Programming with Dependent Types in Coq: a Study of Square Matrices

Nicolas Magaud

Programming Languages and Compilers
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia

Abstract. The Coq proof system allows users to write (functional) programs and to reason about them in a formal way. We study how to program using dependently typed data structures in such a setting. Using dependently typed data structures enables programmers to have more precise specifications for their programs before starting proving any significant properties (e.g. total correctness) about these programs.
We particularly focus on an *operational* description of square matrices and their operations. Matrices are represented using dependent types. A matrix is a vector of rows, which are themselves vectors indexed by natural numbers. We also take advantage of Coq modules system to have matrices parametrised by a carrier set. Finally, this *operational* description of square matrices can be extracted into a mainstream functional programming language like Ocaml.

1 Introduction

Dependent types allow programmers to provide more precise specifications for their programs. Several programming languages [2, 17] have been developed in the past few years to add some (restricted) level of dependency to types in functional programming languages. More recently, McBride and McKinna proposed a new framework [12, 6] to develop dependently typed programs more easily. On the other hand, proof systems such as Coq [5, 3] can be viewed as dependently typed languages rather than theorem provers. The Coq system itself is based on the Calculus of Inductive Constructions [13], a λ-calculus with dependent types and capabilities for inductive definitions. It therefore can be used as a programming language equipped with a very expressive type system.

In this paper, we show, via the example of operations on square matrices and their properties, that one can actually program using dependent types within the Coq system. However, programming with dependent types within a type-theoretic framework like Coq [5] remains a challenging activity. In particular defining functions handling dependently typed structures can be awkward.

1.1 Programming with Dependent Types

In order to prove a program correct within the Coq system, one can use two rather different approaches. The first one consists in having two distinct steps

to write a program and to prove properties about it. In this case, we first define a function as we would do in ML or Haskell (i.e. no dependent types involved yet) and then prove some statements about its properties. The other technique consists in using dependent types to embed the full specification of a function into its type.
Let us illustrate this with the definition of a predecessor function pred for Peano's integers nat. In Coq, Peano's integers are defined as an inductive type with two constructors:

```
Inductive nat:Set := O:nat | S:nat->nat.
```

We present three distinct types for the pred function. We also add some statements we need to prove to make sure the definition of pred is correct.

- We can define pred exactly the way it would be defined in ML or Haskell, not using any dependent types to write its type: $\mathsf{pred} : \mathsf{nat} \rightarrow \mathsf{nat}$. Then we need to prove the following statement to make sure we actually defined the predecessor function.

$$\forall n : \mathsf{nat}, n = 0 \vee n = (\mathsf{S}\ (\mathsf{pred}\ n))$$

- We can define it as a fully-specified function:

$$\mathsf{pred} : \forall n : nat, \{n = 0\} + \{p : \mathsf{nat}\ \mid\ n = (S\ p)\}.$$

  In this case, the type of pred is actually a full specification of pred. It states that, given an n, the function either returns a proof that n is equal to 0, or returns a p which verifies the property $n = (\mathsf{S}\ p)$ i.e. a p which is the predecessor of n. As the type of pred contains its full specification, there is no need to make any additional proofs.
- Dependent types allow us to be flexible. Even if we do not want to write the full specification of a function in its type, we do not have to stick to types available in Haskell or ML. Instead, we can actually use dependent types to refine the type of the function, without including its whole specification. For instance, we can choose to write a predecessor function whose type is:

$$\mathsf{pred} : \forall n : nat, n \neq 0 \rightarrow nat.$$

  This specification rules out the case of 0, however it does not provide any proof that it computes the predecessor of n. We still need to prove

$$\forall n : \mathsf{nat}, \forall H : n \neq 0, n = (\mathsf{S}\ (\mathsf{pred}\ n\ H))$$

  on the side.

1.2 Vectors and Matrices

In this paper, we study how to actually write programs using dependently typed data structures within the Coq system. We are also interested in how to carry out proofs about dependently typed functions.

