Small inversions for smaller inversions

Jean-François MONIN
### Even natural numbers

Inductive even : n, Prop :=
| Ev0 : even 0
| Ev2 n : even n → even (S (S n)).

### Basic usage

Lemma even_plus_left n m : even n → even (n + m) → even m.

IH\: even \: even \: even \: even

IHen : even (n + m) → even m
enm : even (S (S (n + m)))

=================================

even m
Inversion

Purpose
Extract the information contained in a hypothesis $H$ of type $T$

- where $T$ is an inductive relation
- with some arguments having an inductive type

Expectations
- For each case (constructor), decompose $H$ into ALL its components
- In particular, remove irrelevant cases

Essentially: (subtle) case analysis on $H$

- Simultaneous case analysis on $H$ and its arguments
- game on dependent pattern-matching
Joint work with Dominique Larchey Wendling [TYPES’18], [Proof&Computation II 2021]

Half of even numbers

Fixpoint half n (e: even n) {struct e} : nat :=
  match n return even n → nat with
  | O => λ _, 0
  | 1 => λ e, match even_inv e with end
  | S (S n) => λ e, S (half n (πeven e))
end e.

Projection: getting ONE STRUCTURALLY SMALLER component

Definition πeven n (e: even (S (S n))) : even n :=
  match e in even m return
    let n := match m with S (S n) => n | _ => n end in
    let G := match m with S (S n) => True | _ => False end in G → even n
  with
  | Ev2 n e => λ _, e
  | _ => λ fa, match fa with end
end I.
Reasoning on half

Easy (induction on e)
Lemma double_half : n e, half n e + half n e = n.

Less easy: induction on e and inversion on e'
Lemma half_pirr : n (e e' : even n), half n e = half n e'.

\begin{align*}
e & : \text{even } n \\
e' & : \text{even } (S (S n))
\end{align*}

S (half n e) = half (S (S n)) e'
Unicity of $e$

Again: induction on $e$ and inversion on $e'$

Lemma even_unique : $n \ (e \ e' : \ \text{even} \ n), \ e = e'$.

But proof unicity should not be overrated here

- The returned result (sort Set/Type) cannot depend on an argument of sort Prop
- Simple example: unbounded linear search algorithm (see ConstructiveEpsilon.v in the std lib)
More sophisticated inversions

- Even bounded natural numbers
- Half of even bounded natural numbers
- Proof unicity for $=$ and $\not=$ in nat

Bounded natural numbers

```coq
Inductive t : nat → Set :=
| FO {n} : t (S n)
| FS {n} : t n → t (S n).
```

Failures for standard inversion.
Inversion technologies

Standard tactic of Coq: fully automated [Cornes & Terrasse, 1995; Murthy?]
- Improved over the years, very impressive black box
- Lack of control
- Big underlying terms
- Failures with dependent inductive types

Small inversions: handcrafted [Monin 2010, Monin & Shi 2013]
- Flexible approach with several variants
- Developed for a big experiment with CompCert
- Attempts towards automation (Braibant, Boutillier)

TYPE’2022
- Made clearer with auxiliary inductive types
- Improvement needed for dependent types
Small inversions with auxiliary inductive types

Recipe

Given an inductive relation \( \text{rel} : \text{T}x \rightarrow \text{Ty}_1 \rightarrow \ldots \text{Prop} \) with “input” argument \( x : \text{T}x \), define:

- For each input case (constructor \( C \)) in \( \text{T}x \), an auxiliary inductive relation of type \( \text{Ty}_1 \rightarrow \ldots \text{Prop} \) by copy and paste of relevant telescopes of \( \text{rel} \)
  - No recursion

- A dispatch function \( \text{rel}\_\text{disp} \) from \( x : \text{T}x \) to \( \text{Ty}_1 \rightarrow \ldots \text{Prop} \) by pattern matching on \( x \)

- Inversion lemma \( \text{rel}\_\text{inv} : \text{rel} \rightarrow \text{rel}\_\text{disp} \) (easy proof)

Usage

- Given a hypothesis \( R : \text{rel} (C\ldots) \text{expr}_1\ldots \)
  perform match \( \text{rel}\_\text{inv} R \) with...

