Trustworthy Programming with Precise Types: a Minimalist Approach using Small Inversions

Pierre Corbineau, Basile Gros and

Jean-François Monin

LPAR, Σπέτσες, October 2026

UNIVERSITÉ Grenoble Alpes

Verimag

Grenoble INP

Material

https://github.com/BasileGros/proxy-based-small-inversions

- Automated support
- Tutorial
- More advanced examples

The paper

- Explanations on PBSI
- The principles behind our plugin

This talk

- Invitation to program with dependent types using PBSI and our tool

Motivating example: certified compilation

- Compiler
  `compile : source → code`
- Expectation
  `∀ s, semC (compile s) = semE e`

Very basic example in CIC, the language of Coq and Rocq:
a challenge coming from [McCarthy, Paynes 67]

- `exp`: a simple language of bool+arith expressions
  constants, addition, if-then-else
- `code`: for a stack machine
- $\forall\, \mathsf{e}\, \forall\, \pi$, semC (compile e) π = semE e $\bullet$ π.

Agda solution by P.E. Dagand (collège de France, 2018)
Programming a compiler with dependent types.

Motivating example: certified compilation

- Compiler
  `compile : source → code`
- Expectation
  `∀ s, semC (compile s) = semE e`

Very basic example in CIC, the language of Coq and Rocq:
a challenge coming from [McCarthy, Paynes 67]

- `exp`: a simple language of bool+arith expressions
  constants, addition, if-then-else
- `code`: for a stack machine
- $\forall$ e $\forall \pi$, semC (compile e) π = semE e $\bullet$ π.

Agda solution by P.E. Dagand (collège de France, 2018)
Programming a compiler with dependent types.

Naive certified compilation (1) – Expressions

if true then 1+2 else 0

```
Ifte (Val Bool true)
     (Plus (Val Nat 1) (Val Nat 2))
     (Val Nat 0).
```

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

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

Naive certified compilation (2) – Code

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

Infix "#"  := SEQ.
```

Naive certified compilation (3) – Compiler

```
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.
```

Naive certified compilation (4) – 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 := ...

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

Naive certified compilation (5) – Semantics

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

```
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.
```

Naive certified compilation (5) – Semantics

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

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 using extrinsically well-typed expressions (1)

Software Foundation style [Pierce & al.]

```
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 using extrinsically well-typed expressions (2) – Semantics

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

Inversion: if `well_typed (Plus e1 e2) t`
then

- `t` is `Nat`
- `well_typed e1 Nat`
- `well_typed e2 Nat`

allowing recursive calls on `e1` and `e2`.

Certified compilation using extrinsically well-typed expressions (2) – Semantics

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

Inversion: if `well_typed (Plus e1 e2) t`
then

- `t` is `Nat`
- `well_typed e1 Nat`
- `well_typed e2 Nat`

allowing recursive calls on `e1` and `e2`.

Certified compilation using extrinsically well-typed expressions (3) – Correctness

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

Certified compilation using intrinsically well-typed expressions (1)

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.

(* Elegant, 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 using intrinsically well-typed expressions (2) – 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.
```

NB: we also use intrinsically WT stacks!

```
Fixpoint semC {σ1 σ2} (c : code σ1 σ2) : stack σ1 → stack σ2 :=
  match c with
  | PUSH v     => λ π, (v • π)
  | ADD        => λ π, ???
  | IFTE c1 c2 => λ π, ???
  | c1 # c2    => λ π, semC c2 (semC c1 π)
  end.
```

In summary

Dependent types allow for very nice programs

provided you have the right tools.

But what about trust?

Contents

About trust

- Trust criteria
- Dependent pattern-matching and inversion

Small inversions in programming with dependent types

- Explained on vectors and matrices:
  map2, transposition and multiplication of matrices
- Back to certified compilation:
  intrinsically well-typed expressions

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

Trust and 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`
- the meaning of the piece of code between
  Proof and Defined is deliberately not formally defined,
  you are not supposed to rely on it!

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O   => m
  | S n => S (plus n m)
  end.
```

THE TRAP

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O   => m
  | S n => S (plus n m)
  end.
Infix "+"  := plus.
```

The meaning of statements of theorems such as

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

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

Of course, nobody writes `plus`, `append` and `len` in interactive proof mode.

But `semE` on *extrinsically* well-typed expressions was written in interactive mode, if you used the very popular Rocq's inversion!

THE TRAP

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O   => m
  | S n => S (plus n m)
  end.
Infix "+"  := plus.
```

The meaning of statements of theorems such as

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

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

Of course, nobody writes `plus`, `append` and `len` in interactive proof mode.