We focus on matrices (represented as vectors of vectors) which look like the next step into dependently typed programming after vectors. Our main goal is to build an *operational* formalisation of square matrices, i.e. to write executable functions handling square matrices and then prove some properties about these functions. Indeed, after proving lots of properties about matrices and their operations in Coq, we want to be able to run these *then certified* programs, either within the Coq toplevel or in regular functional programming languages such as ML or Haskell (after extracting them from Coq). Other people have been working on formalising algebra from a mathematical point of view. L. Pottier proposed a contribution Algebra [14] to the Coq system to deal with algebraic notions from a mathematical point of view. Subsequent developments, including describing matrices and their properties (Linear Algebra) have been carried out by J. Stein [15] recently. These formalisations only consider the mathematical properties of the notions it deals with, without worrying about how to compute on these notions.

Lots of programs, including large libraries for parallel computations such as LAPACK [7], deal with programming with matrices and their various, as efficient as possible (sometimes designed on purpose for a single algorithm), representations. In this paper, we really focus on the practicability of programming matrices operations using dependent types in Coq rather than efficiency issues. The numerous existing representations for matrices are one of the reasons why we got interested in programming with matrices in the Coq system using matrices. It actually looks like a promising experimentation field for our work about changing data representation in type theory [11, 9]. Indeed, matrices actually have numerous implementations as concrete datatypes, depending on which language they are implemented with, which applications are targeted, etc.

1.3 Outline

In section 2, we introduce the (dependent) datatypes we choose to represent vectors and matrices. In section 3, we define functions on these data types and highlight the technicalities arising from using dependent types. In section 4, we show how to make the implementation of matrices and their operations independent of the carrier set by using modules. In section 5, we survey the proofs of some properties of square matrices, focusing on some useful proof techniques. In section 6, we extract our development of square matrices from Coq to Ocaml. Finally, we conclude with a discussion on the practicability of programming with dependent types in Coq, especially in the case of matrices.

2 A Dependently Typed Data Structure for Matrices

In this section, we define vectors and matrices as inductive data types and show some basic properties about them. We start with vectors and some of their properties. Then from vectors, we define matrices as vectors of vectors.

2.1 Vectors and their Basic Properties

Vectors are described by this dependent inductive data type:

```
Inductive vect (A : Set) : nat -> Set :=
  | vnil : vect A 0
  | vcons : forall n : nat, A -> vect A n -> vect A (S n).
```

This definition is parametrised by a set A. Vectors are defined in the same way lists are; with a constructor for the empty vector (vnil) and a constructor to add an element in front of a vector (vcons). The new feature is that vectors are indexed with their length, represented by a natural number. Type information about vectors may allow to know whether they are empty or not. In particular, it is clear that an element of type (vect A 0) is necessarily the empty vector vnil. Given an element u of type (vect A (S n)) for some n, it can be decomposed into a head v and a tail vs of length n such that $u =$ (vcons n v vs). However, these two interesting properties are not derived automatically from the definition. They have to be proven as equations by the user. Performing such proofs require using a dependent notion of equality.

Dependent Equality There is no primitive notion of equality in Coq. Leibniz equality is defined as a polymorphic inductive predicate:

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  refl_equal : x = x.
```

When we deal with dependent types, we sometimes need to talk about equality between two objects inhabiting two different instances of a type family, e.g. v : (vect t_1) and w : (vect t_2) before actually figuring out that t_1 and t_2 are actually definitionally equal (i.e. convertible according to Coq reduction rules), and therefore before knowing (vect t_1) and (vect t_2) denote the same type. For this purpose, Leibniz equality is too restrictive. However, Coq provides a notion of dependent equality eq_dep we can use instead.

