Trustworthy programming with dependent types and small inversions

Jean-François Monin

Joint work with Pierre Corbineau and Basile Gros

Tribute to Gilles Dowek – July 18, 2026

Université Grenoble Alpes

Verimag

Grenoble INP

Preliminary remarks

This talk is mainly about:

- Coq (Rocq), the first proof assistant that received important contributions from Gilles Dowek;
- more specifically *dependent types*, where the *conversion rule* of Type Theory plays an essential role, as pointed out by Gilles many years ago.

  Note that, most of the time, this essential rule is used *silently*.

  Special case: the conversion rule plays a silent but key role in proxy-based small inversions to be presented below.

Summary

Introduction

- About trust
- Dependent pattern-matching and small inversion

Dependent types at work 1: vectors and matrices

- map2, transposition and multiplication of matrices

Dependent types at work 2: certified compilation

- Compiling arith/bool expressions to a stack machine
- Correctness in traditional style e.g., Software Foundation [Pierce & all]
- Correctness expressed with dependent types

Conclusion

Trustworthy use of a trustworthy proof assistant

1. De Bruijn criterion
   'proof objects' (of some form) can be checked by an 'easy' algorithm
2. Pollack consistency
   parsing and displaying remains consistent
3. Coherence criterion (or computational WYSIWYG):
   when calculations are performed, the link with the source code is kept – output traces can be related to the source code

A trap about trust related to tactics

Functions/programs can be defined by a script (Curry-Howard).

```
Fixpoint plus (n m : nat) : nat.
Proof.
destruct n. exact m. apply S. apply plus. exact n. exact m.
Defined.
```

By design, following de Bruijn criterion, the typing of such a definition is trustworthy (`plus : nat → nat → nat`), but the code between Proof and Defined has no special meaning.

However, the meaning of statements of theorems such as

$$\forall\, u\ v, \mathtt{len}\,(u +\!\!+ v) = \mathtt{len}\, u + \mathtt{len}\, v$$

is based on the body of the functions-programs occurring in this formula, in the syntax of CIC.

CIC dependent pattern-matching (1)

A standard example: length-indexed lists, aka *vectors*.

```
Inductive vect (A : Type) : nat → Type :=
| nil : vect A 0
| cons : ∀ {n}, A → vect n → vect A (S n).
```

Conventions and Notations:

- `A` is implicit in `nil` and `cons`.
- `[]` for `nil`
- `x :: u` for `(cons x u)`.
- `x :n: u` for `(cons (n:=n) x u)`.

CIC dependent pattern-matching (2)

Common scheme

The type of the result depends on the index.

```
match vexp in vect _ n return Q n with
| []          => (term0 : Q 0)
| x :n': u' => (term1 : Q (S n'))
end
```

General scheme

The type of the result depends on the vector as well.

```
match vexp as u in vect _ n return P n u with
| []          => (term0 : P 0 [])
| x :n': u' => (term1 : P (S n') (x :: u'))
end
```

Inversion

As indices are generalized over, additional work is required when, in the type of *vexp*, the index is constructed.

Performed by the inversion tactic for the common scheme, using additional equalities, discrimination and so on.

NB1. The more recent Equations plugin of Sozeau is more general and principled than inversion.
Also based on additional equalities, discrimination and so on.

NB2. Do not be too quick to criticize the lack of generality of CIC dependent pattern-matching:

simplicity is good for the de Bruijn criterion.

Schematic script using inversion

Complex types – dependent types – encourage to write scripts based on destruct (for comfort) and inversion (for good reasons).

```
Fixpoint some_fct x (y : T x) (z : T' x) : R x y z.
destruct y.
    ...
    inversion z.
    ...
Defined.
```

This violates the criteria 2 and 3.

Small inversions

- Original motivation: a new lightweight approach of inversion that is handcrafted but more robust, understandable and controllable than inversion [Monin 2010, Monin & Shi 2013].
- Integrated in Rocq's elaboration mechanism (to be documented).
- Adapted using auxiliary inductive types for teaching purposes: proxy-based small inversions (PBSI).
- Improved for supporting the Braga Method [Monin 2022]: dependent PBSI.
- Automated support available [Corbineau, Gros & Monin 20??].