But `semE` on *extrinsically* well-typed expressions was written in interactive mode, if you used the very popular Rocq's inversion!

CIC dependent pattern-matching (1)

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

```
Inductive vect (A : Type) : nat → Type :=
| nil : vect A O
| 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
```

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. CIC dependent pattern-matching is simple, but
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.
```

Even when it works, criteria 2 and 3 are violated.

Additionally, it may fail.

Small inversions: a minimalist approach

Originally: for proofs [Monin 2010, Monin & Shi 2013]

Lightweight approach ⇒
more robust, understandable and controllable than inversion.

Downsides:

- Based on continuations (not for beginners)
- Systematic/heuristic approach, no automation
- Dependent inversion not considered

This talk: proxy-based small inversions (PBSI)

- Based on auxiliary inductive types (familiar)
- Automated support (for a large class of common cases)
- Dependent PBSI

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 (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 (3a) – map2 with inversion...

What you see is (at best):

```
#[refine]
Fixpoint map2 (f : A → B → C) n (u : vect A n) :
  vect B n → vect C n :=
  match u with
  | []      => λ _, []
  | x :: u' => λ 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 (3b) – ... 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 (4) – 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.
```

Vectors and matrices (5) – proxy-based small inversions

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

```
(* Same arity as vect; also named vect_proxy_type *)
Definition vect_dispatch A 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 A n :=
  match v with
  | nil      => nil_O _
  | cons x v => cons_S A _ x v
  end.
```

Vectors and matrices (6) – dependent PBSI

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

```
Definition vect_ddispatch A 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_ddispatch A n v :=
  match v with
  | nil      => nil_O_d _
  | cons x v => cons_S_d A _ x v
  end.
```

Vectors and matrices (7) – Basic algebraic results

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

- transposition
- involutivity of transposition (dependent PBSI)
- matrix multiplication in a general algebraic setting
- 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.
```

Vectors and matrices (7) – Basic algebraic results

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

- transposition
- involutivity of transposition (dependent PBSI)
- matrix multiplication in a general algebraic setting
- 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.
```

Back to certified compilation – 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_proxy π in
      let (n, π'') := stack_proxy π' in
      (n + m : value Nat) • π''
  | IFTE c1 c2 => λ π,
      let (b, π') := stack_proxy π in
      if b then semC c1 π' else semC c2 π'
  | c1 # c2 => λ π, semC c2 (semC c1 π)
  end.
```

The engine of small inversion

Intuition: if `v` is of type `vect A (S n)`, then `v` is `(y :: v')` with

- `y` of type `A`
- `v'` of type `vect A n`

Actually:

- Pattern matching on an index

  here: `S n` of type `nat`

  restricts pattern matching on the term of interest

  here: `v` of type `vect A (S n)`

- Knowledgeable use of parameters:

  `n` becomes a parameter of `vect_S`

PBSI: PM on index occurs once for all in `vect_dispatch`,
the type of `vect_proxy`

Other approaches: based on additional equalities

$\Rightarrow$ repeated uses of *discrimination* or `noConfusion`

$\Rightarrow$ pattern matching on indices behind the scene.

Equations (same style as Agda)

Code for `map2` provided by 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 $\Rightarrow$ issues with co-inductive data-structures

Comparison table

https://www-verimag.imag.fr/DIST-TOOLS/PBSI/

	inversion	SI'2013	EQUATIONS	PBSI
map2	5 | 8	6	3	6
map2 CIC	35	10	22	8
map2 proof	✗	✗✓	✓	✓
comap2 (cofixpoint)	✓	✓	✗	✓
remap2	✗	✗ 6	3	6
remap2 CIC	✗	✗10	55	12
remap2 proof	✗	✗✓	✓	✓
even on bounded nat	✗	2	2	2*
even CIC	✗	2	21	2
sqeven	✗	3	3	3*
sqeven CIC	✗	3	54	4
source code readability	+	+	+ + +	++
trust criteria	✗✗✗	✓✓✓	✓✗✗	✓✓✓

Conclusion

Features of PBSI

- Small and readable bare CIC
- De Bruijn's criterion, Pollack consistency and the coherence criterion satisfied
- Also works on co-inductive data structures

Programming with dependent types made funny in Coq/Rocq

`https://github.com/BasileGros/proxy-based-small-inversions`