Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 140 (2005) 93–100

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

About Constructive vectors

Jean Duprat

Equipe Plume, Laboratoire Lip, ENS de Lyon, CNRS, INRIA

Abstract

We explore several ways of constructively implementing vectors in proof assistants and we discuss their advantages and their drawbacks.

Keywords: Proof assistant, computational geometry

1 Introduction

In computer science vectors are basic structures, usually implemented by arrays [2], whereas in proof assistants, their dimension and the type of their components are structural informations that have to be explicitly written. In section 2, we explore several ways to implement vectors constructively and we discuss their advantages and their drawbacks. In section 3, we present a Coq implementation that uses dependent types. Section 4 shows a problem related to equality and proposes a way to solve it. Section 5 is the conclusion.

2 Vectors

In computer science, vectors are usually viewed as arrays of terms of the same type, with no structure. In that framework, if A is a set and n an integer, a vector V is an element of the Cartesian product $A^n = A \times A \times ... \times A$. A natural tool which we want to make easy to use is the ability to map on each element of a n-vector a unary function or a binary operation on A.

[1] email:Jean.Duprat@ens-lyon.fr

doi:10.1016/j.entcs.2005.06.021

2.1 Cartesian product

Let us consider first a natural implementation of Cartesian products of a set using pair constructors. Starting form A, we define iteratively A^2 as $A \times A$ in which two-dimension vectors are $\{(x, y) \mid x \in A \wedge y \in A\}$; A^3 as $A^2 \times A$ in which three-dimensional vectors are $((x, y), z))$ and so on. Such a method is inefficient because it requires building all the products A^p from $p = 1$ to $p = n - 1$ before being able to build A^n.

An alternative solution is to use a static structure like a record. Then, for example, an element of A^4 is a record (x_1, x_2, x_3, x_4) of four elements of A. In computer science, this can be a good solution, for example a byte is usually implemented as a record $(b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0)$ of bits. But, in general, the main drawback of this approach is its lack of genericity.

2.2 The map function

Suppose given a function $f : A \rightarrow A$. Mapping f on A^n means defining a functional, i.e., a function of function, *map* such that *map* $f : A^n \rightarrow A^n$ applies f on each element of a vector in A^n, for instance *map* $f\ (x_1, x_2, x_3, x_4) = (f\ x_1, f\ x_2, f\ x_3, f\ x_4)$. Because of their non recursive nature, a generic *map* function cannot be defined on records. Instead one has to build a specific function $map^n : (A \rightarrow A) \rightarrow A^n \rightarrow A^n$ for each particular value of n we are interested in.

2.3 Lists

The above objection about the Cartesian product leads us to consider the list construction [5]. Mapping a function $f : A \rightarrow A$ on a list of elements of A is defined whatever the length of the list is.

Consider now A^n as defined as the set $\{l \in (list\ A) \mid length(l) = n\}$. If *map* is easy to perform on functions, it is not on binary operations. Assume given an operation on A written $+$. Suppose we want to induce a $+_n$ defined by applying $+$ on each pairs of components taken from two lists of same length n. There are two choices, either a partial function defined only if the two operands are lists of same length, or a total function that is meaningful only when the two operands have the same length. The first solution leads to introduce tests as guards before performing the computation, yielding a rather complex algorithm. The second solution allows adding two vectors of unequal dimensions, which is odd and says nothing about the length of the result.

2.4 Dependent types

Clearly, the dimension is an information that has to appear explicitly, whereas it is implicit in mathematical notation. To maintain the requirement of genericity, it is handy to make the dimension to appear as a parameter in the vector.

So, we define A^n by induction as a set depending on n

$$\begin{cases} \overrightarrow{0} \in A^0; \\ (x, \overrightarrow{v}) \in A^{n+1} \textit{ if } x \in A \textit{ and } \overrightarrow{v} \in A^n. \end{cases}$$

then, mapping functions on A or binary operations on A is done by induction on the dimension n. This provides a unique definition for all values of n. By structure we will get the sum of vectors as correctly typed binary operation: $A^n \times A^n \rightarrow A^n$.

