Certified programming with dependent types made simple with proxy-based small inversions

Pierre Corbineau   Basile Gros   Jean-François Monin

VERIMAG, Univ. Grenoble Alpes, CNRS, Grenoble INP[1]

27 September, 2025

[1] Institute of Engineering Univ. Grenoble Alpes

Motivation

Proxy-based small inversions
allow for dependent programming
with simplified and readable code.

Examples

- Definition of transposition of size-indexed matrices (vectors of vectors) and proof that this transposition is involutive.
- Manipulation of finite sets `Fin.t`, following a challenging use-case proposed by Clément Pit-Claudel

Example : RGB

```
Inductive rgb := R | G | B.

Definition my_rgb_rect
  (P : rgb → Type)
  (p1 : P R) (p2 : P G) (p3 : P B)
  (r : rgb) : P r :=
match r as r' return P r' with
| R ⇒ p1
| G ⇒ p2
| B ⇒ p3
end.
```

Example : Fin.t 3

```
Inductive Fin.t : nat → Set :=
| F1 : ∀ n : nat, t (S n)
| FS : ∀ n : nat, t n → t (S n).
```

Example : Fin.t 3

```
Inductive Fin.t : nat → Set :=
| F1 : ∀ n : nat, t (S n)
| FS : ∀ n : nat, t n → t (S n).

Definition Fin_3_rect
(P : Fin.t 3 → Type)
(p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
(x : Fin.t 3) : P x :=
match x as _x return P _x with
| F1 ⇒ p1
| FS F1 ⇒ p2
| FS (FS F1) ⇒ p3
end.
```

Example : Fin.t 3

```
Inductive Fin.t : nat → Set :=
| F1 : ∀ n : nat, t (S n)
| FS : ∀ n : nat, t n → t (S n).

Definition Fin_3_rect
(P : Fin.t 3 → Type)
(p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
(x : Fin.t 3) : P x :=
match x as _x return P _x with
| F1 ⇒ p1
| FS F1 ⇒ p2
| FS (FS F1) ⇒ p3
end.

Error: The term "_x" has type "t n"
while it is expected to have type "t 3".
```

Example : Fin.t 3

```
Definition Fin_3_rect
  (P : Fin.t 3 → Type)
  (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
  (x : Fin.t 3) : P x :=
match x as _x return P _x with
| F1 ⇒ p1
| FS x' ⇒
    match x' return P (FS x') with
    |F1 ⇒ p2
    |FS x'' ⇒
        match x'' return P (FS (FS x'')) with
        | F1 ⇒ p3
        | FS x''' ⇒ _ (* ??? *)
        end end end.
```

Example : small inversion

```
Definition Fin_3_rect_smallinv
  (P : Fin.t 3 → Type)
  (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
  (x : Fin.t 3) : P x :=
match Fin_proxy 3 x with
|is_F1 _ ⇒ p1
|is_FS _ x' ⇒
    match Fin_proxy 2 x' with
    |is_F1 _ ⇒ p2
    |is_FS _ x'' ⇒
        match Fin_proxy 1 x'' with
        |is_F1 _ ⇒ p3
        |is_FS _ x''' ⇒
            match Fin_proxy 0 x''' with end
    end end end.
```

Example : small inversion intermediary objects

```
Inductive Fin_0 : Fin.t 0 → Set :=.
Inductive Fin_S (n : nat) : Fin.t (S n) → Set :=
| is_F1 : Fin_S n F1
| is_FS (r:Fin.t n) : Fin_S n (FS r).

Definition Fin_proxy_type (n:nat) : Fin.t n → Set :=
  match n with
  | 0 ⇒ Fin_0
  | S m ⇒ Fin_S m end.

Definition Fin_proxy(n:nat) (r : Fin.t n) : Fin_proxy_type n r :=
  match r as r' in Fin.t n' return Fin_proxy_type n' r' with
  | F1 n ⇒ is_F1 n
  | FS n t' ⇒ is_FS n t' end.
```

Example : inversion tactic

```
Definition Fin_3_rect_inv
  (P : Fin.t 3 → Type)
  (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
  (x : Fin.t 3) : P x.
Proof.
  inversion x as [n' eq| n' i' eq].
```

Goal : P x

Example : inversion tactic

```
Definition Fin_3_rect_inv
  (P : Fin.t 3 → Type)
  (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
  (x : Fin.t 3) : P x.
Proof.
  inversion x as [n' eq| n' i' eq].
```

Goal : P x

```
  exact p1.
Error: The term "p1" has type "P F1"
while it is expected to have type "P x".
```

Example : dependent inversion tactic

```
Definition Fin_3_rect_inv
  (P : Fin.t 3 → Type)
  (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
  (x : Fin.t 3) : P x.
Proof.
  dependent inversion x.
```

Example : dependent inversion tactic

```
Definition Fin_3_rect_inv
  (P : Fin.t 3 → Type)
  (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
  (x : Fin.t 3) : P x.
Proof.
  dependent inversion x.
```

```
Error: The term "P" of type "t 3 → Type"
cannot be applied to the term "x0" : "t n"
This term has type "t n" which should be a subtype of "t 3".
```

