for NatFun. The definition of the matrix types in many schemes conforms to MatrixSig. However, the NatFun scheme must be modified because its two parameters cannot constrain the type, resulting in an inability to distinguish matrices of different shapes. See the code below.

```
Variable A : Type.  Definition Matrix (r c : ℕ) := ℕ → ℕ → ℂ.
Variable mat1 : Matrix 2 3.  Check mat1 : Matrix 3 5.
Record mat (r c : ℕ) := mkMat {mdata : ℕ → ℕ → A}.
Variable mat2 : mat 2 3.  Fail Check mat2 : mat 3 5.
```

Here, $Matrix$ is the matrix definition used in NatFun. $mat1$ appears to be a 2×3 matrix; however, it also passes a type check as a 3×5 matrix. It seems the parameters r and c have no type constraints. If wrapped with a record type mat, it becomes a dependent type. Therefore, the shape parameters become part of the type, e.g., $mat2$ has a unique 2×3 matrix type.

Collection of All Matrix Implementations. After multiple matrix models are provided based on the same interface, we organize all these implementations; then, we can manage matrix module instances at the module level.

```
Module MatrixAll (E : FieldSig).
  Module DP := DepPair.MatrixThy E.  Module DL := DepList.MatrixThy E.
  Module DR := DepRec.MatrixThy E.  Module NF := NatFun.MatrixThy E.
  Module FF := FinFun.MatrixThy E.
End MatrixAll.
Module MatrixR := MatrixAll FieldR.  Module MatrixR_DP := MatrixR.DP.
Module MatrixR_DL := MatrixR.DL.  Module MatrixR_DR := MatrixR.DR.
Module MatrixR_NF := MatrixR.NF.  Module MatrixR_FF := MatrixR.FF.
```

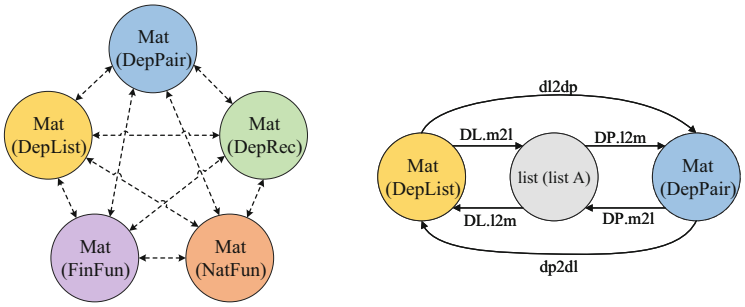
Here, $MatrixAll$ is the collection or entry of all matrix model implementations, DP, DL, DR, NF, and FF are shorthand for specific matrix models. Then, we take the real number field type $FieldR$ as the module parameter and obtain $MatrixR$ after instantiation. For concrete use, the specific name of the instantiated matrix model is also provided. For example, $MatrixR\_DR$ denotes the formal matrix theory, with R as the carrier and $DepRec$ as the model.

Extract OCaml Code. The Formal matrix in Coq is not only used to verify abstract mathematical properties but also to obtain correct functional programs by program extraction. OCaml program extraction is used as an example to discuss the differences between these matrix models during extraction. First, all these schemes extract valid OCaml programs. Secondly, from the perspective of the simplicity of the extracted code, DepList and DepPair have more redundancy, mainly including the encoding of the matrix shape, whereas DepRec has virtually no redundancy because it directly corresponds to the list type. Again, the matrices resulting from NatFun and FinFun are defined as functions. Although the elements of the matrix are not available as a whole, each element can be accessed individually.

3.5 Conversion Between Matrix Models

After building multiple models that can perform the same function and have the same signature, we aim to create a connection between these models. In fact, the pair of conversion functions $l2m$ and $m2l$ and the properties associated with them given in $MatrisSig$ were prepared for this purpose. This pair of functions are available for all five models. These functions can be used to convert between any of the models as indicated in Fig. 3.

Here, (a) displays all conversions between all models; (b) displays the detailed process for a pair of conversions. In the following section, we discuss the conversion between DepList and DepPair as an example. For example, $dl2dp$ is the conversion of DepList to DepPair, which comprises $DL.m2l$ and $DP.l2m$. Furthermore, we can demonstrate that dl2dp and dp2dl are bijective and their composition is an identity function.