```
Inductive eq_dep (U : Type) (P : U -> Type) (p : U) (x : P p) :
    forall q : U, P q -> Prop :=  eq_dep_intro : eq_dep U P p x p x
```

Such an equality is useful to establish properties relating the length of vectors with their shape. For instance, as stated in (1), a vector of length 0 will necessarily be vnil. Proving the two following properties (note these statements feature Leibniz equality)

$$\forall v : (\mathsf{vect}\ 0).\ v = \mathsf{vnil} \tag{1}$$

$$\forall n : \mathsf{nat};\ v : (\mathsf{vect}\ (\mathsf{S}\ n)).\ \exists n : \mathsf{nat}; v' : (\mathsf{vect}\ n).\ v = (\mathsf{vcons}\ n\ a\ v') \tag{2}$$

requires using a dependent equality (here eq_dep) as an intermediate step in the proof. In the end, we can prove these equalities only because data involved on both sides on the equality actually lives in the exactly same type (up to convertibility).

2.2 Matrices on top of Vectors

Matrices are defined as vectors of vectors:

$$\mathsf{matrix} := \lambda A : \mathsf{Set}.\ \lambda n : \mathsf{nat}.\ \lambda m : \mathsf{nat}.\ (\mathsf{vect}\ (\mathsf{vect}\ A\ n)\ m).$$

By convention, we consider matrices are defined by rows. This means an element of (matrix n m) is a matrix of m rows and n columns. From an implementation point of view, it means it is a vector of m rows, each row being itself a vector of length n.

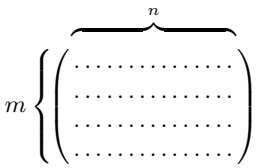

Fig. 1. Representation of matrices as vectors of rows

Obviously, the data structure we choose to represent matrices is polarised. Lines and columns play two very different roles: extracting a line simply consists in searching this line in the outermost vector structure of matrices. However getting a column is a bit more tedious, it requires picking up the right element in each row and combining all these elements into a new vector of length m.

Operations such as extracting a row or a column require preconditions to be checked in order to make sure the line (resp. column) is actually within the bounds of the matrix. For instance, the type of the function getcolumn features some preconditions:

$$\mathsf{getcolumn} : (\forall n, m : \mathsf{nat}.\ (\mathsf{matrix}\ n\ m) \to \forall i : \mathsf{nat}.\ 0 < i \to i \leq n \to (\mathsf{vect}\ m)$$

As we said before, we are interested in a formalisation of square matrices. Square matrices are actually elements of type (matrix n m) where n and m are the same. So why did we not define matrices as (vect (vect A n) n) ? The main reason for this is that the structure of matrices is not symmetric. We want to compute on matrices row by row (this would be easy to achieve by structural recursion on the outermost vector structure), but with a single index, case analysis on the number of rows will affect the number of columns and vice-versa leading to a lot of trouble. This illustrates that while dependent types allow to have more precise types for data and programs, we should be careful not to over-specify these objects, otherwise we will not be able to handle them at all.

Once we defined dependently typed data structures for vectors and matrices and some of their basic properties, we need to write functions dealing with these data.

3 Operations on Vectors and Matrices

Operations on vectors and matrices are defined either by structural recursion and case analysis on the structure of vectors or by structural recursion of the length of one of the vectors involved in the computation. In most cases, these functions are defined using *interactive programming*. It consists in stating the type of the function we want to define as a goal and taking advantage of the interactive proof system to build the function step by step keeping track of the expected type at every single stage. This technique is particularly useful in presence of dependent pattern matching which is usually very difficult to get correct in one go. In the rest of this section, we first define operations on vectors and then reuse them to write functions on matrices.

3.1 Operations on Vectors

Let us consider a single operation first: computing the opposite of a vector.