Vectors and matrices (1) – Warning

The above definition of `vect` is available in the standard library.

```
From Stdlib Require Import Vector.
```

```
Warning:
Using Vector.t is known to be technically difficult,
see <https://github.com/coq/stdlib/.../Vectors/Vector.v>.
```

Vectors and matrices (2) – map2

Given $f : A \to B \to C$,

$$\texttt{map2}\ f\ []\ []\ =\ [] \tag{1}$$

$$\texttt{map2}\ f\ (x :: u)\ (y :: v)\ =\ f\ x\ y\ ::\ \texttt{map2}\ f\ u\ v \tag{2}$$

Expected program

```
Fixpoint map2 (f : A → B → C) {n} (u : vect A n) :
          vect B n → vect C n :=
  match u with
  | []      => λ v, []
  | x :: u' => λ v, let (y, v') := v in
               f x y :: map2 f u' v'
  end.
```

Fails because of the simplicity of CIC pattern-matching

Indices (here: `n`) are generalized over

Vectors and matrices (3) – QWIRE, QuantumLib

A large Coq library for reasoning about quantum programs.
[Rand & all, 2017]

```
Definition Matrix (m n : nat) := nat → nat → C.
Definition WF_Matrix {m n} (A : Matrix m n) : Prop :=
  ∀ x y, x ≥ m ∨ y ≥ n → A x y = 0.
Notation Vector n := (Matrix n 1).
```

Issues

- Finite structures represented by:
  - An infinity of elements, together with
  - An infinity of proofs of nullity
- Axiom needed (extensionality)
- Domain-specific

Vectors and matrices (4) – map2 in Agda|Equations style

Code similar to Equations (1) and (2) above.

Not for free

Requires a complex pattern-matching mechanism studied from [Coquand 1992] to [Cockx & Devriese 2018] and [Sozeau & Mangin 2019]

- Axiomatic in Agda
- Definitional in Rocq/Equations
  but heavy underlying CIC terms
- Entanglement of concerns between pattern-matching and termination, generating issues with co-inductive data-structures

Vectors and matrices (5a) – map2 with inversion...

What you see is:

```
#[refine]
Fixpoint map2 (f : A → B → C) n (u : vect A n) :
  vect B n → vect C n :=
  match u with
  | []      => fun _ => []
  | x :: u' => fun v => _
  end.
Proof (* filling the hole in the above program *).
  inversion v as [ | n' y v' e].
  refine (f x y :: map2 f _ u' v').
Defined.
```

But what you get is...

Vectors and matrices (5b) – ... map2 with inversion

```
fix map2 (f : A -> B -> C) (n : nat) (u : vect A n) struct u : vect B n -> vect C n :=
  match u in (vect _ n0) return (vect B n0 -> vect C n0) with
  | [] => fun _ : vect B 0 => []
  | @cons _ n0 x u' =>
      fun v : vect B (S n0) =>
      let X :=
        match v in (vect _ n1) return (n1 = S n0 -> vect C (S n0)) with
        | [] =>
            fun H : 0 = S n0 =>
            (fun H0 : 0 = S n0 =>
             let H1 : False :=
               eq_ind 0 (fun e : nat => match e with
                                        | 0 => True
                                        | S _ => False
                                        end) I (S n0) H0
               in
             False_rect (vect C (S n0)) H1) H
        | @cons _ n1 x0 x1 =>
            (fun (n' : nat) (y : B) (v' : vect B n') (H : S n' = S n0) =>
             (fun e : S n' = S n0 =>
              let H0 : n' = n0 := f_equal (fun e0 : nat => match e0 with
                                                           | 0 => n'
                                                           | S n2 => n2
                                                           end) e in
              (fun e0 : n' = n0 =>
               let H1 : n' = n0 := e0 in
               eq_rect_r (fun n2 : nat => B -> vect B n2 -> vect C (S n0))
                 (fun (y0 : B) (v'0 : vect B n0) => f x y0 :: map2 f n0 u' v'0) H1)
                H0)
               H y v')
              n1 x0 x1
        end in
      X eq_refl
  end
```

Vectors and matrices (6) – map2 with small inversions

```
Fixpoint map2 (f : A → B → C) {n} (u : vect A n) :
         vect B n → vect C n :=
  match u with
  | []      => λ v, []
  | x :: u' => λ v, let (y, v') := vect_proxy v in
               f x y :: map2 f u' v'
  end.
```

NB. Conversion rule in action here!