- Boils down to the relevant aux. inductive relation corresponding to \( (C\ldots) \)
Small inversions with auxiliary inductive types

Recipe

Given an inductive relation \( \text{rel} : \text{Tx} \rightarrow \text{Ty}_1 \rightarrow \ldots \text{Prop} \)
with “input” argument \( x : \text{Tx} \), define:

- For each input case (constructor \( C \)) in \( \text{Tx} \),
  an auxiliary inductive relation of type \( \text{Ty}_1 \rightarrow \ldots \text{Prop} \)
  by copy and paste of relevant telescopes of \( \text{rel} \)
  *No recursion*

- A dispatch function \( \text{rel\_disp} \) from \( x : \text{Tx} \) to \( \text{Ty}_1 \rightarrow \ldots \text{Prop} \)
  by pattern matching on \( x \)

- Inversion lemma \( \text{rel\_inv} : \text{rel} \rightarrow \text{rel\_disp} \) (easy proof)

Usage

- Given a hypothesis \( R : \text{rel} (C\ldots) \text{expr\_1}\ldots \)
  perform \( \text{match rel\_inv R with}\ldots \)

- Boils down to the relevant aux. inductive relation corresponding to \( (C\ldots) \)
Small inversion for dependent (data) types

Explicit injectivity

When $R$ occurs as an argument in the goal we need also the left inverse $\text{rel\_back}$ of $\text{rel\_inv}$ (trivial as well), and a proof of $R = \text{rel\_back} (\text{rel\_inv} R)$.

Then rewrite the occurrences of $R$ with $\text{rel\_back} (\text{rel\_inv} R)$ before the pattern-matching on $\text{rel\_inv} R$.

Improvement: built-in injectivity

- In the previous recipe, add a last argument of shape $C$...
- Same code for $\text{rel\_disp}$ and $\text{rel\_inv}$
- Bonus: inline $\text{rel\_disp}$ in the statement of $\text{rel\_inv}$
Basic small inversion on even [2021 talks]

Inductive even : n, Prop :=
  | Ev0 : even 0
  | Ev2 n : even n → even (S (S n)).

Inductive even0 : Prop := even0_Ev0 : even0.
Inductive even1 : Prop :=.
Inductive even2 n : Prop := even2_Ev2 : even n → even2 n.

Definition even_inv {n} (e : even n) :
  match n return Prop with
  | 0 => even0
  | 1 => even1
  | S (S n) => even2 n
  end.
Proof. destruct e; constructor; assumption. Defined.

Definition even_back {n} (e : match n return Prop with...) : even n.
Proof... Defined.

Lemma even_inv_mono {n} (e : even n) : e = even_back (even_inv e).
Proof. destruct e; reflexivity. Qed.
Basic small inversion on even [2021 talks]

Inductive even : n, Prop :=
| Ev0 : even 0
| Ev2 n : even n → even (S (S n)).

Inductive even0 : Prop := even0_Ev0 : even0.
Inductive even1 : Prop :=.
Inductive even2 n : Prop := even2_Ev2 : even n → even2 n.

Definition even_inv {n} (e : even n) :
  match n return Prop with
  | 0 => even0
  | 1 => even1
  | S (S n) => even2 n
end.
Proof. destruct e; constructor; assumption. Defined.

Definition even_back {n} (e : match n return Prop with...) : even n.
Proof... Defined.

Lemma even_inv_mono {n} (e : even n) : e = even_back (even_inv e).
Proof. destruct e; reflexivity. Qed.
Basic small inversion on even [2021 talks]

Inductive even : n, Prop :=
  | Ev0 : even 0
  | Ev2 n : even n → even (S (S n)).

Inductive even0 : Prop := even0_Ev0 : even0.
Inductive even1 : Prop :=.
Inductive even2 n : Prop := even2_Ev2 : even n → even2 n.

Definition even_inv {n} (e : even n) :
  match n return Prop with
  | 0    => even0
  | 1    => even1
  | S (S n) => even2 n
  end.
Proof. destruct e; constructor; assumption. Defined.