(a) All conversions (b) Conversion details

Fig. 3. Conversion between different matrix models.

`Variable` r c: $\mathbb{N}$.
`Definition` $dp2dl$ (m:$DP.mat$ r c) : $DL.mat$ r c := $DL.l2m$ ($DP.m2l$ m).
`Definition` $dl2dp$ (m:$DL.mat$ r c) : $DP.mat$ r c := $DP.l2m$ ($DL.m2l$ m).
`Lemma` $dl2dp\_bij$: bij (@$dl2dp$ r c). `Lemma` $dp2dl\_bij$: bij (@$dp2dl$ r c).
`Lemma` $dl2dp\_dp2dl\_id$: $\forall$ (m: $DP.mat$ r c), $dl2dp$ ($dp2dl$ m) = m.
`Lemma` $dp2dl\_dl2dp\_id$: $\forall$ (m: $DL.mat$ r c), $dp2dl$ ($dl2dp$ m) = m.

Here, the lemma $dl2dp\_bij$ means that $dl2dp$ is bijective, which is proved by the fact that both DP.l2m and DL.m2l are bijective functions. The lemma $dp2dl\_dl2dp\_id$ indicates that the composition of $dp2dl$ and $dl2dp$ yields an identity function, which can be proven by the following steps: 1). Unfolding dl2dp and dp2dl, we obtain DP.l2m (DL.m2l (DL.l2m (DP.m2l m))) = m; 2). Using DL.m2l_l2m_id to remove the middle part, we obtain DP.l2m (DP.m2l m) = m; 3). Completes the proof using DP.l2m_m2l_id.

3.6 Isomorphism of Matrix Models

Certain matrix operations form some algebraic systems under each matrix model, such as matrix addition, scalar multiplication, and square matrix multiplication. The structures of these algebraic systems are similar and we demonstrate the isomorphism between them. Isomorphic algebraic systems preserve certain properties such as associative and distributive laws, as follows:

`Variable` A B C: `Type`. `Variable` fc: $C \to C \to C$.
`Variable` fa ga: $A \to A \to A$. `Variable` fb gb: $B \to B \to B$.
`Definition` $isomor$:= $\exists(\phi$: $A \to B)$, $homo$ fa fb $\phi \wedge bij$ ϕ.
`Definition` $isomor2$:= $\exists(\phi$: $A \to B)$, $homo$ fa fb $\phi \wedge homo$ ga gb $\phi \wedge bij$ ϕ.
`Lemma` $isoComm$: $isomor$ fa $fb \to (comm$ $fa \leftrightarrow comm$ $fb)$.
`Lemma` $isoAssoc$: $isomor$ fa $fb \to (assoc$ $fa \leftrightarrow assoc$ fb.
`Lemma` $iso2DistrL$: $isomor2$ fa ga fb $gb \leftrightarrow distr\_l$ fa $ga \leftrightarrow distr\_l$ fb gb.

Here, $isomor$ denotes the isomorphism between $\langle A, fa\rangle$ and $\langle B, fb\rangle$ and $isomor2$ denotes the isomorphism between $\langle A, fa, ga\rangle$ and $\langle B, fb, gb\rangle$. $isoComm$

indicates that the commutative law of fa and fb are equivalent. Similarly, associativity and left distributive laws are equivalent. We use matrix addition and square multiplication on DL and DP as examples; denoted as $\langle M_{dl}, +_{dl}, \times_{dl} \rangle$ and $\langle M_{dp}, +_{dp}, \times_{dp} \rangle$. Isomorphism theory can be applied to them.

```
Variable r c n : ℕ.
Lemma isoMadd: isomor (@DL.madd r c) (@DP.madd r c).
Lemma isoMmul: isomor (@DL.mmul n n n) (@DP.mmul n n n).
Lemma addComm: comm (@DL.madd r c) ↔ comm (@DP.madd r c).
Lemma addAssoc: assoc (@DL.madd r c) ↔ assoc (@DP.madd r c).
Lemma distrL: distr_l DL.madd DL.mmul ↔ distr_l DP.madd DP.mmul.
```