Example : dependent elimination

```
From Equations Require Import Equations.
Definition Fin_3_rect_depelim
  (P : Fin.t 3 → Type)
  (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
  (x : Fin.t 3) : P x.
Proof.
  dependent elimination x as [F1|FS F1|FS (FS F1)].
  - exact p1.
  - exact p2.
  - exact p3.
Defined.
```

Example : result of dependent elimination

```
match x as t0 in (t n) return
    ({| pr1 := n; pr2 := t0 |} = {| pr1 := 3; pr2 := x |} → P x)
with
| F1 n ⇒
  DepElim.eq_simplification_sigma1_dep (S n) 3 F1 x
    (apply_noConfusion (S n) 3
        (fun H : n = 2 ⇒
        DepElim.solution_left_dep 2
          (fun H0 : eq_rect 3
            (fun n0 : nat ⇒ t n0)
            F1 3
            (noConfusion eq_refl) = x
          ⇒
DepElim.solution_right (eq_rect 3 (fun n0 : nat ⇒ t n0) F1 3
                        (noConfusion eq_refl)) p1 x H0 ) n H))
```

Small inversion

- The conclusion of the elimination scheme for `Fin.t` is
  `∀ n, ∀ (x:Fin.t n), P n x`
- Objective: constrain n to be 3 : `∀(x:Fin.t 3), P x`
- Historical methods change the conclusion:
  `∀ n, ∀ (x:Fin.t n), n = 3 => P n x.`
- Proxy-based small inversions change the matched objet.
  - Create a proxy inductive type that mimics `Fin.t 3`, and can be eliminated without loss of information.
  - We go from `(x:Fin.t 3)` $\longrightarrow$ `P x`

    to `(x:Fin.t 3)` $\longrightarrow$ *proxy* `(Fin.t (S 2))` $\longrightarrow$ `P x`

Partial inductive types

- First, *partial inductive types* mimic the comportment of the inductive type when specialised to a given pattern of the index.
- We work with inductive indices, the possible primitive patterns for the index are built from the constructors of its type.

```
Inductive Fin.t : nat → Set :=
| F1 : ∀ n : nat, Fin.t (S n)
| FS : ∀ n : nat, Fin.t n → Fin.t (S n).

Inductive Fin_0 : Set :=.
Inductive Fin_S (n : nat) : Set :=
| is_F1 : Fin_S n
| is_FS (r:Fin.t n) : Fin_S n.
```

Partial inductive types for dependent inversion

For dependent inversion, we also keep trace of the structure of the object we invert.

```
Inductive Fin_0 : Fin.t 0 -> Set :=.
Inductive Fin_S (n : nat) : Fin.t (S n) -> Set :=
| is_F1 : Fin_S n F1
| is_FS (r:Fin.t n) : Fin_S n (FS r).
```

Selecting the inductive type

- Then, two translation functions translate the original object into an object of the corresponding partial inductive type.
- The first maps index values to the partial inductive types.

```
Definition Fin_proxy_type (n:nat) : Fin.t n → Set :=
  match n with
  | 0   ⇒ Fin_0
  | S m ⇒ Fin_S m
  end.
```

Translating the inductive type

The second maps constructors to their proxy counterpart.

```
Definition Fin_proxy{n} (r : Fin.t n) : Fin_proxy_type n r :=
  match r as r' in Fin.t n' return Fin_proxy_type n' r' with
  | F1 n     ⇒ is_F1 n
  | FS n t'  ⇒ is_FS n t'
  end.
```

Using the proxy

- These objects only need to be created once.
- To use them, we then perform an elimination of the translated proxy object:

```
match Fin_proxy x with
| is_F1 _     ⇒ p1
| is_FS _ x' ⇒
    match Fin_proxy x' with
    | is_F1 _      ⇒ p2
    | is_FS _ x'' ⇒ ...
```

Using the proxy : typeclass

It is possible to wrap the proxy in a typeclass so that remembering the proxy name is not necessary.

```
Class Proxy (T:Type) :=
{  proxy_type: Type;
   proxy:      T → proxy_type }.

Class dProxy (T:Type) :=
{ dproxy_type: T → Type;
  dproxy:      ∀ t:T, dproxy_type t }.


match dProxy/proxy (x : Fin.t 3) with ...
```

Systematic creation

Partial inductive types and proxies are systematically derived by successive refinements of the inductive type through different transformations:

- Derecursivation: removing recursive references to the inductive type.
- Deparameterisation: transforming parameters into indices.
- Transformation into dependent inversion *if needed*.
- Specialisation: creating partial inductives for a given inductively typed index;
  can be iterated for deep or multiple patterns.
- Parameterisation: transforming as many indices as possible into parameters.

Current and future work

Ongoing work:

- MetaRocq plugin that automates the definition of proxies.
- Exploration of edge cases in the transformations.
- Case studies (CompCert...)

Future objectives:

- Support for inversion with dependently typed indices.
- Support for inversion with non-linear patterns.
- Eventually: integration of proxy-based small inversions into the Equations plugin?