Definition even_back {n} (e : match n return Prop with...) : even n.
Proof... Defined.

Lemma even_inv_mono {n} (e : even n) : e = even_back (even_inv e).
Proof. destruct e; reflexivity. Qed.
Inductive even : n, Prop :=
  | Ev0 : even 0
  | Ev2 n : even n → even (S (S n)).

Inductive is_Ev0 : even 0 → Prop := is_Ev0_intro : is_Ev0 Ev0.
Inductive no_Ev1 : even 1 → Prop :=.
Inductive is_Ev2 n : even (S (S n)) → Prop :=
  is_Ev2_intro : (e : even n), is_Ev2 n (Ev2 n e).

Definition even_inv {n} (e : even n) :
  match n return even n → Prop with
  | 0 => is_Ev0
  | 1 => no_Ev1
  | S (S n) => is_Ev2 n
  end e.

Proof. destruct e; constructor. Defined.

(* Basic version *)
Inductive even0 : Prop := even0_Ev0 : even0.
Inductive even1 : Prop :=.
Inductive even2 n : Prop := even2_Ev2 : even n → even2 n.
Exercise: equality in nat with obvious UP

Inductive \texttt{diag} : \texttt{nat \to nat \to Prop} :=
| \texttt{dia0} : \texttt{diag 0 0}
| \texttt{diaS x y} : \texttt{diag x y} \to \texttt{diag (S x) (S y)}.

(* small inversion: standard receipt with built-in injectivity *)
Inductive \texttt{is_dia0} : \texttt{diag 0 0} \to \texttt{Prop} := \texttt{ii00} : \texttt{is_dia0 dia0}.
Inductive \texttt{is_diaS x y} : \texttt{diag (S x) (S y)} \to \texttt{Prop} :=
  \texttt{iiSS} : (d : \texttt{diag x y}), \texttt{is_diaS x y (diaS x y d)}.
Inductive \texttt{no_dia} x y : \texttt{diag x y} \to \texttt{Prop} := .

Definition \texttt{diag_inv \{x y\} (d : \texttt{diag x y})} :
  \texttt{match x, y return diag x y \to Prop with}
  | 0, 0 \to is_dia0
  | S x, S y \to is_diaS x y
  | x, y \to no_dia x y
end d.

Proof. destruct d; constructor. Qed.
Exercise: equality in nat with obvious UP

Inductive diag : nat → nat → Prop :=
| dia0 : diag 0 0
| diaS x y : diag x y → diag (S x) (S y).

(* small inversion : standard receipe with built-in injectivity *)
Inductive is_dia0 : diag 0 0 → Prop := ii00 : is_dia0 dia0.
Inductive is_diaS x y : diag (S x) (S y) → Prop :=
  iiSS : (d : diag x y), is_diaS x y (diaS x y d).
Inductive no_dia x y : diag x y → Prop := .

Definition diag_inv {x y} (d : diag x y) :
  match x, y return diag x y → Prop with
  | 0, 0 => is_dia0
  | S x, S y => is_diaS x y
  | x, y => no_dia x y
  end d.

Proof. destruct d; constructor. Qed.
Simple explicit UIP in nat

**Definition diag_refl** \{x\} : diag x x.
Proof. induction x as [ | x IHx]; constructor. apply IHx. Defined.

**Definition eq_diag** \{x y\} (e : x = y) : diag x y.
Proof. case e. apply diag_refl. Defined.

**Definition diag_back** \{x\} : \(\forall y, \text{diag} x y \rightarrow x = y\).
Proof. induction x; destruct y; intro d; destruct (diag_inv d);
[reflexivity | apply f_equal, (IHx _ d)]. Defined.

**Lemma diag_back_isrefl** \{x\} : (d : diag x x), eq_refl = diag_back d.
Proof. induction x as [ | x IHx]; simpl; intro d; destruct (diag_inv d);
[reflexivity | case (IHx d). cbn. reflexivity]. Qed.