Vectors and matrices (7) – proxy-based small inversions

Bare CIC : no axiom, nothing predefined – in particular : no equality

```
(* Same arity as vect *)
Definition vect_dispatch n : Type :=
  match n with
  | O    => vect_O A
  | S n' => vect_S A n'
  end.
(* Where *)
Variant vect_O (A : Type) : Type :=
| nil_O : vect_O A.
Variant vect_S (A : Type) (n : nat) : Type :=
| cons_S : A → vect A n → vect_S A n.

Definition vect_proxy A n (v : vect A n) : vect_dispatch n :=
  match v with
  | nil      => nil_O _
  | cons x v => cons_S A _ x v
  end.
```

Vectors and matrices (8) – dependent PBSI

For more advanced examples, and for reasoning on programs written with PBSI or dependent PBSI.

```
Definition vect_dispatchd n : vect A n → Type :=
  match n with
  | O    => vect_O_d A
  | S n' => vect_S_d A n'
  end.

Variant vect_O_d (A : Type) : vect A O → Type :=
| nil_O : vect_O A [].
Variant vect_S_d (A : Type) (n : nat) : vect A (S n) → Type :=
| cons_S : ∀ (x : A ) (u : vect A n), vect_S_d A n (x :: u).

Definition vect_dproxy A n (v : vect A n) : vect_dispatchd n v :=
  match v with
  | nil      => nil_O_d _
  | cons x v => cons_S_d A _ x v
  end.
```

Vectors and matrices (9) – Basic algebraic results

(m, n) matrix $=$ m-vector of n-vectors

- transposition (PBSI)
- involutivity of transposition (intensive use of dependent PBSI)
- matrix multiplication (PBSI)

  in a general algebraic setting, where scalar multiplication is not commutative and has no unit
- associativity of matrix multiplication (dependent PBSI)

```
Fixpoint transpose A m n (mat : matrix A m n) : matrix A n m :=
  match mat with
  | []         => map0 []
  | v :: mat' => map2 cons v (transpose mat')
  end.
```

Certified compilation (1) – Expressions

[McCarthy, Paynes 67]
Example given in Agda by P.E. Dagand, collège de France, 2018

```
Inductive exp : Set :=
| Val  : ∀ t, value t → exp
| Plus : exp → exp → exp
| Ifte : exp → exp → exp → exp.

Definition value (t : ty) : Set :=
  match t with   Bool => bool   |   Nat => nat   end.


Example ex120 :=
  Ifte (Val Bool true)
       (Plus (Val Nat 1) (Val Nat 2))
       (Val Nat 0).
```

Certified compilation (2) – Code

```
Inductive code : Set :=
| PUSH : ∀ t, value t → code
| ADD  : code
| IFTE : code → code → code
| SEQ  : code → code → code.
Infix "#"  := SEQ.

Fixpoint compile (e : exp) : code :=
  match e with
  | Val t v => PUSH t v
  | Plus e1 e2 => compile e1 # compile e2 # ADD
  | Ifte eb e1 e2 =>
      compile eb # IFTE (compile e1) (compile e2)
  end.
```

Certified compilation (3) – Expected correctness

```
Fixpoint semE {t} (e : exp) : value t := ...

Inductive stack : Set :=
| ϵ : stack
| push : ∀ {t}, value t → stack → stack.
Infix "•"  := push.

Fixpoint semC (c : code) : stack → stack := ...

Lemma compile_corr (e : exp) :
  ∀ π : stack, semC (compile e) π = semE e • π.
```