Here, *isoMadd* denotes that $\langle M_{dl}, +_{dl} \rangle$ and $\langle M_{dp}, +_{dp} \rangle$ are isomorphic; *isoMmul* denotes that $\langle M_{dl}, \times_{dl} \rangle$ and $\langle M_{dp}, \times_{dp} \rangle$ are isomorphic; the remaining lemmas denote that commutative, associative, and distributive laws are separately equivalent on the two structures, which can be proved directly using isomorphism theory. The main task of the first two proofs is the homomorphic mapping between the operations, which is not discussed here due to space limitations.

4 Comparison of Different Matrix Models

Section 2 presents different matrix models with a separate analysis. Basically, the first three are "programming thinking" and the last two are "mathematical thinking". We provide ratings based on our current understanding and practical experience in the process of constructing the integrated library, see Table 1.

Table 1. Comparison of different matrix models.

Models	DepList	DepPair	DepRec	NatFun	FinFun
Maturity	⋆	⋆	⋆⋆	⋆⋆	⋆⋆⋆
Conciseness of the definitions	⋆	⋆	⋆	⋆⋆⋆	⋆⋆⋆
Conciseness of the proofs	⋆	⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆
Conciseness of the extracted OCaml code	⋆	⋆	⋆⋆⋆	⋆⋆	⋆⋆
Simplicity of the syntax or skill	⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆	⋆

Each rating is indicated by an asterisk from 1–3 stars, where a higher number is better. Note that there is a lack of quantifiable evaluation methods for theorem-proving libraries, and our ratings are subjective and one-sided. Nonetheless, these ratings can help developers to make certain choices.

5 Example of Using the Matrix Library

After a unified interface matrix library is built, verification projects using the matrix library are not required to consider the limitations of the underlying model. For example, in the application scenario of coordinate transformation,

it is necessary to prove that the product of the three transposed matrices is equal to the given matrix. We can choose matrix library DL at will, and there is virtually no change in the code when switching to DP, DR, NF, or FF.

```
Import MatrixR_DL. (* _DP/_DR/_FUN/_FF. *)
Infix "×" := mmul.  Notation "m ^T" := (mtrans m).  Variable ψ,θ,ϕ: ℝ.
Definition Rx := mkMat33 1 0 0 0 (cos ϕ) (sin ϕ) 0 (-sin ϕ) (cos ϕ).
Definition Ry := mkMat33 (cos θ) 0 (-sin θ) 0 1 0 (sin θ) 0 (cos θ).
Definition Rz := mkMat33 (cos ψ) (sin ψ) 0 (-sin ψ) (cos ψ) 0 0 0 1.
Lemma Rbe_ok := (Rz)^T× (Ry)^T× (Rx)^T = mkMat33
  (cos θ × cos ψ) (cos ψ × sin θ × sin ϕ - sin ψ × cos ϕ)
  (cos ψ × sin θ × cos ϕ + sin ϕ × sin ψ) (cos θ × sin ψ)
  (sin ψ × sin θ × sin ϕ + cos ψ × cos ϕ) (sin ψ × sin θ × cos ϕ
  - cos ψ × sin ϕ) (-sin θ) (sin ϕ × cos θ) (cos ϕ × cos θ).
```

6 Conclusion

Matrices are important tools in mathematics and computer science, and numerous problems can be modeled and solved using matrix. However, owing to the complexity of matrix theory, typical system designs or programs related to matrices require the use of formal methods, particularly theorem proving, to ensure correctness. We focus on matrix formalization in the Coq theorem prover. We collected five existing matrix models from the Coq community. We found that these models varied widely, and none of them achieved the formalization of large-scale matrix theory. Therefore, developers face difficult choices and high switching costs when they want to switch between different models in later stages.

In this study, we made several contributions to solving this problem. First, a unified matrix interface was created. Subsequently, five models were integrated based on this interface. Then, bijection functions were created, thus establishing the isomorphism between certain matrix operations and providing a new method for proving the properties of these operations. Finally, evaluation ratings were provided for developer reference. In summary, we evaluated matrix models in the Coq community, which can assist developers when making choices at an early stage. Moreover, using the unified interface and implementation we provided significantly reduces the cost of switching models at a later stage.