**Lemma diag_mono** \{x y\} (e : x = y) : e = diag_back (eq_diag e).
Proof. destruct e; destruct x as [ | x]; simpl.
  + destruct (diag_inv dia0); reflexivity.
  + destruct (diag_inv (diaS x x diag_refl)) as [d]. case (diag_back_isrefl d); reflexivity.
Qed.

**Corollary UIP_nat** (x: nat) (e : x = x) : eq_refl = e.
Proof. rewrite (diag_mono e). apply diag_back_isrefl. Qed.
Simple explicit UIP in nat

Definition diag_refl {x} : diag x x.
  Proof. induction x as [ | x IHx]; constructor. apply IHx. Defined.

Definition eq_diag {x y} (e : x = y) : diag x y.
  Proof. case e. apply diag_refl. Defined.

Definition diag_back {x} : y, diag x y → x = y.
  Proof. induction x; destruct y; intro d; destruct (diag_inv d);
  [reflexivity | apply f_equal, (IHx _ d)]. Defined.

Lemma diag_back_isrefl {x} : (d : diag x x), eq_refl = diag_back d.
  Proof. induction x as [ | x IHx]; simpl; intro d; destruct (diag_inv d);
  [reflexivity | case (IHx d). cbn. reflexivity]. Qed.

Lemma diag_mono {x y} (e : x = y) : e = diag_back (eq_diag e).
  Proof. destruct e; destruct x as [ | x]; simpl.
  + destruct (diag_inv dia0); reflexivity.
  + destruct (diag_inv (diaS x x diag_refl)) as [d]. case (diag_back_isrefl d); reflexivity.
  Qed.

Corollary UIP_nat (x: nat) (e : x = x) : eq_refl = e.
  Proof. rewrite (diag_mono e). apply diag_back_isrefl. Qed.
Simple explicit UIP in nat

Definition diag_refl \{x\} : diag x x.
   Proof. induction x as [ | x IHx]; constructor. apply IHx. Defined.

Definition eq_diag \{x y\} (e : x = y) : diag x y.
   Proof. case e. apply diag_refl. Defined.

Definition diag_back \{x\} : \forall y, diag x y \rightarrow x = y.
   Proof. induction x; destruct y; intro d; destruct (diag_inv d);
   [reflexivity | apply f_equal, (IHx _ d)]. Defined.

Lemma diag_back_isrefl \{x\} : (d : diag x x), eq_refl = diag_back d.
   Proof. induction x as [ | x IHx]; simpl; intro d; destruct (diag_inv d);
   [reflexivity | case (IHx d). cbn. reflexivity]. Qed.

Lemma diag_mono \{x y\} (e : x = y) : e = diag_back (eq_diag e).
   Proof. destruct e; destruct x as [ | x]; simpl.
   + destruct (diag_inv dia0); reflexivity.
   + destruct (diag_inv (diaS x x diag_refl)) as [d]. case (diag_back_isrefl d); reflexivity.
   Qed.

Corollary UIP_nat (x: nat) (e : x = x) : eq_refl = e.
   Proof. rewrite (diag_mono e). apply diag_back_isrefl. Qed.
Simple explicit UIP in nat

Definition diag_refl \{x\} : diag x x.
  Proof. induction x as [ | x IHx]; constructor. apply IHx. Defined.

Definition eq_diag \{x y\} (e : x = y) : diag x y.
  Proof. case e. apply diag_refl. Defined.

Definition diag_back \{x\} : y, diag x y → x = y.
  Proof. induction x; destruct y; intro d; destruct (diag_inv d);
  [reflexivity | apply f_equal, (IHx _ d)]. Defined.

Lemma diag_back_isrefl \{x\} : (d : diag x x), eq_refl = diag_back d.
  Proof. induction x as [ | x IHx]; simpl; intro d; destruct (diag_inv d);
  [reflexivity | case (IHx d). cbn. reflexivity]. Qed.

Lemma diag_mono \{x y\} (e : x = y) : e = diag_back (eq_diag e).
  Proof. destruct e; destruct x as [ | x]; simpl.
  + destruct (diag_inv dia0); reflexivity.
  + destruct (diag_inv (diaS x x diag_refl)) as [d]. case (diag_back_isrefl d); reflexivity.
  Qed.