Certified compilation (4) – Semantics, naive attempt

```
Fail Fixpoint semE {t} (e : exp) : value t :=
  match e with
  | Val _ v c      => v
  | Plus e1 e2     => semE e1 + semE e2
  | Ifte eb e1 e2 => if semE eb then semE e1 else semE e2
  end.

Fail Fixpoint semC (c : code) : stack → stack :=
  match c with
  | PUSH t v    => λ π, (v • π)
  | ADD         => λ π, match π with  m • n • π =>
       n + m • π ...
  | IFTE c1 c2 => λ π, match π with b • π =>
      if b then semC c1 π else semC c2 π
  | c1 # c2     => λ π, semC c2 (semC c1 π)
  end.
```

Certified compilation (5) – Well-typed expressions

```
Inductive well_typed : exp → ty → Prop :=

| WTVal t (v : value t) : well_typed (Val t v) t

| WTPlus (e1 e2 : exp) :
  well_typed e1 Nat → well_typed e2 Nat →
  well_typed (Plus e1 e2) Nat

| WTIfte (eb e1 e2 : exp) (t : ty) :
  well_typed eb Bool → well_typed e1 t → well_typed e2 t →
  well_typed (Ifte eb e1 e2) t.
```

Certified compilation (6) – Semantics on WT expressions

```
Fixpoint semE t (e : exp) : well_typed e t → value t :=
    ...
    inversions needed
    ...
```

Certified compilation (7) – Correctness OK

```
Lemma compile_corr e t (w : well_typed e t) :
  ∀ π, semC (compile e) π = semE e w • π.
```

Proof by *dependent* induction on `w`.

Actually:

```
Fixpoint compile_corr e t (w : well_typed e t) {struct w} :
  ∀ π, semC (compile e) π = semE e w • π.
Proof.
  destruct w as ...; intro π; cbn.
  ...
Qed.
```

Certified compilation (8) – Intrinsically WT expressions

Here, expressions are indexed by a concrete type.

```
Inductive exp : ty → Set :=
| Val  : ∀ t, value t → exp t
| Plus : exp Nat → exp Nat → exp Nat
| Ifte : ∀ t, exp Bool → exp t → exp t → exp t.

(* Very simple! *)
Fixpoint semE t (e : exp t) : value t :=
  match e with
  | Val _ v => v
  | Plus e1 e2 => semE e1 + semE e2
  | Ifte eb e1 e2 => if semE eb then semE e1 else semE e2
  end.
```

Certified compilation (9) – Intrinsically WT code

```
Inductive code : List ty → List ty → Set :=
| PUSH : ∀ {t σ}, value t → code σ (t :: σ)
| ADD  : ∀ {σ}, code (Nat :: Nat :: σ) (Nat :: σ)
| IFTE : ∀ {σ1 σ2} : code σ1 σ2 → code σ1 σ2 →
         code (Bool :: σ1) σ2
| SEQ  : ∀ {σ1 σ2 σ3}, code σ1 σ2 → code σ2 σ3 →
         code σ1 σ3.

Fixpoint semC {σ1 σ2} (c : code σ1 σ2) : stack σ1 → stack σ2 :=
  match c with
  | PUSH v => λ π, (v • π)
  | ADD => λ π,
      let (m, π') := stack_sinv π in
      let (n, π'') := stack_sinv π' in
      (n + m : value Nat) • π''
  | IFTE c1 c2 => λ π,
      let (b, π') := stack_sinv π in
      if b then semC c1 π' else semC c2 π'
  | c1 # c2 => λ π, semC c2 (semC c1 π)
  end.
```

Conclusion

"Golden" dependent types are also available in Coq/Rocq

- Equations plugin by Sozeau
- Our *proxy-based small inversions*

Features of PBSI

- Small and readable bare CIC
- Satisfy the de Bruijn's criterion, Pollack consistency and the coherence criterion
- Bonus : agnostic w.r.t. the inductive viz co-inductive nature of data structures.
  E.g., the *same* `map2` works on co-inductive lists indexed by co-nats.

Automation available

Generates the expected readable definitions
`https://github.com/BasileGros/proxy-based-small-inversions`