```
Fixpoint oppvect (n : nat) (v : vect A n) {struct v} :
 vect A n :=
  match v in (vect _ w) return (vect A w) with
  | vnil => vnil A
  | vcons p v vs => vcons A p (Aopp v) (oppvect p vs)
  end.
```

oppvect proceeds by structural recursion on v. Once the above definition is accepted by the Coq system, two new computational rules are added, namely:

$$\mathsf{oppvect}\ 0\ (\mathsf{vnil}\ A) \xrightarrow{\iota} (\mathsf{vnil}\ A)$$
$$\mathsf{oppvect}\ (\mathsf{S}\ n)\ (\mathsf{vcons}\ A\ n\ v\ vs) \xrightarrow{\iota} \mathsf{vcons}\ A\ n\ (\mathsf{Aopp}\ v)\ (\mathsf{oppvect}\ n\ vs)$$

In this example, all goes well because case analysis on the vector v updates the index n representing its length.

However, if we want to write a function to add two vectors, things get a bit more technical. We want the addition function to have the following type:

$$\forall n : \mathsf{nat}.\ (\mathsf{vect}\ A\ n) \to (\mathsf{vect}\ A\ n) \to (\mathsf{vect}\ A\ n)$$

It means we only add vectors of the same length. The code for addvect shown in figure 2. It proceeds by structural recursion on the first vector, say v and then perform two case analysis in a row (on v, and then on the other vector v'). Case analysis on v yields a pattern looking like (vcons $n1$ $x1$ $v1$) whereas case analysis on v' yields (vcons $n2$ $x2$ $v2$). Then we would like to recursively apply addvect with arguments $v1$ and $v2$. The trouble is $v1$ has type (vect $n1$) and $v2$ has type (vect $n2$). Somehow we lost track of the fact $n1$ and $n2$ are actually equal. To be able to apply addvect, we rewrite $v1$ of length $n1$ into a vector (the same one actually) of length $n2$. To do so, we need to know $n1$ and $n2$ are the same. This

is achieved by adding an equation $k = (\mathsf{S}\ n1)$ to the pattern matching structure `match v' in (vect _ k) return (k = S n1 -> vect A k) with`. Inside the second branch, k will expand to $(\mathsf{S}\ n2)$ and from that, we will get back the link between $n1$ and $n2$ using the theorem `eq_add_S_tr`.

```
Fixpoint addvect (n : nat) (v : vect A n) {struct v} :
 vect A n -> vect A n :=
  match v in (vect _ k) return (vect A k -> vect A k) with
  | vnil => fun v' => vnil A
  | vcons n1 x1 v1 =>
      fun v' : vect A (S n1) =>
      match v' in (vect _ k) return (k = S n1 -> vect A k) with
      | vnil => fun h => vnil A
      | vcons n2 x2 v2 =>
          fun h =>
          vcons A n2 (Aplus x1 x2)
            (addvect n2
               (eq_rec n1 (fun n : nat => vect A n) v1 n2
                  (eq_add_S_tr n1 n2 (sym_eq h))) v2)
      end (refl_equal (S n1))
  end.

with eq_add_S_tr : forall (n m : nat), S n = S m -> n = m.
```

Fig. 2. Implementation in Coq of addvect

As a general rule, functions whose computations are structurally recursive can be defined directly, as shown in figure 2. However, it is usually far more easy to define them by *interactive programming*, and then retrieve the actual term built in the process. Accordingly, it would only be practicable to write functions defined by structural recursion on Peano's number in this way, especially, if one need to retrieve the shape of a vector from the shape of the index using one of the two lemmas (1) and (2) introduced before.

3.2 Operations on Matrices

Once we have defined all operations on vectors, we can go a step further and define functions on matrices. Addition on matrices is straightforward and follows a similar pattern to addvect. We also define a function computing opposites of matrices and implement neutral elements $0_{n,m}$ and I_n. Defining matrices product is the most technical part of the formalisation. We had to build several auxiliary functions.

– We start from the product of vectors:

$$\mathsf{scalprod} : \ \forall n : \mathsf{nat}.\ (\mathsf{vect}\ n) \to (\mathsf{vect}\ n) \to A$$

This function is defined in exactly the same way as addvect was defined in the previous section.
The next two steps aim at defining a vector-matrix product.

- prodvectmatrix computes the partial product (actually only the c last elements of the product) of a vector of length n and a matrix with n rows and m columns.

$$\begin{aligned}\mathsf{prodvectmatrix} : \forall n, m : \mathsf{nat}.\ \forall v : (\mathsf{vect}\ n).\ \forall w : (\mathsf{matrix}\ m\ n).\\ \forall c : \mathsf{nat}.\ O \leq c \rightarrow c \leq m \rightarrow (\mathsf{vect}\ c).\end{aligned}$$

  It is defined by structural recursion on n, rather than by structural recursion on a vector or a matrix.
- prodvectmat is the function computing the product of a vector and a matrix. It corresponds to the difficult bit in the definition of product. The main reason is that we need to get each column of the matrix to multiply it with the vector, and as we have seen before, the data structure for matrices is not well-suited for extract columns of a matrix.

$$\mathsf{prodvectmat} :\ \forall n, m : \mathsf{nat}.\ \forall v : (\mathsf{vect}\ n).\ \forall w : (\mathsf{matrix}\ m\ n).\ (\mathsf{vect}\ m)$$

  From an implementation point of view, this function simply calls prodvectmatrix with $c := m$ and proofs that $0 \leq m$ and $m \leq m$. All the tricky code is actually hidden in prodvectmatrix.
- Finally, we define the product of two matrices by structural recursion on the first matrix. Each time we get a new row of the first matrix, we use prodvectmat to compute a new row of the output matrix.

$$\mathsf{prodmat} :\ \forall n, m, p : \mathsf{nat}.\ (\mathsf{matrix}\ m\ n) \rightarrow (\mathsf{matrix}\ p\ m) \rightarrow (\mathsf{matrix}\ p\ n)$$

At this point, we have defined data structures for matrices as well as operations on them. However, we postponed the description of the modules system we used to write our formal development.

4 Introducing Modules and Functors

To introduce some modularity and abstraction in our development, we would like to use Coq modules [4]. Coq modules will allow us to describe in a simple manner which features we expect from an implementation of matrices.

We start by specifying what signature a carrier set should have. Then we provide the signatures we want for vectors and matrices (see figure 3). They will be functors parametrised by the carrier set.

4.1 Specification of a Carrier Set

A carrier set can be specified using the following module declaration:

```
Module Type Carrier.
Parameters A :  Set.
Parameters Aopp :  A ->  A.
Parameters (Aplus : A -> A ->  A) (Amult : A -> A ->  A).
Parameters (A0 : A) (A1 : A).
Parameters Aeq :  A -> A ->  bool.

Axiom A_ring :  Ring_Theory Aplus Amult A1 A0 Aopp Aeq.

Add Abstract Ring A Aplus Amult A1 A0 Aopp Aeq A_ring.

End Carrier.
```

A carrier set consists of a set A, equipped with an opposite function Aopp, addition Aplus, product Amult, two distinct neutral elements for addition (A0) and multiplication (A1) and a procedure to decide equality Aeq. The set A, together with these operations, must form a ring structure (all the required properties are summed up in A_ring, see [5, Chap. 19] for a definition of Ring_Theory).

As an example, we build the carrier set based on integers $\mathbb{Z}$:

```
Module Zc : Carrier.

Definition A := Z.            Definition Aopp := Zopp.
Definition Aplus := Zplus.    Definition Amult := Zmult.
Definition A0 := 0%Z.         Definition A1 := 1%Z.
Definition Aeq := Zeq.

Definition A_ring := ZTheory.

End Zc.
```

4.2 A Signature for Matrices

We omit the module type declaration for Vectors and directly present the module type declaration for Matrices. This module type (see figure 3) lists the basic operations on matrices as well as all the required properties to ensure that matrices equipped with addition and product form a ring structure.

Once we would have provided proofs for all the required properties, we would be able to instantiate our module with the carrier set Zc for instance.

```
Module matrixZ := Matrices Zc.
```

Eventually, we also provide a module declaration for square matrices, which basically boils down to the same as Matrices with all indexes equal.

```
Module Type TMatrices.

Parameter A : Set.
Parameter Aopp : A -> A.
Parameters (Aplus : A -> A -> A) (Amult : A -> A -> A).
Parameters (A0 : A) (A1 : A).
Parameter Aeq : A -> A -> bool.
Parameter A_ring : Ring_Theory Aplus Amult A1 A0 Aopp Aeq.

Parameter matrix : nat -> nat -> Set.
Parameter addmatrix : forall n m : nat, matrix n m -> matrix n m -> matrix n m.
Parameter prodmat : forall n m p : nat, matrix m n -> matrix p m -> matrix p n.
Parameter oppmatrix : forall n m : nat, matrix n m -> matrix n m.
Parameter o : forall n m : nat, matrix n m.
Parameter I : forall n : nat, matrix n n.