Corollary UIP_nat (x: nat) (e : x = x) : eq_refl = e.
  Proof. rewrite (diag_mono e). apply diag_back_isrefl. Qed.
Horribly simpler proof of UIP in nat along the same scheme...

Fixpoint diagTF (x y : nat) : Prop :=
  match x, y with
  | 0, 0   => True
  | S x, S y => diagTF x y
  | _, _    => False
  end.

Definition diagTF_refl x : diagTF x x :=...

Definition eq_diagTF {x y} (e : x = y) : diagTF x y :=...

Definition diagTF_back {x} : y, diagTF x y → x = y :=...

Lemma diagTF_back_isrefl {x} : (d : diagTF x x), eq_refl = diagTF_back d.

Lemma diagTF_mono {x y} (e : x = y) : e = diagTF_back (eq_diagTF e).

Corollary UIP_nat (x: nat) (e : x = x) : eq_refl = e.

Proof. rewrite (diagTF_mono e). apply diagTF_back_isrefl. Qed.

... without diag and its inversion :(
Equality is too easy, what about ?

Inversion performed “as if” was defined as

\[
\text{Inductive le n : nat → Prop :=}
\]
\[
| \text{le_e_0} : n = 0 \rightarrow n \ 0 \\
| \text{le_e_S m} : n = S m \rightarrow n \ S m \\
| \text{le_S m} : n \ m \rightarrow n \ S m.
\]

Definition eq_le n m (e : n = m) : n \ m :=

match e with eq_refl => le_n n end.

Inductive le_0 [n] : n \ 0 → Prop :=

| le_0_e : e, le_0 (eq_le e).

Inductive le_Sm [m n] : n \ S m → Prop :=

| le_Sm_e : e, le_Sm (eq_le e)
| le_Sm_S : l, le_Sm (le_S n m l).

Lemma le_inv {n m} (l : n \ m) :

match m with
| 0 => le_0
| S m => @le_Sm m
end n l.
Equality is too easy, what about ?

Inversion performed “as if” was defined as

Inductive le n : nat → Prop :=
| le_e_0 : n = 0 → n 0
| le_e_S m : n = S m → n S m
| le_S m : n m → n S m.

Definition eq_le n m (e : n = m) : n m :=
match e with eq_refl => le_n n end.

Inductive le_0 [n] : n 0 → Prop :=
| le_0_e : e, le_0 (eq_le e).

Inductive le_Sm [m n] : n S m → Prop :=
| le_Sm_e : e, le_Sm (eq_le e)
| le_Sm_S : l, le_Sm (le_S n m l).

Lemma le_inv {n m} (l : n m) :
match m with
| O => le_0
| S m => @le_Sm m
end n l.
Lemma eq_is_le_n {n} (e : n = n) : le_n n = eq_le e.
Proof. rewrite (UIP_refl_nat n e). reflexivity. Qed.

Lemma lenn_unique {n} (l : n ≤ n) : le_n n = l.
Proof. destruct n; destruct (le_inv l); try apply eq_is_le_n. case (lt_irrefl _ l).
Qed.

Inductive is_le_S {n m} : n ≤ S m → Prop :=
| is_le_S_intro : l, is_le_S (le_S n m l).

Lemma leS_is_le_S n m (lS : n ≤ S m) : n ≤ m → is_le_S lS.
Proof. destruct (le_inv lS) as [ e | ll ]; intro l; try constructor.
exfalso; rewrite e in l; apply (lt_irrefl _ l).
Qed.

Fixpoint le_unique {n m} (p : n ≤ m) : q, p = q.
Proof. destruct p as [ l m p]; intro q; cbn.
- destruct (lenn_unique q); reflexivity.
- destruct (leS_is_le_S q p). apply f_equal, le_unique.
Qed.
The Braga method


Dominique Larchey-Wendling and Jean-François Monin.


In Klaus Mainzer, Peter Schuster, and Helmut Schwichtenberg, editors.

*Proof and Computation II: From Proof Theory and Univalent Mathematics to Program Extraction and Verification.*


Small inversions

http://home/jf/www/Proof/Small_inversions/2022/