2.5 Indexed set

Another way to write vectors of A^n is $(x_1, x_2, \ldots, x_n)$ where the use of indices is a notation for a function $\mathit{ind} : [1, \ldots, n] \rightarrow A$. The advantage of this formulation is that the problem of the finiteness is concentrated in the definition of the interval $[1, \ldots, n]$.

But, in constructive type theory, unlike classical mathematics, the equality between functions is this of Leibniz instead of the extensional equality. So we do not have the equivalence: $v = v' \Leftrightarrow \forall\ i < n, v_i = v'_i$. Adding this equivalence in the environment can be costly.

3 The Coq implementation

Coq [4] is a proof assistant developed at the INRIA and based on the Calculus of Inductive Constructions. The Coq language is derived from the typed λ-calculus and is powerful and expressive, both for reasoning and for programming [3].

3.1 Definition

Let us adopt the dependent type definition of vectors and see how it can be implemented.

```
Inductive vector (A : Set) : nat -> Set :=
  | Vnil : vector A 0
  | Vcons : forall n : nat, A -> vector A n -> vector A (S n).
```

This definition makes the vectors a set dependent on another set A and on a natural n. Therefore `vector` is inductively defined by the two constructors: $Vnil$ which is the only inhabitant of the vector set of dimension 0 and $Vcons$ which builds a vector of dimension $n+1$ ($S\ n$ is the successor of n) from an element of A and a vector of dimension n.

3.2 How to proceed ?

Suppose now that one wants to map a unary function $f : A \to A$ onto vectors of a fixed but unknown dimension n. The function called `map_unary` can be defined directly using the *fixpoint* constructor:

```
Fixpoint map_unary (A:Set) (f:A->A) (n:nat) (v:vector A n)
    {struct v} : vector A n
:= match v in (vector _ n) return (vector A n) with
  | Vnil => Vnil A
  | Vcons p a w => Vcons A p (f a) (map_unary A f p w)
end.
```

It is much likely that without a good practice of Coq, the user will not be able to read it as easily as the definition of vector. The difficulty lies in the use of dependent types which requires the writer to provide a lot of structural information. Another way to proceed is to use the tactic language and instead of defining the function `map_unary` to prove its existence:

```
Definition map_unary :
  forall (A:Set) (f:A->A) (n:nat), vector A n -> vector A n.
```

This definition is interpreted by Coq as a lemma to be proved leading the system to answer:

```
1 subgoal

  ============================
   forall A : Set, (A -> A) -> forall n : nat, vector A n -> vector A n
```

The following sequence of tactics invocations completes the proof:

```
intros A f n v; induction v.
 apply Vnil.
 apply (Vcons A n (f e) IHv).
```

What has been done is easily understood. The first line introduces the variables `A`, `f`, `n` and, `v` in the environment and then asks for an induction on the variable `v`. The second line solves the base case providing the unique inhabitant of `A 0`. The third line solves the induction step. It builds the image of a vector `Vcons A n e v`, using the image by `map f` of `e` and the value `IHv` built by induction. For that, it applies the constructor `Vcons A n` to `f e` and to the value built by induction namely IHv. Notice that IHv is a name provided by the system. Except perhaps this difficulty due to a name that seems arbitrarily given by the system, such a process looks rather natural.

The command "Defined." ends the construction and now map_unary can be used like if it would have been defined by a fixpoint definition.

If one sees the first as defining a computation and the second as providing a proof, the fact that both assertions return the same internal Coq object is an application of the Curry-Howard correspondence.