Axiom addmatrix_sym :
    forall (n m : nat) (w w' : matrix n m), addmatrix n m w w' = addmatrix n m w' w.

Axiom addmatrix_assoc :
    forall (n m : nat) (w w' w'' : matrix n m),
    addmatrix n m (addmatrix n m w w') w'' = addmatrix n m w (addmatrix n m w' w'').

Axiomaddmatrix_oppmatrix :
    forall (n m : nat) (w : matrix n m), addmatrix n m w (oppmatrix n m w) = o n m.

Axiom addmatrix_zero_l :
    forall (n m : nat) (w : matrix n m), addmatrix n m (o n m) w = w.

Axiom I_mat : forall (m n : nat) (w : matrix n m), prodmat m m n (I m) w = w.

Axiom mat_I : forall (m n : nat) (w : matrix n m), prodmat m n n w (I n) = w.

Axiom prodmat_distr_l :
    forall (n m p : nat) (a b : matrix m n) (w : matrix p m),
    prodmat n m p (addmatrix m n a b) w =
    addmatrix p n (prodmat n m p a w) (prodmat n m p b w).

Axiom prodmat_distr_r :
    forall (n m p : nat) (a b : matrix p m) (w : matrix m n),
    prodmat n m p w (addmatrix p m a b) =
    addmatrix p n (prodmat n m p w a) (prodmat n m p w b).

Axiom prodmat_assoc :
    forall (n m p q : nat) (a : matrix m n) (b : matrix p m) (c : matrix q p),
    prodmat n p q (prodmat n m p a b) c = prodmat n m q a (prodmat m p q b c).

End TMatrices.
```

Fig. 3. Interface of the module Matrices in Coq

4.3 Modules and Reduction Behaviour

Our modules implementations have been declared as transparent using the $<:$ notation. It means scope of reduction rules for functions defined in a module A extends to a module B into which the module A is loaded. It was actually more convenient to do it this way rather than stating each reduction rule we may need as a propositional equality in the signature of the module.

5 Proofs and Proof Techniques

We do not go through all the lemmas we proved to complete this formalisation. Instead we refer the interested reader to the on-line description of the development:

`http://www.cse.unsw.edu.au/~nmagaud/Coq/Matrices/index.html`

We rather focus of some proofs patterns we faced and present some techniques designed to make proof development go smoothly.

5.1 Effective Induction Principles for Vectors

In order to make it easier to reason about vectors, we would like to have an induction principle taking into account that one can only add vectors of the same length. As an example, let us consider the following statement:

$$\forall n : \mathsf{nat}; v, w : (\mathsf{vect}\ A\ n).\ (\mathsf{addvect}\ v\ w) = (\mathsf{addvect}\ w\ v).$$

To prove it, we can proceed by double induction on v and then on w to decompose these two vectors and then rule out the absurd cases. It is really convenient to factorise these steps into a new induction principle:

$$\begin{array}{l}\forall P : \forall n : \mathsf{nat}.\ (\mathsf{vect}\ A\ n) \to (\mathsf{vect}\ A\ n) \to \mathsf{Prop}.\\ (P\ \mathsf{O}\ (\mathsf{vnil}\ A)\ (\mathsf{vnil}\ A)) \to\\ \left(\begin{array}{l}\forall n : \mathsf{nat}; v, v' : (\mathsf{vect}\ A\ n).\ (P\ n\ v\ v') \to\\ \qquad \forall a, b : A.\ (P\ (\mathsf{S}\ n)\ (\mathsf{vcons}\ A\ n\ a\ v)\ (\mathsf{vcons}\ A\ n\ b\ v'))\end{array}\right) \to\\ \forall n : \mathsf{nat}; v, v' : (\mathsf{vect}\ A\ n).\ (P\ n\ v\ v')\end{array}$$

It makes proofs more readable, with avoiding the clutter generated by the straightforward method. The above-mentioned induction principle can be easily proven by induction on the length on the vectors n; it allows requires to use the two basic equalities (1) and (2) on vectors. As proofs do not only involve two vectors, we have similar induction principles for three vectors (useful for associativity proofs), and for one single vector (for proofs related to opposite and neutral elements).

5.2 Proof Irrelevance

Some of the functions we defined are partial and therefore have one (or more) precondition(s) as arguments. As a result, computations and lemmas statements rely on proof terms. This can be annoying as one proof of a given property A cannot be replaced by another proof of the same property A. To avoid such inconvenience, we decide to add as an axiom the principle of proof-irrelevance

$$\forall A : \mathsf{Prop}. \forall p, q : A. p == q$$

to make two proofs of the same statement A coincide whether it is required.

Anyway, to avoid trouble related to these matters, we should always universally quantify over proofs of statements needed for partial functions in their definitions. In that way, we only pass variables (denoting proofs) around. Anyway, it is not always possible, especially because functions might generated proofs of weaker properties to pass onto the functions used in their body.

5.3 Convenient Notations and Structured Editing

The initial version of this work has been developed using a graphical interface PCoq [1] for the Coq proof assistant. It provides some graphical two-dimensional notations and support for structured editing of formula and tactics sequence.

Notations PCoq provides user-friendly notations for addition and product of vectors and matrices. It also provides some notations for the identity matrix I_n. Altogether, it allows us to build a more readable development.