Future work will be in the following direction. Regarding the hierarchy of matrices, we have explored techniques such as type classes, canonical structures, and modules; however, these have not yet been investigated sufficiently. We hope to build a more detailed hierarchy in the future. In addition, special tactic is defined with Ltac in matrix models; however, there is no unified design to date; this will be unified in the future. Furthermore, and most importantly, there is a large amount of matrix theory that has not yet been implemented in any matrix model; this will be our main task for future work.

Acknowledgments. I would like to thank ZhenWei Ma and YingYing Ma for their research on matrix formalization techniques, and my colleagues for their discussions and suggestions.

References

1. Zhang, X.D.: Matrix Analysis and Applications, 2nd edn. Tsinghua University Press, Beijing (2013)
2. Wang, J., Zhan, N.J., Feng, X.Y., Liu, Z.M.: Overview of formal methods. Ruan Jian Xue Bao/J. Softw. **30**(1), 33–61 (2019). (in Chinese with English abstract). https://doi.org/10.13328/j.cnki.jos.005652
3. Fisher, K., Launchbury, J., Richards, R.: The HACMS program: using formal methods to eliminate exploitable bugs. Philos. Trans. R. Soc. A, **375**, 20150401 (2017). https://doi.org/10.1098/rsta.2015.0401
4. Chen, G., Shi, Z.P.: Formalized engineering mathematics. Commun. CCF **13**(10) (2017). (in Chinese with English abstract)
5. Coq Development Team. The Coq Reference Manual 8.13.2. INRIA (2019)
6. Isabelle proof assistant. https://isabelle.in.tum.de
7. LEAN Theorem Prover: Microsoft Research. https://leanprover.github.io
8. The HOL Interactive Theorem Prover. https://hol-theorem-prover.org
9. Ma, Z.W., Chen, G.: Matrix formalization based on Coq record. Comput. Sci. **46**(7), 139–145 (2019). (in Chinese with English abstract). https://doi.org/10.11896/j.issn.1002-137X.2019.07.022
10. Ma. Y.Y., Ma, Z.W., Chen. G.: Formalization of operations of block matrix based on Coq. Ruan Jian Xue Bao/J. Softw. **32**(6), 1882–1909 (2021) (in Chinese with English abstract). https://doi.org/10.13328/j.cnki.jos.006255
11. Mathematical Components. https://math-comp.github.io
12. Boldo, S., Lelay, C., Melquiond, C.: Coquelicot (2015). https://coquelicot.saclay.inria.fr/
13. Magaud, N.: Programming with Dependent Types in Coq: a Study of Square Matrices (2004). https://hal.inria.fr/hal-00955444
14. Blanqui, F., Koprowski, A.: CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Math. Struct. Comp. Sci. **21**(4), 827–859 (2011). https://doi.org/10.1017/S0960129511000120
15. Hietala, K., Rand, R., Hung, S.-H., Xiaodi, W., Hicks, M.: A verified optimizer for quantum circuits. In: ACM SIGPLAN Symposium on Principles of Programming Languages (POPL (2021)
16. Rand, R., Quantum, V.: Computing. Software Foundations Inspired Volume Q. https://www.cs.umd.edu/~rrand/vqc/index.html
17. Geuvers, H., et al.: A constructive algebraic hierarchy in Coq. J. Symbol. Comput. **34**(4), 271–286 (2002)
18. Casteran, P., Sozeau, M.: A Gentle introduction to type classes and relations in Coq (2016)
19. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_23
20. Jung, R.: Exponential blowup when using unbundled typeclasses to model algebraic hierarchies (2019). https://www.ralfj.de/blog/2019/05/15/typeclasses-exponential-blowup.html. Accessed 1 Feb 2022

21. Baanen, A.: Use and abuse of instance parameters in the lean mathematical library, 2 May 2022. arXiv, https://doi.org/10.48550/arXiv.2202.01629
22. Mahboubi, A., Tassi, E.: Canonical structures for the working Coq user (2013). https://hal.inria.fr/hal-00816703v1