3.3 Library

Coq provides a vector library [1]. It uses quite the above dependent inductive definition :

```
Section VECTORS.

Variable A : Set.

Inductive vector : nat -> Set :=
  | Vnil : vector 0
  | Vcons : forall (a:A) (n:nat), vector n -> vector (S n).
```

The main definitions are by proof terms. For example, the definitions of the first component and the vector made of all the components but the first are:

```
Definition Vhead : forall n:nat, vector (S n) -> A.
Proof.
        intros n v; inversion v; exact a.
Defined.

Definition Vtail : forall n:nat, vector (S n) -> vector n.
Proof.
        intros n v; inversion v; exact H0.
Defined.
```

Notice the use of rules aimed to make the printing lighter, namely a parameter is dropped when it can be induced automatically by the system. For example, whenever possible the set A is implicitly assumed.

4 A problem

A solution always looks nice for its creator, but on that matter the best critics remains the users. An important community of Coq users contributes to improve continuously the system. A lot of pertinent questions are published on the forum *Coq_club*.

4.1 A user question

Short after the vector library has been delivered, a question was raised in the forum.

Is the following theorem provable by Coq within the vector library?

```
Theorem VSn_eq :
  forall (n:nat) (v:vector (S n)),
     v = Vcons (Vhead v) (Vtail v).
```

This theorem says that each non null vector is made by consing its head with its tail. A really basic fact, isn't?

Unfortunately, the command `inversion v` (a basic Coq command) gives as hypothesis an element e of A and a vector w of $vector\ n$ but it does not return the equality $v = Vcons\ e\ w$ and Coq does not offer a simple command to prove such a theorem. How can such a basic theorem be not proved by a basic command in Coq? Where is the problem ? Actually the inversion algorithm uses a filter which succeeds when the constructors match. But here, only the case `Vcons` should be selected because the type is this of a vector with a non zero parameter. Such a selection does not allows us to deduce, in general, the constructor, even if, like in the present problem, it looks quite trivial.

But, a good reason is not a proper answer. Clearly the user expects not only a proof for this theorem but a proof which is as simple as the statement of the theorem.

4.2 My solution

The above theorem can be seen as the statement that a partial function, defined on n-dimension vectors with $n > 0$, is the identity function. But Coq works better on total functions. So, I build an ad hoc function:

```
Definition Vid : forall n:nat, vector n -> vector n.
  Proof.
    destruct n.
    intros; exact Vnil.
    intros v; exact (Vcons (Vhead v) (Vtail v)).
  Defined.
```

`Vid` is a function defined by cases on the dimension n. If $n = 0$, and `v: vector 0` then `Vid v` is the null vector. If $n > 0$, `Vid v` is built using $Vcons$ on the first component of `v` and on the rest of `v`, in fact on $(Vhead\ v)$ and $(Vtail\ v)$. Actually Vid is a recursive definition of the identity function, let us prove it:

```
Lemma Vid_eq : forall n:nat, forall v:vector, v=(Vid n v).
  Proof.
    destruct v; auto.
  Qed.
```

In both cases, the automatic tactic *auto* of Coq proves easily the equality since, after reduction, the internal structures of the two terms are the same. Eventually the lemma `VSn_eq` proposed in the forum is obtained as a consequence of Lemma `Vid_eq`.

```
Theorem VSn_eq : forall n:nat, forall v:vector (S n),
    v=Vcons (Vhead v) (Vtail v).
  Proof.
    intros.
    change (Vcons (Vhead v) (Vtail v)) with (Vid (S n) v).
    apply Vid_eq.
  Qed.
```

5 Conclusion

An interesting aspect of revisiting mathematics from a constructive point of view is that difficulties are not where they are usually expected. Because of powerful constructors, mainly the use of induction, complex structures are so well defined that proofs come easily. In another hand, notions, so obvious for a human being that often he leaves them implicit, require to be dealt with carefully in order to make them explicit. Whereas in classical mathematics, finite sets are usually written with ellipsis, in a constructive point of view, more information has to be given. A proof assistant like Coq valuably helps the mathematician to mechanize notions and to check their correctness, making the constructions usable for further applications.

References

[1] Users contributions to the Coq system. http://coq.inria.fr/.

[2] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[3] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[4] Coq Development team. *The Coq reference manual.*

[5] Pierre Weis and Xavier Leroy. *Le langage Caml.* InterEditions, 1994.