```
Lemma mat_I: ∀m, n : nat. ∀w : (Lmatrix n m). w ⊗ I_n = w.
- Intros m n w;Try Assumption.
- Replace w with (last (vect A n) m w m (le_O_n m) (le_n m)).
- Apply mat_I_last.
- Apply last_n'.
Qed.
```

Fig. 4. A snapshot from the PCoq interface

Structured Editing Structured editing was really useful to write dependent types of functions and statements of theorems. It was a quick way to shuffle arguments around to get the right order for the preconditions; especially when proving an equality between two applications of functions whose preconditions are almost but not the same (e.g. $0 < n$ and $0 \leq n$).

Overall, using a nice interface like PCoq saved us quite a lot of time while developing the formalisation.

5.4 Some Remarks about the Development

Altogether this formal description of square matrices is about 2000 line long. It took us about one month to get it up and running. Programming with dependent types was a bit tricky at first, but then it went on quite comfortably. Proofs took a bit longer, and once we got the right lemmas it was rather practicable. This shows that though it remains really technical, programming with dependent types with objects such as matrices can be achieved in a reasonable amount of time.

6 Extraction

Using the extraction mechanism [8] provided by Coq, we can extract the code we wrote in Coq into ML code. Basically, extraction will discard the proofs, and only keep the computational content of functions. In addition, Coq modules will be translated into OCaml modules. It eventually yields a certified implementation of matrices and their operations in Ocaml.

However extraction will not exactly yield the programs we would expect. Indeed the extracted code for a function like addvect still features an index n whereas this index is now useless. At the time the data structure vect was extracted to Ocaml, the index n actually stopped being related to the length of the vector argument of vcons. As a consequence, we would like to get rid of it during the extraction process.

```
type 'a vect =
  | Coq_vnil
  | Coq_vcons of nat * 'a * 'a vect

let rec addvect n v x : nat -> coq_A vect -> coq_A vect -> coq_A vect =
  match v with
    | Coq_vnil -> Coq_vnil
    | Coq_vcons (n1, x1, v1) ->
        (match x with
           | Coq_vnil -> Coq_vnil
           | Coq_vcons (n2, x2, v2) -> Coq_vcons (n2,
               (C.coq_Aplus x1 x2), (addvect n2 v1 v2)))
```

Fig. 5. After extraction, indexes like n are useless variables.

Indexes remain in the extracted code, because they live in nat, a data type which is defined on the computational side of Coq rather than the logic side. Our idea to solve this issue was to write a dual for nat on the logical side of Coq, namely defining a new datatype Pnat:

```
Inductive Pnat: Prop := P0: Pnat | PS : Pnat -> Pnat.
```

This would be a new version of Peano's numbers, erasable at extraction time. Unfortunately, it can not be as easy! In the course of the formal development, we need a proof of disjointness of constructors for Pnat:

$$\forall n : \mathsf{Pnat}, \neg PO = (PS\ n). \tag{3}$$

Because of restrictions to allowed eliminations of objects of sort Prop (i.e. objects on the logic side), this statement is not provable within the Coq system. In addition, in the current state of our formalisation, it can not be added as an axiom. Indeed we already use the proof-irrelevance principle as a axiom. And from proof irrelevance and (3), one can derive False.

An alternative solution is to index vectors with finite sets $\{1 \dots n\}$ instead of Peano's integers. In this case, we would avoid using preconditions to ensure an index is actually within the bounds. Consequently, we would not need the proof irrelevance principle and therefore be able to state as an axiom that elements in the finite set are distinct even if this finite set is defined in the sort Prop.

7 Discussion

In this paper, we presented a *operational* formalisation of square matrices in Coq. This formalisation is part of the contributions to the Coq system [10]. It shows that one can reasonably program using dependently typed data structures within the Coq proof system. This formalisation is modular and allows having matrices on $\mathbb{Z}$, $\mathbb{R}$, etc. at no extra cost. We proved all the properties required for the set of square (n, n) matrices to be a ring. Note that this ring is non commutative, therefore there is no opportunity for using *Add Ring* to deal with equations on matrices. However it would be interesting to have some sort of *Ring* tactic available for non-commutative rings.

There are numerous work what can be built on top of this formalisation of matrices; we can formalise determinants, write programs to solve systems of linear equations and prove them correct.

We choose a representation of matrices which favours lines to columns. This kind of representation is close to what happens in C where two-dimensional arrays (like matrices) are stored row by row. However, in Fortran, two-dimensional arrays are stored column by column making it easier to access a column that a line (for a line one has to cross the whole matrix to retrieve the components). Other representations such as block matrices, scarce matrices are widely used. It would be interested to experiment how tools [11, 9] we already produced to make changing representation of datatypes easier in Coq can be adapted to be used with dependently typed data structures.

References

1. A. Amerkad, Y. Bertot, L. Pottier, and L. Rideau. Mathematics and Proof Presentation in Pcoq. In *Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*, 2001. Sienna, Italy, also available as INRIA RR-4313.

2. L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
3. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.
4. J. Chrząszcz. Implementing Modules in the Coq System. In *TPHOLs'2003, Roma, Italy*, volume 2758, pages 271–286. LNCS, Springer-Verlag, 2003.
5. Coq development team, INRIA and LRI. *The Coq Proof Assistant Reference Manual*, Apr 2004. Version 8.0.
6. Epigram Team. *Epigram*, 2004. http://www.dur.ac.uk/CARG/epigram/.
7. LAPACK. LAPACK – Linear Algebra PACKage, 1999. http://www.netlib.org/lapack/.
8. Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
9. N. Magaud. Changing Data Representation within the Coq System. In *TPHOLs'2003, Roma, Italy*, volume 2758, pages 87–102. LNCS, Springer-Verlag, 2003.
10. N. Magaud. Ring Properties for Square Matrices, 2003. Contribution to the Coq system: http://coq.inria.fr/contribs-eng.html.
11. N. Magaud and Y. Bertot. Changing Data Structures in Type Theory:A Study of Natural Numbers. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *International Workshop on Types for Proofs and Programs (TYPES'2000)*, volume 2277 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 2000.
12. C. McBride and J. McKinna. The View from the Left. *Journal of Functional Programming*, 14:1–43, 2004.
13. Christine Paulin-Mohring. Inductive definitions in the system coq: Rules and properties. In Mark Bezem and Jan-Friso Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 328–345. Springer-Verlag, March 1993.
14. L. Pottier. Basics notions of algebra., 1999. Contribution to the Coq system: http://coq.inria.fr/contribs-eng.html.
15. J. Stein. Linear algebra, 2003. Contribution to the Coq system: http://coq.inria.fr/contribs-eng.html.
16. J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2003.
17. H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL'99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 214–227, New York, NY, 1999.